Constrained and Phased Scheduling of Synchronous Data Flow Graphs for StreamIt Language

by

Michal Karczmarek

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 2002

Author . . . .
Department of Electrical Engineering and Computer Science
December 4, 2002

Certified by . . . .
Saman Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Constrained and Phased Scheduling of Synchronous Data Flow Graphs for StreamIt Language

by

Michal Karczmarek

Submitted to the Department of Electrical Engineering and Computer Science
on December 4, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Applications that are structured around some notion of a "stream" are becoming increasingly important and widespread. There is evidence that streaming media applications are already consuming most of the cycles on consumer machines [20], and their use is continuing to grow. *StreamIt* is a language and compiler specifically designed for modern stream programming. Despite the prevalence of these applications, there is surprisingly little language and compiler for practical, large-scale stream programming. *StreamIt* is a language and compiler specifically designed for modern stream programming. The *StreamIt* langauge holds two goals: first, to provide high-level stream abstractions that improve programmer productivity and program robustness within the streaming domain; second, to serve as a common machine language for grid-based processors. At the same time, *StreamIt* compiler aims to perform stream-specific optimizations to achieve the performance of an expert programmer. This thesis develops several techniques for scheduling execution of *Filter*s in *StreamIt*. The work focuses on correctness as well as minimizing buffering requirements and stored schedule size.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

I would like to thank my advisor for infinite patience and guidance. The *StreamIt* team has been invaluable in both the research and writing. Bill Thies helped me with many initial ideas and literature search. Michael Gordon provided the best testing environment by writing the front end for our compiler and embedding the scheduler in his RAW backend. David Maze contributed to the internal compiler framework, and help with our printing of program graphs. Jasper Lin hacked our compiler to no end. Chris Leger, Ali Meli, Andrew Lamb and Jeremy Wong wrote scores of applications which I could use for testing.

I want to thank my officemates and friends for putting up with me and convincing me that there is an end to writing. They were right.

Most importantly, I want to thank my all my family and friends, for helping me get here in the first place, and providing constant support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Applications that are structured around some notion of a ”stream” are becoming increasingly important and widespread. There is evidence that streaming media applications are already consuming most of the cycles on consumer machines [20], and their use is continuing to grow. In the embedded domain, applications for hand-held computers, cell phones, and DSP's are centered around stream of voice or video data. The stream abstraction is also fundamental to high-performance applications such as intelligent software routers, cell phone base stations and HDTV editing consoles.

Despite the prevalence of these applications, there is surprisingly little language and compiler for practical, large-scale stream programming. The notion of a stream as a programming abstraction has been around for decades [1], and a number of special-purpose stream languages have been designed (see [22] for a review). Many of these languages and representations are elegant and theoretically sound, but they often lack features and are too inflexible to support straightforward development of modern stream applications, or their implementations are too inefficient to use in practice. Consequently most programmers turn to general-purpose languages such as C or C++ to implement stream programs.

There are two reasons that general-purpose languages are inappropriate for stream programming. Firstly, they are a mismatch for the application domain. That is they do not provide a natural or intuitive representation of streams thereby having a negative effect on readability, robustness, and programmer productivity. Furthermore, general-purpose languages do not communicate well the inherent parallelism of stream computations. Secondly, general-purpose languages are a mismatch for the emerging class of grid-based architectures [17, 25, 21].

StreamIt is a language and compiler specifically designed for modern stream programming. The *StreamIt* langauge holds two goals: first, to provide high-level stream abstractions that improve programmer productivity and program robustness within the streaming domain; second, to serve as a common machine language for grid-based processors. At the same time, *StreamIt* compiler aims to perform stream-specific optimizations to achieve the performance of an expert programmer.

In order to achieve these goals, *StreamIt* provides a number of features, designed to allow the programmer to easily and naturally express the required computation, while keeping the program easy to analyze by a compiler: all *StreamIt* streaming constructs are single-input, single-output; all computation happens in *Filter*s; data is passed around between *Filter*s using three streaming constructs: *Pipeline*, which allows stacking of *Filter*s one after another, *SplitJoin*, which allows splitting and joining of data amongst multiple streams, and *FeedbackLoop*, which constructs cyclic streams. In *StreamIt*, every *Filter* must declare the rate at which it processes data: how much data is consumed and produced on every invocation of the *Filter*'s *work* function. This model of data passing is called Synchronous Data Flow (SDF).

In addition to *SDF*, *StreamIt* allows the programmer to pass data between *Filter*s in an asynchronous manner, similar to a combination of message passing and function calls. Timing of such data delivery is expressed in terms of amount of information wavefronts - the programmer can specify a delay between message delivery and destination *Filter*'s processing of data currently being produced or consumed by the source *Filter*. Such timing mechanism introduces latency and buffering constraints on execution of *StreamIt* programs.

Using the features present in *StreamIt*, the programmer can express complex algorithms and computation models. One of the difficulties faced by *StreamIt* is scheduling of the execution of the program. Since *StreamIt* uses SDF computation model with latency constraints, it is possible to schedule the order of execution of *Filter*s at compile time. Scheduling SDF programs presents a difficult challenge to the compiler: as the complexity of the program grows, the amount of memory required to execute the program increases. This increase comes from two sources: the schedule size is creases, as well as amount of data needed for buffering increases. These two sources are closely coupled. There exist tradeoffs between the schedule size and the buffer size.

This problem is further complicated by message latency constraints placed on the program by the programmer. While *StreamIt* programs are meant to provide relatively lax latency requirements, it is possible to write programs with latency constraints so tight that very few valid schedules exist. Finding these schedules is a challenging task.

This thesis develops several techniques for scheduling execution of *StreamIt* programs. This thesis will present techniques which take advantage of structure of *StreamIt* to create compact schedules. These schedules will be purely hierarchical in nature. The concept of a phasing schedule will be introduced to reduce the requirement for buffering data between *Filter*s without overly increasing the size of the schedule. Finally, the problem of scheduling programs with message latency constraints will be solved using integer programming.

The contributions of this thesis are:

- hierarchical scheduling of streaming application, a concept enabled by *StreamIt* language,
- first formal handling of SDF graphs with peeking,
- novel phasing scheduling technique,
- a minimal latency schedule using hierarchical phases,
- novel SDF program abstraction called the information buffering model that simplifies information latency analysis,
- a solution to scheduling of *StreamIt* programs with latency constraints.

The remainder of this thesis is organized as follows: chapter 2 describes relevant *StreamIt* constructs in detail; chapter 3 explains basic concepts in scheduling *StreamIt* graphs; chapter 4 describes hierarchical phasing techniques; chapter 5 describes phasing scheduling techniques, including phased scheduling, a more advanced approach to scheduling; chapter 6 introduces techniques for scheduling of *StreamIt* programs with latency constraints; chapter 7 analyzes performance of various algorithms described here.

Chapter 2

StreamIt Language

This chapter introduces relevant constructs of the *StreamIt* language. Syntax is not explored here, as it is not relevant to *StreamIt* scheduling.

Section 2.1 introduces the structured streaming concept, while Section 2.2 introduces the low bandwidth messaging semantics of *StreamIt*.

2.1 Structure

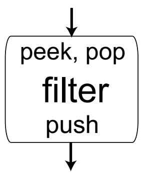
Perhaps the most distinguishing feature of *StreamIt* language is that it introduces structure to the concept of stream computation. *StreamIt* concept of structure is conceptually similar to structured constructs in functional languages such as *C*.

In *StreamIt* programs are composed out of streaming components called streams. Each stream is a single-input, single-output component, possibly made up of a hierarchical composition of other streams. Streams can only be arranged in a limited number of ways, using *Pipeline*s, *SplitJoin*s, and *FeedbackLoop*s. Data passed between *Filter*s is read from and written to *channel*s. Figure 2-1 contains examples of various *StreamIt* streams. The restrictions on arrangement of streams enforces the structure imposed by *StreamIt*.

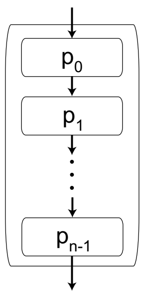
2.1.1 *Filter*s

The basic unit of computation in *StreamIt* is the *Filter*. The central aspect of a filter is the *work* function, which describes the filter's atomic execution step. Within the *work* function, the filter can communicate with its neighbors using the *input* and *output* channels, which are typed FIFO queues declared during initialization of a *Filter*. Figure 2-1(a) depicts a *Filter*.

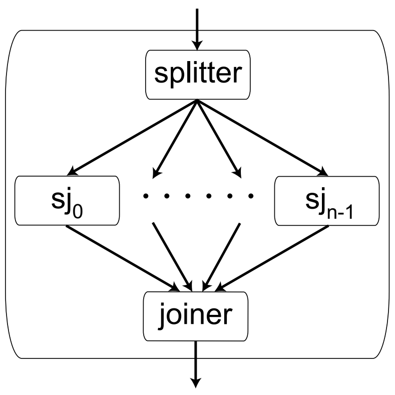
*Filter*s also have the restriction of requiring a static amount of data to be consumed and produced for each execution of a *work* function. The amount of data produced by a



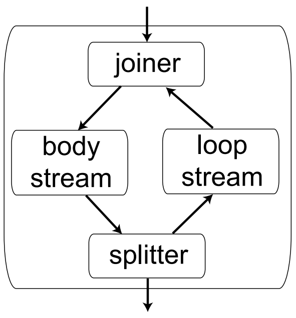
(a) A *Filter*



(b) A *Pipeline* with n children.



(c) A *SplitJoin* with n children



(d) A *FeedbackLoop*

Figure 2-1: All *StreamIt* streams

Filter F upon execution of its *work* function is called a push amount, denoted *push*. The amount of data consumed from *input channel* by a *Filter F* upon execution of its *work* function is called a pop amount, denoted *pop*. *Filter*s may require that additional data be available in the *input channel* for the *Filter* to examine. This data can be read by the *Filter*'s *work* function, but it will not be consumed, and will remain in the *channel* for the next execution of the *work* function. The amount of data necessary on the *input channel* to execute *Filter*'s *work* function is called peek amount, denoted *peek*. Note, that for all *Filter*s $peek >= pop$. Extra peek amount is the amount of data required on by the *Filter* that will be read but will not be consumed, namely $peek - pop$. The *peek*, *pop* and *push* values in Figure 2-1(a) correspond to the *peek*, *pop* and *push* amounts of the *Filter*'s *work* function.

A *Filter* can be a source, if it does not consume any data, but it produced data. Namely, a *Filter* is a source if it has $peek = pop = 0$. Likewise, a *Filter* can be a sink, if it consumes data, but does not produce any, or $push = 0$.

2.1.2 ***Pipeline*s**

*Pipeline*s are used to connect *StreamIt* structures in a chain fashion: each child stream's output is the next child stream's input. *Pipeline*s have no *work* function, as they do not perform any computation themselves. *Pipeline*s are simply containers of other *StreamIt* structures. Figure 2-1(b) depicts a *Pipeline*.

2.1.3 ***SplitJoin*s**

*SplitJoin*s are used to specify independent parallel structures that diverge from a common *splitter* and merge into a common *joiner*. There are two types of *splitter*s:

(a) *Duplicate*, which replicates each data item and sends a copy to each parallel stream, and

(b) *RoundRobin* $(w_0, \ldots, w_{n-1})$, which sends the first w_0 items to the first stream, the next w_1 items to the second stream, and so on. If all w_i are equal to 0, all child streams of the *SplitJoin* must be sources.

RoundRobin is also the only type of a *joiner* supported in *StreamIt*; its function is analogous to a *RoundRobin splitter*.

Figure 2-1(c) depicts a *SplitJoin*.

2.1.4 ***FeedbackLoop*s**

*FeedbackLoop*s are used to create cycles in the stream graph. A *FeedbackLoop* contains a *joiner*, a body stream, a *splitter*, and a loop stream. Figure 2-1(d) depicts a *FeedbackLoop*.

A *FeedbackLoop* has an additional feature required to allow a *FeedbackLoop* to begin computation: since there is no data on the feedback path at first, the stream instead inputs data from a special function defined by the *FeedbackLoop*. The amount of data pushed onto the feedback path is called delay amount, denoted $delay_{fl}$, for a *FeedbackLoop* fl.

2.2 Messages

In addition to passing data between *Filter*s using structured streams, *StreamIt* provides a method for low-bandwidth data passing, similar to a combination of sending messages and function calls. Messages are sent from within the body of a *Filter*'s *work* function, perhaps to change a parameter in another *Filter*. The sender can continue to execute while the message is en route. When the message arrives at its destination, a special message receiver method is called within the destination *Filter*. Since message delivery is asynchronous, there can be no return value; only void methods can be message targets. This allows the send to continue execution while the message is en route - the sender does not have to wait for the receiver to receive the message and send a return value back. If the receiver wants to send a return value to the sender, it can send a message back to the sender.

Although message delivery in *StreamIt* is asynchronous in principle, *StreamIt* does include semantics to restrict the latency of delivery of a message. Since *StreamIt* does not provide any shared resources to *Filter*s (including global memory, global clock, etc), the timing mechanism uses a concept of flow of information.

One motivating example for messaging in *StreamIt* can be found in cell phone processing application. Modern cellular phone protocols involve a technique called frequency hopping - the cell phone base station selects a new frequency or channel for the phone to communicate with the base station and informs the phone of this change. The phone must switch to the new channel within a certain amount of time, or it risks losing connection with the base station.

If the phone decoder application is written in *StreamIt*, the *Filter* controlling the antenna and the *Filter* which will process control signals are likely far apart, and may not have a simple way of communicating data directly with each other. In *StreamIt*, the *Filter* which decodes control signals can simply send a message to the *Filter* controlling the antenna. The message can be sent with a specific latency corresponding to the timing required by the base station. When the antenna controller receives the message it can change the appropriate settings in the hardware to switch to the appropriate new frequency, without having to wait for the appropriate time. The timing of delivery is taken care of by *StreamIt*.

2.2.1 Information Wavefronts

When a data item enters a stream, it carries with it some new information. As execution progresses, this information cascades through the stream, affecting the state of *Filter*s and the values of new data items which are produced. We refer to an information wavefront as the set of *Filter* executions that first sees the effects of a given input item. Thus, although each *Filter*'s *work* function is invoked asynchronously without any notion of global time, two invocations of a work function occur at the same information-relative time if they operate on the same information wavefront.

2.2.2 Message Sending

Messages can be sent upstream or downstream between any two *Filter*s. Sending messages across branches of a *SplitJoin* is not legal. Timing of message delivery uses the concept of information wavefront. The sender specifies that the message is supposed to be delivered with a certain delay of information wavefront. The delays are specified as ranges, $[l_0, l_1], l_0 \leq l_1$. l_0 and l_1 specify the information wavefront in executions of the *work* function of the sender *Filter*.

If the message is being sent downstream, the sender specifies that the receiver will receive the data just before it sees the information wavefront produced by the sender between l_0 and l_1 executions of its *work* function from when it sends the message. If the message is being sent upstream, the sender specifies that the receiver must receive the message just before it produces an information wavefront the sender will see between l_0 and l_1 executions of its *work* function from when it sends the message.

Message sending is meant to be a low-bandwidth method of communication between *Filter*s. Message sending is not a fast operation and is intended not to interfere with the high bandwidth *StreamIt* communication and processing. However, depending on how tight the latency constraints are (both the magnitude of the latency as well as the range), declaring that messages can be sent may slow program execution down considerably.

Figure 2-2 presents an example of a *Pipeline* in which the last *Filter* sends a message to the first *Filter*. *Filter*$_3$ sends a message to *Filter*$_0$. The message is sent with latency $[3, 8]$. This means that after at least 3 and at most 8 executions of sender's *work* function, it will see data produced by the receiver just after receiving the message.

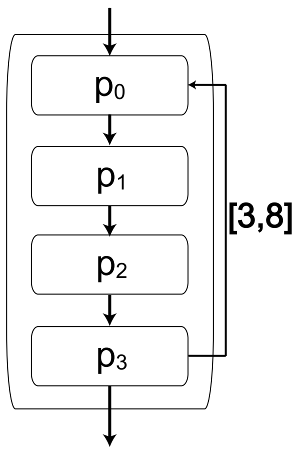


Figure 2-2: Example of a *Pipeline* with a message being sent

Chapter 3

General *StreamIt* Scheduling Concepts

This chapter introduces the general concepts used for scheduling *StreamIt* programs. Concepts presented here are are common with other languages [16] [3] [14].

Section 3.1 presents the *StreamIt* execution model. Section 3.2 introduces the concept of a steady state and shows how to calculate it. Section 3.3 explains the need for initialization of *StreamIt* program. Section 3.4 introduces simple notation for expressing schedules while Section 3.5 presents the tradeoff between schedule and buffer storage requirements.

3.1 *StreamIt* execution model

A *StreamIt* program is represented by a directed graph, $G = (N, E)$. A node in G is either a *Filter*, a *splitter* or a *joiner*. Edges in G represent data *channel*s. Each node in G takes data from its *input channel*(s), processes this data, and puts the result on the *output channel*(s). Each data *channel* is simply a FIFO queue.

Each *Filter* node n_f has exactly one incoming edge and one outgoing edge. The incoming edge is referred to as an *input channel*, while the outgoing edge is called an *output channel*. A *splitter* node n_s has exactly one incoming edge (*input channel*), but has multiple outgoing edges (*output channel*s). A *joiner* node has multiple incoming edges (*input channel*s) but only one outgoing edge (*output channel*).

Each node of graph G can be executed. An execution of a node causes some data to be collected from the node's *input channel*(s), the data to be processed and the result to be put on the *output channel*(s). An execution of a node transfers the smallest amount of data across the node - it is an atomic operation. *StreamIt* uses a static data flow model, meaning that every execution of a node n will require the same amount of data to be present on node's *input channel*(s) for consumption or inspection, same amount to be consumed from the *input channel*(s) and same amount of data to be pushed onto its *output channel*(s).

Each *Filter* node n_f is associated with a 3-tuple (e_f, o_f, u_f). These three values represent the rate of data flow for the *Filter* for each execution. The first value represents the amount of data necessary to be present in its *input channel* in order to execute the *Filter*. This is also called the peek amount of the *Filter*. The second value represents the amount of data which will be consumed by the *Filter* from its *input channel*. This is called the pop amount of the *Filter*. Note, that $e_f \geq o_f$. The final value represents the amount of data that will be put on the *output channel* of the *Filter*. This is called the push amount of a *Filter*. The amount of data present in the *input channel* of a *Filter* node n_f is denoted in_f, while data present in the *output channel* is denoted out_f.

Each *splitter* node n_s is associated with a tuple (o_s, w_s). The first value represents the amount amount of data that will be consumed by n_s from its *input channel*. Thus, in order to execute n_s, there must be at least o_s data in its *input channel*. w_s is a vector of integers, each representing the amount of data that will be pushed onto a corresponding *output channel* of n_s. The amount of data present in the *input channel* of a *splitter* node n_s is denoted in_s, while data present in the ith *output channel* is denoted $out_{s,j}$.

Each *joiner* node n_j is associated with a tuple (w_j, u_j). The first value is a vector of integers, each representing the amount of data that will be consumed by n_j from its corresponding *input channel*s. In order to execute n_j, each of its *input channel*s must have at least as much data in it as the corresponding value in w_j indicates. u_j represents the amount of data that will be pushed by n_j onto its *output channel*. The amount of data present in the ith *input channel* of a *joiner* node n_j is denoted $in_{j,i}$, while data present in the *output channel* is denoted in_s.

A schedule for a *StreamIt* program is a list of executions of nodes of graph G. The list describes the order in which these nodes are to be executed. In order for a schedule to be legal, it must satisfy two conditions. The first one is that for every execution of a node, a sufficient amount of data must be present on its *input channel*(s), as described above. The second is that the execution of the schedule must require a finite amount of memory.

3.2 Steady State

A *StreamIt* schedule is an ordered list of firings of nodes in the *StreamIt* graph. Every firing of a node consumes some data from *input channel*(s) and pushes data onto the *output channel*(s).

One of the most important concepts in scheduling streaming applications is the steady state schedule. A steady state schedule is a schedule that the program can repeatedly execute forever. It has a property that the amount of data buffered up between any two nodes does not change from before to after the execution of the steady state schedule. This property is important, because it allows the compiler to statically schedule the program at compile time, and simply repeat the schedule forever at runtime. A schedule without this property cannot be repeated continuously. This is because the delta in amount of data buffered up on between nodes will continue accumulating, requiring an infinite amount of buffering space.

A steady state of a program is a collection of number of times that every node in the program needs to execute in a steady state schedule. It does not impose an order of execution of the nodes in the program.

Not every *StreamIt* program has a steady state schedule. As will be explained in Section 3.2.2, it is possible for a program to have unbalanced production and consumption of data in *SplitJoin*s and *FeedbackLoop*s. The amount of data buffered continually increases, and cannot be reduced, thus making it impossible to create a steady state schedule for them. It is also possible that a *FeedbackLoop* does not have enough data buffered up internally in order to complete execution of a full steady state, and thus deadlocks. Programs without a valid steady state schedule are not considered valid *StreamIt* programs. In other words, all valid *StreamIt* programs have a steady state schedule.

3.2.1 Minimal Steady State

The size of a steady state is defined as the sum of all executions of all the nodes in the program per iteration of the steady state.

Definition 1 *A steady state of stream s is represented by vector m of non-negative integers. Each of the elements in m represents the number of times a corresponding node in s must be executed in the steady state.*

Note that m does not impose an order of execution of nodes. Size of a steady state is the total number of executions of all the nodes in the steady state, and is represented by $\sum_i m_i$.

Next we will summerize the properties of schedules prsented in [15].

Theorem 1 (Minimal Steady State Uniqueness) *A* StreamIt *program that has a valid steady state, has a unique minimal steady state.*

Proof 1 (Minimal Steady State Uniqueness) *Assume that there are two different minimal steady states with same size. Let m and q denote vectors representing the two steady states. Let $\sum_i m_i$ denote size of schedule m and $\sum_i q_i$ denote size of schedule q. Note* *that since both m and q are minimal steady states, $\sum_i m_i = \sum_i q_i$. Since the schedules are different, there must be some j for which $m_j \neq q_j$. Assume without loss of generality that $m_j < q_j$. Since a steady state does not change the amount of data buffered between nodes, the node producing data for node i must also execute less times than corresponding node in q. Similarly, the node consuming data produced by node j also must execute less times than the corresponding node in schedule q. Since a* StreamIt *program describes a connected graph, it follows that $\forall i, m_i < q_i$. Thus $\sum_i m_i \neq \sum_i q_i$, which is a contradiction. Thus there cannot be two different minimal steady state.*

Corollary 1 (Minimal Steady State Uniqueness) *The additional property we have from the above proof is that if m represents a minimal steady and q any other steady state, then $\forall i, m_i < q_i$.*

Lemma 1 (Composition of Steady Schedules) *If m and q are two steady states for a* StreamIt *program, then $m + q$ is also a steady state.*

The above lemma is true because neither m nor q change the amount of data buffered in the *channel*s. Thus a composition of the steady states does not change the amount of data buffered in the *channel*s, which makes the composition also a steady schedule.

Corollary 2 (Composition of Steady Schedules) *If m and q are two steady states, and $\forall i, m_i > q_i$, then $w = m - q$ is also a steady state.*

If q is a steady state and $m = w + q$ is a steady state, then w must not change the amount of data buffered in *channel*s. Thus w must be a steady state.

Theorem 2 (Multiplicity of Minimal Steady States) *If a* StreamIt *program has a valid steady state, then all its steady states are strict multiples of its minimal steady state.*

Proof 2 (Multiplicity of Minimal Steady State) *Assume that there exists a steady state that is not a multiple of the minimal steady state. Let m denote the minimal steady state. Let q denote the other steady state. Note that $w = q - m$ is still a steady state, as long as all elements of w remain non-negative (by Corollary 2). Repeat subtracting m from q until no more subtractions can be performed without generating at least one negative element in vector w. Since q is not a multiple of m, $w \neq 0$. But since we cannot subtract m from w any further, $\exists i, m_i > w_i$. Since m is a minimal steady state and w is a steady state, this is impossible due to Corollary 1. Thus there are no steady states that are not multiples of the minimal steady schedule.*

3.2.2 Calculating Minimal Steady State

This section presents equations used for calculating minimal steady states. Minimal steady states are calculated recursively in a hierarchical manner. That is, a minimal steady state is calculated for all children streams of *Pipeline*, *SplitJoin* and *FeedbackLoop*, and then the schedule is computed for the actual parent stream using these minimal states as atomic executions. This yields a minimal steady state because all child streams must execute their steady states (to avoid buffering changes), and all steady states are multiples of the minimal steady states (per Theorem 2). Executing a full steady state of a stream is referred to as "executing a stream".

Notation of Steady States

In this section, the notation for *peek*, *pop* and *push* will be extended to mean entire streams in their minimal steady state execution. That is, a *Pipeline* p will consume o_p data, produce u_p data and peek e_p data on every execution of its steady state. Again, in the hierarchical view of *StreamIt* programs, a child stream of a *Pipeline* will execute its steady state atomically.

A steady state of a stream s is represented by a set S_s of elements, $S_s = \{m, N, c, v\}$. The set includes a vector m, which describes how many times each *StreamIt* node of the stream will be executed in the steady state, a corresponding ordered set N which stores all the nodes of the stream, a vector c, which holds values $[e_s, o_s, u_s]$ for stream s, and a vector v which holds number of steady state executions of all direct children of s. m and v are not the same vector, because m refers to nodes in the subgraph, while v refers only to the direct children, which may be *Filter*s, *Pipeline*s, *splitter*s and *FeedbackLoop*s.

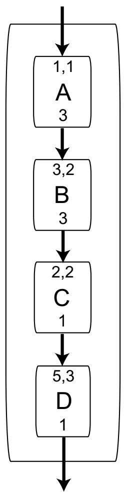
For a stream s, set S is denoted as S_s and the elements of S_s are denoted as $S_{s,m}$, $S_{s,N}$, $S_{s,c} and S_{s,v}$.

Note, that a steady state does not say anything about the ordering of the execution of nodes, only how many times each node needs to be executed to preserve amount of data buffered by the stream.

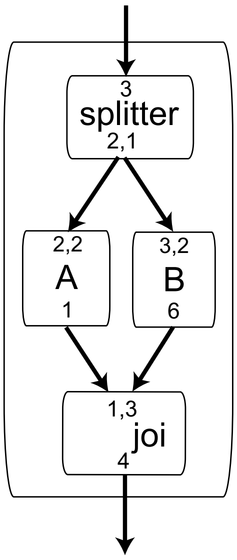
Filter

Since *Filter*s do not have any internal buffering, their minimal steady state is to execute the *Filter*'s *work* function once. This is the smallest amount of execution a *Filter* can have.

