Recognizing Linear Data Transformations in StreamIt

Andrew Lamb

August 14, 2002

1 Introduction/Overview

In general, high level computer programming languages abstract away non-critical details from the programmer that are not focused on solving the problem at hand. Low level languages force programmers to deal with details about the complex hardware that their programs are run on to achieve the fastest possible speed. Optimizing compilers were invented in the late 1950s as a way to automate the process of code generation and detail management while keeping much of the same speed of hand coded assembly. Therefore, the optimizations that are implemented compilers need to be as clever as possible so as to produce the best possible code. The most limiting factor faced by optimizations is that they need to be able to figure what a program is doing, which is not a trivial problem. Determining what is happening in a program is accomplished by a process called dataflow analysis[1].

Languages like C and FORTRAN abstracted away details about Von Neumann machines (register based, monolithic memories, program counter, stack pointers, etc.) for the programmer. Languages like Java and C# abstract away even more details such as explicit memory management and dynamic library use. However, even these high level languages of today still target the Von Neumann machines of yesterday. As computer architectures become increasingly complicated, with multiple processors, multiple memories and often explicit communication that must be scheduled by the programmer, the Von Neumann architecture no longer adequately describes the computational facilities available. Therefore, newer even higher level languages are needed which are appropriate for this new class of architectures. The COMMIT group at LCS is currently working on such a new high level language called StreamIt that provides a new level of abstraction. StreamIt is a programming language for applications that operate on streams of data. By focusing our attention on this particular domain of programs, our compiler knows more about programs that are written. We therefore have many new opportunities for optimization.

One large and important subclass of streaming programs are programs which do Digital Signal Processing (DSP). The goal of my MEng thesis is to lay the foundation for automatic performance improving signal processing transformations within StreamIt. I will design, implement and test a compiler pass which determines filters which compute linear functions. Using this information about linear filters, I hope to leverage existing research done by the DSP community in the area of fast signal processing implementations to programs that are written in StreamIt. The less time programmers have to spend thinking about the details of implementation, means the more time that they can spend writing new and useful applications, resulting in an increase in programmer productivity.

2 StreamIt

StreamIt[12, 13, 15, 9] is a novel language for representing streaming applications. Streaming applications operate on a (potentially) infinite amount of data. Examples of streaming applications include media processing (MP3 audio, video), digital communication (cell phones, wireless networking), digital signal processing (FFT, DCT, image processing) and any other application that needs to operate on a large amount of data.

StreamIt also attempts to provide a common, high level machine language for fabrics of interconnected microprocessors such as the RAW[14, 11] microprocessor being developed at MIT or the CELL[10] microprocessor being developed by IBM, Sony and Toshiba. Much like the Von Neumann machine became the prototypical computation model for 30 years, computational fabrics stand poised to be the prototypical computation model for the next 30 years. RAW consists of 16 processors (called tiles) in a 4x4 grid on a single monolithic chip. The processors are modified MIPS RISC R10K cores and the communication between cores is achieved over a static network or a dynamic network. The movement of data around the networks is explicitly controlled by the programmer.

The major challenge of writing code for RAW is designing the network communication to take effective use of all of the processors on the chip. To write a program for the raw chip, first the programmer determines what computation to put on what chip. Then he or she writes code in C for each tile that implements that computation. Then the programmer has to set up the communication between the various tiles and connect up the processor cores in the appropriate fashion.

StreamIt automates the entire process partitioning the computation among the different processors (to take advantage of parallelization) and arranging communication as well as managing buffers.

A StreamIt program is a structured stream graph, composed of hierarchically arranged computation units. Filters are the basic building block of a stream graph. They process data from a single stream and produce data that leaves the filter as a single stream.

3 Digital Signal Processing and Linear Transformations

One important class of applications for streaming computation is digital signal processing (DSP). Most of the DSP literature is focused on systems which are linear and time invariant (LTI) because of their nice mathematical properties. Almost all real systems are not actually linear or time invariant. However, most real systems can be very accurately modeled as an LTI system over specific ranges of inputs and outputs, and many DSP algorithms take advantage of this fact. In addition, many useful signal processing operations compute linear functions of their inputs. It is these Linear Filters that I will be trying to identify.

