Linear Analysis and Optimization of Stream Programs

Andrew A. Lamb, William Thies and Saman Amarasinghe

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

Laboratory for Computer Science
Massachusetts Institute of Technology

ABSTRACT

As DSP programming is becoming more complex, there is an increasing need for high-level abstractions that can be efficiently compiled. 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. Linear filters are common in DSP applications; examples include FIR filters, expanders, compressors, FFTs and DCTs.

We present a linear extraction analysis that automatically detects linear filters based on the C-like code in their `work` function. Once linear filters are identified, we show how neighboring nodes can be collapsed into a single linear representation, thereby eliminating many redundant computations. Also, we describe a method for automatically translating linear nodes into the frequency domain, thereby yielding algorithmic savings for convolutional filters.

We have completed a fully-automatic implementation of the above techniques as part of the StreamIt compiler, and we demonstrate performance improvements that average 400% over our benchmark applications.

1. INTRODUCTION

Digital computation is becoming an increasingly ubiquitous element of modern life. Everything from cell phones to GPS systems to satellite radios require increasingly sophisticated algorithms. Optimization is especially important for this domain, as embedded devices often have high performance requirements and tight resource constraints. Even with the best available C compilers for DSP chips, programmers still turn to assembly code to implement critical parts of embedded applications. This process is time-consuming, error-prone and costly, and must be repeated for each generation of the target architecture. As algorithms and applications continue to grow in complexity, these factors will become unmanageable. There is a pressing need for high-level DSP abstractions that a compiler can consistently reduce to efficient low-level code.

In this paper, we demonstrate that a domain-specific stream language can enable novel high-level DSP optimizations that would otherwise be intractable in a general-purpose language. Our source language is StreamIt, which is specifically designed for high-performance signal processing applications [9, 20]; our analysis focuses on filters that are *linear*. StreamIt is distinguished from a general purpose language in that it makes explicit the large-scale parallelism and regular communication patterns that are characteristic of streaming programs. By analyzing the primitive building block in StreamIt–the filter–our analysis can detect large portions of the application that produce outputs as a linear combination of the inputs; we can exploit this linearity for a number of large-scale optimizations. Though each filter is programmed using imperative C-like code, the separation of filters into autonomous units of the stream graph enables our analysis to be far more effective and efficient than it could be on an equivalent implementation in C alone.

This paper makes the following contributions:

- A linear dataflow analysis that can extract a linear transfer function from the imperative code within a StreamIt filter.
- Combination rules for collapsing neighboring linear nodes into a single linear representation.
- An automated procedure for translating a stream computation into the frequency domain in order to optimize computationally intensive linear nodes.
- An implementation of the above techniques in the StreamIt compiler that automatically improves performance by a factor of five on average and by a factor of 6.5 in the best case.

In the rest of this section, we give a motivating example and background information on StreamIt. Then we present our linear representation (Section 2) and our supporting dataflow analysis (Section 3). Next we describe the combination of linear filters (Section 4) and the translation to the frequency domain (Section 5) before giving results (Section 6), related work (Section 7), and conclusions (Section 8).

1.1 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 2. The imperative C style code that implements this simple DSP application is also shown. 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 3 shows the same filtering process implemented in StreamIt. The StreamIt version is more abstract than the C version. It indicates the communication pattern between filters; it shows the structure of the original block diagram; and it leaves the complexities of buffer management and scheduling to the compiler.

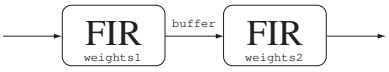


Figure 1: Block diagram of two FIR filters.

```
/* 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;
}

void main() {
  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(weights1, data, i, N);
    data[i] = get_next_input();
  }

  i = 0;
  while(true) {
    /* generate next output item */
    push_output(filter(weights2, buffer, i, N));
    /* generate the next element in the inter-filter buffer */
    buffer[i] = filter(weights1, data, i, N);
    /* get next data item */
    data[i] = get_next_input();
    /* update current start of buffer */
    i = (i+1)%N;
  }
}
```

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

Two optimized versions of the FIR program are shown in Figures 4 and 5. In Figure 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 5, the programmer has done the filtering in the frequency domain, using the FFT and IFFT to translate between time and frequency. Computationally intensive filters and streams are more efficient when done in frequency instead of time.

Our linear analysis can automatically derive both of the implementations in Figures 4 and 5, starting with the code in Figure 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, relying on the compiler to do the analysis and combination.

1.2 StreamIt

StreamIt is a language and compiler for high-performance signal processing [8, 9, 20]. 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

```
float->float pipeline TwoPipe {
  add FIRFilter(weights1);
  add FIRFilter(weights2);
}

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

```
float->float filter CollapsedTwoPipe() {
  float[N] combined_weights;

  init {  /* calculate combined_weights as
             combination of weights1 and weights2 */ }

