Linear State-Space Analysis and Optimization of StreamIt Programs

by

Sitij Agrawal

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2004

Author . . . .
Department of Electrical Engineering and Computer Science
August 26, 2004

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

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

Linear State-Space Analysis and Optimization of StreamIt Programs

by

Sitij Agrawal

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Digital devices and their underlying DSP technology are widespread in modern society. Consequently, optimizing DSP applications is necessary to satisfy tight power, space, and bandwidth constraints. As code length and complexity increases for these applications, the value of time-intensive, manual expert analysis has decreased. In its place, robust, compiler-generated optimizations have become necessary.

We target linear state-space sections of applications to analyze and optimize, and use the programming language StreamIt to implement and test our ideas. StreamIt enables its users to create filters and connect them in a simple, structured manner. A linear state-space filter may have a set of state variables that it uses from execution to execution. Such a filter has the property that its outputs are a linear combination of its inputs and states, and the states are updated by a linear combination of the inputs and states. Examples of such filters that use states include IIR filters and linear difference equations.

We automate the following steps in the StreamIt compiler. We extract the representation for each filter that is linear state-space. We combine representations of filters to form a single linear state-space filter. We remove unnecessary states and reduce the number of computations needed per filter execution. Lastly, we convert the optimized filter back to StreamIt code.

We have compiled a number of StreamIt benchmark applications using our analysis and optimizations. Our data indicates state-space replacement works as well as linear replacement (a filter is linear if its outputs are an affine combination of its inputs) for stateless filters, and improves performance by as much as 66% for combinations of filters with state.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

First, I would like to thank my family for all their support throughout my college years. I would like to thank the members of the StreamIt group - in particular Jasper Lin and David Maze - for patiently answering all my questions and for helping me understand the StreamIt language and compiler. I would like to thank Andrew Lamb, whose work on linear analysis of StreamIt programs provided the foundation for my own work on state-space analysis. His thesis and well-constructed code were invaluable to me. Rodric Rabbah, another member of our group, gave me excellent comments about the writing in this thesis. I would like to thank my advisor, Saman Amarasinghe, for giving me the opportunity to work on the StreamIt project and for funding my research. Finally, I would like to thank Bill Thies for guiding me through every step of my project. That I was able to complete this thesis is a testament to his mentoring ability. I could not have done it without him.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Overview

Digital devices are increasingly common in everyday life. Examples include cell phones, modems, CD players, and high definition television. These products require DSP (Digital Signal Processing) applications to operate on their real-time streaming data. Applications have wide-ranging uses, such as signal compression and decompression, noise reduction, and error correction.

DSP applications often must process massive amounts of data quickly with limited power consumption. Therefore, it is crucial they are optimized appropriately. Unfortunately, DSP optimizations typically defy high level language compiler analysis. Consequently, DSP applications must be hand-coded or at the very least fine-tuned at the assembly level. This leads to a host of problems: DSP experts must spend valuable hours writing optimized low level code; every change in the design of the application necessitates rewriting the code; the optimizations are typically architecture dependent, hence they are not portable or robust. These factors indicate there is a need to effectively analyze DSP applications, and automate their optimizations in a compiler.

1.2 DSP Analysis

In order to properly analyze DSP applications, we must use an appropriate framework to model them. This framework should contain a number of simplifications in order to make our analysis workable, but not too many simplifications that our analysis fails to be robust.

We start with the top level notion of an application, defined as a large module that receives inputs, performs computations, and outputs results. This definition, while correct, does not lend itself to any type of application analysis. The first simplification we make is to divide an application into blocks, which are abstract input-output modules. These blocks are interconnected in a certain way to form the full application. We can think of each block as a mini-application: it takes its own inputs, performs calculations, and produces outputs.

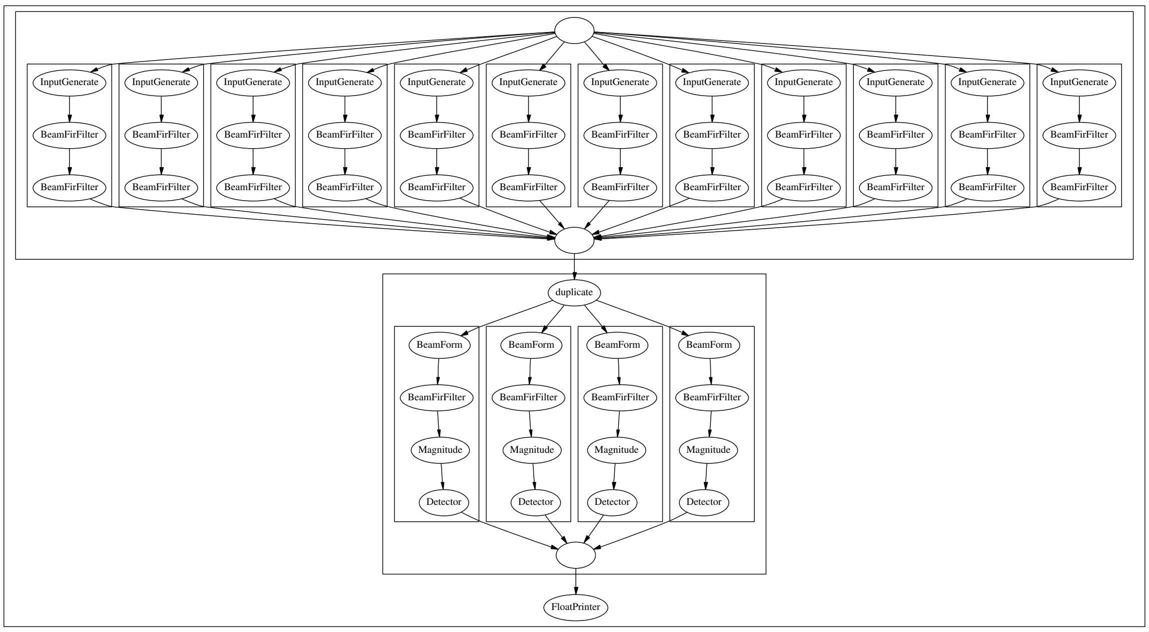

Figure 1-1: A DSP block diagram of the application Beamformer

Blocks can be characterized in various ways. The simplest characterization of blocks is a *linear* block, defined as a module that outputs a linear combination of its inputs plus a constant term. A linear block can be represented by a matrix relating inputs to outputs and a vector of constants. The next simplest characterization of blocks is *linear state-space.* Such a block uses a set of state variables. The output of this block is a linear combination of its inputs and state variables. In addition, the state variables are updated by a linear combination of themselves and inputs. A linear state-space block can be represented by four independent matrices.

A linear state-space characterization is more general than a linear characterization - all linear blocks are also linear state-space blocks, but the converse is not true. The intuitive reason for this fact is that a linear block is memoryless, meaning the outputs only depend on current inputs. However, a linear state-space block has memory in the form of state variables, so the outputs depend on current inputs and past inputs.

We will perform analysis and optimization of DSP applications at the linear state-space level. We choose this representation because it models a wide class of applications or parts of applications, and it is simple to work with.

Our work with state-space representations will be done in the context of StreamIt, a programming language designed for streaming applications [17]. StreamIt allows users to create their own blocks, but limits the way these blocks can be connected. We perform the following steps on a StreamIt program:

1. Examine each block and determine whether or not it can be characterized as linear state-space. If it can, extract the appropriate state-space representation.

2. Combine connected blocks that each have a state-space representation, using an appropriate set of rules depending on the type of connection.

3. Optimize representations through the use of state-space transformations.

4. Convert the state-space representation(s) back to StreamIt code.

1.3 Organization

The rest of this thesis is organized as follows. In Chapter 2 we provide background information about StreamIt and formal linear and linear state-space models. Chapter 3 is devoted to state-space analysis of StreamIt programs (Items 1, 2, and 4). Chapter

4 describes optimizations (Item 3). In Chapter 5 we discuss our implementation and results. Chapter 6 details related work. In Chapter 7 we provide our conclusions and list possible future work.

Chapter 2

Background Information

2.1 The StreamIt Programming Language

StreamIt is a programming language specifically tailored to DSP streaming applications. The user creates a graph composed of four types of StreamIt constructs: *filters*, *pipelines*, *splitjoins*, and *feedback loops*. Filters encapsulate the computation done within an application - they represent the blocks mentioned in the previous chapter. Each filter operates on a one-dimensional 'tape' of values (of any type, including structures and arrays). The other three constructs dictate the type of connections possible between filters. Every construct explicitly states its input type and output type, and can be passed parameters as would be to a procedure.

StreamIt uses a buffer between every pair of filters to hold values. When the input buffer of a construct (equivalent to the output buffer of the previous construct) is appropriately filled, the construct can execute. Execution involves three steps: reading and removing items from the input buffer (consumption); performing computations; putting items in the output buffer (production). We will not consider the intricacies of managing these buffers, and instead refer to the more abstract notion of a tape.

A filter has pre-defined *peek*, *pop*, and *push* rates (StreamIt code examples are given below). During each execution, the filter accesses a maximum of peek values from its input tape, consumes exactly pop input values from its input tape, and produces exactly push values onto its output tape. Since the removal of an input value is technically an access of that input, the peek rate of a filter must be greater than or equal to the pop rate of that filter. The push or pop rate can be zero - the former corresponds to a filter that consumes items but does not produce them (typically the last filter in a sequence) and the latter corresponds to a filter that produces items but does not consume them (typically the first filter in a sequence). All the accesses, outputs, and removals, as well as all the computation is done inside the main body of the filter, known as the work function.

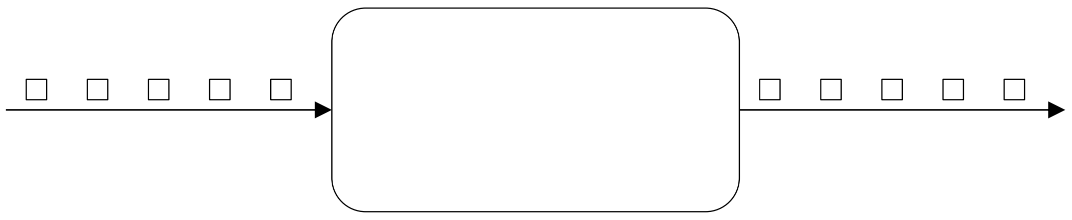

Figure 2-1: StreamIt filter

StreamIt also supports a *prework* function, which has its own push, pop, and peek rates. The prework function executes in place of the work function for the first computation sequence, and is never run again. Additionally, there is an *init* function which is run only once upon creation of the filter, and is usually used to initialize variables. The init and prework functions are both optional.

A filter can store two types of variables - *field* and *local*. Field variables are declared outside of the specific functions (work, prework, init), and can be accessed from anywhere within the filter. Local variables are declared within a specific function, and only have scope within that function. For example, a variable declared within the init function is local, and could not be accessed within the work function. Therefore, the init function is used to initialize field variables.

Code examples of StreamIt filters are shown below.

```
// This filter adds the parameter scalar to each input.
// It does not have an init or prework function
float -> float filter scalarAdd(float scalar) {
  work push 1 pop 1 peek 1 {
    push(scalar + pop());
  }
}

// This filter outputs a running average of every three consecutive inputs.
// The first time it runs, it ouputs the average of the first two inputs without removing anything from the tape.
// It does not have an init function.
float -> float filter threeWayAverage() {
  prework push 1 pop 0 peek 2 {
    float temp;  // example of a local variable
    temp = (peek(0)+peek(1))/2;
    push(temp);
  }
  work push 1 pop 1 peek 3 {
    float temp;  // example of a local variable
    temp = (peek(0) + peek(1) + peek(2))/3
    push(temp);
    pop()
  }
}

// This filter computes an infinite impulse response function.
// It does not have a prework function.
float->float filter IIR() {
    float curr;  // example of a field variable
    init {
      curr = 0;
    }
    work push 1 pop 1 peek 3 {
      float temp;  \\ example of a local variable
      temp = (peek(0) + peek(1) + peek(2))/6;
      curr = temp + curr/2;
      push(curr);
      pop();
    }
}
```

Pipelines, splitjoins, and feedback loops are higher level constructs created from filters. Each structures the layout of its filters in a certain format. Even though these three constructs do not directly provide the syntax to perform computations and work from an input or output tape, they can be thought of as filters in the following way: the construct recieves inputs which are passed to one or more of the filters; all the filters perform computations and pass values to one another through their input and output tapes; the construct outputs values from one or more of its filters. In fact, for every pipeline, splitjoin, and feedback loop there is an equivalent filter representation. Therefore, these three constructs are not strictly necessary for writing a StreamIt program. However, they simplify and structure writing a large application.

The higher level constructs are not limited to combine filters - they can also combine each other. This follows directly from the fact that a higher level construct has some equivalent filter. Therefore, if a pipeline can be composed of filters, it can also be composed of pipelines, splitjoins, and feedback loops, which are all like filters. We shall refer to all four StreamIt constructs generically as blocks. This corresponds to the fact that any StreamIt construct behaves as a block: it takes inputs, performs calculations, and produces outputs.

Pipelines combine a set of blocks in sequential fashion, so that the output of the first block is the input to the second block, the output of the second block is the input to the third block, etc. The blocks are placed in order using the add statement.

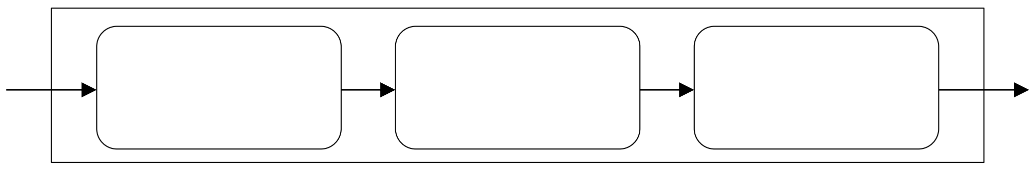

Figure 2-2: StreamIt pipeline

```
// This pipeline connects the filters scalarAdd and threeWayAverage.
// The parameter scalar passed to this pipeline is passed to the
// filter scalarAdd.
float -> float pipeline combinedWork(float scalar) {
  add scalarAdd(scalar);
  add threeWayAverage();
}
```

A splitjoin arranges blocks in a parallel fashion. The inputs to a splitjoin are sent to each block in a *roundrobin* or *duplicate* manner, and the outputs of each block are joined in a roundrobin manner. Duplicate splitting means the inputs to the splitjoin are copied and sent to each block, so that each block receives exactly the same set of inputs. Roundrobin splitting means the inputs to the splitjoin are sent to each block according to user defined weights. For example, the first block receives two inputs, the second block receives one input, the third block receives two inputs. Therefore, each block sees a different set of inputs. Roundrobin joining (the only type of joining permitted) means the outputs of each block are combined according to user defined weights, and these represent the outputs of the entire splitjoin. Blocks are listed in the order which they recieve inputs using add statements. The way inputs are sent is determined by using the split statement before the block list, and the way outputs are recieved is determined by using the join statement after the block list.

