Linear Analysis and Optimization of Stream Programs

by

Andrew Allinson Lamb

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

May 2003

Author . . . .
Department of Electrical Engineering and Computer Science
May 9, 2003

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

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

Linear Analysis and Optimization of Stream Programs

by

Andrew Allinson Lamb

Submitted to the Department of Electrical Engineering and Computer Science
on May 9, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

As more complex DSP algorithms are realized in practice, there is an increasing need for high-level stream abstractions that can be compiled without sacrificing efficiency. Toward this end, we present a set of aggressive optimizations that target linear sections of a stream program. Our input language is StreamIt, which represents programs as a hierarchical graph of autonomous filters. A filter is linear if each of its outputs can be represented as an affine combination of its inputs. Linearity is common in DSP components; examples include FIR filters, expanders, compressors, FFTs and DCTs.

We demonstrate that several algorithmic transformations, traditionally hand-tuned by DSP experts, can be completely automated by the compiler. First, we present a linear extraction analysis that automatically detects linear filters from the C-like code in their work function. Then, we give a procedure for combining adjacent linear filters into a single filter, a specialized caching strategy to remove redundant computations, and a method for translating a linear filter to operate in the frequency domain. We also present an optimization selection algorithm, which finds the sequence of combination and frequency transformations that yields the maximal benefit.

We have completed a fully-automatic implementation of the above techniques as part of the StreamIt compiler. Using a suite of benchmarks, we show that our optimizations remove, on average, 86% of the floating point instructions required. In addition, we demonstrate an average execution time decrease of 450% and an 800% decrease in the best case.

Thesis Supervisor: Saman P. Amarasinghe
Title: Associate Professor

Acknowledgments

First and foremost, I would like to thank my family and fiancée. Without their continuous support and encouragement I do not know where I would be today (but it certainly wouldn't be here). I would like to thank Dave Maze, who first suggesting using FFTW and did the initial implementation; William Thies, who both conceived of and executed the automatic selection algorithm which has become a core idea of this thesis; Alex Salcianu, who provided helpful comments on the presentation of the dataflow analysis; and Michal Karczmarek, Jasper Lin, and Michael Gordon who provided extensive support with the StreamIt infrastructure. The StreamIt project is supported by DARPA contract `F26901-01-2-0166`[1], NSF contract `EIA-0071841`[2], and the MIT LCS Oxygen Alliance. Further, I would like to thank my adviser, Saman Amarasinghe, for providing an environment where I experienced the thrill of research firsthand and learned so much. Finally, I would like to thank Bill Thies. He wrote the original text of many of the sections in this thesis, and under his tutelage I have learned what research, technical writing, and hard work are all about.

[1] RAW Fabric: A Technology for Rapid Embedded Systems

[2] Cise Experimental Partnerships: MIT RAW

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter motivates our work and gives an overview of the problem domain (1.1). Next, we describe the organization of this thesis (1.2). We conclude with a motivating example that illustrates the types of optimizations our techniques automate (1.3).

1.1 Motivation and Overview

Digital computation is a ubiquitous element of modern life. Everything from cell phones to HDTV systems to satellite radios require increasingly sophisticated algorithms for digital signal processing. Optimization is especially important in this domain, as embedded devices commonly have high performance requirements and tight resource constraints. Consequently, there are often two stages to the development process: first, the algorithm is designed and simulated at a high level of abstraction, and second, it is optimized and re-implemented at a low level by an expert DSP programmer. In order to achieve high performance, the DSP programmer needs to take advantage of architecture-specific features and constraints (usually via extensive use of assembly code) as well as global properties of the application that can be exploited to obtain algorithmic speedups. Apart from requiring expert knowledge, this effort is time-consuming, error-prone, costly, and must be repeated for every change in the target architecture and every adjustment to the high-level system design. As embedded applications continue to grow in complexity, these factors will become unmanageable. There is a pressing need for high-level DSP abstractions that can be compiled without any performance penalty.

According to Texas Instruments[10], more than fifty percent of the code that runs the DSPs in a modern cell phone is written in assembly (the rest is written in annotated C). Even provided the best available C compilers, programmers must still turn to hand written code to meet the tight speed and power constraints of cell phones. Generating code tailored for the power constraints, specialized coprocessors and specialized instructions of modern DSP chips from a program written in standard C is a daunting task for modern compiler writers. The sheer volume of analysis required to automatically use special purpose instructions leaves direct assembly language programming as the only option.

In this thesis, we develop a set of optimizations that lower the entry barrier for high-performance stream programming. Our work is done in the context of StreamIt [15, 33], which is a high-level language for high performance signal processing applications. A program in StreamIt is comprised of a set of concurrently executing filters, each of which contains its own address space and communicates with its neighbors using FIFO queues. Our analysis focuses on filters which are *linear*: their outputs can be expressed as an affine combination of their inputs. Linear filters are common in DSP applications; examples include FIR filters, expanders, compressors, FFTs and DCTs.

In practice, there are a host of optimizations that are applied to linear portions of a stream graph. In particular, neighboring linear nodes can be combined into one, and large linear nodes can benefit from translation into the frequency domain. However, these optimizations require detailed mathematical analysis and are tedious and complex to implement. They are only beneficial under certain conditions — conditions that might change with the next version of the system, or that might depend on neighboring components that are being written concurrently by other developers. To improve the modularity, portability, and extensibility of stream programs, the compiler should be responsible for identifying linear nodes and performing the appropriate optimizations. Toward this end, we make the following contributions:

1. A linear dataflow analysis that extracts an abstract linear representation from imperative C-like code.
2. An automated transformation of neighboring linear nodes into a single collapsed representation.
3. An automated translation of linear nodes into the frequency domain.
4. An optimization selection algorithm that determines which transformations are most beneficial to apply.
5. A fully-automatic implementation of these techniques in the StreamIt compiler, demonstrating an average speedup of 450% and a best-case speedup of 800%.

1.2 Organization

In the rest of this chapter, we give a motivating example. In Chapter 2 we present appropriate background material on StreamIt (2.1), a brief summary of digital signal processing (2.2) and a derivation of the fast Fourier transform (2.3). In Chapter 3 we describe our analysis methods. We first present our linear node representation (3.1) and our supporting dataflow analysis (3.2). Next we describe structural transformations on linear nodes (3.3). In Chapter 4 we discuss our optimizations: frequency domain translation (4.1), redundant computation elimination (4.2), and automated application (4.3). We end with comments on our implementation experience (4.4). Chapter 5 presents experimental validation of our methods. We first describe our measurement methodology (5.1) and provide overall validation of our optimizations (5.2). Then we present additional experiments and results (5.3 - 5.8) that are of interest. We present related works in Chapter 6 and we conclude in Chapter 7 where we also mention future research opportunities (7.1).

1.3 Motivating Example

To illustrate the program transformations that our technique is designed to automate, consider a sequence of finite impulse response (FIR) filters as shown in Figure 1-1. The imperative C style code that implements this simple DSP application is

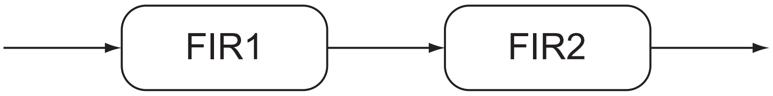


Figure 1-1: Block diagram of two FIR filters.

```
/* perform two consecutive FIR filters with weights w1, w2 */
void two_filters(float* w1, float* w2, int N) {
  int i;
  float data[N];         /* input data buffer */
  float buffer[N];       /* inter-filter buffer */

  for (i=0; i<N; i++) {  /* initialize the input data buffer */
    data[i] = get_next_input();
  }

  for (i=0; i<N; i++) {  /* initialize inter-filter buffer */
    buffer[i] = filter(w1, data, i, N);
    data[i] = get_next_input();
  }

  i = 0;
  while(true) {
    /* generate next output item */
    push_output(filter(w2, buffer, i, N));
    /* generate the next element in the inter-filter buffer */
    buffer[i] = filter(w1, data, i, N);
    /* get next data item */
    data[i] = get_next_input();
    /* update current start of buffer */
    i = (i+1)%N;
  }
}

/* perform N-element FIR filter with weights and data */
float filter(float* weights, float* data, int pos, int N) {
  int i;
  float sum = 0;

  /* perform weighted sum, starting at index pos */
  for (i=0; i<N; i++, pos++) {
    sum += weights[i] * data[pos];
    pos = (pos+1)%N;
  }
  return sum;
}
```

Figure 1-2: Two consecutive FIR filters in C. Channels are represented as circular buffers, and the scheduling is done by hand.

```
float->float pipeline TwoFilters(float[N] w1, float[N] w2) {
  add FIRFilter(w1);
  add FIRFilter(w2);
}

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

Figure 1-3: Two consecutive FIR filters in StreamIt. Buffer management and scheduling are handled by the compiler.

```
float->float filter CollapsedTwoFilters(float[N] w1, float[N] w2) {
  float[N] combined_weights;

  init {  /* calculate combined_weights from w1 and w2 */  }

  work push 1 pop 1 peek N {
    float sum = 0;
    for (int i=0; i<N; i++) {
      sum += combined_weights[i]*peek(i);
      }
    push(sum);
    pop();
  }
}
```

Figure 1-4: Combined version of the two FIR filters. Since each FIR filter is linear, the weights can be combined into a single `combined_weights` array.

```
float->float pipeline FreqTwoFilters(float[N] w1, float[N] w2) {
  float[N] combined_weights = ... ;     // calc. combined weights
  complex[N] H = fft(combined_weights); // take FFT of weights
  add FFT();                            // add FFT stage to stream
  add ElementMultiply(H);               // add multiplication by H
  add IFFT();                           // add inverse FFT
}
```

Figure 1-5: Combined version of two FIR filters in the frequency domain.

shown in Figure 1-2. The program largely defies many standard compiler analysis and optimization techniques because of its use of circular buffers and the muddled relationship between `data`, `buffer` and the output.

Figure 1-3 shows the same filtering process in StreamIt. The StreamIt version is more abstract than the C version. It indicates the communication pattern between filters, shows the structure of the original block diagram and leaves the complexities of buffer management and scheduling to the compiler.

Two optimized versions of the FIR program are shown in Figures 1-4 and 1-5. In Figure 1-4, the programmer has combined the `weights` arrays from the two filters into a single, equivalent array. This reduces the number of multiply operations by a factor of two. In Figure 1-5, the programmer has done the filtering in the frequency domain. Computationally intensive streams are more efficient in frequency than in time.

Our linear analysis can automatically derive both of the implementations in Figures 1-4 and 1-5, starting with the code in Figure 1-3. These optimizations free the programmer from the burden of combining and optimizing linear filters by hand. Instead, the programmer can design modular filters at the natural granularity for the algorithm in question and rely on the compiler for combination and transformation.

Chapter 2

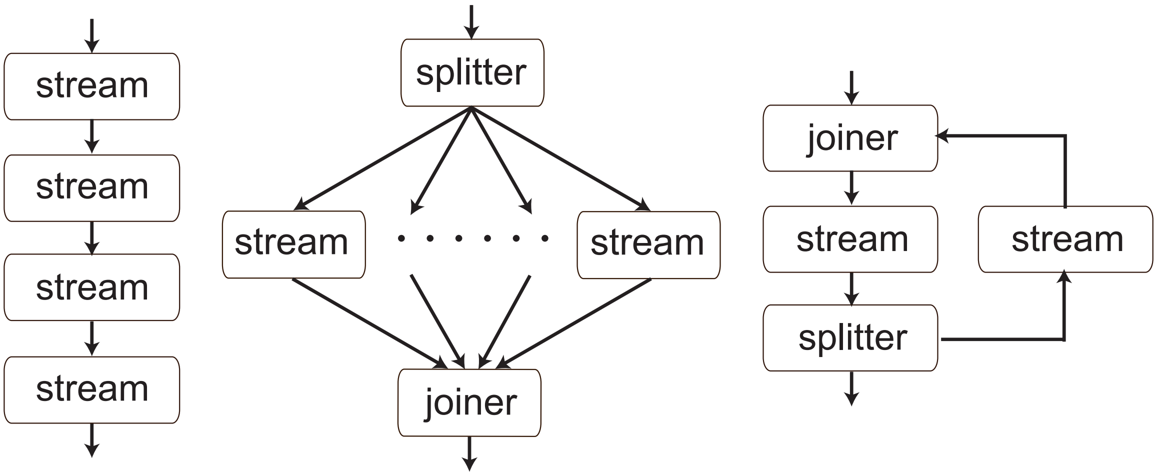
Background

This chapter is organized as follows. First we present an overview of our input language, StreamIt (2.1). Then we give a high level overview and some key results of basic digital signal processing (2.2). Finally, we provide a derivation of the FFT algorithm (2.3) as justification for our frequency replacement optimization described in Chapter 4.

2.1 The StreamIt Language

StreamIt is a language and compiler for high-performance signal processing [14, 15, 33]. In a streaming application, each data item is in the system for only a small amount of time, as opposed to scientific applications where the data set is used extensively over the entire execution. Also, stream programs have abundant parallelism and regular communication patterns. The StreamIt language aims to expose these properties to the compiler while maintaining a high level of abstraction for the programmer.

StreamIt programs are composed of processing blocks called *filters*. Each filter has an input tape from which it can read values and an output tape to which it can write values. Each filter also contains a *work* function which describes the filter's atomic execution step in the steady state. If the first invocation of the work function has different behavior than other executions, a special *initWork* function is defined.



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

Figure 2-1: Stream structures supported by StreamIt.

The work function contains C-like imperative code, which can access filter state, call external routines and produce and consume data. The input and output channels are treated as FIFO queues, which can be accessed with three primitive operations:

`pop()` returns the first item on the input tape and advances the tape by one item.

`peek(`i`)` returns the value at the ith position on the input tape

`push`(v) pushes value v onto the output tape.

Each filter must declare the maximum element it will peek at, the number of elements it will pop, and the number of elements that it will push during an execution of work. These rates must be resolvable at compile time and constant from one invocation of work to the next.

A program in StreamIt consists of a hierarchical graph of filters. Filters can be connected using one of the three predefined structures shown in Figure 2-1: 1) *pipelines* represent the serial computation of one filter after another, 2) *splitjoins* represent explicitly parallel computation, and 3) *feedbackloops* allow cycles to be introduced into the stream graph. A *stream* is defined to be either a filter, pipeline, splitjoin or feedbackloop. Every subcomponent of a structure is a stream, and all streams have exactly one input tape and exactly one output tape.