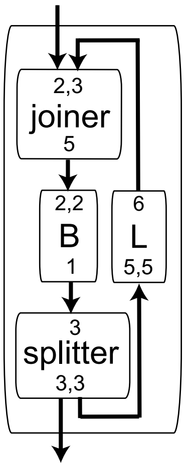
Thus, for a *Filter* f,



(a) A sample *Pipeline*



(b) A sample *SplitJoin*



(c) A sample *FeedbackLoop*. The *L Filter* has been flipped upside-down for clarity. $peek_L = pop_L = 5, push_L = 6$

Figure 3-1: Sample *StreamIt* streams

$$S_f = \left\{ [1], \{f\}, \begin{bmatrix} e_f \\ o_f \\ u_f \end{bmatrix}, [] \right\}$$

Notice that $S_{f,v}$ is empty, because a *Filter* does not have any children.

Pipeline

Let the *Pipeline* p have n children and let p_i denote the ith child of the *Pipeline* (counting from *input* to *output*, starting with 0, the children may be streams, not necessarily *Filter*s). We must find S_p.

We start with calculating all $S_{p_i}, i \in \{0, \ldots, n-1\}$. This task is achieved recursively.

Next we find a fractional vector v'' such that executing each p_i v''_i times will not change the amount of data buffered in the *Pipeline* and the first child is executed exactly once. Since the children streams are executed fractional amount of times, we calculate the amount of data they produce and consume during this execution by multiplying S_{p_i,c_o} and S_{p_i,c_u} by v''_i. Thus v'' must have the following property

$$v''_0 = 1, \forall i \in \{0, \ldots, n-1\}, v''_i * u_{p_i} = v''_{i+1} * o_{p_{i-1}}$$

We compute v'' as follows. The first child executes once, thus $v''_0 = 1$. The second child must execute $v''_1 = \frac{u_{p0}}{o_{p_1}}$ times to ensure that all data pushed on the the first *channel* is consumed by the second child. The third child must execute $v''_2 = v''_1 \frac{u_{p1}}{o_{p_2}} = \frac{u_{p0}}{o_{p_1}} \frac{u_{p1}}{o_{p_2}}$ times to ensure that it consumes all the data produced by the second child. Thus,

$$v''_i = \frac{\prod_{j=0}^{i-1} u_{p_j}}{\prod_{j=1}^{i} o_{p_j}}$$

Next we will find an integral vector v' such that executing each p_i v'_i times will not change the amount of data buffered in the *Pipeline*. v' will be a valid steady state of the *Pipeline*.

In order to calculate v' we multiply v'' by $\prod_{j=1}^{n-1} o_{p_j}$. Thus

$$v'_i = \left(\frac{\prod_{j=0}^{i-1} u_{p_j}}{\prod_{j=1}^{i} o_{p_j}} \right) \left(\prod_{j=1}^{n-1} o_{p_j} \right) = \left(\prod_{j=0}^{i-1} u_{p_j} \right) \left(\prod_{j=i+1}^{n-1} o_{p_j} \right)$$

Now we find an integral vector v, such that, for some positive integer g, $v' = g * v$, and $\sum_i v_i$ is minimal. In other words, we find the greatest integer g, such that $v' = g * v$, with v consisting of integers. v represents the minimal steady state for pipeline p.

This is achieved by finding the gcd of all elements in v', and dividing v' by g. Thus

$$v = \frac{v'}{\text{gcd}(v')}$$

v represents the number of times each child of p will need to execute its steady state in order to execute the minimal steady state of p, thus $S_{p,v} = v$. v holds a steady state because amount of data buffered in p does not change, and it is a minimal steady state, because $\sum_i v_i$ is minimal.

We construct set S_p as follows:[1]

$$S_p = \left\{ \begin{array}{c} v_0 * S_{p_0,m} \circ \ldots \circ v_{n-1} * S_{p_{n-1},m}, S_{p_0,N} \circ \ldots \circ S_{P_{n-1},N}, \\ \left[\begin{array}{c} e_{p_0} + (v_0 - 1) * o_{p_0} \\ v_0 * o_{p_0} \\ v_{n-1} * u_{p_{n-1}} \end{array} \right], v \end{array} \right\}$$

An example is presented in Figure 3-1 (a). For this *Pipeline*, we have the following steady states for all children of the *Pipeline*:

[1]Here we use symbol $\circ$ to denote concatenation of vectors and sets. Thus $[1\ 2\ 3] \circ [4\ 5\ 6] = [1\ 2\ 3\ 4\ 5\ 6]$ and $\{A\ B\ C\} \circ \{D\ E\ F\} = \{A\ B\ C\ D\ E\ F\}$.

$$S_A = \left\{ [1], \{A\}, \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix}, [] \right\}, \quad S_B = \left\{ [1], \{B\}, \begin{bmatrix} 3 \\ 2 \\ 3 \end{bmatrix}, [] \right\}$$

$$S_C = \left\{ [1], \{D\}, \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}, [] \right\}, \quad S_D = \left\{ [1], \{D\}, \begin{bmatrix} 5 \\ 3 \\ 1 \end{bmatrix}, [] \right\}$$

Using the steady states above, we get the following vector v':

$$v' = \begin{bmatrix} (2*2*3) \\ (3)(2*3) \\ (3*3)(3) \\ (3*3*1) \end{bmatrix} = \begin{bmatrix} 12 \\ 18 \\ 27 \\ 9 \end{bmatrix}$$

We now calculate $g = \gcd(v') = \gcd(12, 18, 27, 9) = 3$. We thus have

$$v = \frac{v'}{3} = \frac{1}{3} \begin{bmatrix} 12 \\ 18 \\ 27 \\ 9 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \\ 9 \\ 3 \end{bmatrix}$$

Finally, we construct S_p:

$$S_p = \left\{ \begin{array}{c} 4S_{A,m} \circ 6S_{B,m} \circ 9S_{C,m} \circ 3S_{D,m}, S_{A,N} \circ S_{B,N} \circ S_{C,N} \circ S_{D,N} \\ \begin{bmatrix} 1 + (4-1)*1 \\ 4*1 \\ 3*1 \end{bmatrix}, \begin{bmatrix} 4 \\ 6 \\ 9 \\ 3 \end{bmatrix} \end{array} \right\}$$

SplitJoin

Let the *SplitJoin* have n children and let sj_i denote the ith child of the *SplitJoin* (counting from left to right, starting with 0). Let sj_s and sj_j denote the *splitter* and the *joiner* of the *SplitJoin*, respectively. Let $w_{s,i}$ denote the number of items sent by the *splitter* to ith child on *splitter*'s every execution. Let $w_{j,i}$ denote the number of items consumed by the *joiner* from the ith child on *joiner*'s every execution. We are computing S_{sj}.

We start by calculating all $S_{sj_i}, i \in \{0, \ldots, n-1\}$.

Next we compute a fraction vector v'' and a fraction a''_j such that executing the *splitter* exactly once, each child sj_i v''_i times and the *joiner* a''_j times does not change the amount of data buffered in the *SplitJoin*. Again, since v'' and a''_j are fractions, we multiply the steady-state pop and push amounts by appropriate fractions to obtain the amount of data pushed and popped. For convenience we define a''_s to be the number of executions of the *splitter* and set it to 1.

We thus have that each child sj_i must execute $v''_i = \frac{w_{s,i}}{o_{sj_i}}$ times. To compute the number of executions of the *joiner*, a''_j, we select an arbitrary kth child ($0 \leq k < n$) and have that the *joiner* executes $a''_j = \frac{w_{s,k}}{o_{s_k}} \frac{u_{sj_k}}{w_{j,k}}$ times.

Next we compute integer vector v' and integers a_s and a_j such that executing the *splitter* a_s times, each child sj_i v'_i times and the *joiner* a_j times still does not change the amount of data buffered in the *SplitJoin*. We do this by multiplying a''_s, v'' and a''_j by $w_{j,k} \left(\prod_{r=0}^{n-1} o_{sj_r} \right)$. Thus we get

$$\begin{aligned}
a'_s = &\quad w_{j,k} \left(\prod_{r=0}^{n-1} o_{sj_r} \right) \\
v'_i = &\quad w_{j,k} \left(\prod_{r=0}^{n-1} o_{sj_r} \right) * \frac{w_{s,i}}{o_{sj_i}} = w_{s,i} * w_{j_k} \left(\prod_{r=0}^{i-1} o_{s_r} \right) \left(\prod_{r=i+1}^{n-1} o_{s_r} \right) \\
a'_j = &\quad w_{j,k} \left(\prod_{r=0}^{n-1} o_{sj_r} \right) * \frac{w_{s,k}}{o_{s_k}} \frac{u_{sj_k}}{w_{j,k}} = w_{s,k} * u_{sj_k} * \left(\prod_{r=0}^{k-1} o_{s_r} \right) \left(\prod_{r=k+1}^{n-1} o_{s_r} \right)
\end{aligned}$$

Now we use v', a'_s and a'_j to compute minimal steady state of the *SplitJoin*. Since v', a'_s and a'_j represent a steady state, they represent a strict multiple of the minimal steady state. Thus we find the multiplier by computing g, the gcd of all elements in v' and integers a'_s and a'_j, and dividing v', a'_s and a'_j by g. We have that

$$\begin{aligned}
g = &\quad \gcd(v', a'_s, a'_j) \\
v = &\quad \frac{v'}{g} \\
a_s = &\quad \frac{a'_s}{g} \\
a_j = &\quad \frac{a'_j}{g}
\end{aligned}$$

Finally, we use v, a_s and a_j to construct S_{sj}:

$$S_{sj} = \left\{ \begin{array}{c}
v_0 * S_{sj_0,m} \circ \ldots \circ v_{n-1} * S_{sj_{n-1},m} \circ [a_s \ a_j], \\
S_{sj_0,N} \circ \ldots \circ S_{sj_{n-1},N} \circ \{sj_s, sj_j\}, \\
\begin{bmatrix} n_s * o_s \\ n_s * o_s \\ n_j * u_j \end{bmatrix}, \\
v \circ [a_s] \circ [a_j]
\end{array} \right\}$$

Figure 3-1 (b) depicts a sample *SplitJoin*. The following are the steady states of the

SplitJoin's children:

$$S_A = \left\{ [1], \{A\}, \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}, [] \right\}, \quad S_B = \left\{ [1], \{B\}, \begin{bmatrix} 3 \\ 2 \\ 6 \end{bmatrix}, [] \right\}$$

For this *SplitJoin*, we select $k = 0$ (we use the left-most child to compute a'_j). We get the following v', a'_s and a'_j

$$\begin{aligned} v' &= \begin{bmatrix} 2 * 2(2) \\ 1 * 2(2) \end{bmatrix} = \begin{bmatrix} 8 \\ 4 \end{bmatrix} \\ a'_s &= 1 * 2(2 * 2) = 8 \\ a'_j &= 2 * 1(2 * 2) = 8 \end{aligned}$$

Thus $\gcd(u', a'_s, a'_j) = \gcd(8, 4, 8, 8) = 4$. Now we obtain

$$\begin{aligned} v &= \tfrac{v}{4} = \tfrac{1}{4} \begin{bmatrix} 8 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \\ a_s &= \tfrac{a'_s}{4} = \tfrac{8}{4} = 2 \\ a'_j &= \tfrac{a'_j}{4} = \tfrac{8}{4} = 2 \end{aligned}$$

Finally, we construct S_{sj}:

$$S_{sj} = \left\{ \begin{array}{c} 2 * S_{sj0,m} \circ 1 * S_{sj1,m} \circ [2\ 2], \\ S_{sj0,N} \circ S_{sj1,N} \circ \{sj_s, sj_j\}, \\ \begin{bmatrix} 2 * 3 \\ 2 * 3 \\ 2 * 4 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \\ 2 \\ 2 \end{bmatrix} \end{array} \right\}$$

It is important to note, that it is not always possible to compute a unique v'' for all possible *SplitJoins*. The reason is that unbalanced production/consumption ratios between different children of a *SplitJoin* can cause data to buffer up infinitely.

Definition 2 (Valid *SplitJoin*) *A* SplitJoin *is valid* iff $\forall k, 0 \leq k < n - 1, a''_{j,k} = a''_{j,k+1}$, *using notation of* $a''_{j,k}$ *to indicate that kth child of the* SplitJoin *was used to compute the value of* a''_j.

An example of an illegal *SplitJoin* is depicted in Figure 3-2. The rates of throughput of data for the left child mean that for every execution of the *splitter*, the *joiner* needs to

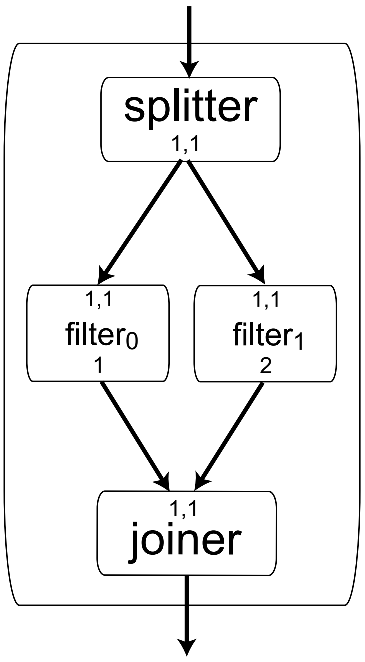


Figure 3-2: An illegal *SplitJoin*

be executed exactly once to drain all data entering the *SplitJoin*. The rates of throughput of data for the right child mean that for every execution of the *splitter*, the *joiner* needs to be executed exactly twice to drain all data entering the *SplitJoin*. That means that consumption of data by the *joiner* will be relatively slower on the right side, causing data to buffer up. This means that the given *SplitJoin* does not have a steady state.

If a *SplitJoin* is such that it does not have a steady state, it is considered an illegal *SplitJoin*. It cannot be executed repeatedly without infinite buffering, so a practical target for *StreamIt* cannot execute it. The calculations presented here assume that the *SplitJoin* is legal. In order to check if a given *SplitJoin* is legal, we test if selecting a different child for calculation of a''_j yields a different a''_j. If it does, then the two paths tested have different production/consumption rates, and the *SplitJoin* does not have a steady state.

FeedbackLoop

Let *FeedbackLoop* fl have children B (the body child) and L (the feedback loop child). Let the *joiner* and the *splitter* of the *FeedbackLoop* be denoted fl_j and fl_s. Let $w_{j,I}$ and $w_{j,L}$ denote the number of data items consumed by the *joiner* from the *input channel* to the *FeedbackLoop* and from fl_L, respectively. Let $w_{s,O}$ and $w_{s,F}$ denote the number of data items pushed by the *splitter* onto the *FeedbackLoop*'s *input channel* and to fl_L respectively. We are computing S_{fl}.

First we calculate S_B and S_L.

Now we compute a fractional vector $v'' = [a''_B \; a''_L \; a''_s \; a''_j]$ such that executing the body child a''_B times, the *splitter* a''_s times, the loop child a''_F times and the *joiner* a''_j times will not change the amount of data buffered up in the *FeedbackLoop*. Thus

$$\begin{aligned} a'_B * u_B &= a'_s * o_s \\ a'_L * u_B &= a'_j * w_{j,L} \\ a'_s * w_{s,F} &= a'_L * o_B \\ a'_j * u_j &= a'_B * o_B \end{aligned}$$

We begin with setting $a''_j = 1$. B needs to be executed $\frac{a''_B = u_j}{o_B}$ times, the *splitter* needs to be executed $a''_s = \frac{u_j}{o_B}\frac{u_B}{o_s}$ times and L needs to be executed $a''_L = \frac{u_j}{o_B}\frac{u_B}{o_s}\frac{w_{s,L}}{o_L}$ times. Furthermore, in order to assure that the *FeedbackLoop* has a valid steady state, we continue going around the loop, the *joiner* must require $\frac{u_j}{o_B}\frac{u_B}{o_s}\frac{w_{s,L}}{o_L}\frac{u_L}{w_{j,L}} = 1$. If this condition is not satisfied, the *FeedbackLoop* does not have a steady state. This is a necessary, but not a sufficient condition for a *FeedbackLoop* to be valid.

Next we compute an integer vector $v' = [a'_B \; a'_L \; a'_s \; a'_j]$ such that executing B a'_B times, *splitter* a'_s times, L a'_L times and *joiner* a'_j times will not change the amount of data buffered in the *SplitJoin*. We do this by multiplying v'' by $o_B * o_s * o_L$.

$$\begin{aligned} a'_B &= u_j * o_s * o_L \\ a'_L &= u_j * u_B * w_{s,L} \\ a_j &= o_B * o_s * o_L \\ a_s &= u_j * u_B * o_L \end{aligned}$$

We now use v' to compute $v = [a_B \; a_L \; a_s \; a_j]$, a minimal steady state for the *FeedbackLoop*. We do this by finding an integer g, the gcd of all elements in v' and computing $v = \frac{v'}{g}$.

Finally, we construct S_{fj} as follows:

$$S_{fj} = \left\{ \begin{array}{c} a_B * S_{B,m} \circ a_L * S_{L,m} \circ [a_s \; a_j], \\ S_{B,N} \circ S_{L,N} \circ \{fl_s, fl_j\}, \\ \begin{bmatrix} a_j * w_{j,I} \\ a_j * w_{j,I} \\ a_s * w_{s,O} \end{bmatrix}, v \end{array} \right\}$$

Figure 3-1(c) depicts a sample *FeedbackLoop*. The following are the steady states of the

SplitJoin's children:

$$S_B = \left\{ [1], \{B\}, \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}, [] \right\}, \quad S_L = \left\{ [1], \{L\}, \begin{bmatrix} 5 \\ 5 \\ 6 \end{bmatrix}, [] \right\}$$

We compute v' for this *FeedbackLoop*:

$$v' = \begin{bmatrix} 5*3*5 \\ 5*1*3 \\ 5*1*5 \\ 2*3*5 \end{bmatrix} = \begin{bmatrix} 75 \\ 15 \\ 25 \\ 30 \end{bmatrix}$$

Thus $g = \gcd(75, 15, 25, 30) = 5$ and

$$v = \frac{1}{5} \begin{bmatrix} 15 \\ 3 \\ 5 \\ 6 \end{bmatrix}$$

Finally, we construct S_{fl}

$$S_{fl} = \left\{ \begin{array}{c} 15 * S_{B,m} \circ 3 * S_{L,m} \circ [5\ 6], \\ S_{B,N} \circ S_{L,N} \circ \{fl_s, fl_j\}, \\ \begin{bmatrix} 6*2 \\ 6*2 \\ 5*3 \end{bmatrix}, \begin{bmatrix} 15 \\ 3 \\ 5 \\ 6 \end{bmatrix} \end{array} \right\}$$

3.3 Initialization for Peeking

Consider a *Filter* f, with peek amount of 2 and a pop amount of 1. When a *StreamIt* program is first run, there is no data present on any of the *channel*s. This means that for the first execution, filter f requires that two data items be pushed onto its *input channel*. After the first execution of f, it will have consumed one data item, and left at least one data item on its *input channel*. Thus in order to execute f for the second time, at most one extra data item needs to be pushed onto f's *input channel*. The same situation persists for all subsequent executions of f - at most one additional data item is required on f's *input channel* in order to execute f.

This example illustrates that first execution of a *Filter* may require special treatment. Namely, the source for *Filter*'s data may need to push more data onto *Filter*'s *input channel* for *Filter*'s first execution. Due to this condition, a *StreamIt* program may need to be initialized before it can enter steady state execution.

There are other constraints (latency constraints) which may require more complex initialization. These will be discussed in Chapter 6.

After an execution, a *Filter* f must leave at least $e_f - o_f$ data on its *input channel*. Thus, if the only constraints on initialization are peek-related, it is a sufficient condition for entering steady state schedule that $\forall f \in \mathit{Filters}, in_f \geq e_f - o_f$.

Specific strategies for generating initialization schedules for peeking will be presented in Chapter 4 and Chapter 5.

3.4 Schedules

Once a program has been initialized, it is ready to execute its steady state. In order to do this, a steady state schedule needs to be computed. The steady states computed above do not indicate the ordering of execution of the nodes, only how many times the nodes need to be executed.

A schedule is an ordering of nodes in a *StreamIt* streams. In order to execute the schedule, we iterate through all of its nodes in order of appearance and execute them one by one. For example in order to execute schedule $\{ABBCCBBBCC\}$ we would execute node A once, then node B, node B again, C two times, B three times and C twice again, in that order.

In order to shorten the above schedule we can run-length encode it. The schedule becomes $\{A\{2B\}\{2C\}\{3B\}\{3C\}\}$.

3.5 Schedule Size vs. Buffer Size

When creating a schedule, two very important properties of it are schedule size and amount of buffering required. Schedule size depends on encoding the schedule in an efficient way, while amount of space required depends only on order of execution of nodes. The two are related, however, because order of execution of *Filter*s affects how efficiently the schedule can be encoded.

For example, execution of *Filter*s in *Pipeline* depicted in Figure 3-3 can be ordered in two simple ways, one resulting in a large schedule but minimal amount of buffering, the other resulting in a small schedule but a large amount of buffering.

The steady schedule of the *Pipeline* in Figure 3-3 executes *Filter A* 4 times, *Filter B* 6

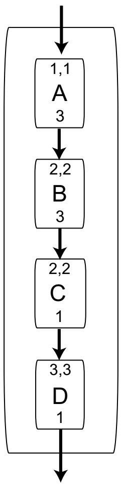


Figure 3-3: Sample 4 *Filter Pipeline*. This *Pipeline* is the same as one in Figure 3-1 (a), except that its children do not peek extra data

times, *Filter* C 9 times and *Filter* D 3 times. Writing out a schedule that requires minimal buffering results in schedule $\{AB\{2C\}BCDAB\{2C\}ABCDB\{2C\}ABCD\}$. This schedule requires a buffer for 4 data items between *Filter*s A and B, 4 items between B and C and 3 items between C and D, resulting in total buffers size 11, assuming data items in all buffers require the same amount of space. The schedule itself has 18 entries.

To compare, writing the schedule in the most compact method we get

$$\{4A\}\{6B\}\{9C\}\{3D\}$$

This schedule requires a buffer for 12 data items between *Filter*s A and B, 18 items between B and C, and 9 data items between C and D, resulting in total buffers size 39. The schedule has 4 entries.

We can compare the storage efficiency of these two schedules by assuming that one data item in a buffer requires x amount of memory and each entry in a schedule requires y amount of memory. Thus the two schedules will require the same amount of storage to store themselves and execute if $11x + 18y = 39x + 4y$.

$$\begin{aligned} 11x + 18y &= 39x + 4y \\ 14y &= 28x \\ y &= 2x \end{aligned}$$

Thus the smaller schedule is more efficient if every data item requires less than twice the amount of storage than every entry in the schedule.

One of the difficulties in scheduling *StreamIt* programs lies in finding a good set of trade-offs between schedule size and buffering requirements.

Chapter 4

Hierarchical Scheduling

In this chapter we present Hierarchical Scheduling, a technique which is quite effective for scheduling *StreamIt* programs, but which cannot schedule all programs, and which may require the buffers to be very large.

Section 4.1 provides some motivation for hierarchical scheduling. Section 4.2 presents the notation used for hierarchical notation. Section 4.3 provides an algorithm for computing hierarchical schedules.

4.1 Motivation

As has been explained in Section 3.5, the ordering of execution of nodes in a *StreamIt* program can have a significant effect on the amount of resources necessary to execute the schedule. The two important factors to consider when creating the schedule is amount of buffering necessary to execute the schedule, and the amount of space necessary to store the schedule. The amount of buffering necessary is controlled by the ordering of execution of nodes of the *StreamIt* graph. The amount of storage necessary to store the schedule is controlled by the encoding of the schedule. As a general rule, ordering which minimizes the buffering space requirements is fairly irregular and difficult to encode efficiently.

One technique used for encoding schedules is to form loop-nests of sub-schedules and repeat them multiple times, until a steady-state schedule is reached. For example, the stream in Figure 4-1 has a following steady state:

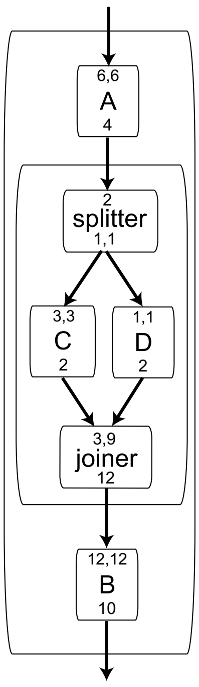


Figure 4-1: A sample stream used for hierarchical scheduling.

$$S_s = \left\{ \begin{bmatrix} 9 \\ 6 \\ 18 \\ 18 \\ 4 \\ 4 \end{bmatrix}, \begin{bmatrix} A \\ C \\ D \\ split \\ join \\ B \end{bmatrix}, \begin{bmatrix} 54 \\ 54 \\ 40 \end{bmatrix}, \begin{bmatrix} 9 \\ 2 \\ 4 \end{bmatrix} \right\}$$

Thus one steady state schedule for this stream can be

$$\{9\{A\{2split\}\{2D\}\}\}\{2\{\{3C\}\{2split\}\{2B\}\}\}$$

Here, $\{A\{2split\}\{2D\}\}$ and $\{\{3C\}\{2split\}\{2B\}\}$ are the inner nests, executed 9 and 2 times respectively.

If, the overall schedule has every *StreamIt* node appear only once (as in the example above), the technique is called Single Appearance Scheduling [7]. One of difficulties in using Single Appearance Scheduling is finding a good way to form loop-nests for the sub-schedules, because the buffering requirements can grow quite large. An example of this has been presented in Section 3.5.

StreamIt provides the scheduler with a pre-existing hierarchical structure. While it is possible to use techniques developed for Single Appearance Scheduling to create valid schedules for *StreamIt* programs, Single Appearance Scheduling does not satisfy all requirements of an effective *StreamIt* scheduler. This is because some *FeedbackLoop*s cannot be scheduled using Single Appearance Scheduling techniques. This difficulty arises because the amount of data provided to the *FeedbackLoop* by the $delay_{fl}$ variable is not sufficient to perform a complete steady-state execution of the loop, thus preventing the schedule for the *FeedbackLoop* to be encoded with only a single appearance of every node in the schedule.

The solution to this problem is to have the same node appear multiple times in the schedule. While this solves the problem of inability to schedule some *FeedbackLoop*s, it introduces another problem: which nodes should appear several times, and how many times should they be executed on each appearance. The solution proposed here goes half-way to solve the problem. A more effective solution will be proposed in Chapter 5.

In hierarchical scheduling we use the pre-existing structure (hierarchy) to determine the nodes that belong in every loop-nest. Basically, every stream receives its own loop-nest, and treats steady-state execution of its children as atomic (even if those children are streams whose executions can be broken down into more fine-grained steps). In the example above, the *Pipeline* has a *SplitJoin* child. The *SplitJoin* is responsible for scheduling its children (nodes C, B, *split* and *join*). The *Pipeline* will use the *SplitJoin*'s schedule to create its own steady state schedule. Here the *SplitJoin*'s schedule can be $T_{sj} = \{\{9split\}\{3C\}\{9D\}\{2join\}\}$, thus making the *Pipeline*'s schedule [1]

$$T_{pipe} = \{\{9A\}\{2T_{sj}\}\{4B\}\} = \{\{9A\}\{2\{\{9split\}\{3C\}\{9D\}\{2join\}\}\}\{4B\}\}$$

The problem of inability to schedule some *FeedbackLoop*s is alleviated by allowing *FeedbackLoop* to interleave the execution of its children (the body, the loop, and the *splitter* and *joiner*). This results in *FeedbackLoop* containing multiple appearances of its children. All other streams use their children's schedules in their schedules only once. This technique is called Pseudo Single Appearance Scheduling, since it results in schedules that are very similar to proper single appearance schedules. While it does not allow scheduling of all *FeedbackLoop*s (a *FeedbackLoop* may have a child which requires more data for steady state execution then made available by the $delay_{fl}$ variable) it has been found to be very effective, and only one application has been found which cannot be scheduled using this technique.

