Optimizing Stream Programs Using Linear State Space Analysis

Sitij Agrawal, William Thies, and Saman Amarasinghe
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{sitij, thies, saman}@csail.mit.edu

ABSTRACT

Digital Signal Processing (DSP) is becoming increasingly widespread in portable devices. Due to harsh constraints on power, latency, and throughput in embedded environments, developers often appeal to signal processing experts to hand-optimize algorithmic aspects of the application. However, such DSP optimizations are tedious, error-prone, and expensive, as they require sophisticated domain-specific knowledge.

We present a general model for automatically representing and optimizing a large class of signal processing applications. The model is based on linear state space systems. A program is viewed as a set of filters, each of which has an input stream, an output stream, and a set of internal states. At each time step, the filter produces some outputs that are a linear combination of the inputs and the state values; the state values are also updated in a linear fashion. Examples of linear state space filters include IIR filters and linear difference equations.

Using the state space representation, we describe a novel set of program transformations, including combination of adjacent filters, elimination of redundant states and reduction of the number of system parameters. We have implemented the optimizations in the StreamIt compiler and demonstrate improved generality over previous techniques.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Optimization; compilers; code generation*; D.2.2 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; D.3.2 [**Programming Languages**]: Language Classifications—*Data-flow languages*

General Terms

Languages, Design, Algorithms, Performance

Keywords

Stream Programing, StreamIt, Synchronous Dataflow, Linear State Space Systems, Embedded

CASES'05, September 24–27, 2005, San Francisco, California, USA.

1. INTRODUCTION

Digital devices are increasingly common in everyday life. Examples include cell phones, modems, CD players, and high definition televisions. These products utilize Digital Signal Processing (DSP) for a variety of applications, including signal compression and decompression, noise reduction, and error correction. Because such programs often run in embedded environments with limited power and strict real-time requirements, this is a domain where optimization is still very important.

Due to the emphasis on performance, DSP programs are typically implemented in two phases. First, the functionality of the algorithm is specified at a high level of abstraction by applications designers. Then, a set of DSP experts examine the global flow of data and perform specialized transformations to efficiently map the algorithm to C and assembly code. This process is tedious and costly, as every change in the high-level design necessitates new optimizations and re-implementation of the code; in addition, the optimizations are typically architecture-dependent, thereby sacrificing portability and robustness. In an ideal world, the compiler would provide a unified framework for optimizing high-level DSP algorithms, thereby providing an automatic path from the functional specification to an efficient implementation.

As a small step towards this vision, this paper introduces a new framework for analyzing and optimizing a large class of DSP applications. The framework, which we call linear state space analysis, applies a large body of work on linear state space systems to the domain of programming languages and compilers. A state space system is one in which a computational element produces and consumes some values on each execution. In addition, some internal data may be preserved between executions; we refer to these values as *states*.

A state space system is linear if it satisfies two criteria. First, each output must be a linear combination of the inputs and the current state values. Second, on each execution, the state values must be updated as a linear combination of the inputs and the previous state values. Linear state space systems can model a large class of DSP operations, including FIR filters, IIR filters, DCTs, upsamplers / downsamplers and linear difference equations. They are a generalization of linear systems, in which a linear input-output relationship holds but there is no state retained between executions. There is a strong theoretical foundation for reasoning about linear state space systems; it has been shown, for example, how to minimize the number of states [26] as well as the number of parameters in the system [1, 25, 31].

Our technique leverages domain-specific knowledge of linear state space systems to perform novel optimizations on stream programs. Our framework applies to languages based on the synchronous dataflow model of computation. In this model, a program is a set of autonomous filters that communicate over FIFO channels. On each execution, a filter consumes some items from its input channels and produces some items on its output channels; it may also retain states between executions. A filter can be modeled as a linear state space system if it satisfies the criteria described previously. In this paper, we use StreamIt as the input language; StreamIt is a high-level language and compiler infrastructure for DSP applications.

This paper makes the following contributions:

- An extraction algorithm that examines each filter and, where possible, builds a linear state space representation to describe its behavior.
- Rules for combining adjacent linear state space blocks into a single representation, thereby eliminating redundant computations and enabling further optimizations. Each possible configuration of blocks (sequential, parallel, and cyclic) is handled.
- A state removal optimization that detects and eliminates redundant states within a block.
- A parameter reduction optimization that adjusts the coefficients of the state update and output calculation in order to decrease the number of operations used.

While the principle contribution of this paper is in the elegance and generality of its theoretical formulation, we also demonstrate that state space analysis is tractable by implementing it in the StreamIt compiler. We evaluate state space analysis over a small set of micro-benchmarks and illustrate that it is more general than linear optimizations alone.

In the rest of this section, we give an overview of StreamIt and illustrating examples of state space analysis[1]. Section 2 gives the details of our state space representation, including extraction and combination rules, while Section 3 describes the optimizations. Section 4 discusses our implementation, Section 5 details related work, and Section 6 concludes.

1.1 The StreamIt Language

StreamIt [33] is an architecture-independent language for signal processing applications. The model of computation in StreamIt is Synchronous Dataflow [23], in which independent filters communicate at fixed rates over FIFO channels. StreamIt aims to expose the abundant parallelism and regular communication patterns in stream programs for the benefit of the compiler. The optimizations described in this paper would be infeasible in a general-purpose language such as C. As detailed elsewhere [33], C obscures the high-level structure of a streaming application due to possible aliasing between autonomous filters, complex modulo expressions on circular buffers, and interleaving of atomic execution steps with global control flow. In addition, StreamIt offers improved programmer productivity over C due to its parameterizable and composable stream constructs.

[1] As the rest of this paper is devoted to linear state space analysis, we often say only "state space analysis" for brevity.

```
float->float filter MovingAverage(int N) {
  float[N] weights;  // a filter field

  // init function initializes the weights
  init {
    for (int i=0; i<N; i++)
      weights[i] = 1/N;
  }

  // work function declares push, pop, peek rates
  work push 1 pop 1 peek N {
    float result = 0;  // a local variable
    for (int i=0; i<N; i++) {
      result += weights[i] * peek(i);
    }
    push(result);
    pop();
  }
}
```

Figure 1: Example of a StreamIt filter.

The basic programmable unit in StreamIt is a filter, which executes a user-defined work function as its atomic execution step; for example, see the `MovingAverage` filter in Figure 1. Filters communicate with each other using FIFO channels. On each execution, a filter consumes (*pops*) a fixed number of items from its input channel and produces (*pushes*) a fixed number of items to its output channel. A filter can also *peek* at a given index on its input channel without consuming the item; this makes it simple to write sliding-window applications such as the `MovingAverage`. The push, pop, and peek rates are declared on the same line as the work function, thereby enabling the compiler to construct a static schedule of filter firings [23].