```
void->float filter FloatSource {
    float x;
    init { x = 0; }
    work push 1 { push(x++);}
}
float->void filter FloatPrinter {
    work pop 1 { print(pop()); }
}
float->float filter LowPassFilter(float g, float cutoffFreq, int N) {
  float[N] h;
  init {
    ... // initialize the N weights appropriately
  }
  /* implement the FIR filtering operation as a 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();
  }
}
float->float filter Compressor(int M) {
  work peek M pop M push 1 {
    push(pop());
    for (int i=0; i<(M-1); i++)
      pop();
  }
}
void->void pipeline Downsample {
    add FloatSource();
    add LowPassFilter(2, pi/2, 64);
    add Compressor(2);
    add FloatPrinter();
}
```

Figure 2-2: Example StreamIt program: Downsample.

It has been our experience that most applications can be represented using StreamIt's hierarchical structures. Though sometimes a program needs to be reorganized to fit into the structured paradigm, there are benefits for both the programmer and the compiler in having a structured language [33]. In particular, the linear analyses described in this thesis rely heavily on the structure of StreamIt since they focus on each hierarchical primitive rather than dealing with the complexity of arbitrary graphs.

Figure 2-2 shows an example StreamIt program that implements downsampling by a factor of two. `FloatSource` pushes an incrementing value each execution and `FloatPrinter` consumes one item and prints it to the screen. `LowPassFilter` implements a digital FIR filter[1]. `Compressor` passes the first value on its input to its output tape and discards the next $M-1$ values from the input. These filters are connected serially using the `Downsample` pipeline. The `void->void` type of the `Downsample` pipeline represents that it is the top level construct of the program.

[1]The coefficient calculations are left out for the sake of brevity.

2.2 A Crash Course in Digital Signal Processing

This section provides sufficient background in digital signal processing (DSP) for this thesis to make sense on its own. Those already endowed with an undergraduate background in signal processing can safely skip this section.

2.2.1 LTI Filtering

One of the most fundamental DSP operations is linear time-invariant (LTI) filtering. A full treatment is well outside the scope of this thesis — we aim only to provide enough background for an intelligent reading. We refer the reader to one of the many excellent introductory textbooks on the subject such as Oppenheim and Wilskey [25] for a thorough treatment.

A discrete time LTI filter has a single input and a single output which operates on signals which are defined for discrete indices. An LTI filter is completely characterized by its *impulse response* which is defined as the output of the filter when the input is the unit impulse, $\delta[n]$. The unit impulse signal, $\delta[n]$, is defined such that it takes the value 1 at index 0 and the value 0 everywhere else. Figure 2-3 graphically depicts a filter excited by $\delta[n]$ and its resulting impulse response. It is useful to divide filters into two classes as follows:

FIR Finite Impulse Response — impulse responses is of finite length.

IIR Infinite Impulse Response — impulse response is of infinite length (*e.g.* $(\frac{1}{2})^n$).

IIR filters require a feedback loop to implement, and FIR filters do not. FIR filters have other advantages over IIR filters such as better immunity to finite precision arithmetic, fewer stability concerns and highly efficient hardware implementations. Again, see Oppenheim [27] for a thorough discussion. Because of their advantages, actual applications typically use FIR filters for a majority of DSP designs.

We now explain the standard terminology of the DSP literature. The input to a filter is typically called $x[n]$ where the index n represents the discrete "time." $x[n]$ is very much like a C-style array, except that negative indices are also permissible.

Figure 2-3: Schematic of the impulse response of an LTI filter.

$x[0]$ represents the input at time 0, $x[1]$ represents the input at time 1, and so on. The impulse response of the filter is similarly described with $h[n]$. The output of a filter is called $y[n]$ and its value at time n is given by the convolution sum defined in Equation 2.1.

$$y[n] = \sum_{m=-\infty}^{\infty} x[m]h[n-m] \tag{2.1}$$

Even though Equation 2.1 contains ∞, $h[n]$ in FIR filters is non zero for a finite range of indices. Therefore, the convolution can be implemented on finite computing hardware. To calculate the output $y[n]$ at index n with a computer, we place the values of $h[n]$ in an array $weights$ such that $weights[i] = h[-i]$. There are issues involved such as shifting the $weights$ array so that it is not negatively indexed which we will not consider here. Armed with $x[n]$ and $weights[n]$ we can then calculate the output at time 0 with Equation 2.2. Equation 2.2 assumes that $weights$ is an array of N elements and that the input has more than N elements.

$$y[0] = \sum_{i=0}^{N-1} x[i]weights[i] \tag{2.2}$$

To calculate the next output value, $y[1]$, we simply shift the input by one (in C we would increment the x pointer by one) and then evaluate Equation 2.2 again. We keep calculating output values in this manner until we run out of inputs. For the corner cases at the beginning and end of the convolution sum, special care must be taken to ensure that we only use the appropriate part of the input. The convolution operation can be interpreted as computing a weighted average over the input. We slide the $weights$ array along the input array, computing the weighted sum defined by the values of $weights$ matrix.

Since the convolution sum is such a fundamental operation, DSP processors often contain specialized hardware to calculate Equation 2.1 in a highly optimized way. For example, the `FIRS` instruction [32] in the Texas Instruments TMS320C54x family of DSP processors has hardware that is capable of computing each $x[i]weights[i]$ term in a single processor cycle in the steady state.

2.2.2 Frequency Analysis

Analyzing the frequency content of signals is important to almost all fields of engineering. For discrete time signal processing, it is especially important. A fundamental way to calculate the frequency spectrum of a discrete time signal is to use the discrete time Fourier transform (DTFT). The DTFT is a continuous time function of the variable ω and is defined in Equation 2.3.

$$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n} \tag{2.3}$$

Because the complex exponential terms $e^{-j\omega n}$ are periodic, the DTFT of a discrete time signal is also periodic with a period of 2π. The periodicity of the DTFT is demonstrated in Equation 2.4.

$$\begin{aligned} X(e^{j\omega+2\pi}) &= \textstyle\sum_{n=-\infty}^{\infty} x[n]e^{-j(\omega+2\pi)n} \\ &= \textstyle\sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n}e^{-j2\pi n} \\ &= \textstyle\sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n} \\ &= X(e^{j\omega}) \end{aligned} \tag{2.4}$$

Clearly, computers can not calculate continuous functions, they can calculate the values of a function only at a finite number of points. To calculate the value of the DTFT for the N points $\omega = \frac{2\pi}{N}i$ for $i = [0..N-1]$, the DFT can be used. The DFT is defined in the next section as Equation 2.5.

2.3 FFT Derivation

This section derives the fast Fourier transform (FFT) algorithm from the definition of the discrete Fourier transform (DFT) using matrix notation. There are many different derivations for the FFT and their details are well beyond the scope of this thesis. We present a derivation for the FFT which works for DFTs of sizes which are powers of two. The derivation follows closely the derivation given by Sewell[30].

2.3.1 Notation, Definitions and Identities

In this section, we will explain the notation and prove the key identities of complex exponentials used in the FFT derivation. The N-point DFT, $X[k]$, for a N sample discrete time signal is defined by the following equation (see Oppenheim[27] for more details).

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk} \tag{2.5}$$

where W_N is the Nth root of unity such that

$$W_N = e^{-\frac{j2\pi}{N}} \quad and \quad j = \sqrt{-1} \tag{2.6}$$

Euler's relation relates complex exponentials to sinusoids in the following way.

$$e^{j\omega} = \cos(\omega) + j\sin(\omega) \tag{2.7}$$

One interpretation of $e^{j\omega}$ is a unit vector starting at the origin in the complex plane. The vector is at an angle of ω relative to the real axis as shown in Figure 2-4. This interpretation might be helpful to the reader in verifying properties of W_N. W_N plays prominently in the definition of the DFT and its properties are essential for deriving the FFT. The required identities of W_N and their derivations are shown in Equation 2.8.

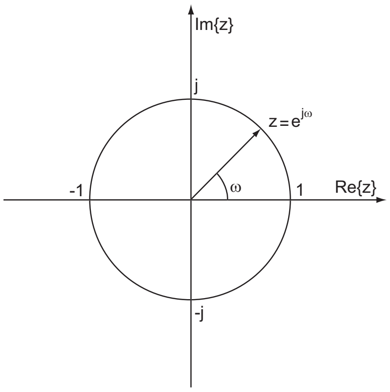


Figure 2-4: $z = e^{j\omega}$ as a unit vector in the complex plane.

$$
\begin{aligned}
W_N^2 &= e^{-\frac{j2\pi}{N}2} = e^{-\frac{j2\pi}{N/2}} = W_{N/2} \\
W_N^N &= e^{-\frac{j2\pi}{N}N} = e^{-j2\pi} = 1 \\
W_N^{N/2} &= e^{-\frac{j2\pi}{N}\frac{N}{2}} = e^{-j\pi} = -1
\end{aligned}
\tag{2.8}
$$

2.3.2 Derivation

We first cast the problem of computing the N values of $X[k]$ from the N values of $x[n]$ as a matrix multiplication such that $\vec{x}\mathbf{A} = \vec{X}$. $\vec{x}$ is a row vector of length N such that $x_i = x[i]$ and $\vec{X}$ is also a row vector of length N such that $X_i = X[i]$. We assume that N is a power of 2. If the length of $x[n]$ is not a power of two, we can "zero pad" $x[n]$ to the appropriate length. Zero padding can always be done safely. See Oppenheim[27] for a full theoretical treatment on the subject. From Equation 2.5 $\mathbf{A}$ must be a $N \times N$ matrix such that $\mathbf{A}[i,k] = W_N^{ik}$. We can write out the equation $\vec{x}\mathbf{A} = \vec{X}$ in the following way.

$$
\begin{bmatrix} x_0 \; x_1 \; \cdots \; x_{N-1} \end{bmatrix}
\begin{bmatrix}
W_N^{0\cdot 0} & W_N^{0\cdot 1} & \cdots & W_N^{0\cdot(N-1)} \\
W_N^{1\cdot 0} & W_N^{1\cdot 1} & \cdots & W_N^{1\cdot(N-1)} \\
\vdots & \vdots & \ddots & \vdots \\
W_N^{(N-1)\cdot 0} & W_N^{(N-1)\cdot 1} & \cdots & W_N^{(N-1)\cdot(N-1)}
\end{bmatrix}
=
\begin{bmatrix} X_0 \; X_1 \; \cdots \; X_{N-1} \end{bmatrix}
\tag{2.9}
$$

The non obvious but essential step of the derivation is rewriting Equation 2.9 by splitting $\vec{x}$ and $\vec{X}$ into their even indexed elements followed by their odd indexed elements (Equation 2.10). In order to maintain the same semantics, $\mathbf{A}$ must also be rewritten. With these changes, Equation 2.9 becomes Equation 2.11.

$$\begin{aligned} \left[\begin{array}{c|c} \vec{x}_{even} & \vec{x}_{odd} \end{array}\right] &= \left[\begin{array}{cccc|cccc} x_0 & x_2 & \cdots & x_{N-2} & x_1 & x_3 & \cdots & x_{N-1} \end{array}\right] \\ \left[\begin{array}{c|c} \vec{X}_{even} & \vec{X}_{odd} \end{array}\right] &= \left[\begin{array}{cccc|cccc} X_0 & X_2 & \cdots & X_{N-2} & X_1 & X_3 & \cdots & X_{N-1} \end{array}\right] \end{aligned} \tag{2.10}$$

$$\left[\begin{array}{c|c} \vec{x}_{even} & \vec{x}_{odd} \end{array}\right] \left[\begin{array}{c|c} \mathbf{P} & \mathbf{Q} \\ \hline \mathbf{R} & \mathbf{S} \end{array}\right] = \left[\begin{array}{c|c} \vec{X}_{even} & \vec{X}_{odd} \end{array}\right] \tag{2.11}$$

Our goal is to derive relationships between $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{R}$ and $\mathbf{S}$ such that we can reduce the computational requirements. We start by defining two new matrices $\mathbf{B}$ and $\mathbf{D}$. We let $\mathbf{B}$ be the $(N/2) \times (N/2)$ DFT matrix (*i.e.* $\mathbf{B}[i,k] = W_{N/2}^{ik}$).

$$\mathbf{B} = \begin{bmatrix} W_{N/2}^{0\cdot 0} & W_{N/2}^{0\cdot 1} & \cdots & W_{N/2}^{0\cdot(N/2-1)} \\ W_{N/2}^{1\cdot 0} & W_{N/2}^{1\cdot 1} & \cdots & W_{N/2}^{1\cdot(N/2-1)} \\ \vdots & \vdots & \ddots & \vdots \\ W_{N/2}^{(N/2-1)\cdot 0} & W_{N/2}^{(N/2-1)\cdot 1} & \cdots & W_{N/2}^{(N/2-1)\cdot(N/2-1)} \end{bmatrix} \tag{2.12}$$

And we let $\mathbf{D}$ be a diagonal $(N/2) \times (N/2)$ matrix such that $\mathbf{D}[i,i] = W_N^i$.

$$\mathbf{D} = \begin{bmatrix} W_N^0 & 0 & \cdots & 0 \\ 0 & W_N^1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & W_N^{(N/2-1)} \end{bmatrix} \tag{2.13}$$

We now express $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{R}$ and $\mathbf{S}$ in terms of $\mathbf{B}$ and $\mathbf{D}$.

$$
\begin{aligned}
\mathbf{P}[i,k] = \;& \mathbf{A}[2i,k] = W_N^{2ik} = W_{N/2}^{ik} = \mathbf{B}[i,k] && \Rightarrow \;\mathbf{P} = \mathbf{B} \\
\mathbf{Q}[i,k] = \;& \mathbf{A}[2i,N/2+k] = W_N^{2i(N/2+k)} = W_N^{Ni} W_N^{2ik} = 1^i W_{N/2}^{ik} = \mathbf{B}[i,k] && \Rightarrow \;\mathbf{Q} = \mathbf{B} \\
\mathbf{R}[i,k] = \;& \mathbf{A}[2i+1,k] = W_N^{(2i+1)k} = W_N^{2ik} W_N^k = W_{N/2}^{ik} W_N^k = \mathbf{B}[i,k]\mathbf{D}[k,k] && \Rightarrow \;\mathbf{R} = \mathbf{BD} \\
\mathbf{S}[i,k] = \;& \mathbf{A}[2i+1,\tfrac{N}{2}+k] = W_N^{(2i+1)(N/2+k)} = W_N^{Ni+2ik+N/2+k} && \\
= \;& W_N^{Ni} W_N^{2ik} W_N^{N/2} W_N^k && \\
= \;& (1^i) W_{N/2}^{ik}(-1) W_N^k = -W_{N/2}^{ik} W_N^i = -\mathbf{B}[i,k]\mathbf{D}[k,k] && \Rightarrow \;\mathbf{S} = -\mathbf{BD}
\end{aligned}
$$

We now rewrite Equation 2.11 in terms of $\mathbf{B}$ and $\mathbf{D}$ in the following manner:

$$
\left[\begin{array}{c|c} \vec{x}_{even} & \vec{x}_{odd} \end{array}\right]
\left[\begin{array}{c|c} \mathbf{B} & \mathbf{B} \\ \hline \mathbf{BD} & -\mathbf{BD} \end{array}\right]
= \left[\begin{array}{c|c} \vec{X}_{even} & \vec{X}_{odd} \end{array}\right] \tag{2.14}
$$

The FFT algorithm falls out of expanding Equation 2.14.

$$
\begin{array}{rcrcl}
\vec{x}_{even}\mathbf{B} & + & \vec{x}_{odd}\mathbf{BD} & = & \vec{X}_{even} \\
\vec{x}_{even}\mathbf{B} & - & \vec{x}_{odd}\mathbf{BD} & = & \vec{X}_{odd}
\end{array} \tag{2.15}
$$

We can now calculate the DFT of $\vec{x}$ using the following algorithm.

1. Compute $\vec{x}_{even}\mathbf{B}$ and $\vec{x}_{odd}\mathbf{B}$. This can be done recursively by using the fact that $\mathbf{B}$ is the $N/2$ DFT matrix — we are calculating two $N/2$-point DFTs on $\vec{x}_{even}$ and $\vec{x}_{odd}$. Since N is a power of two, $N/2$ is also a power of two. The base case is $N/2 = 1$ (*i.e.* $\mathbf{B} = [1]$).

2. Compute $\vec{u} \equiv (\vec{x}_{odd}\mathbf{B})\mathbf{D}$ using the following recursive formula which requires $N/2 + N/2 = N$ multiplications. Note that $\vec{u}$ is a $N/2$ length row vector.

$$
\begin{aligned}
\mathbf{D}[0,0] &= W_N \\
u[k] &= (\vec{x}_{odd}\mathbf{B})[k]\mathbf{D}[k,k] \\
\mathbf{D}[k+1,k+1] &= \mathbf{D}[k,k]W_N
\end{aligned} \tag{2.16}
$$

3. Calculate $\vec{X}_{even}$ and $\vec{X}_{odd}$ in the following manner:

$$\begin{aligned} \vec{X}_{even} &= \vec{x}_{even}\mathbf{B} + \vec{u} \\ \vec{X}_{odd} &= \vec{x}_{even}\mathbf{B} - \vec{u} \end{aligned} \tag{2.17}$$

4. Finally, interleave $\vec{X}_{even}$ and $\vec{X}_{odd}$ appropriately to form the overall $\vec{X}$.

The computational complexity of the above algorithm is described by the recurrence $T(N) = 2T(\frac{N}{2}) + N$ which has as its solution $T(N) = O(N \lg N)$. This is the well know result that the FFT requires $O(N \lg N)$ time.

The derivation above demonstrates the origin of the complexity savings of the FFT. There are myriads of other ways to optimize the computation of the DFT and a wide body of literature devoted to the subject. For a very fast runtime implementation, the author suggests using FFTW [6, 7, 8], which is a library for calculating the FFT which uses runtime tuning to maximize performance.

Chapter 3

Linear Analysis

In this chapter, we first describe a matrix framework which describes linear operations (3.1). Then, we present the automatic method by which our compiler detects filters that perform linear operations (3.2). We then describe the method by which we combine the action of linear filters within higher level StreamIt structures (3.3).

3.1 Representing Linear Nodes

There is no general relationship that must hold between a filter's input data and its output data. In actual applications, the output is typically derived from the input, but the relationship is not always clear since a filter can have state and can call external functions.

However, we note that a large subset of DSP operations produce outputs that are some affine function of their input, and we call filters that implement such operations *linear*. Examples of such filters are finite impulse response (FIR) filters, compressors, expanders and signal processing transforms such as the discrete Fourier transform (DFT) and discrete cosine transformation (DCT). Our formal definition of a linear node is as follows (refer to Figure 3-1 for an illustration).

Definition 1 (Linear node) *A linear node $\lambda = \{A,\ \vec{b},\ e,\ o,\ u\}$ represents an abstract stream block which performs an affine transformation $\vec{y} = \vec{x}A + \vec{b}$ from input elements $\vec{x}$ to output elements $\vec{y}$. A is an $e \times u$ matrix, $\vec{b}$ is a u-element row vector, and e, o and u are the peek, pop and push rates, respectively.*

A "firing" of a linear node λ corresponds to the following series of abstract execution steps. First, an e-element row vector $\vec{x}$ is constructed with $\vec{x}[i] = peek(e - 1 - i)$. The node computes $\vec{y} = \vec{x}A + \vec{b}$, and then pushes the u elements of $\vec{y}$ onto the output tape, starting with $\vec{y}[u-1]$ and proceeding through $\vec{y}[0]$. Finally, o items are popped from the input tape.

The notation in Definition 1 is related to the standard DSP notation introduced in Section 2.2 as follows. The input $\vec{x}$ is the same as $x[e-1-n]$, for $n \in [0, e-1]$. The input element at time 0, $x[0]$, is the same as $\vec{x}[e-1]$ and corresponds to `peek`(0). Similarly, the output $\vec{y}$ is the same as the $y[u-1-n]$ for $n \in [0, u-1]$. The first output element at time 0, $y[0]$, is the same as $\vec{y}[u-1]$ and corresponds to the first `push` statement. For a linear stream, the output is related to the input by $\vec{y}[j] = (\sum_{i=0}^{e-1} A[e-1-i, u-1-j]\vec{x}[i]) + \vec{b}[u-1-j]$.

The intuition of the computation represented by a linear node is simply that specific columns generate specific outputs and specific rows correspond to using specific inputs. The values found in row $e-1-i$ of A (*i.e.,* the ith row from the bottom) and column $u-1-j$ of A (*i.e.,* the jth column from the right) represents a term in the formula to compute the jth output item using the value of peek(i). The value in column $u-1-j$ of $\vec{b}$ is a constant offset added to output j. Figure 3-1 shows a concrete example of a work function and its corresponding linear node.

3.2 Linear Extraction Algorithm

Our linear extraction algorithm can identify a linear filter and construct a linear node λ that fully captures its behavior. The technique, which appears as Algorithm 1 and Algorithm 2, is a flow-sensitive, forward dataflow analysis similar to constant propagation. Unlike a standard dataflow analysis, we can afford to symbolically execute all loop iterations, since most loops within a filter's work function have small

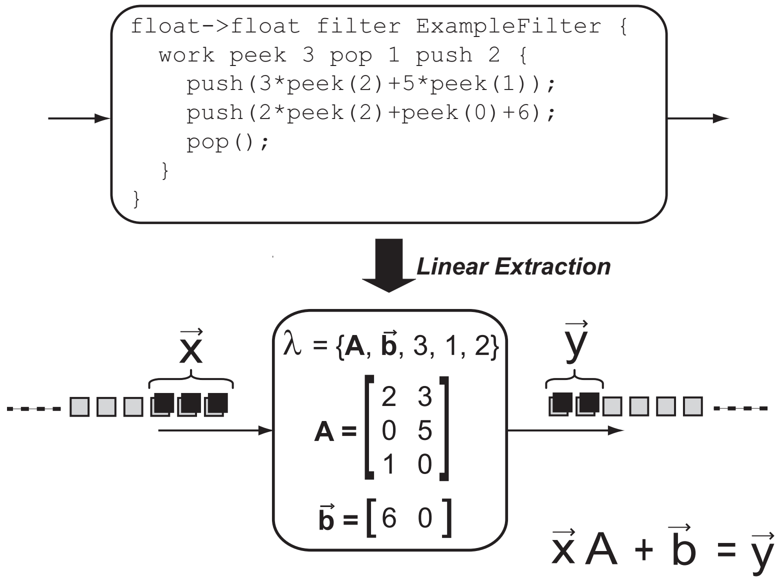


Figure 3-1: Representation of a linear node.

bounds that are known at compile time (if a bound is statically unresolvable, the filter is unlikely to be linear and we disregard it). During symbolic execution, the algorithm computes the following for each point of the program (refer to Figure 3-2 for notation):

- A *map* between each program variable y and a linear form $\langle \vec{v}, c \rangle$ where $\vec{v}$ is a *Peek*-element column vector and c is a scalar constant. In an actual execution, the value of y would be given by $y = \vec{x} \cdot \vec{v} + c$, where $\vec{x}$ represents the input items.
- Matrix A and vector $\vec{b}$, which will represent the linear node. These values are constructed during the operation of the algorithm.
- *pushcount*, which indicates how many items have been pushed so far. This is used to determine which column of A and $\vec{b}$ correspond to a given push statement.
- *popcount*, which indicates how many items have been popped so far. This is used to determine the input item that a given peek or pop expression refers to.

We now briefly discuss the operation of **Extract** at each program node. The algorithm is formulated in terms of a simplified set of instructions, which appear in Figure 3-2. First are the nodes that generate fresh linear forms. A constant assignment $y = c$ creates a form $\langle \vec{0}, c \rangle$ for y, since y has constant part c and does not

$$
\begin{array}{rcl}
y & \in & \text{program-variable} \\
c & \in & \text{constant}^\top \\
\vec{v}, \vec{b} & \in & \text{vector}^\top \\
\langle \vec{v}, c \rangle & \in & \text{linear-form}^\top \\
\mathit{map} & \in & \text{program-variable} \rightarrow \text{linear-form (a hashtable)} \\
A & \in & \text{matrix}^\top \\
\mathit{code} & \in & \text{list of instructions, each of which can be:}
\end{array}
$$

$$
\begin{array}{ll}
y_1 := \mathit{const} & \mathtt{push}(y_1) \\
y_1 := \mathtt{pop}() & (\mathtt{loop}\ N\ \mathit{code}) \\
y_1 := \mathtt{peek}(i) & (\mathtt{branch}\ \mathit{code}_1\ \mathit{code}_2) \\
y_1 := y_2\ \mathit{op}\ y_3 &
\end{array}
$$

Figure 3-2: Data types for the extraction analysis.

yet depend on the input. A pop operation creates a form $\langle \textbf{BuildCoeff}(\mathit{popcount}), 0 \rangle$, where **BuildCoeff** introduces a coefficient of 1 for the current index on the input stream. A peek(i) operation is similar, but offset by the index i.

Next are the instructions which combine linear forms. In the case of addition or subtraction, we simply add the components of the linear forms. In the case of multiplication, the result is still a linear form if either of the terms is a known constant (*i.e.,* a linear form $\langle \vec{0}, c \rangle$). For division, the result is linear only if the divisor is a non-zero constant[1] and for non-linear operations (*e.g.,* bit-level and boolean), both operands must be known constants. If any of these conditions are not met, then the LHS is assigned a value of $\top$, which will mark the filter as non-linear if the value is ever pushed.

The final set of instructions deal with control flow. For loops, we resolve the bounds at compile time and execute the body an appropriate number of times. For branches, we have to ensure that all the linear state is modified consistently on both sides of the branch. For this we apply the confluence operator $\sqcup$, which we define for scalar constants, vectors, matrices, linear forms, and maps. $c_1 \sqcup c_2$ is defined according to the lattice constant$^\top$. That is, $c_1 \sqcup c_2 = c_1$ if and only if $c_1 = c_2$; otherwise, $c_1 \sqcup c_2 =$

[1]Note that if the dividend is zero and the divisor has a non-zero coefficients vector, we cannot conclude that the result is zero, since certain runtime inputs might cause a singularity.

Algorithm 1 Linear extraction analysis.

proc **Toplevel**(filter F) returns linear node for F

1. Set globals Peek, Pop, Push to I/O rates of filter F.
2. Let $A_0 \leftarrow$ new float[Peek, Push] with each entry $= \perp$
3. Let $\vec{b}_0 \leftarrow$ new float[Push] with each entry $= \perp$
4. $(map, A, \vec{b}, popcount, pushcount) \leftarrow$
   $\quad$ $\mathbf{Extract}(F_{work}, (\lambda x.\perp), A_0, \vec{b}_0, 0, 0)$
5. **if** A and $\vec{b}$ contain no $\top$ or $\perp$ entries **then**
   return linear node $\lambda = \{A, \vec{b}, \text{Peek}, \text{Pop}, \text{Push}\}$
   else
   fail
   endif

proc **BuildCoeff**(int *pos*) returns $\vec{v}$ for peek at index *pos*

$\vec{v} = \vec{0}$
$\vec{v}[\text{Peek} - 1 - pos] = 1$
return $\vec{v}$

$\top$. For vectors, matrices, and linear forms, $\sqcup$ is defined element-wise; for example, $A' = A_1 \sqcup A_2$ is equivalent to $A'[i, j] = A_1[i, j] \sqcup A_2[i, j]$. For maps, the join is taken on the values: $map_1 \sqcup map_2 = map'$, where $map'.get(x) = map_1.get(x) \sqcup map_2.get(x)$.

Our implementation of linear extraction is also interprocedural. It is straightforward to transfer the linear state across a call site, although we omit this from the pseudocode for the sake of presentation. Also implicit in the algorithm description is the fact that all variables are local to the work function. If a filter has persistent state, all accesses to that state are marked as $\top$.

Algorithm 2 Linear analysis.

proc **Extract**(*code*, *map*, A, $\vec{b}$, int *popcount*, int *pushcount*)
returns updated *map*, A, $\vec{b}$, *popcount*, and *pushcount*

for $i \leftarrow 1$ to *code*.length
  switch *code*[i]
    case $y := const$
      $map.put(y, (\vec{0}, const))$

    case $y :=$ `pop`()
      $map.put(y, \langle \textbf{BuildCoeff}(popcount), 0\rangle)$
      popcount++

    case $y :=$ `peek`(i)
      $map.put(y, \langle \textbf{BuildCoeff}(popcount + i), 0\rangle)$

    case `push`(y)
      $\langle \vec{v}, c\rangle \leftarrow map.get(y)$
      if $pushcount = \top$ **then** *fail*
      $A[*, \mathrm{Push} - 1 - pushcount] \leftarrow \vec{v}$
      $\vec{b}[\mathrm{Push} - 1 - pushcount] \leftarrow c$
      pushcount++

    case $y_1 := y_2 \ op\ y_3$, for $op \in \{+, -\}$
      $\langle \vec{v}_2, c_2\rangle \leftarrow map.get(y_2)$
      $\langle \vec{v}_3, c_3\rangle \leftarrow map.get(y_3)$
      $map.put(y_1, \langle \vec{v}_2\ op\ \vec{v}_3, c_2\ op\ c_3\rangle)$

    case $y_1 := y_2 * y_3$
      $\langle \vec{v}_2, c_2\rangle \leftarrow map.get(y_2)$
      $\langle \vec{v}_3, c_3\rangle \leftarrow map.get(y_3)$
      if $\vec{v}_2 = \vec{0}$ **then**
        $map.put(y_1, (c_2 * \vec{v}_3, c_2 * c_3))$
      else if $\vec{v}_3 = \vec{0}$ **then**
        $map.put(y_1, (c_3 * \vec{v}_2, c_3 * c_2))$
      else
        $map.put(y_1, \top)$

    case $y_1 := y_2 / y_3$
      $\langle \vec{v}_2, c_2\rangle \leftarrow map.get(y_2)$
      $\langle \vec{v}_3, c_3\rangle \leftarrow map.get(y_3)$
      if $\vec{v}_3 = \vec{0} \wedge c_3 \neq 0$ **then**
        $map.put(y_1, (\frac{1}{c_3} * \vec{v}_2, c_2/c_3))$
      else
        $map.put(y_1, \top)$

    case $y_1 := y_2\ op\ y_3$, for $op \in \{\&, |, \wedge, \&\&, ||, !, etc.\}$
      $\langle \vec{v}_2, c_2\rangle \leftarrow map.get(y_2)$
      $\langle \vec{v}_3, c_3\rangle \leftarrow map.get(y_3)$
      $map.put(y_1, (\vec{0} \sqcup \vec{v}_2 \sqcup \vec{v}_3, c_2\ op\ c_3))$

    case (`loop` N *code*′)
      for $j \leftarrow 1$ **to** N **do**
        $(map, A, \vec{b}, popcount, pushcount) \leftarrow$
          Extract$(code, map, A, \vec{b}, popcount, pushcount)$

    case (`branch` $code_1$ $code_2$)
      $(map_1, A_1, \vec{b}_1, popcount_1, pushcount_1) \leftarrow$
        Extract$(code_1, map, A, \vec{b}, popcount, pushcount)$
      $(map_2, A_2, \vec{b}_2, popcount_2, pushcount_2) \leftarrow$
        Extract$(code_2, map, A, \vec{b}, popcount, pushcount)$
      $map \leftarrow map_1 \sqcup map_2$
      $A \leftarrow A_1 \sqcup A_2$
      $\vec{b} \leftarrow \vec{b}_1 \sqcup \vec{b}_2$
      $popcount \leftarrow popcount_1 \sqcup popcount_2$
      $pushcount \leftarrow pushcount_1 \sqcup pushcount_2$
return (*map*, A, $\vec{b}$, *popcount*, *pushcount*)

3.3 Combining Linear Filters

A primary benefit of linear filter analysis is that neighboring filters can be collapsed into a single matrix representation if both of the filters are linear. This transformation can automatically eliminate redundant computations in linear sections of the stream graph, thereby allowing the programmer to write simple, modular filters and leaving the combination to the compiler. In this section, we first describe a *linear expansion* operation that is needed to match the sizes of A and $\vec{b}$ for different linear nodes and is therefore an essential building block for the other combination techniques. We then give rules for collapsing pipelines and splitjoins into linear nodes; we do not deal with feedbackloops as they require "linear state," which we do not describe here.

3.3.1 Linear Expansion

In StreamIt programs, the input and output rate of each filter in the stream graph is known at compile time. The StreamIt compiler leverages this information to compute a static schedule — that is, an ordering of the node executions such that each filter will have enough data available to atomically execute its work function, and no buffer in the stream graph will grow without bound in the steady state. A general method for scheduling StreamIt programs is given by Karczmarek [20].

A fundamental aspect of the steady-state schedule is that neighboring nodes might need to be fired at different frequencies. For example, if there are two filters, F_1 and F_2, in a pipeline and F_1 produces 2 elements during its work function but F_2 consumes 4 elements, then it is necessary to execute F_1 twice for every execution of F_2.

Consequently, when we combine hierarchical structures into a linear node, we often need to *expand* a matrix representation to represent multiple executions of the corresponding stream. Expansion allows us to multiply and interleave columns from matrices that originally had mismatching dimensions. The transformation can be done as follows.

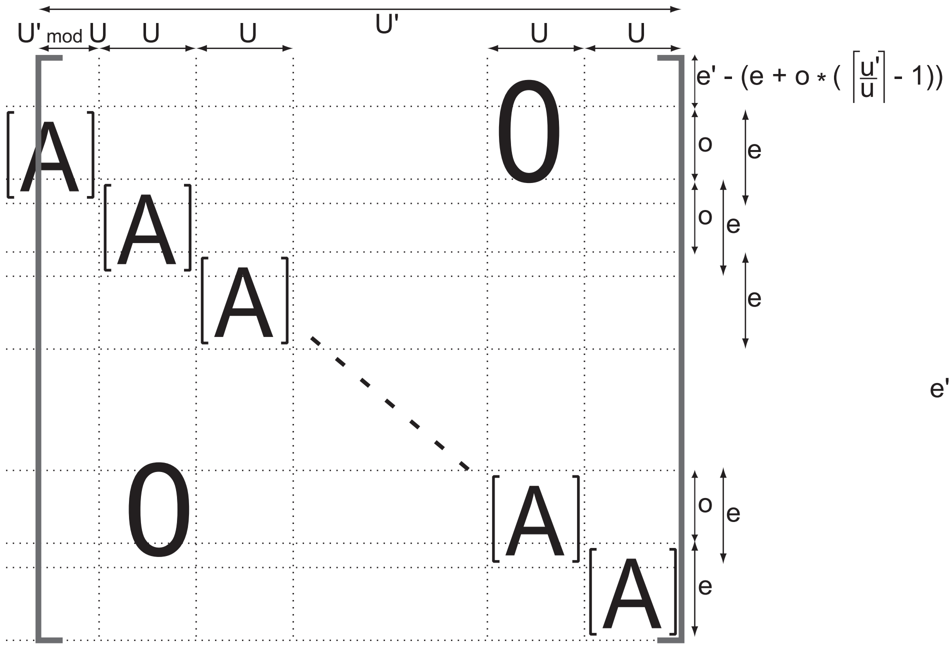


Figure 3-3: Expanding a linear node to rates (e', o', u').

Transformation 1 (Linear expansion) *Given a linear node $\lambda = \{A, \vec{b}, e, o, u\}$, the expansion of λ to a rate of (e', o', u') is given by* $\textbf{expand}(\lambda, e', o', u') = \{A', \vec{b}', e', o', u'\}$, *where $\mathbf{A}'$ is a $e' \times u'$ matrix and $\vec{\mathbf{b}}'$ is a u'-element row vector:*

$$\textit{shift}(r, c) \text{ is a } u' \times e' \text{ matrix}:$$

$$\textit{shift}(r, c)[i, j] = \begin{cases} A[i - r, j - c] \\ \quad \text{if } i - r \in [0, e-1] \wedge j - c \in [0, u-1] \\ 0 \ \textit{otherwise} \end{cases}$$

$$\begin{aligned} \mathbf{A}' &= \textstyle\sum_{m=0}^{\lceil u'/u \rceil} \textit{shift}(u' - u - m * u, e' - e - m * o) \\ \vec{\mathbf{b}}'[j] &= \vec{b}[u - 1 - (u' - 1 - j) \ \textit{mod} \ u] \end{aligned}$$

The intuition behind linear expansion is straightforward (see Figure 3-3). Linear expansion aims to scale the peek, pop and push rates of a linear node while preserving the functional relationship between the values pushed and the values peeked on a given execution. To do this, we construct a new matrix A' that contains copies of A along the diagonal starting from the bottom right. To account for items that are popped between invocations, each copy of A is offset by o from the previous copy. The complexity of the definition is due to the end cases. If the new push rate u' is not a multiple of the old push rate u, then the last copy of A includes only some of its columns. Similarly, if the new peek rate e' exceeds that which is needed by the diagonal of As, then A' needs to be padded with zeros at the top (since it peeks at some values without using them in the computation).

Note that a sequence of executions of an expanded node λ' might not be equivalent to any sequence of executions of the original node λ, because expansion resets the push and pop rates and can thereby modify the ratio between them. However, if $u' = k * u$ and $o' = k * o$ for some integer k, then λ' is completely interchangeable with λ. In the combination rules that follow, we utilize linear expansion both in contexts that do and do not satisfy this condition.

3.3.2 Collapsing Linear Pipelines

The pipeline construct is used to compose streams in sequence, with the output of stream i connected to the input of stream $i+1$. The following transformation describes how to collapse two linear nodes in a pipeline; it can be applied repeatedly to collapse any number of neighboring linear nodes.

Transformation 2 (Pipeline combination) *Given two linear nodes λ_1 and λ_2 where the output of λ_1 is connected to the input of λ_2 in a pipeline construct, the combination* $\mathbf{pipeline}(\lambda_1, \lambda_2) = \{\mathbf{A}', \vec{\mathbf{b}}', \mathbf{e}', \mathbf{o}', \mathbf{u}'\}$ *represents an equivalent node that can replace the original two. Its components are as follows:*

$$
\begin{aligned}
chanPop &= lcm(u_1, o_2) \\
chanPeek &= chanPop + e_2 - o_2 \\
\lambda_1^e &= expand(\lambda_1, (\left\lceil \frac{chanPeek}{u_1} \right\rceil - 1) * o_1 + e_1, \\
&\qquad chanPop * \tfrac{o_1}{u_1}, chanPeek) \\
\lambda_2^e &= expand(\lambda_2, chanPeek, \\
&\qquad chanPop, chanPop * \tfrac{u_2}{o_2}) \\
\mathbf{A}' &= A_1^e A_2^e \\
\vec{\mathbf{b}}' &= \vec{b}_1^e A_2^e + \vec{b}_2^e \\
\mathbf{e}' &= e_1^e \\
\mathbf{o}' &= o_1^e \\
\mathbf{u}' &= u_2^e
\end{aligned}
$$

The basic forms of the above equations are simple to derive. Let $\vec{x}_i$ and $\vec{y}_i$ be the input and output channels, respectively, for λ_i. Then we have by definition that $\vec{y}_1 = \vec{x}_1 A_1 + \vec{b}_1$ and $\vec{y}_2 = \vec{x}_2 A_2 + \vec{b}_2$. But since λ_1 is connected to λ_2, we have that $\vec{x}_2 = \vec{y}_1$ and thus $\vec{y}_2 = \vec{y}_1 A_2 + \vec{b}_2$. Substituting the value of $\vec{y}_1$ from our first equation gives $\vec{y}_2 = \vec{x}_1 A_1 A_2 + \vec{b}_1 A_2 + \vec{b}_2$. Thus, the intuition is that the two-filter sequence can be represented by matrices $A' = A_1 A_2$ and $\vec{b}' = \vec{b}_1 A_2 + \vec{b}_2$, with peek and pop rates borrowed from λ_1 and the push rate borrowed from λ_2.

There are two implicit assumptions in the above analysis which complicate the equations for the general case. First, the dimensions of A_1 and A_2 must match for the matrix multiplication to be well-defined. If $u_1 \neq e_2$, we construct expanded nodes λ_1^e and λ_2^e in which the push and peek rates match so A_1^e and A_2^e can be multiplied.

The second complication is with regards to peeking. If the downstream node λ_2 peeks at items which it does not consume (*i.e.*, if $e_2 > o_2$), then there needs to be a buffer to hold items that are read during multiple invocations of λ_2. However, in our current formulation, a linear node has no concept of internal state, such that this buffer cannot be incorporated into the collapsed representation. To deal with this issue, we adjust the expanded form of λ_1 to recalculate items that λ_2 uses more than once, thereby trading computation for storage space. This adjustment is evident in the push and pop rates chosen for λ_1^e: though λ_1 pushes u_1 items for every o_1 items that it pops, λ_1^e pushes $chanPeek * u_1$ for every $chanPop * o_1$ that it pops. When $chanPeek > chanPop$, this means that the outputs of λ_1^e are overlapping, and $chanPeek - chanPop$ items are being regenerated on every firing.

Note that although λ_1^e performs duplicate computations in the case where λ_2 is peeking, this computation cost can be amortized by increasing the value of *chanPop*. That is, though the equations set *chanPop* as the *least* common multiple of u_1 and o_2, any common multiple is legal. As *chanPop* grows, the regenerated portion $chanPeek - chanPop$ becomes smaller on a percentage basis.

It is the case that some collapsed linear nodes are always less efficient than the original pipeline sequence. The worst case is when A_1^e is a column vector and A_2^e is a row vector, which requires $O(N)$ operations originally but $O(N^2)$ operations if combined (assuming vectors of length N). To avoid such performance-degrading combinations, we employ an automated selection algorithm that only performs beneficial transformations (see Section 4.3).

Figure 3-4 illustrates the combination of back to back FIR filters. Since the push rate of the first filter ($u_1 = 1$) differs from the peek rate of the second ($e_2 = 3$), the first filter must be expanded to $\lambda_1^e = expand(\lambda_1, 4, 1, 3)$. There is no need to expand the second filter, so $\lambda_2^e = \lambda_2$. By construction, the matrix product of A_1^e and A_2^e corresponds to the matrix for the overall linear node with peek rate $e = 4$, pop rate $o = 1$ and push rate $u = 1$.

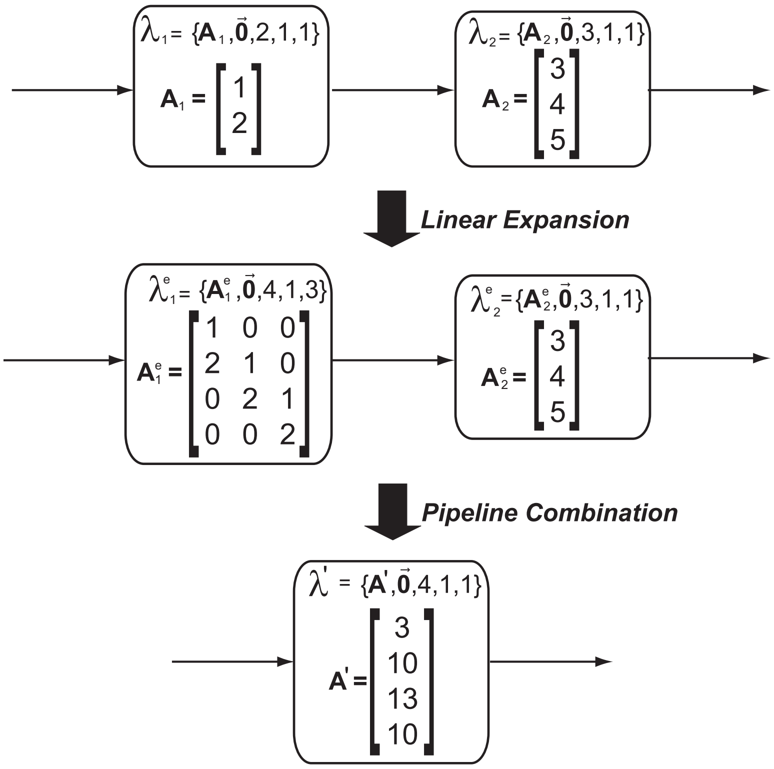


Figure 3-4: Pipeline combination example.

3.3.3 Collapsing Linear SplitJoins

The splitjoin construct allows the StreamIt programmer to express explicitly parallel computations. Data elements that arrive at the splitjoin are directed to the parallel child streams using one of two pre-defined splitter constructs: 1) *duplicate*, which sends a copy of each data item to all of the child streams, and 2) *roundrobin*, which distributes items cyclically according to an array of weights. The data from the parallel streams are combined back into a single stream by means of a roundrobin joiner with an array of weights w. First, w_0 items from the leftmost child are placed onto the overall output tape, then w_1 elements from the second leftmost child are used, and so on. The process repeats itself after $\sum_{i=0}^{n-1} w_i$ elements has been pushed.

In this section, we demonstrate how to collapse a splitjoin into a single linear node when all of its children are linear nodes. Since the children of splitjoins in StreamIt can be parameterized, it is often the case that all sibling streams are linear if any one of them is linear. However, if a splitjoin contains only a few adjacent streams that are linear, then these streams can be combined by wrapping them in a hierarchical splitjoin and then collapsing the wrapper completely. Our technique also assumes that each splitjoin admits a valid steady-state schedule; this property is verified by the StreamIt semantic checker.

Our analysis distinguishes between two cases. For duplicate splitters, we directly construct a linear node from the child streams. For roundrobin splitters, we first convert to a duplicate splitter and then rely on the transformation for duplicate splitters. We describe these translations below.

Duplicate Splitter

Intuitively, there are three main steps to combining a duplicate splitjoin into a linear node. Since the combined node will represent a steady-state execution of the splitjoin construct, we first expand each child node according to its multiplicity in the schedule. Secondly, we ensure that each child's matrix representation has the same number of rows — that is, that each child peeks at the same number of items. Once these conditions are satisfied, we can construct a matrix representation for the splitjoin by simply arranging the columns from child streams in the order specified by the joiner. Reordering columns is equivalent because with a duplicate splitter, each row of a child's linear representation refers to the same input element of the splitjoin. The transformation is described in Transformation 3.

The formulation is derived as follows. The *joinRep* variable represents how many cycles the joiner completes in an execution of the splitjoin's steady-state schedule; it is the minimal number of cycles required for each child node to execute an integral number of times and for all of their output to be consumed by the joiner. Similarly, rep_k gives the execution count for child k in the steady state. Then, in keeping with the procedure described above, λ_k^e is the expansion of the kth node by a factor of rep_k, with the peek value set to the maximum peek across all of the expanded children. Following the expansion, each λ_i^e has the same number of rows, as the peek uniformization causes shorter matrices to be padded with rows of zeros at the top.

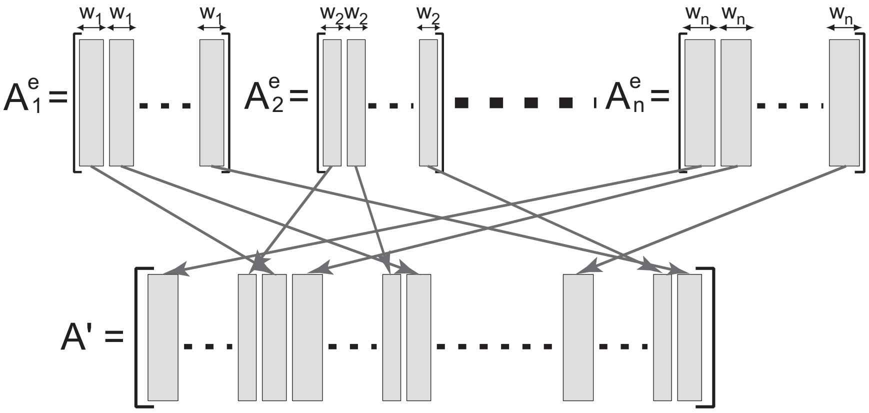


Figure 3-5: Matrix resulting from combining a splitjoin of rate-matched children.

Transformation 3 (Duplicate splitjoin combination) *Given a splitjoin s containing a duplicate splitter, children that are linear nodes $\lambda_0 \dots \lambda_{n-1}$, and a roundrobin joiner with weights $w_0 \dots w_{n-1}$, the combination* **splitjoin**$(s) = \{\mathbf{A}', \vec{\mathbf{b}}', \mathbf{e}', \mathbf{o}', \mathbf{u}'\}$ *represents an equivalent node that can replace the entire stream s. Its components are as follows:*

$$\begin{aligned} joinRep &= lcm\left(\frac{lcm(u_0, w_0)}{w_0}, \dots, \frac{lcm(u_{n-1}, w_{n-1})}{w_{n-1}}\right) \\ maxPeek &= max_i(o_i * rep_i + e_i - o_i) \end{aligned}$$

$\forall k \in [0, n-1]$:

$$\begin{aligned} wSum_k &= \textstyle\sum_{i=0}^{k-1} w_i \\ rep_k &= \frac{w_k * joinRep}{u_k} \\ \lambda_k^e &= expand(\lambda_k, maxPeek, o_k * rep_k, u_k * rep_k) \end{aligned}$$

$\forall k \in [0, n-1], \forall m \in [0, joinRep - 1], \forall p \in [0, u_k - 1]$:

$$\mathbf{A}'[*, u' - 1 - p - m * wSum_n - wSum_k] = A_k^e[*, u_k^e - 1 - p]$$

$$\vec{\mathbf{b}}'[u' - 1 - p - m * wSum_n - wSum_k] = b_k^e[u_k^e - 1 - p]$$

$$\begin{aligned} \mathbf{e}' &= e_0^e = \dots = e_{n-1}^e \\ \mathbf{o}' &= o_0^e = \dots = o_{n-1}^e \\ \mathbf{u}' &= joinRep * wSum_n \end{aligned}$$

The final phase of the transformation is to re-arrange the columns of the child matrices into the columns of A' and $\vec{b}'$ such that they generate the correct order of outputs. Though the equations are somewhat cumbersome, the concept is simple (see Figure 3-5): for the kth child and the mth cycle of the joiner, the pth item that is pushed by child k will appear at a certain location on the joiner's output tape. This location (relative to the start of the node's execution) is $p + m * wSum_n + wSum_k$, as the reader can verify. But since the right-most column of each array A holds the formula to compute the first item pushed, we need to subtract this location from the width of A when we are re-arranging the columns. The width of A is the total number of items pushed — u' in the case of A' and u_k^e in the case of A_k^e. Hence the equation as written above: we copy all items in a given column from A_k^e to A', defining each location in A' exactly once. The procedure for $\vec{b}$ is analogous.

It remains to calculate the peek, pop and push rates of the combined node. The peek rate e' is simply $maxPeek$, which we defined to be equivalent for all the expanded child nodes. The push rate $joinRep * wSum_n$ is equivalent to the number of items processed through the joiner in one steady-state execution. Finally, for the pop rate we rely on the fact that the splitjoin is well-formed and admits a schedule in which no buffer grows without bound. If this is the case, then the pop rates must be equivalent for all the expanded streams; otherwise, some outputs of the splitter would accumulate infinitely on the input channel of some stream.

These input and output rates, in combination with the values of A' and $\vec{b}'$, define a linear node that exactly represents the parallel combination of linear child nodes fed with a duplicate splitter. Figure 3-6 provides an example of splitjoin combination. The node on the left pushes four items per work function whereas the node on the right pushes one item per work function. To match the output rates to the rate of the roundrobin joiner the right filter needs to be expanded to $\lambda_2^e = expand(\lambda_2, 2, 2, 2)$. The columns of the two linear nodes are then interleaved into the overall linear node λ' as show in Figure 3-5.

Roundrobin Splitter

In the case of a roundrobin splitter, items are directed to each child stream s_i according to weight v_i: the first v_0 items are sent to s_0, the next v_1 items are sent to s_1, and so on. Since a child never sees the items that are sent to sibling streams, the items that are seen by a given child form a periodic but non-contiguous segment

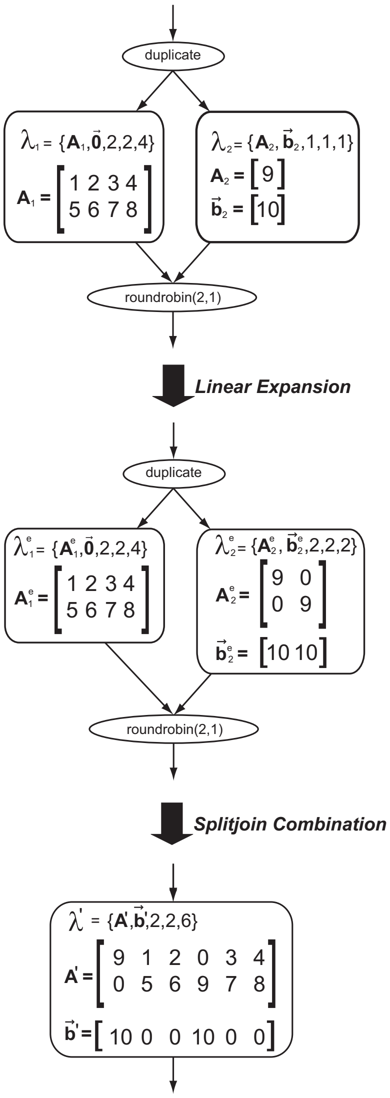


Figure 3-6: Splitjoin combination example.

of the splitjoin's input tape. Thus, in collapsing the splitjoin, we are unable to directly use the columns of child matrices as we did with a duplicate splitter, since with a roundrobin splitter these matrices are operating on disjoint sections of the input. Instead, we collapse linear splitjoins with a roundrobin splitter by converting the splitjoin to use a duplicate splitter. In order to maintain correctness, we add a decimator on each branch of the splitjoin that eliminates items which were intended for other streams.

Transformation 4 (Roundrobin to duplicate) *Given a splitjoin s containing a roundrobin splitter with weights $v_0 \ldots v_{n-1}$, children that are linear nodes $\lambda_0 \ldots \lambda_{n-1}$, and a round-robin joiner j, the transformed* **rr-to-dup**(s) *is a splitjoin with a duplicate splitter, linear child nodes $\lambda'_{\mathbf{0}} \ldots \lambda'_{\mathbf{n-1}}$, and roundrobin joiner j. The child nodes are computed as follows:*

$$
\begin{aligned}
vSum_k &= \textstyle\sum_{i=0}^{k-1} v_i \\
vTot &= vSum_n
\end{aligned}
$$

$$\forall k \in [0, n-1] :$$

$$decimate[k] \text{ is a linear node } \{A, \vec{0}, vTot, vTot, v_k\}$$

$$\text{where } A[i,j] = \begin{cases} 1 \text{ if } i = vTot - vSum_{k+1} + j \\ 0 \text{ otherwise} \end{cases}$$

$$\lambda'_{\mathbf{k}} = pipeline(decimate[k], \lambda_k)$$

In the above translation, we utilize the linear pipeline combinator *pipeline* to construct each new child node λ_i^e as a composition of a decimator and the original node λ_i. The kth decimator consists of a $vTot \times v_k$ matrix that consumes $vTot$ items, which is the number of items processed in one cycle of the roundrobin splitter. The v_k items that are intended for stream k are copied with a coefficient of 1, while all others are eliminated with a coefficient of 0.

3.3.4 Applications of Linear Combination

There are numerous instances where the linear combination transformation could benefit a programmer. For example, although a bandpass filter can be implemented with a low pass filter followed by a high pass filter, actual implementations determine the coefficients of a single combined filter that performs the same computation. While a simple bandpass filter is easy to combine manually, in an actual system several different filters might be designed and implemented by several different engineers, making overall filter combination infeasible.

Another common operation in discrete time signal processing is downsampling to reduce the computational requirements of a system. Downsampling is most often implemented as a low pass filter followed by an M compressor which passes every Mth input item to the output. In practice, the filters are combined to avoid computing dead items in the low pass filter. However, the system specification contains both elements for the sake of understanding. Our analysis can start with the specification and derive the efficient version automatically.

A final example is a multi-band equalizer, in which N different frequency bands are filtered in parallel (see our FMRadio benchmark). If these filters are time invariant, then they can be collapsed into a single node. However, designing this single overall filter is difficult, and any subsequent changes to any one of the sub filters will necessitate a total redesign of the filter. With our automated combination process, any subsequent design changes will necessitate only a recompile rather than a manual redesign.

Chapter 4

Linear Optimization

In this chapter we describe our automatic optimizing transformations of frequency replacement (4.1), redundancy removal (4.2), and optimization selection (4.3). We conclude with some implementation notes (4.4).

4.1 Translation to Frequency Domain

In this section, we demonstrate how we can leverage our linear representation to automatically perform a common domain-specific optimization: translation to the frequency domain. First, we motivate the usefulness of a translation to the frequency domain. Then, we show that a linear node is equivalent to a set of convolution sums and present a naïve code generation strategy for transforming linear nodes to frequency. Finally, we improve on our naïve implementation and present an optimized code generation strategy.

4.1.1 Motivation

Our linear analysis framework provides a compile time formulation of the computation that a linear node performs and we use this information to exploit well known domain specific optimizing transformations. Using linear node information, our compiler identifies convolution operations that require substantially fewer computations if computed in the frequency domain.

Calculating a convolution sum is a common and fundamental operation in discrete time signal processing. If the convolution is sufficiently large, transforming the data to the frequency domain, performing a vector multiply and converting back to the time domain requires fewer operations than the direct convolution.

The transformation from convolution sum into frequency multiplication has always been done explicitly by a human algorithm designer because no compiler analysis has had the information necessary to determine when a convolution sum is computed. As the complexity of DSP programs grow, determining the disparate regions across which these optimizations can be applied is an ever more daunting task. For example, several filters individually may not perform sufficiently large convolutions to merit a frequency transformation, but after a linear combination of multiple filters the transformation will be beneficial. Differing levels of architectural support for various convolution and frequency operations makes the task of determining when to apply and actually implementing specific optimizations even harder.

4.1.2 Basic Frequency Implementation

Our first goal is to show that the computation of a linear node can be represented as a convolution sum. Consider executing m iterations of a linear node $\lambda = \{A, \vec{0}, e, 1, 1\}$ — that is, a node with $\vec{b} = \vec{0}$ and push = pop = 1 (these assumptions will be relaxed below). Let $\vec{out}[i]$ represent the ith value that is pushed during execution, let $\vec{in}[i]$ hold the value of $peek(i)$ as seen before the execution begins, and let $\vec{y}$ be the convolution of the only column of A with the vector $\vec{in}$ (that is, $\vec{y} = A[*, 0] * \vec{in}$). Note that $\vec{out}$ is an m-element vector, $A[*, 0]$ is an e-element vector, $\vec{in}$ is an $(m + e - 1)$-element vector, and $\vec{y}$ is an $(m + 2e - 2)$-element vector.

Then, we make the following claim:

$$\forall i \in [0, m-1] : \quad \vec{out}[i] = \vec{y}[i + e - 1] \tag{4.1}$$

To see that this is true, recall the definition of convolution:

$$\vec{y}[i] = A[i, 0] * \vec{in}[i] = \sum_{k=-\infty}^{\infty} A[k, 0]\vec{in}[i - k] \tag{4.2}$$

Substituting $\vec{in}$ by its definition, and restricting k to range over the valid rows of A, we have:

$$\vec{y}[i] = \sum_{k=0}^{e-1} A[k, 0]\text{peek}(i - k) \tag{4.3}$$

Remapping the index i to $i + e - 1$ makes the right hand side equivalent to $\vec{out}[i]$, by Definition 1. Claim 4.1 follows.

In other words, values pushed by a linear node can be calculated by a convolution of the input tape with the coefficients A. The significance of this fact is that a convolution operation can be implemented very efficiently by using the fast Fourier transform (FFT) to translate into the frequency domain. To compute the convolution, the N-point FFTs of $\vec{in}$ and $A[*, 0]$ are calculated to obtain $\vec{X}$ and $\vec{H}$, respectively, each of which is a complex-valued vector of length N. Element-wise multiplication of $\vec{X}$ and $\vec{H}$ yields a vector $\vec{Y}$, to which the inverse transform (IFFT) is applied to obtain $\vec{y}$. Convolution in the frequency domain requires $O(N \lg(N))$ operations, as each FFT and IFFT has a cost of $O(N \lg(N))$ and the vector multiplication is $O(N)$. By contrast, the complexity is $O(N^2)$ in the time domain, as each of the N output values requires $O(N)$ operations. For more details, refer to [27].

Transformation 5 (Naïve frequency implementation) *Given a linear node $\lambda = \{A, \vec{b}, e, o, u\}$, the following stream is a naïve implementation of λ in the frequency domain:*