[1]Notation for this schedule is explained in next section (Section 4.2).

4.2 Notation

The notation in the above example, is very similar to that presented in Section 3.5. A number in front of a node represents that the node is meant to be executed a certain number of times. The one big difference is that $\{2T_{sj}\}$ means that the schedule for the *Pipeline* is meant to be executed twice. Since $T_{sj} = \{\{9split\}\{3C\}\{9D\}\{2join\}\}$, $\{2T_{sj}\}$ is same as $\{2\{\{9split\}\{3C\}\{9D\}\{2join\}\}\}$.

This means that to execute T_{pipe}, node A is executed 9 times, schedule T_{sj} is executed twice and node B is executed twice, in that order. To execute T_{sj}, the *splitter* is executed 9 times, node C is executed 3 times, node D 9 times and the *joiner* twice.

Thus, writing the schedule of T_{pipe} into a flat schedule (one with no loop-nests) results in schedule $\{9A\}\{9split\}\{3C\}\{9D\}\{2join\}\{9split\}\{3C\}\{9D\}\{2join\}\{4B\}$.

In other words, T_{sj} is a loop-nest, which can be executed multiple times. When storing a schedule, T_{sj} is stored only once, and every use of T_{sj} becomes the reference to the actual schedule.

A steady schedule for a stream s will be denoted by T_s, while an initialization schedule for a stream s will be denoted I_s. A *splitter* of a *SplitJoin* or a *FeedbackLoop* s will be denoted as $split_s$, while the *joiner* will be denoted as $join_s$.

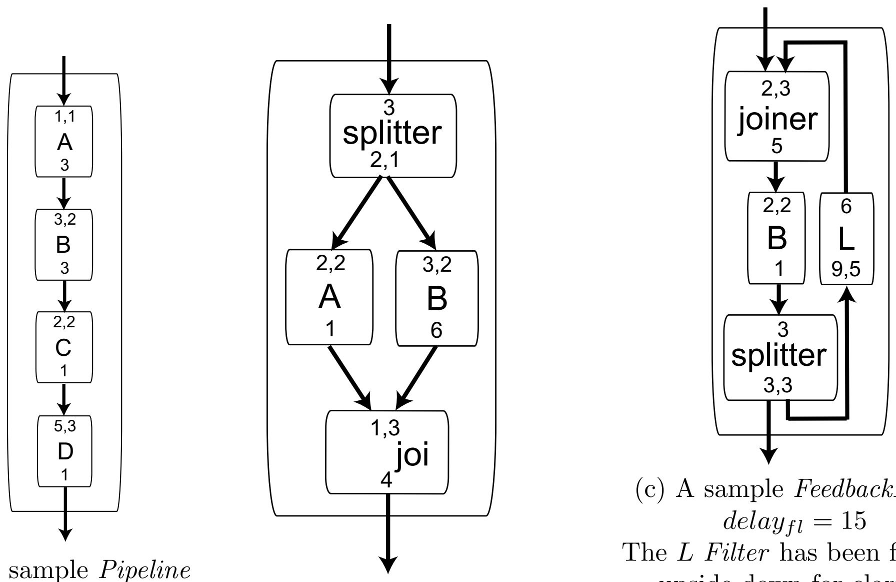
This section will continue using the notation for e, o and u extended to streams. That is, for a stream s, e_s will represent the amount of data needed by s on its *input channel* in order to execute its minimal steady state schedule; o_s represents the amount of data consumed by from its *input channel* s during execution of its steady state schedule; and u_s represents the amount of data pushed by s onto its *output channel*.

The notation for e, o and u will also be extended to initialization schedules. Namely, e^i_s represents the amount of data required by stream s on its *input channel* in order to execute the initialization schedule for s; o^i_s represents the amount of data consumed by s from its *input channel* during its initialization schedule; and u^i_s denotes the amount of data pushed by s onto its *output channel* during execution of its initialization schedule. The initialization schedules are set up in such a way, that after all streams have executed their initialization schedules, the program is ready to enter its steady state execution.

Note, that it is possible that a stream s has $u^i_s \neq 0$. An example of this will be presented in Section 4.3.4.

A hierarchical schedule for a stream s is denoted as

$$H_s = \left\{ T_s, I_s, c_s = \begin{bmatrix} e_s \\ o_s \\ u_s \end{bmatrix}, c^i_s = \begin{bmatrix} e^i_s \\ o^i_s \\ u^i_s \end{bmatrix} \right\}$$



(a) A sample *Pipeline*

(b) A sample *SplitJoin*

(c) A sample *FeedbackLoop*. $delay_{fl} = 15$ The *L Filter* has been flipped upside-down for clarity. $e_L = 9, o_L = 5, u_L = 6$

Figure 4-2: Sample *StreamIt* streams used for Pseudo Single-Appearance Hierarchical Scheduling

4.3 Pseudo Single-Appearance Hierarchical Scheduling

This section will develop hierarchical scheduling techniques to create initialization and steady state schedules. A simple implementation of the hierarchical scheduling creates a single-appearance schedule. While single-appearance scheduling is quite effective in scheduling *StreamIt* programs, it is also easy to construct programs that have *FeedbackLoop*s that are impossible to schedule. To alleviate the problem, the single-appearance scheduling was slightly modified to allow *FeedbackLoop*s to schedule programs using hierarchical push scheduling. This does not solve the problem altogether (some *FeedbackLoop*s are still impossible to schedule using this technique), but this technique is able to schedule many programs which cannot be scheduled with a simple single-appearance scheduler.

Sample streams for techniques described here are taken from Figure 4-2. The streams in Figure 4-2 are identical to those in Figure 3-1 with exception of the *FeedbackLoop*.

4.3.1 ***Filter*s**

An execution of a *Filter* is an atomic operation. Thus a steady state schedule for a *Filter* f is simply $T_f = (f)$.

A *Filter* has no internal buffering. Thus there is no need to initialize a *Filter* for its steady state. *Filter*s may, however, peek data. That means that in order to enter a steady state, sufficient amount of data must be pushed onto *Filter*'s *input channel*. Thus, for a *Filter* f, $e^i_f = e_f - o_f$.

Finally, a hierarchical schedule of a *Filter* is

$$H_f = \left\{ \{f\}, \{\}, \begin{bmatrix} e_f \\ o_f \\ u_f \end{bmatrix}, \begin{bmatrix} e_f - o_f \\ 0 \\ 0 \end{bmatrix} \right\}$$

4.3.2 *Pipeline*s

Here, for examples, we will use the sample *Pipeline* in Figure 4-2(a). The steady state schedule for this *Pipeline* is

$$S_p = \left\{ \begin{bmatrix} 4 \\ 6 \\ 9 \\ 3 \end{bmatrix}, \begin{Bmatrix} A \\ B \\ C \\ D \end{Bmatrix}, \begin{bmatrix} 4 \\ 4 \\ 3 \end{bmatrix}, \begin{bmatrix} 4 \\ 6 \\ 9 \\ 3 \end{bmatrix} \right\}$$

Initialization

Calculation of the initialization schedule starts with computing hierarchical schedules for all children of the *Pipeline*.

In order to create an initialization schedule of a *Pipeline*, all of *Pipeline*'s children's initialization schedules must be executed. Every child must execute its initialization schedule before it can execute its steady-state schedule. Some children may require some data in order to execute their initialization schedules. The upstream children provide this data to them by first executing their own initialization schedule, and then their steady-state schedule. Thus, in the final form, the execution of a *Pipeline*'s initialization schedule first executes the initialization schedule of the top-most child, then executes the steady-state schedule this child several times, then the initialization schedule of the second-from-the-top child, followed by executing this child's steady-state schedule several, etc, until the bottom-most child is reached. Since the bottom-most child does not need to provide any data *Pipeline*'s downstream children (there aren't any), the bottom-most child only executes its initialization schedule.

The initialization schedule is calculated as follows. At every stream of the *Pipeline*, the amount of data necessary to initialize all the streams below is calculated. For kth stream, that amount is denoted $init_k$. If the *Pipeline* has n children, then for the bottom-most child, p_{n-1}, that amount is $init_{n-1} = e^i_{p_{n-1}}$. The data to the kth child is provided by the $k-1$ child, during its initialization and subsequent execution of its steady state schedule.

The initialization provides $u^i_{p_{k-1}}$ data items. Thus the $k-1$ child must execute its steady state schedule $l_{k-1} = \left\lceil \frac{init_k - u^i_{p_{k-1}}}{u_{p_{k-1}}} \right\rceil$ times. The amount of data required for initialization of the *Pipeline* by the $k-1$ child is $init_{k-1} = e^i_{p_{k-1}} + l_{k-1} * o_{p_{k-1}}$.

This calculation is performed for all children of the *Pipeline*, starting at the last (bottom-most) child, and moving up. For the sample *Pipeline* in Figure 4-2(a), the values computed are:

$$\begin{aligned}
l_3 &= 0 & init_3 &= e^i_D + l_3 * o_D = 2 + 0 * 1 = 2\\
l_2 &= \left\lceil \frac{init_3 - u^i_C}{u_C} \right\rceil = \left\lceil \frac{2-0}{1} \right\rceil = 2 & init_2 &= e^i_C + l_2 * o_C = 0 + 2 * 2 = 4\\
l_1 &= \left\lceil \frac{init_2 - u^i_B}{u_B} \right\rceil = \left\lceil \frac{4-0}{3} \right\rceil = 1 & init_1 &= e^i_B + l_1 * o_B = 1 + 1 * 3 = 4\\
l_0 &= \left\lceil \frac{init_1 - u^i_A}{u_A} \right\rceil = \left\lceil \frac{4-0}{3} \right\rceil = 2 & init_0 &= e^i_A + l_0 * o_A = 0 + 2 * 1 = 2
\end{aligned}$$

Now, the initialization schedule is simply constructed by iterating over all children of the *Pipeline*, from top to bottom, and concatenating all initialization and appropriate steady state schedules. Thus $I_p = \{I_A\{2T_A\}I_B T_B I_C\{2T_C\}I_D\}$.

Finally, we need to compute the amount of data peeked, popped and pushed by the *Pipeline* during its initialization.

The amount of data popped is simply the amount of data popped by the top-most child when executing the *Pipeline*'s initialization schedule, that is the amount of data popped by the first child during its own initialization plus the amount of data popped during its steady-state execution times number of steady state executions. That is $o^i_p = o^i_{p_0} + l_0 * o_{p_0}$.

Similarly, the amount of data pushed by the *Pipeline* is simply the amount of data pushed by the bottom-most child during its initialization. Remember that the bottom-most child never executes its steady-state schedule. Thus $u^i_p = u^i_{p_{n-1}}$.

Computing the amount of data peeked by the *Pipeline* during initialization may be a little more complicated, because unlike popping and pushing, peeking is not accumulative. Luckily, we can rely on our knowledge of structure of the *StreamIt* graph to calculate the amount of data peeked by a *Pipeline*. We know that a *Pipeline* is a single-input structure. We also know that this single input will lead directly into a *StreamIt* node. There are only three possibilities for what this node will be.

- If *Pipeline*'s first node is a *Filter* f (the first child of the *Pipeline* is a *Filter* or a *Pipeline* with a *Filter* as its first node) then the extra amount of data peeked by the *Pipeline* on initialization will be $e^i_f - o^i_f$. If the first child is a *Filter*, then p_0 is f and the extra amount peeked is also $e^i_{p_0} - o^i_{p_0}$. If the first child is a *Pipeline* with a *Filter* first node, we can show by induction that this *Pipeline*'s extra peek amount will also be $e^i_{p0} - o^i_{p0}$.

- If *Pipeline*'s first node is a *splitter* (the first child of the *Pipeline* is a *SplitJoin* or a *Pipeline* with a *splitter* as its first node) then the extra amount of data peeked by the *Pipeline* on initialization will be 0, because *splitter*s never peek. Furthermore, for the same reason, the amount of extra data peeked by the first child on its initialization will also be zero, or $e^i_{p0} - o^i_{p0} = 0$.
- If *Pipeline*'s first node is a *joiner* (the first child of the *Pipeline* is a *FeedbackLoop* or a *Pipeline* with a *joiner* as its first node) then the amount of extra data peeked by the *Pipeline* on initialization will be 0, for the same reasons as above. And again $e^i_{p0} - o^i_{p0} = 0$.

Thus on initialization, the *Pipeline* will have an extra peek amount of $e^i_{p0} - o^i_{p0}$, and the total amount of data peeked by the *Pipeline* for initialization is $e^i_p = (e^i_{p0} - o^i_{p0}) + l_0 * o_{p0}$.

Steady State Schedule

The steady state state schedule is calculated as a single-appearance schedule. Calculation of a single-appearance schedule starts with computing S_p, the steady state for the *Pipeline*.

Using S_p, the steady state schedule for the *Pipeline* is constructed by concatenating multiple steady state schedules for all children of the *Pipeline*, from top to bottom. For kth child, its steady state schedule must be executed $S_{p,v,k}$ times. The steady state guarantees that every child receives sufficient amount of data to execute all its steady state schedules, and since the data is being passed from top to bottom, all children will have received all their data before executing.

The consumption and production of data for the steady state schedule is already calculated by the steady state, and is $S_{p,c}$.

Thus, for our example, $T_p = \{\{4T_A\}\{6T_B\}\{9T_C\}\{3T_D\}\}$. The final hierarchical schedule is

$$H_p = \left\{ \{\{4T_A\}\{6T_B\}\{9T_C\}\{3T_D\}\}, \{I_A\{2T_A\}I_B\{1T_A\}I_C\{2T_C\}I_D\}, \begin{bmatrix} 4 \\ 4 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix} \right\}$$

4.3.3 *SplitJoin*s

Creating a schedule for a *SplitJoin* is essentially identical to scheduling a *Pipeline*. The initialization schedule only needs to compute how many times the *splitter* needs to be executed, and construct the actual schedule. The steady state schedule is constructed by concatenating steady state schedule of *SplitJoin*'s children, the *splitter* and *joiner*.

For our example in Figure 4-2(b), the steady state is

$$S_{sj} = \left\{ \begin{bmatrix} 2 \\ 1 \\ 2 \\ 2 \end{bmatrix}, \left\{ \begin{array}{c} A \\ B \\ \textit{splitter} \\ \textit{joiner} \end{array} \right\}, \begin{bmatrix} 6 \\ 6 \\ 8 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \\ 2 \\ 2 \end{bmatrix} \right\}$$

Initialization

In order to initialize a *SplitJoin*, all its children must execute their initialization schedules. The only requirement for executing those schedules is that they have been provided with sufficient data on their *input channel*s. Since the *splitter* provides data for all the children of a *SplitJoin*, it is the only element of a *SplitJoin* that must execute its steady state schedule.

For kth child of a *SplitJoin*, the *splitter* must provide $e^i_{sj_k}$ data items. One execution of the *splitter* causes it to push $w_{s,k}$ data items toward the kth child. Thus the *splitter* must execute at least $l_k = \left\lceil \frac{e^i_{sj_k}}{w_{s,k}} \right\rceil$ times. In order to find out how many times the *splitter* needs to execute to initialize all children, l_s, we simply find the maximum l_k. Thus $l_s = \max_k (l_k)$.

In the sample *SplitJoin* from Figure 4-2(b), we get following l_ks:

$$\begin{aligned} l_0 &= \left\lceil \frac{e^i_A}{w_{s,0}} \right\rceil = \left\lceil \frac{0}{2} \right\rceil = 0 \\ l_1 &= \left\lceil \frac{e^i_B}{w_{s,1}} \right\rceil = \left\lceil \frac{1}{1} \right\rceil = 1 \end{aligned}$$

The maximum l_ks is 1, thus $l_s = 1$, the *splitter* must be executed once for initialization.

The initialization schedule is constructed by concatenating an appropriate number of executions of the *splitter* and initialization schedules of all the children. Thus in our example, $I_{sj} = \{\textit{split}\ I_A\ I_B\}$.

The consumption of an initialization schedule of a *SplitJoin* is computed as follows: $e^i_{sj} = u^i_{sj} = l_s * o_{sj_s}$ and $u^i_{sj} = 0$. The peeking and popping amounts are simply the amount of data popped by the *splitter* for every one of its executions times the number of times it is executed. The *joiner* is never executed, thus the push amount is 0.

Thus for our example, $e^i_{sj} = u^i_{sj} = 1 * 3 = 3$ and $u^i_s = 0$.

Steady State

Similarly to the algorithm for *Pipeline*, the steady state is constructed by using $S_{sj,v}$ to concatenate the executions of the *splitter*, all children of the *SplitJoin* and the *joiner* to-

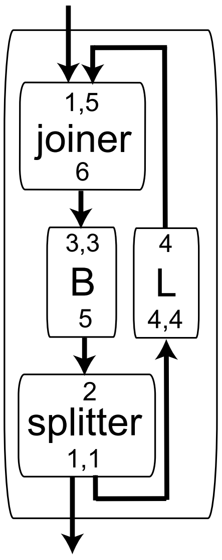


Figure 4-3: Sample *FeedbackLoop*. If this *FeedbackLoop* has a $delay_{fl}$ value set to 7, it does not have a steady state schedule which will allow it to execute forever. If the $dealy_{fl}$ value is increased by 1 to 8, the *FeedbackLoop* has a steady state schedule of $\{join\{2B\}\{5split\}L\ join\{2B\}\{5split\}L\ \ join\{2B\}\{5split\}L\{2\ join\}\{4B\}\{10split\}\{2L\}\ \}$.

gether.

For our example, the steady state schedule is simply

$$T_{sj} = \{\{2\ split\}\{2T_A\}T_B\{2\ join\}\}$$

The consumption vector, c_{sj} is the same as $S_{sj,c}$.

Thus the hierarchical schedule for the *SplitJoin* in Figure 4-2(b) is

$$H_{sj} = \left\{ \{\{2\ split\}\{2T_A\}T_B\{2\ join\}\}, \{split\ I_A\ I_B\}, \begin{bmatrix} 6 \\ 6 \\ 8 \end{bmatrix}, \begin{bmatrix} 3 \\ 3 \\ 0 \end{bmatrix} \right\}$$

4.3.4 ***FeedbackLoop*s**

Scheduling of *FeedbackLoop*s is a task that can be made difficult, if the amount of data provided for the *FeedbackLoop* by the $delay_{fl}$ value is low. Before a *StreamIt* program begins executing, the *FeedbackLoop* needs to be provided with some data in one of the internal *channel*s. Without this data, the *splitter* and the *joiner* of the *FeedbackLoop* will not be able to execute, because they will never have sufficient data on their input *channel*s. This is a consequence of the *FeedbackLoop* having a cyclical structure.

The difficulty in scheduling *FeedbackLoop*s is that if the amount of data made available to the *FeedbackLoop* by the $delay_{fl}$ value (as explained in Section 2.1.4) is small, there will be very limited number of ways to execute the *FeedbackLoop*. In fact, it is possible that the amount of data available to the *FeedbackLoop* is so small, it cannot reach and complete an execution of a steady state schedule. An example of such *FeedbackLoop* is presented in Figure 4-3.

Here we will use *FeedbackLoop* from Figure 4-2(c). The steady state schedule for this *FeedbackLoop* is

$$S_{fl} = \left\{ \begin{bmatrix} 15 \\ 3 \\ 5 \\ 6 \end{bmatrix}, \left\{ \begin{matrix} B \\ L \\ \mathit{splitter} \\ \mathit{joiner} \end{matrix} \right\}, \begin{bmatrix} 12 \\ 12 \\ 15 \end{bmatrix}, \begin{bmatrix} 15 \\ 3 \\ 5 \\ 6 \end{bmatrix} \right\}$$

Initialization Schedule

Initialization for the *FeedbackLoop* is calculated in a similar way to initialization of a *Pipeline*. The number of steady state executions of the children of the *FeedbackLoop* is denoted l_B and l_L for the body child and the loop child, respectively. The number of executions of the *splitter* is denoted l_s and the *joiner* is denoted l_j.

Since the initial data is inserted into the buffer between the loop child and the *joiner* (as explained in 2.1.4), it follows that the loop child should initialize last - it will be the last one receive data to initialize. Since the computation of the initialization schedule is similar to the way it was done for *Pipeline*, we will start with the child which is executed last, namely the loop child. Similarly as with *Pipeline*, the which is initialized last does not execute its steady state schedule for initialization, thus we set $l_L = 0$. The *splitter* must provide the loop child with just enough data to initialize, the body child must provide the *splitter* with just enough data for the *splitter* to pass enough data to the loop child, etc. Thus,

$$\begin{aligned} l_L &= 0 \\ l_s &= \left\lceil \frac{o^i_{fl_L}}{w_{s,1}} \right\rceil \\ l_B &= \left\lceil \frac{o_s * l_s - u^i_{fl_B}}{u_{fl_B}} \right\rceil \\ l_j &= \left\lceil \frac{o^i_{fl_B} + l_B * o_{fl_B}}{u_j} \right\rceil \end{aligned}$$

This initialization schedule will only be valid if there is enough data provided between the loop child and the *joiner*, or $delay_{fl} \geq l_j * w_{j,1}$. If this condition does not hold, the *FeedbackLoop* cannot be scheduled using pseudo single-appearance algorithm.

Referring to the example Figure 4-2(c), we obtain the following values for ns:

$$\begin{aligned}
l_s &= \left\lceil \tfrac{4}{3} \right\rceil = 2 \\
l_B &= \left\lceil \tfrac{2*3-0}{1} \right\rceil = 6 \\
l_j &= \left\lceil \tfrac{0+6*2}{5} \right\rceil = 3
\end{aligned}$$

Furthermore, since $delay_{fl} = 15$, we have $15 \geq 3 * 3$, thus a valid initialization schedule can be constructed.

The initialization schedule is constructed by concatenating executions of the *joiner*, body child, *splitter* and the loop child. The body child will execute both its initialization schedule as well as its steady state schedule, while the loop child will only execute its initialization schedule.

Thus for our example we get $I_{fl} = \{\{3\ join\} I_B \{6 T_B\} \{2\ split\} I_L\}$.

We now compute the consumption of data for the initialization schedule of the *FeedbackLoop*: $e^i_{fl} = o^i_{fl} = n_j * w_{j,0}$ and $u^i_{fl} = n_s * w_{s,0}$. Similarly as in computation for the *SplitJoin*, these values are simply the production and consumption of the *splitter* and *joiner* from their appropriate *input* and *output* channels multiplied by the number of times the *splitter* and *joiner* are executed during initialization schedule.

In our example, $e^i_{fl} = o^i_{fl} = 3 * 2 = 6$ and $u^i_{fl} = 3 * 3 = 9$. Note that the *FeedbackLoop* pushes data out during its initialization.

Finally, we compute the amount of data present in *channel*s after initialization. These amounts are important because they will be used to compute the steady state schedule of the *FeedbackLoop*. These amounts were not necessary for computation of steady state schedules of *Pipeline* and *SplitJoin*. These amounts are calculated by simply subtracting the amount of data popped from a *channel* from amount of data pushed into a *channel*. Here we adopted the notation for *input* and *output channel* from Section 3.1.

$$\begin{aligned}
in^i_B =&\ \ l_j * u_j - l_B * o_{fl_B} \\
out^i_B =&\ \ u^i_{fl_B} + l_B * u_{fl_B} - l_s * o_s \\
in^i_L =&\ \ l_s * w_{s,1} - l_L * o_{fl_L} \\
out^i_L =&\ \ delay_{fl} + u^i_{fl_L} + l_L * u_{fl_L} - l_j * w_{j,1}
\end{aligned}$$

Steady State Schedule

Computing the steady state schedule for a *FeedbackLoop* is more complicated than for the other streams. The reason for this is (as explained above) that *FeedbackLoop*s may require a non single-appearance schedule, while other *StreamIt* construct can always be scheduled

data items in buffer				executions left				element consid-ered	executions
in_B	out_B	in_L	out_L	*split*	B	*join*	L		
1	6	6	0	5	9	4	3	*split*	2
1	0	12	0	3	9	4	3	*L*	1
1	0	7	6	3	9	4	2	*join*	2
11	0	7	0	3	9	2	2	*B*	5
1	5	7	0	3	4	2	2	*split*	1
1	2	10	0	2	4	2	2	*L*	1
1	2	5	6	2	4	2	1	*join*	2
11	2	5	0	2	4	2	1	*B*	4
3	6	5	0	2	0	2	1	*split*	2
3	0	11	0	0	0	0	1	*L*	1
3	0	6	6	0	0	0	0		

Table 4.1: Trace of execution of steady-state algorithm on sample *FeedbackLoop* from Figure 4-2(c). The executions left amount is the number of executions left for a particular child to complete a steady state execution of the *FeedbackLoop*. One this value reaches 0, the element is not executed anymore, even if it has data to execute.

using single-appearance schedules.

The algorithm used for creating of a steady state schedule will work in several phases. The amount of data present in *channel*s between the children of the *FeedbackLoop*, the *joiner* and the *splitter* is kept track of to determine which element is allowed to execute.

The algorithm for creating a steady state schedule of a *FeedbackLoop* iterates over the elements of the *FeedbackLoop* in order of (*joiner*, body child, *splitter*, loop child). The algorithm executes each element as many times as possible, considering the amount of data required to execute the element and the amount of data available internally within the *FeedbackLoop*'s channels to execute the element and not exceeding the steady state number of executions for the element. Each execution of an element is appended to the steady state schedule.

This iteration is repeated until either all elements have executed their steady state number of times, or until a complete iteration has been performed with no element being able to execute. The first case indicates a successful completion of the algorithm. The second case indicates a failure - the algorithm is unable to schedule the *FeedbackLoop*.

Table 4.1 illustrates the execution of this algorithm for our sample *FeedbackLoop*. Notice that in the table, the first row and the last row have the same amount of data buffered in *channel*s, thus indicating that a full steady state schedule has indeed been computed. Furthermore, the last entry considering execution of B has sufficient data to execute B 5 times, but only executes it 4 times to ensure that a steady state schedule is constructed.

The schedule resulting from the above computation is

$$T_{fl} = \{\{2\ \mathit{join}\}\{6T_B\}\{2\ \mathit{split}\}T_L\{2\ \mathit{join}\}\{5T_B\}\ \mathit{split}\ T_L\{2\ \mathit{join}\}\{4T_B\}\{2\ \mathit{split}\}T_L\}$$

This schedule is obtained by going through Table 4.1 from top to bottom and concatenating the appropriate number of executions of every child of the *FeedbackLoop*, as listed in the "executions" column.

