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 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, as well as 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 will give the maximal benefit.

We have completed a fully-automatic implementation of the above techniques as part of the StreamIt compiler, and we demonstrate a 450% performance improvement over our benchmark suite.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors; D.3.2 [**Programming Languages**]: Language Classifications; D.2.2 [**Software Engineering**]: Software Architectures; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Performance, Design, Algorithms

Keywords

Stream Programming, StreamIt, Optimization, Embedded, Linear Systems, Algebraic Simplification, DSP, FFT

PLDI'03, June 9–11, 2003, San Diego, California, USA.

1. INTRODUCTION

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 could be exploited to obtain algorithmic speedups. Apart from requiring expert knowledge, this effort is time-consuming, error-prone, and 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.

In this paper, 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 [7, 19], which is a high-level language for 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 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:

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

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

In the rest of this section, we give a motivating example and background information on StreamIt. Then we present our linear node representation (Section 2) and our supporting dataflow analysis (Section 3). Next we describe structural transformations on linear nodes (Section 4), a frequency domain optimization (Section 5) and an optimization selection algorithm (Section 6). Finally, we present results (Section 7), related work (Section 8) and conclusions (Section 9).

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 1. The imperative C style code that implements this simple DSP application is shown in Figure 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.

```
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 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 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.

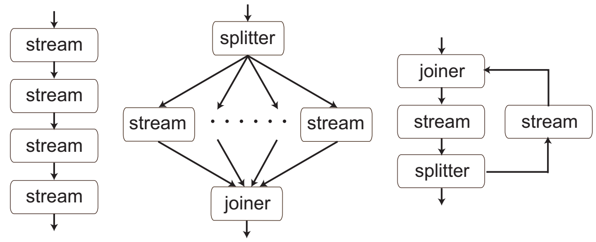
```
float->float pipeline FreqTwoFilters(float[N] w1, float[N] w2) {
  float[N] combined_weights = ... ;     // calc. combined weights
  complex[N] H = fft(combined_weights); // take FFT of weights
  add FFT();                            // add FFT stage to stream
  add ElementMultiply(H);          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

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

Figure 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 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. Computationally intensive streams are more efficient in frequency than in 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 and rely on the compiler for combination and transformation.



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

Figure 6: Stream structures supported by StreamIt.

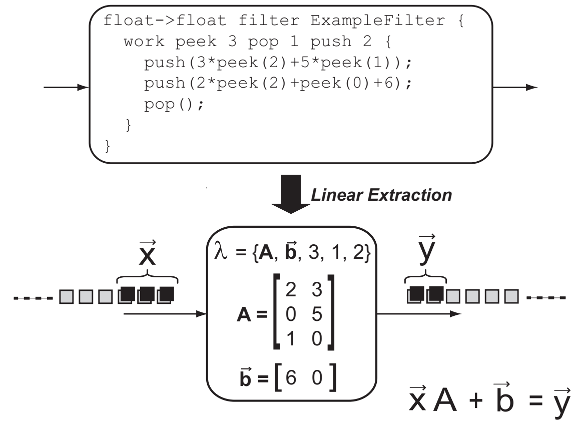


Figure 7: Representation of a linear node.

1.2 StreamIt

StreamIt is a language and compiler for high-performance signal processing [6, 7, 19]. 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 *prework* function is defined.

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 the three predefined structures shown in 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 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.

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 [19]. In particular, the linear analyses described in this paper rely heavily on the structure of StreamIt since they focus on each hierarchical primitive rather than dealing with the complexity of arbitrary graphs.

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* $\vec{y} = \vec{x}A + \vec{b}$ *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 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 7 shows a concrete example of a work function and its corresponding linear node.