```
float → float pipeline naiveFreq (A, b⃗, e, o, u) {
  add float → float filter {
    N ← 2^⌈lg(2e)⌉
    m ← N − 2e + 1

    init {
      for j = 0 to u − 1
        H⃗[∗, j] ← FFT(N, A[∗, u − 1 − j])
    }

    work peek m + e − 1 pop m push u ∗ m {
      x⃗ ← peek(0 . . . m + e − 2)
      X⃗ ← FFT(N, x⃗)
      for j = 0 to u − 1 {
        Y⃗[∗, j] ← X⃗. ∗ H⃗[∗, j]
        y⃗[∗, j] ← IFFT(N, Y⃗[∗, j])
      }
      for i = 0 to m − 1 {
        pop()
        for j = 0 to u − 1
          push(y⃗[i + e − 1, j] + b⃗[j])
      }
    }
  }
  add Decimator(o, u)
}

float → float filter Decimator (o, u) {
  work peek u ∗ o pop u ∗ o push u {
    for i = 0 to u − 1
      push(pop())
    for i = 0 to u − 1
      for j = 0 to o − 2
        pop()
  }
}
```

We can use the procedure described above to implement a linear node in the frequency domain. We simply calculate $\vec{y} = A[*, 0] * \vec{in}$, and extract values $\vec{y}[e - 1] \ldots \vec{y}[m + (e - 1) - 1]$ as the m values pushed by the node. Note that $\vec{y}[i]$ is also defined for $i \in [0, e - 2]$ and $i \in [m + e - 1, m + 2e - 2]$; these values represent partial sums in which some coefficients were excluded. Our naïve implementation simply disregards these values. However, in the next section, we give an optimized implementation that takes advantage of them.

The only task remaining for the implementation is to choose N, the FFT size, and m, the number of iterations to execute at once in the frequency domain. According to Fourier's theorem, an N-point FFT can exactly represent any discrete sequence of N numbers, so the only constraint on N and m is that $N \geq m + 2e - 1$. For performance reasons, N should be a power of two and as large as possible. In our implementation, we set N to the first power of two that is greater than or equal to $2e$, and then set $m = N - 2e + 1$. This strikes a reasonable compromise between storage space and performance for our uniprocessor benchmarking platform; the choice of N should be adjusted for the particular resource constraints of the target architecture.

Transformation 5 gives a naïve translation of a linear node to the frequency domain. In addition, it relaxes all of the assumptions that we made above. The algorithm allows for a non-zero value of $\vec{b}$ by simply adding $\vec{b}$ after returning from the frequency domain. To accommodate a push rate greater than one, the algorithm generates *matrices* for $\vec{Y}$ and $\vec{y}$ and alternates pushing values from each column of $\vec{y}$ in turn. Finally, to accommodate a pop rate greater than one, the algorithm proceeds as if the pop rate was one and adds a special decimator node that discards the extra outputs. Though this introduces inefficiency by calculating values that are never used, it still leaves room for large performance improvements, as the frequency transformation can improve performance by a large factor (see Chapter 5).

4.1.3 Optimized Frequency Implementation

Transformation 6 (Optimized frequency implementation) *Given a linear node $\lambda = \{A, \vec{b}, e, o, u\}$, the following stream is an optimized implementation of λ:*

$$
\begin{array}{l}
\texttt{float} \rightarrow \texttt{float pipeline optimizedFreq } (A, \vec{b}, e, o, u) \ \{ \\
\quad \texttt{add float} \rightarrow \texttt{float filter } \{ \\
\quad\quad N \leftarrow 2^{\lceil \lg(2e) \rceil} \\
\quad\quad m \leftarrow N - 2e + 1 \\
\quad\quad \vec{partials} \leftarrow \texttt{new array}[0 \ldots e-2, 0 \ldots u-1] \\
\quad\quad r \leftarrow m + e - 1 \\
\quad\quad \texttt{init } \{ \\
\quad\quad\quad \texttt{for } j = 0 \texttt{ to } u - 1 \\
\quad\quad\quad\quad \vec{H}[*, j] \leftarrow \mathbf{FFT}(N, A[*, u-1-j]) \\
\quad\quad \} \\
\quad\quad \texttt{initWork peek } r \texttt{ pop } r \texttt{ push } u * m \ \{ \\
\quad\quad\quad \vec{x} \leftarrow \texttt{pop}(0 \ldots m + e - 2) \\
\quad\quad\quad \vec{X} \leftarrow \mathbf{FFT}(N, \vec{x}) \\
\quad\quad\quad \texttt{for } j = 0 \texttt{ to } u - 1 \ \{ \\
\quad\quad\quad\quad \vec{Y}[*, j] \leftarrow \vec{X} .* \vec{H}[*, j] \\
\quad\quad\quad\quad \vec{y}[*, j] \leftarrow \mathbf{IFFT}(N, \vec{Y}[*, j]) \\
\quad\quad\quad\quad \vec{partials}[*, j] \leftarrow \vec{y}[m + e - 1 \ldots m + 2e - 3, j] \\
\quad\quad\quad \} \\
\quad\quad\quad \texttt{for } i = 0 \texttt{ to } m - 1 \\
\quad\quad\quad\quad \texttt{for } j = 0 \texttt{ to } u - 1 \\
\quad\quad\quad\quad\quad \texttt{push}(\vec{y}[i + e - 1, j] + \vec{b}[j]) \\
\quad\quad \} \\
\quad\quad \texttt{work peek } r \texttt{ pop } r \texttt{ push } u * r \ \{ \\
\quad\quad\quad \vec{x} \leftarrow \texttt{pop}(0 \ldots m + e - 2) \\
\quad\quad\quad \vec{X} \leftarrow \mathbf{FFT}(N, \vec{x}) \\
\quad\quad\quad \texttt{for } j = 0 \texttt{ to } u - 1 \ \{ \\
\quad\quad\quad\quad \vec{Y}[*, j] \leftarrow \vec{X} .* \vec{H}[*, j] \\
\quad\quad\quad\quad \vec{y}[*, j] \leftarrow \mathbf{IFFT}(N, \vec{Y}[*, j]) \\
\quad\quad\quad \} \\
\quad\quad\quad \texttt{for } i = 0 \texttt{ to } e - 1 \\
\quad\quad\quad\quad \texttt{for } j = 0 \texttt{ to } u - 1 \ \{ \\
\quad\quad\quad\quad\quad \texttt{push}(\vec{y}[i, j] + \vec{partials}[i, j]) \\
\quad\quad\quad\quad\quad \vec{partials}[i, j] \leftarrow \vec{y}[m + e - 1 + i, j] \\
\quad\quad\quad\quad \} \\
\quad\quad\quad \texttt{for } i = 0 \texttt{ to } m - 1 \\
\quad\quad\quad\quad \texttt{for } j = 0 \texttt{ to } u - 1 \\
\quad\quad\quad\quad\quad \texttt{push}(\vec{y}[i + e - 1, j] + \vec{b}[j]) \\
\quad\quad \} \\
\quad \} \\
\quad \texttt{add Decimator}(o, u) \ \ \textit{// see Transformation 5} \\
\}
\end{array}
\tag{4.4}
$$

The naïve frequency implementation discards $e-1$ elements from the beginning and end of each column of $\vec{y}$ that it computes. These values represent partial sums in which some of the coefficients of A are excluded. However, for $i \in [0, e-2]$, $\vec{y}[i,j]$ in one iteration contains the missing terms from $\vec{y}[m+e-1+i,j]$ in the previous iteration. The sum of these two elements gives a valid output for the filter. This symmetry arises from the convolution of A "off the edges" of the input block that we consider in a given iteration. Reusing the partial sums — which is exploited in the transformation above — is one of several methods that use blocking to efficiently convolve a short filter with a large amount of input [27].

4.1.4 Applications of Frequency Transformation

The transformation to the frequency domain is straightforward in theory and very common in practice. However, the detailed record keeping, transform size selection, and state management make an actual implementation quite involved. Further, as the complexity of DSP programs continues to grow, manually determining the disparate regions across which to apply this optimization is an ever more daunting task. For example, several filters individually may not perform sufficiently large convolutions to merit a frequency transformation, but after a linear combination of multiple filters the transformation could be beneficial. Differing levels of architectural support for various convolution and frequency operations further complicates the task of choosing the best transform. Our compiler automatically determines all the necessary information and transforms the computation into the frequency domain.

4.2 Redundancy Elimination

In this section, we demonstrate another use of our linear representation: to automatically remove redundant multiplications by caching products between filter invocations. First we motivate the theory of redundancy removal. Then we show how to recognize the redundancy in a given linear node and then we present an optimized code generation strategy that takes advantage of this information.