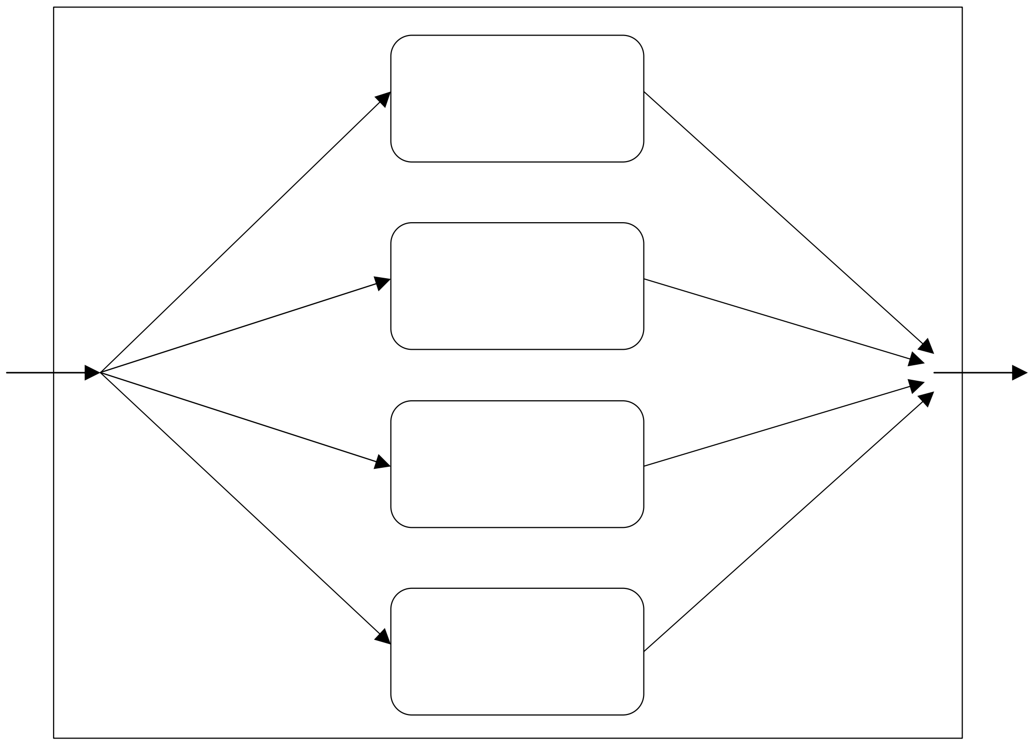

Figure 2-3: StreamIt splitjoin

```
// This splitjoin splits its inputs three ways.
// The first two inputs are sent to the first block, the next
// input to the second block, and the next two inputs to the third
// block.
// The outputs are collected in the following manner: three from
//the first block, five from the second block, and four from the
```

```
// third block.
// For every 2+1+2=5 values inputted, 3+5+4=12 values are
// outputted.
float -> float splitjoin mySplitjoin() {
 split roundrobin(2,1,2);
  add combinedWork(3.5);
  add combinedWork(4.5);
  add threeWayAverage();
join roundrobin(3,5,4);
}
```

A feedback loop uses some of its output as an input. It consists of a body block and a loop block. The input to the entire feedback loop is combined with the output of the loop block and sent to the body block, via a roundrobin joiner. The output of the body block is split two ways in a roundrobin or duplicate manner. The first set of outputs is used as the output of the entire feedback loop, and the second set of outputs is used as the input to the loop block. Note that there must be initial values enqueued on the output tape of the loop block in order for the feedback loop to begin executing. The first statement in a feedback loop is a join, determining how inputs are sent to the body block. The body and loop blocks are listed next. The last statement is a split, determining where outputs are sent from the loop block.

```
// This is a feedback loop implementation of the IIR filter.
// The body and loop are both anonymous filters.
float -> float feedbackloop IIRFeedback() {
  join roundrobin(3,1);
  body float->float filter {
    work push 1 pop 1 peek 4 {
      push((peek(0)+peek(1)+peek(2))/6 + peek(3)/2);
      pop();
    }
  };
  loop float->float filter {
    work push 1 pop 1 peek 1 {
      push(pop());
    }
  };
  split duplicate();
  enqueue(0.0);
}
```

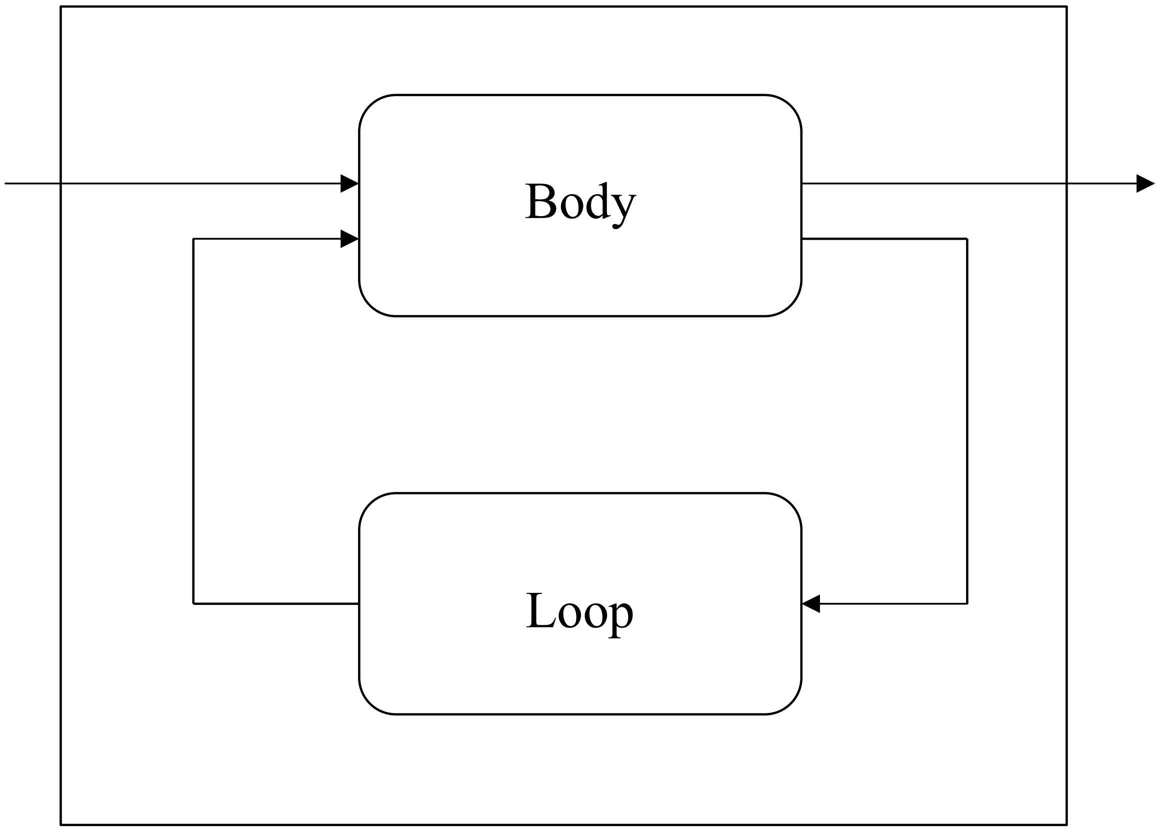

Figure 2-4: StreamIt feedback loop

To run a program, the StreamIt compiler finds a steady-state schedule of the number of times to execute each filter [10]. If such a schedule cannot be found, the user created block diagram is ill-formed and could not represent a real world application.

2.2 Block Representations

The execution of a block (StreamIt or otherwise) can by characterized by a single equation if the block is linear, and a pair of equations if the block is state-space linear. We describe these terms in detail below.

2.2.1 Linear Representations

A block is termed linear if its outputs are a linear combination of its inputs plus a set of constants. In mathematical terms, this relationship can be modelled by the equation $\vec{\mathbf{y}} = \mathbf{D}\vec{\mathbf{u}} + \vec{\mathbf{b}}$, where $\vec{\mathbf{u}}$ is a column vector representing the inputs, $\mathbf{D}$ is a matrix representing the weights applied to each input, $\vec{\mathbf{b}}$ is a column vector representing constants added to the inputs, and $\vec{\mathbf{y}}$ is a column vector representing the outputs.

Suppose we have the following linear model:

$$\vec{\mathbf{y}} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \vec{\mathbf{u}} + \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix}$$

It is exactly described by the following StreamIt filter:

```
int -> int filter linearFilter() {
  work push 3 pop 2 peek 2 {
    push(1*peek(0) + 2*peek(1) + 7);
    push(3*peek(0) + 4*peek(1) + 8);
    push(5*peek(0) + 6*peek(1) + 9);
    pop(); pop();
  }
}
```

A process for analyzing and optimizing linear StreamIt filters is described in [11].

2.2.2 State-Space Representations

A more general way of representing a block is by a state-space model. A set of variables captures the 'state' of the filter, so that the output is a combination of these variables (termed state variables) and the inputs. Additionally, the states themselves change upon every execution of the block. This is represented by the two equations:

$$\begin{aligned} \vec{\mathbf{y}} &= g(\vec{\mathbf{x}}, \vec{\mathbf{u}}) \\ \dot{\vec{\mathbf{x}}} &= f(\vec{\mathbf{x}}, \vec{\mathbf{u}}) \end{aligned}$$

The state vector is denoted by $\vec{\mathbf{x}}$, the inputs by $\vec{\mathbf{u}}$, and the outputs by $\vec{\mathbf{y}}$. $\dot{\vec{\mathbf{x}}}$ represents the new state vector, i.e. the state vector after it is updated. The first equation is for the outputs, the second equation is for the state updates.

A linear state-space model has the additional property that the state updates and outputs are linear in the state variables and inputs. We can use a simpler set of equations:

$$\begin{aligned} \vec{\mathbf{y}} &= \mathbf{C}\vec{\mathbf{x}} + \mathbf{D}\vec{\mathbf{u}} \\ \dot{\vec{\mathbf{x}}} &= \mathbf{A}\vec{\mathbf{x}} + \mathbf{B}\vec{\mathbf{u}} \end{aligned}$$

A, **B**, **C**, and **D** are matrices whose dimensions depend on the number of states, inputs, and outputs. Not all blocks can be represented by a linear state-space model. However, a linear state-space model is more general than a linear model, so a wider class of blocks can be represented. We will not discuss general state-space models any further in this paper, therefore we will write state-space instead of linear state-space for conciseness.

Suppose we have the following state-space model:

$$\begin{aligned} \vec{\mathbf{y}} &= \begin{bmatrix} 11 & 12 \end{bmatrix} \vec{\mathbf{x}} + \begin{bmatrix} 13 & 14 & 15 \end{bmatrix} \vec{\mathbf{u}} \\ \dot{\vec{\mathbf{x}}} &= \begin{bmatrix} 1 & 2 \\ 6 & 7 \end{bmatrix} \vec{\mathbf{x}} + \begin{bmatrix} 3 & 4 & 5 \\ 8 & 9 & 10 \end{bmatrix} \vec{\mathbf{u}} \end{aligned}$$

It is exactly described by the following StreamIt filter:

```
int -> int filter stateSpaceFilter() {
  int x1, x2;
  work push 1 pop 3 peek 3 {
    int x1_temp, x2_temp;
    push(11*x1 + 12*x2 + 13*peek(0) + 14*peek(1) + 15*peek(2));
    x1_temp = 1*x1 + 2*x2 + 3*peek(0) + 4*peek(1) + 5*peek(2);
    x2_temp = 6*x1 + 7*x2 + 8*peek(0) + 9*peek(1) + 10*peek(2);
    x1 = x1_temp;
    x2 = x2_temp;
    pop(); pop(); pop();
  }
}
```

Note we introduced two extra variables - $x1\_temp$ and $x2\_temp$. We do this because we do not want to overwrite the old values for $x1$ and $x2$ until all the new values are calculated. Also, we have made no provisions for constants as in the linear model. This issue is resolved in the next chapter.

Chapter 3

State-Space Analysis

We analyze StreamIt programs at the filter level. We create a data structure representation that fully describes a state-space filter. We parse the code of each StreamIt filter to determine whether or not it is state-space; if so we initialize a data structure, fill it with the appropriate values through a process called *extraction*, and associate the structure with the filter.

We provide a set of rules to combine state-space representations of filters in higher StreamIt blocks—pipelines, splitjoins, and feedback loops. Such a process results in a single state-space representation for the entire block. Some representations may need to change so that they are properly combined. We detail what the changes are and when they need to be made. Finally, we describe how to convert a representation back to StreamIt code for a filter.

3.1 Representation

Our first task is to create a data structure that fully captures the state-space representation of a StreamIt filter. We save a filter's number of states, push rate, and pop rate in variables which we term s, u, and o, respectively. Our data structure also contains the matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$ with dimensions $s \times s$, $s \times o$, $u \times s$, and $u \times o$, respectively. The inputs to a filter are denoted as $\vec{\mathbf{u}}$ (length o), the outputs as $\vec{\mathbf{y}}$ (length u), and the states as $\vec{\mathbf{x}}$ (length s). Upon every execution of the filter, we can calculate the outputs by the formula $\vec{\mathbf{y}} = \mathbf{C}\vec{\mathbf{x}} + \mathbf{D}\vec{\mathbf{u}}$, and update the state matrix by the formula $\vec{\dot{\mathbf{x}}} = \mathbf{A}\vec{\mathbf{x}} + \mathbf{B}\vec{\mathbf{u}}$. For convenience, we will calculate the filter outputs before updating the state matrix. Since the states may have initial values other than zero, we store these values as the vector $\overrightarrow{\mathbf{initVec}}$ (length s).

Since we have not included a constant term in our model, we will set one of the state variables to be the constant 1. This variable will not be updated by any of the states or inputs, and its initial value will be 1, so it will always remain that value. Any state or output that depends on a constant term can now refer to a multiple of the constant state variable instead.

As long as a filter's peek rate (which we term e) equals its pop rate, the data structure as currently designed can fully represent the filter. We must include additional modifications for a filter with a peek rate greater than its pop rate. Note that such a filter still removes o items from its input tape upon every execution, but it accesses $e - o$ additional items on its input tape. Therefore, our current data structure would work as long as there is some way to access these additional items.

We solve the problem of having a peek rate greater than a pop rate by storing $e-o$ items from the input tape in the state vector $\vec{\mathbf{x}}$. Therefore, when a filter executes, it can access all e items it needs, o items from its input vector and $e - o$ items from its state vector. These $e - o$ states must be updated by the inputs and themselves - the specifics are covered in the next section. We store the number of states used for inputs as the variable *stored*. This will be useful when combining representations.

When the filter is executed for the first time, it will have access to the o items in the input vector, but the $e - o$ states it needs will be uninitialized from the input tape. Therefore, we need to update the state vector before computing the output/state update equation pair for every filter execution. We introduce two new matrices, $\mathbf{A_{pre}}$ and $\mathbf{B_{pre}}$ to perform this initialization. Before the filter runs it will perform the state update $\vec{\dot{\mathbf{x}}} = \mathbf{A_{pre}}\vec{\mathbf{x}} + \mathbf{B_{pre}}\vec{\mathbf{u_{pre}}}$. The initialization input vector, $\vec{\mathbf{u_{pre}}}$, has length $o_{pre} = e - o$. For now, o_{pre} and *stored* have the same value, but combining filters might result in o_{pre} being greater than *stored*. $\mathbf{A_{pre}}$ is $s \times s$ and $\mathbf{B_{pre}}$ is $s \times o_{pre}$. Note that initial assignments of the state variables by $\overrightarrow{\mathbf{initVec}}$ are done immediately when a filter is created, while initialization by $\mathbf{A_{pre}}$ and $\mathbf{B_{pre}}$ is afterwards, when there are a sufficient number (o_{pre}) of items on the input tape.

Putting these pieces together, we find a full representation consists of the push and pop rates, the number of state variables, the number of stored inputs, the four state matrices, an initial state vector, and possibly an initial pop rate and two initialization state matrices. We define a state-space representation R as the tuple $\langle u$, o, s, *stored*, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$, $\overrightarrow{\mathbf{initVec}}$, $\mathbf{A_{pre}}$, $\mathbf{B_{pre}}$, $o_{pre}\rangle$. When we introduce a representation $\mathrm{R_i}$, each of its values in the ordered set will be denoted with the index i (for example u_i, $\mathbf{A_i}$). For representations of filters that do not need the initialization matrices, we will write $\mathbf{A_{pre}} = null$, $\mathbf{B_{pre}} = null$, $o_{pre} = 0$. In this case, the filter will not have any stored inputs, so $stored = 0$ as well.

Representations are initially created from StreamIt filters and ultimately converted to StreamIt filters. Between these steps, however, representations of the higher StreamIt block types can be derived by combining the representations of their parts. Therefore, from now on we will say that a representation refers to a block rather than a filter. The exception is in Section 3.2, where we discuss how to create a representation from a StreamIt filter. Hence we explicitly refer to a filter rather than block representation in that section.

3.2 Extraction

We write a module that extracts the state-space representation of a filter. We symbolically execute a single iteration of a filter's work function, maintaining a vector pair representation for each local variable and filter field variable that is encountered (combined, these are termed program variables). If the outputs and field variables all have vector pair representations, then the filter is state-space linear, and the vectors are used as rows of $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$. This type of procedure is termed data flow analysis. See [11] for a treatment of the linear case.

We attempt to find a vector pair ($\vec{\mathbf{v}}$,$\vec{\mathbf{w}}$) for each program variable y where $y = \vec{\mathbf{v}} \cdot \vec{\mathbf{u}} + \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$. $\vec{\mathbf{u}}$ is the filter input vector and $\vec{\mathbf{x}}$ is the filter state vector. When y is on the left hand side of an assignment statement, terms from the right hand side are compared with entries from $\vec{\mathbf{u}}$ (inputs) and $\vec{\mathbf{x}}$ (states). The coefficients from terms that match are used for to fill the corresponding entries in $\vec{\mathbf{v}}$ and $\vec{\mathbf{w}}$, as long as they are constants. If any coefficient is not a constant, then y is non-linear.

The input vector, $\vec{\mathbf{u}}$, is defined as $[peek(e-o)\ peek(e-o+1)\ ...\ peek(o-1)]$. The state vector, $\vec{\mathbf{x}}$, holds $e-o$ variables from the input tape ($peek(0)\ ...\ peek(e-o-1)$), every field variable, and a variable for the constant 1. We do not consider local variables for the state vector, because their values are not saved across filter executions. Therefore, their values should be resolved to constants at compile time. A field variable has the initial vector pair $(\vec{\mathbf{0}}, \begin{bmatrix} 0 & ... & 1 & ... & 0 \end{bmatrix})$, where the 1 corresponds to the field variable itself.

