Phased Computation Graphs in the Polyhedral Model *

William Thies, Jasper Lin and Saman Amarasinghe
MIT Laboratory for Computer Science
Cambridge, MA 02139

{thies, jasperln, saman}@lcs.mit.edu

Abstract

We present a translation scheme that allows a broad class of dataflow graphs to be considered under the optimization framework of the polyhedral model. The input to our analysis is a Phased Computation Graph, which we define as a generalization of the most widely used dataflow representations, including synchronous dataflow, cyclo-static dataflow, and computation graphs. The output of our analysis is a System of Affine Recurrence Equations (SARE) that exactly captures the data dependences between the nodes of the original graph. Using the SARE representation, one can apply many techniques from the scientific community that are new to the DSP domain. For example, we propose simple optimizations such as node splitting, decimation propagation, and steady-state invariant code motion that leverage the fine-grained dependence information of the SARE to perform novel transformations on a stream graph. We also propose ways in which the polyhedral model can offer new approaches to classic problems of the DSP community, such as minimizing buffer size, code size, and optimizing the schedule.

1 Introduction

Synchronous dataflow graphs have become a powerful and widespread programming abstraction for DSP environments. Originally proposed by Lee and Messerchmitt in 1987 [18], synchronous dataflow has since evolved into a number of varieties [5, 1, 26, 6] which provide a natural framework for developing modular processing blocks and composing them into a concurrent network. A dataflow graph consists of a set of nodes with channels running between them; data values are passed as streams over the channels. The *synchronous* assumption is that a node will not fire until all of its inputs are ready. Also, the input/output rates of the nodes are known at compile time, such that a static schedule can be computed for the steady-state execution. Synchronous dataflow graphs have been shown to successfully model a large class of DSP applications, including speech coding, auto-correlation, voice band modulation, and a number of filters. A significant industry has emerged to provide dataflow design environments; leading products include COSSAP from Synopsys, SPW from Cadence, ADS from Hewlett Packard, and DSP Station from Mentor Graphics. Academic projects include prototyping environments such as Ptolemy [17] and Grape-II [16] as well as languages such as Lustre [13], Signal [10], and StreamIt [11].

*This document is Version 1 of MIT Laboratory for Computer Science Technical Memo LCS-TM-630, August 2002. Our latest work on this subject will be available from `http://compiler.lcs.mit.edu/streamit`

Optimization is especially important for signal processing applications, as embedded processors often have tight resource constraints on speed, memory, and power. In the synchronous dataflow domain, this has resulted in a number of aggressive optimizations that are specially designed to operate on dataflow graphs–e.g., minimizing buffer size while restricting code size [25], minimizing buffer size while maintaining parallelism [12], resynchronization [4], and buffer sharing [24]. While these optimizations are beneficial, they unilaterally regard each node as a black box that is invisible to the compiler. Blocks are programmed at a coarse level of granularity, often with hand-tweaked implementations inside, and optimizations are considered only when stitching the blocks together.

In some sense at the other end of the spectrum are the optimization techniques of the scientific computing community. These are characterized by fine-grained analyses of loops, statements, and control flow, starting from sequential C or FORTRAN source code that lacks the explicit parallelism and communication of a dataflow graph. Out of this community has come a number of general methods for precise, fine-grained program analysis and optimization. In the context of array dataflow analysis and automatic parallelization, one such framework is the polyhedral model [7], which represents programs as sets of affine relations over polyhedral domains. In recent years, it has been shown that affine dependences can exactly represent the flow of data in certain classes of programs [8], and that affine partitioning can subsume loop fusion, reversal, permutation, and reindexing as a scheduling technique [21]. Moreover, affine scheduling can operate on a parameterized version of the input program, avoiding the need to expand a graph for varying parameters and problem sizes, and it can often reduce to a linear program for flexible and efficient optimization. Polyhedral representations have also been utilized for powerful storage optimizations [22, 28, 31, 20].

In this paper, we aim to bridge the gap and employ the polyhedral representations of the scientific community to analyze the synchronous dataflow graphs of the DSP community. Towards this end, we present a translation from a dataflow graph to a System of Affine Recurrence Equations (SARE), which are equivalent to a certain class of imperative programs and can be used as input to the affine analysis described above. The input of our analysis is a "Phased Computation Graph" (PCG), which we formalize as a generalization of several existing dataflow frameworks. Once a PCG is represented as a SARE, we demonstrate how affine techniques could be employed to perform novel optimizations such as node splitting, decimation propagation, and steady-state invariant code motion that exploit both the structure of the dataflow graph as well as the internal contents of each node. We also discuss the potential of affine techniques for scheduling parameterized dataflow graphs, as well as offering new approaches to classic problems in the dataflow domain.

The rest of this paper is organized as follows. We begin by describing background information and related work on dataflow graphs and SARE's (Section 2). Then we define a Phased Computation Graph (Section 3) and Phased Computation Program (Section 4), and give the procedure for translating a simple, restricted PCP to a SARE (Section 5). Next, we give the full details for the general translation (Section 6), describe a range of applications for the technique (Section 7), and conclude (Section 8). A detailed example that illustrates our technique appears in the Appendix.

2 Background and Related Work

2.1 Dataflow Graphs

Strictly speaking, synchronous dataflow (SDF) refers to a class of dataflow graphs where the input rate and output rate of each node is constant from one invocation to the next [18, 19]. For SDF graphs, it is straightforward to determine a legal steady state schedule using balance equations [3] (see Section 5.1). The SDF scheduling community focuses on "single-appearance schedules" (SAS) in which each filter appears at only one position in the loop nest denoting the schedule. The interest in SAS is because the runtime overhead of a method call is prohibitive, so the code for each block is inlined. However, code size is so important that they can't afford to duplicate the inlining, thus leading to the restriction that each filter appear only once. They have demonstrated that minimizing buffer size in general graphs is NP-complete, but they explore two heuristics to do it in acyclic graphs [2].

Cyclo-static dataflow (CSDF) is the generalization of SDF to the case where each node has a set of phases with different I/O rates that are cycled through repeatedly [5, 27]. Compared to SDF, CSDF is more natural for the programmer for writing phased filters, since periodic tasks can be separated. For instance, consider a channel decoder that inputs a large matrix of coefficients at regular intervals, but decodes one element at a time for N invocations in between each set of coefficients. By having two separate phases, the programmer doesn't have to interleave this functionality in a single, convoluted work function. Also, CSDF can have lower latency and more fine-grained parallelism than SDF. However, current scheduling techniques for CSDF involve converting each phase to an SDF actor, which can cause deadlock even for legal graphs [5] and does not take advantage of the fine-grained dataflow exposed by CSDF.

The computation graph (CG) was described by Karp and Miller [15] as a model of computation that, unlike the two above, allows the nodes to *peek*. By "peek" we mean that a node can read and use a value from a channel without consuming (popping) it from the channel; peeking is good for the programmer because it automates the buffer management for input items that are used on multiple firings, as in an FIR filter. However, peeking complicates the schedule because an initialization schedule is needed to ensure that enough items are on the channel in order to fire [11]. In [15], conditions are given for when a computation graph terminates and when its data queues are bounded.

Govindarajan et. al develop a linear programming framework for determining the "rate-optimal schedule" with the minimal memory requirement [12]. A rate-optimal schedule is one that takes advantage of parallel resources to execute the graph with the maximal throughput. Though their formulation is attractive and bears some likeness to ours, it is specific for rate-optimal schedules and does not directly relate to the polyhedral model. It also can result in a code size blow up, as the same node could be executed in many different contexts.

2.2 Systems of Affine Recurrence Equations

A System of Affine Recurrence Equations (SARE) is a set of equations $\mathcal{E}_1 \dots \mathcal{E}_{m_\mathcal{E}}$ of the following form [7, 8]:

$$\forall \vec{i} \in \mathcal{D}_\mathcal{E} : \quad X(\vec{i}) = f_\mathcal{E}(\dots, Y(\vec{h}_{\mathcal{E},Y}(\vec{i})), \dots) \tag{1}$$

In this equation, our notation is as follows:

- $\{X, Y, \dots\}$ is the set of *variables* defined by the SARE. Variables can be considered as multi-dimensional arrays mapping a vector of indices $\vec{i}$ to a value in some space $\mathcal{V}$.
- $\mathcal{D}_\mathcal{E}$ is a polyhedron representing the *domain* of the equation $\mathcal{E}$. Each equation for a variable X has a disjoint domain; the domain D_X for variable X is taken as the convex union of all domains over which X is defined.
- $\vec{h}_{\mathcal{E},Y}$ is a vector-valued affine function, giving the index of variable Y that $X(\vec{i})$ depends on. A vector-valued function $\vec{h}$ is affine if it can be written as $\vec{h}(\vec{i}) = C\vec{i} + \vec{d}$, where C is a constant array and $\vec{d}$ is a constant vector that do not vary with $\vec{i}$.
- $f_\mathcal{E}$ is a strict function used to compute the elements of X.

Model of Computation	Items on Channels at Start	Nodes with Pop $\neq$ Push	Nodes with Peek > Pop	Cyclic Steady State Phases	Initialization Phases
Synchronous Dataflow [18]	X	X			
Cyclo-Static Dataflow [5]	X	X		X	
Computation Graphs [14]	X	X	X		
Phased Computation Graphs	X	X	X	X	X

Table 1: Models of Computation for Dataflow Graphs.