```
float->float filter SimpleFIR {
  work peek 3 pop 1 push 1 {
    push(2*peek(2) +
         peek(1) +
         2*peek(0));
    pop();
  }
}
```

Figure 4-1: Example filter with redundant computation across firings.

4.2.1 Motivation

Redundancy removal is motivated by the observation that many linear filters compute the same value using different expressions on different work function invocations. For instance, $a*$`peek`(2) in one invocation might have exactly the same value as $a*$`peek`(0) in the next invocation. By caching a term the first time it is computed we can save the future cost of recomputing it.

In the context of DSP programs in StreamIt, we expect abundant redundancy in our programs because a large portion of our benchmarks are FIR filters. Most FIR filters have symmetric impulse responses[1] and therefore symmetry in the matrices of their linear nodes. Symmetric impulse responses are so prevalent in actual realizations that specialized hardware instructions that implement FIR filtering (e.g. `FIRS`[32]) typically assume symmetry and use that information to optimize the calculation. The prevalence of symmetry in DSP applications presents us with an obvious avenue of optimization — to remove the redundant computations. Common subexpression elimination (CSE), a standard compiler optimization, works well for removing redundant computations within a single work function invocation. Our method also removes redundant computations *across* filter firings. Therefore, our method may be thought of as a more general form of CSE for linear filters.

[1]Symmetry in the time domain implies that the filter has a constant group delay. Group delay is a measure of how much a given filter delays different frequency components of a signal. A filter with constant group delay is desirable because all frequencies of a signal are delayed equally and thus phase dispersion is avoided. Constant group delay is a very useful property in the design and analysis of actual systems.

```
float->float filter NoredundFIR {
  float[3] state; // state array of size 3
  int index;      // index into state array

  initWork peek 3 pop 1 push 1 {
    index = 0;
    state[1] = 2*peek(1);
    state[2] = 2*peek(2);
    work();
  }

  work peek 3 pop 1 push 1 {
    state[index] = 2*peek(0);
    push(state[(index+2)%3] +
         peek(1) +
         state[(index+0)%3]);
    pop();
    index = index - 1;
    if (index < 0)
      index = 3;
  }
}
```

Figure 4-2: Example filter without redundant calculations.

The analysis and optimization presented in this chapter only remove redundant multiplications by caching individual terms. It is possible to remove additions as well by storing reused partial *sums* of terms. For the general case, determining which are the optimal partial sums to compute and store is an NP hard problem. We provide no more discussion of the partial sums problem in this thesis.

Figure 4-1 and Figure 4-2 illustrate the potential advantage of a redundancy removal optimization. Figure 4-1 shows the code for a filter which performs redundant computations. The value of $2 * \texttt{peek}(2)$ is the same as the value of $2 * \texttt{peek}(0)$ two invocations in the future. This same value is calculated two different times and therefore is a candidate for removal. Figure 4-2 shows a different implementation of the same filter that avoids these redundant calculations by introducing state to cache values. The cached values are stored in the field `state`. The `state` field is used as a circular buffer with size 3, and the start of the buffer is held in the `index` field.

4.2.2 Redundancy Analysis

The goal of redundancy analysis is to identify redundant computations using a given linear node (Definition 1). To begin, we define a *linear computation tuple* (LCT):

Definition 2 (Linear computation tuple) *$t = \langle coeff, pos \rangle$ is a linear computation tuple and represents an abstract multiplication of a constant with a runtime input. t has a runtime value of $coeff * \texttt{peek}(pos)$.*

The relevance of an LCT is as follows. The u outputs for each filter with an associated linear node $\lambda = \{A, \vec{b}, e, o, u\}$ are weighted sums of the input values such that $\texttt{push}(j) = \sum_{i=0}^{i=e-1}(A[i, u-1-j]\texttt{peek}(e-1-i))$. Each of the terms in the previous summation can be represented by the LCT $t = \langle A[i, u-1-j], e-1-i \rangle$. Hence, the output of a linear filter can be exactly represented with a set of LCTs and a set of constants. A LCT is defined in terms of the input tape for the present work function invocation, and the goal of redundancy analysis is to determine which LCTs in the present firing of work represent values calculated in future invocations. Redundancy analysis produces a map of the LCTs generated in the current work function execution to their use in future work function executions. The linear analysis algorithm is presented as pseudo code in Algorithm 3. The following redundancy information is generated:

map For $\lceil \frac{e}{o} \rceil$ firings, the linear node will reuse some of the values from the input tape. For the first execution, the algorithm generates an LCT, t, for each element of A and makes a note that t was used in execution 0 (the first execution). On the second iteration, we generate LCTs for only the part of A that overlaps input values that were available on the first execution. Continuing in this manner we generate all LCTs for all future executions that use input values available to the initial execution.

minUse, maxUse *minUse* and *maxUse* record the minimum and maximum work function that uses t. If $minUse(t) = 0$, t is generated in the first work function and is a candidate for caching, but if $minUse(t) > 0$ then t represents an LCT that is computed for the first time in the future. If $maxUse(t) = 0$ then t is only used in the first work function invocation and doesn't need to be cached.

reused The *reused* set holds all LCTs that are candidates for reuse. Only LCTs that are computed in the first work function (*i.e.* $minUse(t) = 0$) and are used in subsequent work functions (*i.e.* $maxUse(t) > 0$) are members of *reused*.

Algorithm 3 Redundancy information extraction algorithm. Given a linear node $\lambda = \{A, \vec{b}, e, o, u\}$ returns: *map* which maps LCTs t to a list of integers $\{k_0, k_1, \cdots, k_{K-1}\}$ where each integer k_l denotes that t is reused in k_l firings from the from the current firing; *maxUse* and *minUse* which are the maximum and minimum future work function invocations in which t is calculated; *reused* which is the set of all LCTs that are calculated in both the first work function and a subsequent one; and *compMap* which maps an LCT *newTuple* in the current work function to the computation of *oldTuple*, *use* firings ago (*i.e.* $newTuple \rightarrow (oldTuple, use)$).

```
proc Redundant(λ = {A, b⃗, e, o, u})
returns map, maxUse, minUse, reused, and compMap.
  map ← {}
  for currentExecution ← 0 ··· ⌈e/o⌉ − 1 do
    for row ← currentExecution ∗ o ··· e − 1 do
      for col ← 0 ··· u − 1 do
        t ← ⟨currentExecution ∗ o + e − 1 − row, A[row, col]⟩
        tlist ← {map.get(t) ∪ currentExecution}
        map.put(t, tlist)
      end for
    end for
  end for
  for each t ∈ map do
    maxUse(t) ← max i : i ∈ map.get(t)
    minUse(t) ← min i : i ∈ map.get(t)
  end for
  reused ← all t : minUse(t) = 0 and maxUse(t) > 0
  for each t ∈ reused do
    compMap.put(t, (0, t))
    for each i ∈ map.get(t) do
      nt ← ⟨t.coeff, t.pos − i ∗ o⟩
      if minUse(nt) = 0 and i > compMap.get(nt).use then
        compMap.put(nt, (t, i))
      end if
    end for
  end for
```

compMap *compMap* has mappings of the form $newTuple \rightarrow (oldTuple, use)$. It maps an LCT in the current work function invocation (*newTuple*) to an equivalent LCT (*oldTuple*) and during which previous work function the old LCT was generated (*use*). This mapping is used to determine which cached LCT value can be used as the value of *newTuple* in the current work function.

4.2.3 Non Redundant Code Generation

With the information contained in *map*, *maxUse*, *minUse*, *reused*, and *compMap* we present the following transformation, an optimized code generation strategy that removes redundant computations with caching. The code generated by Transformation 7 keeps two state variables for each LCT whose value is cached. *tupleState* is a circular buffer that contains the cached values and *tupleIndex* is an index into the buffer representing the location of values computed during the present work function. The code in initWork initializes *tupleState* with the values that would have been computed in prior work functions. The code in work first computes and stores the values of the reused LCTs for the current work function. Then, the set of terms whose sum makes up each output value is computed using either the stored values from *tupleState* or direct calculation.

Note that for any t, its value is computed if $t \notin \mathit{reused}$. If $t \in \mathit{reused}$ and $(\mathit{ot}, \mathit{use}) \leftarrow \mathit{compMap}(t)$, then the value of t is located in $\mathit{tupleState}(\mathit{ot})[(\mathit{tupleIndex}(\mathit{ot})+ \mathit{use})\ \mathit{mod}\ (\mathit{maxUse}(\mathit{ot}) + 1)]$. The fact that $\mathit{tupleState}(t)$ is a circular buffer of size $\mathit{maxUse}(t) + 1$ can be seen in the mod expression of the index value. Finally, it remains to update the *tupleIndex* value for each t. We decrement the index and reset it to $\mathit{maxUse}(t) + 1$ if it is below zero.

While the Transformation 7 does very well at removing redundant computation, it introduces too much overhead to be practical (see Chapter 5). The overhead introduced is the memory required to save $\sum_{t \in reused}(maxUse(t) + 1)$ values, the code to store $|\mathit{reused}|$ values each invocation, and the code to load each cached value when it is needed. Each load and store instruction also carries the cost of an address calculation. The overall time required for loads and stores is much greater than the time required to simply recompute the value.

Transformation 7 (Redundancy elimination) *Given a linear node* $\lambda = \{A, \vec{b}, e, o, u\}$ *and (map, maxUse, minUse, reused, compMap)* $\leftarrow$ **Redundant**(λ) *the following stream is an optimized implementation of* λ*:*

$$
\begin{array}{l}
\texttt{float} \rightarrow \texttt{float filter noRedund } (map, A, \vec{b}, e, o, u)\ \{\\
\ \texttt{init}\ \{\\
\ \ \texttt{for } \mathit{each}\ t \in \mathit{reused}\ \{\\
\ \ \ \mathit{tupleState}(t) \leftarrow \texttt{new array}[0 \ldots \mathit{maxUse}(t)]\\
\ \ \ \mathit{tupleIndex}(t) \leftarrow 0\\
\ \ \}\\
\ \}\\
\\
\ \texttt{initWork}\ \{\\
\ \ \texttt{for } \mathit{each}\ t \in \mathit{reused}\\
\ \ \ \texttt{for } \mathit{use} \leftarrow 1 \ldots \mathit{maxUse}(t)\\
\ \ \ \ \mathit{tupleState}(t)[\mathit{use}] \leftarrow (t.\mathit{coeff}) * \texttt{peek}(t.\mathit{pos} - o * \mathit{use})\\
\ \ \texttt{work}()\\
\ \}\\
\\
\ \texttt{work}\ \{\\
\ \ \texttt{for } \mathit{each}\ t \in \mathit{reused}\\
\ \ \ \mathit{tupleState}(t)[\mathit{tupleIndex}(t)] \leftarrow (t.\mathit{coeff}) * \texttt{peek}(t.\mathit{pos})\\
\ \ \texttt{for } \mathit{col} \leftarrow u - 1 \ldots 0\ \{\\
\ \ \ \mathit{termList} \leftarrow \{\}\\
\ \ \ \texttt{for } \mathit{row} \leftarrow 0 \ldots e - 1\ \{\\
\ \ \ \ t \leftarrow \langle A[\mathit{row}, \mathit{col}], e - 1 - \mathit{row}\rangle\\
\ \ \ \ \texttt{if } \mathit{compMap.contains}(t)\ \texttt{then}\ \{\\
\ \ \ \ \ (\mathit{ot}, \mathit{use}) \leftarrow \mathit{compMap.get}(t)\\
\ \ \ \ \ \mathit{termList} \leftarrow \mathit{termList} \cup \{\mathit{tupleState}(\mathit{ot})[(\mathit{tupleIndex}(\mathit{ot}) + \mathit{use})\ \mathit{mod}\ (\mathit{maxUse}(\mathit{ot}) + 1)]\}\\
\ \ \ \ \}\ \texttt{else}\ \{\\
\ \ \ \ \ \mathit{termList} \leftarrow \mathit{termList} \cup \{A[\mathit{row}, \mathit{col}] * \texttt{peek}(e - 1 - \mathit{row})\}\\
\ \ \ \ \}\\
\ \ \ \}\\
\ \ \ \texttt{push}(\sum_{i \in \mathit{termList}}(i) + b[\mathit{col}])\\
\ \ \}\\
\ \ \texttt{for } \mathit{each}\ t \in \mathit{reused}\ \{\\
\ \ \ \mathit{tupleIndex}(t) \leftarrow \mathit{tupleIndex}(t) - 1\\
\ \ \ \texttt{if } (\mathit{tupleIndex}(t)) < 0\ \texttt{then}\\
\ \ \ \ \mathit{tupleIndex}(t) \leftarrow \mathit{maxUse}(t) + 1)\\
\ \ \}\\
\ \ \texttt{pop}(o)\\
\ \}
\end{array}
\tag{4.5}
$$

There are several obvious optimizations to improve the generated code that we did not pursue because the potential savings didn't justify the effort. The floating point unit in modern CPUs is too efficient to merit reducing its workload using the memory subsystem. An optimized transformation will not be competitive with simply doing the redundant calculations unless 1) the ratio of the costs of computation to memory access increases drastically or 2) redundancy was super abundant. We feel that most filters are not going to be rich enough in intra firing redundancy to merit the redundancy removal technique and the redundancy within the same work function can be eliminated by traditional CSE. Below is a list of optimizations that we considered but did not implement:

1. Make all buffer sizes powers of 2 and use a bitmask expression instead of a modulo expression for index calculations.
2. Make cache aware (architecture dependent) memory layout decisions.
3. Put a minimum reuse floor for values to be cached. For example, only cache a value that is reused more than 10 times.
4. Implement a solution to the partial sum problem.

4.3 Optimization Selection

To reap the maximum benefit from the optimizations described in the previous two sections, it is important to apply them selectively. There are two components of the optimization selection problem: first, to determine the sequence of optimizations that will give the highest performance for a given arrangement of the stream graph, and second, to determine the arrangement of the stream graph that will give the highest performance overall. In this section, we explain the relevance of each of these problems, and we outline an effective selection algorithm that relies on dynamic programming to quickly explore a large space of configurations. The selection algorithm was both conceived and implemented by William Thies. We include this section in the interest of completeness — many of the results in Chapter 5 rely on it. For more information, please refer to [22] and [34].

```
// types of transformations we consider for each stream
enum Transform { ANY, LINEAR, FREQ, NONE }

// a tuple representing a cost and a stream
struct Config {
   int cost             : cost of the configuration
   Stream str           : Stream corresponding to the lowest cost
}

// a hierarchical stream element
struct Stream {
   int height           : number of rows in the container
   int width[y]         : number of columns in row y
   int child[x, y]      : child in position (x, y)  [column x, row y]
}
```

Figure 4-3: Type declarations for code in Figures 4-4, 4-5, and 4-6.

4.3.1 The Selection Problem

The selection of optimizations for a given stream graph can have a large impact on performance. As alluded to in Section 3.3, linear combination can increase the number of arithmetic operations required, *e.g.,* if combining a two-element pipeline where the second filter pushes more items than it peeks. However, such a combination might be justified if it enables further combination with other components and leads to a benefit overall. Another consideration is that as the pop rate grows, the benefit of converting to frequency diminishes; thus, it might be preferable to transform smaller sections of the graph to frequency, or to perform linear combination only. The arrangement of the stream graph might also constrain the transformations that are possible. Since our transformations operate on an entire pipeline or splitjoin construct, the graph often needs to be refactored to put linear nodes in their own hierarchical unit.

4.3.2 Dynamic Programming Solution

Our optimization selection algorithm, shown in Figure 4-3, Figure 4-4, Figure 4-5, and Figure 4-6, automatically derives the example transformations described above. Intuitively, the algorithm works by estimating the minimum cost for each structure in the stream graph. The minimum cost represents the best of three configurations: 1) collapsed and implemented in the time domain, 2) collapsed and implemented in the

```
// global variable holding the lowest-cost Config for nodes
// (x1..x2, y1..y2) of Stream <s> if Transform <t> is applied
Config memoTable[s, t, x1, x2, y1, y2]

// given original Stream <s>, return optimized stream
Stream toplevel (Stream s)
  initialize all entries of memoTable to Config(-1, null)
  return getCost(s, ANY).str
```

```
// returns lowest-cost Config for Stream <s> under Transform <t>
Config getCost (Stream s, Transform t)
  if (t = ANY)
    c1 ← getCost(s, LINEAR)
    c2 ← getCost(s, FREQ)
    c3 ← getCost(s, NONE)
    return ci s.t. ci.cost = min(c1.cost, c2.cost, c3.cost)
  else if (s is Node)  return getNodeCost(s, t)
  else // s is Container
    maxWidth ← max(s.width[0], ..., s.width[s.height-1])
    return getContainerCost(s, t, 0, maxWidth-1, 0, s.height-1)
```

Figure 4-4: Algorithm for optimization selection (part one).

---

frequency domain, and 3) uncollapsed and implemented as a hierarchical unit. The cost functions for the collapsed cases are guided by profiler feedback, as described elsewhere. For the uncollapsed case, the cost is the sum of each child's minimum cost. However, instead of considering the children directly, the children are refactored into many different configurations, and the cost is taken as the minimum over all configurations. This allows the algorithm to simultaneously solve for the best set of transformations and the best arrangement of the stream graph.

4.3.3 Cost Functions

The pseudocode in Figure 4-4, Figure 4-5, and Figure 4-6 refers to functions *getDirectCost* and *getFrequencyCost* that estimate a node's execution time if implemented in the time domain or the frequency domain. These cost functions can be tailored to a specific architecture and code generation strategy. For example, if there is architecture-level support for convolution operations, then this would effect the cost for certain dimensions of matrices; similarly, if a matrix multiplication algorithm is

```
// returns lowest-cost Config for Node <s> under Transform <t>
Config getNodeCost (Stream s, Transform t)
  // scale cost by the number of times <s> executes in the steady-state schedule
  scalingFactor ← executionsPerSteadyState(s)

  if (t = LINEAR)
    if (isLinear(s)) return Config(scalingFactor × getDirectCost(s),
                                   makeLinearImplementation(s))
    else return Config(∞, s)

  else if (t = FREQ)
    if (isLinear(s) ∧ canConvertToFrequency(s))
      return Config(scalingFactor × getFrequencyCost(s),
                    makeFreqImplementation(s))
    else return Config(∞, s)

  else // t = NONE
    if (isLinear(s)) return Config(scalingFactor × getDirectCost(s), s)
    else return Config(0, s)   // don't tally up costs of non-linear nodes
```

Figure 4-5: Algorithm for optimization selection (part two).

available that exploits symmetry or sparsity in a matrix, then this benefit could be accounted for where it applies. In our implementation, we use the following versions of the cost functions. Note that both cost functions are undefined if s is non-linear ($i.e.$, if there is no corresponding λ_s). In this case they are assigned infinite cost. The motivation for this choice of cost functions is described in [22] and [34]. Let $\lambda = (A, \vec{b}, e, o, u)$ be the linear node corresponding to stream s:

$$getDirectCost(s) = \begin{cases} \infty & \text{(if } s \text{ is roundrobin splitjoin)} \\ 185 + 2 * u + & \text{(otherwise)} \\ \quad |\{i \ s.t.\ \vec{b}_i \neq 0\}| + & \\ \quad 3 * |\{(i,j) \ s.t.\ A_{i,j} \neq 0\}| & \end{cases}$$

$$getFrequencyCost(s) = \left[185 + 2 * u + u * ln\left(\frac{1 + 4 * e}{1 + \frac{2^{\lceil lg(2*e) \rceil}}{50}} \right) \right] * max(o, 1) + dec(s)$$

$$dec(s) = (o - 1) * (185 + 4 * u)$$

```
// returns lowest-cost Config for children (x1..x2, y1..y2) of <s> under <t>
Config getContainerCost (Stream s, Transform t, int x1, int x2, int y1, int y2)
  // if we've exceeded the width of this node, then trim down to actual width
  x2 ← min (x2, max (width[y1], ..., width[y2]) - 1)

  // if value is memoized, return it
  if (memoTable[s, t, x1, x2, y1, y2] ≠ -1)
     return memoTable[s, t, x1, x2, y1, y2]

  if (x1 = x2 ∧ y1 = y2)   // if down to one child, descend into it
     result ← getCost(s.child[x1, y1], t)

  // if the transform will collapse children, then treat them as a single node
  if (t = LINEAR ∨ t = FREQ)
     result ← getNodeCost(extractSubstream(s, x1, x2, y1, y2), t)

  if (t = NONE)
     result = Cost (∞, s)
     // try horizontal cut
     for yPivot ← y1 to y2-1 do
        // get cost of 2-element Pipeline; remember Config if it is best so far
        c1 ← getCost(s, ANY, x1, x2, y1, yPivot)
        c2 ← getCost(s, ANY, x1, x2, yPivot+1, y2)
        if (c1.cost + c2.cost < result.cost)
           result ← Config(c1.cost+c2.cost, Pipeline(c1.str, c2.str))

     // can only do vertical cut if all child streams belong to same splitjoin
     if (sameSplitJoinParent(s.child[x1, y1], s.child[x2, y2]))
        for xPivot = x1 to x2-1 do
           // get cost of 2-element SplitJoin; remember Config if it is best so far
           c1 ← getCost(s, ANY, x1, xPivot, y1, y2)
           c2 ← getCost(s, ANY, xPivot+1, x2, y1, y2)
           if (c1.cost + c2.cost < result.cost)
              result ← Config(c1.cost+c2.cost, SplitJoin(c1.str,c2.str))