If the vector pair can be found, then the program variable y can be written as a linear combination of the inputs and state variables, with the vector pair entries representing the weights. Then the final assignment to state variable x_i by some program variable y_i indicates that the i^{th} rows of $\mathbf{A}$ and $\mathbf{B}$ should be $\vec{\mathbf{w_i}}$ and $\vec{\mathbf{v_i}}$, respectively. Similarly, the j^{th} push statement using program variable y_j indicates that the j^{th} rows of $\mathbf{C}$ and $\mathbf{D}$ should be $\vec{\mathbf{w_j}}$ and $\vec{\mathbf{v_j}}$, respectively. For the constant state variable 1, the corresponding rows of $\mathbf{A}$ and $\mathbf{B}$ are all zeros.

We use the same procedure in the init function to find the initial values for each field variable. However, we do not need a vector $\vec{\mathbf{v}}$ for the inputs, since there are no inputs to the init function. The initial value for each stored inputs is zero, and the initial value for the variable 1 is one.

Finally, consider the stored input states (call them $\vec{\mathbf{x_s}}$). They are updated by the inputs; however if $stored > o$, then some of the input states must be updated by other input states. In particular, the first $stored - o$ input states are updated by the last $stored - o$ inputs, and the remaining o input states are updated by the o inputs. The update is described by the equation:

$$\dot{\vec{\mathbf{x_s}}} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \vec{\mathbf{x_s}} + \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} \vec{\mathbf{u}} \tag{3.1}$$

We also create initialization matrices to put values from the input tape into the input states:

$$\dot{\vec{\mathbf{x_s}}} = \mathbf{0}\vec{\mathbf{x_s}} + \mathbf{I}\vec{\mathbf{u_{pre}}}$$

Stored inputs are always updated as shown in the same manner. Therefore, we will use $\mathbf{A_s}$ and $\mathbf{B_s}$ to describe this update, where the values of these two matrices are shown in (3.1).

3.2.1 Example Procedure

Consider the IIR filter from Chapter 2:

```
// This filter computes an infinite impulse response function.
// It does not have a prework function.
float->float filter IIR() {
    float curr;  // example of a field variable
    init {
      curr = 0;
    }
    work push 1 pop 1 peek 3 {
      float temp; // example of a local variable
      temp = (peek(0) + peek(1) + peek(2))/6;
      curr = temp + curr/2;
      push(curr);
      pop();
    }
}
```

The input vector is $\begin{bmatrix} peek(2) \end{bmatrix}$ and the state vector is $\begin{bmatrix} peek(0) \\ peek(1) \\ curr \\ 1 \end{bmatrix}$. The first program variable encountered is *temp*. It is given the vector pair $(\begin{bmatrix} 1/6 \end{bmatrix}, \begin{bmatrix} 1/6 & 1/6 & 0 & 0 \end{bmatrix})$. The variable *curr*, as a state variable, has an initial vector pair: $(\begin{bmatrix} 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix})$. When *curr* is found in an assignment statement, it is given a new vector pair, constructed as 1 times the vector pair for *temp* plus 1/2 times the old vector pair for *curr*: $(\begin{bmatrix} 1/6 \end{bmatrix}, \begin{bmatrix} 1/6 & 1/6 & 1/2 & 0 \end{bmatrix})$. The output is *curr*, so it is given the same vector pair. The final pair for *curr* represents its state update. The stored inputs $peek(0), peek(1)$ are updated as mentioned in (3.1), and the constant 1 is not updated. Therefore, we have:

$$
\begin{aligned}
\mathbf{A} &= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1/6 & 1/6 & 1/2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \\
\mathbf{B} &= \begin{bmatrix} 0 \\ 1 \\ 1/6 \\ 0 \end{bmatrix} \\
\mathbf{C} &= \begin{bmatrix} 1/6 & 1/6 & 1/2 & 0 \end{bmatrix} \\
\mathbf{D} &= \begin{bmatrix} 1/6 \end{bmatrix} \\
\overrightarrow{\mathbf{initVec}} &= \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \\
\mathbf{A_{pre}} &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\mathbf{B_{pre}} &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}
\end{aligned}
$$

The pop and push rates are both one, and we have four states so $o = 1, u = 1, s = 4$. We have two stored input states, so $o_{pre} = 2, stored = 2$.

3.3 Combination

If all blocks within a pipeline, splitjoin, or feedback loop have state-space representations, we can combine them into a single representation using the rules developed in this section. We combine blocks for two reasons. One reason is that it is easier to optimize a single block than multiple blocks. The second reason is that we may eliminate redundant computations across blocks.

3.3.1 Pipeline

Consider two blocks connected in a pipeline with representations R_1 and R_2. Let R denote the combined representation of the two blocks, which we are trying to derive. Suppose the output rate of R_1 equals the input rate of R_2 ($u_1 = o_2$). If this is not the case, we must expand one or both blocks to have their input/output rates match ($u_{1\_new} = o_{2\_new} = lcm(u_1, o_2)$). Block expansion is covered in Section 3.4.1. Since the output of R_1 (y_1) is equivalent to the input of R_2 (u_2), we can write:

$$\begin{aligned}
\dot{\vec{\mathbf{x_1}}} &= \mathbf{A_1}\vec{\mathbf{x_1}} + \mathbf{B_1}\vec{\mathbf{u_1}} \\
\dot{\vec{\mathbf{x_2}}} &= \mathbf{A_2}\vec{\mathbf{x_2}} + \mathbf{B_2}\vec{\mathbf{y_1}}
\end{aligned}$$

$$\begin{aligned}
\vec{\mathbf{y_1}} &= \mathbf{C_1}\vec{\mathbf{x_1}} + \mathbf{D_1}\vec{\mathbf{u_1}} \\
\vec{\mathbf{y_2}} &= \mathbf{C_2}\vec{\mathbf{x_2}} + \mathbf{D_2}\vec{\mathbf{y_1}}
\end{aligned}$$

Substituting for $\vec{\mathbf{y_1}}$ we get:

$$\begin{aligned}
\dot{\vec{\mathbf{x_2}}} &= \mathbf{A_2}\vec{\mathbf{x_2}} + \mathbf{B_2}(\mathbf{C_1}\vec{\mathbf{x_1}} + \mathbf{D_1}\vec{\mathbf{u_1}}) \\
\vec{\mathbf{y_2}} &= \mathbf{C_2}\vec{\mathbf{x_2}} + \mathbf{D_2}(\mathbf{C_1}\vec{\mathbf{x_1}} + \mathbf{D_1}\vec{\mathbf{u_1}})
\end{aligned}$$

Which simplifies to:

$$\begin{aligned}
\dot{\vec{\mathbf{x_2}}} &= \mathbf{A_2}\vec{\mathbf{x_2}} + \mathbf{B_2}\mathbf{C_1}\vec{\mathbf{x_1}} + \mathbf{B_2}\mathbf{D_1}\vec{\mathbf{u_1}} \\
\vec{\mathbf{y_2}} &= \mathbf{C_2}\vec{\mathbf{x_2}} + \mathbf{D_2}\mathbf{C_1}\vec{\mathbf{x_1}} + \mathbf{D_2}\mathbf{D_1}\vec{\mathbf{u_1}}
\end{aligned}$$

Let $\vec{\mathbf{x}} = \begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \end{bmatrix}$, $\vec{\mathbf{u}} = \vec{\mathbf{u_1}}$ (the input to the entire pipeline), and $\vec{\mathbf{y}} = \vec{\mathbf{y_2}}$ (the output of the entire pipeline). The equations relating $\vec{\mathbf{x}}$, $\vec{\mathbf{u}}$, and $\vec{\mathbf{y}}$ are:

$$\begin{aligned}
\dot{\vec{\mathbf{x}}} &= \mathbf{A}\vec{\mathbf{x}} + \mathbf{B}\vec{\mathbf{u}} \\
\vec{\mathbf{y}} &= \mathbf{C}\vec{\mathbf{x}} + \mathbf{D}\vec{\mathbf{u}} \\
\mathbf{A} &= \begin{bmatrix} \mathbf{A_1} & \mathbf{0} \\ \mathbf{B_2}\mathbf{C_1} & \mathbf{A_2} \end{bmatrix} \\
\mathbf{B} &= \begin{bmatrix} \mathbf{B_1} \\ \mathbf{B_2}\mathbf{D_1} \end{bmatrix} \\
\mathbf{C} &= \begin{bmatrix} \mathbf{D_2}\mathbf{C_1} & \mathbf{C_2} \end{bmatrix} \\
\mathbf{D} &= \mathbf{D_2}\mathbf{D_1}
\end{aligned}$$

The input to the pipeline is identical to the input to R$_1$, and the output of the pipeline is identical to the output of R$_2$. Furthermore, the states of the pipeline are the states of the first block appended to the states of the second block. Therefore, $u = u_2$, $o = o_1$, $s = s_1 + s_2$, $\overrightarrow{\mathbf{initVec}} = \begin{bmatrix} \overrightarrow{initVec_1} \\ \overrightarrow{initVec_2} \end{bmatrix}$.

If both blocks do not have initialization matrices, then the entire pipeline does not need initialization matrices, so $\mathbf{A_{pre}} = null$, $\mathbf{B_{pre}} = null$, $o_{pre} = 0$, $stored = 0$. If only the first block has initialization matrices, then we want to initialize the states in the pipeline corresponding to the first block while keeping the states corresponding to the second block unchanged. Therefore:

$$\begin{aligned}
\mathbf{A_{pre}} &= \begin{bmatrix} \mathbf{A_{pre1}} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \\
\mathbf{B_{pre}} &= \begin{bmatrix} \mathbf{B_{pre1}} \\ \mathbf{0} \end{bmatrix} \\
o_{pre} &= o_{pre1} \\
stored &= stored_1
\end{aligned}$$

If the second block has initialization matrices, we must run the first block enough times to provide the necessary inputs to initialize the second block. However, this might result in the first block providing extra initial inputs to the second block. In that case, we must change the representation of the second block to increase its number of stored inputs (the way to do this is covered in Section 3.4.2). Suppose this is done and the first block must run n times (along with its initialization matrices, if it has them) to initialize the second block. Denote $\mathbf{A_1}^\mathbf{e}$, $\mathbf{B_1}^\mathbf{e}$, $\mathbf{C_1}^\mathbf{e}$, and $\mathbf{D_1}^\mathbf{e}$ as the matrices that describe running the first block n times (see Equations (3.6)-(3.9)). Then the initialization of the entire pipeline is derived by combining these matrices with $\mathbf{A_{pre2}}$, $\mathbf{B_{pre2}}$ just as the $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$ matrices are combined for the two blocks:

$$\begin{aligned}
\mathbf{A_{pre}} &= \begin{bmatrix} \mathbf{A_1}^\mathbf{e} & \mathbf{0} \\ \mathbf{B_{pre2}}\mathbf{C_1}^\mathbf{e} & \mathbf{A_{pre2}} \end{bmatrix} \\
\mathbf{B_{pre}} &= \begin{bmatrix} \mathbf{B_1}^\mathbf{e} \\ \mathbf{B_{pre2}}\mathbf{D_1}^\mathbf{e} \end{bmatrix} \\
o_{pre} &= o_{pre1} + n * o_1 \\
stored &= stored_1
\end{aligned}$$

If there are more than two blocks in a pipeline, we collapse the pipeline in the following manner: combine the first two blocks to get one block representation, combine this with the third block, etc.

3.3.2 Splitjoin

There are two types of splitjoins - those with roundrobin and duplicate splitters. In order to collapse the branches of a splitjoin to a single representation, we need the splitjoin to have a duplicate splitter, because then the representation in each branch accesses the same inputs. Therefore, for roundrobin splitjoins we first detail a procedure to convert to a duplicate splitjoin. Then we describe how to create a representation of a duplicate splitjoin.

Conversion from roundrobin to duplicate splitjoin

Suppose the roundrobin splitjoin has k branches and let w_i and $\mathrm{M_i}$ denote the splitter weight and state-space representation, respectively, on the i^{th} branch. In each branch i we add a filter with representation $\mathrm{L_i}$ that outputs to $\mathrm{M_i}$ in a pipeline format. Since the splitjoin now has a duplicate splitter, $\mathrm{M_i}$ receives every input element to the entire splitjoin. In order to exactly simulate the original roundrobin splitter, $\mathrm{M_i}$ should only see w_i elements for every $\sum_{j=1}^{k} w_j$ input elements to the splitjoin. Therefore, we make $\mathrm{L_i}$ input $\sum_{j=1}^{k} w_j$ elements and output w_i elements. In particular, $\mathrm{L_i}$ ignores the first $\sum_{j=1}^{i-1} w_j$ inputs (which correspond to inputs to the previous branches), outputs the next w_i inputs (which correspond to inputs to the i^{th} branch), and ignores the remaining $\sum_{j=i+1}^{k} w_j$ inputs[1] (which correspond to inputs to the later branches).

The values for $\mathrm{L_i}$ are $o = \sum_{j=1}^{k} w_j$, $u = w_i$, $s = 1$, $\mathbf{A} = \mathbf{0}$, $\mathbf{B} = \mathbf{0}$, $\mathbf{C} = \mathbf{0}$, $\mathbf{D} = \begin{bmatrix} \mathbf{0} & \mathbf{I} & \mathbf{0} \end{bmatrix}$, $\overrightarrow{\mathbf{initVec}} = \vec{\mathbf{0}}$, $\mathbf{A_{pre}}, \mathbf{B_{pre}} = null$, $o_{pre} = 0$, $stored = 0$. We use one state in the representation, even though none are needed, to make combinations of representations simpler. Once $\mathrm{L_i}$ is created, it can be combined with $\mathrm{M_i}$ to form a single representation (call it $\mathrm{R_i}$).

Collapsing duplicate splitjoins

Let R be the representation for the entire splitjoin, $\mathrm{R_i}$ be the representation on the i^{th} branch, k be the number of branches. In order to combine the branch representations,

[1]In DSP terminology, $\mathrm{L_i}$ is called a downsampler.

we must derive a steady-state execution of the entire splitjoin. Denote the joiner weight of each branch i as w_i (note that we used w_i earlier to denote a splitter weight). Each branch outputs u_i items but w_i items from that branch are needed to execute the splitjoin once. $\mathrm{R_i}$ can be expanded to output $lcm(u_i, w_i)$ items, which would result in $\frac{lcm(u_i,w_i)}{w_i}$ splitjoin executions. This means we must execute the splitjoin a multiple of $\frac{lcm(u_1,w_1)}{w_1}$ times to satisfy the constraints of the first branch, a multiple of $\frac{lcm(u_2,w_2)}{w_2}$ times to satisfy the constraints of the second branch, etc. Therefore, we shall construct R to execute the splitjoin $lcm(\frac{lcm(u_1,w_1)}{w_1}, \frac{lcm(u_2,w_2)}{w_2}, ..., \frac{lcm(u_k,w_k)}{w_k})$ times. Call this value E. Each representation $\mathrm{R_i}$ must output $w_i * E$ elements, so $\mathrm{R_i}$ must be expanded $\frac{w_i * E}{u_i}$ times.

After these expansions, each branch representation should now have the same input rate o_i. If not, the splitjoin is ill-formed and cannot be compiled by StreamIt. Since these representations will be combined, we need each to have the same number of stored inputs and the same initial pop rate. To satisfy both constraints, we increase the number of stored inputs in each representation to the value $max(stored_i, o_{prei})$ over all i.

Now that the branch representations have been standardized, they can be combined to a single representation. The stored input states in each representation evolve in the same manner, so only one set of them is needed for the entire splitjoin representation. Let $\vec{\mathbf{x_i}} = \begin{bmatrix} \vec{\mathbf{x_{is}}} \\ \vec{\mathbf{x_{ir}}} \end{bmatrix}$, where $\vec{\mathbf{x_{is}}}$ and $\vec{\mathbf{x_{ir}}}$ are the stored input states and remaining states of $\mathrm{R_i}$, respectively. For each representation i denote the state-space equation pair as:

$$\begin{aligned} \begin{bmatrix} \dot{\vec{\mathbf{x_{is}}}} \\ \dot{\vec{\mathbf{x_{ir}}}} \end{bmatrix} &= \begin{bmatrix} \mathbf{A_{is}} & \mathbf{0} \\ \mathbf{A_{irs}} & \mathbf{A_{irr}} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_{is}}} \\ \vec{\mathbf{x_{ir}}} \end{bmatrix} + \begin{bmatrix} \mathbf{B_{is}} \\ \mathbf{B_{ir}} \end{bmatrix} \vec{\mathbf{u}} \\ \vec{\mathbf{y}} &= \begin{bmatrix} \mathbf{C_{is}} & \mathbf{C_{ir}} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_{is}}} \\ \vec{\mathbf{x_{ir}}} \end{bmatrix} + \mathbf{D}\vec{\mathbf{u}} \end{aligned}$$

Since the stored input states in each representation are equivalent, we set them to be $\vec{\mathbf{x_s}}$, and set their corresponding matrix blocks to be $\mathbf{A_s}$ and $\mathbf{B_s}$. Let $\vec{\mathbf{x}} = \begin{bmatrix} \vec{\mathbf{x_s}} \\ \vec{\mathbf{x_{1r}}} \\ \vec{\mathbf{x_{2r}}} \\ \ldots \\ \vec{\mathbf{x_{kr}}} \end{bmatrix}$.