The steady state consumption c_{fl} is again simply $S_{fl,c}$. Thus the hierarchical schedule is:

$$H_{fl} = \left\{ \begin{array}{c} \{\{2\ \mathit{join}\}\{6T_B\}\{2\ \mathit{split}\}T_L\{2\ \mathit{join}\}\{5T_B\}, \\ \mathit{split}\ T_L\{2\ \mathit{join}\}\{4T_B\}\{2\ \mathit{split}\}T_L\}, \\ \{\{3\ \mathit{join}\}I_B\{6T_B\}\{2\ \mathit{split}\}I_L\}, \begin{bmatrix} 12 \\ 12 \\ 15 \end{bmatrix}, \begin{bmatrix} 6 \\ 6 \\ 9 \end{bmatrix} \end{array} \right\}$$

Chapter 5

Phased Scheduling

We then propose Phased Scheduling, a technique which allows to schedule all non-messaging *StreamIt* programs, and which allows for better control of trade-off between schedule size and buffer size.

Section 5.1 provides an introduction to and explanation of Phased Scheduling. Section 5.2 presents a Minimal Latency Schedule implementation using Phased Scheduling.

5.1 Phased Scheduling

The pseudo single-appearance hierarchical scheduling technique presented in Chapter 4, while quite effective in scheduling simple applications, cannot schedule a small number of tight *FeedbackLoop*s. Furthermore, the technique is quite inflexible when it comes to attempting to create a different tradeoff between schedule size and buffer size. The schedules created using single appearance hierarchical scheduling tend to be quite small at the expense of larger buffering requirements. A quite simple situation when such tradeoff is not desired, could be if the schedule is being stored in a large cheap ROM device, while the RAM used for buffering data is more expensive. It is also quite possible that latency constraints cannot be satisfied by a single appearance hierarchical schedule. Clearly, a more flexible technique is required for scheduling.

A key observation in hierarchical scheduling is that each component only needs to worry about the data that enters or leaves its children. The amount of buffering done internally in a child is not noticeable or important to the parent component. This observation changes slightly if latency constraints are placed on the computation. Namely, the important information to keep track of is amount of data that leaves or enters children as well as amount of data that crosses latency constraint boundaries.

This observation leads to a conclusion that scheduling execution of the *StreamIt* programs using hierarchical scheduling can be simpler than scheduling the entire program all at once (scheduling the program all at once requires tracking all buffers and latency constraints at once). Phased scheduling is a concept that expends on hierarchical scheduling, but does not require that a stream has a single or pseudo single appearance schedule. Each stream is allowed to have multiple sub-schedules, also called phases. Each phase consists of phases of the children of the stream that will be executed to execute the phase. The phases must be executed in correct order. When all of the initialization phases of a stream have been executed, the stream has executed its initialization schedule and is ready to enter steady state execution. When all of the steady state phases of a stream have been executed in order, the entire steady state schedule for the stream has been executed.

The granularity of splitting the steady state schedule into phases is left up to the specific scheduler. Different streams can use different granularities of execution. In principle, the parent should not need to know the scheduling granularity of its children. The only exception to this rule are *FeedbackLoop*s, which can have children which are not scheduled tightly enough to allow the *FeedbackLoop* to execute. An example of that may be a pseudo single-appearance hierarchical scheduling algorithm described in Section 4.3 implemented using phase scheduling.

One important observation to make is that it makes little sense to have phases which do not consume or produce any data, and which do not have data cross any latency boundaries. This is because such phases can easily be merged with preceding or following phases without any effect on ability to schedule a particular program. This observation allows to easily bound the size of the resulting schedules to be the sum of executions of first child, last child and children with latency boundaries. For example, the *Pipeline* in Figure 4-2 executes its first child, *Filter* A, 4 times in steady state execution, and its last child, *Filter* D, 9 times. Thus a phasing schedule of this pipeline should at most have $4 + 9 = 13$ phases.

5.2 Minimal Latency Phased Scheduling

One of the problems with pseudo single-appearance scheduling is that it cannot schedule all legal *StreamIt* programs. A program with a *FeedbackLoop* can have requirements for tight execution that cannot be satisfied using a pseudo single-appearance schedule, leading to deadlock. Phasing scheduling can alleviate this problem by allowing the program to be scheduled in a more fine-grained manner. Minimal latency scheduling is an example of a specific scheduling strategy that solves the problem of deadlock. Foregoing latency constraints, minimal latency scheduling can schedule any legal *StreamIt* program.

Minimal latency schedule is a schedule that requires a minimal amount of input data in order to output data. In other words, a minimal latency schedule only buffers as much data as is absolutely necessary.

A minimal latency schedule is not necessarily single appearance. In fact, very few applications can have their minimal latency schedules expressed as a single appearance schedule. One of the consequences of this is that minimal latency schedules require more space for storage of the schedule. Use of phasing scheduling facilitates creation of acceptably small minimal latency schedules. In spirit of hierarchical scheduling, every component is scheduled separately, in hierarchical order.

One important consequence of phased scheduling, one that is highlighted when calculating a minimal latency schedule, is that every phase is allowed to consume a different amount of data and produce a different amount of data.

Below is the description of algorithms used for each type of *StreamIt* component. The remainder of this section, $push_s^m$ will denote amount of data pushed by the mth phase of stream s, pop_s^m will denote amount of data popped by the mth phase of stream s and $peek_s^m$ will denote amount of data peeked by mth phase of stream s.

5.2.1 Peeking

Phased scheduling has interesting consequence for peeking calculations. The reason for this is that not all phases must consume data, thus not all phases will peek. The amount of peeking done by a stream is important for creating an initialization schedule. It is thus important to remember that the amount of peeking done by a stream is not necessarily the amount of peeking done by that stream in its first phase, because on first phase, the stream may not consume or peek any data.

5.2.2 Notation

A phasing schedule of a stream s is a set P_s of elements, $P_s = \{T_s, I_s, c_s, c_s^i\}$. The first element, T_s denotes the phases used for the steady state schedule of s. I_s denotes the phases used for the initialization schedule of s. c_s and c_s^i are defined identically to their definitions in hierarchical schedules: c_s is the consumption rate of the stream during its steady state execution and c_s^i is the consumption rate of the initialization schedule.

T_s and I_s are defined by identical structures. Both are defined as sets of phases. The only real difference between T_s and I_s is that T_s will be executed indefinitely, while I_s will be executed only once. A phase A is defined as $A = \{E, c\}$. E is an ordered list of phases and nodes that are to be executed in order to execute the phase. c is the consumption of the phase, with respect to its stream.

As an example, here is a minimal latency schedule for the sample stream in Figure 4-1. First, the schedule for the internal *SplitJoin*:

$$P_{sj} = \left\{ \begin{array}{c} T_{sj} = \left\{ \begin{array}{c} A_{sj,0} = \left\{ \{\{6\ split\}\{2C\}\{5D\}\ join\}, \begin{bmatrix} 12 \\ 12 \\ 12 \end{bmatrix} \right\}, \\ A_{sj,1} = \left\{ \{\{3\ split\}C\{4D\}\ join\}, \begin{bmatrix} 6 \\ 6 \\ 12 \end{bmatrix} \right\} \end{array} \right\}, \\ I_{sj} = \{\}, \\ c_{sj} = \begin{bmatrix} 18 \\ 18 \\ 24 \end{bmatrix}, c^i_{sj} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{array} \right\}$$

And the following is a schedule for the *Pipeline*:

$$P_p = \left\{ \begin{array}{c} T_p = \left\{ \begin{array}{c} A_{p,0} = \left\{ \{\{3A\}A_{sj,0}\ B\}, \begin{bmatrix} 18 \\ 18 \\ 10 \end{bmatrix} \right\}, \\ A_{p,1} = \left\{ \{\{2A\}A_{sj,1}\ B\}, \begin{bmatrix} 12 \\ 12 \\ 10 \end{bmatrix} \right\}, \\ A_{p,2} = \left\{ \{\{3A\}A_{sj,0}\ B\}, \begin{bmatrix} 18 \\ 18 \\ 10 \end{bmatrix} \right\}, \\ A_{p,3} = \left\{ \{A\ A_{sj,1}\ B\}, \begin{bmatrix} 6 \\ 6 \\ 10 \end{bmatrix} \right\} \end{array} \right\}, \\ I_p = \{\}, \\ c_p = \begin{bmatrix} 54 \\ 54 \\ 40 \end{bmatrix}, c^i_p = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{array} \right\}$$

5.2.3 General Concept

Technique used for calculating minimal latency phasing schedule for a *Pipeline*, *SplitJoin* and *FeedbackLoop* is similar to the technique used to create a pseudo single-appearance hierarchical schedule for a *FeedbackLoop*. Every phase is computed separately. Every phase knows how much data has been left in internal buffers by the previous phase. The goal is to create a phase that consumes the minimum amount of data from the *input channel* in order to push at least one data item out to the *output channel*. Once the minimum amount of data has been consumed by the stream, the maximum amount of data possible is pushed out of the stream without consuming any more data. This is meant to prevent unnecessary buffering of data internally within streams, and reduce the number of phases necessary to compute a complete schedule.

One important technique used for creating phased schedules is borrowing of data from *channel*s. When a child is being executed, it is allowed to borrow some data from the *channel*, and expect that the upstream child will provide the right amount of data in the *channel* for real execution. In the tables, this means that amount of data can fall below 0. This is obviously illegal during real execution for any *channel*. Some *channel*s, however, have even stricter restrictions. If the node reading from a *channel* peeks more than it pops, the amount of data in the *channel* during real execution cannot fall below the $peek - pop$ amount. For this reason, we also need to keep track of amount of data needed from a *channel*. This is illustrated in Tables 5.1, 5.2 and 5.3. At the end of a phase, all *channel*s must have all items borrowed from them returned by the upstream child.

The initialization schedule starts with no internally buffered data (with exception of *FeedbackLoop*s) and executes as many phases as is necessary to ensure that all children have executed all of their initialization phases. Once that has been achieved, the steady state schedule is created. The only difference between computation of an initialization and steady state schedules is that the steady state schedule stops executing children early, if they have already executed all the phases allocated to them for the steady state, while the initialization schedule continues executing until all initialization phases of all children have been executed.

The only significant difference between the algorithms used for minimal latency scheduling of different stream types (*Pipeline*, *SplitJoin* and *FeedbackLoop*) is the order with which children of the stream are considered for execution.

For an ith child of a stream s (stream s_n), the number of phases that must be executed for its steady state schedule to be complete is $S_{s,v,i} * |P_{s_i,T}|$.

5.2.4 *Filter*

Since *Filter*s have no internal buffering and only one *work* function, their schedules are simple. They contain a single phase, which in turn contains a single execution of the filter's *work* function. Although in principle, a *Filter* does not need to be executed to be initialized, it may require some data to be buffered for its execution. This means that if $e_f > o_f$, we insert an artificial initialization phase to phasing schedules of *Filter*s:

$$P_p = \left\{ \begin{array}{c} T_p = \left\{ A_{f,0} = \left\{ \{f\}, \begin{bmatrix} e_f \\ o_f \\ u_f \end{bmatrix} \right\} \right\}, \\ I_p = \left\{ A^i_{f,0} = \left\{ \{\}, \begin{bmatrix} e_f - o_f \\ 0 \\ 0 \end{bmatrix} \right\} \right\}, \\ c_p = \begin{bmatrix} e_f \\ o_f \\ u_f \end{bmatrix}, c^i_p = \begin{bmatrix} e_f - o_f \\ 0 \\ 0 \end{bmatrix} \end{array} \right\}$$

5.2.5 *Pipeline*

The order with which children are considered for execution is as follows. First all the children are considered for execution moving from bottom to top. The last child executes just enough phases to produce some data. The child directly above it executes just enough phases to provide sufficient data for the child below to execute its child. This process is repeated until the top-most child is reached. At this point the direction of traversal is reversed. This time, the top-most child is skipped, and the second top-most child is considered. It only executes as many phases as it can, while only using data already buffered between it and the child above it. Then, the child below it is executed in the same way. This is repeated until the bottom-most child is reached. The number of phases executed by each child is added up, and the phases are inserted in order (all phases of every child together, in order, iterating from top-most child down to bottom-most child). This constitutes one complete phase of the *Pipeline*.

Using the sample *Pipeline* from Figure 4-2(a), the following are phasing schedules for *Filter*s A, B, C and D:

$$P_A = \left\{ \begin{array}{c} T_A = \left\{ A_{A,0} = \left\{ \{A\}, \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix} \right\} \right\}, \\ I_A = \{\}, c_A = \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix}, c^i_B = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{array} \right\}$$

$$P_B = \left\{ \begin{array}{c} T_B = \left\{ A_{B,0} = \left\{ \{B\}, \begin{bmatrix} 3 \\ 2 \\ 3 \end{bmatrix} \right\} \right\}, \\ I_B = \left\{ A^i_{B,0} = \left\{ \{\}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right\} \right\}, \\ c_B = \begin{bmatrix} 3 \\ 2 \\ 3 \end{bmatrix}, c^i_B = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \end{array} \right\}$$

$$P_C = \left\{ \begin{array}{c} T_C = \left\{ A_{C,0} = \left\{ \{C\}, \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix} \right\} \right\}, \\ I_C = \{\}, c_C = \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}, c^i_C = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{array} \right\}$$

$$P_D = \left\{ \begin{array}{c} T_D = \left\{ A_{D,0} = \left\{ \{D\}, \begin{bmatrix} 5 \\ 3 \\ 1 \end{bmatrix} \right\} \right\}, \\ I_D = \left\{ A^i_{D,0} = \left\{ \{\}, \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \right\} \right\}, \\ c_D = \begin{bmatrix} 5 \\ 3 \\ 1 \end{bmatrix}, c^i_D = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \end{array} \right\}$$

Table 5.1 shows a trace of execution of the algorithm on the *Pipeline* from Figure 4-2(a).

The following is the resulting phasing schedule:

data in *channel*			phase executions left				child consid-ered	phases executed	*Pipeline* consumption
in_B	in_C	in_D	A	B	C	D			
0 (0)	0 (0)	0 (-2)	0	1	0	0	C	$\{2A_{C,0}\}$	[0 0 0]
0 (0)	-4 (-4)	2 (0)	0	1	0	0	B	$A^i_{B,0}, \{2A_{B,0}\}$	[0 0 0]
-4 (-5)	2 (0)	2 (0)	0	0	0	0	A	$\{2A_{A,0}\}$	[2 2 0]
2 (0)	2 (0)	2 (0)	0	0	0	0	B	-	[0 0 0]
2 (0)	2 (0)	2 (0)	0	0	0	0	C	$A_{C,0}$	[0 0 0]
2 (0)	0 (0)	3 (0)	0	0	0	0	D	-	[0 0 0]
2 (0)	0 (0)	3 (0)	init phase 0 done, init done						
2 (0)	0 (0)	3 (0)	4	6	9	3	D	$A_{D,0}$	[0 0 1]
2 (0)	0 (0)	0 (-2)	4	6	9	2	C	$\{2A_{C,0}\}$	[0 0 0]
2 (0)	-4 (-4)	2 (0)	4	6	7	2	B	$\{2A_{B,0}\}$	[0 0 0]
-2 (-3)	2 (0)	2 (0)	4	4	7	2	A	$A_{A,0}$	[1 1 0]
1 (0)	2 (0)	2 (0)	3	4	7	2	B	-	[0 0 0]
1 (0)	2 (0)	2 (0)	3	4	7	2	C	$A_{C,0}$	[0 0 0]
1 (0)	0 (0)	3 (0)	3	4	6	2	D	-	[0 0 0]
1 (0)	0 (0)	3 (0)	phase 0 done						
1 (0)	0 (0)	3 (0)	3	4	6	2	D	$A_{D,0}$	[0 0 1]
1 (0)	0 (0)	0 (-2)	3	4	6	1	C	$\{2A_{C,0}\}$	[0 0 0]
1 (0)	-4 (-4)	2 (0)	3	4	4	1	B	$\{2A_{B,0}\}$	[0 0 0]
-3 (-4)	2 (0)	2 (0)	3	2	4	1	A	$\{2A_{A,0}\}$	[2 2 0]
3 (0)	2 (0)	2 (0)	1	2	4	1	B	$A_{B,0}$	[0 0 0]
1 (0)	5 (0)	2 (0)	1	1	4	1	C	$\{2A_{C,0}\}$	[0 0 0]
1 (0)	1 (0)	4 (0)	1	1	2	1	D	-	[0 0 0]
1 (0)	1 (0)	4 (0)	phase 1 done						
1 (0)	1 (0)	4 (0)	1	1	2	1	D	$A_{D,0}$	[0 0 1]
1 (0)	1 (0)	1 (-1)	1	1	2	0	C	$A_{C,0}$	[0 0 0]
1 (0)	-1 (-1)	2 (0)	1	1	1	0	B	$A_{B,0}$	[0 0 0]
-1 (-2)	2 (0)	2 (0)	1	1	1	0	A	$A_{A,0}$	[1 1 0]
2 (0)	2 (0)	2 (0)	0	0	1	0	B	-	[0 0 0]
2 (0)	2 (0)	2 (0)	0	0	1	0	C	$A_{C,0}$	[0 0 0]
2 (0)	0 (0)	3 (0)	0	0	0	0	D	-	[0 0 0]
2 (0)	0 (0)	3 (0)	phase 2 done, steady state schedule done						

Table 5.1: Trace of execution of Minimal Latency Scheduling Algorithm on *Pipeline* from Figure 4-2(a). In the "data in *channel*" columns the first value represents the actual number of data in the *channel*, which can be negative if more data has been popped from the *channel* than has been pushed into it. This is due to borrowing of data from *channel*s. The second value represents the minimal number of data items that the downstream *Filter* has inspected beyond the 0th data. This value can be higher than the negative amount of data in the *channel* because a *Filter* may peek at data without consuming it. In general, for a *Filter* f, the amount of data needed on its input *channel* is $\max(0, -(in_f - (e_f - o_f)))$. The needed amount is 0 until the downstream *Filter* is executed for the first time.

$$P_p = \left\{ \begin{array}{c} T_p = \left\{ \begin{array}{c} A_{p,0} = \left\{ \{A\{2B\}\{3C\}D\}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}, \\ A_{p,1} = \left\{ \{\{2A\}\{3B\}\{4C\}D\}, \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix} \right\}, \\ A_{p,2} = \left\{ \{A\ B\{2C\}D\}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\} \end{array} \right\}, \\ I_p = \left\{ A^i_{p,0} = \left\{ \{\{2A\}A^i_{B,0}\{2B\}\{3C\}A^i_{D,0}\}, \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix} \right\} \right\}, \\ c_p = \begin{bmatrix} 4 \\ 4 \\ 3 \end{bmatrix}, c^i_p = \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix} \end{array} \right\}$$

5.2.6 *SplitJoin*

As explained above, the only difference between the algorithm for a *Pipeline* and a *SplitJoin* is the order in which the children streams are considered for execution. In a *Pipeline*, the children are considered from the bottom-most child to the top child, and then from second top-most child down to the bottom most child again. A *SplitJoin* has only three levels of direct children in it: the top is a *splitter*, the middle is formed by all the child streams of the *SplitJoin* and the bottom is the *joiner*. To schedule a *SplitJoin*, the children are also considered in the bottom to top and top to bottom order, but the child streams are also considered from left to right (this choice is arbitrary - the order does not affect the number of child phase executions per phase of the *SplitJoin*).

Using the sample *SplitJoin* from Figure 4-2(b), the following are phasing schedules for *Filter*s A and B:

$$P_A = \left\{ \begin{array}{c} T_A = \left\{ A_{A,0} = \left\{ \{A\}, \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix} \right\} \right\}, \\ I_A = \{\}, c_A = \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}, c^i_A = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{array} \right\}$$

data in *channel*				phase executions left				child considered	phases executed	*Pipeline* consumption
split	A	B	join	in_A	out_A	in_B	out_B			
0 (0)	0 (0)	0 (0)	0 (0)	0	0	1	0	join	-	[0 0 0]
0 (0)	0 (0)	0 (0)	0 (0)	0	0	1	0	A	-	[0 0 0]
0 (0)	0 (0)	0 (0)	0 (0)	0	0	1	0	B	$A^i_{B,0}$	[0 0 0]
0 (0)	0 (0)	0 (-1)	0 (0)	0	0	0	0	split	split	[3 3 0]
2 (0)	0 (0)	1 (0)	0 (0)	0	0	0	0	A	$A^i_{A,0}$	[0 0 0]
0 (0)	1 (0)	1 (0)	0 (0)	0	0	0	0	B	-	[0 0 0]
0 (0)	1 (0)	1 (0)	0 (0)	0	0	0	0	join	-	[0 0 0]
0 (0)	1 (0)	1 (0)	0 (0)	init phase 0 done, init done						
0 (0)	1 (0)	1 (0)	0 (0)	2	2	1	2	join	join	[0 0 4]
0 (0)	0 (0)	1 (0)	-3 (-3)	2	2	1	2	A	-	[0 0 0]
0 (0)	0 (0)	1 (0)	-3 (-3)	2	2	1	1	B	$A_{B,0}$	[0 0 0]
0 (0)	0 (0)	-1 (-2)	3 (0)	2	2	0	2	split	$\{2split\}$	[6 6 0]
4 (0)	0 (0)	1 (0)	3 (0)	0	2	0	2	A	$\{2A_{A,0}\}$	[0 0 0]
0 (0)	2 (0)	1 (0)	3 (0)	0	0	0	0	B	-	[0 0 0]
0 (0)	2 (0)	1 (0)	3 (0)	0	0	0	1	join	join	[0 0 4]
0 (0)	1 (0)	1 (0)	0 (0)	phase 0 done, steady state schedule done						

Table 5.2: Execution of Minimal Latency Scheduling Algorithm on *SplitJoin* from Figure 4-2(b).

$$P_B = \left\{ \begin{array}{c} T_B = \left\{ A_{B,0} = \left\{ \{B\}, \begin{bmatrix} 3 \\ 2 \\ 6 \end{bmatrix} \right\} \right\}, \\ I_B = \left\{ A^i_{B,0} = \left\{ \{\}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right\} \right\}, \\ c_B = \begin{bmatrix} 3 \\ 2 \\ 6 \end{bmatrix}, c^i_B = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \end{array} \right\}$$

Table 5.2 shows execution of the algorithm on the *SplitJoin* from Figure 4-2(b).

The trace of the execution shows that even though it is strictly necessary to traverse the children of the stream second time from bottom to top, doing so can pay off in reducing the number of phases necessary to construct a phasing schedule. Namely, in its first steady state execution, the *splitter* needs to push enough data to execute the *joiner* again, thus eliminating a need for an additional phase.

Once all the phases are computed, the phasing schedule is constructed. For every phase, the number of child phases executed is added up, and the actual schedule is constructed by concatenating all the phases of all the children, starting with the *splitter*, all stream children (as listed from left to right) and finally the *joiner*. The following is the resulting phasing schedule:

$$P_{sj} = \left\{ \begin{array}{c} T_{sj} = \left\{ A_{sj,0} = \left\{ \{\{2split\}\{2A\}B\{2join\}\}, \begin{bmatrix} 6 \\ 6 \\ 8 \end{bmatrix} \right\} \right\}, \\ I_{sj} = \left\{ A^i_{sj,0} = \left\{ \{split\ A^i_{A,0}\ A^i_{B,0}\}, \begin{bmatrix} 3 \\ 3 \\ 0 \end{bmatrix} \right\} \right\}, \\ c_{sj} = \begin{bmatrix} 6 \\ 6 \\ 8 \end{bmatrix}, c^i_{sj} = \begin{bmatrix} 3 \\ 3 \\ 0 \end{bmatrix}, \end{array} \right\}$$

5.2.7 *FeedbackLoop*

Scheduling of *FeedbackLoop*s is again similar to the above algorithms. The children's phases are executed in order of (*splitter*, body child, *joiner*, body child, *splitter*, loop child). The *splitter* tries to execute exactly one time on its first iteration. The body child and the *joiner* execute just enough times to provide data for the *splitter* to perform its first execution. Then the body child, *splitter* and the loop child are executed as many times as possible with the data available to them on their *input channel*s.

The one big difference between *FeedbackLoop* and the other streams (*Pipeline* and *SplitJoin*) is that in scheduling a *FeedbackLoop*, the *joiner* is *n*ot allowed to borrow elements from out_L *channel*. That is in the trace table, the out_L entry is never allowed to become negative. The reason for this is that *FeedbackLoop*s are cyclical structures, and allowing the *joiner* to borrow elements from out_L would cause a full cycle of borrowing, leading to deadlock.

This one condition does not prevent from scheduling any legal *FeedbackLoop*s. The reason for this is that before the *FeedbackLoop* is initialized, there is data pushed onto the out_L *channel*. At the end of scheduling of any phase, all available data is pushed through the *FeedbackLoop* into the out_L *channel*. Thus any available free data is already always stored in the out_L *channel*, and there is no additional data to borrow from in a *FeedbackLoop*.

If the algorithm is unable to schedule an execution of the *joiner* in a phase without borrowing data from out_L *channel*, then the *FeedbackLoop* cannot be scheduled.

Lemma 2 (*FeedbackLoop* Scheduling) *If all children of a* FeedbackLoop *are scheduled using minimal latency scheduling algorithm, then if the* FeedbackLoop *cannot be scheduled using the minimal latency scheduling algorithm then there is no valid schedule for this* FeedbackLoop*.*

We believe this lemma to be true because minimal latency scheduling always consumes the minimal amount of data to produce some data, and produces the maximal amount of data possible given the amount of data it consumes. Thus no data is being buffered up in *channel*s and if the *FeedbackLoop* cannot be scheduled, then the $delay_{fl}$ value is too low and does not provide enough data to complete a steady state execution. A formal proof is left for future work.

We will again use the sample *FeedbackLoop* from Figure 4-2(c). The following are the phasing schedules for *Filter*s B and L:

$$P_B = \left\{ \begin{array}{c} T_B = \left\{ A_{B,0} = \left\{ \{B\}, \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix} \right\} \right\}, \\ I_B = \{\}, \\ c_B = \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}, c^i_B = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{array} \right\}$$

$$P_L = \left\{ \begin{array}{c} T_L = \left\{ A_{L,0} = \left\{ \{L\}, \begin{bmatrix} 9 \\ 5 \\ 6 \end{bmatrix} \right\} \right\}, \\ I_L = \left\{ A^i_{L,0} = \left\{ \{\}, \begin{bmatrix} 4 \\ 0 \\ 0 \end{bmatrix} \right\} \right\}, \\ c_L = \begin{bmatrix} 9 \\ 5 \\ 6 \end{bmatrix}, c^i_L = \begin{bmatrix} 4 \\ 0 \\ 0 \end{bmatrix} \end{array} \right\}$$

Table 5.3 shows execution of the algorithm on the *FeedbackLoop* from Figure 4-2(c).