  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 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 FreqTwoPipe(int L) {
  float[N] combined_weights = ... ;     // calc. combined weights
  complex[N] H = fft(combined_weights); // take FFT of weights
  add FFT(N+L);                         // add FFT stage to stream
  add ElementMultiply(H);               // add multiplication by H
  add IFFT(N+L);                        // add inverse FFT
}
```

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

these properties to the compiler while maintaining a high level of abstraction for the programmer.

StreamIt programs are composed of processing blocks called *filters* which contain an input tape from which they can read values and an output tape to which they can write. Each filter contains a `work` function which describes its atomic execution step in the steady state. 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: 1) `pop()`, which returns the first item on the input tape and advances the tape by one item, 2) `peek(i)`, which returns the value at the ith position on the input tape, and 3) `push(v)`, which 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 three predefined structures (see Figure 6): 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

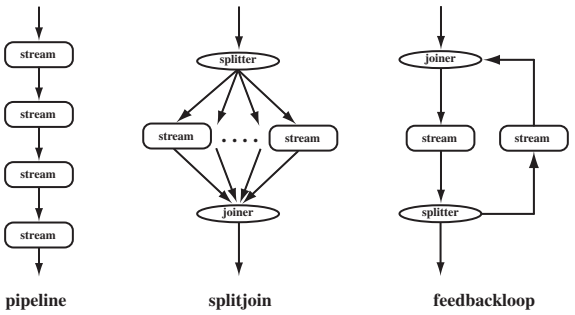


Figure 6: StreamIt structures: `pipeline`, `splitjoin`, and `feedbackloop`.

stream graph. A *stream* is defined to be either a `filter`, `pipeline`, `splitjoin` or `feedbackloop`. Every subcomponents of a structure is a stream, and all streams have exactly one input tape and exactly one output tape.

It has been our experience that most practical 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 [20]. In particular, linear analysis relies heavily on the structure of StreamIt to express stream transformations at a local and hierarchical level.

2. 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 has 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 7 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 $y = xA + \vec{b}$ from input elements x to output elements y. A is a $e \times u$ 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 x is constructed with $x[i] =$ `peek`(i). The node computes $y = xA + \vec{b}$, and then pushes the u elements of y onto the output tape, starting with $y[u-1]$ and proceeding through $y[0]$. Finally, o items are popped from the input tape.

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 $e-i-1$ of A represents the weights given to the ith input element when computing each output. The values in column $u-j-1$ of A and column $u-j-1$ of $\vec{b}$ (*i.e.* the jth column from the right) represent the formula to compute the jth output.

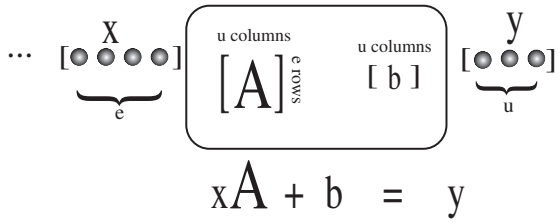


Figure 7: Linear filter as a vector-matrix operation

y	$\in$	program-variable
c	$\in$	constant$_\perp$
$\vec{v}, \vec{b}$	$\in$	vector
$\langle \vec{v}, c \rangle$	$\in$	linear-form
map	$\in$	program-variable $\rightarrow$ linear-form (a hashtable)
A	$\in$	matrix
code	$\in$	list of instructions, each of which can be:

$y_1 :=$ *const* `push`(y_1)
$y_1 :=$ `pop`$()$ (`loop` N *code*)
$y_1 :=$ `peek`(i) (`branch` $code_1$ $code_2$)
$y_1 := y_2$ *op* y_3

Figure 8: Data types for the extraction analysis.

3. 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 on the next page, is a 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 bounds that are known at compile time (if a bound is statically unresolvable, the filter is unlikely to be linear and we disregard it).

Figure 8 contains an overview of our notation for the pseudocode. During symbolic execution, the algorithm maintains a *map* between each program variable y and a linear form $\langle \vec{v}, c \rangle$. In an actual execution, the value of y would be given by $y = \vec{v} \cdot \vec{x} + c$, where $\vec{x}$ represents the input items. The algorithm also maintains two constants *pushcount* and *popcount* that indicate how many items have been pushed and popped so far. As it encounters `push` operations, it builds up the matrix A and vector $\vec{b}$ that will come to represent the linear node.

We 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 8. 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 yet depend on the input. A `pop` operation creates a form $\langle$**BuildCoeff**$(popcount), 0\rangle$, 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 (that is, if either term has a zero vector of input coefficients.) For division, the result is linear only if the divisor is a non-zero constant[1] and for non-linear oper-