The states $\vec{\mathbf{x_{ir}}}$ evolve separately, so:

$$\begin{aligned}
\mathbf{A} &= \begin{bmatrix} \mathbf{A_s} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{A_{1rs}} & \mathbf{A_{1rr}} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{A_{2rs}} & \mathbf{0} & \mathbf{A_{2rr}} & \ldots & \mathbf{0} \\ \ldots & \ldots & \ldots & \ldots & \\ \mathbf{A_{krr}} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{A_{krs}} \end{bmatrix} \\
\mathbf{B} &= \begin{bmatrix} \mathbf{B_s} \\ \mathbf{B_{1r}} \\ \mathbf{B_{2r}} \\ \ldots \\ \mathbf{B_{kr}} \end{bmatrix}
\end{aligned}$$

Similarly for the initialization matrices we have:

$$\begin{aligned}
\mathbf{A_{pre}} &= \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \mathbf{A_{pre1rr}} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A_{pre2rr}} & \ldots & \mathbf{0} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{A_{prekrr}} \end{bmatrix} \\
\mathbf{B_{pre}} &= \begin{bmatrix} \mathbf{B_{pres}} \\ \mathbf{B_{pre1r}} \\ \mathbf{B_{pre2r}} \\ \ldots \\ \mathbf{B_{prekr}} \end{bmatrix}
\end{aligned}$$

These equations are simpler because $\mathbf{A_{pres}} = \mathbf{0}$ and $\mathbf{A_{preirs}} = \mathbf{0}$.

In order to simulate the roundrobin nature of the joiner, we must output w_1 items from R$_1$, then w_2 items from R$_2$, up to w_k items from R$_\mathrm{k}$, and repeat this process E times (because we are running the splitjoin E times). Let $\mathbf{C_i} = \begin{bmatrix} \mathbf{C_{is1}} & \mathbf{C_{ir1}} \\ \mathbf{C_{is2}} & \mathbf{C_{ir2}} \\ \ldots & \ldots \\ \mathbf{C_{isexecutions}} & \mathbf{C_{irexecutions}} \end{bmatrix}$, where $\begin{bmatrix} \mathbf{C_{isj}} & \mathbf{C_{irj}} \end{bmatrix}$ is $w_i \times s_i$. Let $\mathbf{D_i} = \begin{bmatrix} \mathbf{D_{i1}} \\ \mathbf{D_{i2}} \\ \ldots \\ \mathbf{D_{iexecutions}} \end{bmatrix}$, where $\mathbf{D_{ij}}$ is $w_i \times o$. Then we have:

$$\mathbf{C} = \begin{bmatrix} \mathbf{C_{1s1}} & \mathbf{C_{1r1}} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{C_{2s1}} & \mathbf{C_{2r1}} & \mathbf{0} & \ldots & \mathbf{0} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \mathbf{C_{ks1}} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{C_{kr1}} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \mathbf{C_{1sk}} & \mathbf{C_{1rk}} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{C_{2sk}} & \mathbf{C_{2rk}} & \mathbf{0} & \ldots & \mathbf{0} \\ \ldots & \ldots & \ldots & \ldots & \\ \mathbf{C_{ksk}} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{C_{krk}} \end{bmatrix}$$

$$\mathbf{D} = \begin{bmatrix} \mathbf{D_{11}} \\ \mathbf{D_{21}} \\ \cdots \\ \mathbf{D_{k1}} \\ \cdots \\ \mathbf{D_{1k}} \\ \mathbf{D_{2k}} \\ \cdots \\ \mathbf{D_{kk}} \end{bmatrix}$$

We have derived $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$, $\mathbf{A_{pre}}$, and $\mathbf{B_{pre}}$. As mentioned previously, all the pop rates are equal so $o = o_1$. Additionally, all the initial pop rates and stored inputs are equal, so $o_{pre} = o_{pre1}$ and $stored = o_{pre1}$. The splitjoin runs E times, hence $u = E * \sum_{j=1}^{k} w_j$. The states of the entire representation are the non-stored input states of each branch representation concatenated along with one set of the stored input states. Let s_{ir} be the number of non-stored input states in representation i and let $\overrightarrow{\mathbf{initVec_{ir}}}$ be the initial values of these states. Then $s = stored + \sum_{j=1}^{k} s_{jr}$ and

$$\overrightarrow{\mathbf{initVec}} = \begin{bmatrix} \vec{0} \\ \overrightarrow{\mathbf{initVec_{1r}}} \\ \overrightarrow{\mathbf{initVec_{2r}}} \\ \cdots \\ \overrightarrow{\mathbf{initVec_{kr}}} \end{bmatrix}.$$

3.3.3 Feedback Loop

Recall that a feedback loop has a loop block and a body block. Outputs from the body block and inputs to the entire feedback loop are combined via a joiner to form the inputs to the loop block. Outputs from the loop block are used as outputs of the entire feedback loop and inputs of the loop block via a splitter.

Let the loop block have representation R_1, the body block have representation R_2, and the entire feedback loop have representation R. If the splitter is a roundrobin one, we convert it to a duplicate one by adding the appropriate downsamplers to the output branches, as described in Section 3.3.2. The output branches of a feedback loop splitter lead to the loop block and the output of the entire feedback loop. Therefore, one downsampler must be placed before the loop block in a pipeline format, and one downsampler must be placed after the feedback loop in a pipeline format. The first downsampler and loop block is combined to form a new loop block. The second downsampler can be combined with the feedback loop after the feedback loop's representation is computed.

As in the case of a splitjoin, we must derive a steady-state execution of the entire feedback loop in order to combine the loop and body blocks. First we match the output rate of the body block (o_2) with the input rate of the loop block (u_1) by expanding the two representations appropriately. Now consider the roundrobin joiner, and let w_1, w_2 be the weights on the branches from the loop block and input to the body block, respectively. The loop block outputs u_1 items, but w_1 items are needed to run the feedback loop once. Therefore, the loop block can be expanded to output $lcm(u_1, w_1)$ items, which would result in $\frac{lcm(u_1,w_1)}{w_1}$ feedback loop executions. Call this value E. The loop block is expanded to run $\frac{lcm(u_1,w_1)}{u_1}$ times, and the body block is expanded by this amount as well, since we still want the output rate of the body block to equal the input rate of the loop block. Since the feedback loop runs E times, the body block receives $E * (w_1 + w_2)$ inputs, which should equal the input rate of the expanded body block. If not, the feedback loop is ill-formed.

Once the above expansions are implemented, the feedback loop is run by executing the loop and body blocks alternately. However, the loop block depends on outputs from the body block, and the body block depends on outputs from the loop block. In order to begin execution of the entire feedback loop, there must be items enqueued on the output tape of the loop block. The minimal number of enqueued items is u_1, the output rate of the loop block. However, there can be more enqueued items. We create a new representation R_3 that stores the enqueued values. Upon each execution R_3 inputs u_1 items from the loop block and outputs u_1 items to the body block. It has one state for each enqueued item. The equations for R$_3$ are:

$$\begin{aligned}
\dot{\vec{\mathbf{x_3}}} &= \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \vec{\mathbf{x_3}} + \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} \vec{\mathbf{u_3}} \\
\vec{\mathbf{y_3}} &= \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \vec{\mathbf{x_1}}
\end{aligned}$$

R$_3$ does not have initialization matrices, and $\overrightarrow{\mathbf{initVec_3}}$ is assigned the enqueued values.

Note that the output $\vec{\mathbf{y_3}}$ does not depend on the input $\vec{\mathbf{u_3}}$. This is the key to starting the feedback loop: R$_3$ outputs first, the body block uses these outputs along with inputs to the entire feedback loop to execute and produce outputs, the loop body uses these outputs to execute and produce outputs, R$_3$ uses these outputs to execute and produce outputs, etc.

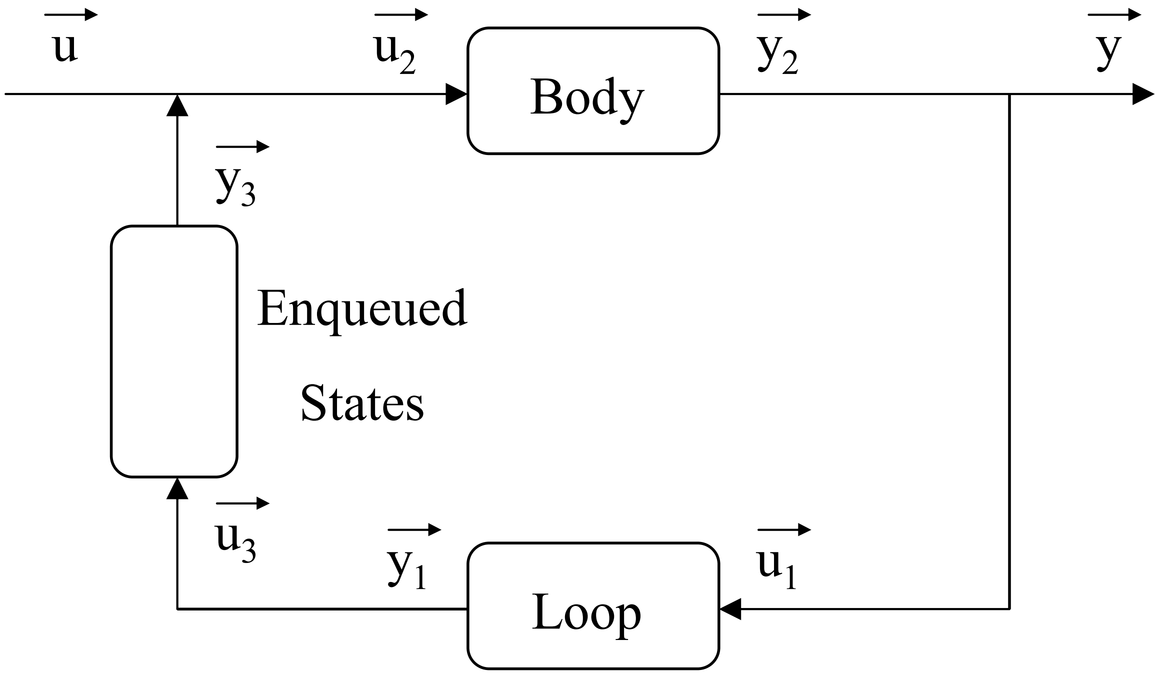

Figure 3-1: Labelled feedback loop

From figure 3-1 it is apparent that $\vec{\mathbf{u_3}} = \vec{\mathbf{y_1}}$, $\vec{\mathbf{y}} = \vec{\mathbf{y_2}} = \vec{\mathbf{u_1}}$, and $\vec{\mathbf{u_2}}$ is composed of

$\vec{\mathbf{u}}$ and $\vec{\mathbf{u_3}}$. We can write the equations for the body block as:

$$\begin{aligned}
\dot{\vec{\mathbf{x_2}}} &= \mathbf{A_2}\vec{\mathbf{x_2}} + \mathbf{B_2}\vec{\mathbf{u_2}} = \mathbf{A_2}\vec{\mathbf{x_2}} + \mathbf{B_{2\_1}}\vec{\mathbf{u}} + \mathbf{B_{2\_2}}\vec{\mathbf{y_3}} = \mathbf{A_2}\vec{\mathbf{x_2}} + \mathbf{B_{2\_1}}\vec{\mathbf{u}} + \mathbf{B_{2\_2}}\mathbf{C_3}\vec{\mathbf{x_3}} \\
\vec{\mathbf{y_2}} &= \mathbf{C_2}\vec{\mathbf{x_2}} + \mathbf{D_2}\vec{\mathbf{u_2}} = \mathbf{C_2}\vec{\mathbf{x_2}} + \mathbf{D_{2\_1}}\vec{\mathbf{u}} + \mathbf{D_{2\_2}}\vec{\mathbf{y_3}} = \mathbf{C_2}\vec{\mathbf{x_2}} + \mathbf{D_{2\_1}}\vec{\mathbf{u}} + \mathbf{D_{2\_2}}\mathbf{C_3}\vec{\mathbf{x_3}}
\end{aligned}$$

Since $\vec{\mathbf{y}} = \vec{\mathbf{y_2}}$, we have written the output of the feedback loop and the update for $\vec{\mathbf{x_2}}$ in terms of the input to the feedback loop and the state vectors. For the updates to $\vec{\mathbf{x_1}}$ and $\vec{\mathbf{x_3}}$ we can write:

$$\begin{aligned}
\dot{\vec{\mathbf{x_1}}} &= \mathbf{A_1}\vec{\mathbf{x_1}} + \mathbf{B_1}\vec{\mathbf{u_1}} = \mathbf{A_1}\vec{\mathbf{x_1}} + \mathbf{B_1}\vec{\mathbf{y}} = \mathbf{A_1}\vec{\mathbf{x_1}} + \mathbf{B_1}(\mathbf{C_2}\vec{\mathbf{x_2}} + \mathbf{D_{2\_1}}\vec{\mathbf{u}} + \mathbf{D_{2\_2}}\mathbf{C_3}\vec{\mathbf{x_3}}) \\
&= \mathbf{A_1}\vec{\mathbf{x_1}} + \mathbf{B_1}\mathbf{C_2}\vec{\mathbf{x_2}} + \mathbf{B_1}\mathbf{D_{2\_1}}\vec{\mathbf{u}} + \mathbf{B_1}\mathbf{D_{2\_2}}\mathbf{C_3}\vec{\mathbf{x_3}}
\end{aligned}$$

$$\begin{aligned}
\dot{\vec{\mathbf{x_3}}} &= \mathbf{A_3}\vec{\mathbf{x_3}} + \mathbf{B_3}\vec{\mathbf{u_3}} = \mathbf{A_3}\vec{\mathbf{x_3}} + \mathbf{B_3}\vec{\mathbf{y_1}} = \mathbf{A_3}\vec{\mathbf{x_3}} + \mathbf{B_3}(\mathbf{C_1}\vec{\mathbf{x_1}} + \mathbf{D_1}\vec{\mathbf{u_1}}) \\
&= \mathbf{A_3}\vec{\mathbf{x_3}} + \mathbf{B_3}(\mathbf{C_1}\vec{\mathbf{x_1}} + \mathbf{D_1}\vec{\mathbf{y}}) = \mathbf{A_3}\vec{\mathbf{x_3}} + \mathbf{B_3}(\mathbf{C_1}\vec{\mathbf{x_1}} + \mathbf{D_1}(\mathbf{C_2}\vec{\mathbf{x_2}} + \mathbf{D_{2\_1}}\vec{\mathbf{u}} + \mathbf{D_{2\_2}}\mathbf{C_3}\vec{\mathbf{x_3}})) \\
&= \mathbf{A_3}\vec{\mathbf{x_3}} + \mathbf{B_3}\mathbf{C_1}\vec{\mathbf{x_1}} + \mathbf{B_3}\mathbf{D_1}\mathbf{C_2}\vec{\mathbf{x_2}} + \mathbf{B_3}\mathbf{D_1}\mathbf{D_{2\_1}}\vec{\mathbf{u}} + \mathbf{B_3}\mathbf{D_1}\mathbf{D_{2\_2}}\mathbf{C_3}\vec{\mathbf{x_3}}
\end{aligned}$$

For the input and output rates we have $o = E * w_2$ and $u = u_2$. We use the states of all three representations, so $s = s_1 + s_2 + s_3$ and $\overrightarrow{\mathbf{initVec}} = \begin{bmatrix} \overrightarrow{\mathbf{initVec_1}} \\ \overrightarrow{\mathbf{initVec_2}} \\ \overrightarrow{\mathbf{initVec_3}} \end{bmatrix}$. For simplicity, we do not consider a loop or body block with initialization matrices.

3.4 Representation Changes

3.4.1 Expansion

We may want to run a block multiple times in order to properly combine it with other blocks. For example, suppose block B_1 inputs three items and outputs two items, and block B_2 inputs five items and outputs seven items. In order to combine these blocks in a pipeline, B_1 must run five times (in order to output ten items) and B_2 must run two times (in order to input ten items). Therefore, we need to have a method to expand a representation so that it models a block running multiple times, rather than once.