Once the number of executions of children's phases is known for every phase of the *FeedbackLoop*'s schedule, the phasing schedule can be constructed. For every phase, the children of the *FeedbackLoop* are iterated over in order of (*joiner*, body child, *splitter*, loop child) and for every child the appropriate number of phases is inserted into the schedule. Below is the schedule for *FeedbackLoop* in Figure 4-2(c):

data in *channel*				phase executions left				child considered	phases executed	*Pipeline* consumption
in_B	out_B	in_F	out_F	join	B	split	F			
0 (0)	0 (0)	0 (0)	15 (0)	0	0	0	1	split	split	[0 0 3]
0 (0)	-3 (3)	3 (0)	15 (0)	0	0	0	1	B	$\{3A_{B,0}\}$	[0 0 0]
-6 (6)	0 (0)	3 (0)	15 (0)	0	0	0	1	join	{2 *join*}	[4 4 0]
4 (0)	0 (0)	3 (0)	9 (0)	0	0	0	1	B	$\{2A_{B,0}\}$	[0 0 0]
0 (0)	2 (0)	3 (0)	9 (0)	0	0	0	1	split	-	[0 0 0]
0 (0)	2 (0)	3 (0)	9 (0)	0	0	0	1	F	-	[0 0 0]
0 (0)	2 (0)	3 (0)	9 (0)	init phase 0 done						
0 (0)	2 (0)	3 (0)	9 (0)	0	0	0	1	split	split	[0 0 3]
0 (0)	-1 (1)	6 (0)	9 (0)	0	0	0	1	B	$\{A_{B,0}\}$	[0 0 0]
-2 (2)	0 (0)	6 (0)	9 (0)	0	0	0	1	join	join	[2 2 0]
3 (0)	0 (0)	6 (0)	6 (0)	0	0	0	1	B	$\{A_{B,0}\}$	[0 0 0]
1 (0)	1 (0)	6 (0)	6 (0)	0	0	0	1	split	-	[0 0 0]
1 (0)	1 (0)	6 (0)	6 (0)	0	0	0	1	F	$\{A^{i}_{F,0}\}$	[0 0 0]
1 (0)	1 (0)	6 (0)	6 (0)	init phase 1 done, init done						
1 (0)	1 (0)	6 (0)	6 (0)	6	15	5	3	split	split	[0 0 3]
1 (0)	-2 (2)	9 (0)	6 (0)	6	15	4	3	B	$\{3A_{B,0}\}$	[0 0 0]
-5 (5)	1 (0)	9 (0)	6 (0)	6	12	4	3	join	join	[2 2 0]
0 (0)	1 (0)	9 (0)	3 (0)	5	12	4	3	B	-	[0 0 0]
0 (0)	1 (0)	9 (0)	3 (0)	5	12	4	3	split	-	[0 0 0]
0 (0)	1 (0)	4 (0)	9 (0)	5	12	4	3	F	$\{A_{F,0}\}$	[0 0 0]
0 (0)	1 (0)	4 (0)	9 (0)	phase 0 done						
0 (0)	1 (0)	4 (0)	9 (0)	5	12	4	2	split	split	[0 0 3]
0 (0)	-2 (2)	7 (0)	9 (0)	5	12	3	2	B	$\{2A_{B,0}\}$	[0 0 0]
-4 (4)	0 (0)	7 (0)	9 (0)	5	10	3	2	join	join	[2 2 0]
1 (0)	0 (0)	7 (0)	6 (0)	4	10	3	2	B	-	[0 0 0]
1 (0)	0 (0)	7 (0)	6 (0)	4	10	3	2	split	-	[0 0 0]
1 (0)	0 (0)	7 (0)	6 (0)	4	10	3	2	F	-	[0 0 0]
1 (0)	0 (0)	7 (0)	6 (0)	phase 1 done						
1 (0)	0 (0)	7 (0)	6 (0)	4	10	3	2	split	split	[0 0 3]
1 (0)	-3 (3)	10 (0)	6 (0)	4	10	2	2	B	$\{3A_{B,0}\}$	[0 0 0]
-5 (5)	0 (0)	10 (0)	6 (0)	4	7	2	2	join	join	[2 2 0]
0 (0)	0 (0)	10 (0)	3 (0)	3	7	2	2	B	-	[0 0 0]
0 (0)	0 (0)	10 (0)	3 (0)	3	7	2	2	split	-	[0 0 0]
0 (0)	0 (0)	10 (0)	3 (0)	3	7	2	2	F	$\{A_{F,0}\}$	[0 0 0]
0 (0)	0 (0)	5 (0)	9 (0)	phase 2 done						
0 (0)	0 (0)	5 (0)	9 (0)	3	7	2	1	split	split	[0 0 3]
0 (0)	-3 (3)	8 (0)	9 (0)	3	7	1	1	B	$\{3A_{B,0}\}$	[0 0 0]
-6 (6)	0 (0)	8 (0)	9 (0)	3	4	1	1	join	{2 *join*}	[4 4 0]
4 (0)	0 (0)	8 (0)	3 (0)	1	4	1	1	B	$\{2A_{B,0}\}$	[0 0 0]
0 (0)	2 (0)	8 (0)	3 (0)	1	2	1	1	split	-	[0 0 0]
0 (0)	2 (0)	8 (0)	3 (0)	1	2	1	1	F	-	[0 0 0]
0 (0)	2 (0)	8 (0)	3 (0)	phase 3 done						
0 (0)	2 (0)	8 (0)	3 (0)	1	2	1	1	split	split	[0 0 3]
0 (0)	-1 (1)	11 (0)	3 (0)	1	2	0	1	B	$\{A_{B,0}\}$	[0 0 0]
-2 (2)	0 (0)	11 (0)	3 (0)	1	1	0	1	join	join	[2 2 0]
3 (0)	0 (0)	11 (0)	0 (0)	0	1	0	1	B	$\{A_{B,0}\}$	[0 0 0]
1 (0)	1 (0)	11 (0)	0 (0)	0	0	0	1	split	-	[0 0 0]
1 (0)	1 (0)	11 (0)	0 (0)	0	0	0	1	F	$\{A_{F,0}\}$	[0 0 0]
1 (0)	1 (0)	6 (0)	6 (0)	phase 4 done, steady state schedule done						

Table 5.3: Execution of Minimal Latency Scheduling Algorithm on *FeedbackLoop* from Figure 4-2(c).

$$
P_{fl} = \left\{ \begin{array}{c}
T_{fl} = \left\{ \begin{array}{c}
A_{fl,0} = \left\{ \{join\ \{3B\}\ split\ F\}, \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix} \right\}, \\
A_{fl,1} = \left\{ \{join\ \{2B\}\ split\}, \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix} \right\}, \\
A_{fl,2} = \left\{ \{join\ \{3B\}\ split\ F\}, \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix} \right\}, \\
A_{fl,3} = \left\{ \{\{2\ join\}\{5B\}\ split\}, \begin{bmatrix} 4 \\ 4 \\ 3 \end{bmatrix} \right\}, \\
A_{fl,4} = \left\{ \{join\ \{2B\}\ split\ F\}, \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix} \right\}
\end{array} \right\}, \\
T_{fl} = \left\{ \begin{array}{c}
A^{i}_{fl,0} = \left\{ \{\{2\ join\}\{5B\}\ split\}, \begin{bmatrix} 4 \\ 4 \\ 3 \end{bmatrix} \right\}, \\
A^{i}_{fl,1} = \left\{ \{join\ \{2B\}\ split\}, \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix} \right\}
\end{array} \right\}, \\
c_{fl} = \begin{bmatrix} 12 \\ 12 \\ 15 \end{bmatrix}, c^{i}_{fl} = \begin{bmatrix} 6 \\ 6 \\ 6 \end{bmatrix},
\end{array} \right\}
$$

Chapter 6

Latency Constrained Scheduling

In the previous chapters, an efficient framework for scheduling *StreamIt* programs was developed. In this framework, there was no concern for possible latency constraints specified by the programmer. This chapter will develop techniques which can be used to schedule *StreamIt* programs that contain various latency constraints. These techniques will be based on linear and integer programming. This chapter is mainly concerned with producing schedules that are correct and respect all specifications of the program. Optimization of such schedules is left for future work.

Section 6.1 provides a detailed introduction to the timing semantics of message sending in *StreamIt*. Section 6.2 provides an example of a simple stream with messages being sent. Section 6.3 introduces the $info$ function which is used to keep track of information buffered between *Filter*s. Section 6.4 shows how the $info$ function relates to messaging constraints. Section 6.5 presents an algorithm for computing schedules that respect messaging constraints. Finally Section 6.6 solves the example from Section 6.2 using the algorithm presented here.

6.1 Messages

In the *StreamIt* computation model, *Filter*s do not share any variables or memory directly. There is no concept of global program time. The purpose of this is to allow different *Filter*s to execute on different devices without need for complicated data sharing techniques and synchronization. Furthermore, this model allows to schedule the execution of *Filter*s in any order that does not violate the execution semantics (*Filter*s always have at least e_f data in their *input channel* before executing).

Due to the structured property of *StreamIt* constructs, flow of data between *Filter*s is very limited. This model works well for regular steady-state execution of streaming programs, because *StreamIt* programs match typical streaming algorithms well. There are, however, some situations where information needs to be communicated only very occasionally and between *Filter*s that cannot easily send data to one another within *StreamIt* structure. In order to enable more flexible communication between *Filter*s in such situations, *StreamIt* provides a concept of message sending.

Messages can be sent between any two *Filter*s which are connected in the directed *StreamIt* graph. In other words, any two *Filter*s which are not located in different branches of the same *SplitJoin* can send messages to one another.

A more formal definition of all messaging concepts can be found in [24].

6.1.1 Timing

The timing of delivery of messages is expressed in terms of latency and information wavefronts. On every execution, a *Filter* consumes and produces some data. The data it consumes affects the data it outputs. Thus the data output on a given iteration carries the same information wavefront as the data it consumed. Using this definition it is possible to find out on which execution *Filter* B receives data that *Filter* A produced on a given execution. In other words, it is possible to find on which execution *Filter* B observes the same information wavefront that *Filter* A observed on a given iteration. Note that if a *Filter* consumes data that carries multiple information wavefronts, the data produced carries the latest information wavefront of all the data consumed.

In order to specify timing of delivery of a message sent by *Filter* A to *Filter* B, we state that the message will be delivered with latency $[l_0, l_1]$. This means that *Filter* B will receive the message no earlier then just before it observes the information wavefront observed by *Filter* A on l_0 execution of *Filter* A's *work* function after sending the message, and no later then just before it observes the information wavefront observed by *Filter* B on l_1 execution of *Filter* A's *work* function after sending the message. In other words, l_0 is the lower bound on message delivery latency, while l_1 is the upper bound on message delivery latency. Obviously, $l_0 \leq l_1$.

In *StreamIt* it is possible to send messages both upstream and downstream. Sending a message downstream allows both latency bounds to be either positive or negative. Sending a message with a positive latency bound is intuitive: the message will be delivered to the receiver when after it observes the information wavefront processed when sending the message. However, sending a message with negative latency bound means that the message is to be delivered *b*efore the receiver sees the information wavefront processed by the sender when sending the message. One way of thinking about this is that the message is being sent back in time, in terms of information wavefronts. Note that it is possible for the lower latency bound to be negative and for the upper latency bound to be positive.

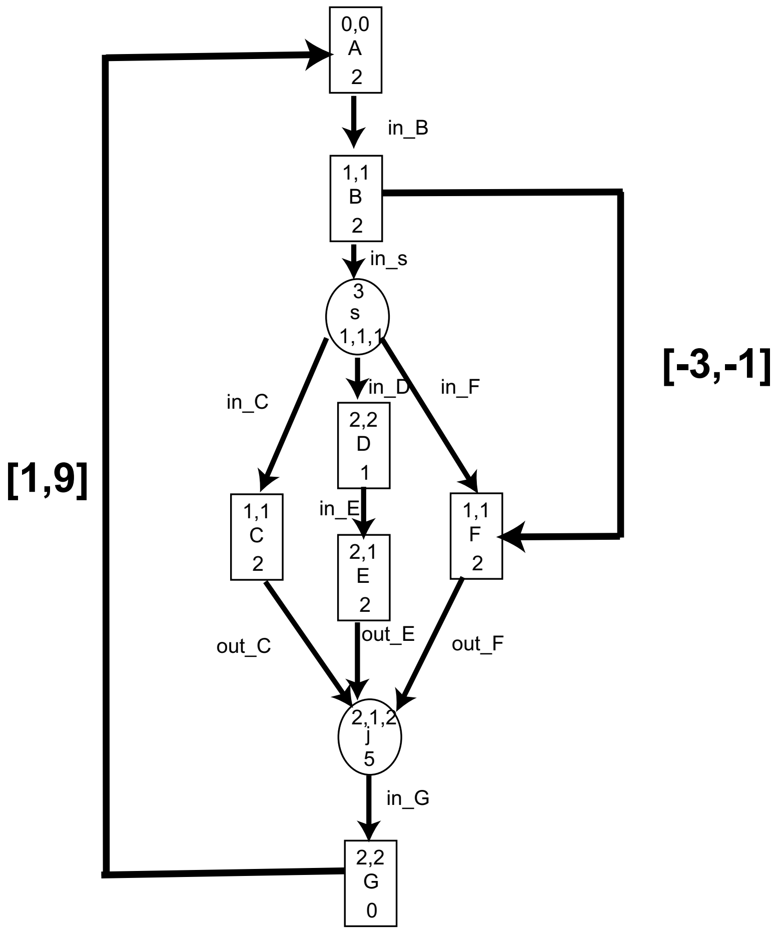


Figure 6-1: Example *StreamIt* program for latency constrained analysis

When a message is being sent upstream, both upper and lower bound on latency of delivery of the message must be positive ($l_0 \geq 1$). This is because when the sender sends the message, the receiver must have already observed sender's current information wavefront.

It is important to note that not all latency bounds are valid. It is not always obvious whether specified latency bounds are valid, as there may be many reasons for latency bounds to be invalid. Those reasons include too tight buffering constraints and contradictory latency bounds.

6.2 Example

Figure 6-1 depicts a sample *Pipeline* which contains message communication. In the example, *Filter* B can send a message to *Filter* F with latency $[-3, -1]$ and *Filter* G can send a message to *Filter* A with latency $[1, 9]$. Note that *Filter* B sends a message that crosses *StreamIt* structure boundary (F's direct parent is not the same as B's direct parent). *Filter* G sends a message that travels upstream.

The following is the steady state for the *Pipeline*:

$$S_p = \left\{ \begin{bmatrix} 3 \\ 6 \\ 4 \\ 4 \\ 2 \\ 2 \\ 4 \\ 4 \\ 10 \end{bmatrix}, \left\{ \begin{matrix} A \\ B \\ \mathit{split} \\ C \\ D \\ E \\ F \\ \mathit{join} \\ G \end{matrix} \right\}, \begin{bmatrix} 6 \\ 6 \\ 10 \end{bmatrix}, \begin{bmatrix} 3 \\ 6 \\ 2 \\ 10 \end{bmatrix} \right\}$$

Below we list a schedule which allows to execute the sample *Pipeline* while respecting the messaging constraints imposed. In the schedule, entry A^m indicates that *Filter* A checks if any messages have been sent to it, and if there are messages waiting to be delivered to A, it receives them. Note that no message sending is allowed during initialization of the *Pipeline*. This is the case for all programs, and will be explained later.

The initialization schedule for the sample *Pipeline* is $\{\{2A\}\{3B\}\{2\mathit{split}\}D\}$. The steady state schedule for the sample *Pipeline* is

$$\{F^mFBA^m\{2A\}F^mF\{2B\}\{2C\}\{2\ \mathit{split}\}DE\{2\ \mathit{join}\}$$
$$\{5G\}F^mFBA^mAF^mF\{2B\}\{2C\}\{2\ \mathit{split}\}DE\{2\ \mathit{join}\}\{5G\}\}$$

An inspection of the schedule above reveals that because of fairly tight constraint on sending messages from *Filter* B to *Filter* F, their execution is interleaved pretty tightly. Latency constraint between *Filter* G and *Filter* A is not as tight, and the interleaving of execution of those *Filter*s is not as fine grained.

The schedules listed above are not unique, and haven't been optimized for any particular criteria.

Table 6.1 depicts the flow of information wavefronts produced by *Filter* B between *Filter* B and *Filter* F when executing the schedule listed above. According to the schedule given above, *Filter* F checks for messages before its every execution. The latency of messages from B to F is given as $[-3, -1]$ in Figure 6-1. Thus before F checks for messages, it is must be the case that B has produced the information wavefront F will see on its next execution. It also must be that B has not produced the information wavefront F will see in four executions. It is easy to see that this condition is respected. We simply inspect all executions of B, and make sure that on every one of its executions, the wavefront generated by B on its previous execution has not passed F yet. As stated above, we skip initialization, as messages cannot be delivered during initialization.

On B's first steady state execution it produces wavefront 3. Its previous execution produced wavefront 2. F has only observed wavefront 1, so message sent by B can be delivered. On B's second steady state execution it produces wavefront 4. Its previous execution produced wavefront 3. F has only observed observed wavefront 2, so message sent by B can be delivered. This analysis continues until an entire steady state has been completed.

Table 6.2 depicting the flow of information wavefronts produced by *Filter* A across the sample *Pipeline* when executing the schedule listed above. We can use this table to verify that messages sent from *Filter* G to *Filter* A will be delivered within the specified latency.

Messages sent from *Filter* G to *Filter* A have to be delivered with latency $[1, 9]$. This means that a message sent by *Filter* G must be delivered to *Filter* A before it produces information wavefront that G will see in 9 executions. The lower bound does not impose any real constraint, because the message cannot be delivered before A has produced an information wavefront that G sees when it sends the message. Since the information wavefronts flow downstream, but the message must be delivered upstream, we must do the verification in terms of the receiver's wavefronts.

On G's first execution it observes A's wavefront 2. Nine executions later it sees wavefront 4. *Filter* A has only produced wavefront 3, so any message sent can be delivered on time. On G's second execution it again observes A's wavefront 2. Nine executions later it observes wavefront 5. *Filter* A has again only produced wavefront 3, so any message sent can also be delivered on time. This analysis can be performed for all executions of G within a single steady state to verify that all possible messages sent by G to A can be delivered on time.

6.3 Information Buffering Model

In order to satisfy possible latency constraints, a global model of accounting for data buffered up needs to be constructed. The model must express the flow of information carried by data.

6.3.1 Intuition

We begin with creating a concept of abstract information. Every data item carries a certain amount of information. Every data item in a particular *channel* carries the same amount of information. A *Filter* consumes some data items from its *input channel* and pushes them to its *output channel*. We define execution of *Filter*s to be an information-preserving operation. This means that the amount of information consumed by a *Filter* during its execution must be pushed out onto *Filter*'s *output channel*. This means that the amount

element executed	B's info wavefronts in buffer	
	in_s	in_F
2 A	-	-
3 B	-	-
2 split	0,0,1,1,2,2	-
1 D	-	1,2
init done	-	1,2
1 F	-	1,2
1 B	-	2
2 A	3,3	2
1 F	3,3	2
2 B	3,3	-
2 C	3,3,4,4,5,5	-
2 split	-	4,5
1 D	-	4,5
1 E	-	4,5
2 join	-	4,5
5 G	-	4,5
1 F	-	4,5
1 B	-	5
1 A	6,6	5
1 F	6,6	5
2 B	6,6	-
2 C	6,6,7,7,8,8	-
2 split	6,6,7,7,8,8	-
1 D	-	7,8
1 E	-	7,8
2 join	-	7,8
5 G	-	7,8
run 1 done	-	7,8
1 F	-	7,8
1 B	-	8
2 A	9,9	8
1 F	9,9	8
2 B	9,9	-
2 C	9,9,10,10,11,11	-
2 split	-	10,11
1 D	-	10,11
1 E	-	10,11
2 join	-	10,11
5 G	-	10,11
1 F	-	10,11
1 B	-	11
1 A	12,12	11
1 F	12,12	11
2 B	12,12	-
2 C	12,12,13,13,14,14	-
2 split	12,12,13,13,14,14	-
1 D	-	13,14
1 E	-	13,14
2 join	-	13,14
5 G	-	13,14
run 2 done	-	13,14

Table 6.1: Flow of information wavefronts between *Filter*s B and F during execution of schedule provided in Section 6.2. Left column provides the node to be executed. The center and right columns show information wavefronts carried by data in *channel*s in_s and in_F. Every number corresponds to a single data item in a buffer. Each number corresponds to the information wavefront carried by the data item. Information wavefronts are counted in terms of executions of *Filter* B's *work* function, starting with 0. Each line shows information wavefronts present in *channel*s before the node is executed. Thus entry 0,0,1,1,2,2 means that there are six data items present in the *channel*, first two carry information wavefront of first execution of B's *work* function, second two carry information wavefront of second execution of B's *work* function, etc.

element executed	A's info wavefronts in buffer									
	in_B	in_s	in_C	in_D	in_E	in_F	out_C	out_E	out_F	in_G
2 A	-	-	-	-	-	-	-	-	-	-
3 B	0,0,1,1	-	-	-	-	-	-	-	-	-
2 split	1	0,0,0,0,1,1	-	-	-	-	-	-	-	-
1 D	1	-	0,1	0,1	-	0,1	-	-	-	-
init done	1	-	0,1	-	1	0,1	-	-	-	-
1 F	1	-	0,1	-	1	0,1	-	-	-	-
1 B	1	-	0,1	-	1	1	-	-	0,0	-
2 A	-	1,1	0,1	-	1	1	-	-	0,0	-
1 F	2,2,3,3	1,1	0,1	-	1	1	-	-	0,0	-
2 B	2,2,3,3	1,1	0,1	-	1	-	-	-	0,0,1,1	-
2 C	3,3	1,1,2,2,2,2	0,1	-	1	-	-	-	0,0,1,1	-
2 split	3,3	1,1,2,2,2,2	-	-	1	-	0,0,1,1	-	0,0,1,1	-
1 D	3,3	-	2,2	2,2	1	2,2	0,0,1,1	-	0,0,1,1	-
1 E	3,3	-	2,2	-	1,2	2,2	0,0,1,1	-	0,0,1,1	-
2 join	3,3	-	2,2	-	2	2,2	0,0,1,1	2,2	0,0,1,1	-
5 G	3,3	-	2,2	-	2	2,2	-	-	-	2,2,2,2,2, 2,2,2,2,2
1 F	3,3	-	2,2	-	2	2,2	-	-	-	-
1 B	3,3	-	2,2	-	2	2	-	-	2,2	-
1 A	3	3,3	2,2	-	2	2	-	-	2,2	-
1 F	3,4,4	3,3	2,2	-	2	2	-	-	2,2	-
2 B	3,4,4	3,3	2,2	-	2	-	-	-	2,2,2,2	-
2 C	4	3,3,3,3,4,4	2,2	-	2	-	-	-	2,2,2,2	-
2 split	4	3,3,3,3,4,4	-	-	2	-	2,2,2,2	-	2,2,2,2	-
1 D	4	-	3,4	3,4	2	3,4	2,2,2,2	-	2,2,2,2	-
1 E	4	-	3,4	-	2,4	3,4	2,2,2,2	-	2,2,2,2	-
2 join	4	-	3,4	-	4	3,4	2,2,2,2	4,4	2,2,2,2	-
5 G	4	-	3,4	-	4	3,4	-	-	-	4,4,4,4,4, 4,4,4,4,4
1st steady state done	4	-	3,4	-	4	3,4	-	-	-	-
1 F	4	-	3,4	-	4	3,4	-	-	-	-
1 B	4	-	3,4	-	4	4	-	-	3,3	-
2 A	-	4,4	3,4	-	4	4	-	-	3,3	-
1 F	5,5,6,6	4,4	3,4	-	4	4	-	-	3,3	-
2 B	5,5,6,6	4,4	3,4	-	4	-	-	-	3,3,4,4	-
2 C	6,6	4,4,5,5,5,5	3,4	-	4	-	-	-	3,3,4,4	-
2 split	6,6	4,4,5,5,5,5	-	-	4	-	3,3,4,4	-	3,3,4,4	-
1 D	6,6	-	5,5	5,5	4	5,5	3,3,4,4	-	3,3,4,4	-
1 E	6,6	-	5,5	-	4,5	5,5	3,3,4,4	-	3,3,4,4	-
2 join	6,6	-	5,5	-	5	5,5	3,3,4,4	5,5	3,3,4,4	-
5 G	6,6	-	5,5	-	5	5,5	-	-	-	5,5,5,5,5, 5,5,5,5,5
1 F	6,6	-	5,5	-	5	5,5	-	-	-	-
1 B	6,6	-	5,5	-	5	5	-	-	5,5	-
1 A	6	6,6	5,5	-	5	5	-	-	5,5	-
1 F	6,7,7	6,6	5,5	-	5	5	-	-	5,5	-
2 B	6,7,7	6,6	5,5	-	5	-	-	-	5,5,5,5	-
2 C	7	6,6,6,6,7,7	5,5	-	5	-	-	-	5,5,5,5	-
2 split	7	6,6,6,6,7,7	-	-	5	-	5,5,5,5	-	5,5,5,5	-
1 D	7	-	6,7	6,7	5	6,7	5,5,5,5	-	5,5,5,5	-
1 E	7	-	6,7	-	5,7	6,7	5,5,5,5	-	5,5,5,5	-
2 join	7	-	6,7	-	7	6,7	5,5,5,5	7,7	5,5,5,5	-
5 G	7	-	6,7	-	7	6,7	-	-	-	7,7,7,7,7, 7,7,7,7,7
2nd steady state done	7	-	6,7	-	7	6,7	-	-	-	-

Table 6.2: Flow of information wavefronts between *Filter*s A and G. The representation is same as in Table 6.1, except wavefronts are given in terms of *Filter* A's *work* function executions.

of information carried by every data item in the *input channel* may differ from the amount of information carried by every data item in the *output channel*.

A *splitter* is defined to consume some information from its *input channel* and provide that amount of information to each of its *output channel*s. That is, a *splitter* increases the amount of information in the system by the number of *output channel*s the *splitter* has multiplied by amount of information it consumes on every execution. Notice that this action has the effect of increasing the amount of information carried by each data item for *RoundRobin splitter*s, but does not change the amount of information carried by each data item across *Duplicate splitter*s. Also, a *splitter* with only a single *output channel* behaves exactly the same as a *Filter*.

A *joiner* behaves in the exact opposite way to the *splitter*. *joiner*s are defined to consume the same amount of information from every one of their *input channel*s, and produce the same amount of information on their *output channel*. As a result, *joiner*s decrease the amount of information carried by every data item passing through them.

The behavior of *splitter*s and *joiner*s is a little counter-intuitive. Intuition dictates that since *splitter*s and *joiner*s do not inspect or modify the data in any way, they should not change the amount of information carried by data passing through them. This approach leads to inconsistency with *SplitJoin*s, however. Consider the *SplitJoin* from Figure 6-1. Assume that every data item in *channel* in_s carries 1 unit of information. By this alternative method, every data item in *channel*s in_C, in_D and in_F also carries 1 unit of information. *Filter* C consumes one data item, thus one unit of information, and produces 2 data items. Thus in *channel* out_C, every data item carries $\frac{1}{2}$ units of information. *Filter* D consumes 2 data items, thus consumes 2 units of information and produces one data item, thus every data item in *channel* in_E carries 2 units of information. *Filter* E consumes 1 data item, thus consumes 2 units of information, and produces 2 data items, thus data items in *channel* out_E carry 1 unit of information each.