Intuitively, the SARE can be considered as follows. Given an array element at a known index $\vec{i}$, the SARE gives the set of arrays and indices that this element depends on. Furthermore, each index expression on the right hand side is an affine expression of $\vec{i}$. Note that in order to map directly from an array element to the elements that it depends on, the index expression on the left hand side must exactly match the domain quantifier $\vec{i}$.

A SARE has no notion of internal state per se; however, state can be modeled by introducing a self-loop in the graph of variables. If the dependence function $\vec{h}$ happens to be a translation ($\vec{h}(\vec{i}) = \vec{i} - \vec{k}$ for constant $\vec{k}$), then the SARE is *uniform*, and referred to as a SURE or SRE [15]. SURE's can be used for systolic array synthesis [29], and there are techniques to uniformize SARE's into SURE's [23].

SARE's became interesting from the standpoint of program analysis upon Feautrier's discovery [8, 9] that a SARE is mathematically equivalent to a "static control flow program". A program has static control flow if the entire path of control can be determined at compile time (see [8, 7] for details). Feautrier showed that using a SARE, one can use linear programming to produce a *parameterized* schedule for a program. That is, if a loop bound is unknown at compile time, then the symbolic bound enters the schedule and it does not affect the complexity of the technique. Feautrier's framework is very flexible in that it allows one to select a desirable affine schedule by any linear heuristic.

Lim and Lam [22] build on Feautrier's technique with an affine partitioning algorithm that maximizes the degree of parallelism while minimizing the degree of synchronization. It also operates on a form that is equivalent to a SARE.

3 Phased Computation Graphs

We introduce the Phased Computation Graph (PCG) as a model of computation for dataflow languages; specifically, we designed the PCG as a model for StreamIt [30]. A PCG is a generalization of the Computation Graph (CG) of Karp and Miller [14] to the case where each node has both an initial and steady-state execution epoch, each of which contains a number of *phases*. It is also a generalization of other popular representations for dataflow graphs, such as synchronous dataflow (SDF) and cyclo-static dataflow (CSDF) (see Table 1). Phases are attractive constructs for the reasons given in describing CSDF in Section 2; also, an initial epoch is critical for applications such as a WAV file decoder that adjusts its parameters based on the data it finds in the header of the stream.

Formally, a PCG is a directed graph with the following components:

- Nodes $n_1 \dots n_{m_n}$.
- Channels $c_1 \dots c_{m_c}$, where each channel is directed from a given node n_a to a given node n_b.
- A non-negative integer $A(c)$ indicating the number of items that are initially present on channel c.

- The number of phases $num(n,t)$ that node n exhibits during epoch t. Here t indicates the type of epoch, which is either initial (*init*) or steady-state (*steady*). For all n, $num(n, init) \geq 0$ and $num(n, steady) \geq 1$.
- Non-negative integers $U(c,t,p)$, $O(c,t,p)$, and $E(c,t,p)$, which give the number of items pushed, popped, and peeked, respectively, over channel c during the p'th phase of epoch t.

3.1 Execution Model

We give an informal description of the execution semantics of a PCG, which is an intuitive extension to that of a CG. An execution consists of a sequence of atomic *firings* of nodes in the graph. The effect of firing a given node depends on its epoch and its phase, which are both local states of the node. At the start of execution, each node is in phase 0 of the initial epoch, and each channel c contains $A(c)$ items.

Let us consider a node n that is in phase p of epoch t. It is legal for n to fire if $p \in [0, num(n,t) - 1]$ and, for each channel c_{in} directed into n, there are at least $E(c_{in}, t, p)$ items on c_{in}. The effect of this firing is to consume $O(c_{in}, t, p)$ items from each channel c_{in} directed into n; to produce $U(c_{out}, t, p)$ new items on each channel c_{out} directed out of n; and to advance the phase p of n to $next(t,p)$. In the initial epoch, each phase executes only once and $next(init, p) = p + 1$; in the steady-state epoch, the phases execute cyclically and $next(steady, p) = (p+1)\ mod\ num(n,t)$. If a node n is in phase $num(n, init)$ of the initial epoch, then it transitions to the steady-state epoch and resets its phase to 0.

From the starting state of the graph, execution proceeds via an infinite sequence of node firings. The order of node firings is constrained only by the conditions given above.

4 Phased Computation Programs

The PCG described above is a representation for a graph, treating each node and channel as a black box. Here we introduce some notation for the internals of the nodes, as well as the ordering of items on the channels. We refer to the aggregate model as a Phased Computation Program (PCP). Before describing a PCP, we will need some additional notation:

- We assume that all items passed over channels are drawn from the same domain of values $\mathcal{V}$.
- An array type[1] of length n and element type τ is denoted by $\tau[n]$. Given a two-dimensional array A of type $\tau[n][m]$, $A[i][*]$ denotes the m-element array comprising the i'th row of A.
- $num\_in(n)$ (resp. $num\_out(n)$) denotes the number of channels that are directed into (resp. out of) node n.
- $chan\_in(n)$ (resp. $chan\_out(n)$) denotes the list of channels that are directed into (resp. out of) node n.
- $pos\_in(n,c)$ (resp. $pos\_out(n,c)$) denotes the position of channel c in $chan\_in(n)$ (resp. $chan\_out(n)$. That is, $chan\_out(n)[pos\_out(n, c)] = c$.

[1]To simplify the presentation, we allow ourselves a slight abuse of notation by sometimes using arrays instead of enumerating individual elements. Our definitions could be expanded into strict element-wise SARE's without any fundamental change to the technique.

A PCP consists of the following:

- A Phased Computation Graph $G = (\{n_1 \dots n_{m_n}\}, \{c_1 \dots c_{m_c}\}, A, \mathit{num}, U, O, E)$ describing the nodes, channels, and I/O rates of the program (see Section 3). Each channel $c = (n_a, n_b)$ is a FIFO queue; n_a pushes items onto the back of c and n_b consumes items from the front of c.
- The initial values $I(c)$ that are enqueued onto channel c at the start of execution. Since there are $A(c)$ initial values on channel c, $I(c)$ has type $\mathcal{V}[A(c)]$.
- A work function $W(n,t,p)$ that represents the computation of node n in phase p and epoch t. The signature of $W(n,t,p)$ is $[\mathcal{V}[E(\mathit{chan\_in}(n)[1],t,p)], \dots, \mathcal{V}[E(\mathit{chan\_in}(n)[\mathit{num\_in}(n)],t,p)]] \rightarrow$ $[\mathcal{V}[U(\mathit{chan\_out}(n)[1],t,p)], \dots, \mathcal{V}[U(\mathit{chan\_out}(n)[\mathit{num\_out}(n)],t,p)]]$. That is, the function inputs a list of arrays, each of which corresponds to an incoming channel c_{in} and contains the $E(c_{in},t,p)$ values that n reads from c_{in} in a given firing. The procedure returns a list of arrays, each of which corresponds to an outgoing channel c_{out} and contains the $U(c_{out},t,p)$ values that n writes to c_{out} in a given firing.

5 Basic Translation: CG to SARE

In order to familiarize the reader with the basics of our technique, we present in this section the translation procedure for a simplified input domain. The translation rules for the general case can be found in Section 6.

Our basic translation operates on computation graphs with all of the channels initially empty. That is, we assume that:

1. No items appear on the channels at the beginning: $\forall c,\ A(c) = 0$.
2. There are no initialization phases: $\forall n,\ \mathit{num}(n, \mathit{init}) = 0$.
3. Each node has only one steady-state phase: $\forall n,\ \mathit{num}(n, \mathit{steady}) = 1$.

Given these restrictions, we can simplify our notation considerably. Since we are only concerned with the steady-state epoch and the 0'th phase of each node, we can omit some arguments to any function that requires an epoch t or a phase p. For instance, the push and pop rates of a channel c are now just $U(c)$ and $O(c)$, respectively; the work function for a node n is simply $W(n)$.

5.1 Calculating the Steady-State Period

Let $S(n)$ denote the number of times that node n fires its first phase for a periodic steady-state execution of the entire graph. A *periodic* schedule is one that does not change the number of items on the channels in the graph after executing; in other words, it is legal to execute in the steady state. For an SDF graph, there is a unique and minimal periodic schedule, of which all other periodic schedules are a multiple [3]. It is straightforward to use a set of balance equations to solve for $S(n)$ given the declared rates of filters in the graph. If the graph is invalid (i.e., it would lead to deadlock or an unbounded buffer size) then the balance equations will have no solution. See [3] for details.

We will use the following helper function in our analysis. Given channel $c = (n_a, n_b)$:

$$\mathit{Period}(c) \equiv S(n_a) * U(c) = S(n_b) * O(c)$$

That is, $\mathit{Period}(c)$ denotes the number of items that are passed over channel c during a single steady-state execution of the graph.