3.0.1 Linearity, Time Invariance, and Linear Functions

Linear Suppose a system, H, operate on some input sequence $x[n]$ and produces some output sequence $y[n]$, $y[n] = H(x[n])$. If H is linear, then the following statement is true: applies: $H(a * x[n] + b) = a * H(x[n]) + b$ where a and b are constants.

Time-invariant Time invariance means that shifting the input, $x[n]$ by a certain amount, k, will shift the output by k as well: if $H(x[n]) = y[n]$ and H is time-invariant then $H(x[n-k]) = y[n-k]$.

Linear Functions If a filter computes a linear function of its inputs, each output element of a filter can be computed using a weighted combination the inputs and possibly an offset. A filter that computes a linear function can also be represented as a matrix operation on an input vector to produce an output vector. The values of the elements in the matrix correspond to the weights of the inputs used to produce various outputs. For instance, if the input to a function is $x[n]$ and the output is $y[n]$, if it is possible to find a matrix A and a vector b such that $y[n] = Ax[n] + b$ then we characterize that filter as computing a linear function.

By letting our representation expand to include all filters that can be described by $y[n] = Ax[n] + b$ where b is a constant, we can described all filters that compute linear functions.

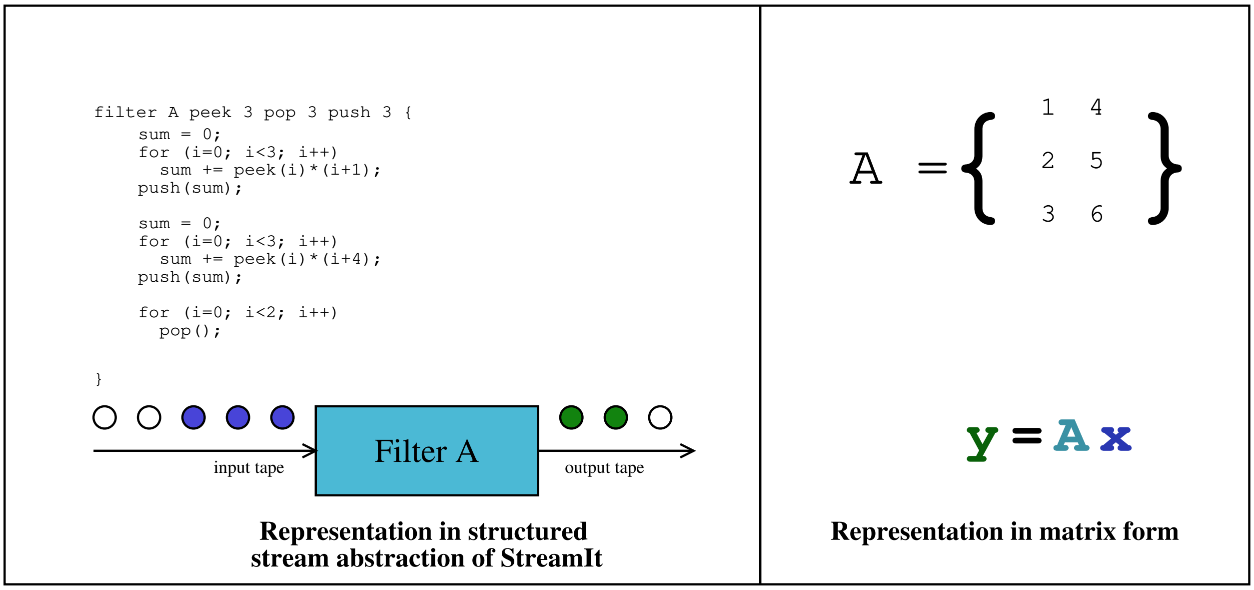

Figure 1: The correspondence between a linear filter and a matrix. The left half of the figure shows example code for Filter A which computes a linear function of its input. The input to filter A is shown as the three blue items on its input tape (since it pops and peeks at 3 items). The output of filter A is shows as the two green items on its output tape. The right half of the figure shows that the same filter can be though of as a matrix multiplication on an input vector of three elements which yields an output vector of two elements.