Each filter has a distinct address space. A filter can store two types of variables: a *field* and a *local*. Fields are declared in the scope of the filter and are preserved across executions, while locals are declared inside the work function and are only live within a single execution. There is also an init function, run once at the beginning of the program, that can be used to initialize fields.

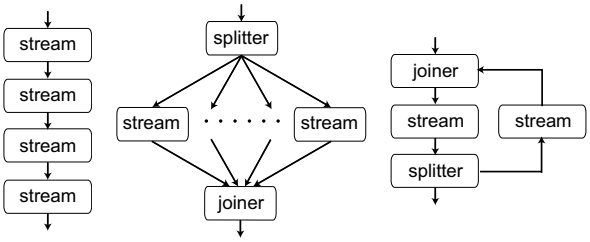
StreamIt provides three hierarchical structures for composing filters into larger stream graphs (see Figure 2). The *pipeline* construct composes streams in sequence, with the output of one connected to the input of the next. The *splitjoin* construct distributes data to a set of parallel streams, which are then joined together in a roundrobin fashion. The *feedback loop* provides a mechanism for introducing cycles in the graph. An example of a pipeline appears in Figure 3. It contains a single Infinite Impulse Response (IIR) filter, which could be implemented as shown at the top of Figure 4.

1.2 State Space Example

As described previously, a linear state space model describes a stream in which both the outputs and the state values are updated as a linear combination of the inputs and the previous states. We use the following notation to describe such systems:

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

In these equations, the state vector is denoted by $\vec{\mathbf{x}}$, the inputs by $\vec{\mathbf{u}}$, and the outputs by $\vec{\mathbf{y}}$. $\vec{\mathbf{x}}$ represents the new



(a) A pipeline. (b) A splitjoin. (c) A feedbackloop.

Figure 2: Stream structures supported by StreamIt.

```
float -> float pipeline Main() {
  add Source();  // code for Source not shown
  add IIR();
  add Output();  // code for Output not shown
}
```

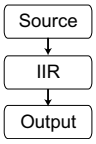


Figure 3: Example pipeline with IIR filter.

```
float->float filter IIR_1 {
  float x1, x2;
  work push 2 pop 3 {
    float u1 = pop();
    float u2 = pop();
    float u3 = pop();

    push(2*x1 + 2*x2 + 3*u1);
    push(4*u3 - 5*x1 - 5*x2);

    float x1_temp = (x1 + x2 + u1) * 0.5;
    float x2_temp = (x1 + x2 + u2) * 0.25;

    x1 = x1_temp;
    x2 = x2_temp;
  }
}
```

Number of multiplications: 8
Number of additions: 8
State space representation:

$$\begin{aligned}\vec{\dot{\mathbf{x}}} &= \begin{bmatrix} 0.5 & 0.5 \\ 0.25 & 0.25 \end{bmatrix}\vec{\mathbf{x}} + \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.25 & 0 \end{bmatrix}\vec{\mathbf{u}} \\ \vec{\mathbf{y}} &= \begin{bmatrix} 2 & 2 \\ -5 & -5 \end{bmatrix}\vec{\mathbf{x}} + \begin{bmatrix} 3 & 0 & 0 \\ 0 & 0 & 4 \end{bmatrix}\vec{\mathbf{u}}\end{aligned}$$

State Removal

```
float->float filter IIR_2 {
  float x;
  work push 2 pop 3 {
    float u1 = pop();
    float u2 = pop();
    float u3 = pop();

    push(2*x + 3*u1);
    push(4*u3 - 5*x);

    x = 0.75*x + 0.5*u1 + 0.25*u2;
  }
}
```

Number of multiplications: 7
Number of additions: 4
State space representation:

$$\begin{aligned}\vec{\dot{\mathbf{x}}} &= \begin{bmatrix} 0.75 \end{bmatrix}\vec{\mathbf{x}} + \begin{bmatrix} 0.5 & 0.25 & 0 \end{bmatrix}\vec{\mathbf{u}} \\ \vec{\mathbf{y}} &= \begin{bmatrix} 2 \\ -5 \end{bmatrix}\vec{\mathbf{x}} + \begin{bmatrix} 3 & 0 & 0 \\ 0 & 0 & 4 \end{bmatrix}\vec{\mathbf{u}}\end{aligned}$$

Parameter Reduction

```
float->float filter IIR_3 {
  float x;
  work push 2 pop 3 {
    float u1 = pop();
    float u2 = pop();
    float u3 = pop();

    push(x + 3*u1);
    push(4*u3 - 2.5*x);

    x = 0.75*x + u1 + 0.5*u2;
  }
}
```

Number of multiplications: 5
Number of additions: 4
State space representation:

$$\begin{aligned}\vec{\dot{\mathbf{x}}} &= \begin{bmatrix} 0.75 \end{bmatrix}\vec{\mathbf{x}} + \begin{bmatrix} 1 & 0.5 & 0 \end{bmatrix}\vec{\mathbf{u}} \\ \vec{\mathbf{y}} &= \begin{bmatrix} 1 \\ -2.5 \end{bmatrix}\vec{\mathbf{x}} + \begin{bmatrix} 3 & 0 & 0 \\ 0 & 0 & 4 \end{bmatrix}\vec{\mathbf{u}}\end{aligned}$$

Figure 4: Example optimization of an IIR filter using linear state space analysis. The top segment shows the original code. The middle segment depicts the action of state removal, in which the quantity $x_1 + x_2$ is replaced by a single variable x. The bottom segment illustrates parameter reduction, in which the coefficients are refactored so as to eliminate two multiplications (two coefficients assume a value of 1).

state vector, i.e., the state vector after it is updated. The first equation is for the state updates, while the second equation is for the outputs. **A**, **B**, **C**, and **D** are matrices whose dimensions depend on the number of states, inputs, and outputs.

Figure 4 illustrates an optimization sequence for an IIR filter. Three versions of the filter are shown: original, following state removal, and following parameter reduction. In each case, the state space representation for the filter is shown on the right, along with the number of multiplications and additions needed per execution of the work function.

The state removal optimization identifies that the states x_1 and x_2 are always used as part of the expression $x_1 + x_2$. Thus, one of the states can be eliminated in favor of a single variable, x, that tracks the value of the sum. While relatively simple in this example, such a transformation can be quite subtle when applied to a large representation (e.g., the result of combining many filters together.) State removal can reduce storage requirements as well as eliminate arithmetic operations (in this example, 1 multiplication and 4 additions). As described in Section 3.2, state removal is formulated as a general sequence of matrix operations.