Variables

For each channel $c = (n_a, n_b)$, do the following:

- Introduce variable BUF_c with the following domain:

$$\mathcal{D}_{BUF_c} = \{\ (i,j) \mid 0 \le i \le N-1 \ \wedge\ 0 \le j \le Period(c) - 1\} \tag{2}$$

- Introduce variable $WRITE_c$ with this domain:

$$\mathcal{D}_{WRITE_c} = \{\ (i,j,k) \mid 0 \le i \le N-1 \ \wedge\ 0 \le j \le S(n_a) - 1 \ \wedge\ 0 \le k \le U(c) - 1\} \tag{3}$$

- Introduce variable $READ_c$ with this domain:

$$\mathcal{D}_{READ_c} = \{\ (i,j,k) \mid 0 \le i \le N-1 \ \wedge\ 0 \le j \le S(n_b) - 1 \ \wedge\ 0 \le k \le E(c) - 1\} \tag{4}$$

Equations

For each node n, introduce the following equations:

- **(READ to WRITE)** For each $c \in chan\_out(n)$:

$$\begin{gathered}\forall (i,j,k) \in \mathcal{D}_{WRITE_c}: \quad WRITE_c(i,j,k) = W(n)(Steady\_Inputs)[pos\_out(n,c)][k] \\ where\ Steady\_Inputs = [READ_{chan\_in(n)[0]}(i,j,*), \ldots, READ_{chan\_in(n)[num\_in(n)-1]}(i,j,*)]\end{gathered} \tag{5}$$

- **(WRITE to BUF)** For each $c \in chan\_out(n)$, and for each $q \in [0, S(n)-1]$:

$$\begin{gathered}\forall (i,j) \in \mathcal{D}_{W\to B}(c,q): \quad BUF_c(i,j) = WRITE_c(i,q,j - q * U(c)) \\ where\ \mathcal{D}_{W\to B}(c,q) = \mathcal{D}_{BUF_c} \cap \{(i,j) \mid q * U(c) \le j \le (q+1) * U(c) - 1\}\end{gathered} \tag{6}$$

- **(BUF to READ)** For each $c \in chan\_in(n)$, and for each $q \in [0, \lfloor \frac{E(c)}{Period(c)} \rfloor]$:

$$\begin{gathered}\forall (i,j,k) \in \mathcal{D}_{B\to R}: \quad READ_c(i,j,k) = BUF_c(i+q, j * O(n) + k - q * Period(c)) \\ where\ \mathcal{D}_{B\to R} = \mathcal{D}_{READ_c} \cap \{(i,j,k) \mid q * Period(c) \le k \le min((q+1) * Period(c), E(c)) - 1\}\end{gathered} \tag{7}$$

Figure 1: Procedure for generating a SARE from a simplifed PCP: one that has only one epoch, one phase, and no initial tokens.

5.2 Generating a SARE

We now generate a SARE for the simplified instance of a PCP. The *domains* of the SARE will be parameterized by N, the number of steady-state cycles that one wishes to execute in the PCP. However, it is important to note that N does not affect the number of variables or equations in the SARE, as that would require a separate compilation and analysis for each value of N. It is a key benefit of the SARE representation that N can be incorporated as a symbolic parameter in the schedule.

The procedure for generating the SARE appears in Figure 1. The basic idea is to introduce a variable BUF_c for each channel c which keeps track of the entire history of values that are transmitted over c (we keep track of the entire history since all elements in a SARE must be assigned only once.) The i dimension of *BUF* counts over steady-state periods, while the j dimension holds elements that were written during a given period. We also introduce variables $WRITE_c$ and $READ_c$ that write and read from channel c. For reasons that become clear below, these variables have a k dimension to index the values that are pushed on a given firing, while the j dimension counts firings of a node and the i dimension counts steady-state periods, as before.

Equation 5 expresses the computation internal to each node, whereas Equations 6 and 7 expresses the communication of the nodes with the channels. In Equation 5, the node's work function is used to calculate all of the *WRITE* values from all of the *READ* values, during a given firing (i, j).

Equation 6 shows the communication from the *WRITE* array to the *BUF* array for a given channel. To understand this equation, it might help to consider a simpler version which expresses the correct relationship but is not a legal SARE:

$$\forall (i,j,k) \in \mathcal{D}_{WRITE_c}: \quad BUF_c(i, j * U(c) + k) = WRITE_c(i,j,k)$$

The equation above is simply copying the contents of *WRITE* into *BUF* while accounting for the differing array dimensions. However, this equation is not valid for a SARE, since there is an affine expression indexing the array on the left hand side.

To deal with this issue, we split up the equation into several pieces, each of which assigns to a different portion of the *BUF* array. In Equation 6, we introduce a variable q that counts up to $S(n)$, which is the extent of the j dimension of $WRITE_c$. For each of these q, we copy over $U(c)$ values from the appropriate section of the *WRITE* array; the domain $\mathcal{D}_{W \to B}(c, q)$ represents the q'th slice of *BUF* into which the section should be written. This equation is a valid component of our SARE, as the boundaries on each equation's domain are fully resolvable at compile time.

Equation 7 is similar to Equation 6 regarding the slicing technique that is used to parameterize the domain of the equation. However, here the motivation is different: the extent of the k dimension of *READ* might exceed the length of the period in *BUF* (this is because nodes are allowed to peek more than they pop.) Thus, the q variable is used to slice the peek amount $E(c)$ into $Period(c)$-sized chunks so as to resolve the proper source location in *BUF*. Again, all the variables in the index expressions are compile-time constants (except, of course, for the index quantifiers i, j, and k.)

6 General Translation: PCP to SARE

In this section we give the formal translation of a Phased Computation Program to a System of Affine Recurrence Equations. Again, the SARE will be parameterized by N, the number of steady-state cycles that one wishes to execute in the PCP. The translation will use the helper functions shown in Figure 2. We

Helper Functions

- The number of integral points in a one-dimensional domain $\mathcal{D}$ is denoted by $|\mathcal{D}|$. The sign and absolute value of an integer i are given by $sign(i)$ and $abs(i)$, respectively.
- $\mathit{PartialWrite}(c,t,p) = \sum_{r=0}^{p-1} U(c,t,r)$
- $\mathit{TotalWrite}(n,c,t) = \sum_{r=0}^{num(n,t)-1} U(c,t,r)$
- $\mathit{PartialRead}(c,t,p) = \sum_{r=0}^{p-1} O(c,t,r)$
- $\mathit{TotalRead}(n,c,t) = \sum_{r=0}^{num(n,t)-1} O(c,t,r)$
- Given channel $c = (n_a, n_b)$:

$$\mathit{Period}(c) \equiv S(n_a) * \mathit{TotalWrite}(n_a, c, \mathit{steady}) = S(n_b) * \mathit{TotalRead}(n_b, c, \mathit{steady})$$

Figure 2: Helper functions for the PCP to SARE translation.

also make use of the $S(n)$ function giving the number of steady-state executions of node n (see Section 5.1). Please refer to the Appendix for a concrete illustration of the techniques described below.

Figure 3 illustrates the variables used for the translation. These are very similar to those in the simple case, except that now there are separate variables for initialization and steady state, as well as variables for each phase. The equations for the translation appear in Figures 5 and 6. A diagram of which variables depend on others appears in Figure 4. Due to space limitations, we discuss only the most interesting parts of the translation below.

Equation 14 writes the initial values of the initial tokens on *IBUF*, the initial buffer. Equation 15 transfers values from the initial epoch of a node to the initial buffer of a channel. Equations 16 and 17 do the work computation of the initial and steady-state items, respectively, just as in the simple translation of Section 5.2. Equation 18 copies items from the node to the channel, using the same slicing technique as Equation 6.

Equation 19 copies values from the initial buffer (*IBUF*) of a channel into the initial read array for a node. However, since the node might read more items in the initial epoch than exist on the initial buffer, we restrict the domain of the equation to only those indices that have a corresponding value in *IBUF*. If it is the case that there are fewer items in the initial buffer than a node reads in its initial epoch, then Equation 20 copies values from the beginning of the channel's steady-state buffer into the remaining locations on the node's initial array. The upper bound for q in Equation 20 is calculating the maximum amount that any phase peeks in the initialization epoch, so that enough equations are defined to fill the node's initial read array. Note that for some phases, the dependence domain of Equation 20 will be empty, in which case no equation is introduced. Also, Equation 20 uses the slicing method (see Section 5.2) because the initial read array is of lower dimensionality than the steady-state buffer from which it is reading.

Equation 21 is dealing with the opposite case as Equation 20: when a node finds items on its channel's initial buffer when it is entering the steady-state epoch. In this case copying over the appropriate items to the beginning of the steady-state read array is rather straightforward.

Much of the complexity of the entire translation is pushed into Equations 22 and 23. There are two distinct complications in these equations. First is one that we tackled in Equation 7: the fact that the read array might peek beyond the limits of the buffer, which requires us to slice the domain in increments of one period. The second complication is that there could be an offset–an initial buffer might have written into

Variables

For each channel $c = (n_a, n_b)$, do the following:

- Introduce variables $IBUF_c$ and $SBUF_c$ with the following domains:

$$\mathcal{D}_{IBUF_c} = \{\ i \mid 0 \le i \le A(c) - 1 + \sum_{p=0}^{num(n_a,init)-1} U(c, init, p)\} \tag{8}$$

$$\mathcal{D}_{SBUF_c} = \{\ (i,j) \mid 0 \le i \le N-1\ \wedge\ 0 \le j \le Period(c) - 1\} \tag{9}$$

- For each $p \in [0, num(n_a, init) - 1]$, introduce variable $IWRITE_{c,p}$ with this domain:

$$\mathcal{D}_{IWRITE_{c,p}} = \{\ i \mid 0 \le i \le U(c, init, p) - 1\} \tag{10}$$

- For each $p \in [0, num(n_b, init) - 1]$, introduce variable $IREAD_{c,p}$ with this domain:

$$\mathcal{D}_{IREAD_{c,p}} = \{\ i \mid 0 \le i \le E(c, init, p) - 1\} \tag{11}$$

- For each $p \in [0, num(n_a, steady) - 1]$, introduce variable $SWRITE_{c,p}$ with this domain:

$$\mathcal{D}_{SWRITE_{c,p}} = \{\ (i,j,k) \mid 0 \le i \le N-1\ \wedge\ 0 \le j \le S(n_a) - 1\ \wedge\ 0 \le k \le U(c, steady, p) - 1\} \tag{12}$$

- For each $p \in [0, num(n_b, steady) - 1]$, introduce variable $SREAD_{c,p}$ with this domain:

$$\mathcal{D}_{SREAD_{c,p}} = \{\ (i,j,k) \mid 0 \le i \le N-1\ \wedge\ 0 \le j \le S(n_b) - 1\ \wedge\ 0 \le k \le E(c, steady, p) - 1\} \tag{13}$$

Figure 3: Variables for the PCP to SARE translation.