4 StreamIt Filters as Matrices

Figure 1 shows an example filter in the StreamIt language, and its corresponding matrix representation as a way to illustrate the correspondence between filters which compute linear functions and matrices operating on an input vector to produce an output vector.

5 Linear Analysis and Optimization

5.1 Introduction

Many useful compiler optimizations rely on specific data about how the program's execution can flow. Analysis techniques that analyze the flow of data through the program are called dataflow analysis. Study of the various canonical optimizations and their associated dataflow techniques (eg constant propagation, common subexpression elimination, dead code elimination) has yielded a common theoretical framework for framing various compiler analyses. The theoretical framework is well understood and presented in undergraduate level textbooks on compiler design[1, 2].

The central purpose of my thesis will be to develop the dataflow equations in the standard dataflow framework and implement a linear analysis pass of the compiler. I will then implement one or more optimizations using the dataflow information to demonstrate the usefulness of this data.

By characterizing systems as LTI, powerful mathematical results (such as Fourier analysis and Fourier synthesis) can be applied. Because a wide range of physical systems can be modeled with LTI systems, LTI systems have been the focus of the DSP community for some time. In particular, because they are so widely studied and used, it is likely that a large subset of all StreamIt programs will contain sub parts that implement LTI systems.

Because we expect a large class of StreamIt applications to contain LIT filters, if we can both detect that a particular filter is LTI and make good use of that information to optimize the program (by leveraging the plethora of existing work in the field) we can reach our goal of freeing programmers from having to write optimized implementations of their algorithms.

5.2 Proposed Dataflow Analysis

To unify the talk about signal processing and the StreamIt language, we identify filter F with system H. If H operates on some vector x with dimension q, and produces a vector y with dimension r, then F has $peek(q)$ and $push(r)$. The goal of our dataflow analysis for each filter that computes a linear function, to identify a matrix A and a vector b such that $y = Ax + b$ (that is the elements of y are weighted combinations of the the elements of x, possibly with a constant offset.) It is interesting to note that given the vector sizes above, we can see that A must have dimensions q x r and b must be a vector of size r.

I will compute A and b using traditional dataflow techniques, based on the prototypical *gen* and *kill* sets of formal dataflow treatments (see chapter 10 of [1], or chapter 17 of [2]). The particular analysis to be performed is very much like the analysis for constant propagation. Instead of propagating mappings of variables to constant values, we will propagate mappings of variables to weighted combinations of the inputs. By defining the appropriate confluence operator (in this case matrix addition) we will be able to determine if all outputs can be derived with an appropriately weighted combinations of the inputs plus some constant.

If we can derive the output with the right combination of the inputs, we can then express the filter as a matrix A and a vector b.

5.2.1 Proposed Optimizations

By recognizing that a filter is LTI and determining A and b for that filter, I hope to be able to implement one or more of the following transformations.

- Generate very fast straight line C code to improve a monolithic filter which computes some linear function. Given a filter in matrix form, there are already systems in place to generate this fast C code [17, 7, 8, 16].

- Convert a computation in time to a computation in frequency using a transformation like $x \to (A, b) \to y$ to $x \to FFT \to (A') \to IFFT \to y$. Even though this looks like a less efficient way to compute y, in some cases it requires much less computation. In addition, this removes the offset vector b and therefore might allow me to take advantage of the first optimization in cases where it otherwise would not apply.

- Compose the work done by several LTI filters automatically, allowing for intra-filter optimizations that would not be possible given the initial arrangement. This will perform filter fusion also (which is already implemented for general filters in the compiler), but possibly generating more intelligent code by using [7] to take advantage of redundant computation.