[1]Note that if the dividend is zero and the divisor has a

Algorithm 1 Linear extraction analysis.

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

1. Set Peek, Pop, Push equal to I/O rates of filter F.
2. Let $A_0 \leftarrow$ new float[Peek][Push] with each entry $= \bot$
3. Let $\vec{b}_0 \leftarrow$ new float[Push] with each entry $= \bot$
4. $(map, A, \vec{b}, popcount, pushcount) \leftarrow$ **Extract**$(F_{work}, (\lambda x.\bot), A_0, \vec{b}_0, 0, \text{Push} - 1)$
5. **if** A and $\vec{b}$ contain no $\bot$ 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*
$\vec{v} = \vec{0}_{Peek}$
$\vec{v}[\text{Peek} - pos] = 1$
return $\vec{v}$

Global Variables: int Peek, Pop, Push

ations (*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 $\bot$, 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 $\sqcap$, which we define for scalar constants, vectors, matrices, linear forms, and maps. $c_1 \sqcap c_2$ is defined according to the lifted lattice constant$_\bot$. That is, $c_1 \sqcap c_2 = c_1$ if and only if $c_1 = c_2$; otherwise, $c_1 \sqcap c_2 = \bot$. For vectors, matrices, and linear forms, $\sqcap$ is defined element-wise; for example, $A' = A_1 \sqcap A_2$ is equivalent to $A'[i,j] = A_1[i,j] \sqcap A_2[i,j]$. For maps, the meet is taken on the values: $map_1 \sqcap map_2 =$ *map'*, where $map'.get(x) = map_1.get(x) \sqcap map_2.get(x)$.

Our implementation of linear extraction also deals with function calls. 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 $\bot$.

4. 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 automatically eliminates 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 serves as a building block for the combination techniques. We then give rules for col-

non-zero coefficients vector, we cannot conclude that the result is zero, since certain values of the inputs might cause a singularity.

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 **do**
switch *code*[i]

case $y :=$ *const*
$map.put(y, (\vec{0}_{Peek}, const))$

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

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

case `push`(y)
$\langle \vec{v}, c\rangle \leftarrow map.get(y)$
if $pushcount = \bot$ **then** *fail*
$A[*, pushcount] \leftarrow \vec{v}$
$b[pushcount] \leftarrow c$
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 $\vec{v}_2 = \vec{0}_{Peek}$ **then**
$map.put(y_1, (c_2 * \vec{v}_3, c_2 * c_3))$
else if $\vec{v}_3 = \vec{0}_{Peek}$ **then**
$map.put(y_1, (c_3 * \vec{v}_2, c_3 * c_2))$
else
$map.put(y_1, \bot)$

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 $\vec{v}_3 = \vec{0}_{Peek} \wedge y_3 \neq 0$ **then**
$map.put(y_1, (\frac{1}{c_3} * \vec{v}_2, c_2/c_3))$
else
$map.put(y_1, \bot)$

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

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$(code_2, map, A, \vec{b}, popcount, pushcount)$
$map \leftarrow map_1 \sqcap map_2$
$A \leftarrow A_1 \sqcap A_2$
$\vec{b} \leftarrow \vec{b}_1 \sqcap \vec{b}_2$
$popcount \leftarrow popcount_1 \sqcap popcount_2$
$pushcount \leftarrow pushcount_1 \sqcap pushcount_2$

end for
return (map, A, $\vec{b}$, *popcount*, *pushcount*)

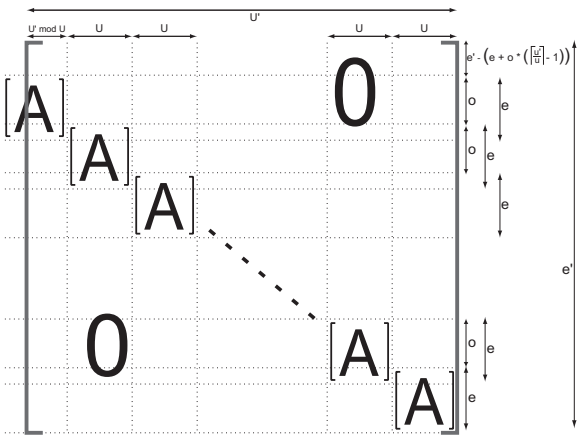


Figure 9: Expanding a linear node to rates (e', o', u').

lapsing `pipeline`s and `splitjoin`s into linear nodes; we do not yet deal with `feedbackloop`s as they require the notion of "linear state" which we do not describe here.

4.1 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 [14].

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. This expansion can be done as follows.

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* $\mathbf{expand}(\lambda, e', o', u') = \{A', \vec{b}', e', o', u'\}$, *where $\mathbf{A}'$ is a $e' \times u'$ matrix and $\tilde{\mathbf{b}}'$ is a u'-element row vector:*

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

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

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

The intuition behind linear expansion is straightforward (see Figure 9.) Linear expansion aims to scale the push, pop, and peek 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. 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 zero's 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.