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

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. $(map, A, \vec{b}, popcount, pushcount) \leftarrow$
   $\textbf{Extract}(F_{work}, (\lambda x.\perp), A_0, \vec{b}_0, 0, 0)$
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 $\vec{v}$

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

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*
$A[*, \text{Push} - 1 - pushcount] \leftarrow \vec{v}$
$\vec{b}[\text{Push} - 1 - pushcount] \leftarrow c$
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**
$(map, A, \vec{b}, popcount, pushcount) \leftarrow$
$\textbf{Extract}(code, map, A, \vec{b}, popcount, pushcount)$

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

the next page, 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 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 8 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 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 $\langle \textbf{BuildCoeff}(popcount), 0\rangle$, 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 (*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

[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.

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

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

Figure 8: Data types for the extraction analysis.

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, $c_1 \sqcup c_2 = \top$. 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: $\mathit{map}_1 \sqcup \mathit{map}_2 = \mathit{map}'$, where $\mathit{map}'.\mathit{get}(x) = \mathit{map}_1.\mathit{get}(x) \sqcup \mathit{map}_2.\mathit{get}(x)$.

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 $\top$.

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 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.

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 [10].

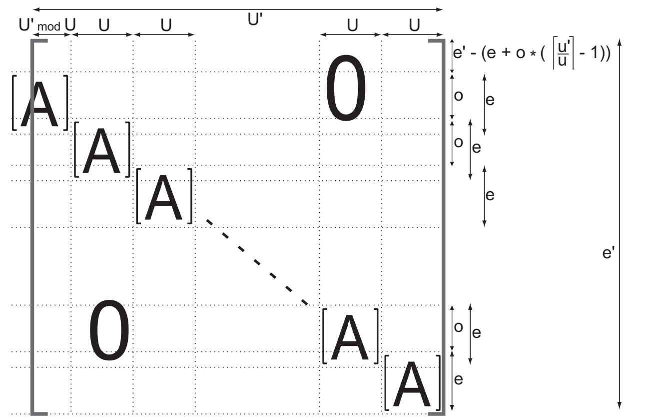


Figure 9: Expanding a linear node to rates (e', o', u').

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.

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* $\vec{\mathbf{b}}'$ *is a* u'*-element row vector:*

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

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

$$\mathbf{A}' = \sum_{m=0}^{\lceil u'/u \rceil} \mathit{shift}(u'-u-m*u, e'-e-m*o)$$

$$\vec{\mathbf{b}}'[j] = \vec{b}[u-1-(u'-1-j) \bmod u]$$

The intuition behind linear expansion is straightforward (see Figure 9). 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.

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* $\mathbf{pipeline}(\lambda_1, \lambda_2) = \{\mathbf{A}', \vec{\mathbf{b}}', \mathbf{e}', \mathbf{o}', \mathbf{u}'\}$ *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.

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 $u_1 \neq e_2$, we first 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.

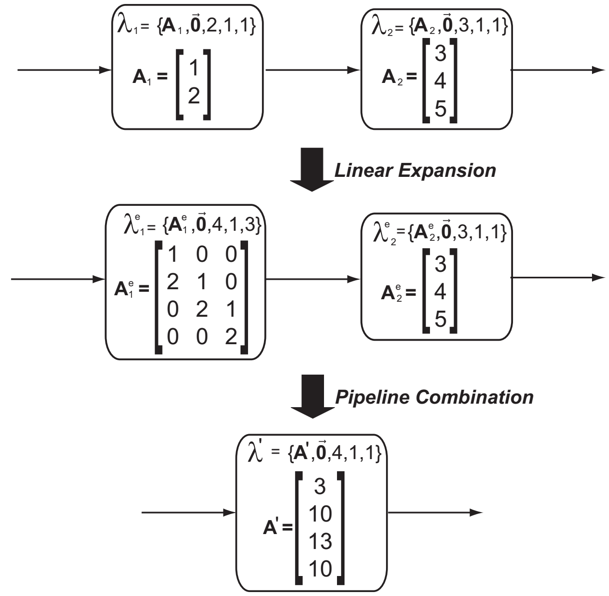


Figure 10: Pipeline combination example.

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 6).