The parameter reduction optimization refactors the coefficients that operate on the state variables in order to reduce the number of operations needed. Following the transformation, x assumes a value that is twice as large as the original (at any given point of execution). However, this change does not affect the output of the filter, as the coefficients in **A** are compensated accordingly. The transformation enables two coefficients in **B** and **C** to change to a value of 1, thereby eliminating two multiplication operations. As described in Section 3.3, this transformation is also formulated as a general series of matrix operations.

2. STATE SPACE ANALYSIS

Our analysis operates on a symbolic representation of linear state space filters. We analyze 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 hierarchical StreamIt blocks—pipelines, splitjoins, and feedback loops. Such a process results in a single state space representation for the entire block. We also describe how to *expand* a representation so that it can be combined with blocks of mis-matching dimensions.

2.1 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 termed s, u, and o, respectively. Our data structure also contains the matrices **A**, **B**, **C**, and **D** with dimensions $s \times s$, $s \times o$, $u \times s$, and $u \times o$, respectively. The inputs to a filter are denoted by $\vec{\mathbf{u}}$ (length o), the outputs by $\vec{\mathbf{y}}$ (length u), and the states by $\vec{\mathbf{x}}$ (length s). Upon every execution of the filter, we can update the state vector by the formula $\vec{\mathbf{x}} = \mathbf{A}\vec{\mathbf{x}} + \mathbf{B}\vec{\mathbf{u}}$ and calculate the outputs by the formula $\vec{\mathbf{y}} = \mathbf{C}\vec{\mathbf{x}} + \mathbf{D}\vec{\mathbf{u}}$. For convenience, we calculate the filter outputs before updating the state vector. 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 always set one of the state variables to be the constant 1. This variable is not updated by any of the inputs or states (besides itself), and its initial value is 1, so it always remains 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 (denoted by e) equals its pop rate, the data structure as currently described can fully represent the filter. We must make 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 $\vec{\mathbf{x}}$. Thus, 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 a filter is executed for the first time, it has access to the o items in the input vector, but the $e - o$ states it needs have yet to be copied from the input. Therefore, we need to initialize the state vector before iteratively computing the output and state update equations. We introduce a new matrix $\mathbf{B_{pre}}$ to perform this initialization. Before the filter runs, it performs the state update $\vec{\mathbf{x}} = \overrightarrow{\mathbf{initVec}} + \mathbf{B_{pre}}\vec{\mathbf{u}}_{\mathbf{pre}}$. The initialization input vector $\vec{\mathbf{u}}_{\mathbf{pre}}$ 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*.

Putting these pieces together, 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 initialization state matrix and an initial pop rate. We define a state space representation $\mathcal{R}$ as the tuple $\langle u,\ o,\ s,\ stored,\ \mathbf{A},\ \mathbf{B},\ \mathbf{C},\ \mathbf{D},\ \overrightarrow{\mathbf{initVec}},\ \mathbf{B_{pre}},\ o_{pre}\rangle$. When we introduce a representation $\mathcal{R}_i$, 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 matrix, we write $\mathbf{B_{pre}} = null$ and $o_{pre} = 0$. In this case, the filter does not have any stored inputs, so $stored = 0$ as well.

Representations are initially created from StreamIt filters and ultimately converted back to StreamIt filters. Between these steps, however, representations of hierarchical StreamIt blocks can be derived by combining the representations of their parts. Thus, from now on we will say that a representation refers to a block rather than a filter. The exception is in the following section, where we discuss how to create a representation from a StreamIt filter.

2.2 Extraction

We use a simple dataflow analysis to extract a state space representation from the imperative code in a filter's work function. While an alternate approach would be to allow

the user to specify the representation explicitly (as part of the program), StreamIt aims to provide a unified development environment in which diverse filters are implemented using a small set of primitives. State space filters are simple to implement using imperative StreamIt constructs, and in this form they are immediately readable by programmers unfamiliar with the state space formalism.

Our dataflow analysis symbolically executes a single iteration of a filter's work function, maintaining a vector pair representation for each local variable and filter field that is encountered (together, these are termed program variables). If the outputs and fields (i.e., states) all have vector pair representations, then the filter is linear state space, and the vectors are used as rows of **A**, **B**, **C**, and **D**. Of course, many filters do not fit the state space model; the optimizations developed in this paper are selectively applied to the portions of the stream graph that contain state space filters.

We attempt to find a vector pair ($\vec{\mathbf{v}}$,$\vec{\mathbf{w}}$) for each program variable p such that $p = \vec{\mathbf{v}} \cdot \vec{\mathbf{x}} + \vec{\mathbf{w}} \cdot \vec{\mathbf{u}}$, where $\vec{\mathbf{x}}$ is the filter's state vector and $\vec{\mathbf{u}}$ is the filter's input vector . When p is on the left hand side of an assignment statement, terms from the right hand side are identified as states (corresponding to entries of $\vec{\mathbf{x}}$) and inputs (corresponding to entries of $\vec{\mathbf{u}}$). The coefficients from terms that match are used 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 p 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 filter field, 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. A filter field has the initial vector pair ($\begin{bmatrix} 0 & \dots & 1 & \dots & 0 \end{bmatrix}$, $\vec{\mathbf{0}}$), where the 1 corresponds to the field itself.

If a vector pair is found for a given program variable p, then p can be written as a linear combination of the inputs and state variables, with the vector pair entries representing the weights. The final assignment to state variable x_i by some program variable p_k indicates that the ith rows of **A** and **B** should be $\vec{\mathbf{v}}_\mathbf{k}$ and $\vec{\mathbf{w}}_\mathbf{k}$, respectively. Similarly, if the ith push statement uses program variable p_k, then the ith rows of **C** and **D** should be $\vec{\mathbf{v}}_\mathbf{k}$ and $\vec{\mathbf{w}}_\mathbf{k}$, respectively.

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

Finally, consider the stored input states (call them $\vec{\mathbf{x}}_\mathbf{s}$). 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* $- o$ input states, and the remaining o input states are updated by the o inputs. The update is described by the equation:

$$\vec{\dot{\mathbf{x}}}_\mathbf{s} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \vec{\mathbf{x}}_\mathbf{s} + \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} \vec{\mathbf{u}} \qquad (1)$$

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

$$\vec{\mathbf{x}}_\mathbf{s} = \mathbf{0} + \mathbf{I}\vec{\mathbf{u}}_\mathbf{pre}$$