Consider the state-space equation pair, where $\vec{\mathbf{u_1}}$ and $\vec{\mathbf{y_1}}$ are the first set of inputs and outputs, and $\vec{\mathbf{x}}$ is the original state vector:

$$\begin{aligned}\dot{\vec{\mathbf{x}}} &= \mathbf{A}\vec{\mathbf{x}} + \mathbf{B}\vec{\mathbf{u_1}} \\ \vec{\mathbf{y_1}} &= \mathbf{C}\vec{\mathbf{x}} + \mathbf{D}\vec{\mathbf{u_1}}\end{aligned}$$

If we run the block again, the equation pair in terms of the original state vector $\vec{\mathbf{x}}$ and the next set of inputs and outputs ($\vec{\mathbf{u_2}}$ and $\vec{\mathbf{y_2}}$) is:

$$\begin{aligned}\dot{\vec{\mathbf{x}}} &= \mathbf{A}(\mathbf{A}\vec{\mathbf{x}} + \mathbf{B}\vec{\mathbf{u_1}}) + \mathbf{B}\vec{\mathbf{u_2}} \\ \vec{\mathbf{y_2}} &= \mathbf{C}(\mathbf{A}\vec{\mathbf{x}} + \mathbf{B}\vec{\mathbf{u_1}}) + \mathbf{D}\vec{\mathbf{u_2}}\end{aligned}$$

Simplifying yields:

$$\begin{aligned}\dot{\vec{\mathbf{x}}} &= \mathbf{A}^2\vec{\mathbf{x}} + \mathbf{AB}\vec{\mathbf{u_1}} + \mathbf{B}\vec{\mathbf{u_2}} \\ \vec{\mathbf{y_2}} &= \mathbf{CA}\vec{\mathbf{x}} + \mathbf{CB}\vec{\mathbf{u_1}} + \mathbf{D}\vec{\mathbf{u_2}}\end{aligned}$$

Let $\vec{\mathbf{u}}$ be the combined input vector ($\vec{\mathbf{u}} = \begin{bmatrix} \vec{\mathbf{u_1}} \\ \vec{\mathbf{u_2}} \end{bmatrix}$) and $\vec{\mathbf{y}}$ be the combined output vector ($\vec{\mathbf{y}} = \begin{bmatrix} \vec{\mathbf{y_1}} \\ \vec{\mathbf{y_2}} \end{bmatrix}$). The representation in terms of these two vectors is:

$$\begin{aligned}\dot{\vec{\mathbf{x}}} &= \mathbf{A_2}\vec{\mathbf{x}} + \mathbf{B_2}\vec{\mathbf{u}} \\ \vec{\mathbf{y}} &= \mathbf{C_2}\vec{\mathbf{x}} + \mathbf{D_2}\vec{\mathbf{u}} \\ \mathbf{A_2} &= \mathbf{A}^2 \\ \mathbf{B_2} &= \begin{bmatrix} \mathbf{AB} & \mathbf{B} \end{bmatrix} \\ \mathbf{C_2} &= \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \end{bmatrix}\end{aligned}$$

$$\mathbf{D_2} = \begin{bmatrix} \mathbf{D} & \mathbf{0} \\ \mathbf{CB} & \mathbf{D} \end{bmatrix}$$

This new representation corresponds to a block that upon every execution runs the old block twice. By induction, a general formula for running a block n times is:

$$\mathbf{A_n} = \mathbf{A}^n \tag{3.2}$$

$$\mathbf{B_n} = \begin{bmatrix} \mathbf{A}^{n-1}\mathbf{B} & \mathbf{A}^{n-2}\mathbf{B} & ... & \mathbf{AB} & \mathbf{B} \end{bmatrix} \tag{3.3}$$

$$\mathbf{C_n} = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ ... \\ \mathbf{CA^{n-2}} \\ \mathbf{CA^{n-1}} \end{bmatrix} \tag{3.4}$$

$$\mathbf{D_n} = \begin{bmatrix} \mathbf{D} & \mathbf{0} & \mathbf{0} & ... & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{CB} & \mathbf{D} & \mathbf{0} & ... & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{CAB} & \mathbf{CB} & \mathbf{D} & ... & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ ... & ... & ... & ... & ... & ... & ... \\ \mathbf{CA}^{n-4}\mathbf{B} & \mathbf{CA}^{n-5}\mathbf{B} & \mathbf{CA}^{n-6}\mathbf{B} & ... & \mathbf{D} & \mathbf{0} & \mathbf{0} \\ \mathbf{CA}^{n-3}\mathbf{B} & \mathbf{CA}^{n-4}\mathbf{B} & \mathbf{CA}^{n-5}\mathbf{B} & ... & \mathbf{CB} & \mathbf{D} & \mathbf{0} \\ \mathbf{CA}^{n-2}\mathbf{B} & \mathbf{CA}^{n-3}\mathbf{B} & \mathbf{CA}^{n-4}\mathbf{B} & ... & \mathbf{CAB} & \mathbf{CB} & \mathbf{D} \end{bmatrix} \tag{3.5}$$

Since initializations are not affected, $\overrightarrow{\mathbf{initVec}}$, **preA**, **preB**, *stored*, and o_{pre} remain unchanged from the initial representation. Since the number of states is not changed, s remains the same. The new representation runs the old representation n times, so $u_{new} = n * u_{old}$, $o_{new} = n * o_{old}$.

As mentioned in the pipeline combination section, we may need to run a block n times, in addition to its initialization matrices, for the purpose of initializing the full pipeline. We denoted the matrices for doing this as $\mathbf{A^e}$, $\mathbf{B^e}$, $\mathbf{C^e}$, and $\mathbf{D^e}$. If the block being run n times does not need initialization, the calculation for these four matrices is exactly the same as described in equations (3.2)-(3.5). Otherwise, we must make some slight modifications:

$$\mathbf{A^e} = \mathbf{A}^n\mathbf{A_{pre}} \tag{3.6}$$

$$\mathbf{B^e} = \begin{bmatrix} \mathbf{A^nB_{pre}} & \mathbf{A^{n-1}B} & \mathbf{A^{n-2}B} & ... & \mathbf{B} \end{bmatrix} \tag{3.7}$$

$$\mathbf{C^e} = \begin{bmatrix} \mathbf{CA_{pre}} \\ \mathbf{CAA_{pre}} \\ ... \\ \mathbf{CA^{n-1}A_{pre}} \end{bmatrix} \tag{3.8}$$

$$\mathbf{D^e} = \begin{bmatrix} \mathbf{CB_{pre}} & \mathbf{D} & \mathbf{0} & \mathbf{0} & ... & \mathbf{0} & \mathbf{0} \\ \mathbf{CAB_{pre}} & \mathbf{CB} & \mathbf{D} & \mathbf{0} & ... & \mathbf{0} & \mathbf{0} \\ \mathbf{CA^2B_{pre}} & \mathbf{CAB} & \mathbf{CB} & \mathbf{D} & ... & \mathbf{0} & \mathbf{0} \\ ... & ... & ... & ... & ... & ... & \\ \mathbf{CA^{n-1}B_{pre}} & \mathbf{CA}^{n-2}\mathbf{B} & \mathbf{CA}^{n-3}\mathbf{B} & \mathbf{CA}^{n-3}\mathbf{B} & ... & \mathbf{CB} & \mathbf{D} \end{bmatrix} \tag{3.9}$$

3.4.2 Increasing the number of Stored Inputs

As mentioned in Section 3.3.1, it may be necessary to changed the stored inputs in a representation in order to combine it with another representation in a pipeline. Suppose we want to change the number of stored inputs from *oldStored* to *newStored*. Consider what happens in the old representation, with *oldStored* stored input variables. The filter accesses $peek(0)$, $peek(1)$, ... $peek(oldStored-1)$ from the *oldStored* stored input state variables. The o inputs to the filter are $peek(oldStored)$, $peek(oldStored+1)$, ... $peek(oldStored+o-1)$. Now we want to add $newStored - oldStored$ stored input variables, so that the total *newStored* stored input variables represent $peek(0)$, $peek(1)$, ... $peek(newStored-1)$, and the o inputs to the filter are $peek(newStored)$, $peek(newStored+1)$, ... $peek(newStored+o-1)$. Therefore, any references in the original representation to $peek(0)$, $peek(1)$, ... $peek(oldStored-1)$ remain the same, while references to $peek(oldStored)$, $peek(oldStored)$, ... $peek(oldStored+o-1)$ must be changed.

The old representation was:

$$\begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \end{bmatrix} = \begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} \\ \mathbf{A_{21}} & \mathbf{A_{22}} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \end{bmatrix} + \begin{bmatrix} \mathbf{B_{11}} & \mathbf{B_{12}} \\ \mathbf{B_{21}} & \mathbf{B_{22}} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{u_1}} \\ \vec{\mathbf{u_2}} \end{bmatrix}$$

$$\vec{\mathbf{y}} = \begin{bmatrix} \mathbf{C_1} & \mathbf{C_2} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \end{bmatrix} + \begin{bmatrix} \mathbf{D_1} & \mathbf{D_2} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{u_1}} \\ \vec{\mathbf{u_2}} \end{bmatrix}$$

We have divided the state vector $\vec{\mathbf{x}}$ into the non-stored input variables ($\vec{\mathbf{x_1}}$) and the stored input variables ($\vec{\mathbf{x_2}}$), and divided the input vector $\vec{\mathbf{u}}$ into the first $newStored - oldStored$ inputs ($\vec{\mathbf{u_1}}$) and the remaining inputs ($\vec{\mathbf{u_2}}$). We will assume $newStored - oldStored <= o$ (If not we can run this algorithm multiple times). The matrices **A**, **B**, **C**, and **D** are put into block-matrix form according to the state and input vector divisions.

In our new representation, we use $\vec{\mathbf{x_3}}$ to denote the added $newStored - oldStored$ states. As mentioned early, references to the first *oldStored* stored input states ($\vec{\mathbf{x_2}}$) remain the same. Additionally, references to the non-input states ($\vec{\mathbf{x_1}}$) also remain the same. Our new representation so far is:

$$\begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix} = \begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} & ? \\ \mathbf{A_{21}} & \mathbf{A_{22}} & ? \\ ? & ? & ? \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix} + \begin{bmatrix} ? & ? \\ ? & ? \\ ? & ? \end{bmatrix} \begin{bmatrix} \vec{\mathbf{u_1}} \\ \vec{\mathbf{u_2}} \end{bmatrix}$$

$$\vec{\mathbf{y}} = \begin{bmatrix} \mathbf{C_1} & \mathbf{C_2} & ? \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix} + \begin{bmatrix} ? & ? \end{bmatrix} \begin{bmatrix} \vec{\mathbf{u_1}} \\ \vec{\mathbf{u_2}} \end{bmatrix}$$

The ? indicates yet to be determined entries. In the old representation, the first $newStored - oldStored$ input elements (u_1) were $peek(oldStored)$ ... $peek(newStored - 1)$. In the new representation, these values are stored as states (x_3). Therefore, any matrix block that was previously multiplied by u_1 should be multiplied by x_2 instead.

Now the new representation is:

$$\begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix} = \begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} & \mathbf{B_{11}} \\ \mathbf{A_{21}} & \mathbf{A_{22}} & \mathbf{B_{21}} \\ ? & ? & ? \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix} + \begin{bmatrix} ? & ? \\ ? & ? \\ ? & ? \end{bmatrix} \begin{bmatrix} \vec{\mathbf{u_1}} \\ \vec{\mathbf{u_2}} \end{bmatrix}$$

$$\vec{\mathbf{y}} = \begin{bmatrix} \mathbf{C_1} & \mathbf{C_2} & \mathbf{D_1} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix} + \begin{bmatrix} ? & ? \end{bmatrix} \begin{bmatrix} \vec{\mathbf{u_1}} \\ \vec{\mathbf{u_2}} \end{bmatrix}$$

In the old representation, the remaining $o - (newStored - oldStored)$ input elements (u_2) were $peek(newStored)$ ... $peek(o + oldStored - 1)$. In the new representation, these are the first $o - (newStored - oldStored)$ input elements. We divide the input vector into the first $o - (newStored - oldStored)$ elements ($\vec{u_{1'}}$) and the remaining $newStored - oldStored$ elements ($\vec{u_{2'}}$). Any matrix block that was previously multiplied by $\vec{\mathbf{u_2}}$ should be multiplied by $\vec{u_{1'}}$ instead. Additionally, there is no dependence on $\vec{u_{2'}}$ by $\vec{\mathbf{x_1}}$, $\vec{\mathbf{x_2}}$, or $\vec{\mathbf{y}}$. The new representation is:

$$\begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix} = \begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} & \mathbf{B_{11}} \\ \mathbf{A_{21}} & \mathbf{A_{22}} & \mathbf{B_{21}} \\ ? & ? & ? \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix} + \begin{bmatrix} \mathbf{B_{12}} & \mathbf{0} \\ \mathbf{B_{22}} & \mathbf{0} \\ ? & ? \end{bmatrix} \begin{bmatrix} \vec{u_{1'}} \\ \vec{u_{2'}} \end{bmatrix}$$

$$\vec{\mathbf{y}} = \begin{bmatrix} \mathbf{C_1} & \mathbf{C_2} & \mathbf{D_1} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix} + \begin{bmatrix} \mathbf{D_2} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \vec{u_{1'}} \\ \vec{u_{2'}} \end{bmatrix}$$

The entries for the state update $\vec{\mathbf{x_3}}$ remain to be determined. Any stored input variable representing $peek(i)$ must get updated by $peek(i + o)$. $\vec{\mathbf{x_3}}$ is $peek(oldStored)$ ... $peek(newStored - 1)$, so it must be updated by $peek(o + oldStored)$ ... $peek(o +$

$newStored - 1$). This is precisely $\vec{u_{2'}}$, so the final new representation is:

$$\begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix} = \begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} & \mathbf{B_{11}} \\ \mathbf{A_{21}} & \mathbf{A_{22}} & \mathbf{B_{21}} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix} + \begin{bmatrix} \mathbf{B_{12}} & \mathbf{0} \\ \mathbf{B_{22}} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \vec{u_{1'}} \\ \vec{u_{2'}} \end{bmatrix}$$

$$\vec{\mathbf{y}} = \begin{bmatrix} \mathbf{C_1} & \mathbf{C_2} & \mathbf{D_1} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix} + \begin{bmatrix} \mathbf{D_2} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \vec{u_{1'}} \\ \vec{u_{2'}} \end{bmatrix}$$

Similarly, let the original initialization equation be:

$$\begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \end{bmatrix} = \begin{bmatrix} \mathbf{A_{pre11}} & \mathbf{A_{pre12}} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x_1}} \\ \vec{\mathbf{x_2}} \end{bmatrix} + \begin{bmatrix} \mathbf{B_{pre11}} & \mathbf{B_{pre12}} \\ \mathbf{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{u_{pre1}}} \\ \vec{\mathbf{u_{pre2}}} \end{bmatrix}$$

Where $\vec{\mathbf{u_{pre1}}}$ has length *oldStored*, and $\vec{\mathbf{u_{pre2}}}$ has length $o_{pre} - oldStored$. Now we simply consider $\vec{\mathbf{u_{pre1}}}$ to have length *newStored* and $\vec{\mathbf{u_{pre2}}}$ to have length $o_{pre} - newStored$. If $o_{pre} < newStored$, we set $o_{pre} = newStored$. Then the initialization equation is the same as before, except the original stored input states ($\vec{\mathbf{x_2}}$) are replaced by the new stored input states ($\begin{bmatrix} \vec{\mathbf{x_2}} \\ \vec{\mathbf{x_3}} \end{bmatrix}$).

We have derived $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$, $\mathbf{A_{pre}}$, and $\mathbf{B_{pre}}$ for the new representation. Clearly, $stored = newStored$ and $o_{pre} = o_{preold} + newStored - oldStored$. The input/output rate remains the same, so $o = o_{old}$ and $u = u_{old}$. We have added $newStored - oldStored$ total states, so $s = s_{old} + (newStored - oldStored)$ and $\overrightarrow{\mathbf{initVec}} = \begin{bmatrix} \overrightarrow{\mathbf{initVec_1}} \\ \overrightarrow{\mathbf{initVec_2}} \\ \vec{\mathbf{0}} \end{bmatrix}$.

3.5 Replacement

Once we have combined filters to a single representation and performed optimizations on it (see Chapter 4), we would like to convert it to StreamIt code. Given a representation R we can create the following StreamIt filter:

```
float -> float filter replacementFilter() {
  float x0, ... , x{s-1};

  prework push 0 pop preu peek preu {
    x0 = preA[0,0]*x0 + ... + preA[0,s-1]*x{s-1} + preB[0,0]*peek(0) + ... + preB[0,preu-1]*peek(preu-1);
    x1 = preA[1,0]*x0 + ... + preA[1,s-1]*x{s-1} + preB[1,0]*peek(0) + ... + preB[1,preu-1]*peek(preu-1);
    ...
    x{s-1} = preA[s-1,0]*x0 + ... + preA[s-1,s-1]*x{s-1} + preB[s-1,0]*peek(0) + ... + preB[s-1,preu-1]*peek(preu-1);
  }

  work push u pop o peek o {
    float x0_temp, ... , x{s-1}_temp;

    push(C[0,0]*x0 + ... + C[0,s-1]*x{s-1} + D[0,0]*peek(0) + ... + D[0,o-1]*peek(o-1));
    push(C[1,0]*x0 + ... + C[1,s-1]*x{s-1} + D[1,0]*peek(0) + ... + D[1,o-1]*peek(o-1));
    ...
    push(C[u,0]*x0 + ... + C[u,s-1]*x{s-1} + D[u,0]*peek(0) + ... + D[u,o-1]*peek(o-1));

    x0_temp = A[0,0]*x0 + ... + A[0,s-1]*x{s-1} + B[0,0]*peek(0) + ... + B[0,o-1]*peek(o-1);
    x1_temp = A[1,0]*x0 + ... + A[1,s-1]*x{s-1} + B[1,0]*peek(0) + ... + B[1,o-1]*peek(o-1);
    ...
    x{s-1}_temp = A[s-1,0]*x0 + ... + A[s-1,s-1]*x{s-1} + B[s-1,0]*peek(0) + ... + B[s-1,o-1]*peek(o-1);

    x0 = x0_temp;
    ...
    x{s-1} = x{s-1}_temp;

    pop(); pop(); ... pop(); // o pops
  }
}
```

We make two modifications to this filter. If a matrix entry is zero, any term involving that matrix entry is not placed in the filter. If a matrix entry is one, the multiplication of a peek or variable by this matrix entry is removed.