  memoTable[s, t, x1, x2, y1, y2] ← result
  return result
```

Figure 4-6: Algorithm for optimization selection (part three).

4.4 Implementation Notes

This section presents some notes about our implementation. The StreamIt compiler is built upon the KOPI [13] java compiler infrastructure. The *stream intermediate representation* (SIR) is used as the internal representation of StreamIt programs. Each node of the SIR represents a stream construct: filter, pipeline, splitjoin or feedback-loop. Filter SIR nodes contain the code for the filter's work function(s). Functions are represented with modified versions of the KOPI *intermediate representation* (IR) nodes. The StreamIt IR nodes consist of minimally modified KOPI IR nodes and several new nodes that represent StreamIt specific constructs like `peek`, `pop`, and `push`. At compile time, the structure of the SIR mirrors the hierarchal stream structure present in the original program. Compiler passes in KOPI are implemented using the *visitor* design pattern [9] to visit each IR node. There are two types of visitors. One type iterates over a program's SIR representation (the stream structure) and the other type iterates over the IR nodes in a given function body (the C-like code).

Our linear analysis is implemented as two different types of visitors, one for each type of IR. The linear analyzer visits each of the program's SIR nodes and determines the associated linear node (or that the SIR node is nonlinear). When analyzing filters, the linear analyzer sends another visitor through the IR nodes that make up the body of the work function. The IR visitor implements the dataflow analysis algorithm presented in Section 3.2. There are very few differences between the implementation and the abstract algorithm. As the linear analyzer progresses through the SIR nodes, it keeps a mapping of SIR nodes to linear nodes. When a hierarchal construct such as a pipeline or splitjoin is encountered, the combination rules defined in Section 3.3 are applied where possible to determine the linear nodes that correspond to the overall construct. At the end of the analysis pass, the linear analyzer has generated a complete mapping of SIR nodes to linear nodes. This mapping is used as input by the optimization passes.

Both linear replacement and frequency replacement require replacing various nodes in the SIR tree. The SIR in StreamIt is mutable, so visitors can modify the program as they descend through the hierarchy. The linear replacement and the frequency replacement visitors replace the appropriate nodes in the SIR with new filters that either 1) implement the corresponding linear node directly (linear replacement) or 2) use the FFT (frequency replacement). The IR nodes that represent the body of the new work functions are built directly within the compiler. Automatic IR generation is cumbersome but it allows the new filters to behave like any other SIR node within the compiler. As a general rule, the fewer special cases required the better. In the StreamIt compiler, not special casing means that the unmodified linear analysis and linear replacement passes were usable with the Raw backend and needed almost no extra work. Unfortunately, the frequency replacement optimization needs an *external routine* to efficiently calculate the FFT. As our external library of choice, FFTW, has not been compiled or tested on the Raw architecture, the frequency replacement optimization is unusable with the Raw backend at the present time.

FFTW is used with the uniprocessor backend. To keep the impact of calling FFTW localized and to avoid adding unnecessary external dependencies, we take advantage of the existing infrastructure to add the necessary callouts to FFTW. At present, the output from the StreamIt uniprocessor backend must be linked against a runtime library[2] anyway. This affords an ideal location for the required hooks. Because creating code in the IR is so cumbersome, the frequency replacer generates only the code necessary for the external interface from within the compiler. The generated code relies on additional handwritten C code linked in with the runtime system to perform the details. Below we describe our interface approach at a high level.

Automatically generated frequency filters first copy data from the input tape into a temporary buffer and call an external function that converts the buffer to the frequency domain. The external function handles the details of interfacing with FFTW (such as generating the appropriate FFTW plan). When control returns to the work function, the original buffer contains the transform of the input data in

[2]The runtime library provides support for managing buffer resources (tapes) among other housekeeping chores.

half-complex form. Then, an external function is called to multiply the transformed input data with the transformed impulse response. One final external call arranges for FFTW to transform the output data back to the time domain. Finally, the work function pushes the contents of the buffer onto the output tape. Note that the overhead we incur by copying data to the buffer is more than made up for by our algorithmic savings and the speed of FFTW. Please see Chapter 5 for details.

One interesting optimization (directly due to FFTW) is using half-complex or "Hermitian" arrays to store frequency transforms. Since the FFT of a real sequence is symmetric, only $\frac{N}{2}$ complex values need to be stored to represent an N-point FFT. Our wrapper functions take advantage of this fact and all operations are done on half-complex arrays . The external routines are slightly more complicated when using the half-complex format (corner cases require special attention) but since they are written in C and not generated by the compiler, the additional complexity for the sake of speed is acceptable.

Chapter 5

Results

This chapter is organized as follows: First, we introduce our measurement methodology (5.1). Then, we present results that validate our overall optimizations in the context the StreamIt compiler (5.2). Finally, we present experimental results that provide additional justification of our methods: the effect of combination on performance (5.3); the effect of using ATLAS (5.4); the effect of problem size scaling on the FIR benchmark (5.5); the effect of removing redundancy (5.6); the effect of problem size scaling on the Radar benchmark (5.7); and the effect of implementation choices on the frequency transformation (5.8).

5.1 Measurement Methodology

The StreamIt compiler currently has two code generation backends. The uniprocessor backend generates sequential C code that is compiled and linked against a supporting library. The second backend generates code for the Raw microprocessor [35, 24], which features a grid of processors interconnected via various communication structures. We chose to use the uniprocessor backend for our measurements to control for the variability inherent in mapping a StreamIt program to the Raw architecture. This mapping is complicated by issues such as communication, load balancing and partitioning [15]. It is difficult to separate the effects of our optimizations from the effects of the different tile placement, routing, and fusion which result from modifying the program's structure.

The most appropriate performance metric is always debatable. In this thesis, we choose to measure the strength of our optimizations in terms of execution time and floating point instruction counts.

Execution Time Execution time is a complex function of the operating environment, complicated cache dependencies, and the super scalar architectures of modern processors. Additionally, the uniprocessor backend for the StreamIt compiler is meant for prototyping and the supporting runtime library is not optimized for speed. Improvements in uniprocessor running time *do* provide an indication of the potential real world benefits of our optimizations, but they should be interpreted with the above factors in mind.

Operation Reduction Our optimizations reduce the fundamental amount of computation required by our benchmark programs. To capture this reduction, we present the changes in floating point operation count alongside timing information. Our optimizations most commonly remove multiplications, so in the rest of this chapter we also present reductions in multiplications. In this thesis, our measurement platform uses the IA-32 instruction set. We define FLOPS as in Table 5.1[1]. We define multiplication instructions to be any of `fmul fmulp fimulp fdiv fdivp fidivp fdivr fdivrp fidivr`.

Our measurement platform is a dual 2.2 GHz Intel P4 Xenon system with 2GB of memory running GNU/Linux. We compile our benchmarks using StreamIt's uniprocessor backend and generate executables from the resulting C files using `gcc -O2`. To measure the number of floating point operations, we use an instruction counting DynamoRIO[1] client. We normalize both execution time and instruction counts to the number of program outputs generated. Since StreamIt is a new language, there are no external sources of benchmarks. Thus, we have assembled the following set of representative streaming components and have rewritten them in StreamIt:

[1]By Bill Thies from `http://courses.ece.uiuc.edu/ece291/resources/instructionref.pdf`

Flop?	Opcode	Description
▷	FABS	Absolute Value
▷	FADD	Add
▷	FADDP	Add
▷	FIADD	Add
	FBLD	Load Binary Coded Decimal
	FBSTP	Store BCD Integer and Pop
▷	FCHS	Change Sign
	FCLEX	Clear Exceptions
	FNCLEX	Clear Exceptions
	FCMOVcc	Floating-Point Conditional Move
▷	FCOM	Compare Floating Point Values
▷	FCOMP	Compare Floating Point Values
▷	FCOMPP	Compare Floating Point Values
▷	FCOMI	Compare Floating Point Values and Set EFLAGS
▷	FCOMIP	Values and Set EFLAGS
▷	FUCOMI	Values and Set EFLAGS
▷	FUCOMIP	Values and Set EFLAGS
▷	FCOS	Cosine
	FDECSTP	Decrement Stack-Top Pointer
▷	FDIV	Divide
▷	FDIVP	Divide
▷	FIDIV	Divide
▷	FDIVR	Reverse Divide
▷	FDIVRP	Reverse Divide
▷	FIDIVR	Reverse Divide
	FFREE	Free Floating-Point Register
▷	FICOM	Compare Integer
▷	FICOMP	Compare Integer
	FILD	Load Integer
	FINCSTP	Increment Stack-Top Pointer
	FINIT	Initialize Floating-Point Unit
	FNINIT	Initialize Floating-Point Unit
	FIST	Store Integer
	FISTP	Store Integer
	FLD	Load Floating Point Value
	FLD1	Load Constant
	FLDL2T	Load Constant
	FLDL2E	Load Constant
	FLDPI	Load Constant
	FLDLG2	Load Constant
	FLDLN2	Load Constant
	FLDZ	Load Constant
	FLDCW	Load x87 FPU Control Word

Flop?	Opcode	Description
	FLDENV	Load x87 FPU Environment
▷	FMUL	Multiply
▷	FMULP	Multiply
▷	FIMUL	Multiply
	FNOP	No Operation
▷	FPATAN	Partial Arctangent
▷	FPREM	Partial Remainder
▷	FPREM1	Partial Remainder
▷	FPTAN	Partial Tangent
▷	FRNDINT	Round to Integer
	FRSTOR	Restore x87 FPU State
	FSAVE	Store x87 FPU State
	FNSAVE	Store x87 FPU State
▷	FSCALE	Scale
▷	FSIN	Sine
▷	FSINCOS	Sine and Cosine
▷	FSQRT	Square Root
	FST	Store Floating Point Value
	FSTP	Store Floating Point Value
	FSTCW	Store x87 FPU Control Word
	FNSTCW	Store x87 FPU Control Word
	FSTENV	Store x87 FPU Environment
	FNSTENV	Store x87 FPU Environment
	FSTSW	Store x87 FPU Status Word
	FNSTSW	Store x87 FPU Status Word
▷	FSUB	Subtract
▷	FSUBP	Subtract
▷	FISUB	Subtract
▷	FSUBR	Reverse Subtract
▷	FSUBRP	Reverse Subtract
▷	FISUBR	Reverse Subtract
▷	FTST	TEST (compares with 0.0)
▷	FUCOM	Unordered Compare Floating Point Values
▷	FUCOMP	Unordered Compare Floating Point Values
▷	FUCOMPP	Unordered Compare Floating Point Values
	FWAIT	Wait
▷	FXAM	Examine
	FXCH	Exchange Register Contents
	FXRSTOR	Restore x87 FPU, MMX, SSE, and SSE2 State
	FXSAVE	Save x87 FPU, MMX, SSE, and SSE2 State
▷	FXTRACT	Extract Exponent and Significand
▷	FYL2X	Compute y * log2x
▷	FYL2XP1	Compute y * log2(x +1)

Table 5.1: Intel IA-32 FLOPS.

FIR a single 256 coefficient low pass FIR filter.

RateConvert an audio down sampler that converts the sampling rate by a non-integral factor ($\frac{2}{3}$).

TargetDetect four matched filters in parallel with threshold target detection.

FMRadio an FM software radio with equalizer.

Radar the core functionality in modern radar signal processors, based on a system from the Polymorphic Computing Architecture [23].

FilterBank a multi-rate signal decomposition processing block common in communications and image processing.

Vocoder a channel voice coder, commonly used for speech analysis and compression.

Oversampler a 16x oversampler, a function found in many audio CD players.

DToA an audio post-processing stage prior to a 1-bit D/A converter with an oversampler and a first order noise shaper.

Table 5.2 shows characteristics of our benchmarks both before and after our optimizations. The stream graphs are presented in Appendix B.

5.2 Overall Performance

We have completed a fully automatic implementation of the linear combination, frequency replacement, and optimization selection algorithms described in the previous chapter. In this section, we evaluate three configurations of linear optimizations for the uniprocessor backend:

- *Linear replacement*, which transforms maximal linear sections of the stream graph into a single linear node, which we implement as a matrix multiply. For small nodes (less than 256 operations), this takes the form of an unrolled arithmetic expression, whereas for large nodes we implement an indexed matrix multiply that avoids zero entries at the top and bottom of each column.

- *Frequency replacement*, which transforms maximal linear sections of the stream graph into a single node which performs the calculation in the frequency domain.

	Originally			
Benchmark	**Filters (linear)**	**Pipelines (linear)**	**SplitJoins (linear)**	**Average vector size**
FIR	3 (1)	1(0)	0 (0)	256
RateConvert	5 (3)	2 (1)	0 (0)	102
TargetDetect	10 (4)	6 (0)	1 (0)	300
FMRadio	26 (22)	11 (9)	2 (2)	40
Radar	76 (60)	17 (0)	2 (0)	4412
FilterBank	27 (24)	17 (9)	4 (4)	52
Vocoder	17 (13)	10 (8)	2 (1)	60
Oversampler	10 (8)	1 (1)	0 (0)	33
DToA	14 (10)	3 (1)	0 (0)	52

	After Optimizations		
Benchmark	**Filters**	**Pipelines**	**SplitJoins**
FIR	3	1	0
RateConvert	4	1	0
TargetDetect	7	1	1
FMRadio	5	1	0
Radar	38	17	2
FilterBank	3	1	0
Vocoder	6	2	1
Oversampler	3	1	0
DToA	7	2	0

Table 5.2: Characteristics of benchmarks before and after running automatic selection optimizations.

To implement the necessary basis conversions, we use FFTW [6, 7, 8], which is an adaptive and high-performance FFT library.

- *Automatic selection*, which employs both of the previous transformations judiciously in order to obtain the maximal benefit. We use the algorithm presented in Section 4.3.

One interesting aspect of our optimizations is that they eliminate floating point operations (FLOPS) from the program, as shown in Figure 5-1. Figure 5-2 shows that multiplies are removed in roughly the same proportion. The removal of FLOPS represents fundamental computation savings that is independent of the streaming runtime

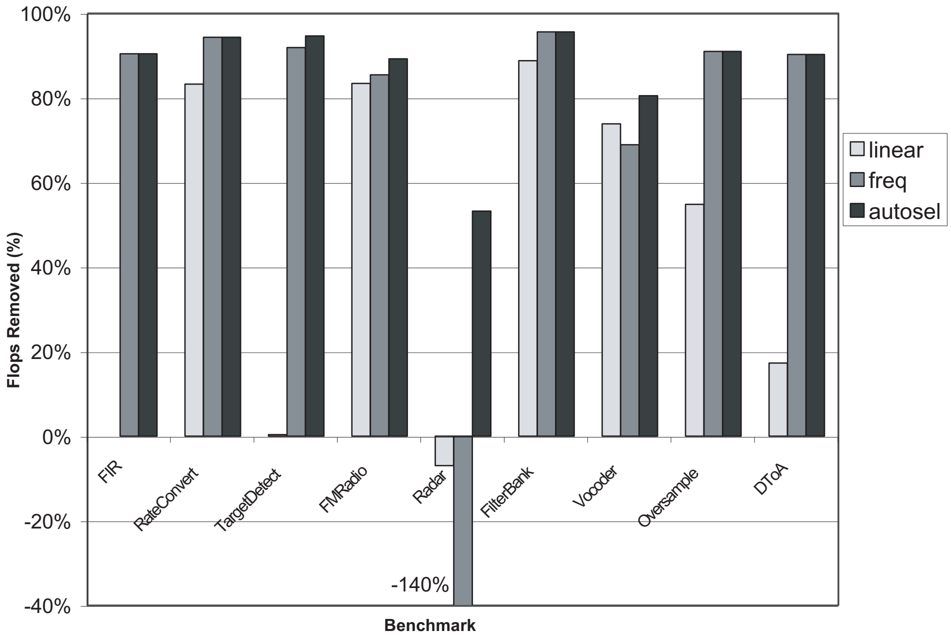


Figure 5-1: Elimination of floating point operations by maximal linear replacement, maximal frequency replacement, and automatic optimization selection.

---

system and other (FLOPS-preserving) optimizations in the compiler. As shown in Figure 5-1, each optimization leads to a significant reduction of FLOPS. Linear replacement eliminates FLOPS for all except for FIR, TargetDetect, and Radar, while frequency replacement eliminates FLOPS for all except Radar.

The automatic selection option eliminates more FLOPS than either of the other options for TargetDetect, FMRadio, Radar, and Vocoder. The effect is especially pronounced in Radar, where linear and frequency replacement increase the number of FLOPS, but automatic selection decreases FLOPS; the selection algorithm chooses to combine only some of the filters, transforming none to the frequency domain. Automatic selection always performs at least as well as the other two options, which implies that our cost functions have some level of accuracy.

Execution speedup for each benchmark is shown in Figure 5-3. With automatic selection, our benchmarks speed up by an average factor of 450% and by a factor of 800% in the best case. While the graph suggests that frequency replacement almost always performs better than linear replacement, this is not strictly the case; in FM-Radio, Radar, and Vocoder, the automatic selection algorithm obtains its speedup

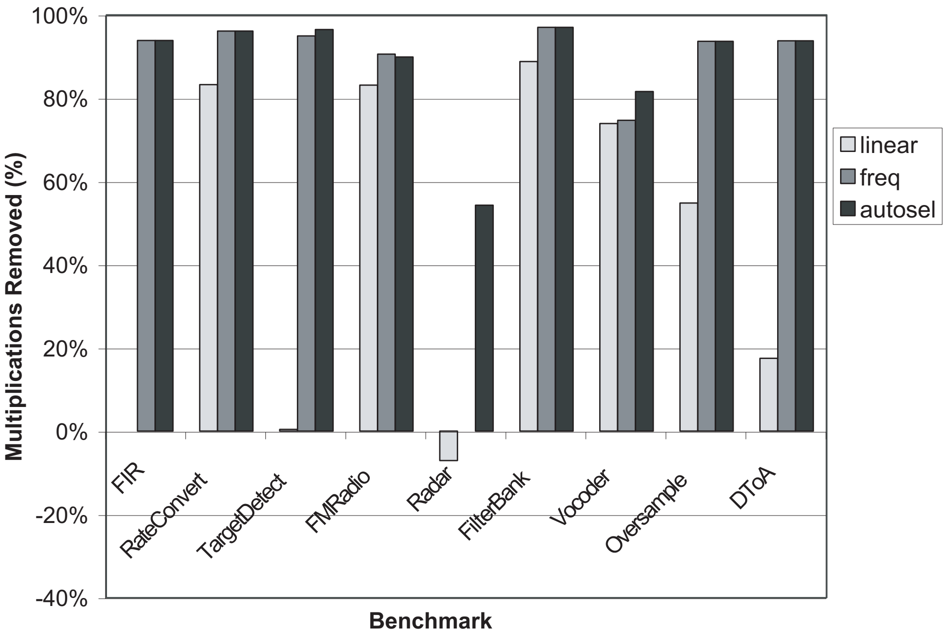


Figure 5-2: Elimination of floating point multiplications by maximal linear replacement, maximal frequency replacement, and automatic optimization selection.

---

by using linear replacement instead of frequency replacement for part of the stream graph. However, linear replacement does reduce performance for FIR, TargetDetect, and DToA despite reducing the number of FLOPS. We believe that this is due to inefficiencies in our implementation of the matrix multiplication routine, as well as auxiliary effects on the runtime overhead in the StreamIt library. We have experimented with using the machine-tuned ATLAS library for the matrix multiply [36] (see Section 5.4), but performance varies widely: linear replacement with ATLAS performs anywhere from -36% (on FMRadio) to 58% (on Oversampler) better than it does with our own matrix multiply routine, and average performance with ATLAS is 4.3% lower. Note that these numbers reflect our overhead in interfacing with ATLAS rather than the performance of ATLAS itself.

Perhaps the most interesting benchmark is Radar[2] (see Figure B-4 for stream graph). Maximal linear and frequency replacement lead to abysmal performance on

[2]This is the same Radar application presented in [15], with some filters adjusted to work at a coarser level of granularity. This eliminates persistent state in exchange for increased I/O rates. Also, frequency replacement caused an internal error in gcc for this program, so we used egcs 2.91 instead.

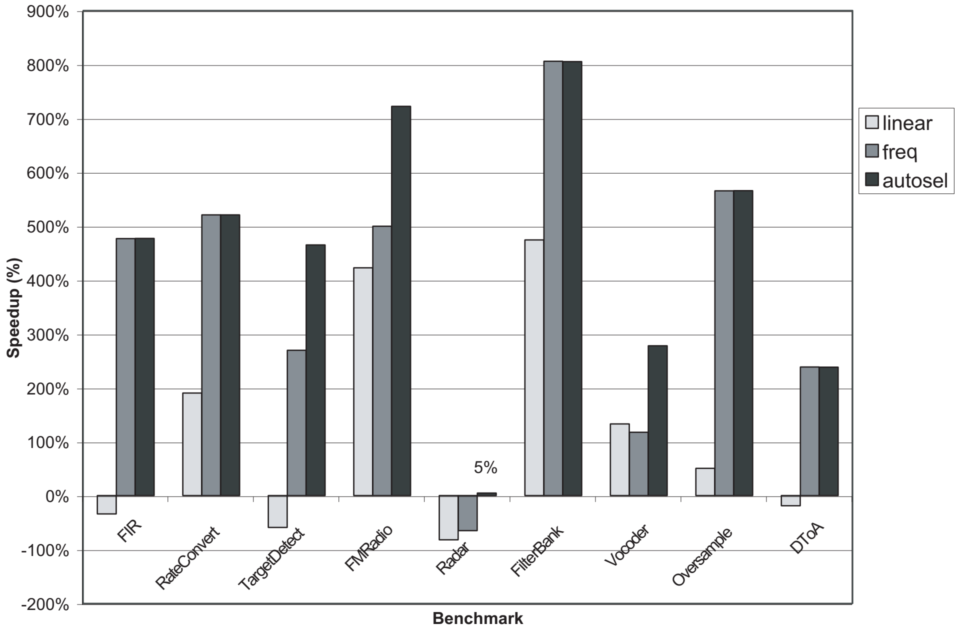


Figure 5-3: Execution speedup for maximal linear replacement, maximal frequency replacement, and automatic optimization selection.

Radar, due to a vector-vector multiply filter named "Beamform" at the top of a pipeline construct. The Beamform filter pushes 2 items, but pops and peeks 24; thus, when the replacement algorithms combine it with a downstream FIR filter, much of its work is duplicated. Moreover, the frequency replacement option suffers from the large pop rates in the application (as high as 128 for some filters), thereby increasing FLOPS and execution time by more than a factor of 30. The automatic selection algorithm is essential in this case: it averts the performance-degrading combinations and benefits from linear combinations elsewhere in the program, resulting in a significant reduction in FLOPS and a 5% performance gain.

5.3 Effect of Combination

While a good portion of our execution speedup and our FLOP reduction come from the efficiency of the FFTW library, the combination methods presented in this thesis inherently reduce FLOPS as well. We ran the same execution time and operation count measurements when combination was disabled. Figure 5-4 compares the per-

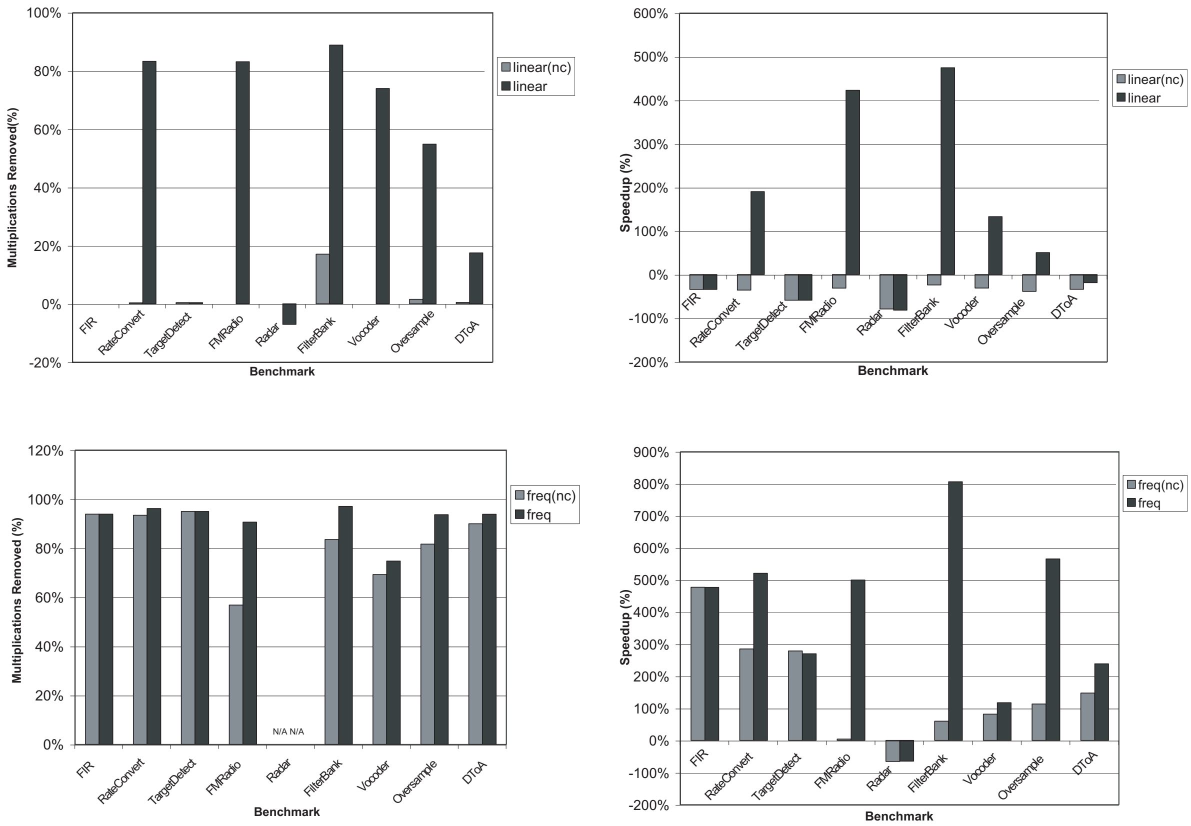


Figure 5-4: Elimination of multiplications (left) and speedup (right) with linear replacement (top) and frequency replacement (bottom) with and without combination. The *(nc)* label denotes that combination was disabled.

---

formance with no combination to that with combination.

The top two graphs in Figure 5-4 show that for linear replacement, combination leads to most of the multiplication reduction. FilterBank is the only application that has significant multiplication reduction without combination. We attribute the reduction to our implementation of matrix multiply which does not generate unused expressions. Execution time for linear replacement is better in all cases where combination is enabled than when it is disabled except for Radar. We interpret these result as validation of the fundamental strength of our combination algorithms in their own right as well as their usefulness as components of other optimizations (*i.e.* frequency replacement).

The bottom two graphs in Figure 5-4 show the effect of combination on speedup and multiplication reduction when using the frequency replacer. In all of the bench-

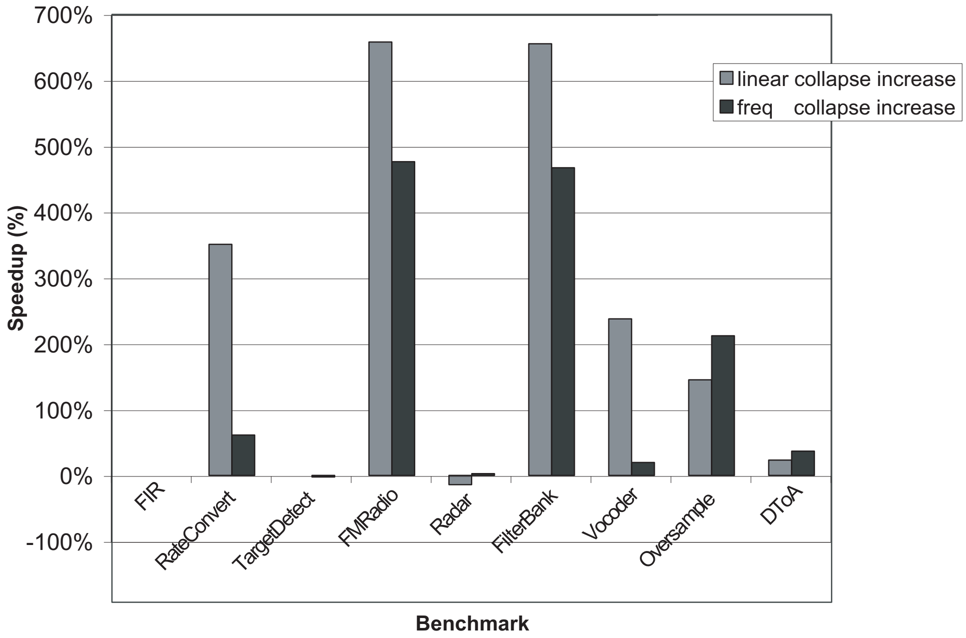


Figure 5-5: Differences in speedup due to the addition of combination with linear replacement and frequency replacement.

---

marks except Radar and FIR, frequency replacement generates impressive reductions even when combination disabled. Enabling combination further improves the reduction for all benchmarks. Execution time is decreased in all benchmarks except TargetDetect. Since the multiplication reduction is the same for TargetDetect both with and without combination, we attribute the 3% decrease in speedup to overhead in the runtime system and variations in our timing platform.

Figure 5-5 summarizes the increase in execution time due to combination for linear and frequency replacement. It is interesting to note that combination makes no difference for the FIR benchmark. This result makes intuitive sense because there are no adjacent linear filters that combination might improve. Due to the nature of the Radar benchmark (described above in Section 5.2), the numbers in Figure 5-5 for Radar reflect the fact that filters with a pop rate other than one were not transformed by frequency replacement[3]. The unit pop rate restriction disallows frequency transformations for the entire Radar benchmark and it explains the lack of multiplication reduction for frequency replacement.

[3]Without the unit pop rate restriction, code size explodes for the Radar benchmark.

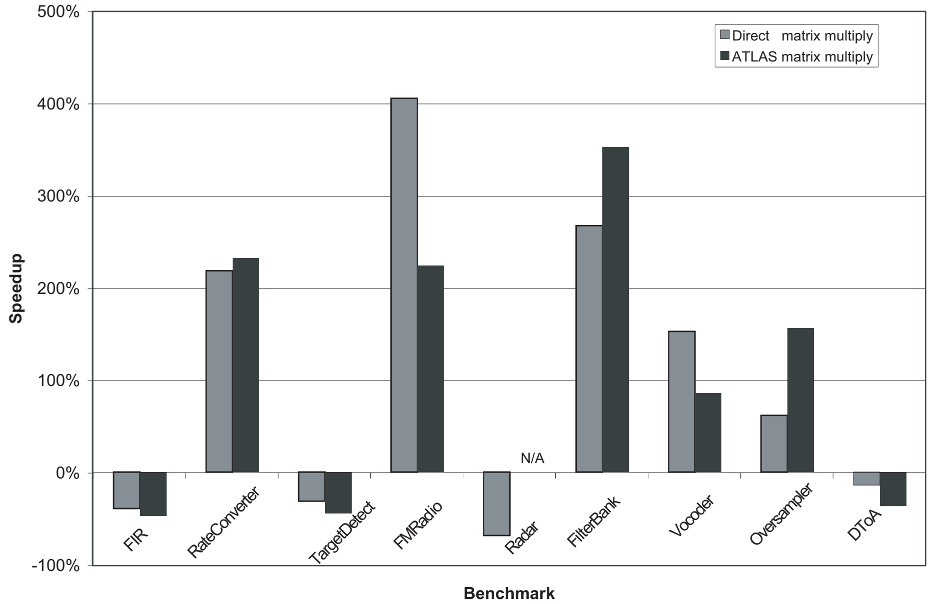


Figure 5-6: Speedups using ATLAS to implement linear replacement.

5.4 Effect of ATLAS

Using the ATLAS linear algebra system to implement matrix multiplies was inspired by the abysmal performance of our linear replacement optimization without combination. The FIR benchmark executes 34% slower with linear replacement even though the code is implementing exactly the same operation as the original program and it requires exactly the same number of FLOPS. This observation implies that the code generation strategy for pure linear forms is sub optimal.

For small filters ($e < 256$), code generated by linear replacement contains one push statement per column of the associated linear node whose argument is an expression of the form $a * \texttt{peek}(i)$ (*i.e.* $\texttt{push}(\texttt{a0} * \texttt{peek}(0) + \texttt{a1} * \texttt{peek}(1) + \ldots)$). For larger filters ($e \geq 256$), we store the matrix coefficients in an array A and compute the result in a loop nest, rather than inlining the computation into a single expression as shown in Figure 5-7. Note that our code is optimized for the case in which a column of A begins and/or ends with a series of zero entries. We avoid storing these zero entries by using a sparse A matrix in which the zeros have been removed from the ends of each column; we also adjust the loop bounds accordingly. This optimization improves performance for a number of our benchmarks. We also experimented with a

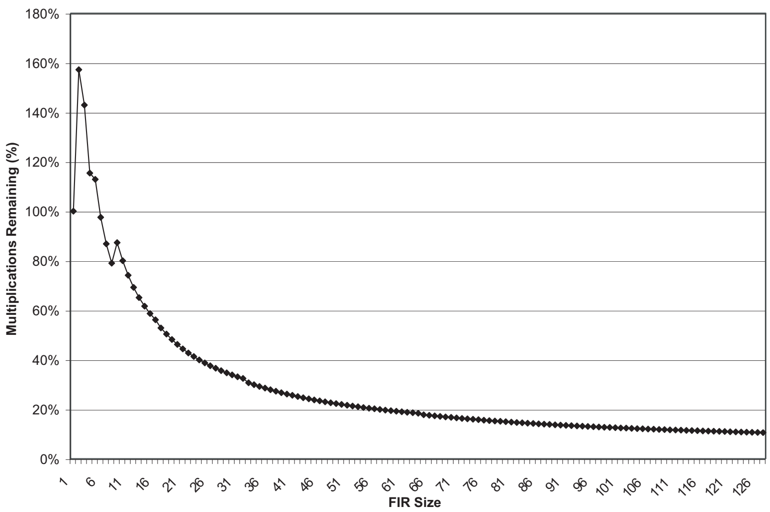
```
int sum, count;
for (int j=0; j<numPush; j++) {
  float sum = 0.0;
  // don't multiply the zero-entries
  // at beginning and end of column
  int count = firstNonZero[j];
  int length = lastNonZero[j] - firstNonZero[j];
  for (int i=0; i<length; i++) {
    sum += sparseA[i][j] * peek(count);
    count++;
  }
  sum += b[j];
  push (sum);
}
```

Figure 5-7: Code for diagonal matrix multiply for large filter sizes.

---

sparse matrix multiply in which all of the zero entries are removed and an additional array is used to index the non-zero elements. However, the overhead incurred by this level of indirection outweighs the benefits over the programs in our benchmark suite. In the uniprocessor backend, a peek expression is transformed into a complicated array access whose address calculation includes a logical bitmask operation. As the original program contains the same number of peek expressions, complicated processor instruction scheduling interactions are probably to blame for the poor performance.

The execution slowdown doesn't represent any inherent penalty due to our optimization technique; It merely represents a deficiency in the specific code we generate. We experimented using the ATLAS machine tuned linear algebra library to do the necessary computation instead of generating C code from within the compiler. Figure 5-6 shows execution speedups when using the linear replacement optimization with ATLAS compared to the original code generation strategy. In some applications ATLAS helps (RateConvert, FilterBank and Oversampler) but in the others it has no effect and even decreases performance (FIR, TargetDetect, FMRadio, Vocoder, and DToA). Again, this is not due to any deficiency in ATLAS but rather in the overhead incurred in our interface with it.



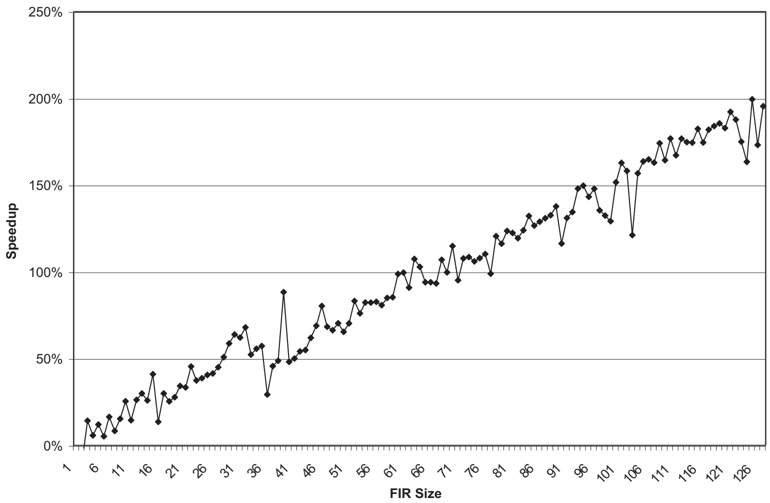


Figure 5-8: Elimination of floating point multiplications (top) and speedup (bottom) with frequency replacement as a function of problem size for the FIR benchmark.

5.5 FIR Scaling

Because the performance improvement due to our frequency replacement optimization is a function of the problem size, N, the extent to which problem size matters for our benchmarks is of interest. To determine the effect of problem size on performance we varied the size of the FIR benchmark from 1 filter coefficient to 128 filter coefficients and measured the multiplication reduction and the execution speedup for each size. Figure 5-8 shows the results obtained from this experiment. The original FIR benchmark contains 256 coefficients, but we do not show the plot to 256 co-

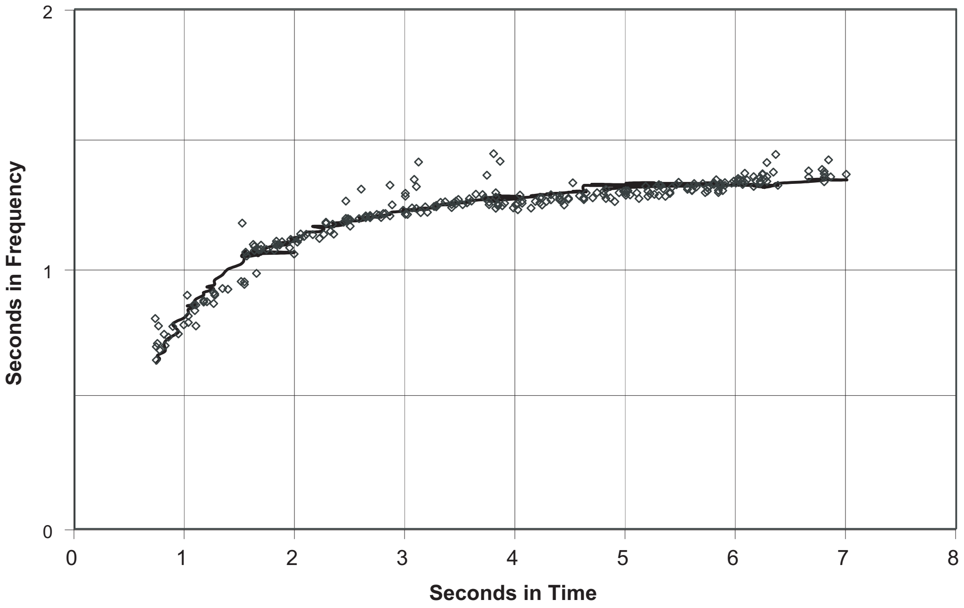


Figure 5-9: Scatter plot of original execution time versus post optimization execution time for FIR scaling experiments. Plotted as a solid line is the cost function used with the automatic selection algorithm.

efficients because all of the pertinent features appear closer to the origin. The top graph in Figure 5-8 shows the multiplications remaining after the optimization was applied and demonstrates good agreement with the theoretical prediction[4] of $\frac{\lg(N)}{N}$. The bottom graph of Figure 5-8 shows execution speedup. The graph is more noisy than the multiplication reduction graph and it looks approximately linear. However, in Figure 5-9 we plot the timing data as a function of original execution time and post optimization execution time. It is much easier to see the expected $\lg(N)$ shape[5]. Figure 5-9 also plots the cost function used by our automated selection algorithm which is shown in the following equation.

$$freqVal(firSize, timeVal) \quad = \quad 0.65 + \ln\left(1 + \frac{timeVal - timeVal_0}{1 + \frac{1}{50} firSize}\right)$$

[4]The theoretical $\frac{\lg(N)}{N}$ prediction is due to the $N \lg(N)$ frequency implementation compared to the N^2 original implementation.

[5]We expect $\frac{N}{\lg(N)}$ speedup if speedup is inversely proportional to multiplication reduction.

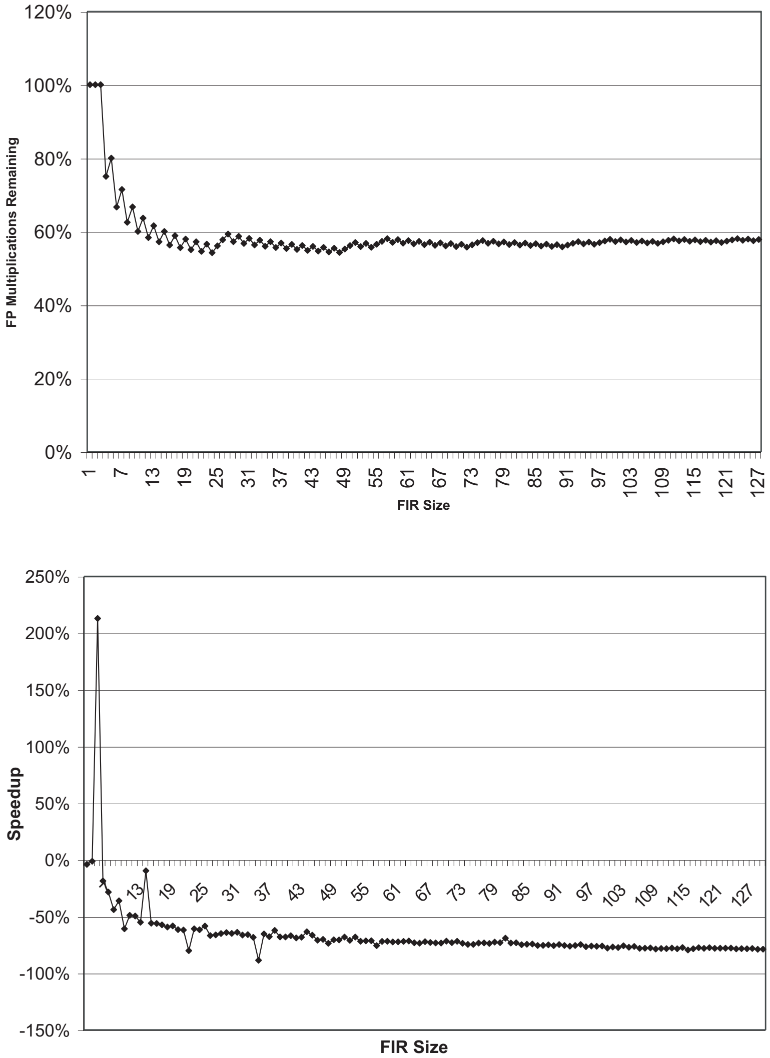


Figure 5-10: Multiplications remaining (top) and speedup (bottom) after redundancy replacement as a function of problem size for the FIR benchmark.

5.6 Redundancy Elimination

We now present simple measurements for the redundancy removal optimization described in Section 4.2. Because of the extensive state required for our implementation of redundancy removal, the code size for the benchmark programs grows out of control when we apply the optimization. To understand the effect of redundancy removal on performance, we measured multiplications and speedup as a function of FIR size. The FIR benchmark has approximately 50% redundancy due to the fact that A is a symmetric column vector yet the program was small enough that `gcc` compiled it without complaint.

The results of the experiments are shown in Figure 5-10. Redundancy removal does a good job of removing multiplications as shown in the top graph. However, the overhead necessary to affect this removal erases any potential benefit and slows the program down. Even worse, as the program size grows and the amount of removed redundancy increases, the program slows down even more. This decrease in performance as a function of FIR size is due to the fact that as the redundancy increases, more time must be spent to load and store cached values. Since the overhead to load and store is greater than the savings from removing the multiplication the overall program becomes slower.

One interesting feature of the top graph in Figure 5-10 is its peculiar "zig-zag" shape. The saw-toothed curve is not an accident. It results from the difference in redundancy of even and odd sized FIR filters. In an odd sized filter the values computed with the center coefficient can't be reused, while in an even sized filter there is no middle coefficient and all values can be reused. Therefore, if N is odd, after Transformation 7 is applied a filter of size $N+1$ will have fewer multiplications than a filter of size N even though the problem size is larger.

5.7 Radar Scaling

As mentioned previously, the Radar benchmark is challenging for our optimization techniques. Even more interesting is that our optimizations' performance depends on the problem configuration. The Radar benchmark as written has 12 "Channels" and 4 "Beams" representing 12 data sources being focused in 4 different directions. Referring to Figure B-4, "Channels" are in the top of the application and "Beams" are in the bottom. Figure 5-11 shows how maximal linear replacement changes the number of multiplications as a function of problem size. As the parameters grow, linear replacement does worse. The degradation due to increasing the number of Beams is much more pronounced than is increasing the number of Channels.

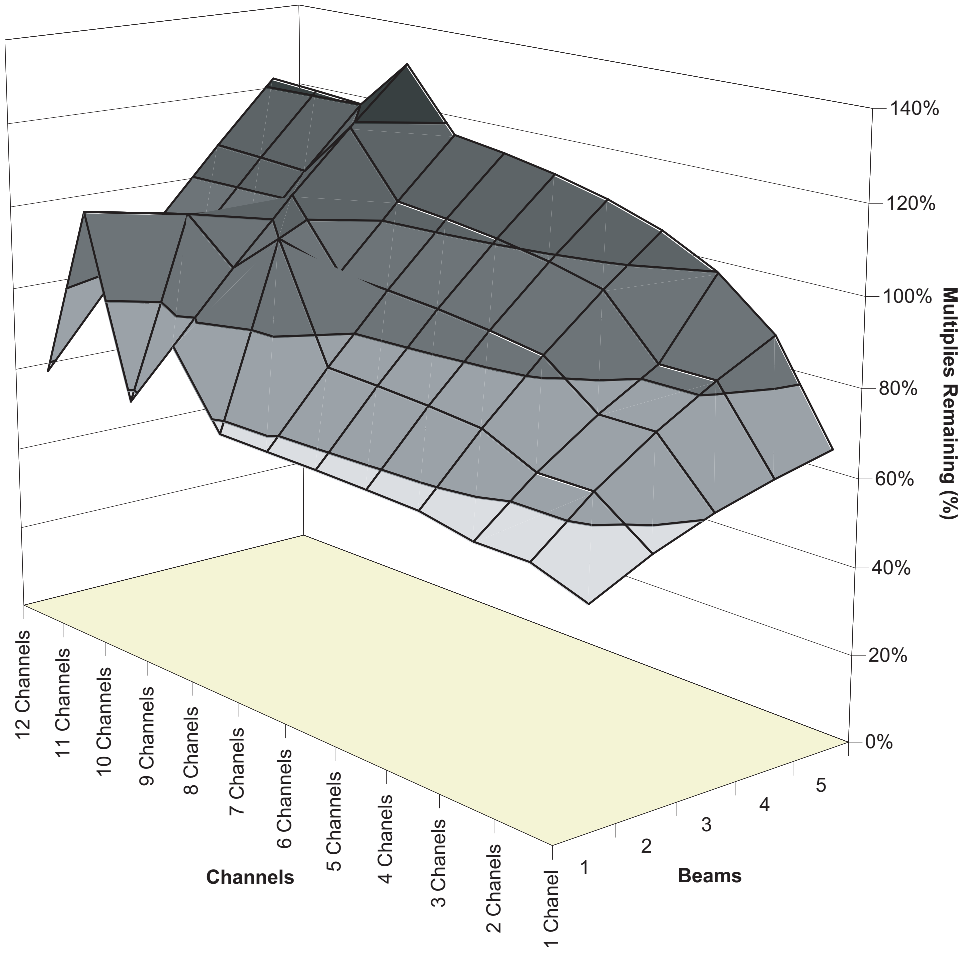


Figure 5-11: Multiplication reduction with maximal linear replacement as a function of problem size for the Radar benchmark.

5.8 FFT Savings: Theory vs. Practice

The strength of our results rely only partially on the efficiency of FFTW. Using the optimized frequency transformation (Transformation 6) over the naïve transformation (Transformation 5) also improves performance. To quantitatively evaluate the performance improvement due to various improvements in our frequency replacement technique we present the experiments summarized in Figure 5-12.

Figure 5-12 shows the effects of different frequency transformation strategies on post transformation multiplications. To highlight the differences between methods, we report the *multiplication reduction factor* which is defined as the number of multiplications originally required per output over the number of multiplications required per output after applying the specified frequency transformation. The x axis is the size of the FIR and the y axis is the size of the FFT. The FFT size was manually specified instead of automatically chosen.

- Figure 5-12 a) shows the theoretical reduction.
- Figure 5-12 b) shows the multiplication reduction using the naïve transformation (Transformation 5) and a simple FFT implementation.
- Figure 5-12 c) shows the improvement from Figure 5-12 b) due to the optimized frequency transformation (Transformation 6).
- Figure 5-12 d) shows the improvement from Figure 5-12 c) due to using FFTW to perform the FFT.

From Figure 5-12 we can see that the optimized transformation improves performance by a factor of 1.5 over the naïve transformation and using FFTW improves it by a factor of 6. While FFTW provides a large performance boost, our optimizations are also worthwhile in their own right.

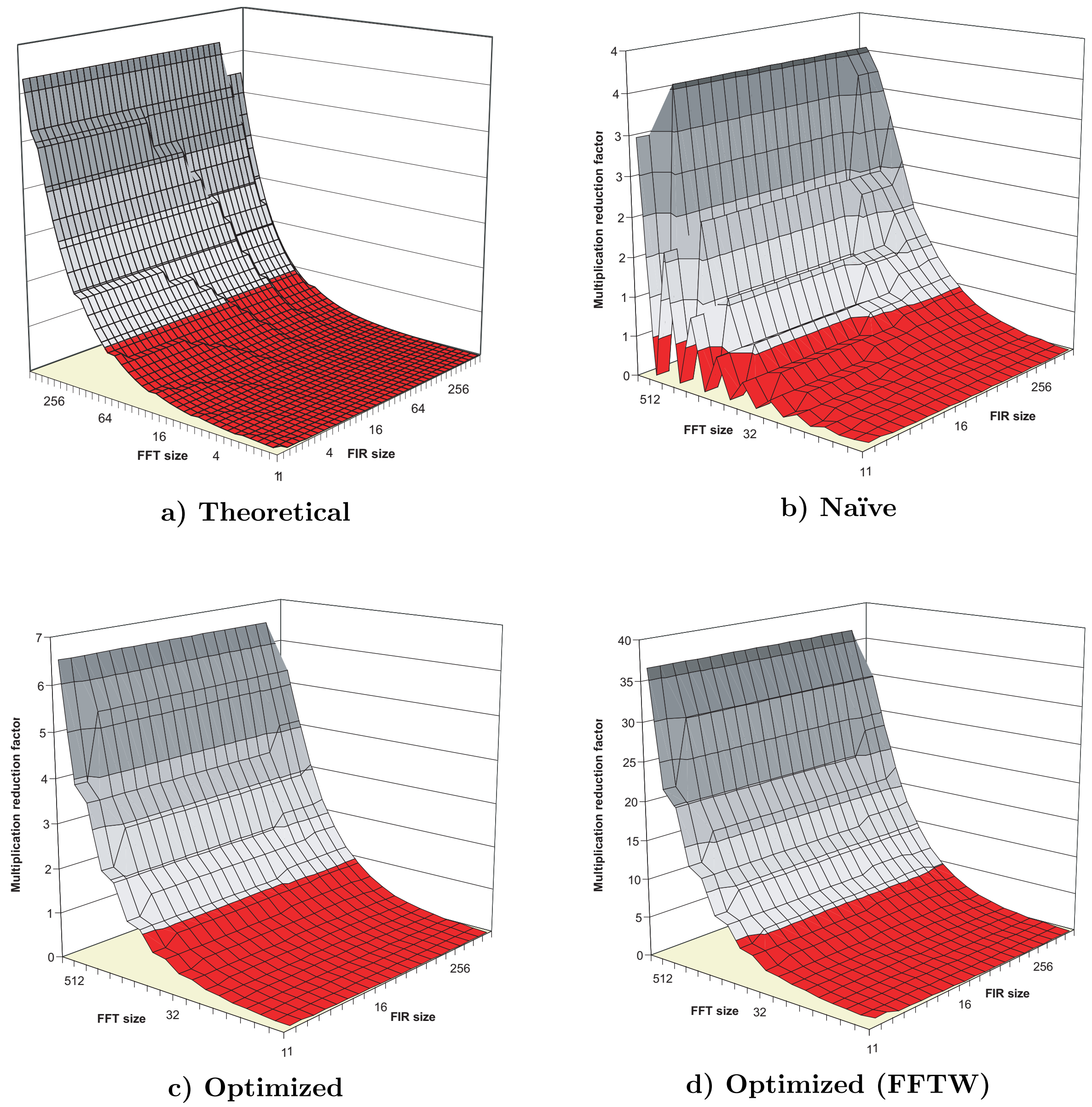


Figure 5-12: Elimination of floating point multiplications with frequency replacement as a function of problem size and FFT length for the FIR program using different transformation strategies.

Chapter 6

Related Work

Several groups are researching strategies for efficient code generation for DSP applications. SPIRAL is a system that generates libraries for signal processing algorithms [5, 18, 19]. Using a feedback-directed search process, DSP transforms are optimized for the underlying architecture. The input language to SPIRAL is SPL [37, 38], which provides a parameterizable way of expressing matrix computations. Given a matrix representation in SPL, SPIRAL generates formulas that correspond to different factorizations of the matrix. It searches for the most efficient formula using several techniques, including dynamic programming and stochastic evolutionary search.

We consider our work to be complementary to SPIRAL. While SPIRAL starts with a matrix representation in SPL, we start with general StreamIt code and use linear dataflow analysis to extract a matrix representation where possible. Our linear combination rules are distinct from the factorizations of SPIRAL, as StreamIt nodes can peek at items that they do not consume. In the future, SPIRAL could be integrated with StreamIt to optimize a matrix factorization for a given architecture.

The ATLAS project [36] also aims to produce fast libraries for linear algebra manipulations, focusing on adaptive library generation for varying architectures. FFTW [6] is a runtime library of highly optimized FFT's that dynamically adapt to architectural variations. StreamIt is again complementary to these packages: it allows programmers to interface with them using general user-level code.

ADE (A Design Environment) is a system for specifying, analyzing, and manipulating DSP algorithms [4]. ADE includes a rule-based system that can search for improved arrangements of stream algorithms using extensible transformation rules. However, the system uses predefined signal processing blocks that are specified in mathematical terms, rather than the user-specified imperative code that appears in a StreamIt filter. Moreover, ADE is intended for algorithm exploration, while StreamIt includes support for code generation and whole-program development. In addition to ADE, other work on DSP algorithm development is surveyed in [26].

Karr [21] and Cousot and Halbwachs [3] describe general methods for detecting linear relationships among program variables. Karr maintains an affine representation (similar to ours) for each program variable, while Cousot and Halbwachs use a polyhedral model in which each dimension corresponds to a program variable. For general programs, the analyses described by these authors is more general than ours. In fact, the novelty of our linear dataflow analysis is in its specialization for the streaming domain. Rather than tracking general relationships, we only track relationships to items on the input tape. This restriction — in combination with the atomic, fine-grained nature of filter work functions — makes it feasible to symbolically execute all loops, thereby obtaining more precise linearity information.

A number of other programming languages are oriented around a notion of a stream; see [31] for a survey. Synchronous languages such as LUSTRE [16], Esterel [2], and Signal [12] target the embedded domain, while languages such as Occam [17], SISAL [11] and StreamC [28] target parallel and vector targets. However, none of the compilers for these languages have coarse-grained, DSP-specific analyses such as linear filter detection. Also note that the “linear data flow analysis” of Ryan [29] is completely unrelated to our work; it aims to do program analysis in linear time.

Chapter 7

Conclusion

This thesis presents a set of automated analyses for detecting, analyzing, and optimizing linear filters in streaming applications. Though the mathematical optimization of linear filters has been a longtime focus of the DSP community, our techniques are novel in the automated application of these techniques to programs that are written in a flexible and high-level programming language. We demonstrate that using our linear dataflow analysis, linear combination, frequency translation and automated optimization selection we can improve execution speed by an average factor of 450% and 800% in the best case. This improvement is primarily due to the removal, on average, of 86% of the floating point instructions.

The ominous rift between the design and implementation of signal processing applications is growing by the day. Algorithms are designed at a conceptual level utilizing modular processing blocks that naturally express the computation. However, in order to obtain good performance, each hand-tuned implementation is forced to disregard the abstraction layers and painstakingly consider specialized whole-program optimizations. The StreamIt project aims to reduce this process to a single stage in which the designers and implementors share a set of high-level abstractions that can be efficiently handled by the compiler.

The linear analysis described in this thesis represents a first step toward this goal. By automatically performing linear combination, frequency translation, and optimization selection, it allows programmers to write linear stream operations in a natural and modular fashion without any performance penalty.

7.1 Future Work

Linear analysis can easily be extended to incorporate a notion of linear state. A stateful linear node is characterized by

$$\lambda = ((A_x, A_s), (C_x, C_s), (\vec{b}_x, \vec{b}_s))$$

Each filter in the stream graph contains a state vector $\vec{s}$ such that $\vec{y}_i$ (the output at time i) and $\vec{s}_{i+1}$ (the state at time $i+1$) are given by

$$\vec{y}_i = \vec{x}A_x + \vec{s}_i A_s + \vec{b}_x$$

$$\vec{s}_{i+1} = \vec{x}C_x + \vec{s}_i C_s + \vec{b}_s$$

The addition of stateful nodes allows us to describe a larger class of programs using our linear analysis framework. Using linear state, our structure combination rules can be extended to include feedbackloops. Examples of programs that exhibit stateful linear nodes are control systems and infinite impulse response (IIR) filters.

Another promising avenue of research is to exploit matrix factorization in order to automatically derive fast implementations of large computations such as DSP transforms. Matrix representations will also be useful for generating load-balanced parallel code in the StreamIt Raw backend [15]. Finally, to increase the class of programs that would fit into our linear framework, the entries of the A and $\vec{b}$ of linear nodes could contain symbolic constants that are resolved at runtime.

Appendix A

Benchmark Source Code

As our benchmarks are not provided by a standard source, this appendix includes the StreamIt source code. Two of the benchmarks, FMRadio and Radar, are written in an older syntax. Thankfully, the StreamIt compiler is capable of compiling several types of syntax. Because of the verbosity of the old syntax, we are unable to provide the source code for the Radar benchmark here.