2.2.1 Extraction Example

Consider another IIR filter. Unlike the example in Section 1.1, this filter uses peeking to read elements from the tape without consuming them.

```
float->float filter IIR() {
    float curr;  // example of a filter field
    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 state vector is $\begin{bmatrix} peek(0) \\ peek(1) \\ curr \\ 1 \end{bmatrix}$; the input vector is $\begin{bmatrix} peek(2) \end{bmatrix}$.

The first program variable encountered is *temp*. It is given the vector pair ($\begin{bmatrix} 1/6 & 1/6 & 0 & 0 \end{bmatrix}$, $\begin{bmatrix} 1/6 \end{bmatrix}$). The variable *curr*, as a state variable, has an initial vector pair: ($\begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$, $\begin{bmatrix} 0 \end{bmatrix}$). When *curr* is found in an assignment statement, it is given a new vector pair, constructed as the vector pair for *temp* plus 1/2 times the old vector pair for *curr*: ($\begin{bmatrix} 1/6 & 1/6 & 1/2 & 0 \end{bmatrix}$, $\begin{bmatrix} 1/6 \end{bmatrix}$). 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) and *peek*(1) are updated as in Equation 1, and the constant 1 updated to itself. Therefore, we have:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1/6 & 1/6 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 \\ 1 \\ 1/6 \\ 0 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} 1/6 & 1/6 & 1/2 & 0 \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} 1/6 \end{bmatrix}$$

$$\overrightarrow{\mathbf{initVec}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad \mathbf{B_{pre}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

The pop and push rates are both 1, and there are four states, so $o = 1$, $u = 1$, and $s = 4$. There are two stored input states, so $o_{pre} = 2$ and *stored* $= 2$.

2.3 Combination

If all blocks within a given pipeline, splitjoin, or feedback loop have state space representations, they can be combined into a single representation using the rules developed in this section. There are two benefits to combining blocks. First, combination can eliminate redundant computations across blocks. Second, combination exposes optimization opportunities, as intra-block optimizations (described in Section 3) can effectively be applied across blocks by combining the blocks first.

2.3.1 Pipeline

Consider two blocks connected in a pipeline with representations $\mathcal{R}_1$ and $\mathcal{R}_2$. We will derive the combined representation of the two blocks, denoted by $\mathcal{R}$. Suppose the output rate of $\mathcal{R}_1$ equals the input rate of $\mathcal{R}_2$ ($u_1 = o_2$). If this is not the case, we must expand one or both blocks to have their input/output rates match ($u_1' = o_2' = lcm(u_1, o_2)$). Block expansion is covered in Section 2.4. Since the output

of $\mathcal{R}_1$ (i.e., $\vec{\mathbf{y}}_1$) is equivalent to the input of $\mathcal{R}_2$ (i.e., $\vec{\mathbf{u}}_2$), we can write:

$$\begin{aligned}\vec{\mathbf{x}}_1 &= \mathbf{A_1}\vec{\mathbf{x}}_1 + \mathbf{B_1}\vec{\mathbf{u}}_1\\ \vec{\mathbf{x}}_2 &= \mathbf{A_2}\vec{\mathbf{x}}_2 + \mathbf{B_2}\vec{\mathbf{y}_1}\\ \\ \vec{\mathbf{y}}_1 &= \mathbf{C_1}\vec{\mathbf{x}}_1 + \mathbf{D_1}\vec{\mathbf{u}}_1\\ \vec{\mathbf{y}}_2 &= \mathbf{C_2}\vec{\mathbf{x}}_2 + \mathbf{D_2}\vec{\mathbf{y}}_1\end{aligned}$$

Substituting for $\vec{\mathbf{y}}_1$ yields:

$$\begin{aligned}\vec{\mathbf{x}}_2 &= \mathbf{A_2}\vec{\mathbf{x}}_2 + \mathbf{B_2}(\mathbf{C_1}\vec{\mathbf{x}}_1 + \mathbf{D_1}\vec{\mathbf{u}}_1)\\ \vec{\mathbf{y}}_2 &= \mathbf{C_2}\vec{\mathbf{x}}_2 + \mathbf{D_2}(\mathbf{C_1}\vec{\mathbf{x}}_1 + \mathbf{D_1}\vec{\mathbf{u}}_1)\end{aligned}$$

Which simplifies to:

$$\begin{aligned}\vec{\mathbf{x}}_2 &= \mathbf{A_2}\vec{\mathbf{x}}_2 + \mathbf{B_2C_1}\vec{\mathbf{x}}_1 + \mathbf{B_2D_1}\vec{\mathbf{u}}_1\\ \vec{\mathbf{y}}_2 &= \mathbf{C_2}\vec{\mathbf{x}}_2 + \mathbf{D_2C_1}\vec{\mathbf{x}}_1 + \mathbf{D_2D_1}\vec{\mathbf{u}}_1\end{aligned}$$

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:

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

$$\mathbf{A} = \begin{bmatrix}\mathbf{A_1} & \mathbf{0}\\ \mathbf{B_2C_1} & \mathbf{A_2}\end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix}\mathbf{B_1}\\ \mathbf{B_2D_1}\end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix}\mathbf{D_2C_1} & \mathbf{C_2}\end{bmatrix} \qquad \mathbf{D} = \mathbf{D_2D_1}$$

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

If neither block has an initialization matrix, then the entire pipeline does not need an initialization matrix, so $\mathbf{B_{pre}} = \mathit{null}$, $o_{pre} = 0$, and $\mathit{stored} = 0$. If only the first block has an initialization matrix, then we initialize the states in the pipeline corresponding to the first block while keeping the states corresponding to the second block unchanged:

$$\mathbf{B_{pre}} = \begin{bmatrix}\mathbf{B_{pre1}}\\ \mathbf{0}\end{bmatrix} \qquad o_{pre} = o_{pre1} \qquad \mathit{stored} = \mathit{stored}_1$$

If the second block has an initialization matrix, the first block must run 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. A full description of this case appears in [2, pp. 46-49].

If there are more than two blocks in a pipeline, they can be collapsed in the following manner: first combine the first two blocks to get one block representation, then combine this representation with the third block, and so on.

2.3.2 Splitjoin and Feedback Loop

The combination rules for splitjoins and feedback loops are somewhat involved, and we omit them due to space considerations. An important benefit of a linear state space representation over a linear representation is that feedback loops can be collapsed; the items on the feedback path become states in the combined block. A thorough treatment of these cases appears in [2, pp. 36-43].

2.4 Expansion

Sometimes it is necessary to simulate multiple executions of a block in order to combine it properly with other blocks. For example, suppose block B_1 outputs two items and block B_2 inputs five 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, a method is needed 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:

$$\begin{aligned}\vec{\mathbf{x}} &= \mathbf{A}\vec{\mathbf{x}} + \mathbf{B}\vec{\mathbf{u}}_1\\ \vec{\mathbf{y}}_1 &= \mathbf{C}\vec{\mathbf{x}} + \mathbf{D}\vec{\mathbf{u}}_1\end{aligned}$$

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:

$$\begin{aligned}\vec{\mathbf{x}} &= \mathbf{A}(\mathbf{A}\vec{\mathbf{x}} + \mathbf{B}\vec{\mathbf{u}}_1) + \mathbf{B}\vec{\mathbf{u}}_2\\ \vec{\mathbf{y}}_2 &= \mathbf{C}(\mathbf{A}\vec{\mathbf{x}} + \mathbf{B}\vec{\mathbf{u}}_1) + \mathbf{D}\vec{\mathbf{u}}_2\end{aligned}$$

Simplifying yields:

$$\begin{aligned}\vec{\mathbf{x}} &= \mathbf{A}^2\vec{\mathbf{x}} + \mathbf{AB}\vec{\mathbf{u}}_1 + \mathbf{B}\vec{\mathbf{u}}_2\\ \vec{\mathbf{y}}_2 &= \mathbf{CA}\vec{\mathbf{x}} + \mathbf{CB}\vec{\mathbf{u}}_1 + \mathbf{D}\vec{\mathbf{u}}_2\end{aligned}$$