We now have an inconsistency. The *joiner* consumes data from *channel* out_C carrying $\frac{1}{2}$ unit of information per data and data from *channel* out_E carrying 1 unit of information per data. Since all data on the *output channel* of the *joiner* must carry the same amount of information per data item, the *joiner* must change the amount of information per data item.

Our approach guarantees that every node produces and consumes the same amount of information from every one of its *input* and *output channel*s. This definition is consistent across all *Filter*s, *splitter*s and *joiner*s.

6.3.2 Information Function

We define an information function, that describes amount of information carried by each data item in a particular *channel*. For a channel c, the function is denoted as $info_c$. The value of the $info$ function must be consistent across the entire *StreamIt* graph, and every *channel* needs an actual numerical value.

We begin computing a set of consistent values for $info$ functions of different *channel*s by selecting a *channel* c to have an $info$ function of 1. We now travel the program graph upstream and downstream, across graph nodes, computing the $info$ function for all other *channel*s connected to the node being crossed.

There are three types of nodes that can be crossed, and they can be crossed from either *input* or *output* sides.

*Filter*s

A *Filter* f has only one *input* and one *output channel*. If we know $info_{in_f}$, we can easily compute $info_{out_f}$, because we know that on every execution of f, we consume the same amount of information we produce: $info_{out_f} = info_{in_f} \frac{o_f}{u_f}$. Similarly, if we know $info_{out_f}$, we compute $info_{in_f}$ as follows: $info_{in_f} = info_{out_f} \frac{u_f}{o_f}$.

*splitter*s

We have defined *splitter*s to provide their every *output channel* with the same amount of information as they consume from their *input channel*. Thus for a *splitter* s we know that $info_{out_{s,i}} = info_{in_s} \frac{o_s}{w_{s,i}}$. Similarly, $info_{in_s} = info_{out_{s,i}} \frac{w_{s,i}}{o_s}$. With these two equations we can easily compute values for $info$ of all of s's *channel*s, given the $info$ function for any of s's *channel*s.

*joiner*s

We have defined *joiner*s to consume the same amount of information from every one of their *input channel*s and push that same amount of information to their *output channel*. Thus for a *joiner* j we know that $info_{out_j} = info_{out_{j,i}} \frac{w_{j,i}}{u_j}$. Similarly, $info_{in_{j,i}} = info_{out_j} \frac{u_j}{w_{j,i}}$. With these two equations we can easily compute values for $info$ of all of j's *channel*s, given the $info$ function for any of j's *channel*s.

Example

Table 6.3 lists the values of $info$ function for *channel*s in Figure 6-1. The computation of above $info$ values began with assigning $info_{in_B} = 1$.

buffer	$info$ function	information per data item
in_B	$info_{in_B}$	1
in_s	$info_{in_s}$	0.5
in_C	$info_{in_C}$	1.5
in_D	$info_{in_D}$	1.5
in_E	$info_{in_E}$	3
in_F	$info_{in_F}$	1.5
out_C	$info_{out_C}$	0.75
out_E	$info_{out_E}$	1.5
out_F	$info_{out_F}$	0.75
in_G	$info_{in_G}$	0.3

Table 6.3: Information per data item in buffers

6.4 Latency Constraints and Information

The next step is to decide how the latency constraints correspond to the amount of information consumed, produced or buffered in the application.

6.4.1 Checking for Messages

One of the most important issues to solve is to find out how often an intended recipient of a message needs to check for messages. This frequency can easily be calculated in terms of number of executions of the *work* function. The assumption being made here is that the message is delivered to the destination as soon as it is generated. Such a model can be easily achieved on a single processor machine. Different models may require different calculations.

Filter f_s sends a message to *Filter* f_r with latency $[l_0, l_1]$. On every execution of its *work* function, *Filter* f_s processes $x = o_{f_s} * info_{in_{f_s}}$ information. Similarly, *Filter* f_r processes $y = o_{f_r} * info_{in_{f_r}}$ information on every execution of its *work* function.

Latency $[l_0, l_1]$ means that the receiver must check for messages from the sender every time it processes as much information, as the sender will send over $l_1 - l_0$ executions. The sender will send $x*(l_1-l_0)$ information over that many executions. The receiver will process that much information over $\frac{x*(l_1-l_0)}{y}$ executions of its *work* function.

This value may be fractional, but execution of *Filter*s is an atomic operation in *StreamIt*. Thus the receiver must actually check for new messages from the sender at least every $\left\lfloor \frac{x*(l_1-l_0)}{y} \right\rfloor$ executions of its *work* function.

Note that calculation assumes a fairly dumb message delivery method, where the latency of the message is not taken into account. More sophisticated models of message delivery can allow to reduce the frequency of checking for new messages significantly.

6.4.2 Information Buffering

In computing a schedule that respects the latency constraints, it is necessary to compute the amount of information stored between the sender and the receiver. The amount of information stored between two *Filter*s is the amount of information that entered the interval between those two *Filter*s minus the amount of information that is destroyed due to peeking of *Filter*s plus some data possibly put in the feedback path of a *FeedbackLoop* (due to $delay_{fl} > 0$).

When a *Filter* with $e_f > o_f$ executes for the first time, it observes $info_{in_f} * e_f$ information, but pushes out only $info_{out_f} * u_f$ information. Since for every *Filter* $info_{in_f} * o_f = info_{out_f} * u_f$ and $e_f > o_f$, the amount of information observed is not the same as the amount of information pushed out. Thus some information is lost during the first execution of such a *Filter*. This amount is $(e_f - o_f) * info_{in_f}$.

We account for this lost information by setting the initial amount of information in a the *input channel* of every *Filter* to $(e_f - o_f) * info_{in_f}$. If $e_f = o_f$, the initial amount of information in the *input channel* of f is set to 0. If $e_f > o_f$, the inital amount of information is set to a negative number.

When a *StreamIt* program is executing, we define the amount of information stored in a *channel* to be equal to the initial amount of information in the *channel* plus the amount of information pushed into the *channel* minus the amount of information popped from the *channel*.

Now, in order to compute the amount of information between two *Filter*s we simply need to sum up the amount of information stored in all *channel*s between these two *Filter*s along a directed non-cyclical path. The selection of this path is important, as not all paths between two *Filter*s will have equal amount of information stored. The path we select is the path that stores the least amount of information before any *Filter*s are executed. In other words, we select the path that has the most negative information stored in it at initialization. This is also equivalent to the amount of information entering the path through the upstream *Filter* minus the amount of information leaving the path through the downstream *Filter* plus the sum of information along the path at initialization (a non-positive value).

In our sample *Pipeline*, we have $in_E = -3$ and all other *channel*s initialized with 0 information. Thus the path selected between *Filter*s A and G is $A \rightarrow B \rightarrow s \rightarrow D \rightarrow E \rightarrow j \rightarrow G$, which contains -3 units of information at initialization.

6.4.3 Information Buffering and Latency

The last element of relating information flow and latency constraints is expressing the latency constraints in terms of information buffered up between the sender and recipient of a message. The three types of latency constraints are analyzed below.

Downstream Positive

The downstream positive latency constraint is the easiest one to analyze. A downstream positive delay specifies that a downstream recipient should receive the message before it observes information wavefront that will be produced by the sender in the future. In other words, there is no restriction on the amount of data buffered up between the sender and receiver, because the information wavefront cannot have possibly entered the path between the sender and recipient. Thus downstream positive latency constraint are effectively ignored by the scheduler.

Upstream Positive

An upstream positive latency constraint specifies that the recipient should receive the message just before it produces an information wavefront that will be observed by the sender between l_0 and l_1 executions later. This specifies an upper limit on the amount of information stored between the *Filter*s. If f_s is the recipient, the amount of information stored between the sender and the recipient must be less than $l_1 * info_{in_{f_s}} * o_{f_s}$.

In our example, the latency for messages sent from *Filter* G to *Filter* A is $[1, 9]$. Thus when *Filter* G is executed, the amount of information between A and G must be less than $9 * 0.3 * 2 = 5.4$ An inspection of the sample schedule and amount of information stored between A and G reveals that amount of information stored between A and G peeks at 5.

Downstream Negative

A downstream negative latency constraint specifies that the recipient should receive the message just before it observes the information wavefront produced by the sender between $-l_1$ and $-l_0$ executions of the *work* function before it sent the message. This specifies a lower limit on the amount of information stored between the *Filter*s. If f_s is the sender, the amount of information stored between the sender and the recipient must be at least $-l_1 * info_{out_{f_s}} * u_{f_s}$.

In our example, the latency of the messages sent from *Filter* B to *Filter* F is $[-3, -1]$. Thus when *Filter* B is executed, the amount of information between B and F must be at least $-(-1) * 0.5 * 2 = 1$. An inspection of the sample schedule and amount of information stored between B and F reveals that amount of information stored between B and F bottoms out at 1.

6.5 Computing the Schedule

Now that we have the tools to restrict information flow in a way that will guarantee a schedule that respects the rules imposed on the schedule by the program, we are ready to compute the schedule.

6.5.1 Initialization Schedule

Construction of the initialization schedule with latency constraints is very different from initialization schedule without latency constraints. The reason for this is that buffering requirements imposed by peeking are much easier to satisfy than requirements imposed by latency constraints. The requirements imposed by latency constraints require global analysis of data (information) buffering. This is because different overlapping latency constraints can contain conflicting requirements (minimum versus maximum amount of buffered information).

The approach used here will create a solution using a simple set of linear equations. If the constraints are not too tight, it is easy to convert the fractional solutions of linear equations into a real schedule. If the constraints are tight, it may be necessary to use integer programming to obtain integer solutions which will automatically map exactly to a valid schedule.

It is important to note, that the goal of construction of an initialization schedule here is to create a buffering that satisfies all constraints imposed by the program. During initialization of the program, not all constraints will be satisfied. In fact, it is impossible to satisfy all constraints before initialization completes, because when the program begins executing it has no data buffered up, and there are latency constraints that require minimal buffering of data.

With all that in mind, equations that govern number of *StreamIt* graph node executions for the initialization schedule simply need to be written down. Here all filters will be represented by f_m with $0 \leq m < n_f$ with n_f filters in the program. *splitter*s will be represented by s_m and *joiner*s with j_m, $0 \leq m < n_{sj}$, with n_{sj} representing number of *SplitJoin*s and *FeedbackLoop*s. *channel*s will be represented by in_{f_m}, in_{s_m} and $in_{j_m,k}$ with k representing the kth branch of the *joiner*. *channel*s are designated by *StreamIt* nodes that use the *channel* as an *input channel*. Number of executions of node s (*Filter*, *splitter* or *joiner*) will be represented by c_s.

First equations required represent the amount of information present in *channel*s need to be written down. For every *channel*, that is simply the amount of information pushed in by the source node minus the amount of information popped by the drain node minus amount of information lost on first execution of the downstream node, namely $in_{s_{dst}} = info_{in_{s_{dst}}}(c_{s_{src}} *$

$u_{s_{src}} - c_{s_{dst}} * o_{s_{dst}} - (e_{s_{dst}} - o_{s_{dst}}))$. The only exception to this rule are joiner buffers in *FeedbackLoop*s. Those buffers start with $delay_{fl_m}$ data items, thus $info_{in_{j_m,1}} * delay_{fl_m}$ information needs to be added the amount of information stored in $in_{j_m,1}$.

Next the minimal amount of data required in buffers is required to at least match the extra peeking amount of buffers (otherwise the steady schedule would not be able to repeat indefinitely). This is simply restricted by $b_{f_m} \geq 0$. This takes care of the the peeking amount, because *Filter*s that peek destroy some information, which is accounted for by setting the amount of information to be negative. For buffers belonging to *splitter*s and *joiner*s, the only requirement is that the amount of information in the buffers is at least zero. This is easily expressed by $b_{s_m} \geq 0$ and $b_{j_m,k} \geq 0$. Thus for all buffers we simply require that $b \geq 0$.

The last set of equations needed puts restrictions based on latency constraints. As described above, those equations simply sum up information stored in all buffers between the source and destination *Filter*s, and make sure that it is less or more than what the latency constraint requires.

Solving equations above for cs yields number of executions of *Filter*s, *splitter*s and *joiner*s required in order to initialize a *StreamIt* program with latency constraints. The numbers of execution obtained may be non-integer, if only a linear programming solution is sought. Simple rounding of the solution may be able to result in valid schedule. If that is not the case, integer programming solutions may be obtained, which would guarantee satisfying all requirements. If there are multiple solutions, any of the solutions will satisfy the requirements of a valid initialization schedule.

Example

Here we present equations which can be used to initialize the *Pipeline* in Figure 6-1.

First we compute the amount of information in each buffer after initialization.

$$\begin{array}{ll}
b_B = info_{in_B} * (c_A * o_A - c_B * o_B) & b_C = info_{in_C} * (c_s * w_{s,0} - c_C * o_C) \\
b_D = info_{in_D} * (c_s * w_{s,1} - c_D * o_D) & b_F = info_{in_F} * (c_s * w_{s,2} - c_F * o_F) \\
b_G = info_{in_G} * (c_j * o_j - c_G * o_G) & b_{j,0} = info_{in_{j,0}} * (c_C * o_C - c_j * w_{j,0}) \\
b_{j,1} = info_{in_{j,1}} * (c_E * o_E - c_j * w_{j,1}) & b_{j,2} = info_{in_{j,2}} * (c_F * o_F - c_j * w_{j,2}) \\
b_E = info_{in_E} * (c_D * o_D - c_E * o_E - (e_E - o_E)) &
\end{array}$$

Next we ensure that all *channel*s store non-negative amounts of information after initialization:

$$\begin{array}{lll} b_B \geq 0 & b_C \geq 0 & b_D \geq 0 \\ b_E \geq 0 & b_F \geq 0 & b_G \geq 0 \\ b_{j,0} \geq 0 & b_{j,1} \geq 0 & b_{j,2} \geq 0 \\ b_s \geq 0 & & \end{array}$$

Finally, following equations ensure that all buffers contain enough information for the graph to be considered initialized for steady state execution.

$$\begin{aligned} b_B + b_s + b_D + b_E + b_{j,1} + b_G &< 5.4 \\ b_{s_0} + b_{f_5} &\geq 1 \end{aligned}$$

One solution to the equations above, which can be found using an integer linear programming solver) is:

$$\begin{array}{lll} c_A = 2 & c_B = 3 & c_C = 0 \\ c_D = 1 & c_E = 0 & c_F = 0 \\ c_G = 0 & c_s = 2 & c_j = 0 \end{array}$$

This solution corresponds to the initialization schedule for the example provided in Section 6.2.

6.5.2 Steady State Schedule

Calculating the steady state schedule should in most cases be a fairly simple task, but may, in some cases be very difficult. The distinction between these two situations is not very easy to define. Basically, if the imposed set of constraints is very tight (not much space to maneuver buffered data), creating such a schedule may be difficult. On the other hand, if there is some space to maneuver buffered data, applying a simple scheduling technique like minimum latency scheduling should work. Simple techniques tailored to satisfying latency constraints can be applied to programs that cannot be scheduled using minimum latency scheduling. The common case for computing steady state schedules should be relatively easy to compute, because message sending is meant to be a low bandwidth activity, and the delivery constraints are meant to have large ranges. Absence of such conditions will result in reduced performance of the compiled code, and the program should be redesigned to pass appropriate information using regular data flow through channels.

The technique described here uses as an input the *StreamIt* program with all of its latency constraints, as well as the amount of information stored in all *channel*s after initialization. The output of this algorithm is a steady state schedule which starts with the initialized program and executes one iteration of minimal steady state while respecting all constraints placed on the program. The algorithm uses integer programming to assure that if a valid schedule exists, it will be found. The resulting schedule is expressed as a flat list of *Filter* firings.

The first step is to compute the multiplicities of execution of all components. Let t represent the sum of all execution multiplicities of all components, $t = \sum_i S_{m,i}$, where S is the steady state for the program. Thus, there will be a total of t steps in the final schedule. Let variable c_s^r represent an execution of a component s during the rth step. The first restriction on cs is that $\sum_{r=0}^{t-1} c_s^r = v_s$ (here the v notation is taken from subsection 3.2.2). Furthermore, we must have that $\sum_{\forall s \in \{all\ nodes\}} c_s^k = 1$. These two conditions assure that the schedule executed will indeed be the steady state schedule.

Next, the amount of information in *channel* in_s before step k is represented by

$$in_s^k = in_s + info_{in_{s_{dst}}} \left(\sum_{r=0}^{k-1} c_{c_{src}}^r - \sum_{r=0}^{k-1} c_s^r \right)$$

We impose the buffering requirements placed upon in_s for every in_s^i, thus ensuring that after every step of the program, all latency and peeking requirements are met.

Solving the resulting system of equations for all cs (including the cs required for initialization) will yield a correct schedule for the given program. The integer programming requirement should be that $\forall c, c \in \{0, 1\}$. The schedule is extracted for step r by finding the node s for which $c_s^r = 1$, and firing this component.

6.6 Example

Here we provide equations and a solution for construction of a steady state schedule for the *Pipeline* in Figure 6-1. The steady state for the sample *Pipeline* has been provided in Section 6.2.

The equations for determining the steady schedule are presented below. For the large part they are identical to equations governing the initialization schedule, except amount of information buffered is calculated differently, and there are far more variables. There are a total of 39 steady state execution steps. Variable r will be used here to denote a particular step of the schedule.

$$\forall r \in \{0,1,\ldots,38\} \begin{cases} b^r_B &= b_B + info_{in_B} \sum_{q=0}^{r}(c^q_A * u_A - c^q_B * o_B) & b^r_B \geq 0 \\ b^r_s &= b_s + info_{in_s} \sum_{q=0}^{r}(c^q_B * u_B - c^q_s * o_s) & b^r_s \geq 0 \\ b^r_C &= b_C + info_{in_C} \sum_{q=0}^{r}(c^q_s * w_{s,0} - c^q_C * o_C) & b^r_C \geq 0 \\ b^r_D &= b_D + info_{in_D} \sum_{q=0}^{r}(c^q_s * w_{s,1} - c^q_D * o_D) & b^r_D \geq 0 \\ b^r_E &= b_E + info_{in_E} \sum_{q=0}^{r}(c^q_D * u_D - c^q_E * o_E) & b^r_E \geq 0 \\ b^r_F &= b_F + info_{in_F} \sum_{q=0}^{r}(c^q_s * w_{s,2} - c^q_F * o_F) & b^r_F \geq 0 \\ b^r_{j,0} &= b_{j,0} + info_{in_{j,0}} \sum_{q=0}^{r}(c^q_C * u_C - c^q_j * w_{j,0}) & b^r_{j,0} \geq 0 \\ b^r_{j,1} &= b_{j,1} + info_{in_{j,1}} \sum_{q=0}^{r}(c^q_E * u_E - c^q_j * w_{j,1}) & b^r_{j,1} \geq 0 \\ b^r_{j,2} &= b_{j,2} + info_{in_{j,2}} \sum_{q=0}^{r}(c^q_F * u_F - c^q_j * w_{j,2}) & b^r_{j,2} \geq 0 \\ b^r_G &= b_G + info_{in_G} \sum_{q=0}^{r}(c^q_j * u_j - c^q_G * o_G) & b^r_G \geq 0 \\ \\ b^r_B + b^r_s + b^r_D + b^r_E + b^r_{j,1} + b^r_G < 5.4 \\ b^r_s + b^r_F \geq 1 \end{cases}$$

The following equations are bookkeeping equations that ensure that each component is executed the correct number of times, and that every step has exactly one execution.

$$\begin{array}{llllllll} \sum_{r=0}^{38} c^r_A = & 3 & \sum_{r=0}^{38} c^r_B = & 6 & \sum_{r=0}^{38} c^r_C = & 4 & \sum_{r=0}^{38} c^r_D = & 2 \\ \sum_{r=0}^{38} c^r_E = & 2 & \sum_{r=0}^{38} c^r_F = & 4 & \sum_{r=0}^{38} c^r_G = & 10 & \sum_{r=0}^{38} c^r_s = & 4 \\ \sum_{r=0}^{38} c^r_j = & 4 \end{array}$$
$$\forall r \in \{0,1,\ldots,38\} \sum_{s \in \{A,B,C,D,E,F,G,s,j\}} c^r_s = 1$$

Solving the equations above for non-negative integral c_s variables, yields a schedule which can be executed safely, without fear of violating any requirements imposed by the programmer. If no solution exists, then there is no schedule for the program, because the equations above do not overrestrict the execution of the program.

Table 6.4 provides a solution to the equations above. The solution corresponds to schedule provided in Section 6.2. Once again that schedule is

$$\{F^m FBA^m \{2A\} F^m F \{2B\}\{2C\}\{2\ split\} DE \{2\ join\}\{5G\}$$
$$F^m FBA^m AF^m F\{2B\}\{2C\}\{2\ split\} DE\{2\ join\}\{5G\}\}$$

step \ node	A	B	s	C	D	E	F	j	G
0	0	0	0	0	0	0	1	0	0
1	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0
5	0	1	0	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0	0
7	0	0	0	1	0	0	0	0	0
8	0	0	0	1	0	0	0	0	0
9	0	0	1	0	0	0	0	0	0
10	0	0	1	0	0	0	0	0	0
11	0	0	0	0	1	0	0	0	0
12	0	0	0	0	0	1	0	0	0
13	0	0	0	0	0	0	0	1	0
14	0	0	0	0	0	0	0	1	0
15	0	0	0	0	0	0	0	0	1
16	0	0	0	0	0	0	0	0	1
17	0	0	0	0	0	0	0	0	1
18	0	0	0	0	0	0	0	0	1
19	0	0	0	0	0	0	0	0	1
20	0	0	0	0	0	0	1	0	0
21	0	1	0	0	0	0	0	0	0
22	1	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	1	0	0
24	0	1	0	0	0	0	0	0	0
25	0	1	0	0	0	0	0	0	0
26	0	0	0	1	0	0	0	0	0
27	0	0	0	1	0	0	0	0	0
28	0	0	1	0	0	0	0	0	0
29	0	0	1	0	0	0	0	0	0
30	0	0	0	0	1	0	0	0	0
31	0	0	0	0	0	1	0	0	0
32	0	0	0	0	0	0	0	1	0
33	0	0	0	0	0	0	0	1	0
34	0	0	0	0	0	0	0	0	1
35	0	0	0	0	0	0	0	0	1
36	0	0	0	0	0	0	0	0	1
37	0	0	0	0	0	0	0	0	1
38	0	0	0	0	0	0	0	0	1

Table 6.4: A solution to equations for steady state of stream from Figure 6-1.This solution corresponds to the schedule given in 6.2.

Chapter 7

Results

In this chapter we present results of creating schedules using techniques described in Chapters 4 and 5. No results are presented for latency constrained scheduling, as no applications have been written to exploit the usefulness of messaging.

Section 7.1 presents the applications used for testing. Section 7.2 presents the methodology used for testing. Section 7.3 presents the results and analysis.

7.1 Applications

Our benchmark suite contains 13 applications. Out of those applications, 11 represent useful practical computation taken from real-life applications, while two were chosen to highlight effectiveness of phasing scheduling.

Nine test applications (bitonic sort, FFT, filter bank, FIR, radio, GSM, 3GPP, radar and vocoder) used are code-complete and perform the computations intended. Some results of compiling these applications can be found in [12].

Two test applications (QMF and CD-DAT) are applications used in another publication on scheduling streaming applications ([18]). The code inside of the *Filter*s has not been implemented.

The QMF application is a qmf12_3d. It had to be modified slightly to account for *StreamIt splitter*s and *joiner*s not allowing any computation. The high-pass and low-pass filtering in the *splitter*s has been moved to just after data been separated into two channels. The re-combining of data in the *joiner*s has been moved to a *Filter* just after the *joiner*s. The low and high pass filters have also been given a peek amount of 16 so they can perform their function in the way intended in *StreamIt*.

CD-DAT is exactly the same application as that described in [18].

The last two applications (SJ_PEEK_1024 and SJ_PEEK_31) are a synthetic benchmarks.

Figures illustrating the layout of all of the applications are available in Appendix A.

7.2 Methodology

The following data has been collected: number of nodes, number of node executions per steady state, schedule size and buffer size for pseudo single appearance and minimal latency schedules.

7.2.1 Schedule Compression

The size of schedules for minimal latency technique contains two numbers. The first one is an uncompressed schedule, while the second is a compressed schedule. During testing it was found that in some applications some streams had many phases that were identical to other phases of the stream. Instead of including these phases in the final schedule multiple, they were listed only once, and references to the duplicate phases have been replaced with references to their copies.

This optimization lead to improvements in schedule size for two reasons. First, streams now had less phases, so their schedules took up less space. Second, applications using the phasing schedules could now execute the same phase multiple times in a row, which was optimized out using run length encoding.

This compression has no negative effects on speed of execution, and never increases the size of a schedule. This compression has no effect on the pseudo single-appearance schedules, thus is not included in the results as a separate value.

7.2.2 Sinks

Any application in *StreamIt* must receive its data from somewhere, and its data must be sent somewhere. *Filter*s that perform these functions are called sinks and sources. In particular, sinks have the property of having $u_f = 0$ while sources have $e_f = o_f = 0$. In other words, sinks do not push any data out and sources do not consume any data.

Sinks are problematic for minimal latency scheduling purposes, because they do not produce any data. Remember that a minimal latency schedule will execute a bottom-most *Filter* of a *Pipeline* in every phase as many times as is necessary to produce some data. Since sinks do not produce any data, the sinks are executed their steady state number of executions. This leads to the minimal latency schedule of the outer-most *Pipeline* becoming a single appearance schedule, thus destroying some of the benefit of using phasing scheduling.

This problem has been alleviated by detecting sinks at the end of a *Pipeline* and scheduling them in a unique way. Namely, a simple attempt is made to minimize the amount of

benchmark	number of nodes	number of node execu-tions	pseudo single appearance		minimal latency		
			schedule size	buffer size	schedule size	compressed schedule size	buffer size
bitonic sort	370	468	439	2112	448	448	2112
CD-DAT	6	612	7	1021	170	72	72
FFT	26	488	31	3584	31	31	3584
filter bank	53	312	166	2063	160	145	1991
FIR	132	152	133	1560	133	133	1560
radio	30	43	58	1351	50	50	1351
GSM	47	3356	-	-	724	78	3900
3GPP	94	356	147	986	149	137	970
QMF	65	184	143	1225	132	122	1225
radar	68	161	100	332	100	100	332
SJ_PEEK_1024	6	3081	11	7168	40	16	4864
SJ_PEEK_31	6	12063	11	19964	250	24	12063
vocoder	117	415	172	1285	293	206	1094

Table 7.1: Results of running pseudo single appearance and minimal latency scheduling algorithms on various applications.

storage necessary to store the phases of the *Pipeline*.