```
/**
 * Simple StreamIt filter that simply absorbs floating
 * point numbers without printing them.
 **/
float->void filter FloatSink {
  work pop 1 {
    pop();
  }
}
```

Figure A-1: Source code for FloatSink.

```
/**
 * 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();
  }
}
```

Figure A-2: Source code for LowPassFilter.

```
/**
 * 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;
  }
}
```

Figure A-3: Source code for the FIR benchmark.

```
/**
 * 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();
    }
  }
}
```

Figure A-4: Source code for Compressor.

```
/**
 * This filter expands the input by a factor L. Eg in takes in one
 * sample and outputs L samples. The first sample is the input
 * and the rest of the samples are zeros.
 **/
float->float filter Expander(int L) {
  work peek 1 pop 1 push L {
    push(pop());
    for (int i=0; i<(L-1); i++) {
      push(0);
    }
  }
}
```

Figure A-5: Source code for Expander.

```
/**
 * This program uses multirate filtering to change the sampling rate of
 * the input by a non integer amount. We assume that the original sampling
 * rate was every T seconds (eg at a frequency of 2*pi/T radians)
 * and the output of the system will be the same as if the original
 * signal had been sampled at every 3T/2 seconds instead. As this
 * is a non-integer ratio conversion, we need to upsample, low-pass filter
 * (eg interpolate) and then downsample.
 **/

void->void pipeline SamplingRateConverter {
  add SampledSource(pi/10);
  add pipeline {
    add Expander(2);
    /* gain 3, cutoff freq of pi/3, 300 sample FIR */
    add LowPassFilter(3, pi/3, 300);
    add Compressor(3);
  };
  add FloatPrinter();
}



/**
 * Simple filter that provides a source of sampled data.
 * For now we are going to produce just a cosine at a frequency
 * specified.
 */
void->float filter SampledSource(float w) {
  int n = 0;
  work push 1 {
    push(cos(w*n));
    n++;
  }
}
```

Figure A-6: Source code for the RateConvert benchmark.