Figure 10 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, we can now form the matrix product of A_1^e and A_2^e, which corresponds to the matrix for the overall linear node.

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 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.

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 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 mathematical terms below.

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* $\mathbf{splitjoin}(s) = \{\mathbf{A}', \vec{\mathbf{b}}', \mathbf{e}', \mathbf{o}', \mathbf{u}'\}$ *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 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; 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.

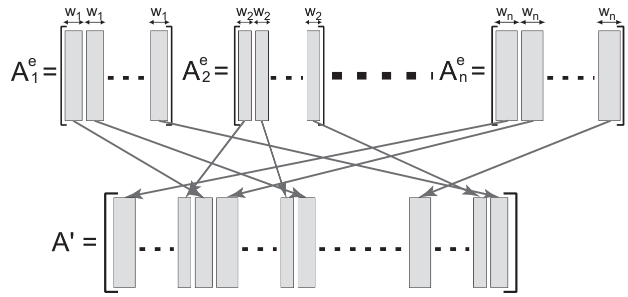


Figure 11: Matrix resulting from combining a splitjoin of rate-matched children.

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 11): 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 12 provides an example of splitjoin combination.

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, 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 roundrobin 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:*

$$vSum_k \;=\; \sum_{i=0}^{k-1} v_i$$
$$vTot \;=\; vSum_n$$

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

$$decimate[k] \text{ is a linear node } \{A, \vec{0}, vTot, vTot, v_k\}$$
$$\text{where } A[i,j] = \begin{cases} 1 \text{ if } i = vTot - vSum_{k+1} + j \\ 0 \text{ otherwise} \end{cases}$$

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

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.

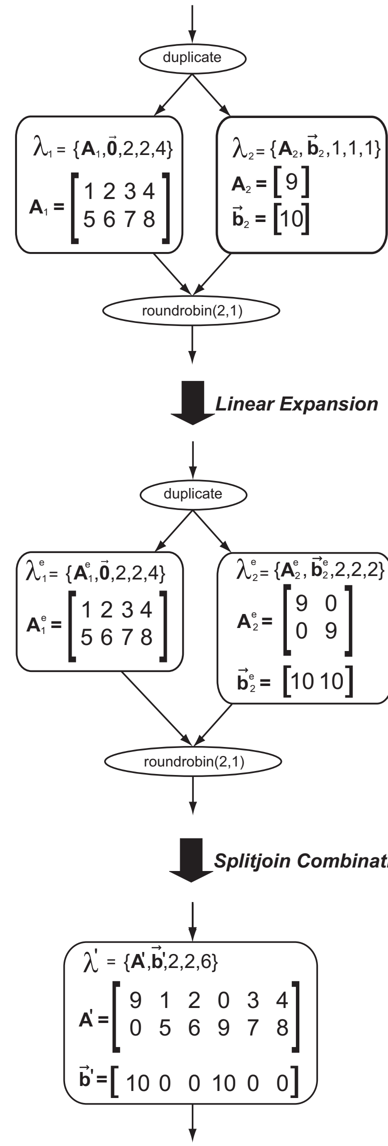


Figure 12: Splitjoin combination example.

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 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 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

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 show that a linear node is equivalent to a set of convolution sums, which can benefit from algorithmic gains if performed in frequency rather than time. We then present an optimized code generation strategy for transforming linear nodes to frequency.

5.1 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:

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

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

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

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

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

Remapping the index i to $i + e - 1$ makes the right hand side equivalent to $\vec{out}[i]$, by Definition 1. Claim 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 [15].

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.

The transformation below 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 Section 7).