4.2 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* $\mathbf{pipe}(\lambda_1, \lambda_2) = \{\mathbf{A}', \tilde{\mathbf{b}}', \mathbf{e}', \mathbf{o}', \mathbf{u}'\}$ *represents an equivalent node that can replace the original two. Its components are as follows:*

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

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.

However, 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 $u_1 \neq e_2$, this will require constructing expanded nodes λ_1^e and λ_2^e in which the push and peek rates match (and thus 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.

However, 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). In general, the compiler can identify performance-degrading combinations when the number of non-zero elements in A' exceeds the sum of non-zero elements in A_1^e and A_2^e.

4.3 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 output tape of the leftmost child are placed onto the overall output tape, then w_1 elements are taken from the second leftmost child, and so on. The process repeats itself after one complete set of $\sum_{i=0}^{N-1} w_i$ 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 `splitjoin`s 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 translate the `splitjoin` to use a duplicate splitter and then rely on the first analysis to construct a linear node. We describe these translations below.

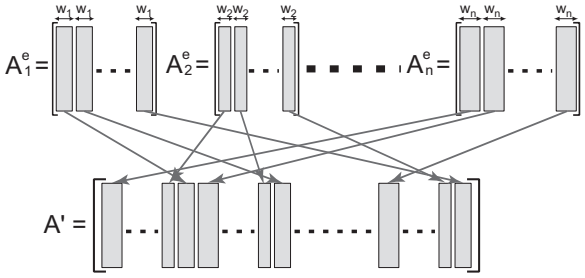


Figure 10: Matrix resulting from combining a `splitjoin` of rate matched sub streams.

4.3.1 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 need to expand each child node according to its multiplicity in the schedule. Secondly, we need to 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 roundrobin joiner (see Figure 10). This third step is simplified by the fact that, with a duplicate splitter, each row of a child's linear representation refers to the same input element to the splitjoin.

The following transformation describes splitjoin combination in mathematical terms.

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* $\mathbf{splitjoin}(s) = \{\mathbf{A}', \tilde{\mathbf{b}}', \mathbf{e}', \mathbf{o}', \mathbf{u}'\}$ *represents an equivalent node that can replace the entire stream s. Its components are as follows:*

$$\begin{aligned} joinRep &= lcm(\tfrac{lcm(u_1,w_1)}{w_1}, \dots, \tfrac{lcm(u_n,w_n)}{w_n}) \\ maxPeek &= max_i(o_i * rep_i + e_i - o_i) \end{aligned}$$

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

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

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

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

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

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

The above 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; *joinRep* 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 k'th 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 caused shorter matrices to be padded with rows of zero's at the top.

The final phase of the transformation is to re-arrange the columns of the child matrices into the columns of A' and $\vec{b}'$. Figure 10 elucidates this process, though its notation is somewhat cumbersome. The equation can be understood as follows: for the k'th child and the m'th cycle of the joiner, the n'th item that is pushed by child k will appear appear at a certain location on the joiner's output tape. This location (relative to the start of the node's execution) is $n + m * wSum_n + wSum_k$, as the reader can verify. But since the right-most column of each array A holds the first item to be 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 b is analogous.

Finally, 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 $joinRep * wSum_m$ 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 $\vec{b}'$ defined above–define a linear node that exactly represents the parallel combination of child nodes that are fed by a duplicate splitter.

4.3.2 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 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, this involves adding 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 roundrobin joiner j, the transformed* **rr-to-dup**(s) *is a splitjoin with a duplicate splitter, linear child nodes $\lambda'_0 \ldots \lambda'_{n-1}$, and roundrobin joiner j. The child nodes are computed as follows:*

$$
\begin{aligned}
vSum_k &= \textstyle\sum_{i=0}^{k-1} v_i \\
vTot &= vSum_{n-1} \\