```
/**
 * Target detection application. This app has several matched filters that
 * run in parallel (in a splitjoin). The results
 * are then combined back into a single stream and a threshold detector
 * prints off the filters which detect targets.
 **/
void->void pipeline TargetDetect {
  /* N is the length of the filters */
  int N = 300;
  add TargetSource(N);
  add TargetDetectSplitJoin(N,8);
  /* there are four matched filters */
  add FloatPrinter();
}
float->float splitjoin TargetDetectSplitJoin(int N, float thresh) {
  split duplicate;
  add pipeline {
    add MatchedFilterOne(N);
    add ThresholdDetector(1,thresh);
  };
  add pipeline {
    add MatchedFilterTwo(N);
    add ThresholdDetector(2, thresh);
  };
  add pipeline {
    add MatchedFilterThree(N);
    add ThresholdDetector(3,thresh);
  };
  add pipeline {
    add MatchedFilterFour(N);
    add ThresholdDetector(4,thresh);
  };
  join roundrobin;
}
float->float filter ThresholdDetector(int number, float threshold) {
  work pop 1 push 1 {
    /* pop the input value off of the input stream. */
    float t = pop();
    /* do the threshold test -- if it is greater, then push out number, otherwise push 0 */
    if (t > threshold) {
      push(number);
    } else {
      push(0);
    }
  }
}
void->float filter TargetSource(int N) {
  int currentPosition = 0;
  work push 1 {
    if (currentPosition < N) {
      push(0);
    } else if (currentPosition < (2*N)) {
      float trianglePosition = currentPosition - N;
      // figure out which side of the triangle we are on.
      if (trianglePosition < (N/2)) {
        push((trianglePosition * 2) / N);
      } else {
        push(2-((trianglePosition * 2)/ N));
      }
    } else {
      push(0);
    }
    // update the current position.
    currentPosition = (currentPosition + 1)%(10*N);
  }
}
```

Figure A-7: Source code for the TargetDetect benchmark.

```
float->float filter MatchedFilterOne(int N) {
  float[N] h;
  init {
    for (int i=0; i<N; i++) {
      float trianglePosition = i;
      /* make the triangle */
      h[i] = ((i < (N/2)) ? ((trianglePosition * 2) / N) :
              (2 - ((trianglePosition * 2) / N))) - 0.5;
  }}
  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();
}}
float->float filter MatchedFilterTwo(int N) {
  float[N] h;
  init {
    for (int i=0; i<N; i++) {
      float pos = i;
      /* make the sine (mean is zero) */
      h[i] = (1/(2*pi)) * sin(pi * pos / N) - 1;
  }}
  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();
}}
float->float filter MatchedFilterThree(int N) {
  float[N] h;
  init {
    for (int i=0; i<N; i++) {
      float pos = i;
      /* make the sine (mean is already 0) */
      h[i] = (1/(2*pi)) * sin(2 * pi * pos / N);
  }}
  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();
}}
float->float filter MatchedFilterFour(int N) {
  float[N] h;
  init {
    for (int i=0; i<N; i++) {
      float pos = i;
      /* make the ramp -- have to worry about time reversal */
      h[(N-i)-1] = (0.5) * ((pos / N) - 0.5);
  }}
  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();
}}
```

Figure A-8: Source code for matched filters in the TargetDetect benchmark.

```
import streamit.*; import streamit.io.*;
import java.lang.Math;
class FloatNAdder extends Filter {
    int N;
    public FloatNAdder(int count) {super(count);}
    public void init (final int count) {
        N = count;
        input = new Channel (Float.TYPE, count,
                             count);
        output = new Channel (Float.TYPE, 1);
    }
    public void work() {
        float sum = 0.0f;
        int i;
        for (i = 0; i < N; i++)
            sum += input.popFloat();
        output.pushFloat(sum);
    }
}
class FloatDiff extends Filter {
    public void init() {
        input = new Channel(Float.TYPE, 2, 2);
        output = new Channel(Float.TYPE, 1);
    }
    public void work() {
        output.pushFloat(input.peekFloat(0) -
                         input.peekFloat(1));
        input.popFloat(); input.popFloat();
    }
}

class FloatDup extends Filter {
    public void init() {
        input = new Channel(Float.TYPE, 1, 1);
        output = new Channel(Float.TYPE, 2);
    }
    public void work() {
        float val = input.popFloat();
        output.pushFloat(val);
        output.pushFloat(val);
    }
}
class EqualizerInnerPipeline extends Pipeline  {
    public EqualizerInnerPipeline(float rate,
                                  float freq){
        super(rate, freq);
    }
    public void init(final float rate,
                     final float freq) {
        add(new LowPassFilter(rate, freq, 64, 0));
        add(new FloatDup());
    }
}
class EqualizerInnerSplitJoin extends SplitJoin {
    public EqualizerInnerSplitJoin(
           float rate, float low,
           float high, int bands) {
        super(rate, low, high, bands);
    }
    public void init(final float rate,
                     final float low,
                     final float high,
                     final int bands) {
                     int i;
        setSplitter(DUPLICATE());
        for (i = 0; i < bands - 1; i++)
            add(new EqualizerInnerPipeline
                (rate,
                 (float)java.lang.Math.exp
                 ((i+1) *
                  (java.lang.Math.log(high) -
                   java.lang.Math.log(low)) /
                  bands + java.lang.Math.log(low))));
        setJoiner(ROUND_ROBIN(2));
    }
}
class EqualizerSplitJoin extends SplitJoin {
    public EqualizerSplitJoin(
           float rate, float low,
           float high, int bands) {
        super(rate, low, high, bands);
    }
    public void init(final float rate, final float low,
                     final float high, final int bands) {
        setSplitter(DUPLICATE());
        add(new LowPassFilter(rate, high, 64, 0));
        add(new EqualizerInnerSplitJoin(rate, low,
                                        high, bands));
        add(new LowPassFilter(rate, low, 64, 0));
        setJoiner(WEIGHTED_ROUND_ROBIN(1, (bands-1)*2, 1));
    }
}
class Equalizer extends Pipeline {
    public Equalizer(float rate) {
        super(rate);
    }
    public void init(final float rate) {
        final int bands = 10;
        final float low = 55;
        final float high = 1760;
        add(new EqualizerSplitJoin(rate, low,
                                   high, bands));
        add(new FloatDiff());
        add(new FloatNAdder(bands));
    }
}
class FloatOneSource extends Filter {
    int x;
    public void init () {
        output = new Channel(Float.TYPE, 1);
        x = 0;
    }
    public void work() {
        output.pushFloat(x++);
    }
}
class FloatPrinter extends Filter {
    public void init () {
        input = new Channel(Float.TYPE, 1);
    }
    public void work () {
        System.out.println(input.popFloat ());
    }
}
```

Figure A-9: Source code for the FMRadio benchmark (part one).

```
class FMRadio extends Pipeline {
    public FMRadio() {super();}
    public void init()
    {
        final float samplingRate = 200000;
        final float cutoffFrequency = 108000000;
        final int numberOfTaps = 64;
        final float maxAmplitude = 27000;
        final float bandwidth = 10000;
        add(new LowPassFilter(samplingRate,
            cutoffFrequency, numberOfTaps, 4));
        add(new FMDemodulator(samplingRate,
            maxAmplitude, bandwidth));
        add(new Equalizer(samplingRate));
    }
}
class FMDemodulator extends Filter {
    float mGain;
    float sampleRate;
    float maxAmplitude;
    float modulationBandwidth;
    public FMDemodulator (float sampRate,
        float max, float bandwidth) {
        super (sampRate, max, bandwidth);
    }
    public void init(float sampRate,
             float max, float bandwidth) {
        input = new Channel (Float.TYPE, 1, 2);
        output = new Channel (Float.TYPE, 1);
        sampleRate = sampRate;
        maxAmplitude = max;
        modulationBandwidth = bandwidth;
        mGain = maxAmplitude *
            (sampleRate /
             (modulationBandwidth*(float)Math.PI));
    }
    public void work() {
        float temp = 0;
        temp = (float)((input.peekFloat(0)) *
                       (input.peekFloat(1)));
        temp = (float)(mGain * Math.atan(temp));
        input.popFloat();
        output.pushFloat(temp);
    }
}
public class LinkedFMTest extends StreamIt {
    static public void main(String[] t) {
        new LinkedFMTest().run(t);
    }
    public void init() {
        add(new FloatOneSource());
        add(new FMRadio());
        add(new FloatPrinter());
    }
}
class LowPassFilter extends Filter {
    int numberOfTaps;
    float COEFF[];
    float cutoffFreq, samplingRate, tapTotal;
    int mDecimation;
    public LowPassFilter(float sampleRate,
      float cutFreq, int numTaps, int decimation){
      super(sampleRate, cutFreq, numTaps,
            decimation);
    }
    public void init(final float sampleRate,
      final float cutFreq, final int numTaps,
      final int decimation) {
        float pi, m, w;
        int i;
        samplingRate = sampleRate;
        cutoffFreq = cutFreq;
        numberOfTaps = numTaps;
        pi = (float)java.lang.Math.PI;
        m = numberOfTaps -1;
        // m is the order of filter
        mDecimation = decimation;
        input = new Channel (Float.TYPE,
                 1+decimation, numTaps);
        output = new Channel (Float.TYPE, 1);
        //all frequencies are in hz
        COEFF = new float[numTaps];
        if(cutoffFreq == 0.0) {
            //Using a Hamming window
            tapTotal = 0;
            for(i=0;i<numberOfTaps;i++) {
                COEFF[i] = (float)(0.54 -
                 0.46*java.lang.Math.cos((2*pi)*(i/m)));
                tapTotal = tapTotal + COEFF[i];
            }
            //normalize all the taps to a sum of 1
            for(i=0;i<numberOfTaps;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
            w = (2*pi) * cutoffFreq/samplingRate;
            for(i=0;i<numberOfTaps;i++) {
                //check for div by zero
                if(i-m/2 == 0)
                    COEFF[i] = w/pi;
                else
                    COEFF[i] =
           (float)(java.lang.Math.sin(w*(i-m/2)) / pi
           / (i-m/2)
           * (0.54 - 0.46
           * java.lang.Math.cos((2*pi)*(i/m))));
            }
        }
     }
    public void work() {
        float sum = 0;
        int i;
        for (i=0; i<numberOfTaps; i++)
            sum += input.peekFloat(i)*COEFF[i];
        input.popFloat();
        for(i=0;i<mDecimation;i++)
            input.popFloat();
        output.pushFloat(sum);
    }
}
```

Figure A-10: Source code for the FMRadio benchmark (part two).

```
/* 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 same 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);
}
```

Figure A-11: Source code for BandPassFilter.

```
/* This is a bandstop filter with the rather simple implementation of
 * a low pass filter cascaded with a high pass filter. The relevant parameters
 * are: end of passband=wp and end of stopband=ws, such that 0<=wp<=ws<=pi
 * gain of passband and size of window same for both filters. Note that the high
 * pass and low pass filters currently use a rectangular window.
 *
 * We take the signal, run both the low and high pass filter separately and then
 * add the results back together.
 **/
float->float pipeline BandStopFilter(float gain, float wp, float ws, int numSamples) {
  add splitjoin {
    split duplicate;
    add LowPassFilter(gain, wp, numSamples);
    add HighPassFilter(gain, ws, numSamples);
    join roundrobin;
  };
  /* sum the two outputs together. */
  add Adder(2);
}
```

Figure A-12: Source code for BandStopFilter.

```
/**
 * This is a generic filter bank that decomposes an incoming stream into
 * M frequency bands. It then performs some processing on them
 * and then reconstructs them.
 **/
void->void pipeline FilterBank {
  add DataSource();
  add FilterBankPipeline(3);
  add FloatPrinter();
}
float->float pipeline FilterBankPipeline(int M) {
  add FilterBankSplitJoin(M);
  add Adder(M);
}
/**
 * Filterbank splitjoin (everything before the final adder. )
 **/
float->float splitjoin FilterBankSplitJoin(int M) {
  split duplicate;
  for (int i=0; i<M; i++) {
    add ProcessingPipeline(M,i);
  }
  join roundrobin;
}
/**
 * The main processing pipeline: analysis, downsample, process, upsample, synthesis.
 * I use simple bandpass filters for the Hi(z) and Fi(z).
 **/
float->float pipeline ProcessingPipeline(int M, int i) {
  /* analysis pipeline */
  add pipeline {
    /* take the subband from i*pi/M to (i+1)*pi/M */
    add BandPassFilter(1, (i*pi/M), ((i+1)*pi/M), 100);
    /* decimate by M */
    add Compressor(M);
  };
  /* process the subband */
  add ProcessFilter(i);
  /* synthesis pipeline */
  add pipeline {
    /* upsample by M */
    add Expander(M);
    /* synthesize (eg interpolate) */
    add BandStopFilter(M, (i*pi/M), ((i+1)*pi/M), 100);
  };
}
/* This is (obviously) the data source. */
void->float filter DataSource() {
  int n = 0;
  float w1 = pi/10;
  float w2 = pi/20;
  float w3 = pi/30;
  work push 1 {
    push(cos(w1*n) + cos(w2*n) + cos(w3*n));
    n++;
  }
}
/* this is the filter that we are processing the sub bands with. */
float->float filter ProcessFilter(int order) {
  work pop 1 push 1 {
    push(pop());
  }
}
```

Figure A-13: Source code for the FilterBank benchmark.

```
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;
  int DECIMATION   = 50;
  int NUM_FILTERS  = 4;

  split duplicate;
  add PitchDetector(PITCH_WINDOW, DECIMATION);
  add VocoderFilterBank(NUM_FILTERS, DECIMATION);
  join roundrobin(1,4);
}
/** 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);
}
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);
    }
  }
}
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();
    }
  }
}
```

Figure A-14: Source code for the Vocoder benchmark.

```
/**
 * This app is intended to implement a one bit d to a oversampler.
 **/
void->void pipeline Oversampler {
  // the source of data (eventually, this should be an audio stream)
  add DataSource();
  // go through the process of oversampling
  add OverSampler();
  add DataSink();
}
/**
 * This is a 16x oversampler -- eg it upsamples
 * its input by a factor of 16. The 16 factor is
 * achieved in four steps. In each step we upsample
 * and then low pass filter with cutoff of pi/2.
 **/
float->float pipeline OverSampler {
  for (int i=0; i<4; i++) {
    add Expander(2);
    add LowPassFilter(2,pi/2,64);
  }
}
/**
 * Simple data source that outputs a combination of
 * three sine waves with a period of 100 samples.
 **/
void->float filter DataSource {
  int SIZE = 100;
  float[SIZE] data;
  int index;
  init {
    // initialize the data array.
    for (int i=0; i<SIZE; i++) {
      float t = i;
      data[i] = (sin((2*pi)*(t/SIZE)) +
                 sin((2*pi)*(1.7*t/SIZE) + (pi/3)) +
                 sin((2*pi)*(2.1*t/SIZE) + (pi/5)));
    }
    index = 0;
  }
  work push 1 {
    push(data[index]);
    index = (index + 1)%SIZE;
  }
}
/** Simple sink that prints out the data that it receives. **/
float->void filter DataSink {
  work pop 1 {
    println(pop());
  }
}
/** Simple data selector filter (for filtering out multiplexed output for debugging) **/
float->float filter DataSelector {
  work pop 2 push 1 {
    //push(pop());
    pop();
    push(pop());
  }
}
```

Figure A-15: Source code for the Oversample benchmark.

```
void->void pipeline OneBitDToA {
  // the source of data
  add DataSource();
  add OverSampler();
  add NoiseShaper();
  add LowPassFilter(1,pi/100,256);
  add DataSink();
}
/**
 * This is a 16x oversampler -- eg it upsamples
 * its input by a factor of 16. The 16 factor is
 * achieved in four steps. In each step we upsample
 * and then low pass filter with cutoff of pi/2.
 **/
float->float pipeline OverSampler {
  for (int i=0; i<4; i++) {
    add Expander(2);
    add LowPassFilter(2,pi/2,64);
  }
}
/**
 * This is a first order noise shaper. It is
 * built with a feedback loop. See Oppenheim,
 * Shafer and Buck, 2nd ed.
 * page 211 for a more detailed explaination.
 **/
float->float feedbackloop NoiseShaper {
  join roundrobin(1,1);
  body pipeline {
    add AdderFilter();
    add QuantizerAndError();
  }
  loop Delay();
  split roundrobin(1,1);
  enqueue 0;
}
/**
 * Simple data source that outputs a combination of
 * three sine waves with a period of 100 samples.
 **/
void->float filter DataSource {
  int SIZE = 100;
  float[SIZE] data;
  int index;
  init {
    // initialize the data array.
    for (int i=0; i<SIZE; i++) {
      float t = i;
      data[i] = (sin((2*pi)*(t/SIZE)) +
        sin((2*pi)*(1.7*t/SIZE) + (pi/3)) +
        sin((2*pi)*(2.1*t/SIZE) + (pi/5)));
    }
    index = 0;
  }
  work push 1 {
    push(data[index]);
    index = (index + 1)%SIZE;
  }
}
float->void filter DataSink {
  work pop 1 {
    println(pop());
  }
}
float->float filter DataSelector {
  work pop 2 push 1 {
    //push(pop());
    pop();
    push(pop());
  }
}
/**
 * Quantizer filter that converts the input into
 * either 1 or -1 as a first output and then
 * outputs the error value (input-output) as second
 * output.
 **/
float->float filter QuantizerAndError {
  work pop 1 push 2 {
    float inputValue = pop();
    float outputValue;

    // calculate[5~ output based on sign of input
    if (inputValue < 0) {
      outputValue = -1;
    } else {
      outputValue = 1;
    }

    // now calculate the error value
    float errorValue = (outputValue-inputValue);

