An empirical characterization of stream programs and its implications for language and compiler design

Citation: Thies, William and Saman Anarasinghe. "An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design." PACT '10, Proceedings of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques, September 11-15, 2010, Vienna, Austria.

As Published: http://dx.doi.org/10.1145/1854273.1854319

Publisher: Association for Computing Machinery

Persistent URL: http://hdl.handle.net/1721.1/64763

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design

William Thies
Microsoft Research India
thies@microsoft.com

Saman Amarasinghe
Massachusetts Institute of Technology
saman@mit.edu

ABSTRACT

Stream programs represent an important class of high-performance computations. Defined by their regular processing of sequences of data, stream programs appear most commonly in the context of audio, video, and digital signal processing, though also in networking, encryption, and other areas. In order to develop effective compilation techniques for the streaming domain, it is important to understand the common characteristics of these programs. Prior characterizations of stream programs have examined legacy implementations in C, C++, or FORTRAN, making it difficult to extract the high-level properties of the algorithms.

In this work, we characterize a large set of stream programs that was implemented directly in a stream programming language, allowing new insights into the high-level structure and behavior of the applications. We utilize the StreamIt benchmark suite, consisting of 65 programs and 33,600 lines of code. We characterize the bottlenecks to parallelism, the data reference patterns, the input/output rates, and other properties. The lessons learned have implications for the design of future architectures, languages and compilers for the streaming domain.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*dataflow languages; parallel languages*; D.2.11 [**Software Engineering**]: Software Architectures—*domain-specific architectures*; D.3.4 [**Programming Languages**]: Processors—*compilers; optimization*

General Terms

Languages, Design, Experimentation, Performance

1. INTRODUCTION

The domain of stream programs has attracted interest because it stands at the intersection of recent application and architectural trends. By encompassing applications such as audio, video, and digital signal processing, stream programs are following the expansion of desktop computing to the mobile and embedded space. And by virtue of their structure – a set of independent processing stages that operate on regular sequences of data – stream programs are a natural fit for multicore architectures. The interest in streaming applications has spawned a number of programming languages that target the streaming domain, including StreamIt [53], Brook [9], StreamC/KernelC [28], Cg [36], Baker [12], SPUR [58] and Spidle [14].

In the case of streaming as well as other domains, the design of a programming language is deeply influenced by one's understanding of the application space. Only by characterizing the common-case behaviors can one provide the functionality and performance that is important in practice. Typically this understanding is gleaned via inspection of benchmarks that are representative of the domain. However, in the case of stream programs, existing benchmarks are implemented in von-Neumann languages that often obscure the underlying parallelism and communication patterns. While collections such as MediaBench [32], ALPBench [34], Berkeley Multimedia Workload [46], HandBench [15], MiBench [22], and NetBench [38] (and to a lesser extent SPEC [48], Splash-2 [57], PARSEC [5], and Perfect Club [2]) include several examples of stream programs, they are all written in C, C++, or FORTRAN. Any characterization of these benchmarks thus conflates the issue of understanding the streaming patterns with the more difficult question of extracting those patterns from a low-level description of the algorithm.

In this paper, we present the first characterization of a streaming benchmark suite that was developed directly in a stream programming language. This enables a new understanding of the fundamental properties of stream programs, without struggling to extract those properties from a general-purpose programming model. We utilize the StreamIt language [53], a mature system that is rooted in the synchronous dataflow model [33] and has been used as a research infrastructure for many projects outside of the StreamIt group [13, 23, 24, 25, 26, 35, 41, 45, 47, 55, 56]. Our benchmark suite consists of 65 programs and 33,600 lines of code, which were developed by 22 programmers during the last 8 years. The suite spans many sub-domains of streaming, including video processing, audio processing, signal processing, bit-level processing, and scientific processing. To isolate the study from any performance artifacts of the StreamIt compiler, we limit our attention to architecture-independent program characteristics that are inherent properties of the algorithm rather than its mapping to any given machine. We assess the characteristics along three axes, in each case summarizing the impact of our observations on the design of future programming languages.