\forall k \in [0, n-1]&: \\
&decimate[k] \text{ is a linear node } \{A, \vec{0}, vTot, vTot, vTot\} \\
&\text{where } A[i,j] = \begin{cases} 1 \text{ if } i = j \ \wedge \\ \quad vSum_k < i < vSum_{k+1} \\ 0 \text{ otherwise} \end{cases} \\
\lambda'_{\mathbf{k}} &= pipe(decimate[k], \lambda_k)
\end{aligned}
$$

In the above translation, we utilize the linear pipeline combinator *pipe* to construct each new child node λ_i^e as a composition of a decimator and the original node λ_i. Each decimator is a square matrix that produces and consumes $vTot$ items, which is the number of items processed in one cycle of the roundrobin joiner. Those items intended for stream i are copied with a coefficient of 1, while all others are eliminated with a coefficient of 0.

4.4 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 tend to 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 sample 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 filters 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 FM-Radio 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.

5. TRANSLATION TO FREQUENCY DOMAIN

Our linear analysis framework provides a compile time formulation of the computation that a linear stream is performing and we use this information to exploit well known domain specific optimizing transformations. Using linear node information, our compiler identifies convolution regions that require substantially fewer computations when they are translated into the frequency domain.

Calculating a convolution sum is a common and fundamental operation in discrete time signal processing. If the

convolution region is sufficiently large, transforming the data to the frequency domain, performing a simple vector multiply and converting back to the time domain requires fewer operations than the straightforward convolution.

The transformation from convolution sum into frequency multiplication has always been done explicitly by the programmer because no compiler analysis has had the information to determine when a convolution sum is being 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, individual filters may not perform sufficiently large convolutions to merit this transformation, but after a linear combination of multiple filters the transformation will be beneficial. Furthermore, differing architectural features makes the task of portably implementing computational transforms even more daunting.

5.1 Transformation Overview

The convolution sum $y[n] = x[n] * h[n]$ is defined as $y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$. In StreamIt, if a stream is calculating a convolution sum the input ($x[n]$) and output ($y[n]$) correspond exactly to the input and output tapes where where n denotes an index in the time domain. Furthermore, a stream will be computing a convolution sum when $o = 1$ in which case we can identify the values $h[n]$ as exactly the columns of A in the corresponding linear node.

Calculating the convolution in the frequency domain is more efficient because of the existence of the Fast Fourier Transform (FFT) algorithm for quickly calculating the Discrete Fourier Transform (DFT) of a signal. Calculating a convolution takes $O(N^2)$ time, and performing an equivalent computation using an FFT takes only two $O(Nlg(N))$ time-frequency conversions coupled with an $O(N)$ frequency domain vector multiply.

To compute the convolution of two discrete time signals, $x[n] * h[n]$, first the DFT of both sequences ($X[k]$ and $H[k]$) is calculated where k denotes an index in the frequency domain. Multiplying $X[k]$ and $H[k]$ element-wise produces a new sequence $Y[k]$, and taking the inverse DFT (IDFT) of $Y[k]$ produces $y[n]$ which is precisely the same as $x[n] * h[n]$.

When the compiler identifies a stream that computes a convolution sum, it generates a new filter which computes $H[k]$ at compile time. The stream's `work` function is changed to perform the following:

1. $X[k]$ is is calculated from the input tape using an FFT algorithm.
2. $X[k]$ is multiplied element-wise with $H[k]$ to produce $Y[k]$.
3. $y[n]$ is obtained by transforming $Y[k]$ back to the time domain using an inverse FFT.
4. The appropriate values of $y[n]$ are pushed on to the output tape.

5.2 Automatic Transformation