Let $\vec{\mathbf{u}}$ denote the combined input vector ($\vec{\mathbf{u}} = \begin{bmatrix}\vec{\mathbf{u}}_1\\ \vec{\mathbf{u}}_2\end{bmatrix}$) and $\vec{\mathbf{y}}$ denote 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:

$$\begin{aligned}\vec{\mathbf{x}} &= \mathbf{A_2}\vec{\mathbf{x}} + \mathbf{B_2}\vec{\mathbf{u}}\\ \vec{\mathbf{y}} &= \mathbf{C_2}\vec{\mathbf{x}} + \mathbf{D_2}\vec{\mathbf{u}}\end{aligned}$$

$$\mathbf{A_2} = \mathbf{A}^2 \qquad \mathbf{B_2} = \begin{bmatrix}\mathbf{AB} & \mathbf{B}\end{bmatrix}$$

$$\mathbf{C_2} = \begin{bmatrix}\mathbf{C}\\ \mathbf{CA}\end{bmatrix} \qquad \mathbf{D_2} = \begin{bmatrix}\mathbf{D} & \mathbf{0}\\ \mathbf{CB} & \mathbf{D}\end{bmatrix}$$

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:

$$\mathbf{A_n} = \mathbf{A}^n \qquad \mathbf{B_n} = \begin{bmatrix}\mathbf{A}^{n-1}\mathbf{B} & \mathbf{A}^{n-2}\mathbf{B} & \ldots & \mathbf{AB} & \mathbf{B}\end{bmatrix}$$

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

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

Since initializations are not affected, $\overrightarrow{\mathbf{initVec}}$, $\mathbf{B_{pre}}$, *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 $u_{new} = n * u_{old}$ and $o_{new} = n * o_{old}$.

As mentioned in Section 2.3.1, it is sometimes necessary to simulate the initialization stage of a block (in addition to simulating n executions) for the purpose of initializing a full pipeline. In this case, the equations are very similar to above, but also include terms for $\mathbf{B_{pre}}$. Full details appear in [2, pp. 45-46].

3. OPTIMIZATIONS

We consider two types of optimizations. The first is to remove redundant state variables from the linear state space representation. This reduces the memory allocation for a program as well as the number of loads and stores, which are typically slow and power-hungry operations. It also eliminates computations that involve the removed states. The second optimization is to reduce the parametrization of a state space representation by refactoring the matrices to contain more zero and one entries. This directly eliminates computations, as the compiler statically evaluates $0 \cdot x = 0$ and $1 \cdot x = x$ rather than performing the multiplications at runtime. Both the state removal optimization and parameter reduction optimization are formulated as a series of general transformations on the underlying state space representation.

3.1 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:

$$\mathbf{T}\dot{\vec{\mathbf{x}}} = \mathbf{TA}\vec{\mathbf{x}} + \mathbf{TB}\vec{\mathbf{u}}$$

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

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

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 above equations yields:

$$\begin{aligned} \dot{\vec{\mathbf{z}}} &= \mathbf{TAT}^{-1}\vec{\mathbf{z}} + \mathbf{TB}\vec{\mathbf{u}} \\ \vec{\mathbf{y}} &= \mathbf{CT}^{-1}\vec{\mathbf{z}} + \mathbf{D}\vec{\mathbf{u}} \end{aligned}$$

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

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

3.2 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. If there are two redundant states in a filter, then both may reachable and observable as the program is written. However, following a series of transformations, one of the redundant states can be converted to an unreachable or unobservable state, allowing it to be removed.

More formally, the ith state is reachable if and only if at least one of the following is true:

1. The state is initialized to a non-zero value. That is, the ith entry of $\overrightarrow{\mathbf{initVec}}$ is non-zero or $\exists j$ s.t. $\mathbf{B_{pre}}[i, j] \neq 0$.
2. The state directly depends on an input. That is, $\exists j$ s.t. $\mathbf{B}[i, j] \neq 0$.
3. The state directly depends on another reachable state. That is, $\exists j \neq i$ s.t. $\mathbf{A}[i, j] \neq 0$ and j is a reachable state.

All states in the system are either reachable or unreachable. Unreachable states always have a value of zero, as they are initialized to zero and are never updated by a non-zero value (i.e., by a reachable state or an input). Therefore, unreachable states can be removed from the state space representation, since they have no effect on any other states or output values.

The ith state is observable if and only if at least one of the following is true:

1. An output directly depends on the state. That is, $\exists j$ s.t. $\mathbf{C}[j, i] \neq 0$.
2. Another observable state directly depends on the state. That is, $\exists j \neq i$ s.t. $\mathbf{A}[j, i] \neq 0$ and j is an observable state.

All states in the system are either observable or unobservable. 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).

There is a simple algorithm to refactor the states of a system and expose the unreachable and unobservable states [25]. For unreachable states, the algorithm assumes that there is no initialization stage, i.e., that $\overrightarrow{\mathbf{initVec}}$ and $\mathbf{B_{pre}}$ are zero. We first describe the basic algorithm and then extend it to handle the initialization stage.

To detect unreachable states, the algorithm performs row operations[2] on the augmented matrix $\left[\begin{array}{cc} \mathbf{A} & \mathbf{B} \end{array}\right]$. To maintain the proper input/output relationship of the system, corresponding inverse column operations are performed on $\mathbf{A}$ and $\mathbf{C}$. The row operations achieve a special type of row-echelon form. In this form, the last non-zero entry in each row is a 1 (called the ending 1) and the ending 1 in a given row is to the left of the ending 1 in lower rows. Once the augmented matrix is in the desired form, row i represents an unreachable state if there are no non-zero entries past the ith column. This naturally expresses the constraint that the ith state does not depend on any input (columns of $\mathbf{B}$) or on any possibly reachable state (later columns of $\mathbf{A}$). In the absence of an initialization stage, all unreachable states identified in this way can be removed from the system.

[2]Performing a row operation operation on a matrix is equivalent to left-multiplying it by some invertible matrix, while performing a column operation is equivalent to right-multiplying by some invertible matrix.

For unobservable states, the same procedure is applied to the augmented matrix $\left[\begin{array}{cc}\mathbf{A}^T & \mathbf{C}^T\end{array}\right]$. In the echelon form, row i represents an unobservable state if there are no non-zero entries past the ith column. Intuitively, the rows of the transposed matrices represent how a given state is used, rather than how it is calculated. The identified states are unobservable because they are used neither in the calculation of an output (columns of $\mathbf{C}^T$) nor in possibly observable states (later columns of $\mathbf{A}^T$). All of these unobservable states can be safely removed from the system (even if they are assigned an initial value).