Our first axis of inquiry aims to identify the potential barriers to parallelizing stream programs. While stream programs are understood to be rich in data parallelism, there also exist sequential

PACT'10, September 11–15, 2010, Vienna, Austria.

bottlenecks, some of which can be averted via appropriate language design. Our results are as follows:

1. Sliding windows are common. The language must expose the parallelism in sliding windows or they become bottlenecks to the parallel throughput.
2. Startup behaviors often differ from steady-state behaviors. The compiler must recognize such behaviors to avoid a throughput bottleneck.
3. Sequential (stateful) filters are required in one quarter of our benchmarks. Further state could be eliminated via new language constructs, compiler analyses, or programmer interventions.
4. Feedback loops are uncommon in our benchmarks, but represent significant throughput bottlenecks when present.

The second axis concerns the scheduling characteristics of stream programs. Because stream programs execute in parallel, constrained only by the availability of data items on communication channels, all scheduling decisions are made by the compiler and runtime system. Our observations are:

1. Neighboring filters often have matched I/O rates. This reduces the opportunity and impact of advanced scheduling strategies proposed in the literature.
2. It is not useful for filters to divide their atomic execution step into a cycle of smaller steps. However, multiple execution steps are important for scatter/gather stages.
3. Dynamic I/O rates are necessary for expressing several applications.

The third axis of characterization examines the programming style of the benchmarks. These results reflect our experience in observing and coaching over 20 programmers as they developed large benchmarks within the stream programming model:

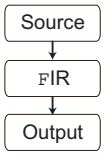
1. It is useful and tractable to write programs using structured streams, in which all modules have a single input and a single output. However, structured streams are occasionally unnatural and, in rare cases, insufficient.
2. Programmers can accidentally introduce unnecessary sequential bottlenecks (mutable state) in filters.

The lists above also serve as a detailed outline for the paper. After describing the StreamIt language (Section 2) and the benchmark suite (Section 3), we present each axis of characterization (Sections 4, 5, and 6) as well as related work (Section 7). We conclude (Section 8) by reflecting on the impact that this characterization would have had on our own direction, if it were available at the start of the StreamIt project. We hope that this paper has similar relevance for future design and evaluation of architectures, languages, and compilers for the streaming domain.

2. THE STREAMIT LANGUAGE

StreamIt is an architecture-independent language for high-performance stream programming [53]. The compiler is publicly available [49] and includes backends for multicore architectures, clusters of workstations, and the MIT Raw architecture.

The model of computation in StreamIt is grounded in (but not limited to) synchronous dataflow [33]. In this model, the programmer implements independent actors, or *filters*, which translate data items from input channels to output channels. Filters are composed into graphs that represent the overall computation. The key property of synchronous dataflow is that the number of items consumed and produced during each execution of a filter is known at compile time, allowing the compiler to perform static scheduling and optimization. The StreamIt language also allows filters to declare a dynamic data rate, which requires the support of a runtime system.



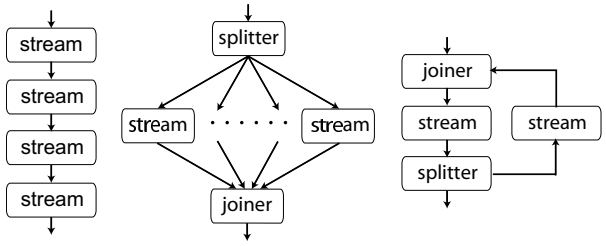

```
float->float pipeline Main {
  add Source();  // code for Source not shown
  add FIR();
  add Output();  // code for Output not shown
}

float->float filter FIR (float sampRate, int N) {
    float[N] weights;

    init {
        weights = calcImpulseResponse(sampRate, N);
    }

    prework push N-1 pop 0 peek N {
        for (int i=1; i<N; i++) {
            push(doFIR(i));
        }
    }

    work push 1 pop 1 peek N {
        push(doFIR(N));
        pop();
    }

    float doFIR(int k) {
        float val = 0;
        for (int i=0; i<k; i++) {
            val += weights[i] * peek(k-i-1);
        }
        return val;
    }

    handler changeWeights(float[N] newWeights) {
        weights = newWeights;
    }
}
```

Figure 1: Example StreamIt program with FIR filter.

An example StreamIt program appears in Figure 1. It is based on an FIR filter, which contains three stages of execution. The most important is the *work* function, which represents the steady-state execution step and is called repeatedly by the runtime system. Within the work function, a filter may *peek* at a given element on the input tape, *pop* an item off the input tape, or *push* an item to the output tape. The total number of items peeked, popped, and pushed are declared as part of the work function. Note that if the peek rate exceeds the pop rate, it represents a sliding window computation in which some input elements are accessed across multiple invocations of the filter.