    // and push both of the values
    push(outputValue);
    push(errorValue);
  }
}
/** Simple adder (add two sequential
    values on tape). **/
float->float filter AdderFilter {
  work pop 2 push 1 {
    push (pop() + pop());
  }
}
/** Unit delay -- delays values by one. **/
float->float filter Delay {
  float state;
  init {state = 0;}
  work pop 1 push 1 {
    push(state);
    state = pop();
  }
}
```

Figure A-16: Source code for the DToA benchmark.

Appendix B

Benchmark Stream Graphs

This section contains the stream graphs of the benchmark programs used to measure the performance of our optimizations. Stream graphs are figures automatically generated by the StreamIt compiler which show the interconnections among the filters in a StreamIt program. We include the following for each benchmark:

1. The program as originally written *(Original)*.
2. The program after automatic optimizations *(After optimizations)*.
3. The program annotated to show linear filters *(Linear filters)*. Linear filters are denoted by blue and linear structures are denoted by pink.
4. The program annotated to show the optimization regions determined by the automatic selection algorithm *(Optimization regions marked)*. Nodes that are the same color are collapsed together before the appropriate optimization is performed.

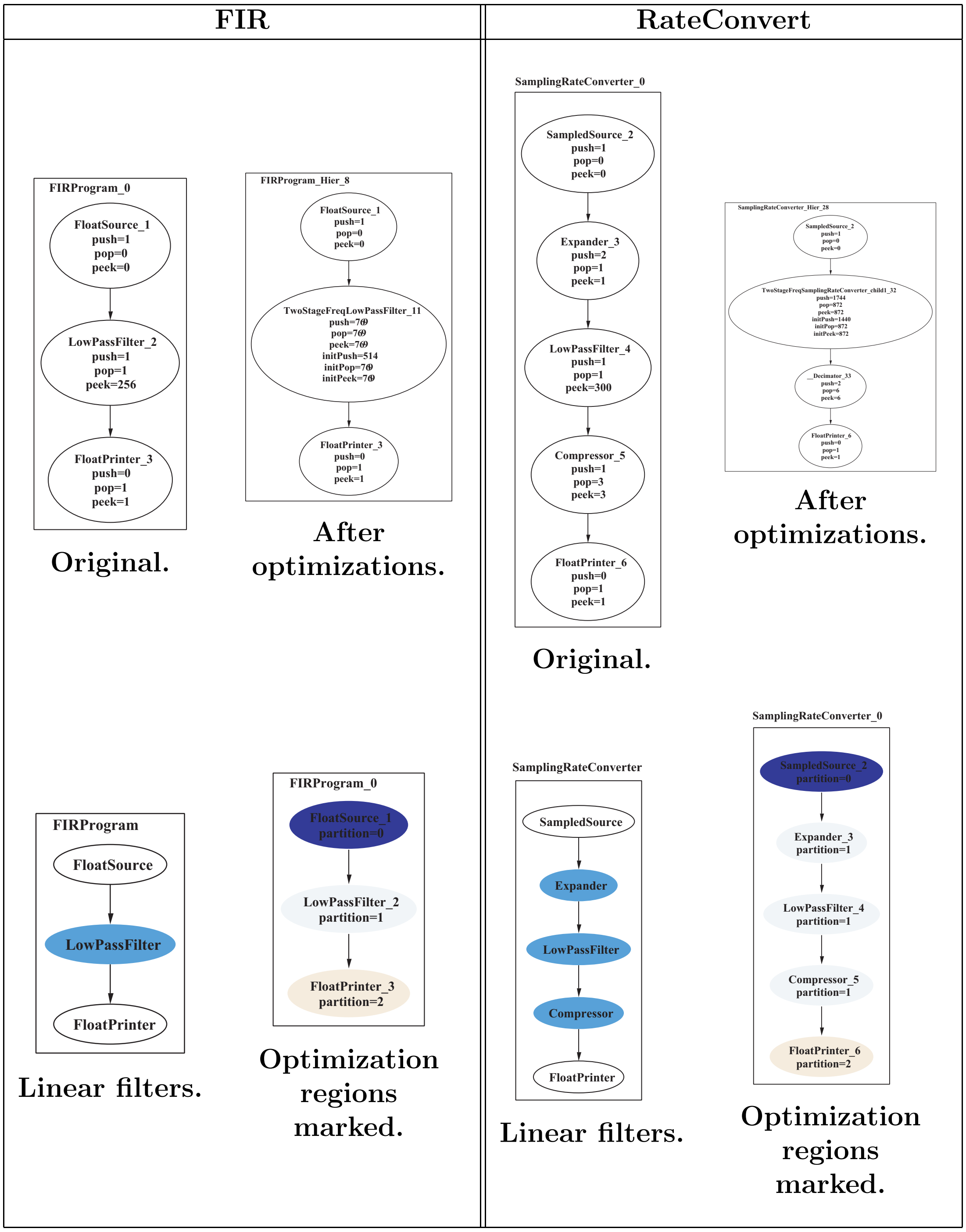


Figure B-1: FIR and RateConvert stream graphs.

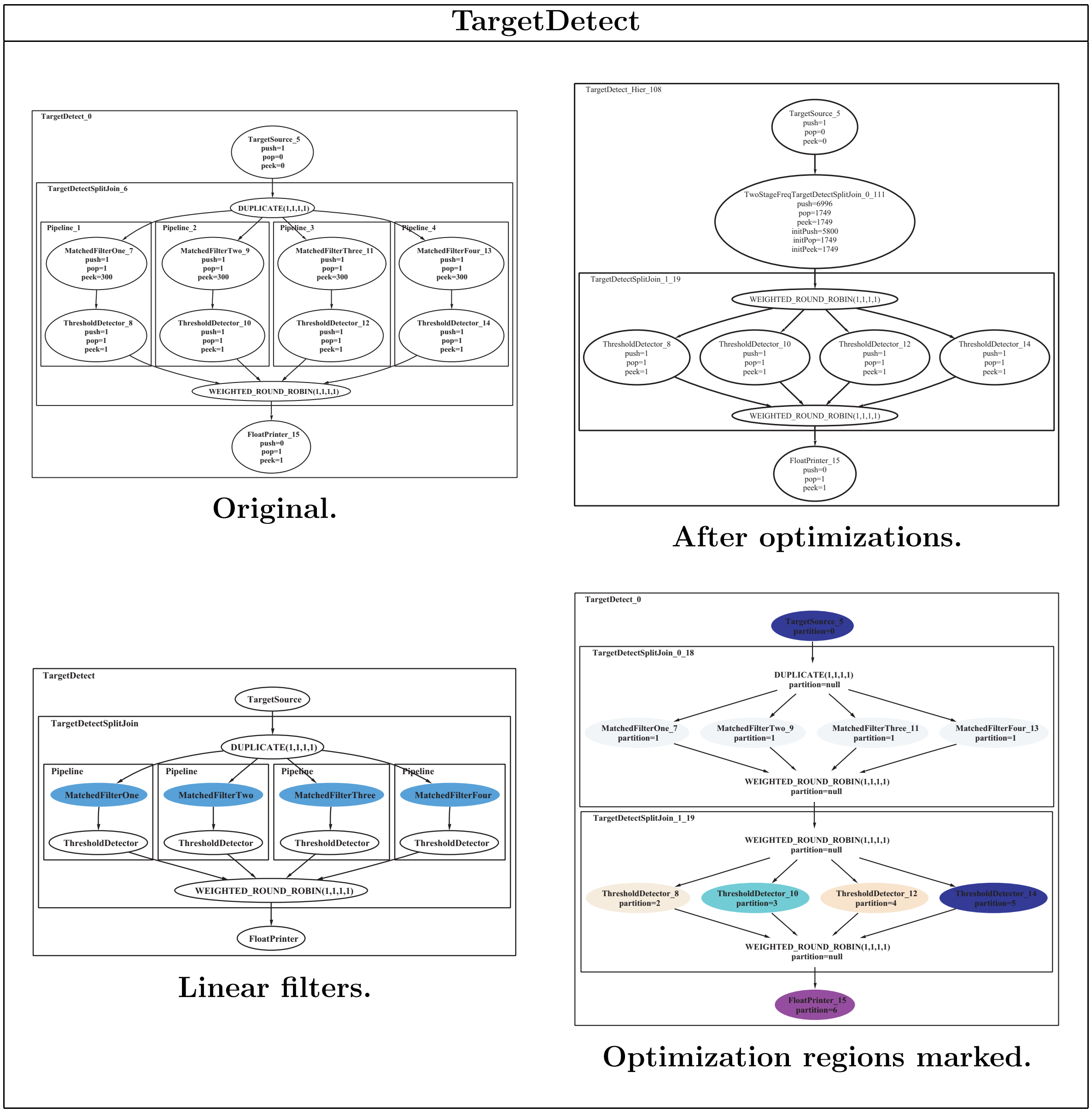


Figure B-2: TargetDetect stream graphs.

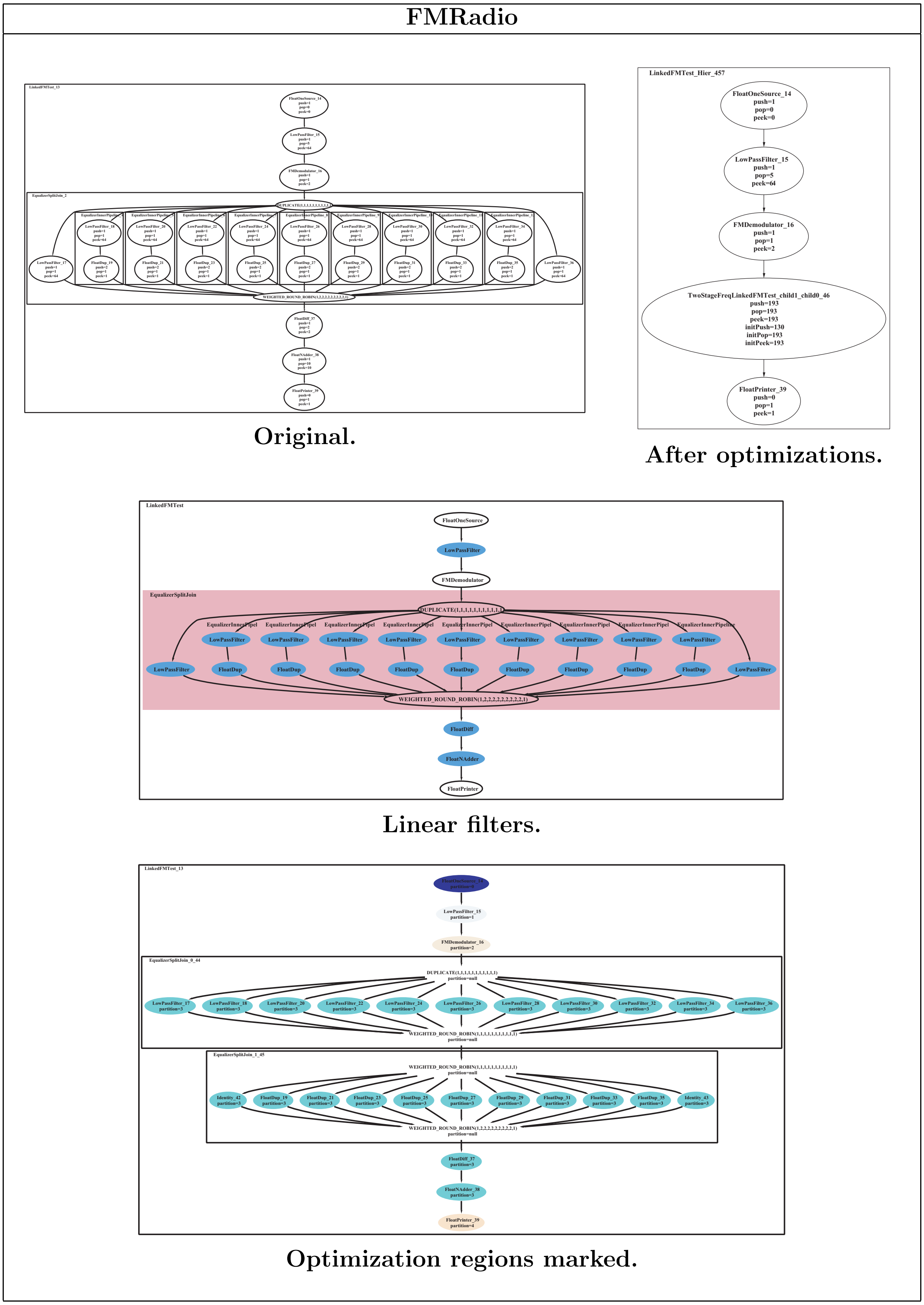
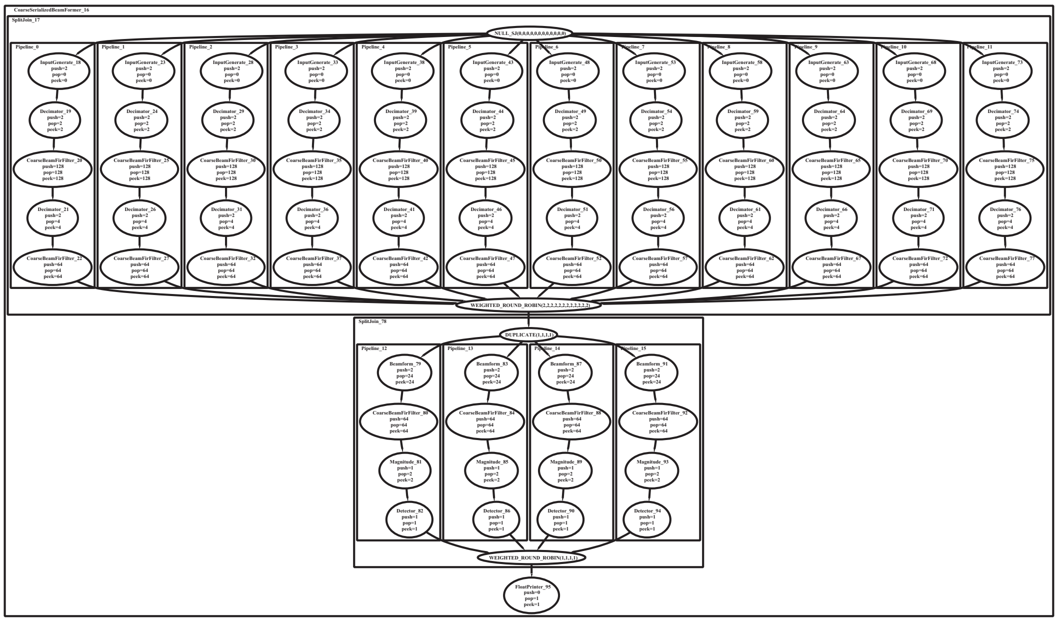
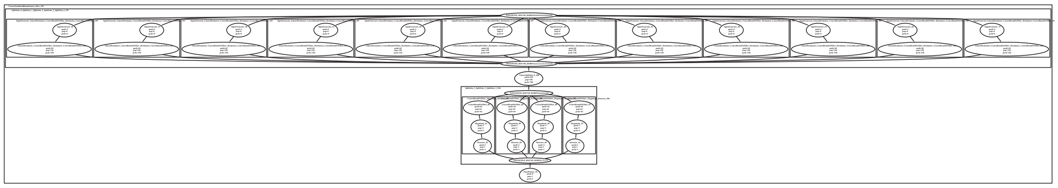


Figure B-3: FMRadio stream graphs.

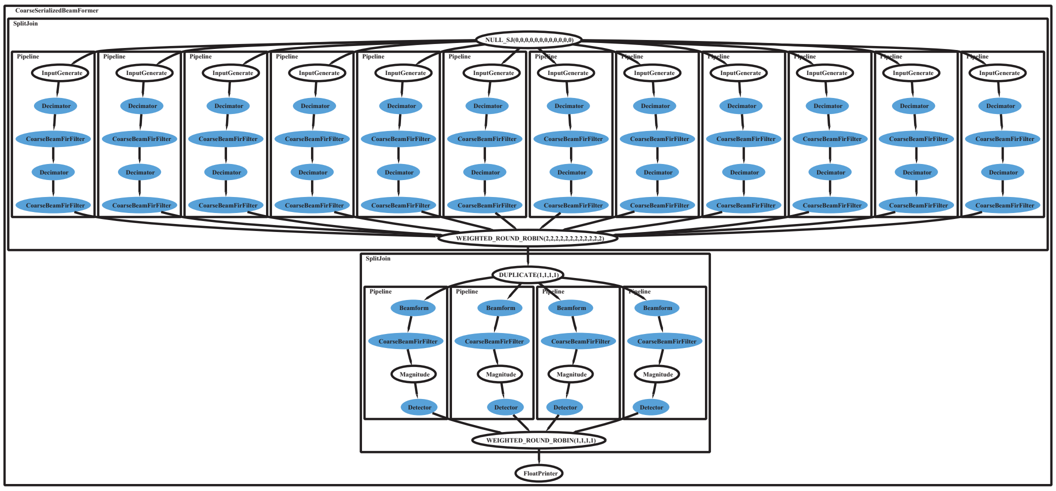
Radar



Original.



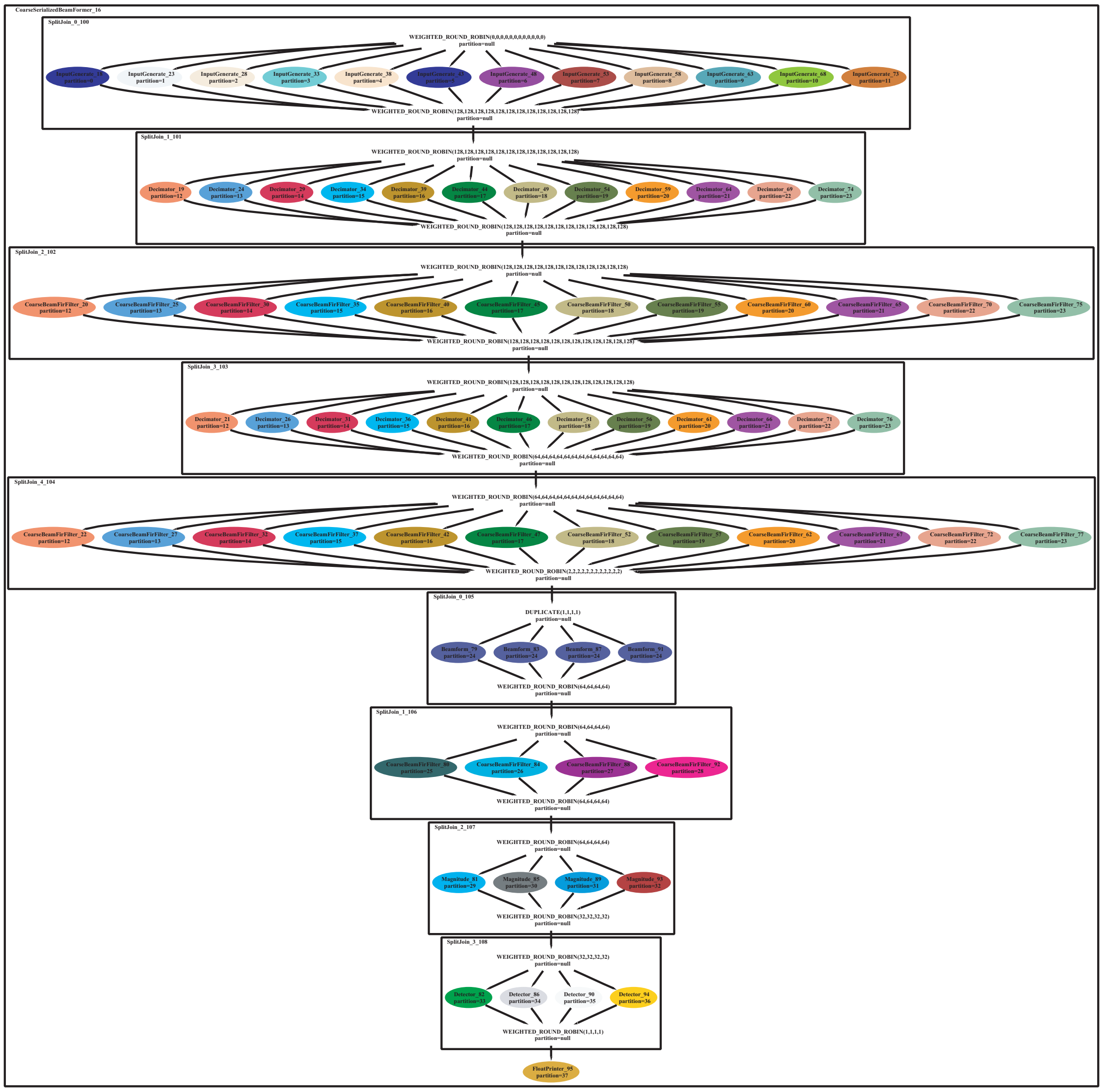
After optimizations.



Linear filters.

Figure B-4: Radar stream graphs (part one).

Radar



Optimization regions marked.

Figure B-5: Radar stream graphs (part two).

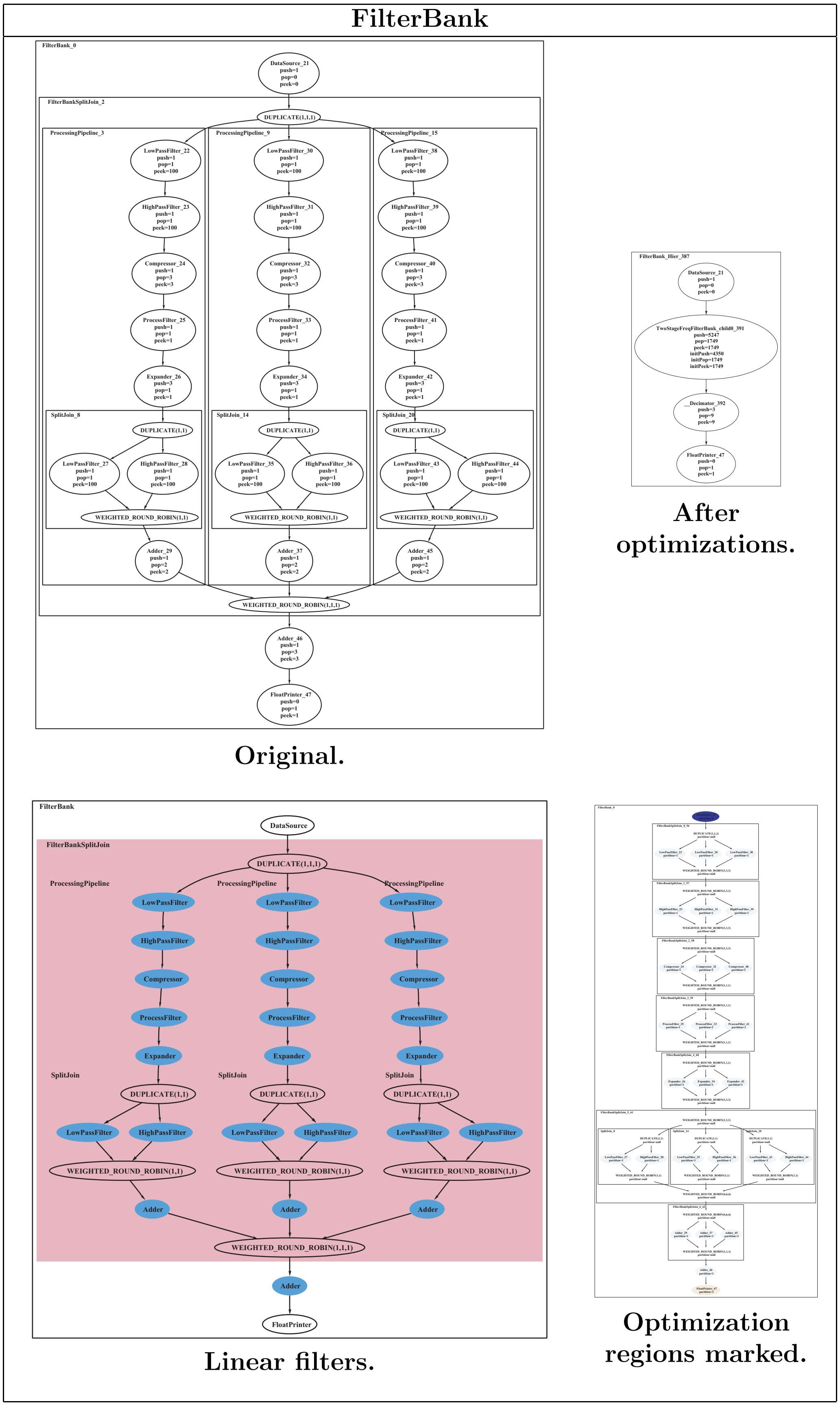


Figure B-6: FilterBank stream graphs.

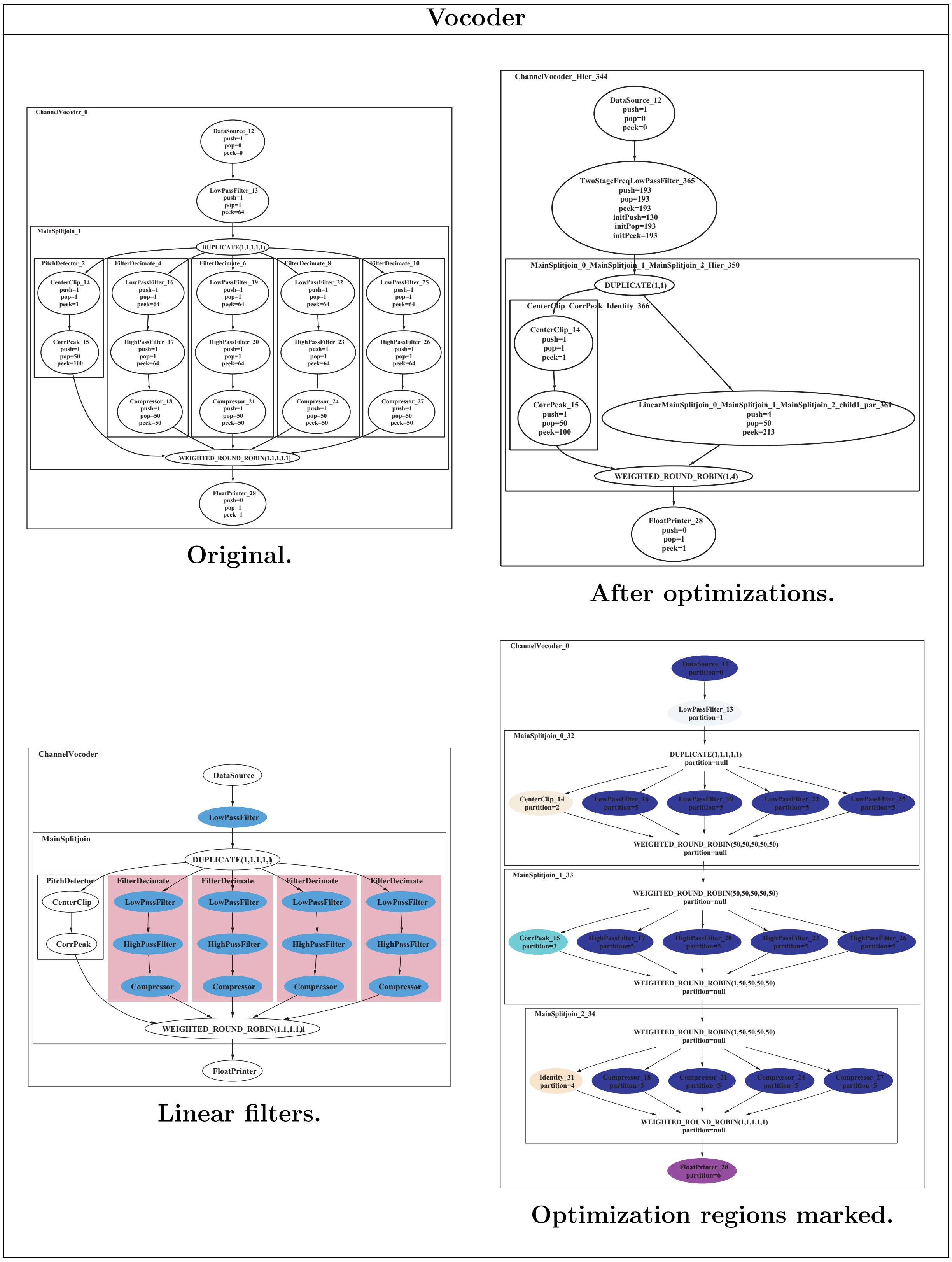


Figure B-7: Vocoder stream graphs.

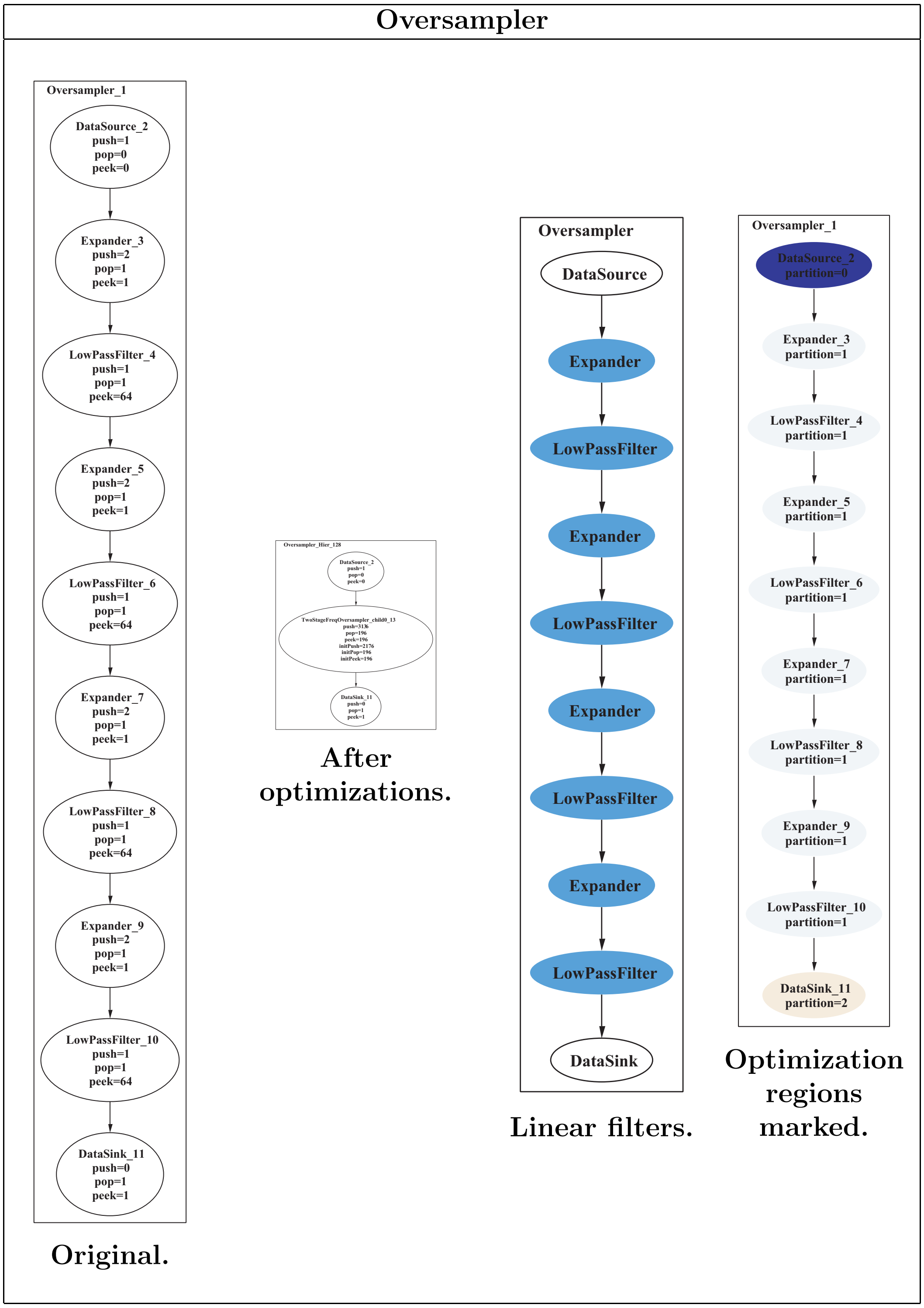
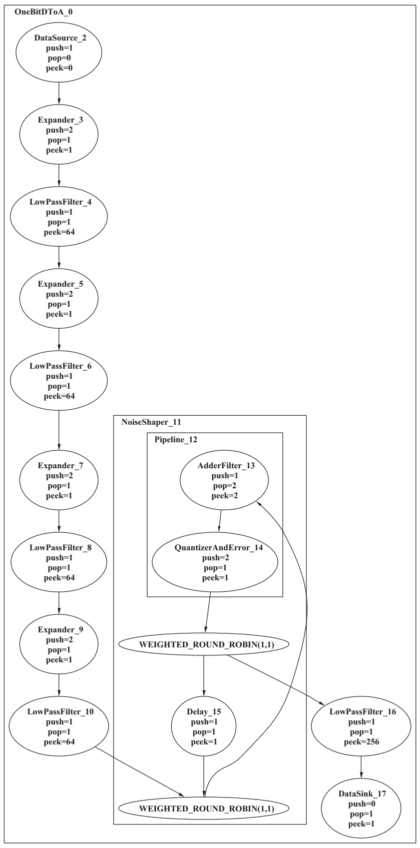
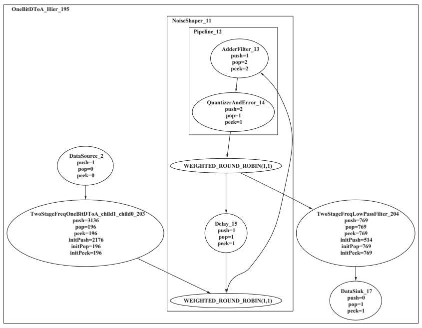


Figure B-8: Oversampler stream graphs.

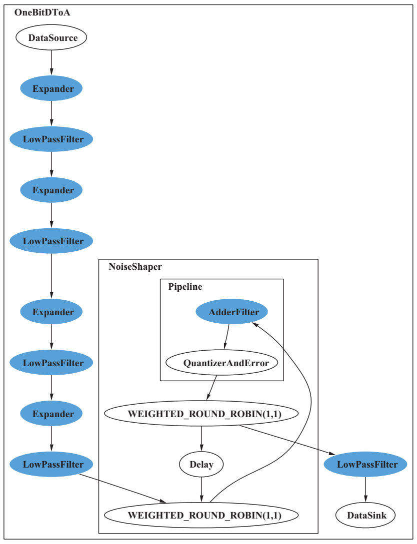
DToA



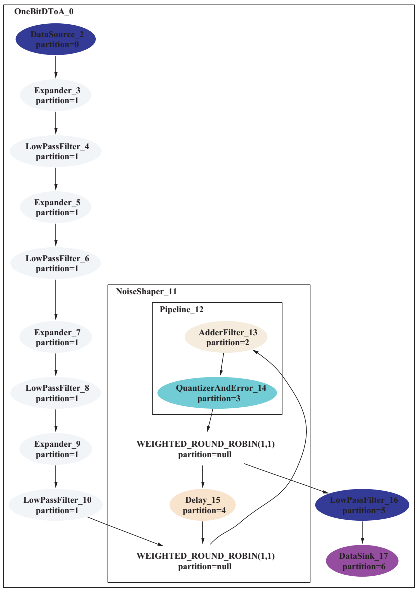
Original.



After optimizations.



Linear filters.



Optimization regions marked.

Figure B-9: DToA stream graphs.

Bibliography

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

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

[3] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1978.

[4] Michele Mae Covell. *An Algorithm Design Environment for Signal Processing.* PhD thesis, Massachusetts Institute of Technology, Research Laboratory for Electronics, 1989.

[5] S. Egner, J. Johnson, D. Padua, M. Püschel, and J. Xiong. Automatic derivation and implementation of signal processing algorithms. *SIGSAM Bulletin*, 35(2):1–19, 2001.

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

[7] M. Frigo and S. Johnson. Homepage of FFTW. http://www.fftw.org.

[8] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference of Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[10] Wanda Gass. Digital signal processors: Past, present, and future. *Massachusetts Institute of Technology, RLE VLSI Seminar Series*, November 5 2002.

[11] 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.

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

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

[14] Michael Gordon. A stream-aware compiler for communication-exposed architectures. Master's thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, August 2002.

[15] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

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

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

[18] Jeremy Johnson, Robert W. Johnson, David A. Padua, and Jianxin Xiong. SPIRAL Home Page. http://www.ece.cmu.edu/ spiral/.

[19] Jeremy Johnson, Robert W. Johnson, David A. Padua, and Jianxin Xiong. Searching for the best FFT formulas with the SPL compiler. *Springer Verlag Lecture Notes in Computer Science*, 2017, 2001.

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

[21] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–155, 1976.

[22] Andrew A. Lamb, William Thies, and Saman Amarasinghe. Linear analysis and optimization of stream programs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[23] J. Lebak. Polymorphous Computing Architecture (PCA) Example Applications and Description. External Report, Lincoln Laboratory, Massachusetts Institute of Technology, 2001.

[24] M.B. Taylor et. al . The raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, March/April 2002.

[25] A. V. Oppenheim and A. S. Willsky with S. H. Nawab. *Signals and Systems*. Prentice-Hall, second edition, 1997.

[26] Alan V. Oppenheim and S. Hamid Nawab, editors. *Symbolic and Knowledge-Based Signal Processing*. Prentice Hall, 1992.

[27] Alan V. Oppenheim, Ronald W. Shafer, and John R. Buck. *Discrete-Time Signal Processing*. Prentice Hall, second edition, 1999.

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

[29] Steven Ryan. Linear data flow analysis. *ACM SIGPLAN Notices*, 27(4):59–67, 1992.

[30] Granville Sewell. *Computational Methods of Linear Algebra*. Ellis Horwood, Chichester, England, 1990.

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

[32] Texas Instruments. *TMS320C54x DSP Reference Set*, volume 2: Mnemonic Instruction Set. 2001.

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

[34] William Thies, Jasper Lin, and Saman Amarasinghe. Partitioning a structured stream graph using dynamic programming. *Work in progress*, April 2003.

[35] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

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

[37] Jianxin Xiong. *Automatic Optimization of DSP Algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[38] Jianxin Xiong, Jeremy Johnson, Robert W. Johnson, and David A. Padua. SPL: A language and compiler for DSP algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, 2001.