- Split up the work done by one monolithic filter by breaking A and b up into blocks and computing y in parallel fashion.

5.3 Example: Combining Linear Filters in a Pipeline

Figure 2 shows how linear filter information could be used to implement pipeline fusion of linear filters. Pipeline fusion is the process by where a pipeline of filters is fused together to form a single monolithic filter which does the computation of both filters in its main body. In Figure 2 filters A and B are combined into a single filter. Filter A pops, pushes and peeks a, b and c data items respectively. Filter B pops, pushes

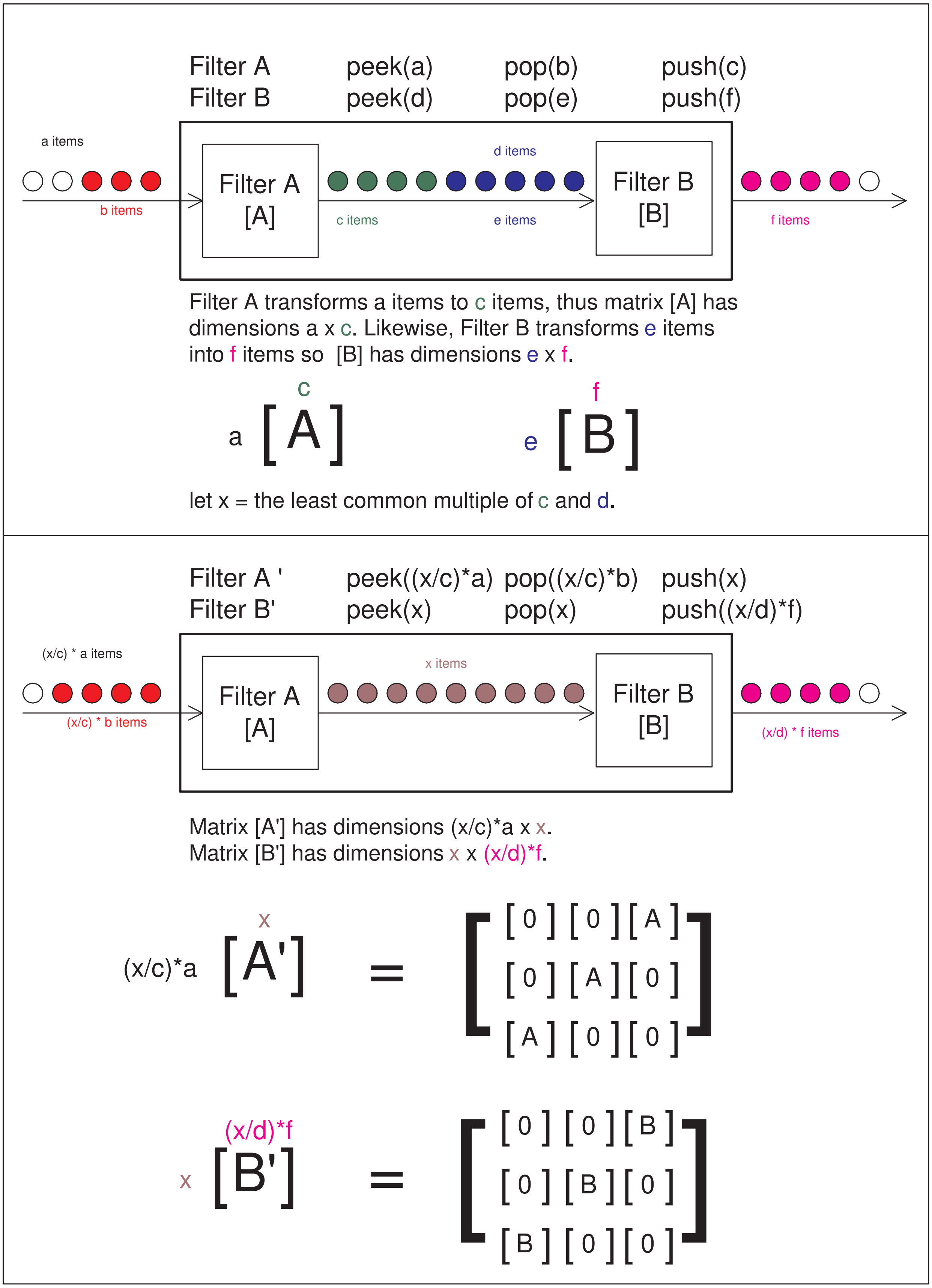

Figure 2: Combining pipelines of consecutive filters with equal peek/pop rates

and peeks d, e and f data items respectively. For this case, we need the fact that peek(B)=pop(B). If filter A can be represented by matrix A and filter B can be represented by matrix B then we can then combine the two filters by first expanding them to filters A' and B' which can be represented by matrices A' and B' which can be multiplied together. The new filter then is represented by $A' * B'$, which can then be fed to other stages if necessary.

6 Related Work

One of the most important linear computations that is done by filters is that of taking the Fast Fourier Transform (FFT). FFTW[4, 5, 6] produces very fast optimized FFT implementations tuned for specific hardware platforms. It is very specific for generating fast code that performs the FFT and therefore FFTW is not of immediate use in more a more general class of programs.

The SPL language[17], a part of the SPIRAL[7] signal processing package, can be used to express DSP filters as matrices and provides a way to automatically generate fast code to implement DSP algorithms expressed in matrix form. It also supports composing multiple filters together (eg FFT followed by a DCT). It also appears to support limited matrix decomposition into the product of sparse matrices which can lead to fast implementations. Theoretical research into the area of automatically factoring matrices has also been explored[16]. See [8] for an example of generating a fast FFT implementation using the SPL language.