In addition to the work function, a filter may declare an *init* function to initialize internal data structures, as well as a *prework* function to perform specialized processing of data items prior to the steady state. The prework function is needed in cases where the initial processing has a different input or output rate than the steady-state processing.

In addition to supporting steady-state flows of data, the StreamIt language also provides a mechanism for sending out-of-band con-



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

Figure 2: Hierarchical stream structures supported by StreamIt.

trol information between actors. Termed *teleport messaging*, this feature allows[1] filters to deliver messages that are timed with respect to the data items in the stream, even if they are not embedded in the stream itself [52]. For example, a distant filter could invoke the *changeWeights* handler in the FIR filter in order to adjust the weights in the filter. All message senders and message latencies are known at compile time to avoid non-deterministic outcomes. Due to space limitations, we refer the reader to an accompanying report for a characterization of teleport messaging [52].

As depicted in Figure 2, StreamIt provides three hierarchical primitives for composing filters into stream graphs. A *pipeline* represents a sequential composition of streams, in which the output of one stream feeds into the input of the next. A *splitjoin* represents a parallel set of streams, which diverge from a common *splitter* and converge to a common *joiner*. The types of splitters and joiners are predefined by the StreamIt language; they encompass duplication and weighted round-robin behaviors. Finally, a *feedbackloop* represents a cycle in the stream graph.

Because pipelines, splitjoins, and feedbackloops are all single-input and single-output, they can be hierarchically composed. By analogy to structured control flow, we designate these primitives as *structured streams*.

3. BENCHMARK SUITE

An overview of the StreamIt benchmark suite appears in Table 1. At the time of this writing, the suite consists of 65 programs, including 30 realistic applications, 4 graphics rendering pipelines, 23 libraries and kernels, and 8 sorting routines. Benchmarks range in size from 31 lines (VectAdd) to over 4,000 lines (MPEG2 encoder), with a total of 33,600 non-comment, non-blank lines in the suite[2] Twenty two people contributed to the suite, including 6 from outside our group. Almost all benchmarks were based on a reference version in C or MATLAB, which we obtained from real-world sources such as DARPA (GMTI, FAT, SAR), industrial partners (3GPP, OFDM), or international standards (MPEG2, MP3, JPEG, HDTV, etc.) The benchmark suite is broadly representative of the streaming domain, as it spans video processing (MPEG, HDTV, etc.), audio processing (MP3, FMRadio, etc.), signal processing (GMTI, SAR, etc.), bit-level processing (DES, Serpent, etc.) and scientific processing (MatMul, Cholesky, etc.)

Graphical depictions of the stream graphs for each benchmark can be found in an accompanying report [52]. A subset of the benchmarks have also been released on the StreamIt website [49].

[1]While fully supported by the StreamIt language, teleport messaging and dynamic rates have only basic, unoptimized support under our current compiler infrastructure.

[2]Counting commented lines (8,100) and blank lines (7,300), the benchmark suite comes to 49,000 lines.

At the time of this writing, some of the larger benchmarks (MPEG2, GMTI, Mosaic, FAT, HDTV) are not fully supported by the compiler. However, their functional correctness has been verified in the Java runtime for the StreamIt language.

It is important to recognize that most of the benchmarks are parameterized, and we study only one assignment of those parameters in our quantitative evaluation. Table 1 details the parameterization of the StreamIt benchmarks. In two-thirds (44) of the benchmarks, the parameters affect the structure of the stream graph, often by influencing the length of pipelines, the width of splitjoins, the depth of recursion hierarchies, or the absence or presence of given filters. The same number of benchmarks contain parameters that affect the I/O rates of filters (e.g., the length of an FIR filter), but do not necessarily affect the structure of the graph. Changes to the I/O rates also imply changes to the schedule and possibly the balance of work across filters. In selecting values for these parameters, our primary goal was to faithfully represent a real-life application of the algorithm. In some cases we also decreased the sizes of the parameters (e.g., sorting 16 elements at a time) to improve the comprehensibility of the stream graph. For benchmarking purposes, researchers may wish to scale up the parameters to yield larger graphs, or to vary the ratio between parameters to obtain graphs of varying shapes and work distributions.

More detailed properties of the filters and streams within each benchmark are given in Table 2. In terms of size, benchmarks declare (on average) 12 filter types and instantiate them 65 times in the stream graph. GMTI contains the most filters, with 95 static types and 1,111 dynamic instances; it also contains 1,757 instances of the Identity filter, to assist with data reordering.