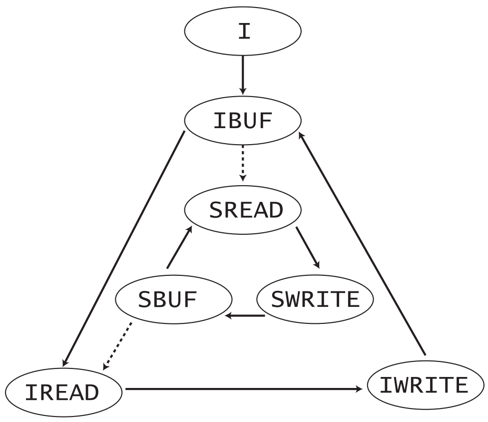

Figure 4: Flow of data between SARE variables. Data flows from the base of each arrow to the tip. Dotted lines indicate communication that might not occur for a given channel.

Equations (Part 1 of 2)

For each channel $c = (n_a, n_b)$, introduce the following equations:

- **(I to IBUF)**:
$$\forall i \in [0, A(c)-1] : \quad \mathit{IBUF}_c(i) = I(c)[i] \tag{14}$$

- **(IWRITE to IBUF)** For each $p \in [0, num(n_a, \mathit{init}) - 1]$:
$$\begin{aligned} &\forall i \in [A(c) + \mathit{PartialWrite}(c, \mathit{init}, p), A(c) + \mathit{PartialWrite}(c, \mathit{init}, p+1) - 1] : \\ &\quad \mathit{IBUF}_c(i) = \mathit{IWRITE}_{c,p}(i - A(c) - \mathit{PartialWrite}(c, \mathit{init}, p)) \end{aligned} \tag{15}$$

For each node n, introduce the following equations:

- **(IREAD to IWRITE)** For each $c \in \mathit{chan\_out}(n)$, and for each $p \in [0, num(n, \mathit{init}) - 1]$:
$$\begin{aligned} &\forall i \in \mathcal{D}_{\mathit{IWRITE}_{c,p}} : \quad \mathit{IWRITE}_{c,p}(i) = W(n, \mathit{init}, p)(\mathit{Init\_Inputs})[\mathit{pos\_out}(n,c)][i], \\ &\mathit{where}\ \mathit{Init\_Inputs} = [\mathit{IREAD}_{\mathit{chan\_in}(n)[0],p}(*), \ldots, \mathit{IREAD}_{\mathit{chan\_in}(n)[\mathit{num\_in}(n)-1],p}(*)] \end{aligned} \tag{16}$$

- **(SREAD to SWRITE)** For each $c \in \mathit{chan\_out}(n)$, and for each $p \in [0, num(n, \mathit{steady}) - 1]$:
$$\begin{aligned} &\forall (i,j,k) \in \mathcal{D}_{\mathit{SWRITE}_{c,p}} : \quad \mathit{SWRITE}_{c,p}(i,j,k) = \\ &\quad W(n, \mathit{steady}, p)(\mathit{Steady\_Inputs})[\mathit{pos\_out}(n,c)][k], \mathit{where} \\ &\mathit{Steady\_Inputs} = [\mathit{SREAD}_{\mathit{chan\_in}(n)[0],p}(i,j,*), \ldots, \mathit{SREAD}_{\mathit{chan\_in}(n)[\mathit{num\_in}(n)-1],p}(i,j,*)] \end{aligned} \tag{17}$$

- **(SWRITE to SBUF)** For each $c \in \mathit{chan\_out}(n)$, for each $p \in [0, num(n, \mathit{steady}) - 1]$, and for each $q \in [0, S(n) - 1]$:
$$\begin{aligned} &\forall (i,j) \in \mathcal{D}_{SW \to SB}(c,p,q) : \quad \mathit{SBUF}_c(i,j) = \\ &\quad \mathit{SWRITE}_{c,p}(i, q, j - \mathit{Offset}_{SW \to SB}(q,n,c,p)), \mathit{where} \\ &\mathcal{D}_{SW \to SB}(c,p,q) = \mathcal{D}_{\mathit{SBUF}_c} \cap \{(i,j) \mid \mathit{Offset}_{SW \to SB}(q,n,c,p) \le j \le \mathit{Offset}_{SW \to SB}(q,n,c,p+1) - 1\} \\ &\mathit{and}\ \mathit{Offset}_{SW \to SB}(q,n,c,p') = q * \mathit{TotalWrite}(n,c,\mathit{steady}) + \mathit{PartialWrite}(c, \mathit{steady}, p')) \end{aligned} \tag{18}$$

- **(IBUF to IREAD)** For each $c \in \mathit{chan\_in}(n)$, and for each phase $p \in [0, num(n, \mathit{init}) - 1]$:
$$\begin{aligned} &\forall i \in \mathcal{D}_{IB \to IR}(c,p) : \quad \mathit{IREAD}_{c,p}(i) = \mathit{IBUF}_c(\mathit{PartialRead}(c, \mathit{init}, p) + i) \\ &\mathit{where}\ \mathcal{D}_{IB \to IR}(c,p) = \{i \mid i \in \mathcal{D}_{\mathit{IREAD}_{c,p}} \ \wedge\ i + \mathit{PartialRead}(c, \mathit{init}, p) \in \mathcal{D}_{\mathit{IBUF}_c}\} \end{aligned} \tag{19}$$

- **(SBUF to IREAD)** For each $c \in \mathit{chan\_in}(n)$, for each phase $p \in [0, num(n, \mathit{init}) - 1]$, and for each $q \in [0, \lfloor \frac{\mathit{argmax}_{p'}(\mathit{PartialRead}(c,\mathit{init},p') + E(c,\mathit{init},p'))}{\mathit{Period}(c)} \rfloor]$:
$$\begin{aligned} &\forall i \in \mathcal{D}_{SB \to IR}(c,p) : \quad \mathit{IREAD}_{c,p}(i) = \mathit{SBUF}_c(q, i - \mathit{Offset}_{SB \to IR}(q,n,c,p)), \mathit{where} \\ &\mathcal{D}_{SB \to IR}(c,p) = \{i \mid i \in \mathcal{D}_{\mathit{IREAD}_{c,p}} \ \wedge\ \mathit{Offset}_{SB \to IR}(q,n,c,p) \le i \le \mathit{Offset}_{SB \to IR}(q+1,n,c,p) - 1\} \\ &\mathit{and}\ \mathit{Offset}_{SB \to IR}(q',n,c,p) = q' * \mathit{Period}(c) + |\mathcal{D}_{\mathit{IBUF}_c}| - \mathit{PartialRead}(c, \mathit{init}, p) \end{aligned} \tag{20}$$

Figure 5: Equations for the PCP to SARE translation.

Equations (Part 2 of 2)

For each node n, introduce the following equations:

- **(IBUF to SREAD)** For each $c \in chan\_in(n)$, and for each phase $p \in [0, num(n, steady) - 1]$:

$$\begin{aligned}
&\forall (i,j,k) \in \mathcal{D}_{IB\to SR}(n,c,p): \quad SREAD_{c,p}(i,j,k) = IBUF_c(ReadIndex(n,c,p,i,j,k)), \textit{ where} \\
&ReadIndex(n,c,p,i,j,k) = TotalRead(n,c,init) + (i * S(n) + j) * TotalRead(n,c,steady) \\
&\qquad + PartialRead(c, steady, p) + k \\
&\textit{and } \mathcal{D}_{IB\to SR}(n,c,p) = \{(i,j,k) \mid ReadIndex(n,c,p,i,j,k) \in \mathcal{D}_{IBUF_c}\}
\end{aligned} \tag{21}$$

- **(SBUF to SREAD)** For each $c \in chan\_in(n)$, for each phase $p \in [0, num(n, steady) - 1]$, and for each $q \in [0, \lfloor \frac{E(c,\ steady,\ p)}{Period(c)} \rfloor]$:

$$\begin{aligned}
&\forall (i,j,k) \in \mathcal{D}1_{SB\to SR}(c,p,q): \quad SREAD_{c,p}(i,j,k) = SBUF_c(i + q + Int_Offset(n,c,p), \\
&\qquad j * TotalRead(n,c,steady) + PartialRead(c,steady,p) + k - q * Period(c) + Mod_Offset(n,c,p)), \\
&\textit{where } \mathcal{D}1_{SB\to SR}(c,p,q) = (\mathcal{D}_{SREAD_{c,p}} - PUSHED(n,c,p)) \ \cap \\
&\qquad \{(i,j,k) \mid q * Period(c) - min(0, Mod_Offset(n,c,p)) \le k \le \\
&\qquad\qquad min((q+1) * Period(c), E(c,steady,p)) - 1 - max(0, Mod_Offset(n,c,p))\}
\end{aligned} \tag{22}$$