To implement this transformation, the compiler needs to compute $H[k]$ at compile time. The compiler transforms FIR filters which have $h[n]$ of length e and push rate $u = 1$, which requires expanding the filter to overcome the constant overhead factors. Therefore, the transformed filter needs to produce more than one output on each execution of `work`. The number of outputs, N, to produce on each execution of `work` is determined automatically by the compiler and set to approximately $2e$, a number determined by empirical observations. N is then rounded up such that $N + 2(e-1)$ is a power of two which results in the most efficient FFT calculations.

The frequency transformation generates a new filter that peeks $e' = N + e - 1$ items each execution where the original stream used only e. The compiler automatically computes the complex values of $H[k] = FFT(N + 2(e-1), h[n])$, the $N + 2(e-1)$ point DFT of $h[n]$ at compile time and saves them as constants in the filter. A new compiler-generated `work` function is generated that calculates the complex valued $X[k] = DFT(N+2(e-1), x[n])$, the $N+2(e-1)$ DFT of the input and then calculates $Y[k]$, the element-wise vector product of $X[k]$ and $H[k]$. Finally, the new `work` function performs the inverse FFT $y[n] = IFFT(N + 2e - 2, y[n])$.

Using $N+e-1$ input items produces a length $N+2(e-1)$ convolution sum, of which both the first and last $e-1$ values are incorrect. Since every output requires the value of e inputs to calculate, without filter state only the middle N items of $y[n]$ are actual output values. In an *overlap-discard* implementation, the `work` function simply uses the middle N values of $y[n]$ and discards the $e-1$ elements on both ends, and advances the input tape by N. The following $N+e-1$ input values are then used to produce the next N outputs.

The *overlap and add* method [16] is well known. This algorithm exploits the fact that the overlapping values of $y[n]$ contain partial output computations due to both the previous and the next $N+e-1$ inputs. The first $e-1$ of $y[n]$ are part of the computation from the previous invocation of the `work` function and the last $e-1$ are part of the next invocation. For the automatic frequency transformation, the compiler creates a filter which first pushes $y[n] + p[n]$ for $0 \leq n \leq (e-1)-1$, where $p[n]$ contains partial results from the previous invocation. Then the filter pushes the values of $y[n]$ for $e-1 \leq n \leq N+(e-1)-1$. Finally, $p[n]$ is updated such that $p[n] = y[n+(N+e-1)]$ for $0 \leq n \leq N+2(e-1)-1$ which are used on the next invocation of `work`; the input tape is then advanced by $N + e - 1$.

6. RESULTS

Our compiler currently has two linear analysis optimizations. The first, *linear replacement*, replaces the largest linear hierarchical streams possible with filters that directly compute the calculation specified by the corresponding linear nodes. The second optimization, *frequency replacement*, replaces all streams which implement a sufficiently long convolution using the frequency transformation described in Section 5. Below we describe experiments and results that demonstrate performance improvements due to these two optimizations.

6.1 Measurement Methodology

Both linear replacement and frequency replacement increase performance by decreasing the number of floating point computations (principally multiplications) required per output. Our measurement platform is a Dual Intel 2.2 GHz P4 Xenon processor 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` with `-O2` optimization.

To measure the number of runtime multiplications we use a simple instruction counting program written using the Dy-

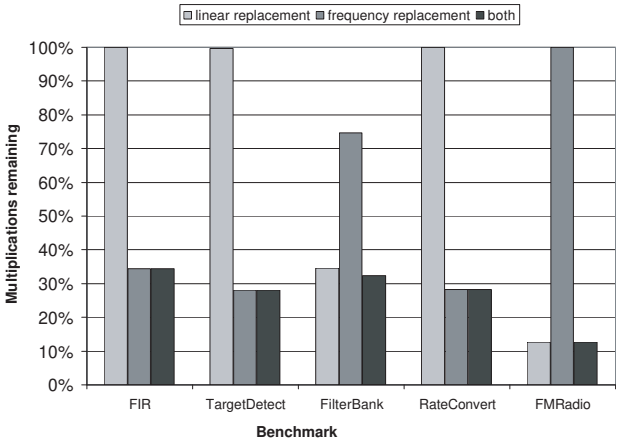


Figure 11: Percent of multiplication operations remaining after performing linear replacement, frequency replacement, and both.

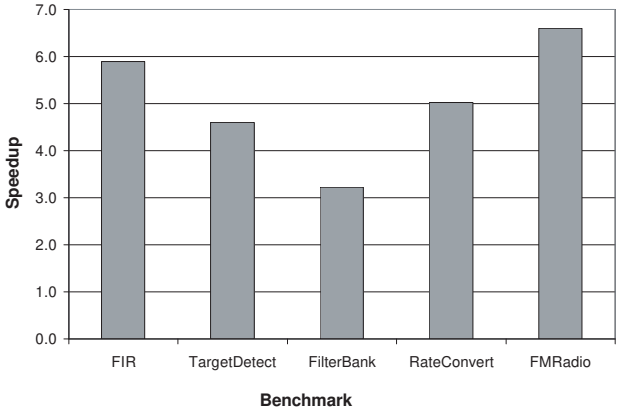


Figure 12: Execution speedup for each of the benchmarks with both linear replacement and frequency replacement optimizations.

namoRIO[1] infrastructure. There are no standard benchmarks written yet for StreamIt, so we use a set of representative programs[2] which perform computations that are found in actual streaming applications: 1) **FIR**, a single 256 point rectangularly windowed low pass FIR filter ($\omega_c = \frac{\pi}{3}$); 2) **TargetDetect**, four matched filters performing threshold target detection in parallel; 3) **FilterBank**, a multi-rate signal decomposition processing block common in communications and image processing; 4) **RateConvert**, an audio signal down sampler that converts the sampling rate by a non-integral factor; 5) **FMRadio**, an FM software radio with equalizer.