4. THROUGHPUT BOTTLENECKS

As stream programs are a prime target for parallelization, it is important to understand the real and artificial bottlenecks to achieving high parallel throughput. In this section, we use the term "stateful" to refer to filters that retain mutable state from one execution to the next; filters containing only read-only state are classified as "stateless". Stateless filters are amenable to data parallelism, as they can be replicated any number of times to work on different parts of the input stream [20]. However, stateful filters must be run in a serial fashion, as there is a dependence from one iteration to the next. While separate stateful filters can be run in a task-parallel or pipeline-parallel mode, the serial nature of each individual filter represents an eventual bottleneck to the parallel computation.

4.1 Sliding windows are common. The language must expose the parallelism in sliding windows or they become bottlenecks to the parallel throughput.

Twenty one benchmarks – and 57% of the realistic applications – contain at least one filter that peeks. (That is, these filters declare a peek rate larger than their pop rate, and thus implement a sliding window computation; such filters re-read the same input items on successive invocations of the filter.) In von-Neumann languages, peeked items are typically stored as internal states to the filter and appear as serializing dependences across iterations, though in StreamIt, the peek primitive avoids storing the items and allows the parallelism between iterations to be exposed. Benchmarks contain up to 4 filter types that peek; in programs with any peeking, an average of 10 peeking filters are instantiated.

While peeking is used for many purposes, there are a few common patterns. The most common is that of an FIR filter, where a filter peeks at N items, pops one item from the input, and pushes a weighted sum to the output. FIR filters account for slightly less than half (15 out of 34) of the peeking filter declarations. They are