To handle the initialization stage for unreachable states, a minor extension is needed. If a state is assigned a non-zero value during initialization, either as a constant (a non-zero entry in $\overrightarrow{\mathbf{initVec}}$) or from the input (a non-zero entry in $\mathbf{B_{pre}}$), the state must be considered reachable. Further, any dependent states must also be considered reachable. This classification can easily be performed as a post-processing step on the set of candidate unreachable states identified by the algorithm above. If any candidate is initialized to a non-zero value or directly depends (via the $\mathbf{A}$ matrix) on a state outside the set, then the candidate is removed from the set. When no further candidate can be removed from the set, the set contains nothing but genuine unreachable states.

3.2.1 Expanding the Scope

So far we have considered optimizations that affect $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. Since the optimizations are entirely the result of state transformations, they do not affect $\mathbf{D}$, which is independent of the choice of state space basis. However, if all of the inputs are stored as states, then all of the entries of $\mathbf{D}$ are moved into $\mathbf{A}$ 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, the new state-equation pair is:

$$\left[\begin{array}{c}\vec{\mathbf{x}}\\ \vec{\mathbf{x}}_{\mathbf{in}}\end{array}\right] = \left[\begin{array}{cc}\mathbf{A} & \mathbf{B}\\ \mathbf{0} & \mathbf{0}\end{array}\right]\left[\begin{array}{c}\vec{\mathbf{x}}\\ \vec{\mathbf{x}}_{\mathbf{in}}\end{array}\right] + \left[\begin{array}{c}\mathbf{0}\\ \mathbf{I}\end{array}\right]\vec{\mathbf{u}}$$

$$\vec{\mathbf{y}} = \left[\begin{array}{cc}\mathbf{C} & \mathbf{D}\end{array}\right]\left[\begin{array}{c}\vec{\mathbf{x}}\\ \vec{\mathbf{x}}_{\mathbf{in}}\end{array}\right] + \mathbf{0}\vec{\mathbf{u}}$$

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 $\mathbf{D}$ which were not considered before. Removing some of these states can result in reducing computations involving $\mathbf{D}$.

3.3 Parameter Reduction

After removing as many states as possible, additional computations can be eliminated by transforming the state space system to one with fewer non-zero, non-one entries (termed parameters). If $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are completely filled, there are $s*(s+o+u)$ parameters. Ackermann and Bucy [1] show a general form in which $\mathbf{A}$ and $\mathbf{C}$ have at most $s*(o+u)$ parameters ($\mathbf{B}$ may contain any number of parameters), assuming there are no unobservable or unreachable states. They derive this form using system impulse responses. We achieve the same form using row operations on the augmented matrix $\left[\begin{array}{cc}\mathbf{A}^T & \mathbf{C}^T\end{array}\right]$. The desired form is:

$$\mathbf{A}^T = \left[\begin{array}{ccccc}\mathbf{L_1} & \mathbf{A_{12}} & \mathbf{A_{13}} & ... & \mathbf{A_{1u}}\\ \mathbf{0} & \mathbf{L_2} & \mathbf{A_{23}} & ... & \mathbf{A_{2u}}\\ \mathbf{0} & \mathbf{0} & \mathbf{L_3} & ... & \mathbf{A_{3u}}\\ ... & ... & ... & ... & ...\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & ... & \mathbf{L_u}\end{array}\right]$$

$$\mathbf{C}^T = \left[\begin{array}{ccccc}1 & 0 & 0 & ... & 0\\ 0 & 0 & 0 & ... & 0\\ ... & ... & ... & ... & ...\\ 0 & 1 & 0 & ... & 0\\ 0 & 0 & 0 & 0 & 0\\ ... & ... & ... & ... & ...\\ 0 & 0 & 0 & ... & 1\end{array}\right]$$

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

$$\mathbf{A_{ij}} = \left[\begin{array}{cccc}0 & 0 & ... & *\\ ... & ... & ... & ...\\ 0 & 0 & ... & *\end{array}\right] \quad \mathbf{L_i} = \left[\begin{array}{ccccc}0 & 0 & ... & 0 & *\\ 1 & 0 & ... & 0 & *\\ 0 & 1 & ... & 0 & *\\ ... & ... & ... & ... & ...\\ 0 & 0 & ... & 1 & *\end{array}\right]$$

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 $\mathbf{A}$ and $\mathbf{B}$ instead of $\mathbf{A}^T$ and $\mathbf{C}^T$ (there may be any number of parameters in $\mathbf{C}$ rather than $\mathbf{B}$).

Figure 5 gives pseudocode for a simple algorithm to attain the form above. The pseudocode does not include the corresponding inverse column operations that must go with all row operations.

It is possible that one type of form has fewer parameters than the other. Thus, we perform the above algorithm on $\left[\begin{array}{cc}\mathbf{A}^T & \mathbf{C}^T\end{array}\right]$ as noted to produce the observable form, as well as on $\left[\begin{array}{cc}\mathbf{A} & \mathbf{B}\end{array}\right]$ to produce the reachable form. We compare the forms and use the one with fewest parameters.

3.3.1 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—$\vec{\mathbf{x}}_{\mathbf{in1}}$, $\vec{\mathbf{x}}_{\mathbf{in2}}$, etc. After each execution, the first set is moved to the second set, the second set is moved to the third set, and so on. Suppose there are k input sets. We can write the state space equation pair as follows:

$$\left[\begin{array}{c}\vec{\mathbf{x}}\\ \vec{\mathbf{x}}_{\mathbf{ink}}\\ ...\\ \vec{\mathbf{x}}_{\mathbf{in2}}\\ \vec{\mathbf{x}}_{\mathbf{in1}}\end{array}\right] = \left[\begin{array}{ccccc}\mathbf{A} & \mathbf{B} & \mathbf{0} & ... & \mathbf{0}\\ \mathbf{0} & \mathbf{0} & \mathbf{I} & ... & \mathbf{0}\\ ... & ... & ... & ... & ...\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & ... & \mathbf{I}\\ \mathbf{0} & \mathbf{0} & \mathbf{0} & ... & \mathbf{0}\end{array}\right]\left[\begin{array}{c}\vec{\mathbf{x}}\\ \vec{\mathbf{x}}_{\mathbf{ink}}\\ ...\\ \vec{\mathbf{x}}_{\mathbf{in2}}\\ \vec{\mathbf{x}}_{\mathbf{in1}}\end{array}\right] + \left[\begin{array}{c}\mathbf{0}\\ \mathbf{0}\\ ...\\ \mathbf{0}\\ \mathbf{I}\end{array}\right]\vec{\mathbf{u}}$$

$$\vec{\mathbf{y}} = \left[\begin{array}{ccccc}\mathbf{C} & \mathbf{D} & ... & \mathbf{0} & \mathbf{0}\end{array}\right]\left[\begin{array}{c}\vec{\mathbf{x}}\\ \vec{\mathbf{x}}_{\mathbf{ink}}\\ ...\\ \vec{\mathbf{x}}_{\mathbf{in2}}\\ \vec{\mathbf{x}}_{\mathbf{in1}}\end{array}\right] + \mathbf{0}\vec{\mathbf{u}}$$

Reduce_Parameters(A, C) {

- $currRow = 0$;
- $colA = 0$;
- $colC = 0$;

while ($currRow < totalRows$) {

- Find a non-zero entry in column $colC$ at or below row $currRow$ of C^T, and swap it with the entry in row $currRow$
- Set $C^T[currRow, colC] = 1$ 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^T, and swap it with the entry in row $currRow$
- Set $A^T[currRow, colA] = 1$ 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

- $colA = colA + 1$

}

}

Figure 5: Algorithm for parameter reduction.

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-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.

4. IMPLEMENTATION

We have implemented the extraction, combination, and optimization (except multiple execution stages) procedures within the StreamIt compiler, which uses the Kopi Java Compiler infrastructure [17]. A small set of micro-benchmarks is used to demonstrate the functionality of the technique; complete code for the benchmarks appears in [2, pp. 67-93]. We measure performance by counting the number of floating point operations (additions and multiplications) executed in a given benchmark. The DynamoRIO [5, 7] system is employed to count operations.