6.2 Performance

To determine the effects of our linear replacement and frequency replacement optimizations, we compiled each benchmark program with linear replacement, with frequency replacement and with both linear replacement and frequency replacement. Figure 11 shows the reduction of multiplications due to our optimizations.

For the FIR filter, all of the multiplication reduction comes from the frequency replacement optimization because the entire application is comprised of a single filter calculating a convolution sum so there is nothing to combine. The multiplication reduction in TargetDetect is also solely due to the frequency transformation because threshold detection is non-linear which makes the parallel computation blocks uncollapsible. Finally, RateConvert contains a large low pass filter which also benefits from frequency replacement.

All of the multiplication reduction in the FMRadio benchmark is due to the automatic combination of parallel equalizer computations. Table 1 shows the number of nodes in each benchmark both before and after linear replacement.

FilterBank is the only benchmark where multiplications are reduced more by both optimizations than either alone. Linear replacement reduces the required multiplications because it can combine the action of parallel analysis and synthesis channels into an overall FIR filter. Frequency replacement alone helps multiplication reduction only somewhat because FilterBank contains fairly small FIR filters. However, frequency replacement speeds up the calculation of the overall filter generated by linear replacement, and thus decreases the number of multiplies even further.

Reducing computation does not necessarily translate into execution time improvement, but as Figure 12 demonstrates, our benchmarks speedup on average by a factor of five and by a factor of 6.5 in the best case. Our current implementation takes advantage of the machine tuned FFT package FFTW [5]), to perform the necessary time-frequency conversions. The benchmarks where multiplication reduction is due only to linear replacement (FMRadio) also show a large speedup.

[2]Stream graphs appear in Appendix A.

7. 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[12, 13, 4]. Using a feedback-directed search process, DSP transforms are optimized for the underlying architecture. The input language to SPIRAL is SPL[23, 22], 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 [21] also aims to produce fast libraries for linear algebra manipulations, focusing on adaptive library generation for varying architectures. FFTW [5] is a runtime library of highly optimized FFT's that dynamically adapt to architectural variations. Again, StreamIt is distinguished by its extraction and optimization of linear filters from general user-level code.

ADE (A Design Environment) is a system for specifying, analyzing, and manipulating DSP algorithms [3]. ADE includes a rule-based system that can search for improved

	Originally				After Linear Replacement		
Benchmark	Filters (linear)	Pipelines (linear)	SplitJoins (linear)	Average vector size	Filters	Pipelines	SplitJoins
FIR	3 (1)	1(0)	0 (0)	256	3	1	0
TargetDetect	10 (4)	1 (0)	1 (0)	300	10	1	1
FilterBank	27 (25)	17 (9)	4 (3)	51	15	8	1
RateConvert	5 (3)	2 (0)	0 (0)	335	5	2	0
FMRadio	25 (22)	3 (1)	2 (2)	33	5	1	0

Table 1: Statistics for benchmarks before and after transformations.

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 [15].

A number of other programming languages are oriented around a notion of a stream (see [19] for a survey.) Synchronous languages such as LUSTRE [10], Esterel [2], and Signal [7] target the embedded domain, while languages such as Occam [11], SISAL [6] and StreamC [17] target parallel and vector targets. However, none of the compilers for these languages have coarse-grained, DSP-specific analyses such as linear filter detection.

Note that the "linear data flow analysis" of Ryan [18] is completely unrelated to our work; it aims to do program analysis in linear time.

8. CONCLUSION

This paper 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 linear dataflow analysis, linear combination, and automated frequency translation, we can improve execution speed by a factor of 6.5.

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

9. REFERENCES

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

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