Benchmark	Description	Parameters and default values	Lines of Code[1]
Realistic Apps (30):			
MPEG2 encoder	MPEG2 video encoder (Drake, 2006)	image size (320x240)	4041
MPEG2 decoder	MPEG2 vivdeo decoder (Drake, 2006)	image size (320x240)	3961
GMTI	Ground moving target indicator	over 50 parameters	2707
Mosaic	Mosaic imaging with RANSAC algorithm (Aziz, 2007)	frame size (320x240)	2367
MP3 subset	MP3 decoder (excluding parsing + huffman coding)	--	1209
MPD	Median pulse compression doppler radar (Johnsson et al., 2005)	FFT size (32); rows (104); cols (32)	1027
JPEG decoder	JPEG decoder	image size (640x480)	1021
JPEG transcoder	JPEG transcoder (decode, then re-encode at higher compression)	image size (640x480)	978
FAT	Feature-aided tracker[2]	15 parameters, mostly matrix dimensions	865
HDTV	HDTV encoder/decoder[2]	trellis encoders (12); interleave depth (5)	845
H264 subset	16x16 intra-prediction stage of H264 encoding	image size (352x288)	788
SAR	Synthetic aperture radar	over 30 parameters	698
GSM	GSM decoder	--	691
802.11a	802.11a transmitter	--	690
DES	DES encryption	number of rounds (16)	567
Serpent	Serpent encryption	number of rounds (32); length of text (128)	550
Vocoder	Phase vocoder, offers independent control over pitch and speed (Seneff, 1980)	pitch & speed adjustments, window sizes	513
RayTracer	Raytracer (ported from Intel's)	no parameters, though data read from file	407
3GPP	3GPP radio access protocol - physical layer	matrix dimensions Q, W, N, K (2, 2, 4, 8)	387
Radar (coarse)	Radar array front end (coarse-grained filters, equivalent functionality)	channels (12); beams (4); others (see [52])	203
Radar (fine)	Radar array front end (fine-grained filters, equivalent functionality)	channels (12); beams (4); others (see [52])	201
Audiobeam	Audio beamformer, steers channels into a single beam	channels (15)	167
FHR (feedback loop)	Frequency hopping radio (using feedback loop for hop signal)	window size (256)	161
OFDM	Orthogonal frequency division multiplexer (Tennenhouse and Bose, 1996)	decimation rates (825, 5); others (see [52])	148
ChannelVocoder	Channel voice coder	number of filters (16); others (see [52])	135
Filterbank	Filter bank for multi-rate signal processing	bands (8); window size (128)	134
TargetDetect	Target detection using matched filters and threshold	window size (300)	127
FMRadio	FM radio with equalizer	bands (7); win size (128); others (see [52])	121
FHR (teleport messaging)	Frequency hopping radio (using teleport messaging for hop signal)	window size (256)	110
DToA	Audio post-processing and 1-bit D/A converter	window size (256)	100
Graphics Pipelines (4):			
GP - reference version	General-purpose rendering pipeline: 6 vertex shaders, 15 pixel pipes[3]	--	641
GP - phong shading	Phong shading rendering pipeline: 1 vertex shader, 12 two-part pixel pipelines	--	649
GP - shadow volumes	Shadow volumes rendering pipeline: 1 vertex shader, 20 rasterizers	--	460
GP - particle system	Particle system pipeline: 9 vertex shaders, 12 pixel pipelines, split triangle setup	--	631
Libraries / Kernels (23):			
Autocor	Produce auto-correlation series	vector length (32); autocor series length (8)	29
Cholesky	NxN cholesky decomposition	matrix size (16x16)	85
CRC	CRC encoder using 32-bit generator polynomial	--	131
DCT (float)	N-point, one-dimensional DCT (floating point)	window size (16)	105
DCT2D (NxM, float)	NxM DCT (floating point)	window size (4x4)	115
DCT2D (NxN, int, reference)	NxN DCT (IEEE-compliant integral transform, reference version)	window size (8x8)	59
IDCT (float)	N-point, one-dimensional IDCT (floating point)	window size (16)	105
IDCT2D (NxM, float)	NxM IDCT (floating point)	window size (4x4)	115
IDCT2D (NxN, int, reference)	NxN IDCT (IEEE-compliant integral transform, reference version)	window size (8x8)	60
IDCT2D (8x8, int, coarse)	8x8 IDCT (IEEE-compliant integral transform, optimized version, coarse-grained)	--	139
IDCT2D (8x8, int, fine)	8x8 IDCT (IEEE-compliant integral transform, optimized version, fine-grained)	--	146
FFT (coarse - default)	N-point FFT (coarse-grained)	window size (64)	116
FFT (medium)	N-point FFT (medium-grained butterfly, no bit-reverse)	window size (64)	53
FFT (fine 1)	N-point FFT (fine-grained butterfly, coarse-grained bit-reverse)	window size (64)	139
FFT (fine 2)	N-point FFT (fine-grained butterfly, fine-grained bit-reverse	window size (64)	90
Lattice	Ten-stage lattice filter	number of stages (10)	58
MatrixMult (fine)	Fine-grained matrix multiply	matrix dims NxM, MxP (12x12, 9x12)	79
MatrixMult (coarse)	Blocked matrix multiply	matrix dims (same as above); block cuts (4)	120
Oversampler	16x oversampler (found in many CD players)	window size (64)	69
RateConvert	Audio down-sampler, converts rate by 2/3	expand / contract rates (2, 3); win size (300)	58
TDE	Time-delay equalization (convolution in frequency domain)	number of samples (36); FFT size (64)	102
Trellis	Trellis encoder/decoder system, decodes blocks of 8 bytes[2]	frame size (5)	162
VectAdd	Vector-vector addition	--	31
Sorting Routines (8):			
BitonicSort (coarse)	Bitonic sort (coarse-grained)	number of values to sort (16)	73
BitonicSort (fine, iterative)	Bitonic sort (fine-grained, iterative)	number of values to sort (16)	121
BitonicSort (fine, recursive)	Bitonic sort (fine-grained, recursive)	number of values to sort (16)	80
BubbleSort	Bubble sort	number of values to sort (16)	61
ComparisonCounting	Compares each element to every other to determine n'th output	number of values to sort (16)	67
InsertionSort	Insertion sort	number of values to sort (16)	61
MergeSort	Merge sort	number of values to sort (16)	66
RadixSort	Radix sort	number of values to sort (16)	52

Table 1: Overview of the StreamIt benchmark suite.