Chapter 4

Optimization

There are multiple metrics used to analyze performance of a computer program - speed (throughput, or outputs per second), space, power consumption, etc. We focus on speed and attempt to minimize the computation performed to produce each output. Obviously, this type of optimization has positive effects on the other parameters. However, we are mainly concerned with speed because it is simple to track, and due to falling hardware costs, is frequently a program's bottleneck.

There are two types of optimizations we consider. The first is to remove extraneous state variables from the linear state-space representation. This reduces the memory allocation for a program and reduces the number of loads and stores executed, which are typically time intensive operations. It also eliminates computations that involve the removed states. The second optimization is to reduce the parametrization of a state-space representation, by changing the representation to one with more zero and one entries in its matrices. This directly eliminates computations, since all multiplications by zero or one are not processed by the replacement algorithm.

4.1 State-Space Transformations

For any state-space equation pair, there are an infinite number of transformations to an equivalent state-space system. These transformations involve a change of basis of the state vector $\vec{\mathbf{x}}$ to $\mathbf{T}\vec{\mathbf{x}}$, where $\mathbf{T}$ is an invertible matrix. Consider the state-update equation $\dot{\vec{\mathbf{x}}} = \mathbf{A}\vec{\mathbf{x}} + \mathbf{B}\vec{\mathbf{u}}$. Multiplying the entire equation by $\mathbf{T}$ yields:

$$\mathbf{T}\dot{\vec{\mathbf{x}}} = \mathbf{T}\mathbf{A}\vec{\mathbf{x}} + \mathbf{T}\mathbf{B}\vec{\mathbf{u}}$$

Since $\mathbf{T}^{-1}\mathbf{T} = \mathbf{I}$, we can write:

$$\begin{aligned}
\mathbf{T}\dot{\vec{\mathbf{x}}} &= \mathbf{T}\mathbf{A}(\mathbf{T}^{-1}\mathbf{T})\vec{\mathbf{x}} + \mathbf{T}\mathbf{B}\vec{\mathbf{u}} = \mathbf{T}\mathbf{A}\mathbf{T}^{-1}(\mathbf{T}\vec{\mathbf{x}}) + \mathbf{T}\mathbf{B}\vec{\mathbf{u}} \\
\vec{\mathbf{y}} &= \mathbf{C}(\mathbf{T}^{-1}\mathbf{T})\vec{\mathbf{x}} + \mathbf{D}\vec{\mathbf{u}} = \mathbf{C}\mathbf{T}^{-1}(\mathbf{T}\vec{\mathbf{x}}) + \mathbf{D}\vec{\mathbf{u}}
\end{aligned}$$

Where we have introduced the output equation as well. Let $\vec{\mathbf{z}} = \mathbf{T}\vec{\mathbf{x}}$. $\vec{\mathbf{z}}$ is a new state vector related to the old state vector $\vec{\mathbf{x}}$ by the change of basis $\mathbf{T}$. Substituting into the equations above we get:

$$\begin{aligned}
\dot{\vec{\mathbf{z}}} &= \mathbf{T}\mathbf{A}\mathbf{T}^{-1}\vec{\mathbf{z}} + \mathbf{T}\mathbf{B}\vec{\mathbf{u}} \\
\vec{\mathbf{y}} &= \mathbf{C}\mathbf{T}^{-1}\vec{\mathbf{z}} + \mathbf{D}\vec{\mathbf{u}}
\end{aligned}$$

These is precisely the original state-space equation pair, with $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ transformed to $\mathbf{T}\mathbf{A}\mathbf{T}^{-1}$, $\mathbf{T}\mathbf{B}$, and $\mathbf{C}\mathbf{T}^{-1}$, respectively.

For a StreamIt state-space representation R, we must determine how the other values change. The initialization state update equation is essentially the same as the regular state update equation, so $\mathbf{A_{pre}}$ and $\mathbf{B_{pre}}$ are transformed to $\mathbf{T}\mathbf{A_{pre}}\mathbf{T}^{-1}$ and $\mathbf{T}\mathbf{B}$ respectively. Since the old state vector $\vec{\mathbf{x}}$ is multiplied by $\mathbf{T}$, the old initial state vector is multiplied by $\mathbf{T}$. The number of states, inputs, and outputs is the same, so s, o, and u are unchanged.

4.2 State Removal

There are two types of states that can be removed from a state-space system without changing its behavior - unreachable and unobservable states. Informally, unreachable states are unaffected by inputs and unobservable states have no effect on outputs. More formally, the set of states in a system can be divided into reachable and unreachable states where:

1. The unreachable states are not updated by any of the reachable states.

2. The unreachable states are not updated by any inputs.

In terms of the state-space equation pair, this means $\mathbf{A}[i, j] = 0, \mathbf{B}[i, k] = 0$ where i is the row of an unreachable state, j is the column of a reachable state, and k is any of the inputs. If all the unreachable states are initially zero, they remain zero because they are not updated by a non-zero value (either a reachable state or an input). Therefore, all unreachable states that are not initialized can be removed from a representation, since they do not effect the reachable states or the outputs.

The set of states in a system can also be divided into observable and unobservable states where:

1. The observable states are not updated by any of the unobservable states.

2. The outputs do not depend on the unobservable states.

In terms of the state-space equation pair, this means $\mathbf{C}[i, j] = 0, \mathbf{D}[k, j] = 0$ where j is the column of an observable state, i is the row of an unobservable state, and k is any of the outputs. The unobservable states are not used to update the observable states and are not used to determine the outputs. Therefore, all unobservable states can be removed from a representation (regardless of their initial values).

A simple algorithm to isolate the unreachable and unobservable states in a system by use of transformations is explained in [12]. The algorithm works as follows: Perform row operations on the augmented matrix $\begin{bmatrix} \mathbf{A} & \mathbf{B} \end{bmatrix}$ to put it into a type of row-echelon form[1], and perform the corresponding inverse column operations on $\mathbf{A}$ and $\mathbf{C}$ to keep the system equivalent to the original (Performing a row operation on a matrix is equivalent to left multiplying it by some invertible matrix, and performing a column operation on a matrix is equivalent to right multiplying it by some invertible

[1]A matrix is in standard row-echelon form if the first non-zero entry in each row is a 1 (called the leading 1) and the leading 1 in a higher row is to the left of the leading 1 in a lower row. For our type of row-echelon form, the *last* non-zero entry in each row is a 1 (call it the ending 1) and the ending 1 in a higher row is to the left of the ending 1 in a lower row.

matrix). Once the augmented matrix is in the desired form, row i of the combined matrix represents an unreachable state if there are no non-zero entries past the i^{th} column. For unobservable states, the combined matrix $\begin{bmatrix} \mathbf{A}^T & \mathbf{C}^T \end{bmatrix}$ is operated on instead.

Using this algorithm, we can find the entire set of unobservable states and remove them all. The only exceptions are those unobservable states that affect observable states in the initialization matrix A_{pre} . If j is the column of an observable state then we must have $\mathbf{A_{pre}}[i, j] = 0$ for all values of i, where i is the row of an observable state. Otherwise, the unobservable state j cannot be removed, because it affects at least one observable state, and therefore may affect the outputs.

More care must be taken when removing unreachable states. If an unreachable state has a non-zero starting value, or is affected by the initialization matrices, it cannot be removed. In either of these cases, the unreachable state may attain a non-zero value, and therefore may have an affect on the reachable states and/or outputs. Additionally, an unreachable state x_1 that is updated by a different unreachable state x_2 that cannot be removed may eventually have a non-zero value, even if it (x_1) is initially zero. Therefore, the unreachable state x_1 cannot be removed as well.

The last case may cause problems when trying to remove unreachable states. If an unreachable state x_1 is updated by unreachable states x_2 and x_3, we must check if those states can be removed before determining if state x_1 can be removed. If one of those states, say x_2, depends on x_1, we must determine if x_1 can be removed before determining whether x_2 can be removed - resulting in an impossible 'loop-like' determination. Clearly, a more robust approach is necessary.

Suppose we have found the set of unreachable states and they form the first k states of the state vector (we can do both of these steps by isolating the unreachable states, then moving them to the top of the state vector if necessary). Consider the sub-matrix $\mathbf{A}[1 : k; 1 : k]$ consisting of the first k rows and first k columns of $\mathbf{A}$. This sub-matrix represents how the unreachable states are updated based on each other. Suppose this sub-matrix is in upper-triangular form, which means that all entries below the main diagonal are zero. We can remove states in the following manner:

1. Check the states in reverse order, from state k to state 1.

2. For the i^{th} state, check whether the state has an initial value, is updated by the initialization matrices, or depends on a state with a higher index. If any of these are true, we cannot remove the state. Otherwise, we can remove the state.

Since the unreachable state sub-matrix is in upper-triangular form, all unreachable states can only have dependencies on states with a higher index. Furthermore, since we are working from the state with highest index first, at each step in the algorithm we can immediately determine whether or not a given state is removable. Therefore we have found our robust approach to remove unreachable states. What remains to be done is transforming the sub-matrix to upper-triangular form.

The QR algorithm, described in [18], is an iterative method of converting any square matrix $\mathbf{P}$ to upper-triangular form. The algorithm is essentially the following two step procedure, applied as many times as necessary.

1. $\mathbf{QR} = \mathbf{P}$ (QR factorization of P)

2. $\mathbf{P} = \mathbf{RQ}$

The QR factorization of a matrix $\mathbf{P}$ factors $\mathbf{P}$ into the product of an orthogonal matrix $\mathbf{Q}$[2] and an upper-triangular matrix $\mathbf{R}$. Since $\mathbf{R} = \mathbf{Q}^{-1}\mathbf{P}$, the QR algorithm is repeatedly transforming $\mathbf{P}$ to $\mathbf{Q}^{-1}\mathbf{PQ}$.

Since $\mathbf{Q}$ is invertible, we can apply this transformation to the unreachable state sub-matrix, where the transformation matrix $\mathbf{T}$ is $\mathbf{Q}^{-1}$. Since we want to keep the other states unchanged, the full transformation matrix applied to $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ is $\mathbf{T} = \begin{bmatrix} \mathbf{Q}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$

[2]An orthogonal matrix has the property that its transpose is equal to its inverse

4.3 Putting Inputs into States

So far we have considered optimizations that affect **A**, **B**, and **C**. Since the optimizations are entirely the result of state transformations, they do not affect **D**, which is independent of the choice of state-space basis. By storing every input as a state, however, all the entries of **D** are moved into **A** and can then be changed by state optimizations.

We have already discussed how to store inputs as states. When every input is stored as a state, we find the new state-equation pair is:

$$\begin{aligned}
\begin{bmatrix} \dot{\vec{\mathbf{x}}} \\ \dot{\vec{\mathbf{x}}}_{\mathbf{inputs}} \end{bmatrix} &= \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x}} \\ \vec{\mathbf{x}}_{\mathbf{inputs}} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} \vec{\mathbf{u}} \\
\vec{\mathbf{y}} &= \begin{bmatrix} \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x}} \\ \vec{\mathbf{x}}_{\mathbf{inputs}} \end{bmatrix} + \mathbf{0}\vec{\mathbf{u}}
\end{aligned}$$

These states should be added before state-removal is performed. It may seem counter-intuitive that we first add states, then seek to remove them. However, the added states represent computations involving **D**, which were not considered before. Removing some of these states results in reducing computations involving **D**.

4.4 Parameter Reduction

After removing as many states as possible, including input states, we want to change the state-space system to one with the fewest number of non-zero, non-one entries (termed parameters). If **A**, **B**, and **C** are completely filled, there are $s*(s+o+u)$ parameters. Ackermann and Bucy [1] show a general form for **A** and **C** (**B** can be filled with parameters) to have at most $s*(o+u)$ parameters, assuming there are no unobservable or unreachable states. They derive this form using system impulse responses. We will achieve this same form using row operations on the augmented matrix $\begin{bmatrix} \mathbf{A}^T & \mathbf{C}^T \end{bmatrix}$. The form we want is:

$$\mathbf{A}^T = \begin{bmatrix} \mathbf{L_1} & \mathbf{A_{12}} & \mathbf{A_{13}} & ... & \mathbf{A_{1u}} \\ \mathbf{0} & \mathbf{L_2} & \mathbf{A_{23}} & ... & \mathbf{A_{2u}} \\ \mathbf{0} & \mathbf{0} & \mathbf{L_3} & ... & \mathbf{A_{3u}} \\ ... & ... & ... & ... & ... \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & ... & \mathbf{L_u} \end{bmatrix}$$

$$\mathbf{C}^T = \begin{bmatrix} 1 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ ... & ... & ... & ... & ... \\ 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 0 & 0 & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & 1 \end{bmatrix}$$

The matrices $\mathbf{L_i}$ are rectangular, and the matrices $\mathbf{A_{ij}}$ are square, but do not necessarily have the same dimensions as each other. These matrices have the form:

$$\mathbf{L_i} = \begin{bmatrix} 0 & 0 & ... & 0 & * \\ 1 & 0 & ... & 0 & * \\ 0 & 1 & ... & 0 & * \\ ... & ... & ... & ... & ... \\ 0 & 0 & ... & 1 & * \end{bmatrix}$$

$$\mathbf{A_{ij}} = \begin{bmatrix} 0 & 0 & ... & * \\ ... & ... & ... & ... \\ 0 & 0 & ... & * \end{bmatrix}$$

The entries marked with a * are the parameters of the system. This is known as the observable canonical form of the system. In contrast, the reachable canonical form defines **A** and **B** instead of $\mathbf{A}^T$ and **C**, and **C** may be filled with parameters instead of **B**.

We present a simple algorithm, in pseudocode to attain the form above. We do not include the necessary inverse column operations that must go with all row operations.

```
Reduce Parameters {
  currRow = 0; colA = 0; colC = 0;

  while(currRow < totalRows) {

   -find a non-zero entry in column colC at or below row currRow of C{transpose}, and swap it with the
    entry in row currRow;
   -set C{transpose}[currRow,colC] = 1 by scaling the row appropriately;
    make all entries above and below it zero by adding appropriate multiple of row currRow to other rows;

    currRow = currRow + 1;
    colC = colC + 1;

    do {
     -find a non-zero entry in column colA at or below row currRow of A{transpose}, and swap it with the
      entry in row currRow;
     -set A{transpose}[currRow,colA] = 1 by scaling the row appropriately;
      make all entries below it zero by adding appropriate multiple of row currRow to other rows;

      currRow = currRow + 1;
      colA = colA + 1;
    } while a non-zero entry in column colA is found

    colA = colA + 1;
  }
}
```

It is possible that one type of form has fewer parameters than the other. Therefore, we perform the above algorithm on $\begin{bmatrix} \mathbf{A}^T & \mathbf{C}^T \end{bmatrix}$ as noted to produce the observable form, and on $\begin{bmatrix} \mathbf{A} & \mathbf{B} \end{bmatrix}$ to produce the reachable form, and check which one has fewer parameters.

4.5 Staged Execution

Using input state variables corresponds to executing a state-space block in two stages:

1. Put inputs into input state variables.

2. Execute the original block, using input states instead of actual inputs.

We can add additional stages by having multiple sets of input states - $\vec{\mathbf{x}_{\mathbf{inputs1}}}$, $\vec{\mathbf{x}_{\mathbf{inputs2}}}$, etc. The first set gets saved in the second set, the second set gets saved in the third set, etc. Suppose there are k input sets. We can write our state-space equation pair as follows:

$$\begin{bmatrix} \dot{\vec{\mathbf{x}}} \\ \dot{\vec{\mathbf{x}_{\mathbf{inputsk}}}} \\ \dots \\ \dot{\vec{\mathbf{x}_{\mathbf{inputs2}}}} \\ \dot{\vec{\mathbf{x}_{\mathbf{inputs1}}}} \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \dots & \mathbf{0} \\ \dots & \dots & \dots & \dots & \dots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{I} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x}} \\ \vec{\mathbf{x}_{\mathbf{inputsk}}} \\ \dots \\ \vec{\mathbf{x}_{\mathbf{inputs2}}} \\ \vec{\mathbf{x}_{\mathbf{inputs1}}} \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \dots \\ \mathbf{0} \\ \mathbf{I} \end{bmatrix} \vec{\mathbf{u}}$$

$$\vec{\mathbf{y}} = \begin{bmatrix} \mathbf{C} & \mathbf{D} & \dots & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{x}} \\ \vec{\mathbf{x}_{\mathbf{inputsk}}} \\ \dots \\ \vec{\mathbf{x}_{\mathbf{inputs2}}} \\ \vec{\mathbf{x}_{\mathbf{inputs1}}} \end{bmatrix} + \mathbf{0}\vec{\mathbf{u}}$$

By itself, executing the work of a filter in stages does not result in any gain in performance. However, minimally parameterizing the resulting system may be more productive than minimally parameterizing the one or two execution stage system. The canonical forms of the previous section do not in general minimally parameterize the system; hence evaluating staged execution remains an area of future research.