Let the amount of storage necessary to store one data item in *input channel* to the sink be x, the amount of storage necessary to store a phase be y, the sink consume a data per steady state execution of its parent *Pipeline* and b be the number of phases of the parent pipeline, then we have that amount of storage necessary to store the phases and the buffer is

$$\frac{ax}{b} + by$$

We want to minimize this amount, with b being the variable. We take a derrivative of the above expression, set it to zero and solve:

$$\begin{aligned} -\tfrac{ax}{b^2} + y &= 0 \\ yb^2 &= ax \\ b &= \sqrt{\tfrac{ax}{y}} \end{aligned}$$

For simplicity, we set $x = y = 1$, thus obtaining that $b = \sqrt{a}$.

Now, for every phase of the parent *Pipeline* of the sink, the sink is executed $\sqrt{a}$ times on the first step of scheduling a phase of the *Pipeline*.

7.3 Results

Table 7.1 presents buffer and schedule sizes necessary to execute various applications using the algorithms developed in this thesis.

The GSM application cannot be scheduled using pseudo single-appearance algorithm, because it has a loop which is too tight for execution under the SAS.

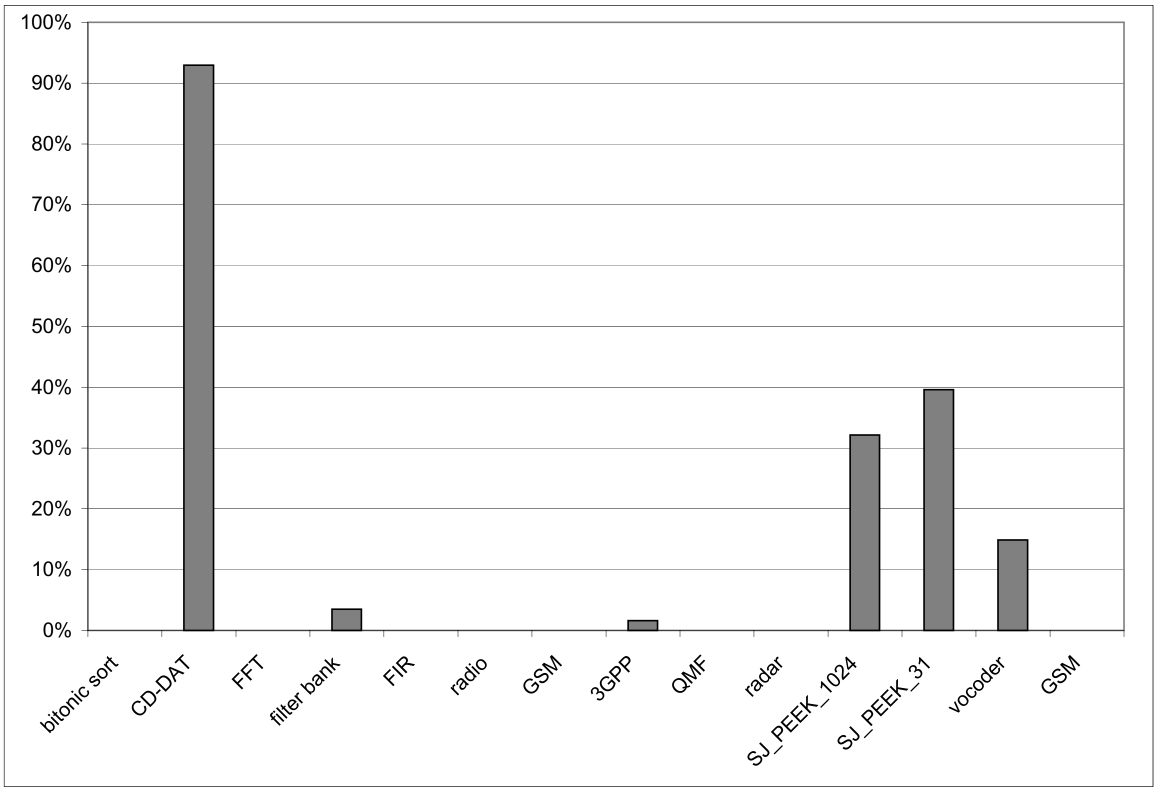


Figure 7-1: Buffer storage space savings of Phased Minimal Latency schedule vs. Hierarchical schedule. All data in all *channel*s is assume to consume same amount of space.

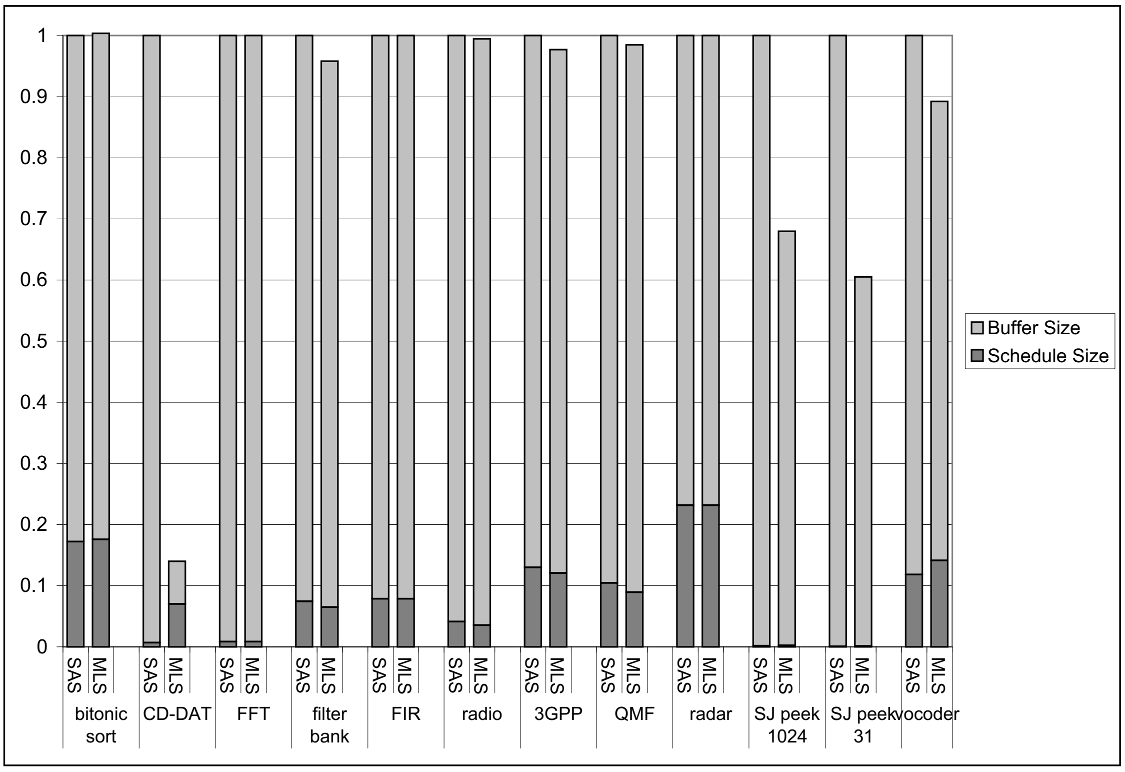


Figure 7-2: Storage usage for compressed Minimal Latency Phased schedule vs. Hierarchical schedule. Left bars are for Hierarchical schedules. Numbers are normalized to total storage required by Hierarchical schedule. Each entry in every schedule and data items in all *channel*s are assumed to consume same amount of space.

Several applications show a very large improvement in buffer size necessary for execution. Namely, CD-DAT decreases from 1021 to 72, a 93% improvement. [18] reports a buffer size of 226 after applying buffer merging techniques. Our improvement is due to reducing the combinatorial growth of the buffers using phasing scheduling.

Our synthetic benchmarks decrease from 7168 to 4864 and from 19964 to 12063, a 32% and 40% improvements. The first improvement is due to creating fine grained phases which allow the initialization schedule to transfer smaller amount of data and allow the children of the *SplitJoin* to drain their data before the *splitter* provides them with more. This improvement is only created in presence of peeking. The second improvement is due to reducing combinatorial growth and due to finer grained schedules to deal with peeking.

Other applications show no or little improvement in buffer requirements. As expected, no application requires more buffer space.

It is interesting to note that the schedule sizes have decreased between the single appearance and compressed minimal latency phasing schedules. This is due to slightly different encoding technique of single appearance schedules.

Chapter 8

Related Work

There has been a wealth of research of various stream languages and projects. This chapter will introduce some of the other projects.

A large number of programming languages have included a concept of a stream; see [22] for a survey. Synchronous languages such as LUSTRE [14], Esterel [3], and Signal [9] also target the embedded domain, but they are more control-oriented than StreamIt and are not aggressively optimized for performance. Sisal (Stream and Iteration in a Single Assignment Language) is a high-performance, implicitly parallel functional language [8]. The Distributed Optimizing Sisal Compiler [8] considers compiling Sisal to distributed memory machines, although it is implemented as a coarse-grained master/slave runtime system instead of a fine-grained static schedule.

Ptolemy [16] is a simulation environment for heterogenous embedded systems, including Synchronous Data Flow that is similar to static-rate stream graphs in *StreamIt*. SDF programs, however, do not include the peeking and messaging constructs of *StreamIt*. In SDF languages, actors are the active computational elements (*Filter*s). SDF computation model does not impose structure on the program. All actors are allowed to have multiple input and output channels. [2] provides an overview of dataflow synchronous languages.

There are many results of scheduling SDF programs [6]. Many of these results concentrate on reducing buffering requirements. Many of the systems inline all actor code into a single function, and try to minimize the code and buffer size at once [4]. This approach leads to a strong preference for single appearance schedules. *StreamIt* scheduler stores a the schedule and *Filter* code separately, thus allowing for more efficient schedule storage.

Other publications focus on synthesis of software for embedded systems. [5] recognizes that non single-appearance schedules which inline actor invocations have problems with code growth. A hybrid model is introduced, where actor invocations are inlined unless the resulting code grow too large. If inlining is not performed, actors are invoked through a function call.

Other approaches include to reducing the buffering requirements use buffer merging [18], a technique which will be explored for *StreamIt* in the future.

Many publications focus on synthesis for hardware systems. Such systems can execute multiple actors at once [13]. Currently *StreamIt* compiler produces code for uniprocessor architectures [24] and for the RAW processor [11]. RAW processor can execute multiple *Filter*s in parallel, but every parallel execution is given its own separate set of resources like memory.

There are some streaming computation models which are less constrained than SDF. Most popular is Cyclic Synchronous Data Flow CSDF computation model [19]. CSDF actors have multiple *work* functions, with each one being allowed to consume/produce different number of data items. The current *StreamIt* phasing scheduler is able to accommodate this model of computation, but the *StreamIt* language does not support this feature yet. [27] provides an elegant solution to scheduling CSDF programs using Systems of Affine Recurrence Equations, which provides a closed-form solution scheduling for any valid CSDF graph.

[26] proposes a model where the flow of data is not static, but may depend on data being processed. The model is called Cyclo-Dynamic Data Flow (CDDF). This greatly helps flexibility of programming, but prevents fully static scheduling of programs.

The U.S. Navy Processing Graph Method (PGM) uses a version of SDF with an equivalent of peeking [10]. The paper is focused on real-time execution and provides analysis and verification of latency of data flow through their system.

Chapter 9

Conclusion and Future Work

This thesis presents a variety of techniques used for scheduling Synchronous Data Flow Graphs as used by the *StreamIt* language. Unlike other langauge, *StreamIt* enforces a structure on the stream graph, thus allowing a variety of new approaches to scheduling execution. Algorithms presented here improve current current scheduling techniques in multiple ways.

Hierarchical approach to scheduling execution of streaming applications allows for a simplification of algorithms. Program graphs do not have to be considered globally, thus less data needs to be kept track of. In hierarchical approaches presented here, we only need to consider immediate children of a given stream.

Phasing approach to scheduling allows to schedule arbitrarily tight *FeedbackLoop*s and allows for more fine-grained control of buffering requirements. The fine-grained control of buffering requirements can provide dramatic improvements in buffer requirements when scheduling streaming applications, as has been presented here. Furthermore, phased schedules lend themselves to some easy forms of compression, thus reducing the schedule size. Future work will concentrate on expanding phasing scheduling to implement schedules that conform to specific buffering constraints, take advantage of cache sizes, etc. Producing of a single schedule for many instances of identical streams will also be explored.

The solution to latency constrained scheduling presented here is an important contribution to development of *StreamIt*. It will be extended to allow for morphing graphs ([23]). It will also be adapted to use phasing scheduling to reduce buffer and schedule size.

Appendix A

Diagrams of Test Applications

This appendix presents the applications used for testing and collecting results in this thesis.

There are two formats of Figures in this appendix. CD-DAT, QMF and the two SJ_PEEK benchmarks have nodes denoted by ovals. The name of *splitter*s and *joiner*s indicates their type (*Duplicate* or *RoundRobin*) and possible splitting or joining amounts (if a *RoundRobin splitter* or *joiner* has no numbers, they're all unity). The name of *Filter*s has format $(pop, peek)name(push)$. *Pipeline*s and *SplitJoin*s are represented by rectangles, and their names are given in their top left corner.

The format for the other figures is similar, but the peek, pop and push amounts for *Filter*s is given explicitly.

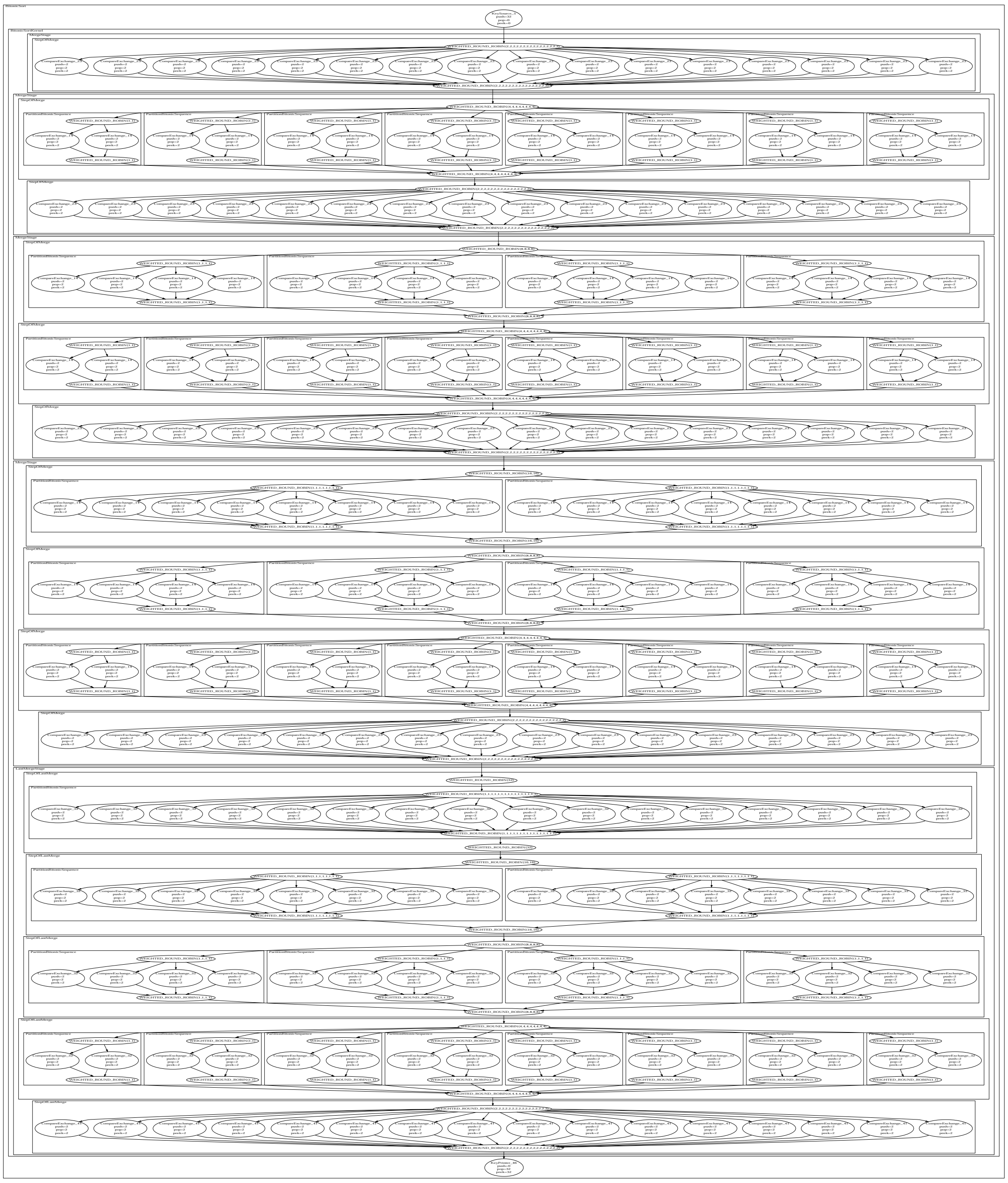


Figure A-1: Diagram of Bitonic Sort Application

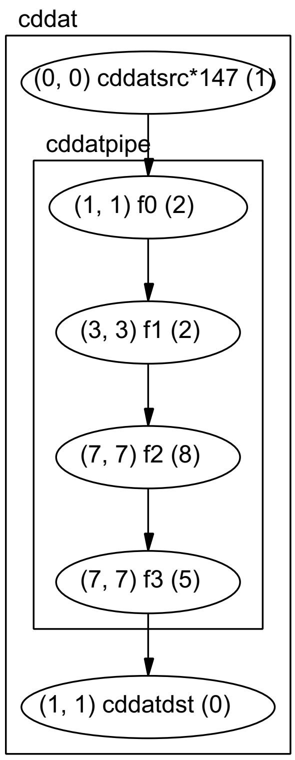


Figure A-2: Diagram of CD-DAT Application

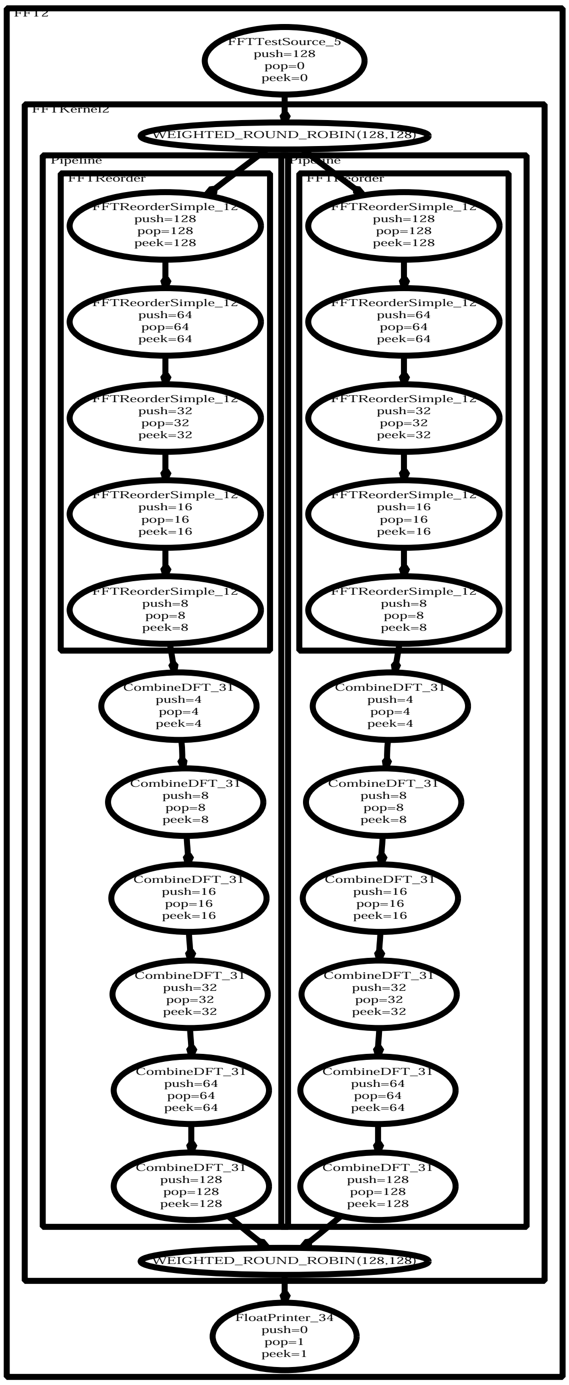


Figure A-3: Diagram of FFT Application

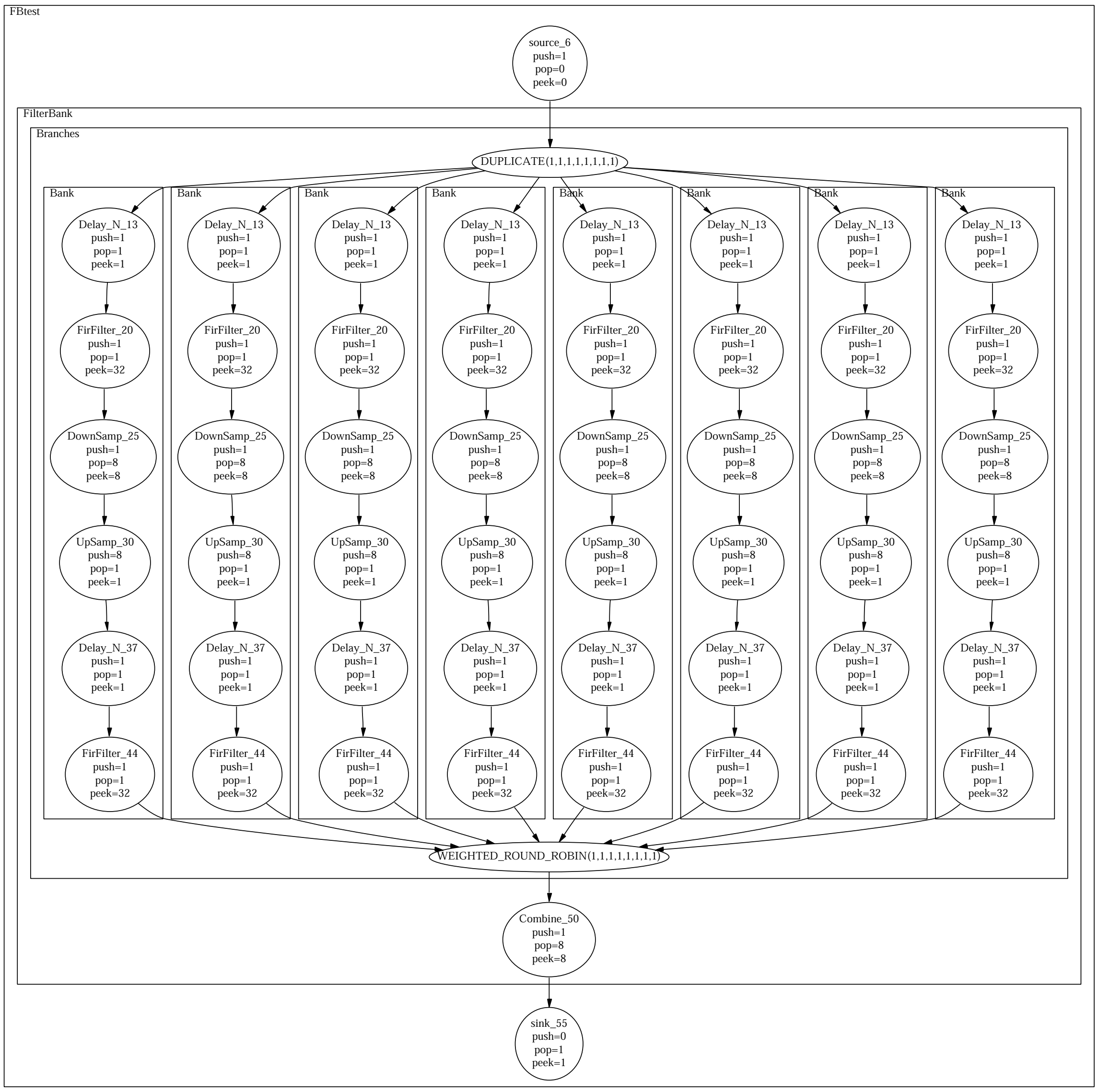

FBtest
source_6 push=1 pop=0 peek=0
FilterBank
Branches
DUPLICATE(1,1,1,1,1,1,1,1)
Bank
Delay_N_13 push=1 pop=1 peek=1
FirFilter_20 push=1 pop=1 peek=32
DownSamp_25 push=1 pop=8 peek=8
UpSamp_30 push=8 pop=1 peek=1
Delay_N_37 push=1 pop=1 peek=1
FirFilter_44 push=1 pop=1 peek=32
Bank
Delay_N_13 push=1 pop=1 peek=1
FirFilter_20 push=1 pop=1 peek=32
DownSamp_25 push=1 pop=8 peek=8
UpSamp_30 push=8 pop=1 peek=1
Delay_N_37 push=1 pop=1 peek=1
FirFilter_44 push=1 pop=1 peek=32
Bank
Delay_N_13 push=1 pop=1 peek=1
FirFilter_20 push=1 pop=1 peek=32
DownSamp_25 push=1 pop=8 peek=8
UpSamp_30 push=8 pop=1 peek=1
Delay_N_37 push=1 pop=1 peek=1
FirFilter_44 push=1 pop=1 peek=32
Bank
Delay_N_13 push=1 pop=1 peek=1
FirFilter_20 push=1 pop=1 peek=32
DownSamp_25 push=1 pop=8 peek=8
UpSamp_30 push=8 pop=1 peek=1
Delay_N_37 push=1 pop=1 peek=1
FirFilter_44 push=1 pop=1 peek=32
Bank
Delay_N_13 push=1 pop=1 peek=1
FirFilter_20 push=1 pop=1 peek=32
DownSamp_25 push=1 pop=8 peek=8
UpSamp_30 push=8 pop=1 peek=1
Delay_N_37 push=1 pop=1 peek=1
FirFilter_44 push=1 pop=1 peek=32
Bank
Delay_N_13 push=1 pop=1 peek=1
FirFilter_20 push=1 pop=1 peek=32
DownSamp_25 push=1 pop=8 peek=8
UpSamp_30 push=8 pop=1 peek=1
Delay_N_37 push=1 pop=1 peek=1
FirFilter_44 push=1 pop=1 peek=32
Bank
Delay_N_13 push=1 pop=1 peek=1
FirFilter_20 push=1 pop=1 peek=32
DownSamp_25 push=1 pop=8 peek=8
UpSamp_30 push=8 pop=1 peek=1
Delay_N_37 push=1 pop=1 peek=1
FirFilter_44 push=1 pop=1 peek=32
Bank
Delay_N_13 push=1 pop=1 peek=1
FirFilter_20 push=1 pop=1 peek=32
DownSamp_25 push=1 pop=8 peek=8
UpSamp_30 push=8 pop=1 peek=1
Delay_N_37 push=1 pop=1 peek=1
FirFilter_44 push=1 pop=1 peek=32
WEIGHTED_ROUND_ROBIN(1,1,1,1,1,1,1,1)
Combine_50 push=1 pop=8 peek=8
sink_55 push=0 pop=1 peek=1