	TOTAL FILTERS			PEEKING FILTERS			STATEFUL FILTERS[5]			OTHER CONSTRUCTS		
Benchmark	**Types**	**Instances (non-Iden.)**	**Instances (Identity)**	**Types**	**Instances**	**Max Work[6]**	**Types**	**Instances**	**Max Work[6]**	**Splitjoins**	**Feedback Loops**	**Work in F. Loop[6]**
Realistic Apps (30):												
MPEG2 encoder[4]	35	113	30	-	-	-	9	11	N/A	15	-	-
MPEG2 decoder[4]	25	49	13	-	-	-	7	10	N/A	8	-	-
GMTI	95	1111	1757	-	-	-	-	-	-	764	-	-
Mosaic	62	176	17	2	2	N/A	7	7	N/A	20	1	N/A
MP3 subset	10	98	36	2	4	8.6%	-	-	-	23	-	-
MPD	42	110	33	1	11	1.9%	5	7	2.0%	11	-	-
JPEG decoder	17	66	13	1	3	0.00%	-	-	-	11	-	-
JPEG transcoder	12	126	8	2	6	0.00%	-	-	-	20	-	-
FAT	27	143	4	-	-	-	-	-	-	5	-	-
HDTV	20	94	-	1	12	<0.01%	4	38	N/A	28	-	-
H264 subset	28	33	20	-	-	-	-	-	-	16	1	97%
SAR	22	42	-	-	-	-	-	-	-	1	-	-
GSM	17	40	3	1	1	4.7%	3	3	42.4%	6	1	25%
802.11a	28	61	35	1	2	7.6%	-	-	-	18	-	-
DES	21	117	16	-	-	-	-	-	-	32	-	-
Serpent	13	135	33	-	-	-	-	-	-	33	-	-
Vocoder	30	96	4	4	32	8.0%	3	45	0.3%	8	-	-
RayTracer	4	4	-	-	-	-	-	-	-	-	-	-
3GPP	13	60	72	1	8	0.1%	-	-	-	41	-	-
Radar (coarse)	6	73	-	-	-	-	-	-	-	2	-	-
Radar (fine)	6	49	-	-	-	-	1	28	3.9%	2	-	-
Audiobeam	3	18	-	1	15	4.4%	-	-	-	1	-	-
FHR (feedback loop)	9	26	1	-	-	-	1	1	5.1%	1	1	80%
OFDM	6	14	-	1	4	0.1%	1	4	12.2%	1	-	-
ChannelVocoder	5	53	-	3	34	5.2%	-	-	-	1	-	-
Filterbank	9	67	-	2	32	3.1%	-	-	-	9	-	-
TargetDetect	7	10	-	4	4	25%	-	-	-	1	-	-
FMRadio	7	29	-	2	14	7.6%	-	-	-	7	-	-
FHR (teleport messaging)	7	23	3	-	-	-	1	1	4.2%	1	-	-
DToA	7	14	-	1	5	67%	-	-	-	-	1	0.7%
Graphics Pipelines (4):												
GP - reference version	6	54	-	-	-	-	1	15	N/A	2	-	-
GP - phong shading	7	52	-	-	-	-	1	12	N/A	1	-	-
GP - shadow volumes	5	44	-	-	-	-	1	20	N/A	1	-	-
GP - particle system	7	37	-	-	-	-	1	12	N/A	2	-	-
Libraries / Kernels (23):												
Autocor	3	10	-	-	-	-	-	-	-	1	-	-
Cholesky	5	35	15	-	-	-	-	-	-	15	-	-
CRC	4	48	2	-	-	-	-	-	-	-	1	99%
DCT (float)	6	31	7	-	-	-	-	-	-	14	-	-
DCT2D (NxM, float)	6	42	8	-	-	-	-	-	-	18	-	-
DCT2D (NxN, int, reference)	3	20	-	-	-	-	-	-	-	2	-	-
IDCT (float)	6	48	7	-	-	-	-	-	-	21	-	-
IDCT2D (NxM, float)	6	50	8	-	-	-	-	-	-	26	-	-
IDCT2D (NxN, int, reference)	3	20	-	-	-	-	-	-	-	2	-	-
IDCT2D (8x8, int, coarse)	2	4	-	-	-	-	-	-	-	-	-	-
IDCT2D (8x8, int, fine)	2	18	-	-	-	-	-	-	-	2	-	-
FFT (coarse - default)	4	13	-	-	-	-	-	-	-	-	-	-
FFT (medium)	5	20	6	-	-	-	-	-	-	12	-	-
FFT (fine 1)	4	195	-	-	-	-	-	-	-	44	-	-
FFT (fine 2)	4	99	64	-	-	-	-	-	-	96	-	-
Lattice	4	18	10	-	-	-	-	-	-	9	-	-
MatrixMult (fine)	4	14	30	-	-	-	-	-	-	5	-	-
MatrixMult (coarse)	4	4	25	-	-	-	-	-	-	7	-	-
Oversampler	5	10	-	1	4	52.7%	-	-	-	-	-	-
RateConvert	5	5	-	1	1	97.6%	-	-	-	-	-	-
TDE	7	29	-	-	-	-	-	-	-	-	-	-
Trellis	14	12	1	1	1	N/A	2	2	N/A	1	-	-
VectAdd	3	4	-	-	-	-	-	-	-	1	-	-
Sorting Routines (8):												
BitonicSort (coarse)	4	6	-	-	-	-	-	-	-	-	-	-
BitonicSort (fine, iterative)	3	82	-	-	-	-	-	-	-	44	-	-
BitonicSort (fine, recursive)	3	62	16	-	-	-	-	-	-	37	-	-
BubbleSort	3	18	-	1	16	5.9%	1	16	5.9%	-	-	-
ComparisonCounting	4	19	1	-	-	-	-	-	-	1	-	-
InsertionSort	3	6	-	-	-	-	-	-	-	-	-	-
MergeSort	3	17	-	-	-	-	-	-	-	7	-	-
RadixSort	3	13	-	-	-	-	-	-	-	-	-	-