Chapter 5

Results

We have implemented the extraction, combination, and optimization (except multiple execution stages) procedures within the StreamIt compiler, which uses the KOPI java compiler infrastructure [8]. We measure performance by counting the number of floating point operations (additions and multiplications) executed in a given benchmark. The program DynamoRIO [2] is used to count operations.

We have collected the data for the base case (no replacement) and with state-space and linear replacement [11]. For each benchmark, values are expressed as a ratio of floating point operations with state-space or linear replacement[1] over operations with no replacement.

For the first 5 applications, which mainly have linear components without state, linear replacement and state-space replacement are equally effective. There is a huge performance downgrade for an FFT (Fast Fourier Transform) for both types of replacement. This is not surprising, since an FFT performs its computations sparsely across multiple filters. Combining these filters creates one filter densely packed with computations. This is exactly a conversion form an FFT to a DFT (Discrete Fourier Transform). We would need staged execution with minimal parameterization to convert the DFT back to an FFT.

[1]We do not consider frequency replacement in our comparison

Application	Statespace	Linear
FM Radio	0.1740	0.1692
FIR Program	0.9961	0.9980
Channel Vocoder	0.2601	0.2620
FilterBank	1.000	1.001
FFT (16 pt)	2.938	3.000
Linear Difference Equation	1.005	1.000
IIR	1.005	1.000
IIR + 1/2 Decimator	0.6441	1.000
IIR + 1/16 Decimator	0.3393	1.000
IIR + FIR	0.9413	1.000
FIR + IIR + IIR	0.9214	1.000

Table 5.1: Floating point operations with state-space and linear replacement normalized for no replacement

The remaining applications have filters with state, thus they cannot be analyzed linearly. Therefore we use the value 1.000 as the ratio in the linear column. For a simple standalone Linear Difference Equation or IIR (Infinite Impulse Response) filter, state-space replacement shows a very slight degradation in performance. The reason for this is that there are a few extra operations performed by the prework function. In terms of steady-state behavior, state-space replacement is equivalent to no replacement. There is no gain in performance because the filters are written optimally.

When we combine state filters with other filters, we notice a performance improvement using state-space replacement. For example, combining an IIR filter with a decimator that leaves 1 out of every 16 values has a 66% improvement. Combining an IIR filter with an FIR filter has a 6% improvement. In the case of an IIR filter with a decimator, there are extraneous computations performed by the IIR filter that are thrown away by the decimator. Combining their respective matrices removes these computations. In the case of an IIR filter with an FIR filter, the computations in both filters can be merged to a single set of computations. This indicates that state-space replacement is more useful when applied to combined filters than when applied to individual filters.

Chapter 6

Related Work

This thesis builds directly on the work done to analyze and optimize linear components in StreamIt graphs [11]. We have extended the theoretical framework for linear analysis to state-space analysis in order to apply our optimizations to a wider class of applications. We have also changed some parts of the underlying representations. Previously, constants were handled separately and peeked items beyond the pop rate were considered inputs. For our current work we have placed both types of items in states.

Many other groups are researching methods for automated DSP application optimizations. SPIRAL [14] is a system developed to generate libraries of DSP transforms. These libraries are designed for specific architectures, and can be re-optimized when hardware is upgraded or replaced. Other such libraries that have been designed include a package for linear algebra manipulations by the ATLAS project [19] and a set of optimized FFTs (Fast Fourier Transforms) [5].

Aside from StreamIt, other programming languages have been designed for streaming data. Synchronous languages which target embedded applications include LUSTRE [9], Esterel [3], and Signal [7]. Other stream-based languages are Occam [4], SISAL [6], and StreamC [15]. These are designed to exploit vector and parallel processing. However, none of these languages have compilers that run state-space or linear analysis.

Chapter 7

Conclusion

We present a methodology for the detection, analysis, combination, and optimization of linear state-space filters in DSP applications. This work is automatized in the compiler of a high-level programming language, StreamIt, designed for streaming applications. This frees the programmer from the burden of writing low-level optimized code that requires expert DSP analysis. Instead, the programmer can focus on the top-level design of a DSP application and write modular code in a structured setting.

Due to the infinite number of possible state-space transforms, the optimizations discussed are not necessarily ideal. Additionally, parts of DSP applications are non-linear and cannot be analyzed in the state-space domain. Therefore, the work presented in this thesis does not fully optimize some portions of DSP applications, and does not apply towards other portions. However, it does represent an additional step towards the goal of a full analysis and optimization of any application. We outline some of the future steps that can be taken, both to improve on the work in this thesis and to expand it to other types of domain-specific analyses and optimizations.

- As mentioned in Chapter 4, minimally parameterize a system, then uses multiple execution stages.

- Use a balanced representation [13] to quantify the relative impact of each state of a filter on its execution. Then the states that have impact values below a certain threshold can be removed, resulting in only a small change in the filter's execution.

- Formulate loops in traditional programming languages as state-space filters, and use state-space work developed in this thesis to detect their induction variables and optimize their execution.

- Create a cost function metric, $f(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$, that balances the traditional program analysis metrics (throughput, power consumption, memory allocation, etc.) in any manner desired. Then find a general way to minimize this cost function over all possible state-space transformations $\mathbf{T}$.

- Formulate a methodology to deal with filters whose outputs are a linear combination of their inputs but a non-linear combination of their state variables.

- Use a 'black box' method to find the appropriate representation of a filter. In this approach the filter is given inputs and the output data is collected. The input/output relations are used to formulate an appropriate state-space model that may not exactly represent the filter, but does so within a tolerable error margin (see [16]).

Appendix A

Benchmark Source Code

This is the StreamIt source code for the applications used in the Results chapter. All code is copyrighted to MIT.

Library Files (for use with FMRadio, FIR Program, and Channel Vocoder)

```
/**
 * Simple sink that just prints the data that is fed to it.
 **/
float->void filter FloatPrinter {
  work pop 1 {
    println(pop());
  }
}
```

```
/**
 * Simple FIR low pass filter with gain=g, wc=cutoffFreq(in radians) and N samples.
 * Eg:
 *                 ^ H(e^jw)
 *                 |
 *          ---------------
 *          |      |      |
 *          |      |      |
 *    <-------------------------> w
 *         -wc            wc
 *
 * This implementation is a FIR filter is a rectangularly windowed sinc function
 * (eg sin(x)/x), which is the optimal FIR low pass filter in
 * mean square error terms.
 *
 * Specifically, h[n] has N samples from n=0 to (N-1)
 * such that h[n] = sin(cutoffFreq*pi*(n-N/2))/(pi*(n-N/2)).
 * and the field h holds h[-n].
 */
float->float filter LowPassFilter(float g, float cutoffFreq, int
N) {
  float[N] h;

  /* since the impulse response is symmetric, I don't worry about reversing h[n]. */
  init {
    int OFFSET = N/2;
    for (int i=0; i<N; i++) {
      int idx = i + 1;
      // generate real part
      if (idx == OFFSET)
    /* take care of div by 0 error (lim x->oo of sin(x)/x actually equals 1)*/
    h[i] = g * cutoffFreq / pi;
      else
    h[i] = g * sin(cutoffFreq * (idx-OFFSET)) / (pi*(idx-OFFSET));
    }
  }

  /* implement the FIR filtering operation as the convolution sum. */
  work peek N pop 1 push 1 {
    float sum = 0;
    for (int i=0; i<N; i++) {
      sum += h[i]*peek(i);
    }
    push(sum);
    pop();
  }
}


/**
 * Simple FIR high pass filter with gain=g, stopband ws(in radians) and N samples.
 *
 * Eg
 *                 ^ H(e^jw)
 *                 |
 *     --------    |    -------
 *     |      |    |    |     |
 *     |      |    |    |     |
 *    <-------------------------> w
 *                  pi-wc pi pi+wc
 *
 *
 * This implementation is a FIR filter is a rectangularly windowed sinc function
 * (eg sin(x)/x) multiplied by e^(j*pi*n)=(-1)^n, which is the optimal FIR high pass filter in
 * mean square error terms.
 *
 * Specifically, h[n] has N samples from n=0 to (N-1)
 * such that h[n] = (-1)^(n-N/2) * sin(cutoffFreq*pi*(n-N/2))/(pi*(n-N/2)).
 * where cutoffFreq is pi-ws
 * and the field h holds h[-n].
 */
float->float filter HighPassFilter(float g, float ws, int N) {
  float[N] h;

  /* since the impulse response is symmetric, I don't worry about reversing h[n]. */
  init {
    int OFFSET = N/2;
    float cutoffFreq = pi - ws;
    for (int i=0; i<N; i++) {
      int idx = i + 1;
      /* flip signs every other sample (done this way so that it gets array destroyed) */
      int sign = ((i%2) == 0) ? 1 : -1;
      // generate real part
      if (idx == OFFSET)
    /* take care of div by 0 error (lim x->oo of sin(x)/x actually equals 1)*/
    h[i] = sign * g * cutoffFreq / pi;
      else
    h[i] = sign * g * sin(cutoffFreq * (idx-OFFSET)) / (pi*(idx-OFFSET));
    }

  }

  /* implement the FIR filtering operation as the convolution sum. */
  work peek N pop 1 push 1 {
    float sum = 0;
    for (int i=0; i<N; i++) {
      sum += h[i]*peek(i);
    }
    push(sum);
    pop();
  }
}


/* This is a bandpass filter with the rather simple implementation
of
 * a low pass filter cascaded with a high pass filter. The relevant parameters
 * are: end of stopband=ws and end of passband=wp, such that 0<=ws<=wp<=pi
 * gain of passband and size of window for both filters. Note that the high
 * pass and low pass filters currently use a rectangular window.
 **/
float->float pipeline BandPassFilter(float gain, float ws, float
wp, int numSamples) {
  add LowPassFilter(1, wp, numSamples);
  add HighPassFilter(gain, ws, numSamples);
}


/**
 * This filter compresses the signal at its input by a factor M.
 * Eg it inputs M samples, and only outputs the first sample.
 **/
float->float filter Compressor(int M) {
  work peek M pop M push 1 {
    push(pop());
    for (int i=0; i<(M-1); i++) {
      pop();
    }
  }
}
```

FM Radio

```
/*
 * Software equalizer.  This version uses n+1 low-pass filters directly,
 * as opposed to n band-pass filters, each with two low-pass filters.
 * The important observation is that we have bands 1-2, 2-4, 4-8, ...
 * This means that we should run an LPF for each intermediate frequency,
 * rather than two per band.  Calculating this in StreamIt isn't that bad.
 * For a four-band equalizer:
 *
 *              |
 *             DUP
 *    +---------+---------+
 *    |         |         |
 *    |        DUP         |
 *    |    +----+----+     |
 *    |    |    |    |     |
 *   16    8    4    2     1
 *    |    |    |    |     |
 *    |  (dup)(dup)(dup)   |
 *    |    |    |    |     |
 *    |    +----+----+     |
 *    |       RR(2)        |
 *    |         |          |
 *    +---------+---------+
 *       WRR(1,2(n-1),1)
 *              |
 *            (a-b)
 *            SUM(n)
 *              |
 *
 * It's straightforward to change the values of 1, 16, and n.  Coming out
 * of the EqualizerSplitJoin is 16 8 8 4 4 2 2 1; we can subtract and scale
 * these as appropriate to equalize.
 */


float->float filter FloatNAdder(int count) {

  work push 1 pop count {

    float sum = 0.0;

    for(int i=0; i<count; i++)
      sum += pop();

    push(sum);
  }
}


float->float filter FloatDiff() {

  work push 1 pop 2 {

    push(peek(0) - peek(1));
    pop();
    pop();
  }
}


float->float filter FloatDup() {

  work push 2 pop 1 {

    float val = pop();
    push(val);
    push(val);
  }
}


float->float pipeline EqualizerInnerPipeline(float rate, float
freq) {

  add FMLowPassFilter(rate,freq,64,0);
  add FloatDup();
}


float->float splitjoin EqualizerInnerSplitJoin(float rate, float
low, float high, int bands) {

  split duplicate();
  for(int i=0; i < bands-1; i++)
    add EqualizerInnerPipeline(rate,(float)exp((i+1)*(log(high)-log(low))/bands + log(low)));
  join roundrobin(2);
}


float->float splitjoin EqualizerSplitJoin(float rate, float low,
float high, int bands) {

  split duplicate();
  add FMLowPassFilter(rate,high,64,0);
  add EqualizerInnerSplitJoin(rate,low,high,bands);
  add FMLowPassFilter(rate,low,64,0);
  join roundrobin(1,(bands-1)*2,1);
}


float->float pipeline Equalizer(float rate) {
  int bands = 10;
  float low = 55;
  float high = 1760;

  add EqualizerSplitJoin(rate,low,high,bands);
  add FloatDiff();
  add FloatNAdder(bands);
}


float->float filter FMLowPassFilter(float sampleRate, float
cutFreq, int numTaps, int decimation) {

  float[numTaps] COEFF;     //all frequencies are in hz
  float tapTotal;

  init {
    float m = numTaps -1;
    //from Oppenheim and Schafer, m is the order of filter

    if(cutFreq == 0.0) {

      //Using a Hamming window for filter taps:
      tapTotal = 0;

      for(int i=0;i<numTaps;i++) {
    COEFF[i] = (float)(0.54 - 0.46*cos((2*pi)*(i/m)));
    tapTotal = tapTotal + COEFF[i];
      }

      //normalize all the taps to a sum of 1
      for(int i=0;i<numTaps;i++) {
    COEFF[i] = COEFF[i]/tapTotal;
      }
    }
    else{
    //ideal lowpass filter ==> Hamming window
    //has IR h[n] = sin(omega*n)/(n*pi)
    //reference: Oppenheim and Schafer

    float w = (2*pi) * cutFreq/sampleRate;

    for(int i=0;i<numTaps;i++) {
      //check for div by zero
      if(i-m/2 == 0)
    COEFF[i] = w/pi;
      else
    COEFF[i] = (float)(sin(w*(i-m/2)) / pi
                   / (i-m/2) * (0.54 - 0.46
                        * cos((2*pi)*(i/m))));
      }
    }
  }

  work push 1 pop decimation+1 peek numTaps {
    float sum = 0.0;
    for(int i=0; i<numTaps; i++) {
      sum += peek(i)*COEFF[i];
    }
    pop();
    for(int i=0; i<decimation; i++)
      pop();

    push(sum);

  }
}


float->float filter FMDemodulator(float sampRate, float max, float
bandwidth) {

  float mGain;

  init {
    mGain = max*(sampRate/(bandwidth*pi));
  }

  work push 1 pop 1 peek 2 {
    float temp = 0;
    //may have to switch to complex?
    temp = (float)(peek(0) * peek(1));
    //if using complex, use atan2
    temp = (float)(mGain * atan(temp));

    pop();
    push(temp);
  }
}
```

```
void->float filter FloatOneSource {

  float x;

  init {
    x = 0;
  }

  work push 1 pop 0 {
    push(x++);
  }
}


/*
 * Early attempt at an FM Radio... probably junk
 */

float->float pipeline FMRadioCore {

    // float samplingRate = 200000; //200khz sampling rate according to jeff at vanu
    float samplingRate = 250000000; // 250 MHz sampling rate much more sensible, though
    float cutoffFrequency = 108000000; //guess... doesn't FM freq max at 108 Mhz?
    int numberOfTaps = 64;
    float maxAmplitude = 27000;
    float bandwidth = 10000;
    //decimate 4 samples after outputting 1
    add FMLowPassFilter(samplingRate, cutoffFrequency, numberOfTaps, 4);
    add FMDemodulator(samplingRate, maxAmplitude, bandwidth);
    add Equalizer(samplingRate);
}


void->void pipeline FMRadio {

  add FloatOneSource();
  add FMRadioCore();
  add FloatPrinter();
}
```

FIR Program

```
/**
 * This streamit program contains a simple low pass filter
 * that filters the data from a source and funnels it directly
 * to a sink. This is more of a "kernel" type benchmark because
 * FIR filtering is widely used in actual DSP applications.
 **/

/**
 * Top level program.
 **/
void->void pipeline FIRProgram {
  add FloatSource();
  add LowPassFilter(1, pi/3, 256);
  add FloatPrinter();
}

/**
 * Simple float source -- puts out a ramp from
 * 0 to 15 over and over again. Note that it
 * generates its output data in its init function
 * and the oly work that occurs in the work function
 * is pushing the data on to the tape and doing some
 * buffer management.
 **/
void->float filter FloatSource {
  float[16] inputs;
  int idx;
  init {
    for(int i=0; i<16; i++) {
      inputs[i] = i;
    }
    idx = 0;
  }
  work push 1 {
    push(inputs[idx]);
    idx = (idx + 1) % 16;
  }
}
```