$$\begin{aligned}
&\forall (i,j,k) \in \mathcal{D}2_{SB\to SR}(c,p,q): \quad SREAD_{c,p}(i,j,k) = \\
&\qquad SBUF_c(i + q + sign(Offset) + Int_Offset(n,c,p), j * TotalRead(n,c,steady) + \\
&\qquad\qquad PartialRead(c,steady,p) + k - q * Period(c) - sign(Offset) * Period(c) + Mod_Offset(n,c,p)), \\
&\textit{where } \mathcal{D}2_{SB\to SR}(c,p,q) = (\mathcal{D}_{SREAD_{c,p}} - PUSHED(n,c,p)) \ \cap \\
&\qquad (\{(i,j,k) \mid q * Period(c) \le k \le min((q+1) * Period(c), E(c,steady,p)) - 1\} - \mathcal{D}1_{SB\to SR}(c,p,q))
\end{aligned} \tag{23}$$

$$\begin{aligned}
&\textit{where } PUSHED(n,c,p) = \mathcal{D}_{IB\to SR}(n,c,p) \\
&\textit{and } Num_Pushed(n,c,p) = (max_{\preceq} PUSHED(n,c,p)) \cdot (S(n) * TotalRead(n,c,steady), \\
&\qquad TotalRead(n,c,steady), 1) + PartialRead(c,steady,p), \textit{ and } PEEKED(c,p) = \mathcal{D}_{SB\to IR(c,p)} \\
&\textit{and } Num_Popped(c,p) = |PEEKED(c,p)| + O(c,steady,p) - E(c,steady,p) \\
&\textit{and } Offset(n,c,p) = Num_Popped(c,p) - Num_Pushed(n,c,p) \\
&\textit{and } Int_Offset(n,c,p) = sign(Offset) * \lfloor \frac{abs(Offset(n,c,p))}{Period(c)} \rfloor) \\
&\textit{and } Mod_Offset(n,c,p) = sign(Offset) * (abs(Offset(n,c,p)) \ mod \ Period(c))
\end{aligned} \tag{24}$$

Figure 6: Equations for the PCP to SARE translation.

the beginning of the read array, or an initial read array might have read from the beginning of the steady-state buffer[2].

Equation 24 provides a set of helper functions to determine the offset for Equations 22 and 23. The result is a number *Offset* that the domain needs to shift. Unfortunately this involves splitting the original equation into two pieces, since after the shift the original domain could be split between two different periods (with different i indices in *SBUF*). Each of these equations is clever about the sign of the offset to adjust the indices in the right direction.

7 Applications

7.1 Node-Level Optimization

Perhaps the most immediate application of our technique is to use the SARE representation to bridge the gap between intra-node dependences and inter-node dependences. Our presentation in the last section was in terms of a coarse-grained work function W mapping inputs to outputs. However, if the source code is available for the node, then the equations containing W can be expanded into another level of affine recurrences that represent the implementation of the node itself. (If the node contained only static control flow, then this formulation as a SARE would be exact; otherwise, one could use a conservative approximation of the dependences.)

The resulting SARE would expose the fine-grained dependences between local variables of a given node with the local variables of neighboring nodes. We believe that this information opens the door for a wide range of novel node optimizations, which we outline in the following sections.

7.1.1 Inter-Node Dataflow Optimizations

Once the SARE has exposed the flow dependences between the internal variables of two different nodes, one can perform all of the classical dataflow optimizations as if the variables were placed in the same node to begin with. For instance, constant propagation, copy propagation, common sub-expression elimination, and dead code elimination can all be performed throughout the internals of the nodes as if they were connected in a single control flow graph.

Some of these optimizations could have a very large performance impact; for instance, consider a 2:1 downsampler node that simply discards half of its input. Using a SARE, it is straightforward to trace the element-wise dependence chain from each discarded item, and to mark each source operation as dead code (assuming the code has no other consumers.) In this way, one can perform fine-grained **decimation propagation** throughout the entire graph and prevent the computation of any items that are not needed. This degree of optimization is not possible in frameworks that treat the node as a black box.

7.1.2 Fission Transformations

It may be the case that many of the operations that are grouped within a node are completely independent, and that the node can be split into sub-components that can execute in parallel or as part of a more fine-grained schedule. For instance, many operations using complex data types are grouped together from the programmer's view, even though the operations on the real and imaginary parts could be separated. Given

[2]Fundamentally, this complication ended up in this equation because we chose to define the initial buffer (*IBUF*) to be the same length as its associated initial write array (*IWRITE*). If we had defined it in terms of the initial read array (*IREAD*) instead, the complexity would be in Equation 18.

a SARE representation of the graph and the node, this "node fission" is completely automatic, as there would be no dependence between the operations in the SARE. A scheduling pass would treat the real and imaginary operations as if they were written in two different nodes to begin with.

7.1.3 Steady-State Invariant Code Motion

In the streaming domain, the analog of loop-invariant code motion is the motion of code from the steady-state epoch to the initialization epoch if it does not depend on any quantity that is changing during the steady-state execution of a node. Quantities that the compiler detects to be constant during the execution of a work function can be computed from the initialization epoch, with the results being passed through a feedback loop for availability on future executions.

7.1.4 Induction Variable Detection

The work functions and feedback paths of a node could be analyzed (as would the body of a loop in the scientific domain) to see if there are induction variables from one steady-state execution to the next. This analysis is useful both for strength reduction, which *adds* a dependence between invocations by converting an expensive operation to a cheaper, incremental one, as well as for data parallelization, which *removes* a dependence from one invocation to the next by changing incremental operations on filter state to equivalent operations on privatized variables.

7.2 Graph Parameterization

In the scientific community, the SARE representation is specifically designed so that a parameterized schedule can be obtained for loop nests without having to unroll each loop and schedule each iteration independently. The same should hold true for dataflow graphs. Often there is redundant structure within a graph, such as an N-level filterbank. However, to the best of our knowledge, all scheduling algorithms for synchronous dataflow operate only on a fully expanded graph ("parameterized dataflow" is designed more for flexible re-initialization, and dynamically schedules an expanded version of each program graph [1].)

We believe that parameterized scheduling would be straightforward using the polyhedral model to represent the graph. For future work, we plan on implementing such a scheduler in the StreamIt compiler [11]. The StreamIt language [30] is especially well-tailored to this endeavor because it provides basic stream primitives that are hierarchical and parameterizable (e.g., an N-way SplitJoin structure that has N parallel child streams.)

7.3 Graph-Level Optimization

The SARE representation could also have something to offer with regards to the graph-level optimization problems that are already the focus of the DSP community. The polyhedral model is appealing because it provides a linear algebra framework that is simple, flexible, and efficient.

7.3.1 Buffer Minimization

Storage optimization is one area in which both the scientific community [22, 28, 31, 20] and the DSP community [25, 12, 24] have invested a great deal of energy. Both communities have invented schemes for detecting live ranges, collapsing arrays across dead locations, and sharing buffers/arrays between different statements. It will be an interesting avenue for future work to use the SARE representation of dataflow graphs to compare the sets of storage optimization algorithms and see what they have to offer to each other.

7.3.2 Scheduling

There is a large body of work in the scientific community regarding the scheduling of SARE's [7]. While they have not faced the same constraints on code size as the embedded domain, there are characteristics of affine schedules that could provide new directions for DSP scheduling algorithms. For instance, when solving for an affine schedule, one can obtain a schedule that places a statement within a loop nest but only executes it conditionally for certain values of the enclosing loop bounds. To the best of our knowledge, the single appearance schedules in the DSP community have never had this notion of conditional execution, perhaps because it represents a much larger space of schedules to consider. It is possible that an affine schedule, then, would give a more flexible solution when code size is the critical issue.

Also, there are a number of sophisticated parallelization algorithms (e.g., Lim and Lam [22]) that could extract high performance from dataflow graphs. It will be an interesting topic for future work to see how they compare with the scheduling algorithms currently in use for dataflow graphs.

8 Conclusion

This paper shows that a very broad class of synchronous dataflow graphs can be cleanly represented as a System of Affine Recurrence Equations, thereby making them amenable to all of the scheduling and optimization techniques that have been developed within the polyhedral model. The polyhedral model is a robust and well-established framework within the scientific community, with methods for graph parameterization, automatic parallelization, and storage optimization that are very relevant to the current challenges within the signal processing domain.

Our analysis inputs a Phased Computation Graph, which we originally formulated as a model of computation for StreamIt [30]. Phased Computation Graphs are a generalization of synchronous dataflow graphs, cyclo-static dataflow graphs, and computation graphs; they also include a novel notion of initial and steady-state epochs, which are an appealing abstraction for programmers.

We believe that the precise affine dependence framework provided by the SARE representation will enable a powerful suite of node optimizations in dataflow graphs. We proposed optimizations such as decimation propagation, node fission, and steady-state invariant code motion that are a first step in this direction.

References

[1] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling of dsp systems. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, June 2000.

[2] C. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Apgan and rpmc: Complementary heuristics for translating dsp block diagrams into efficient software impelementations. *Journal of Design Automation for Embededded Systems.*, pages 33–60, January 1997.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs.* Kluwer Academic Publishers, 1996. 189 pages.

[4] S. S. Bhattacharyya, S. Sriram, and E. A. Lee. Resynchronization for multiprocessor dsp systems. *IEEE Transactions on Circuit and Systems - I: Fundamental Theory and Applications*, 47(11), 2000.

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, pages 397–408, February 1996.

[6] J. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model.* PhD thesis, University of California, Berkeley, 1993.

[7] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization.* Birkhäuser Boston, 2000.

[8] P. Feautrier. Some efficient solutions to the affine scheduling problem. part I. one-dimensional time. *Int. J. of Parallel Programming*, 21(5):313–347, Oct. 1992.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, Dec. 1992.

[10] T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. *Springer Verlag Lecture Notes in Computer Science*, 274:257–277, 1987.