Table 2: Properties of filters and other constructs in StreamIt benchmarks.

```
int->int filter DifferenceEncoder_Stateless {

    prework push 1 peek 1 {
        push(peek(0));
    }

    work push 1 pop 1 peek 2 {
        push(peek(1)-peek(0));
        pop();
    }
}
```

Figure 3: Stateless version of a difference encoder, using peeking and prework.

```
int->int filter DifferenceEncoder_Stateful {
    int state = 0;

    work push 1 pop 1 {
        push(peek(0)-state);
        state = pop();
    }
}
```

Figure 4: Stateful version of a difference encoder, using internal state.

responsible for all of the peeking in 7 benchmarks (3GPP, OFDM, Filterbank, TargetDetect, DToA, Oversampler, RateConvert) and some of the peeking in 3 others (Vocoder, ChannelVocoder, FMRadio).

A second pattern of peeking is when a filter peeks at exactly one item beyond its pop window. An example of this filter is a difference encoder, as used in the JPEG transcoder and Vocoder benchmarks. On its first execution, this filter's output is the same as its first input; on subsequent executions, it is the difference between neighboring inputs. As illustrated in Figure 3, a difference encoder can be written as a stateless filter using peeking (and prework, as described later). Otherwise, the filter is forced to maintain internal state, as illustrated in Figure 4. Across our benchmark suite, this pattern accounts for almost one third (10 out of 34) of the peeking filter declarations. It accounts for all of the peeking in 4 benchmarks (Mosaic, JPEG decode, JPEG transcode, HDTV, BubbleSort) and some of the peeking in 2 others (Vocoder, FMRadio). It should be noted that the operation performed on the two items is sometimes non-linear; for example, Mosaic determines the correlation between successive frames, FMRadio performs an FM demodulation, and HDTV performs an XOR.

The remaining peeking filters (9 out of 34) perform various sliding-window functions. For example, MP3 reorders and adds data across large (>1000 item) sliding windows; 802.11 and Trellis do short (3-7 item) bit-wise operations as part of an error-correcting code; Vocoder and Audiobeam use peeking to skip N items (by default 1-14), analogous to an inverse delay; ChannelVocoder performs a sliding autocorrelation and threshold across N items (by default 100).

Without peeking, the filters described above would have to be written in a stateful manner, as the locations peeked would be converted to internal states of the filter. This inhibits parallelization, as there is a dependence between successive filter executions. To estimate the resulting performance impact, Figure 2 lists the approximate amount of work in the most computationally-heavy peeking filter in each benchmark. For 11 benchmarks, this work represents a significant fraction of the program load (minimum 3.1%, median 8%, maximum 97.6%) and would represent a new bottleneck in a parallel computation. For 7 benchmarks, the state that would be introduced by peeking is dwarfed by state already present for other reasons. For the remaining 3 benchmarks, the peeking filters represent a negligible (0.1%) fraction of work.

4.2 Startup behaviors often differ from steady-state behaviors. The compiler must recognize such behaviors to avoid a throughput bottleneck.

The prework function allows a filter to have different behavior on its first invocation. This capability is utilized by 15 benchmarks, in 20 distinct filter declarations (results not shown in table).