There have been several attempts in the past to create signal processing design environments that ease the job of finding more efficient implementations of algorithms. For instance, ADE[3] searches for efficient implementations of signal processing algorithms by using various techniques to limit the combinatorial growth of implementations while searching for alternative designs. ADE's goal was to create an integrated environment for the design of signal processing algorithms, and does not focus on implementation. It can help you determine what the best implementation of a particular algorithm is in the signal processing domain, but it does not generate the actual implementation.

References

[1] R. S. Alfred V. Aho and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, second edition, 1988.

[2] A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, second edition, 1998.

[3] M. M. Covell. *An Algorithm Design Environment for Signal Processing.* Number RLE Technical Report No. 549. Research Lab for Electronics, Massachusetts Institute of Technology, 1989. PhD thesis, Alan V. Oppenheim supervisor.

[4] M. Frigo. A Fast Fourier Transform Compiler. In *PLDI'99 — Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999.

[5] M. Frigo and S. Johnson. Home page of fftw. http://www.fftw.org.

[6] M. Frigo and S. Johnson. Fftw: An adaptive software architecture for the fft, 1998.

[7] J. Johnson, R. W. Johnson, D. A. Padua, and J. Xiong. Home page of the spiral project. http://www.ece.cmu.edu/ spiral/.

[8] J. Johnson, R. W. Johnson, D. A. Padua, and J. Xiong. Searching for the best FFT formulas with the SPL compiler. *Lecture Notes in Computer Science*, 2017:112–??, 2001.

[9] W. T. Michal. A common machine language for grid-based architectures.

[10] J. G. Spooner. Chip trio allows glimpse into "cell". http://news.com.com/2100-1001-948493.html.

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

[12] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Computational Complexity*, pages 179–196, 2002.

[13] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Ho, M. Brown, and S. Amarasinghe. Streamit: A compiler for streaming applications, 2001.

[14] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[15] M. G. William. A stream compiler for communication-exposed architectures.

[16] J. Xiong. Automatic optimization of dsp algorithms. Technical Report UIUCDCS-R-2001-224, University of Illinois at Urbana-Champaign, 2001.

[17] J. Xiong, J. Johnson, R. W. Johnson, and D. A. Padua. SPL: A language and compiler for DSP algorithms. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 298–308, 2001.