[11] M. Gordon, W. Thies, M. Karczmarek, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *In the Proceedings of the Tenth Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[12] R. Govindarajan, G. Gao, and P. Desai. Minimizing memory requirements in rate-optimal schedules. In *Proceedings of the 1994 International conference on Application Specific Array Processors*, pages 75–86, August 1994.

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

[14] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinancy, termination, queuing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, November 1966.

[15] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM (JACM)*, 14(3):563–590, 1967.

[16] R. Lauwereins, M. Engels, M. Ade, and J. Peperstraete. Grape-II: A System-Level Prototyping Environment for DSP Applications. *IEEE Computer*, 28(2), February 1995.

[17] E. A. Lee. Overview of the ptolemy project. Technical Report Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, March 2001.

[18] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, January 1987.

[19] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proc. of the IEEE*, September 1987.

[20] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3–4):649–671, May 1998.

[21] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3–4):445–475, May 1998.

[22] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 103–112. ACM Press, 2001.

[23] M. Manjunathaiah, G. Megson, T. Risset, and S. Rajopadhye. Uniformization tool for systolic array designs. Technical Report PI-1350, IRISA, June 2000.

[24] P. K. Murthy and S. S. Bhattacharyya. Shared Buffer Implementations of Signal Processing Systems using Lifetime Analysis Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):177–198, February 2001.

[25] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee. Joint Minimization of Code and Data for Synchronous Dataflow Programs. *Journal of Formal Methods in System Design*, 11(1):41–70, July 1997.

[26] P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, July 2002.

[27] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cyclo-static dataflow. In *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, October 1995.

[28] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, September 2000.

[29] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrence equations. In *Proceedings of the 11th Annual Symposium. on Computer Architecture*, pages 208–214, 1984.

[30] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction*, Grenoble, France, 2002.

[31] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 232–242. ACM Press, 2001.

9 Appendix: Detailed Example

In this section we present an example translation from a PCP to a SARE. Our source language is StreamIt, which is a high-level language designed to offer a simple and portable means of constructing phased computation programs. We are developing a StreamIt compiler with aggressive optimizations and backends for communication-exposed architectures. For more information on StreamIt, please consult [30, 11].

9.1 StreamIt Code for an Equalizer

Figure 7 contains a simple example of an equalizing software radio written in StreamIt. The phased computation graph corresponding to this piece of code appears in Figure 8. For simplicity, we model the input to the Equalizer as a RadioSource that pushes one item on every invocation. Likewise, the output of the Equalizer is fed to a Speaker that pops one item on every invocation.

The Equalizer itself has two stages: a splitjoin that filters each frequency band in parallel (using BandPassFilter's), and an Adder that combines the output from each of the parallel filters. The splitjoin uses a duplicate splitter to send a copy of the input stream to each BandPassFilter. Then, it contains a roundrobin joiner that inputs an item from each parallel stream in sequence. The roundrobin joiner is an example of a node that has multiple steady-state phases: in each phase, it copies an item from one of the parallel streams to the output of the splitjoin. In StreamIt, splitter and joiner nodes are compiler-defined primitives.

For an example of a user-defined node, consider the Adder filter. The Adder takes an argument, N, indicating the number of items it should add. The declaration of its steady-state work block indicates that on each invocation, the Adder pushes 1 item to the output channel and pops N items from the input channel. The code within the work function performs the addition.

The BandPassFilter is an example of a filter with both an initial and steady-state epoch. The code within the init block is executed before any node fires; in this case, it calculates the weights that should be used for filtering. The prework block specifies the behavior of the filter for the initial epoch. In the case of an FIR filter, the initial epoch is essential for dealing with the startup condition when there are fewer items available on the input channel than there are taps in the filter. Given that there are N taps in the filter, the prework function[3] performs the FIR filter on the first i items, for $i = 1 \ldots$ N-1. Then, the steady state work block operates on N items, popping an item from the input channel after each invocation.

9.2 Converting to a SARE

We will generate a SARE corresponding to N steady-state executions of the above PCP.

9.2.1 The Steady-State Period

The first step in the translation is to calculate $S(n)$, the number of times that a given node n fires in a periodic steady-state execution (see Section 5.1). Using $S(n)$ we can also derive $\mathit{Period}(c)$, the number of items that are transferred over channel c in one steady-state period. This can be done using balance equations on the steady-state I/O rates of the stream [3]:

$$\forall c = (n_a, n_b): \quad S(n_a) * \mathit{TotalWrite}(n_a, c, \mathit{steady}) = S(n_b) * \mathit{TotalRead}(n_b, c, \mathit{steady})$$

[3] To simplify the equations, we have written the BandPassFilter to have only one phase in the initial epoch. However, it would also be possible to give a more fine-grained description with multiple initial phases.

```
void->void pipeline EqualizingRadio {
   add RadioSource();
   add Equalizer(2);
   add Speaker();
}

float->float pipeline Equalizer (int BANDS) {
   add splitjoin {
      split duplicate;
      float centerFreq = 100000;
      for (int i=0; i<BANDS; i++, centerFreq*=2) {
         add BandPassFilter(centerFreq, 50);
      }
      join roundrobin;
   }
   add Adder(BANDS);
}

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

float->float filter BandPassFilter (float centerFreq,
                                    int N) {
   float[N] weights;

   init {
      weights = calcImpulseResponse(centerFreq, N);
   }

   prework push N-1 pop 0 peek N-1 {
      for (int i=1; i<N; i++) {
         push(doFIR(i));
      }
   }

   work push 1 pop 1 peek N {
      push(doFIR(N));
      pop();
   }

   float doFIR(int k) {
      float val = 0;
      for (int i=0; i<k; i++) {
         val += weights[i] * peek(k-i-1);
      }
      return val;
   }
}
```

Figure 7: StreamIt code for a simple software radio with equalizer.

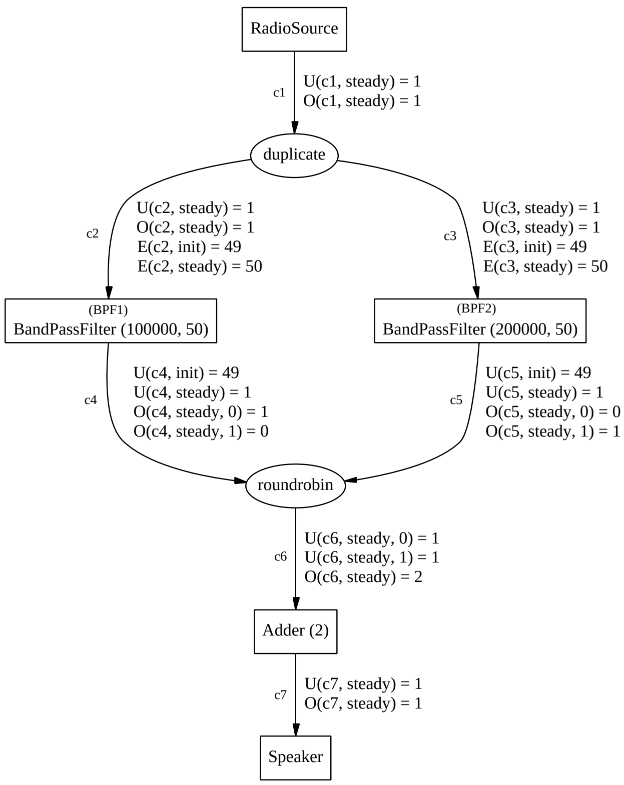

Figure 8: Stream graph of the 2-band equalizer. Channels are annotated with their push (U), pop (O), and peek (E) rates. Rates with a value of zero are omitted.

where *TotatalRead* and *TotalWrite* are defined as in Figure 2 to denote the total number of items that a node reads and writes to a given channel during one cycle of its phases. Expanding these definitions for the stream graph in Figure 8, we have:

$$\begin{aligned}
S(\mathit{RadioSource}) * \mathit{TotalWrite}(\mathit{RadioSource}, c1, \mathit{steady}) &= S(\mathit{duplicate}) * \mathit{TotalRead}(\mathit{duplicate}, c1, \mathit{steady}) \\
S(\mathit{duplicate}) * \mathit{TotalWrite}(\mathit{duplicate}, c2, \mathit{steady}) &= S(\mathit{BPF}_1) * \mathit{TotalRead}(\mathit{BPF}_1, c2, \mathit{steady}) \\
S(\mathit{duplicate}) * \mathit{TotalWrite}(\mathit{duplicate}, c3, \mathit{steady}) &= S(\mathit{BPF}_2) * \mathit{TotalRead}(\mathit{BPF}_2, c3, \mathit{steady}) \\
S(\mathit{BPF}_1) * \mathit{TotalWrite}(\mathit{BPF}_1, c4, \mathit{steady}) &= S(\mathit{roundrobin}) * \mathit{TotalRead}(\mathit{roundrobin}, c4, \mathit{steady}) \\
S(\mathit{BPF}_2) * \mathit{TotalWrite}(\mathit{BPF}_2, c5, \mathit{steady}) &= S(\mathit{roundrobin}) * \mathit{TotalRead}(\mathit{roundrobin}, c5, \mathit{steady}) \\
S(\mathit{roundrobin}) * \mathit{TotalWrite}(\mathit{roudnrobin}, c6, \mathit{steady}) &= S(\mathit{Adder}) * \mathit{TotalRead}(\mathit{Adder}, c6, \mathit{steady}) \\
S(\mathit{Adder}) &= S(\mathit{Speaker})
\end{aligned}$$