Channel Vocoder

```
/**
 * This is a channel vocoder as described in 6.555 Lab 2.
 * It's salient features are a filterbank each of which
 * contains a decimator after a bandpass filter.
 *
 * Sampling Rate is 8000 Hz.
 * First the signal is conditioned using a lowpass filter with
 * cutoff at 5000 Hz. Then the signal is "center clipped" which
 * basically means that very high and very low values are removed.
 *
 * Then, the signal is sent both to a pitch detector and to a
 * filter bank with 200 Hz wide windows (18 overall)
 *
 * Thus, each output is the combination of 18 band envelope values
 * from the filter bank and a single pitch detector value. This
 * value is either the pitch if the sound was voiced or 0 if the
 * sound was unvoiced.
 **/
void->void pipeline ChannelVocoder {
  add DataSource();
  // low pass filter to filter out high freq noise
  add LowPassFilter(1, (2*pi*5000)/8000, 64);
  add MainSplitjoin();
  add FloatPrinter();
}

/** This class is just a wrapper so that we don't have anonymous
inner classes. **/ float->float splitjoin MainSplitjoin {
  int PITCH_WINDOW = 100; // the number of samples to base the pitch detection on
  int DECIMATION   = 50; // decimation factor
  int NUM_FILTERS  = 4; //18;

  split duplicate;
  add PitchDetector(PITCH_WINDOW, DECIMATION);
  add VocoderFilterBank(NUM_FILTERS, DECIMATION);
  join roundrobin(1,4); // can't be NUM_FILTERS b/c const prop didn't work
}


/** a simple data source. **/ void->float filter DataSource() {
  int SIZE = 11;
  int index;
  float[SIZE] x;


  init {
    index = 0;
    x[0] = -0.70867825;
    x[1] = 0.9750938;
    x[2] = -0.009129746;
    x[3] = 0.28532153;
    x[4] = -0.42127264;
    x[5] = -0.95795095;
    x[6] = 0.68976873;
    x[7] = 0.99901736;
    x[8] = -0.8581795;
    x[9] = 0.9863592;
    x[10] = 0.909825;
  }


  work push 1 {
    push(x[index]);
    index = (index+1)%SIZE;
  }
}

/**
 * Pitch detector.
 **/
float->float pipeline PitchDetector(int winsize, int decimation) {
  add CenterClip();
  add CorrPeak(winsize, decimation);
}




/** The channel vocoder filterbank. **/ float->float splitjoin
VocoderFilterBank(int N, int decimation) {

  split duplicate;
  for (int i=0; i<N; i++) {
    add FilterDecimate(i, decimation);
  }
  join roundrobin;
}


/**
 * A channel of the vocoder filter bank -- has a
 * band pass filter centered at i*200 Hz followed
 * by a decimator with decimation rate of decimation.
 **/
float->float pipeline FilterDecimate(int i, int decimation) {
  //add VocoderBandPassFilter(i, 64); // 64 tap filter
  add BandPassFilter(2, 400*i, 400*(i+1), 64);
  add Compressor(decimation);


}

/**
 * This filter "center clips" the input value so that it is always
 * within the range of -.75 to .75
 **/
float->float filter CenterClip {
  float MIN = -0.75;
  float MAX =  0.75;
  work pop 1 push 1 {
    float t = pop();
    if (t<MIN) {
      push(MIN);
    } else if (t>MAX) {
      push(MAX);
    } else {
      push(t);
    }
  }
}

/**
 * This filter calculates the autocorrelation of the next winsize elements
 * and then chooses the max peak. If the max peak is under a threshold we
 * output a zero. If the max peak is above the threshold, we simply output
 * its value.
 **/
float->float filter CorrPeak(int winsize, int decimation) {
  float THRESHOLD = 0.07;
  work peek winsize push 1 pop decimation {
    float[winsize] autocorr; // auto correlation
    for (int i=0; i<winsize; i++) {
      float sum = 0;
      for (int j=i; j<winsize; j++) {
    sum += peek(i)*peek(j);
      }
      autocorr[i] = sum/winsize;
    }
    // armed with the auto correlation, find the max peak
    // in a real vocoder, we would restrict our attention to
    // the first few values of the auto corr to catch the initial peak
    // due to the fundamental frequency.
    float maxpeak = 0;
    for (int i=0; i<winsize; i++) {
      if (autocorr[i]>maxpeak) {
    maxpeak = autocorr[i];
      }
    }

    //println("max peak" + maxpeak);

    // output the max peak if it is above the threshold.
    // otherwise output zero;
    if (maxpeak > THRESHOLD) {
      push(maxpeak);
    } else {
      push(0);
    }
    for (int i=0; i<decimation; i++) {
      pop();
    }
  }
}
```

FilterBank

```
void->void pipeline FilterBankNew {
  int N_sim = 1024 * 2;
  int N_samp = 8;
  int N_ch = N_samp;
  int N_col = 32;
  float[N_sim] r;
  float[N_ch][N_col] H;
  float[N_ch][N_col] F;

  for (int i = 0; i < N_col; i++)
    for (int j = 0; j < N_ch; j++) {
      H[j][i] = i*N_col + j*N_ch + j + i + j + 1;
      F[j][i] = i*j + j*j + j + i;
    }
  add source();
  add FilterBank(N_samp, N_ch, N_col, H, F);
  add sink(N_sim);
}

void->float filter source() {
    float max = 1000.0;
    float current = 0.0;

    work push 1 pop 0 {
    push(current);
    if (current > max) {
        current = 0.0;
    } else {
        current = current+1.0;
    }
    }
}

float->void filter sink(int N) {
  work pop 1 { print(pop()); }
}

float->float pipeline FilterBank(int N_samp, int N_ch, int N_col,
                 float[N_ch][N_col] H,
                 float[N_ch][N_col] F)
{
  add Branches(N_samp, N_ch, N_col, H, F);
  add Combine(N_samp);
}

float->float splitjoin Branches(int N_samp, int N_rows, int N_col,
                  float[N_rows][N_col] H,
                  float[N_rows][N_col] F)
{
  split duplicate;
  for (int i = 0; i < N_rows; i++)
  {
    float[N_col] H_ch;
    float[N_col] F_ch;
    for (int j = 0; j < N_col; j++)
    {
      H_ch[j] = H[i][j];
      F_ch[j] = F[i][j];
    }
    add Bank(N_samp, N_col, H_ch, F_ch);
  }
  join roundrobin;
}

float->float pipeline Bank(int N, int L, float[L] H, float[L] F) {
  add Delay_N(L-1);
  add FirFilter(L, H);
  add DownSamp(N);
  add UpSamp(N);
  add Delay_N(L-1);
  add FirFilter(L, F);
}

float->float filter Delay_N(int N) {
  float[N] state;
  int place_holder;

  init {
    for (int i = 0; i < N; i++)
      state[i] = 0;
    place_holder = 0;
  }

  work pop 1 push 1 {
    push(state[place_holder]);
    state[place_holder] = pop();
    place_holder++;
    if (place_holder == N)
      place_holder = 0;
  }
}

float->float filter FirFilter(int N, float[N] COEFF) {
  work pop 1 peek N push 1 {
    float sum = 0;
    for (int i = 0; i < N; i++)
      sum += peek(i) * COEFF[N-1-i];
    pop();
    push(sum);
  }
}

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

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

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

FFT

```
void->void pipeline FFT2() {

  add FFTTestSource(16);
  add FFTKernel2(16);
  add FloatPrinter();
}


float->float filter CombineDFT(int n) {

  float wn_r, wn_i;

  init {
    wn_r = (float)cos(2 * 3.141592654 / n);
    wn_i = (float)sin(-2 * 3.141592654 / n);
  }

  work push 2*n pop 2*n {
        int i;
        float w_r = 1;
        float w_i = 0;
    float[2*n] results;

        for (i = 0; i < n; i += 2)
        {
        // this is a temporary work-around since there seems to be
        // a bug in field prop that does not propagate nWay into the
        // array references.  --BFT 9/10/02

        //int tempN = nWay;
        //Fixed --jasperln

            // removed nWay, just using n  --sitij 9/26/03

        float y0_r = peek(i);
            float y0_i = peek(i+1);

        float y1_r = peek(n + i);
            float y1_i = peek(n + i + 1);

            float y1w_r = y1_r * w_r - y1_i * w_i;
            float y1w_i = y1_r * w_i + y1_i * w_r;

            results[i] = y0_r + y1w_r;
            results[i + 1] = y0_i + y1w_i;

        results[n + i] = y0_r - y1w_r;
            results[n + i + 1] = y0_i - y1w_i;

            float w_r_next = w_r * wn_r - w_i * wn_i;
            float w_i_next = w_r * wn_i + w_i * wn_r;
            w_r = w_r_next;
            w_i = w_i_next;
        }

        for (i = 0; i < 2 * n; i++)
        {
            pop();
            push(results[i]);
        }
    }

}
```

```
float->float filter FFTReorderSimple(int n) {

  int totalData;

  init {
    totalData = 2*n;
  }

  work push 2*n pop 2*n {
        int i;

        for (i = 0; i < totalData; i+=4)
        {
            push(peek(i));
            push(peek(i+1));
        }

        for (i = 2; i < totalData; i+=4)
        {
            push(peek(i));
            push(peek(i+1));
        }

        for (i=0;i<n;i++)
        {
            pop();
            pop();
        }
    }
}


float->float pipeline FFTReorder(int n) {

  for(int i=1; i<(n/2); i*= 2)
    add FFTReorderSimple(n/i);

}


float->float pipeline FFTKernel1(int n) {

  if(n>2) {
    add splitjoin {
      split roundrobin(2);
      add FFTKernel1(n/2);
      add FFTKernel1(n/2);
      join roundrobin(n);
    }
  }
  add CombineDFT(n);
}


float->float splitjoin FFTKernel2(int n) {

  split roundrobin(2*n);
  for(int i=0; i<2; i++) {
    add pipeline {
      add FFTReorder(n);
      for(int j=2; j<=n; j*=2)
        add CombineDFT(j);
    }
  }
  join roundrobin(2*n);
}


void->float filter FFTTestSource(int N) {

  work push 2*N pop 0 {
    int i;
    push(0.0);
    push(0.0);
    push(1.0);
    push(0.0);

    for(i=0; i<2*(N-2); i++)
      push(0.0);
  }
}


float->void filter FloatPrinter {
    work push 0 pop 1 {

        print(pop());
    }
}
```

Linear Difference Equation

```
void->float filter source() {

  float x;

  init {
    x = 1.0;
  }

  work push 1 pop 0 {
    push(x);
    x = 0.0;
  }
}


float->void filter sink() {

  work push 0 pop 1 {
    print(pop());
  }
}



void->void pipeline diffEq() {

  add source();
  add linDiff();
  add sink();
}


float->float filter linDiff() {

 // these variables save the previous outputs
 float x,y,z;

  init {
    x = 0.0;
    y = 0.0;
    z = 0.0;
  }

  work push 1 pop 1 peek 3 {
    float temp;

    temp = 0.2*peek(0) + 0.4*peek(1) - 0.5*peek(2) + 0.3*x - 0.8*y - 0.6*z;
    push(temp);
    pop();
    x = y;
    y = z;
    z = temp;
  }
}
```

IIR + 1/2 Decimator

```
void->float filter source() {

  float x;

  init {
    x = 1.0;
  }

  work push 1 pop 0 {
    push(x);
    x = 0.0;
  }
}


float->void filter sink() {

  work push 0 pop 1 {
    print(pop());
  }
}


void->void pipeline IIR() {

  add source();
  add IIRFilter();
  add decimate();
  add sink();
}


float->float filter IIRFilter() {

  float curr;

  init {
    curr = 0.0;
  }

  work push 1 pop 1 peek 3 {

    float temp;
    temp = peek(0)/4 + peek(1)/8 + peek(2)/6;
    curr = curr/2 + temp;
    push(curr);
    pop();
  }

}


float->float filter decimate() {

  work push 1 pop 2 {
    push(pop());
    pop();
  }
}
```

IIR + 1/16 Decimator

```
void->float filter source() {

  float x;

  init {
    x = 1.0;
  }

  work push 1 pop 0 {
    push(x);
    x = 0.0;
  }
}


float->void filter sink() {

  work push 0 pop 1 {
    print(pop());
  }
}


void->void pipeline IIR() {

  add source();
  add IIRFilter();
  add decimate();
  add sink();
}


float->float filter IIRFilter() {

  float curr;

  init {
    curr = 0.0;
  }

  work push 1 pop 1 peek 3 {

    float temp;
    temp = peek(0)/4 + peek(1)/8 + peek(2)/6;
    curr = curr/2 + temp;
    push(curr);
    pop();
  }

}


float->float filter decimate() {

  work push 1 pop 16 {
    push(peek(0));
    for(int i=0; i<16; i++)
      pop();
  }
}
```

IIR+FIR

```
void->float filter source() {

  float x;

  init {
    x = 1.0;
  }

  work push 1 pop 0 {
    push(x);
    x = 0.0;
  }
}


float->void filter sink() {

  work push 0 pop 1 {
    print(pop());
  }
}


void->void pipeline IIR() {

  add source();
  add IIRFilter();
  add FIR();
  add sink();
}


float->float filter IIRFilter() {

  float curr;
  init {
    curr = 0.0;
  }

  work push 1 pop 1 peek 3 {

    float temp;
    temp = peek(0)/4 + peek(1)/8 + peek(2)/6;
    curr = curr/2 + temp;
    push(curr);
    pop();
  }

}



float->float filter FIR() {

  work push 1 pop 1 peek 5 {
     push(0.45*peek(0) - 0.8*peek(1) - 0.56*peek(2) - 0.8*peek(3) + 0.45*peek(4));
     pop();
  }
}
```

FIR+IIR+FIR

```
void->float filter source() {

  float x;

  init {
    x = 1.0;
  }

  work push 1 pop 0 {
    push(x);
    x = 0.0;
  }
}


float->void filter sink() {
  work push 0 pop 1 {
    print(pop());
  }
}


void->void pipeline IIR() {

  add source();
  add FIR();
  add IIRFilter();
  add FIR();
  add sink();
}


float->float filter IIRFilter() {

  float curr;

  init {
    curr = 0.0;
  }

  work push 1 pop 1 peek 3 {

    float temp;
    temp = peek(0)/4 + peek(1)/8 + peek(2)/6;
    curr = curr/2 + temp;
    push(curr);
    pop();
  }

}



float->float filter FIR() {

  work push 1 pop 1 peek 5 {
     push(0.45*peek(0) - 0.8*peek(1) - 0.56*peek(2) - 0.8*peek(3) + 0.45*peek(4));
     pop();
  }
}
```

Bibliography

[1] J.E. Achermann and R.S. Bucy. Canonical minimal realization of a matrix of impulse response sequences. *Information and Control*, pages 224–231, 1971.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIG-PLAN 1999 Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[3] Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.

[4] Inmos Corporation. *Occam 2 Reference Manual*. Prentice Hall, 1988.

[5] M. Frigo. A Fast Fourier Transform Compiler. In *Proceedings of the ACM SIG-PLAN 1999 Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[6] J. Gaudiot, W. Bohm, T. DeBoni, J. Feo, and P. Mille. The Sisal Model of Functional Programming and its Implementation. In *Proceedings of the 2nd Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*, 1997.

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

[8] Vincent Gay-Para, Thomas Graf, Andre-Guillaume Lemonnier, and Erhard Wais. *Kopi Reference Manual.* http//www.dms.at/kopi/docs/kopi.html, 2001.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow language lustre. *Proceedings of the IEE*, 79(1), 1991.

[10] Michael A. Karczmarek. Constrained and phased scheduling of synchronous data flow graphs for the streamit language. Master's thesis, Massachusetts Institute of Technology, October 2002.

[11] Andrew A. Lamb. Linear analysis and optimization of stream programs. Master's thesis, Massachusetts Institute of Technology, May 2003.

[12] D. Q. Mayne. An elementary derivation of rosenbrock's minimal realization algorithm. *IEE Transactions on Automatic Control*, pages 306–307, 1973.

[13] B. C. Moore. Principal component analysis in linear sytems: controllability, observability, and model reduction. *IEE Transactions on Automatic Control*, 26(1):17–31, 1981.

[14] M. Püschel, B. Singer, J. Xiong, J. M .F. Moura, J. Johnson, D. Padua, M. M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications, special issue on Automatic Performance Tuning*, 18(1):21–45, 2004.

[15] Scott Rixner, William J. Dally, Ujval J. Kapani, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Manson, and John D. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *International Symposium on High Performance Computer Architecture*, Dallas, TX, 1998.

[16] B. De Schutter. Minimal state-space realization in linear system theory: An overview. *Journal of Computational and Applied Mathematics*, 121(1), 2000.

[17] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction*, Grenoble, France, 2002.

[18] Lloyd N. Trefethen and David Bau III. *Numerical Linear Algebra.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[19] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1), 2001.