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 FreqDecimator(o, u)
}

float → float filter freqDecimator (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()
  }
}
```

5.2 Optimized Frequency Implementation

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 below—is one of several methods that use blocking to efficiently convolve a short filter with a large amount of input [15].

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 in the frequency domain:*

```
float → float pipeline optimizedFreq (A, b⃗, e, o, u) {
  add float → float filter {
    N ← 2^⌈lg(2e)⌉
    m ← N − 2e + 1
    partials⃗ ← new array[0 . . . e − 2, 0 . . . u − 1]
    r ← m + e − 1

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

    prework peek r pop r push u ∗ m {
      x⃗ ← pop(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])
        partials⃗[∗, j] ← y⃗[m + e − 1 . . . m + 2e − 3, j]
      }
      for i = 0 to m − 1
        for j = 0 to u − 1
          push(y⃗[i + e − 1, j] + b⃗[j])
    }

    work peek r pop r push u ∗ r {
      x⃗ ← pop(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 e − 1
        for j = 0 to u − 1 {
          push(y⃗[i, j] + partials⃗[i, j])
          partials⃗[i, j] ← y⃗[m + e − 1 + i, j]
        }
      for i = 0 to m − 1
        for j = 0 to u − 1
          push(y⃗[i + e − 1, j] + b⃗[j])
    }
  }
  add Decimator(o, u)  // see Transformation 5
}
```

5.3 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.

6. 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 present an effective selection algorithm that relies on dynamic programming to quickly explore a large space of configurations.

6.1 The Selection Problem

First, the selection of optimizations for a given stream graph can have a large impact on performance. As alluded to in Section 4, 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. This occurs in our Vocoder and FMRadio benchmarks, in which the selection algorithm improves performance by choosing plain linear combination over frequency translation.

Second, the arrangement of the stream graph can dictate which transformations are possible to apply. 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. For example, in our TargetDetect benchmark, we cut a splitjoin horizontally and collapse the top piece before converting to the frequency domain, thereby amortizing the cost of the FFT on the input items. In our Vocoder benchmark, we cut a splitjoin vertically in order to separate the non-linear filters on the left and combine all of the filters on the right into a single linear node.

6.2 Dynamic Programming Solution

Our optimization selection algorithm, shown in Figures 13 and 14, 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 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 below. 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.

The key to the algorithm's efficiency is the manner in which it considers refactoring the children of hierarchical nodes. Instead of considering arbitrary arrangements of the

```
// 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 13: Type declarations for code in Figure 14.

stream graph, it considers only *rectangular* partitions, in which a given splitjoin is divided into two child splitjoins by either a horizontal or vertical cut. This approach works well in practice, because splitjoins often have symmetrical child streams in which linear components can be separated by a straight line. Moreover, as the child splitjoins are decomposed and evaluated, there are overlapping sub-problems that enable us to search the space of child configurations in polynomial time, using dynamic programming.

A subtlety in the pseudocode of Figure 14 is with regards to the translation from a StreamIt graph to a set of hierarchical *Stream* objects. In the pseudocode, each `Stream` corresponds only to a pipeline; adjacent splitjoin objects are wrapped in a pipeline and their children are considered as direct children of the pipeline. This enables different parts of neighboring splitjoins to be combined. However, it implies that a `Stream` might have a different width at different points (since neighboring splitjoins could have differing widths); this necessitates the addition of the `width` array to the algorithm.

6.3 Cost Functions

The pseudocode in Figure 14 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 (such as the the `FIRS` instruction in the TMS320C54x [18]), then this would effect the cost for certain dimensions of matrices; similarly, if a matrix multiplication algorithm is 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 (let $\lambda = (A, \vec{b}, e, o, u)$ be the linear node corresponding to stream s):