Evaluating the I/O rates, this simplifies to:

$$\begin{aligned}
S(\mathit{RadioSource}) &= S(\mathit{duplicate}) \\
S(\mathit{duplicate}) &= S(\mathit{BPF}_2) \\
S(\mathit{duplicate}) &= S(\mathit{BPF}_2) \\
S(\mathit{BPF}_1) &= S(\mathit{roundrobin}) \\
S(\mathit{BPF}_2) &= S(\mathit{roundrobin}) \\
S(\mathit{roundrobin}) * 2 &= S(\mathit{Adder}) * 2 \\
S(\mathit{Adder}) &= S(\mathit{Speaker})
\end{aligned}$$

Solving for the minimum integral solution for S gives that $S(n) = 1$ for all nodes n:

$$S(\mathit{RadioSource}) = S(\mathit{duplicate}) = S(\mathit{BPF}_1) = S(\mathit{BPF}_2) = S(\mathit{roundrobin}) = S(\mathit{Adder}) = S(\mathit{Speaker}) = 1$$

We can now calculate $\mathit{Period}(c)$ for each channel c in the graph. Using the definition of *Period* from Figure 2, we have the following:

$$\begin{aligned}
\mathit{Period}(c1) &= S(\mathit{RadioSource}) * TotalWrite(\mathit{RadioSource}, c1, \mathit{steady}) \\
\mathit{Period}(c2) &= S(\mathit{duplicate}) * TotalWrite(\mathit{duplicate}, c2, \mathit{steady}) \\
\mathit{Period}(c3) &= S(\mathit{duplicate}) * TotalWrite(\mathit{duplicate}, c3, \mathit{steady}) \\
\mathit{Period}(c4) &= S(\mathit{BPF}_1) * TotalWrite(\mathit{BPF}_1, c4, \mathit{steady}) \\
\mathit{Period}(c5) &= S(\mathit{BPF}_2) * TotalWrite(\mathit{BPF}_2, c5, \mathit{steady}) \\
\mathit{Period}(c6) &= S(\mathit{roundrobin}) * TotalWrite(\mathit{roundrobin}, c6, \mathit{steady}) \\
\mathit{Period}(c7) &= S(\mathit{Adder}) * TotalWrite(\mathit{Adder}, c7, \mathit{steady})
\end{aligned}$$

Evaluating each right hand side gives the following:

$$
\begin{aligned}
Period(c1) = Period(c2) = Period(c3) = Period(c4) = Period(c5) &= 1 \\
Period(c6) &= 2 \\
Period(c7) &= 1
\end{aligned}
$$

9.2.2 Variables of the SARE

Let us consider each variable from Figure 3 in turn.

IBUF

The *IBUF* variables are for holding both initial items and items that are produced during the initial epoch. In our example, the only such nodes are the BandPassFilter's, which push 49 items onto *c4* and *c5* in the initial epoch:

$$
\begin{aligned}
\mathcal{D}_{IBUF_{c4}} &= \{\ i \mid 0 \leq i \leq 48\} \\
\mathcal{D}_{IBUF_{c5}} &= \{\ i \mid 0 \leq i \leq 48\}
\end{aligned}
$$

The domains of all the other *IBUF* variables are empty.

IWRITE

The *IWRITE* variables are for holding items that are produced during the initial epoch. In the general case, $IWRITE_c$ might have a smaller extent than $IBUF_c$, since $IBUF_c$ also holds the $A(c)$ items that appear on channel c before the initial epoch. However, in our example there are no initial items, and the domains for *IWRITE* exactly mirror those of *IBUF*:

$$
\begin{aligned}
\mathcal{D}_{IWRITE_{c4}} &= \{\ i \mid 0 \leq i \leq 48\} \\
\mathcal{D}_{IWRITE_{c5}} &= \{\ i \mid 0 \leq i \leq 48\}
\end{aligned}
$$

IREAD

The *IREAD* variables are for holding items that are read during the initial epoch. In our example, each of the BandPassFilter's read 49 items during the initial epoch, thereby yielding an *IREAD* variable on each of their input channels:

$$
\begin{aligned}
\mathcal{D}_{IREAD_{c2}} &= \{\ i \mid 0 \leq i \leq 48\} \\
\mathcal{D}_{IREAD_{c3}} &= \{\ i \mid 0 \leq i \leq 48\}
\end{aligned}
$$

For all other channels, the domain of *IREAD* is empty.

SBUF

The *SBUF* variables represent the steady-state buffer space on a channel. They contain two dimensions: the first counts over steady-state execution cycles, and the second counts over items that appear on the channel during a given cycle (that is, the *Period* of the channel). In the case of our example, all channels have a period of 1, except for *c6*, which has a period of 2. Thus, the equations for *SBUF* are as follows:

$$
\begin{aligned}
\mathcal{D}_{SBUF_{c1}} &= \{\ (i,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SBUF_{c2}} &= \{\ (i,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SBUF_{c3}} &= \{\ (i,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SBUF_{c4}} &= \{\ (i,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SBUF_{c5}} &= \{\ (i,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SBUF_{c6}} &= \{\ (i,j) \mid 0 \le i \le N-1 \ \wedge\ 0 \le j \le 1\} \\
\mathcal{D}_{SBUF_{c7}} &= \{\ (i,0) \mid 0 \le i \le N-1\}
\end{aligned}
$$

SWRITE

The *SWRITE* variables represent temporary buffers for the output of nodes in the steady state. Here *c6* is distinguished because it has two separate phases, corresponding to the cyclic behavior of the *roundrobin* node. Since each phase outputs an item, there is a buffer to hold the output of each:

$$
\begin{aligned}
\mathcal{D}_{SWRITE_{c1}} &= \{\ (i,0,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SWRITE_{c2}} &= \{\ (i,0,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SWRITE_{c3}} &= \{\ (i,0,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SWRITE_{c4}} &= \{\ (i,0,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SWRITE_{c5}} &= \{\ (i,0,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SWRITE_{c6,0}} &= \{\ (i,0,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SWRITE_{c6,1}} &= \{\ (i,0,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SWRITE_{c7}} &= \{\ (i,0,0) \mid 0 \le i \le N-1\}
\end{aligned}
$$

SREAD

The *SREAD* variables represent temporary buffers for the nodes to read from in the steady state. The k dimension of these buffers represents the number of items that are read at once; this is 50 in the case of the inputs to the BandPassFilter's, and 1 for all other channels:

$$
\begin{aligned}
\mathcal{D}_{SREAD_{c1}} &= \{\ (i,0,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SREAD_{c2}} &= \{\ (i,0,k) \mid 0 \le i \le N-1 \ \wedge\ 0 \le k \le 49\} \\
\mathcal{D}_{SREAD_{c3}} &= \{\ (i,0,k) \mid 0 \le i \le N-1 \ \wedge\ 0 \le k \le 49\} \\
\mathcal{D}_{SREAD_{c4,0}} &= \{\ (i,0,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SREAD_{c5,1}} &= \{\ (i,0,0) \mid 0 \le i \le N-1\} \\
\mathcal{D}_{SREAD_{c6}} &= \{\ (i,0,k) \mid 0 \le i \le N-1 \ \wedge\ 0 \le k \le 1\} \\
\mathcal{D}_{SREAD_{c7}} &= \{\ (i,0,0) \mid 0 \le i \le N-1\}
\end{aligned}
$$

Note that channels *c4* and *c5* have two separate phases, corresponding to the cycle of the roundrobin joiner. However, since the roundrobin joiner does not read from *c5* during phase 0, and does not read from *c4* during phase 1, the domains for these arrays are empty. Instead, there is a buffer only for *c4* at phase 0 and *c5* at phase 1.

9.2.3 Equations of the SARE

We consider each of the equations from Figures 5 and 6 in turn.

I to IBUF

In our example, this equation always has an empty domain, as there are no channels with initial items. (That is, for all c, $A(c) = 0$.)

IWRITE to IBUF

This equation has a non-empty domain for channels that are written to during the initialization epoch, which in the case of our example is *c4* and *c5*. Since both of these channels have only one phase of writing during this epoch, these equations specify a simple copy from *IWRITE* to *IBUF*:

$$\forall i \in [0,48]: \quad IBUF_{c4}(i) = IWRITE_{c4}(i)$$
$$\forall i \in [0,48]: \quad IBUF_{c5}(i) = IWRITE_{c5}(i)$$

IREAD to IWRITE

These equations represent the computation of nodes that fire during the initialization epoch. In our example, the only such nodes are the BandPassFilter's, and there is one equation for each of them:

$$\forall i \in [0,48]: \quad IWRITE_{c4}(i) = W(BPF_1, init)(IREAD_{c2}(*))[0][i]$$
$$\forall i \in [0,48]: \quad IWRITE_{c5}(i) = W(BPF_2, init)(IREAD_{c3}(*))[0][i]$$

In these equations, $W(BPF_1, init)$ and $W(BPF_2, init)$ refer to the `prework` function defined in the BandPassFilter.