We compare linear state space optimizations to linear optimizations [22]. For the state space numbers, all linear state space blocks are combined and then optimizations are applied. For applications this small, it is primarily the combination that yields a performance boost; we have not characterized the individual contributions of state removal and parameter reduction. For the linear numbers, all linear blocks are combined. We do not enable the "frequency translation" optimization [22] because it has no counterpart in the state space domain. A hybrid optimizer could apply frequency translation where appropriate and linear state space optimizations elsewhere.

Application	Linear State Space	Linear
Linear Difference Equation	1.00	1.00
IIR	1.00	1.00
IIR + 1/2 Decimator	0.64	1.00
IIR + 1/16 Decimator	0.34	1.00
IIR + FIR	0.94	1.00
FIR + IIR + IIR	0.92	1.00
FM Radio	0.17	0.17
FIR Program	1.00	1.00
Channel Vocoder	0.26	0.26
FilterBank2	1.00	1.00
FFT (16 pt)	2.94	3.00

Table 1: Floating point operations with state space and linear optimizations, normalized to no optimizations.

Results appear in Table 1. The first six applications have filters with state. Thus, they are not handled by linear analysis, and there is a normalized performance of 1.00 in the linear column. The first two programs (Linear Difference Equation, IIR) do not benefit from linear state space analysis, as there are no opportunities for combination, state removal, or parameter reduction. For the next four programs (IIR in combination with various filters), state space transformations offer performance improvements due to the combination of stateful filters.

For example, combining an IIR filter with a decimator that leaves 1 out of every 16 values yields a 194% improvement. Combining an IIR filter with an FIR filter offers 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 optimizations are more useful when applied to combined filters than when applied to individual filters.

For the last five applications, which mainly have linear components without state, linear optimizations and state space optimizations are equally effective. Compared to the baseline, there is an improvement of 5.9X for FM Radio and 3.8X for Channel Vocoder.

There is a large performance degradation for the Fast Fourier Transform (FFT) using either linear or state space transformations. 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 from an FFT to a DFT (Discrete Fourier Transform). We would need staged execution with minimal parameterization to convert the DFT back to an FFT. However, note that it is straightforward for the compiler to detect that the operations count has increased and refrain from combining the filters when

performance degrades. A simple algorithm for such judicious application of the transformations has been implemented for the linear case [22] and applies directly to this work. The results in Table 1 illustrate only the performance impact of blindly combining as many filters as possible. This degradation would not occur in practice.

In summary, the results show that linear state space analysis is more general than linear analysis. While the experiments consider only a small set of micro-benchmarks, they also demonstrate that a relatively elaborate mathematical framework is manageable within the compiler. It remains an important topic of future work to evaluate linear state space analysis using realistic applications, and to measure actual execution times on an embedded architecture.

5. RELATED WORK

This paper builds directly on the work done to analyze and optimize linear components in StreamIt graphs [22]. We extend the theoretical framework for linear analysis to state space analysis in order to apply our optimizations to a wider class of applications. Specifically, state space analysis applies to filters with persistent state, and feedback loops can be combined into a single state space representation; neither of these cases is handled by linear analysis. The extension from linear analysis to state space analysis required a fundamental change to the underlying representation, as well as a complete reformulation of the rules for combination and expansion. Moreover, this paper introduces novel optimizations of state removal and parameter reduction, both of which operate on the state space representation.

Potkonjak and Rabaey describe optimized hardware synthesis for linear and "feedback linear" computations [28]. Linear state space systems correspond to "constant feedback linear computations" in the authors' terminology. For linear and linear feedback systems, their technique offers 1) a maximally fast implementation under latency constraints, 2) an arbitrarily fast implementation, and 3) an implementation reducing the number of arithmetic operations. In reducing arithmetic operations, they perform common subexpression elimination (CSE) in a manner that resembles our state removal optimization.

However, the benefits of state removal cannot be achieved by CSE alone (or by the Potkonjak and Rabaey algorithm). For example, in Figure 4, state removal replaces references to *x1* + *x2* by a single variable x. While CSE can also perform this substitution, it cannot independently maintain the value of x across iterations of the filter. That is, state removal replaces the assignments to *x1*, *x2*, *x1_temp*, and *x2_temp* with a single assignment to x. This transformation decreases the number of arithmetic operations due to algebraic simplification in the update of x. Further, state removal completely removes the variables *x1*, *x2*, *x1_temp*, and *x2_temp* from the program. We are unaware of any sequence of traditional compiler optimizations that achieves the same effect as state removal (and likewise for parameter reduction).

Several other groups have developed automated frameworks for optimizing linear signal processing kernels. The SPIRAL project [30] uses a formal mathematical language to describe linear DSP operations. Using search and machine learning techniques, SPIRAL evaluates alternate versions of a formula on a given platform and optimized code is generated. The FFTW system [14] generates platform-optimized FFT libraries using a dynamic programming algorithm and profile feedback to match the recursive FFT formulation to a given memory hierarchy. ATLAS [35, 13] produces platform-specific linear algebra routines by searching over blocking strategies and other parameters; Sparsity [13, 19] applies a similar approach to sparse matrices.

While these approaches offer a rich set of optimizations, they are limited to linear systems. The transformations described in this paper apply not only to linear systems, but also to linear systems with state. In particular, the state removal and parameter reduction optimizations apply specifically to linear state space systems. In addition, our focus is on the application of linear state space optimizations in the context of a general-purpose stream language, optimizing across application components rather than within a single library function.