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

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

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

```
// 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)
```

```
// 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
```

```
// 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 14: Algorithm for optimization selection.

	Originally				After Optimizations		
Benchmark	**Filters (linear)**	**Pipelines (linear)**	**SplitJoins (linear)**	**Average vector size**	**Filters**	**Pipelines**	**SplitJoins**
FIR	3 (1)	1(0)	0 (0)	256	3	1	0
RateConvert	5 (3)	2 (1)	0 (0)	102	4	1	0
TargetDetect	10 (4)	6 (0)	1 (0)	300	7	1	1
FMRadio	26 (22)	11 (9)	2 (2)	40	5	1	0
Radar	76 (60)	17 (0)	2 (0)	4412	38	17	2
FilterBank	27 (24)	17 (9)	4 (4)	52	3	1	0
Vocoder	17 (13)	10 (8)	2 (1)	60	6	2	1
Oversampler	10 (8)	1 (1)	0 (0)	33	3	1	0
DToA	14 (10)	3 (1)	0 (0)	52	7	2	0

Table 1: Characteristics of benchmarks before and after running optimizations with automatic selection.

For the direct cost, we consider the cost to be infinite for splitjoins with roundrobin splitters. This is because the splitjoin combination does not eliminate any arithmetic operations, and for the roundrobin case it introduces extra overhead (the duplicate case is different because the input items are shared between streams). For other streams, we count the number of multiplies and adds required to perform the matrix multiplication, giving more weight to the multiplies since they are more expensive. We do not count the zero entries of the arrays, since our matrix multiply routines take advantage of the sparsity of the matrix. We add $185 + 2 * u$ to all costs as a measure of the overhead of the node's execution.

For the frequency cost, our cost curve is guided by profiler feedback. The speedup gained by translating to the frequency domain depends on the peek rate of the filter. To obtain an n-fold speedup of the frequency code over the direct code, the filter has to peek $64 * n$ items. Mathematically, this translates to a logarithmic expression for the cost of the frequency node in terms of the number of items e (we actually use the next power of two above e, as that is the size of the FFT). We multiply the above cost by the number of items pushed, add the constant offset of $185 + 2 * u$ for a node, and then multiply by $max(o, 1)$ because only one out of o items represents a valid output from the frequency domain. Finally, we add $dec(s)$, the cost of the decimator for the frequency node. We estimate an extra cost of 2 per push operation in the decimator, as it must read from the input tape. The other pop operations in the decimator are free, because they can be performed as a single adjustment of the tape position.

Of course, both cost functions are undefined if s is nonlinear (*i.e.*, if there is no corresponding λ_s). If this is the case, then the selection algorithm assigns an infinite cost.

7. RESULTS

We have completed a fully automatic implementation of the linear combination, frequency replacement, and optimization selection algorithms described in the previous sections. The implementation is part of the StreamIt compiler, and works for both the uniprocessor and Raw [13] backends. 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 in the frequency domain. To implement the necessary basis conversions, we use FFTW [5], 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. This works according to the algorithm in Section 6.

Below we describe experiments and results that demonstrate substantial performance improvements due to our methods. For full results, stream graphs, and source code, please visit `http://cag.lcs.mit.edu/linear/`.

7.1 Measurement Methodology

Our measurement platform is a Dual Intel P4 Xeon 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.

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) **FIR**, a single 256 point low pass FIR filter; 2) **RateConvert**, an audio down sampler that converts the sampling rate by a non-integral factor ($\frac{2}{3}$); 3) **TargetDetect**, four matched filters in parallel with threshold target detection; 4) **FMRadio**, an FM software radio with equalizer; 5) **Radar**, the core functionality in modern radar signal processors, based on a system from the Polymorphic Computing Architecture [12]; 6) **FilterBank**, a multi-rate signal decomposition processing block common in communications and image processing; 7) **Vocoder**, a channel voice coder, commonly used for speech analysis and compression; 8) **Oversampler**, a 16x oversampler, a function found in many CD players, 9) **DToA**, an audio post-processing stage prior to a 1-bit D/A converter with an oversampler and a first order noise shaper.