SREAD to SWRITE

These equations represent the steady-state computation of the nodes. For the BandPassFilter and Adder nodes, the computation is according to the user-defined work function W (corresponding to the `work` function in the StreamIt code):

$$\forall i \in [0,N-1]: \quad SWRITE_{c4}(i,0,0) = W(BPF_1, steady)(SREAD_{c2}(i,0,*))[0][0]$$
$$\forall i \in [0,N-1]: \quad SWRITE_{c5}(i,0,0) = W(BPF_2, steady)(SREAD_{c3}(i,0,*))[0][0]$$
$$\forall i \in [0,N-1]: \quad SWRITE_{c7}(i,0,0) = W(Adder, steady)(SREAD_{c6}(i,0,*))[0][0]$$

For the *duplicate* and *roundrobin* nodes, the work function is simple and compiler-defined, so we give the equations directly without appealing to a work function:

$$\forall i \in [0,N-1]: \quad SWRITE_{c2}(i,0,0) = SREAD_{c1}(i,0,0)[0][0] \qquad (c1 \rightarrow duplicate \rightarrow c2)$$
$$\forall i \in [0,N-1]: \quad SWRITE_{c3}(i,0,0) = SREAD_{c1}(i,0,0)[0][0] \qquad (c1 \rightarrow duplicate \rightarrow c3)$$
$$\forall i \in [0,N-1]: \quad SWRITE_{c6,0}(i,0,0) = SREAD_{c4,0}(i,0,0)[0][0] \qquad (c4 \rightarrow roundrobin \rightarrow c6)$$
$$\forall i \in [0,N-1]: \quad SWRITE_{c6,1}(i,0,0) = SREAD_{c5,1}(i,0,0)[0][0] \qquad (c5 \rightarrow roundrobin \rightarrow c6)$$

SWRITE to SBUF

This equation transfers items from the index space in which they were written to the index space of the buffer for a given channel. In the general case (Equation 18) we need to "slice" the domain into $S(n)$ pieces in order to maintain a uniform left-hand side (see Section 5.2). However, in this example, $S(n) = 1$ for all n, and this equation becomes a direct copy between the *SWRITE* and *SBUF* variables. There is one such copy for each channel in the program:

$$
\begin{aligned}
\forall i \in [0, N-1]: &\quad \mathit{SBUF}_{c1}(i, 0) = \mathit{SWRITE}_{c1}(i, 0, 0) \\
\forall i \in [0, N-1]: &\quad \mathit{SBUF}_{c2}(i, 0) = \mathit{SWRITE}_{c2}(i, 0, 0) \\
\forall i \in [0, N-1]: &\quad \mathit{SBUF}_{c3}(i, 0) = \mathit{SWRITE}_{c3}(i, 0, 0) \\
\forall i \in [0, N-1]: &\quad \mathit{SBUF}_{c4}(i, 0) = \mathit{SWRITE}_{c4}(i, 0, 0) \\
\forall i \in [0, N-1]: &\quad \mathit{SBUF}_{c5}(i, 0) = \mathit{SWRITE}_{c5}(i, 0, 0) \\
\forall i \in [0, N-1]: &\quad \mathit{SBUF}_{c6}(i, 0) = \mathit{SWRITE}_{c6,0}(i, 0, 0) \\
\forall i \in [0, N-1]: &\quad \mathit{SBUF}_{c6}(i, 1) = \mathit{SWRITE}_{c6,1}(i, 0, 0) \\
\forall i \in [0, N-1]: &\quad \mathit{SBUF}_{c7}(i, 0) = \mathit{SWRITE}_{c7}(i, 0, 0)
\end{aligned}
$$

Note that *c6* is slightly different because there is one equation required for each phase of writing.

IBUF to IREAD

This equation has a non-empty domain for channels which contain items that are both written and read during the initial epoch. In the case of our example, none such channels exist, because the only nodes with an initial epoch (the BandPassFilter's) do not communicate with each other. Mathematically, one can see that the domain of this equation is empty because there is no channel with both a non-empty *IREAD* variable and a non-empty *IBUF* variable, as would be required for $\mathcal{D}_{IB \to IR}$ to be non-empty in Equation 19.

SBUF to IREAD

This equation has a non-empty domain for channels which contain items that are written during the steady-state epoch but read during the initial epoch. In the case of our example, channels *c2* and *c3* satisfy this criterion. According to Equation 20, we need to introduce 49 equations for each channel (q ranges from 0 to 49, but when $q = 49$ the domain is empty). Doing so would give the following:

$$
\begin{aligned}
\forall q \in [0, 48]: &\quad \forall i \in \{q\}, \mathit{IREAD}_{c2}(i) = \mathit{SBUF}_{c2}(i, 0) \\
\forall q \in [0, 48]: &\quad \forall i \in \{q\}, \mathit{IREAD}_{c3}(i) = \mathit{SBUF}_{c3}(i, 0)
\end{aligned}
$$

In these equations, we have encountered a special case where *Period(c1)* = *Period(c2)* = 1, and it is not necessary to slice the domain (as the index to the second dimension of *SBUF* is constant.) Thus, we can reduce these 98 equations into just 2, where the domain of each equation is the domain of q above:

$$
\begin{aligned}
\forall i \in [0, 48]: &\quad \mathit{IREAD}_{c2}(i) = \mathit{SBUF}_{c2}(i, 0) \\
\forall i \in [0, 48]: &\quad \mathit{IREAD}_{c3}(i) = \mathit{SBUF}_{c3}(i, 0)
\end{aligned}
$$

However, in the general case, it might be necessary to introduce a large number of equations in this step. The number of equations will grow with the extent of a node's peeking during the initial epoch. Note that the number of equations will *not* grow with the number of steady-state cycles that the SARE is simulating.

IBUF to SREAD

This equation has a non-empty domain for channels which contain items that are written during the initial epoch but read during the steady-state epoch. In the case of our example, channels *c4* and *c5* satisfy this criterion. The following equations simply copy the items from the initial buffer into the steady-state buffer:

$$\forall i \in [0,48]: \quad SREAD_{c4,0}(i,0,0) = IBUF_{c4}(i)$$
$$\forall i \in [0,48]: \quad SREAD_{c5,1}(i,0,0) = IBUF_{c5}(i)$$

Note that these equations are somewhat simpler than those in Equation 21 because no items are read from *c4* or *c5* during the initial epoch. If this were the case, then there would be offsets in the domains above.

SBUF to SREAD

These equations represent the copying of items from a channel's buffer to the read array that a node will access. There are two equations for this operation (Equations 22 and 23) because, in the general case, offsets due to reading and writing in the initial epoch could necessitate two different copy operations. This will occur if the offset is not a multiple of the extent of the j dimension of *SBUF*, in which case the copy of the j dimension is split across two different values of i: one equation copies the upper part of the j dimension of *SBUF* for a given i, and the other equation copies the lower part of the j dimension for $i+1$. However, in the case of our example, the extent of the j dimension is 1 (because the period of each channel is 1) and the offset always is a multiple of 1, of course. Thus, Equation 23 always has an empty domain, and we consider only Equation 22 below. As the notation for this equation is rather heavy, we consider each channel individually.

First let us examine *c1* and *c2*, which are simple because they do not peek and they contain only one epoch. In this case, Equation 22 copies items directly from the *SBUF* array to the *SREAD* array:

$$\forall i \in [0,N-1]: \quad SREAD_{c1}(i,0,0) = SBUF_{c1}(i,0)$$
$$\forall i \in [0,N-1]: \quad SREAD_{c7}(i,0,0) = SBUF_{c7}(i,0)$$

Only slightly more complex is the equation for *c6*, in which two items are read from the channel for each invocation of the Adder. This requires an extra subscript q on *SREAD*, that happens to coincide with *SBUF* since the Adder executes once per steady-state execution of the graph:

$$\forall q \in [0,1]: \quad \forall i \in [0,N-1]: \quad SREAD_{c6}(i,0,q) = SBUF_{c6}(i,q)$$

Note that the above translation is naively specified as two separate equations (one for each value of q); however, as we saw above, we can combine these equations into a single one by considering q to be a domain quantifier rather than an enumerator over equations.

Next, let us consider the equations for *c2* and *c3*, which encapsulate the peeking behavior of the BandPassFilter's. In this case, *SREAD* is viewing the channel as overlapping segments of 50 items each, whereas *SBUF* is viewing the channel as a continuous sequence where each item appears only once. Thus, the following equations duplicate the data by a factor of 50 as they copy it into the *SREAD* array:

$$\forall q \in [0,49]: \quad \forall i \in [0,N-1]: \quad SREAD_{c2}(i,0,q) = SBUF_{c2}(i+q,0)$$
$$\forall q \in [0,49]: \quad \forall i \in [0,N-1]: \quad SREAD_{c3}(i,0,q) = SBUF_{c3}(i+q,0)$$

Finally, consider the equations for *c4* and *c5*. On these channels, the beginning of the *SREAD* array was filled by the initial output of the BandPassFilter's (see IBUF to SREAD above). Thus, the steady-state output in the *SBUF* array needs to be copied at an offset into the *SREAD* array so that it does not overwrite the initial output. Following Equation 24, we calculate this offset as the number of items that were pushed onto the channel in the initial epoch; this is given by the size of the domain of the IBUF to SREAD equation: $|\mathcal{D}_{IB \to SR}(roundrobin, c4)| = |\mathcal{D}_{IB \to SR}(roundrobin, c4)| = 49$. Since the *c4* and *c5* both have a period of 1, this offset does not fall on the boundary between two different i indices in *SBUF*, and thus we don't need to worry about the *Int_Offset* and *Mod_Offset* defined in Equation 24. The resulting equations represent a simple shifted copy operation, as follows:

$$\forall i \in [49, N-1]: \quad SREAD_{c4,0}(i,0,0) = SBUF_{c4}(i-49,0)$$
$$\forall i \in [49, N-1]: \quad SREAD_{c5,1}(i,0,0) = SBUF_{c5}(i-49,0)$$

This concludes the translation of the StreamIt-based PCP to a SARE. The equations and variables defined above are exactly equivalent to an execution of the original StreamIt program for N steady-state cycles.