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

[4] 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.

[5] M. Frigo. A Fast Fourier Transform Compiler. PLDI, 1999.

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

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

[8] M. Gordon. A stream-aware compiler for communication-exposed architectures. Master's thesis, MIT Laboratory for Computer Science, August 2002.

[9] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. ASPLOS, 2002.

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

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

[12] J. Johnson, R. W. Johnson, D. A. Padua, and J. Xiong. SPIRAL Home Page. `http://www.ece.cmu.edu/~spiral/`.

[13] J. Johnson, R. W. Johnson, D. A. Padua, and J. Xiong. Searching for the best FFT formulas with the SPL compiler. *LNCS*, 2017, 2001.

[14] M. A. Karczmarek. Constrained and phased scheduling of synchronous data flow graphs for the streamit language. Master's thesis, MIT LCS, October 2002.

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

[16] A. V. Oppenheim, R. W. Shafer, and J. R. Buck. *Discrete-Time Signal Processing*. Prentice Hall, 1999.

[17] S. Rixner, W. J. Dally, U. J. Kapani, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *HPCA*, Dallas, TX, November 1998.

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

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

[20] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. of the Int. Conf. on Compiler Construction (CC)*, 2002.

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

[22] J. Xiong. *Automatic Optimization of DSP Algorithms*. PhD thesis, Univ. of Illinois at Urbana-Champaign, 2001.

[23] J. Xiong, J. Johnson, R. W. Johnson, and D. A. Padua. SPL: A language and compiler for DSP algorithms. In *PLDI*, 2001.

Appendix A: Stream Graphs of Example Programs

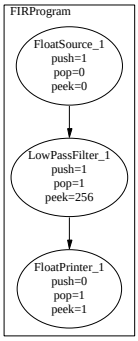


Figure 13: Stream graph for the FIR benchmark.

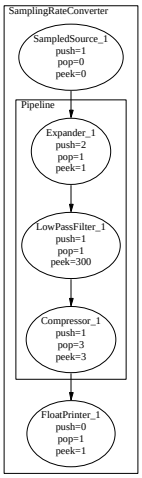


Figure 14: Stream graph for the sampling rate converter benchmark.

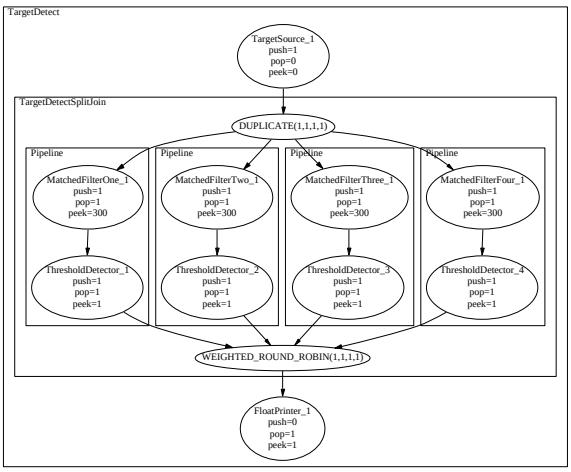


Figure 15: Stream graph for the target detector benchmark.

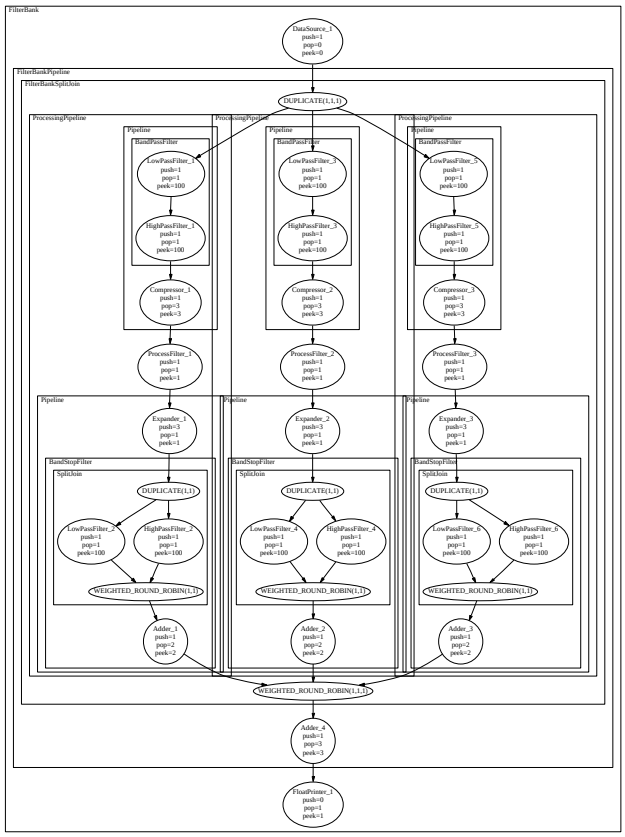


Figure 16: Stream graph for the filter bank benchmark.

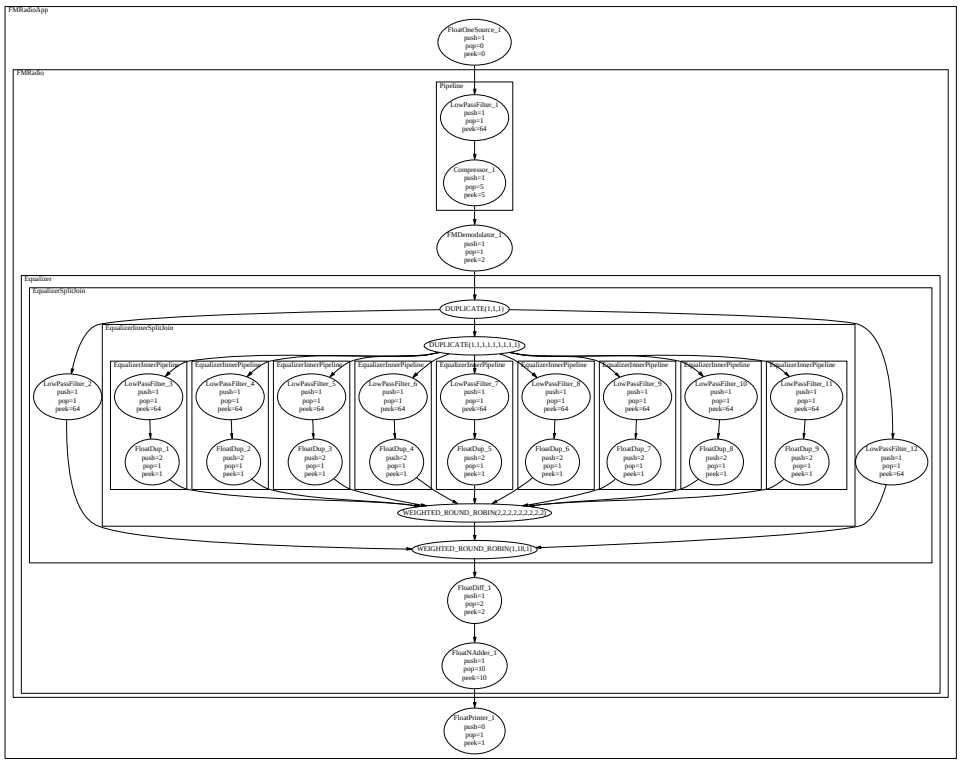
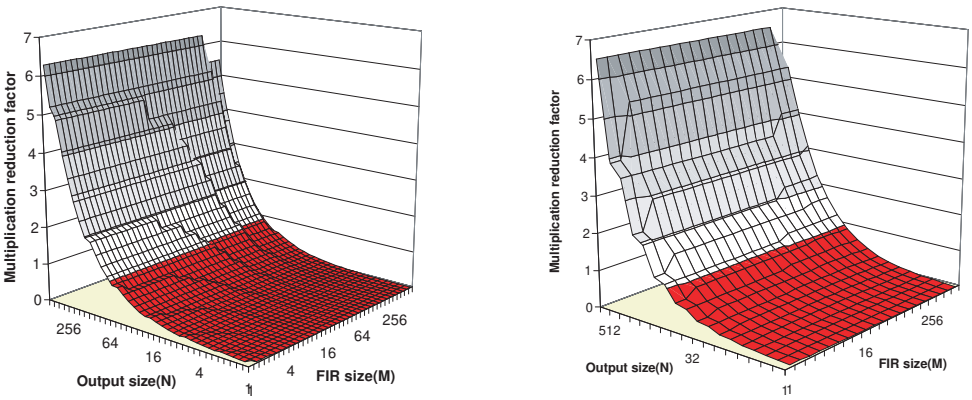


Figure 17: Stream graph for the FM radio benchmark.

Appendix B: Frequency Replacement Scaling



(a) Theoretical **(b) Empirical**

Figure 18: Plots showing the theoretical and empirical multiplication reduction factor as a function of the size of the FIR (M) and the number of outputs produced per calculation (N). The dark regions denote an increase in the required number of multiplications and the light regions a reduction.

Frequency replacement is an effective optimization because the asymptotic bounds for frequency domain computation is lower than the bound for the time domain computation. We determined empirically the point at which frequency replacement improves performance.

Direct convolution requires $O(MN)$ multiplies. The FFT requires $O(N + 2(M-1))lg(N + 2(M-1))$ multiplications for both the conversion to and from the frequency domain, and multiplying two $N+2(M-1)$ vectors in the frequency domain requires $O(N + 2(M-1))$ multiplications. Direct convolution produces N outputs per iteration and the frequency implementation produces $N + M - 1$ outputs every iteration. We define the "multiplication reduction factor" to be the number of multiplies required per output using convolution divided by the the number of multiplies per output using the frequency transformation.

Figure 18 (a) shows a plot of the theoretical multiplication reduction factor and Figure 18 (b) shows the same reduction factor measured empirically. The roughness in both the theoretical results and the data is due to the fact that for the best FFT performance, $N + 2(M-1)$ must be a power of two, and the compiler automatically adjusts N upward to satisfy this requirement. The theoretical reduction numbers account for the fact that our implementation requires four floating point multiplication operations to perform a complex valued multiply in the frequency domain.

Based on the above analysis, our current compiler applies the frequency replacement transformation on FIR filters that have length 90 or greater. The target output rate, N, is automatically set to be twice the FIR length.