A variety of tools have been developed for specifying and deriving DSP algorithms [27]. The SMART project aims to develop an algebraic theory of signal processing, providing a unified framework for deriving, explaining, and classifying fast transform algorithms [29]. ADE (A Design Environment) provides a predefined set of composable signal transforms, as well as a rule-based system that searches for improved algorithms using extensible rewriting rules [12]. Janssen et al. automatically derive low-cost hardware implementations of signal flow graphs using algebraic transformations and hill-climbing search [20]. Our work shares the vision of automatically deriving optimized algorithms from a high-level description, though we start from a general-purpose, imperative stream language rather than a mathematical formalism.

The streaming abstraction has been an integral part of many programming languages, including dataflow, CSP, synchronous and functional languages; see Stephens [32] for a review. Synchronous languages which target embedded applications include Lustre [18], Esterel [6], Signal [16], Lucid [4], and Lucid Synchrone [9]. Other languages of recent interest are Brook [8], Spidle [10], Cg [24], Occam[11], Sisal [15], StreamC/KernelC [21], and Parallel Haskell [3]. The principle differences between StreamIt and these languages are (*i*) StreamIt adopts the Synchronous Dataflow [23] model of computation, which narrows the application class but enables aggressive optimizations such as linear state space analysis, (*ii*) StreamIt's support for a "peek" construct that inspects an item without consuming it from the channel, (*iii*) the single-input, single-output hierarchical structure that StreamIt imposes on the stream graph, and (*iv*) a "teleport messaging" feature for out-of-band communication [34].

6. CONCLUSIONS

As embedded applications come to support more and more functionality, the software will inevitably become more complex, and it will soon be unmanageable to satisfy tight resource constraints via manual tuning of low-level DSP code. In order to replace the signal processing expert in the design flow, compilers need to incorporate domain-specific knowledge and have a clean, unified framework for representing and optimizing the application. This paper presents one such framework, which is based on linear state space systems. Our framework provides for general extraction, expansion, and combination of state space representations.

Using state space analysis, we develop two novel optimizations for stream programs: state removal and para-

meter reduction. These transformations radically alter the internal data space of the program while maintaining the proper input/output relationship. Such transformations are generally infeasible in languages such as C due to aliasing and global variables. We implement the optimizations as part of StreamIt, a high-level stream language based on the synchronous dataflow model. Our implementation demonstrates increased generality over plain linear optimizations.

7. ACKNOWLEDGEMENTS

We thank Rodric Rabbah and the anonymous reviewers for helpful comments on this work. We also thank Andrew Lamb, whose formulation and implementation of linear analysis provided a foundation for ours. Finally, we thank Michael Gordon, Jasper Lin, Janis Sermulins, Michal Karczmarek and David Maze for their help with the StreamIt infrastructure. The StreamIt project is supported by DARPA grants PCA-F29601-03-2-0065 and HPCA/PERCS-W0133890, NSF awards CNS-0305453 and EIA-0071841, and the MIT Oxygen Alliance.

8. REFERENCES

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

[2] S. Agrawal. Linear state-space analysis and optimization of StreamIt programs. M.Eng. thesis, MIT, August 2004.

[3] S. Aidtya, Arvind, L. Augustsson, J. Maessen, and R. S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In *Haskell Workshop*, 1995.

[4] E. Ashcroft and W. Wadge. Lucid, a non procedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI*, 1999.

[6] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2), 1992.

[7] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*, March 2003.

[8] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH*, 2004.

[9] P. Caspi and M. Pouzet. The Lucid Synchrone distribution. `http://www-spi.lip6.fr/lucid-synchrone/`.

[10] C. Consel, H. Hamdi, L. Rveillre, L. Singaravelu, H. Yu, and C. Pu. Spidle: A DSL approach to specifying streaming application. In *2nd Int. Conf. on Generative Programming and Component Engineering*, 2003.

[11] I. Corporation. *Occam 2 Reference Manual*. Prentice Hall, 1988.

[12] M. M. Covell. *An Algorithm Design Environment for Signal Processing*. PhD thesis, MIT, 1989.

[13] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[14] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

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

[16] T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. *Springer Verlag LNCS*, 274, 1987.

[17] V. Gay-Para, T. Graf, A. Lemonnier, and E. Wais. *The Kopi Project*. http://www.dms.at/kopi/, 2001.

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

[19] E. Im, K. A. Yelick, and R. Vuduc. SPARSITY: An Optimization Framework for Sparse Matrix Kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[20] M. Janssen, F. Catthoor, and H. D. Man. A specification invariant technique for operation cost minimisation in flow-graphs. In *Proc. of the 7th Int. Symposium on High-level Synthesis*, pages 146–151, 1994.

[21] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, 2003.

[22] A. A. Lamb. Linear analysis and optimization of stream programs. M.Eng. thesis, MIT, May 2003.

[23] E. Lee and D. Messershmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. on Computers*, C-36(1):24–35, January 1987.

[24] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *SIGGRAPH*, 2003.

[25] D. Q. Mayne. An elementary derivation of Rosenbrock's minimal realization algorithm. *IEEE Transactions on Automatic Control*, pages 306–307, June 1973.

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

[27] A. V. Oppenheim and S. H. Nawab, editors. *Symbolic and Knowledge-Based Signal Processing*. Prentice Hall, 1992.

[28] M. Potkonjak and J. M. Rabaey. Maximally and arbitrarily fast implementation of linear and feedback linear computations. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19(1):30–43, 2000.

[29] M. Püschel and J. Moura. The algebraic approach to the discrete cosine and sine transforms and their fast algorithms. *SIAM Journal of Computing*, 32(5):1280–1316, 2003.

[30] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[31] B. D. Schutter. Minimal state-space realization in linear system theory: An overview. *Journal of Computational and Applied Mathematics*, 121(1–2):331–354+, September 2000.

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

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

[34] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport messaging for distributed stream programs. In *PPoPP*, 2005.

[35] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.