Figure A-4: Diagram of Filter Bank Application

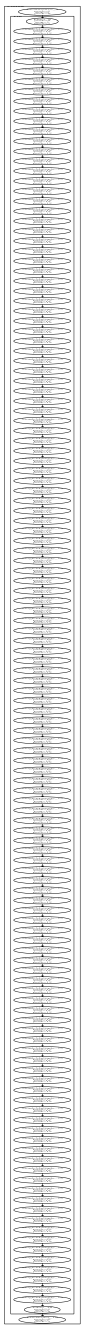


Figure A-5: Diagram of FIR Application

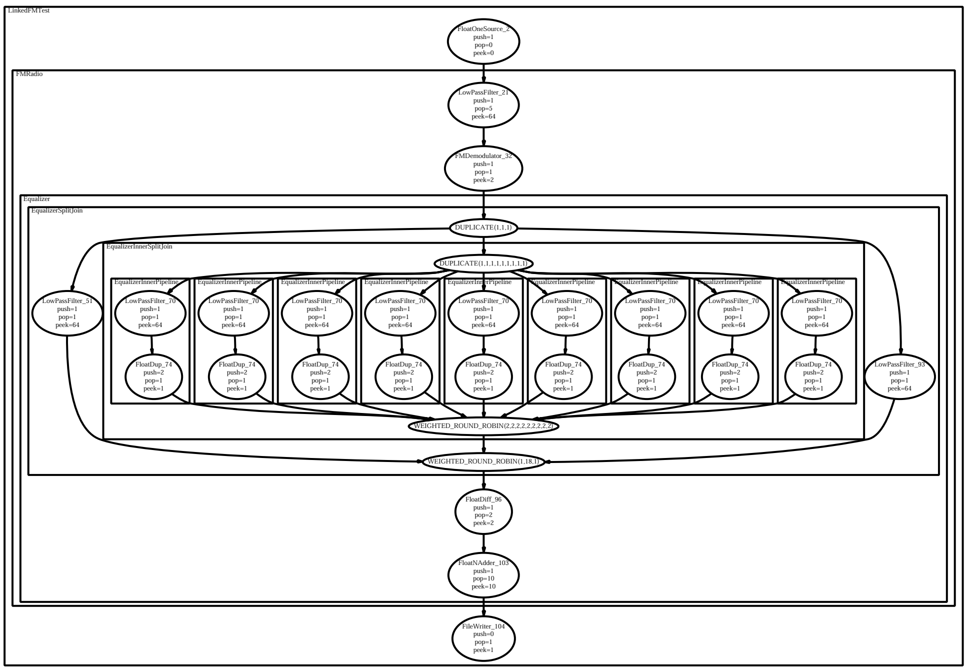

LinkedFMTest
FloatOneSource_2 push=1 pop=0 peek=0
FMRadio
LowPassFilter_21 push=1 pop=5 peek=64
FMDemodulator_32 push=1 pop=1 peek=2
Equalizer
EqualizerSplitJoin
DUPLICATE(1,1,1)
EqualizerInnerSplitJoin
DUPLICATE(1,1,1,1,1,1,1,1,1)
LowPassFilter_51 push=1 pop=1 peek=64
EqualizerInnerPipeline
LowPassFilter_70 push=1 pop=1 peek=64
FloatDup_74 push=2 pop=1 peek=1
LowPassFilter_93 push=1 pop=1 peek=64
WEIGHTED_ROUND_ROBIN(2,2,2,2,2,2,2,2,2)
WEIGHTED_ROUND_ROBIN(1,18,1)
FloatDiff_96 push=1 pop=2 peek=2
FloatNAdder_103 push=1 pop=10 peek=10
FileWriter_104 push=0 pop=1 peek=1

Figure A-6: Diagram of Radio Application

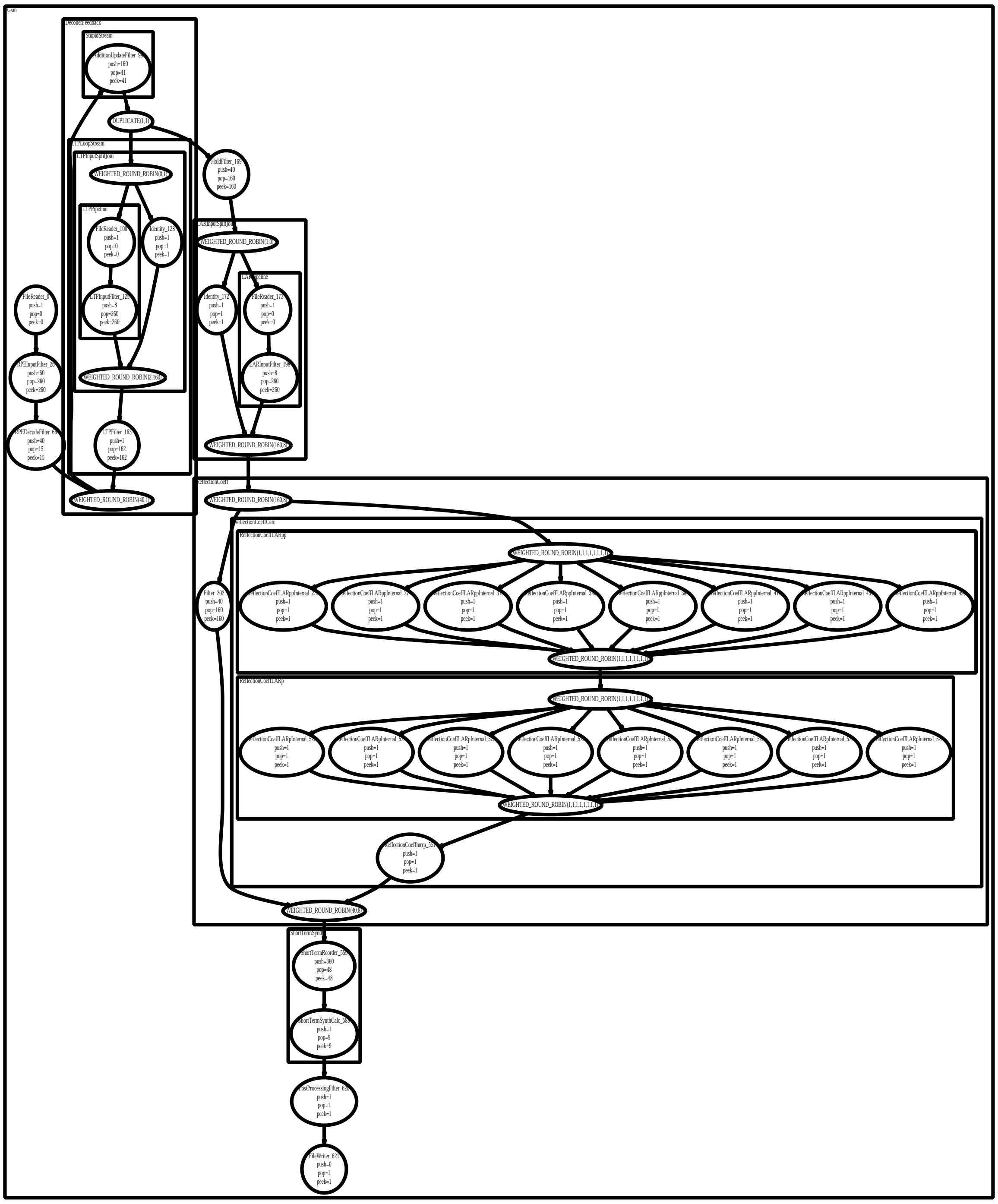


Figure A-7: Diagram of GSM Application

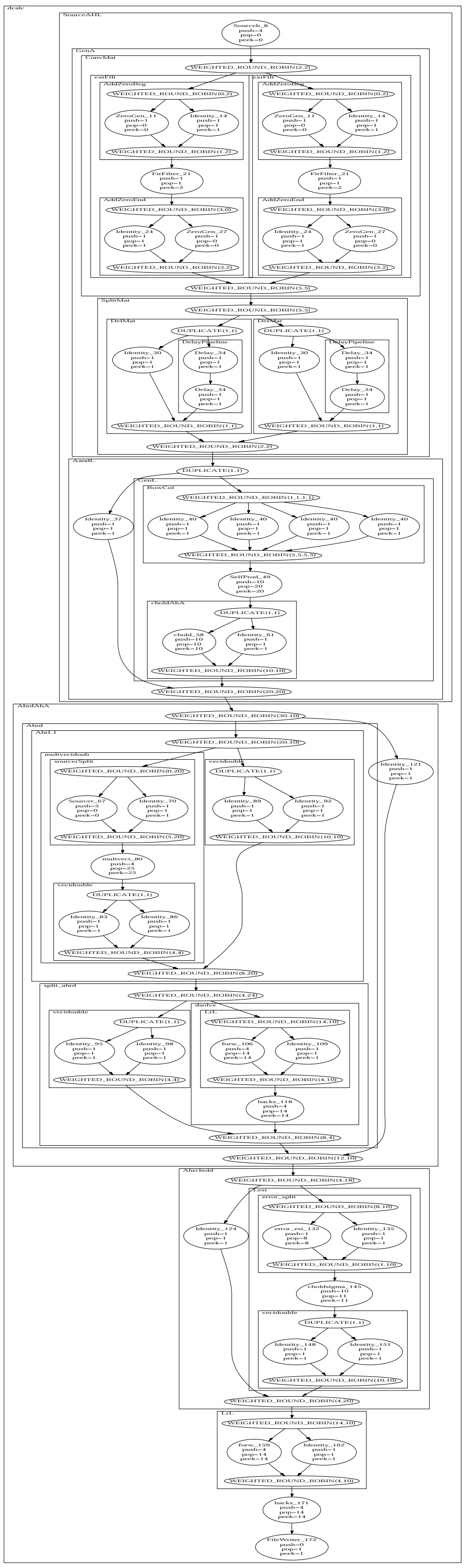


Figure A-8: Diagram of 3GPP Application

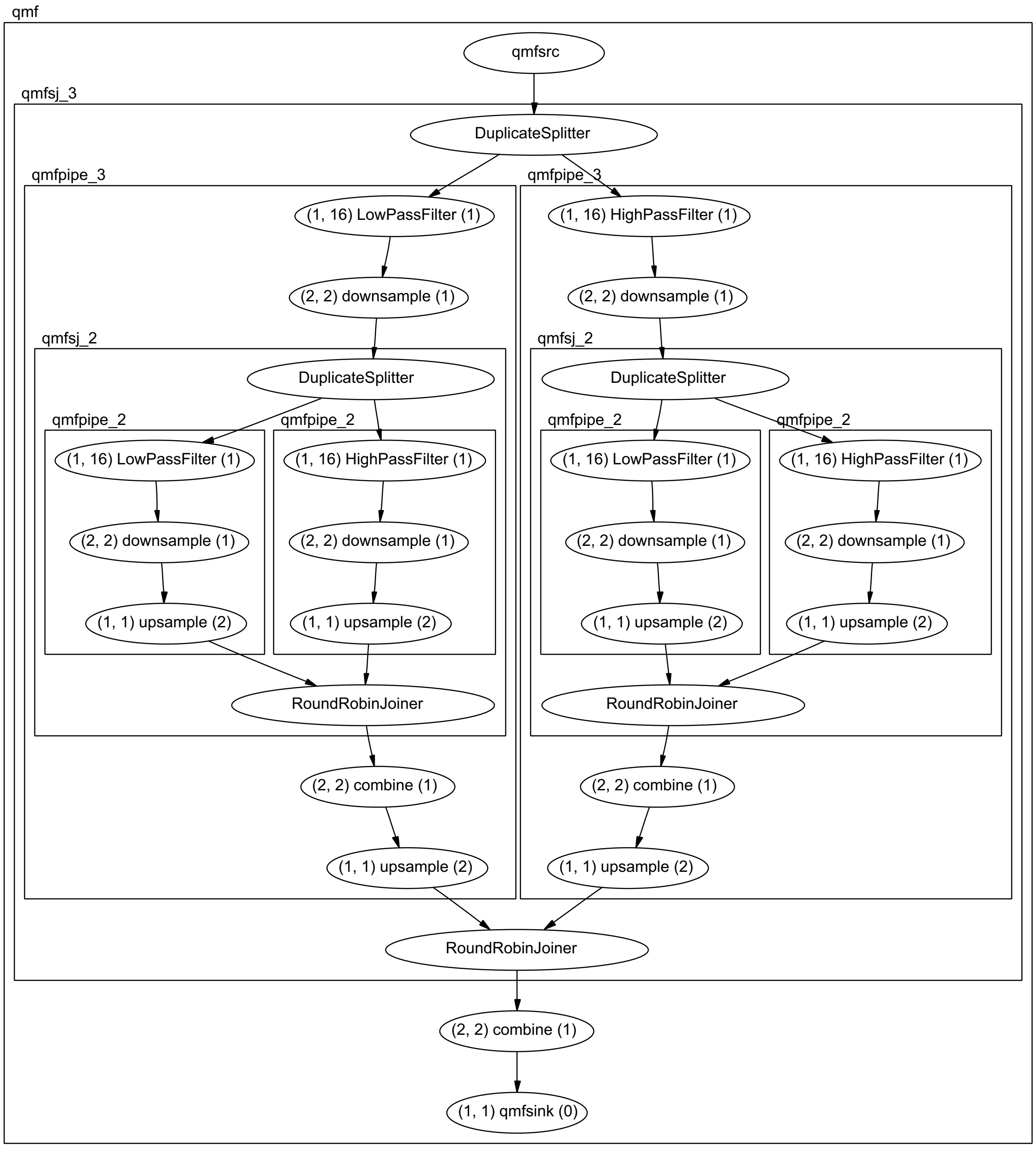


Figure A-9: Diagram of QMF Application

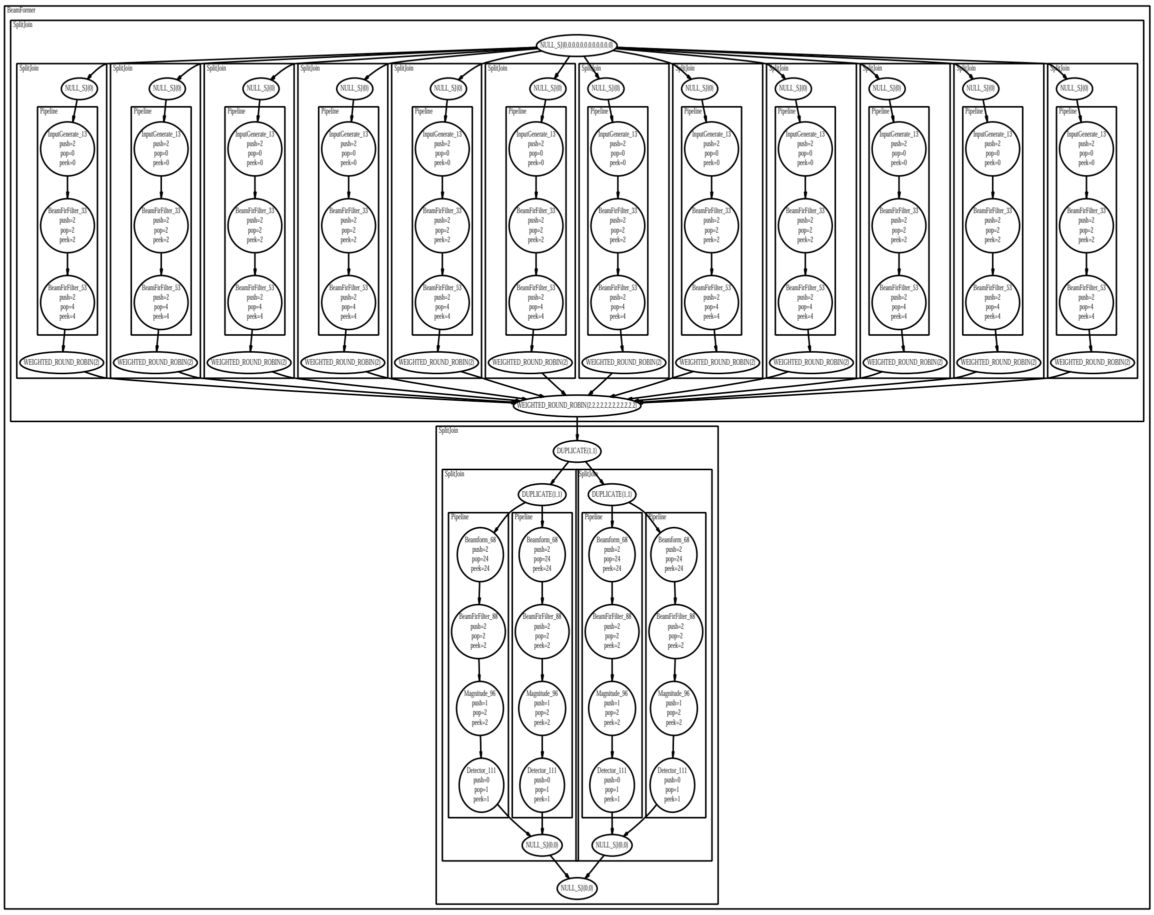

BeamFormer
SplitJoin
NULL_SJ(0)
Pipeline
InputGenerate_13 push=2 pop=0 peek=0
BeamFirFilter_33 push=2 pop=2 peek=2
BeamFirFilter_53 push=2 pop=4 peek=4
WEIGHTED_ROUND_ROBIN(2)
DUPLICATE(1,1)
Beamform_68 push=2 pop=24 peek=24
BeamFirFilter_88 push=2 pop=2 peek=2
Magnitude_96 push=1 pop=2 peek=2
Detector_111 push=0 pop=1 peek=1
NULL_SJ(0,0)

Figure A-10: Diagram of Radar Application

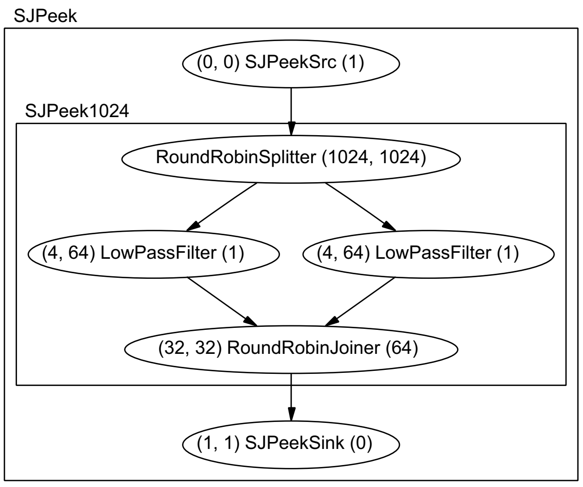

SJPeek
(0, 0) SJPeekSrc (1)
SJPeek1024
RoundRobinSplitter (1024, 1024)
(4, 64) LowPassFilter (1)
(4, 64) LowPassFilter (1)
(32, 32) RoundRobinJoiner (64)
(1, 1) SJPeekSink (0)

Figure A-11: Diagram of SJ_PEEK_1024 Application

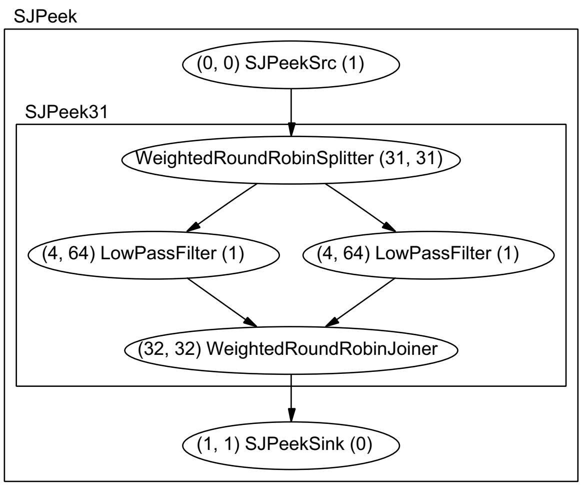

SJPeek
(0, 0) SJPeekSrc (1)
SJPeek31
WeightedRoundRobinSplitter (31, 31)
(4, 64) LowPassFilter (1)
(4, 64) LowPassFilter (1)
(32, 32) WeightedRoundRobinJoiner
(1, 1) SJPeekSink (0)

Figure A-12: Diagram of SJ_PEEK_31 Application

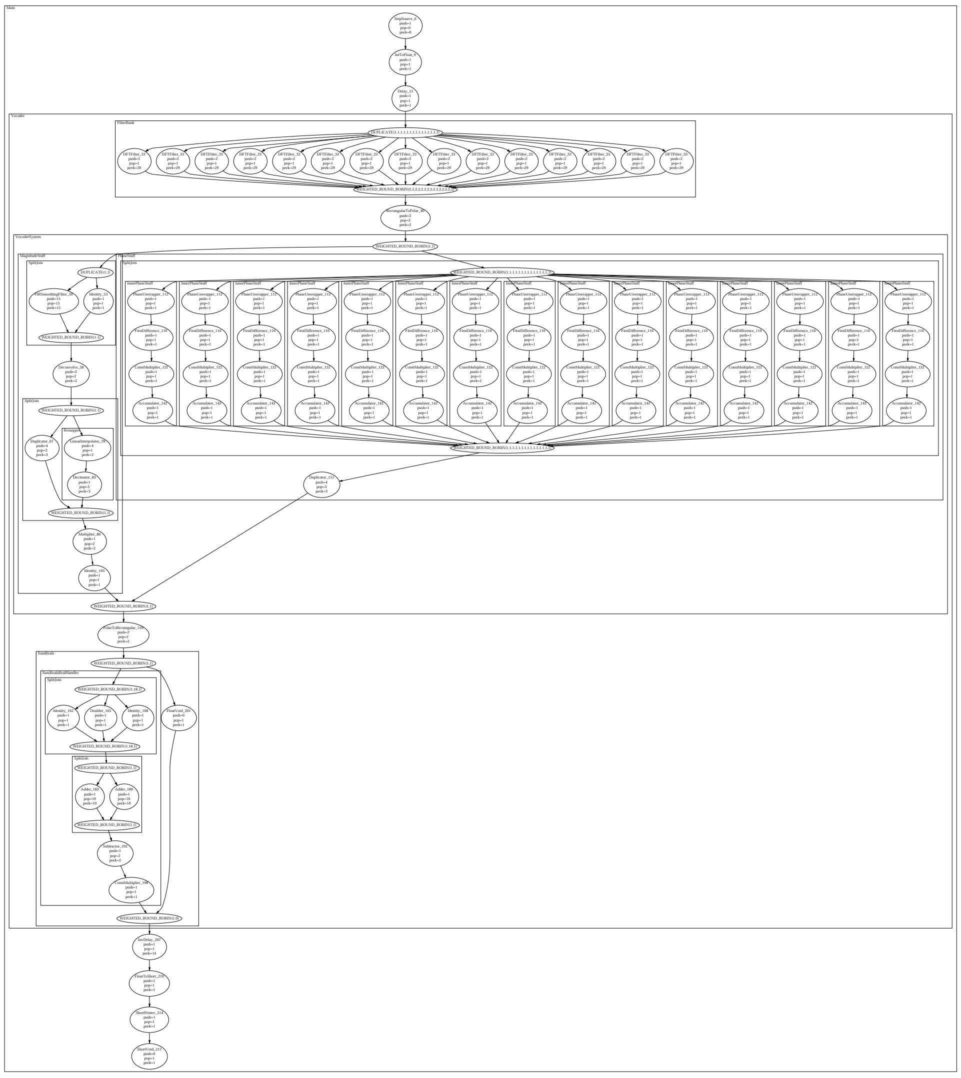


Figure A-13: Diagram of Vocoder Application

Bibliography

[1] H. Abelson and G. Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, 1985.

[2] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-flow synchronous languages. In *REX School/Symposium*, pages 1–45, 1993.

[3] Gerard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, I mplementation. *Science of Computer Programming*, 19(2), 1992.

[4] S. Bhattacharyya, P. Murthy, and E. Lee. Synthesis of embedded software from synchronous dataflow specifications, 1999.

[5] S. S. Bhattacharyya. Optimization Trade-offs in the Synthesis of Software for Embedded DSP. In *Proceedings of the International Workshop on Compiler and Architecture Support for Embedded Systems*, October 1999. Washington, D. C.

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

[7] Shuvra S. Bhattacharyya and Edward A. Lee. Looped schedules for dataflow descriptions of multirate signal processing algorithms. *Formal Methods in System Design*, 5(3):183–205, 1994.

[8] "J. Gaudiot, W. Bohm, T. DeBoni, J. Feo, and P. Mille". The Sisal Model of Functional Programming and its Implementat ion. In *Proceedings of the Second Aizu International Symposium on Paralle l Algorithms/Architectures Synthesis*, 1997.

[9] Thierry Gautier, Paul Le Guernic, and Loic Besnard. Signal: A declarative language for synchronous programming of real-time systems. *Springer Verlag Lecture Notes in Computer Science*, 274, 1987.

[10] S. Goddard and K. Jeffay. Analyzing the real-time properties of a u.s. navy singer processing system, 2000.

[11] Michael Gordon. A stream-aware compiler for communication exposed-architectures.

[12] Michael Gordon, William Thies, Michal Karczmarek, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *In the Proceedings of the Tenth Conference on Architectural Support for Programming Languages and Oper ating Systems*, October 2002.

[13] R. Govindarajan, Guang R. Gao, and Palash Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks.

[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proc. of the IEEE*, 79(9), 1991.

[15] E. Lee and D. Messershmitt. Static scheduling of synchronous data flow programs for digital signal processing, 1987.

[16] Edward A. Lee. Overview of the Ptolemy Project. UCB/ERL Technical Memorandum UCB/ERL M01/11, Dept. EECS, University of California, Berkeley, CA, March 2001.

[17] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular recongurable architecture, 2000.

[18] P. Murthy and S. Bhattacharyya. Buffer merging—a powerful technique for reducing memory requirements in sdf specifications, 1999.

[19] T. Parks, J. Pino, and E. Lee. A comparison of synchronous and cyclostatic dataflow, 1995.

[20] Scott Rixner and et al. A Bandwidth-Efficient Architecture for Media Processing. In *HPCA*, Dallas, TX, November 1998.

[21] K. Sankaralingam, R. Nagarajan, S.W. Keckler, and D.C. Burger. A Technology-Scalable Architecture for Fast Clocks and High ILP. UT Austin Tech Report 01-02, 2001.

[22] Robert Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7), 1997.

[23] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Computational Complexity*, pages 179–196, 2002.

[24] William Thies, Michal Karczmarek, Michael Gordon, David Maze, Jeremy Wong, Henry Hoffmann, Matthew Brown, and Saman Amarasinghe. StreamIt: A Compiler for Streaming Applications. MIT-LCS Technical Memo LCS-TM-622, Cambridge, MA, 2001.

[25] Elliot Waingold and et al. Baring it all to Software: The Raw Machine. MIT-LCS Technical Report 709, Cambridge, MA, 1997.

[26] P. Wauters, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-dynamic dataflow, 1996.

[27] Jasper Lin William Thies and Saman Amarasinghe. Phased Computation Graphs in the Polyhedral Model. MIT-LCS Technical Memo LCS-TM-630, Cambridge, MA, 2002.