Some statistics on our benchmarks appear in Table 1.

7.2 Performance

One interesting aspect of our optimizations is that they eliminate floating point operations (FLOPS) from the program, as shown in Figure 15. The removal of FLOPS represents fundamental computation savings that is indepen-

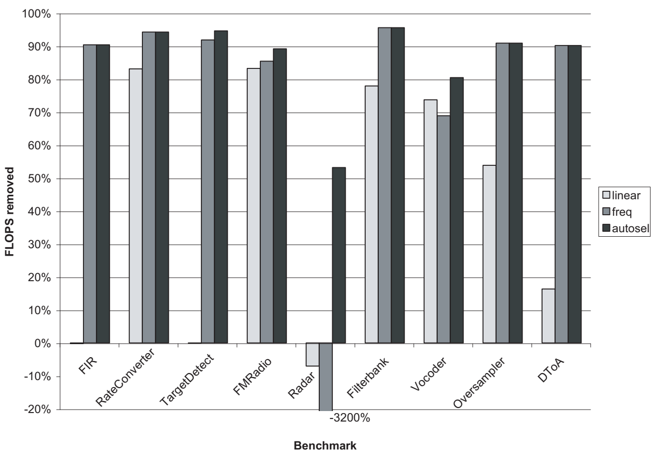


Figure 15: Elimination of floating point operations by maximal linear replacement, maximal frequency replacement, and automatic optimization selection.

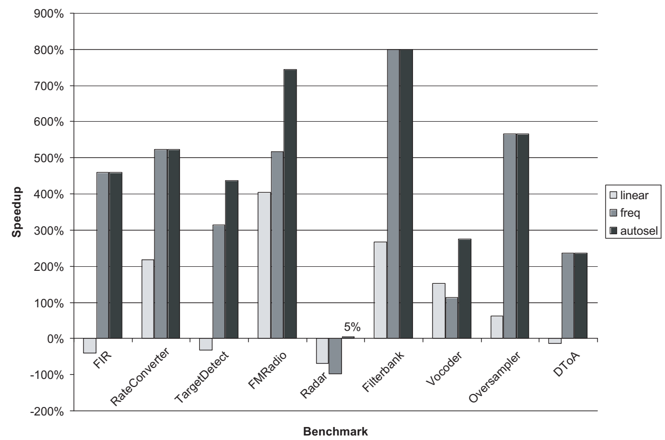


Figure 16: Execution speedup for maximal linear replacement, maximal frequency replacement, and automatic optimization selection.

dent of the streaming runtime system and other (FLOPS-preserving) optimizations in the compiler. As shown in the figure, 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 for 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 in Radar, 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 speedups are shown in Figure 16. 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 FMRadio, Radar, and Vocoder, the automatic selection algorithm obtains its speedup 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 [20], 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. In the future, we plan to further investigate how best to perform the matrix multiply.

Perhaps the most interesting benchmark is Radar[2]. Maximal linear and frequency replacement lead to abysmal performance on 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.

8. 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[8, 9, 4]. Using a feedback-directed search process, DSP transforms are optimized for the underlying architecture. The input language to SPIRAL is SPL[22, 21], 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 [20] 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 dynam-

[2] This is the same Radar application as in [7], 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.

ically 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 [3]. 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 [14].

Karr [11] and Cousot and Halbwachs [2] 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 [17] for a survey. Also note that the "linear data flow analysis" of Ryan [16] is completely unrelated to our work; it aims to do program analysis in linear time.

9. 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 our linear dataflow analysis, linear combination, frequency translation and automated optimization selection we can improve execution speed by an average factor of 450%.

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 paper 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.

10. ACKNOWLEDGEMENTS

We are very grateful to David Maze, Michal Karczmarek, Jasper Lin, and Michael Gordon for extensive support with the StreamIt infrastructure, and to Alex Salcianu for his helpful comments. The StreamIt project is supported by DARPA, NSF, and the MIT Oxygen Alliance.

11. REFERENCES

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

[2] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.

[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. In *PLDI*, 1999.

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

[7] 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.

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

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

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

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

[12] J. Lebak. Polymorphous Computing Architecture (PCA) Example Applications and Description. External Report, Lincoln Laboratory, Mass. Inst. of Technology, 2001.

[13] 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.

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

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

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

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

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

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

[20] 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.

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

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