The most common use of prework is for implementing a delay; on the first execution, the filter pushes N placeholder items, while on subsequent executions it acts like an Identity filter. A delay is used in 8 benchmarks (MPD, HDTV, Vocoder, 3GPP, Filterbank, DToA, Lattice, and Trellis). Without prework, the delayed items would need to be buffered within the filter, introducing a stateful bottleneck to the computation.

Other benchmarks use prework for miscellaneous startup conditions. As mentioned previously, the difference encoder in Figure 3 relies on prework (used in JPEG transcoder and Vocoder), as does the analogous difference decoder (used in JPEG decoder). The MPEG2 encoder and decoder use prework in filters relating to picture reordering, while GSM and CRC use prework for functions analogous to delays. Prework is also used for initialization in MPD, HDTV, and 802.11.

4.3 Sequential (stateful) filters are required in one quarter of our benchmarks. Further state could be eliminated via new language constructs, compiler analyses, or programmer interventions.

After effective use of peeking and prework primitives, one quarter (17 out of 65) of the benchmarks still contain one or more filters with mutable state. There are 49 stateful filter types in the StreamIt benchmark suite, representing approximately 6% of the total filters. The heaviest stateful filter in each benchmark ranges from 0.3% to 42.4% (median 4.7%) of the overall work, representing an eventual bottleneck to parallelization.

Of the stateful filters, at least 22 (about 45%) represent fundamental feedback loops that are an intrinsic part of the underlying algorithm. Filters in this category include the bit-alignment stage of MPEG encoding, which performs data-dependent updates to the current position; reference frame encoding in MPEG encoder, which sometimes stores information about a previous frame; the parser in MPEG decoder, which suspends and restores its cur-

[1]Only non-comment, non-blank lines of code are counted. Line counts do not include libraries used, though other statistics do consider both the application and its libraries.

[2]Some helper functions in FAT, HDTV, and Trellis remain untranslated from the Java-based StreamIt syntax.

[3]The graphics pipelines are described in more detail elsewhere [11].

[4]Due to the large size of MPEG2, splitjoins replicating a single filter are automatically collapsed by the compiler prior to gathering statistics.

[5]Source and sink nodes that generate synthetic input, check program output, or perform file I/O are not counted as stateful.

[6]Work is given as an estimated fraction of the overall program, as calculated by a static analysis. Actual runtimes may differ by 2x or more. Work estimates are not available (N/A) given dynamic rates (MPEG2, Mosaic, Graphics pipelines) or external Java routines (HDTV, Trellis).

rent control flow position in order to maintain a constant output rate; the motion prediction, motion vector decode, and picture reordering stages of MPEG decoder, which contain data-dependent updates of various buffers; the pre-coding and Ungerboeck encoding stages of HDTV, which are simple feedback loops; the Ungerboeck decoding stage of HDTV (and analogously in Trellis) which mutates a persistent lookup table; multiple feedback loops in GSM; an accumulator, adaptive filter, and feedback loop in Vocoder; incremental phase correction in OFDM; and persistent screen buffers in the graphics pipelines.

The remaining filters classified as stateful may be amenable to additional analyses that either eliminate the state, or allow restricted parallelism even in the presence of state. The largest category of such filters are those in which the state variables are modified only by message handlers. Whether such messages represent a genuine feedback loop depends on whether the filter sending the message is data-dependent on the outcome of the filter receiving the message. Even if a feedback loop does exist, it may be possible to exploit bounded parallelism due to the intrinsic delay in that loop, or speculative parallelism due to the infrequent arrival of most teleport messages. In our benchmarks, there are 16 filters in which the state is mutated only by message handlers; they originate from MPEG encoder, MPEG decoder, Mosaic, and both versions of FHR. There are also 4 additional filters (drawn from MPEG encoder, MPEG decoder, and Mosaic) in which message handlers account for some, but not all, of the state.

A second category of state which could potentially be removed is that of induction variables. Several filters keep track of how many times they have been invoked, in order to perform a special action every N iterations. For example, MPEG encoder counts the frame number in assigning the picture type; MPD and Radar (fine grained version) count the position within a logical vector while performing FIR filtering; and Trellis includes a noise source that flips a bit every N items. Other filters keep track of a logical two-dimensional position, incrementing a column counter on every iteration and only incrementing the row counter when a column is complete. Filters in this category include motion estimation from MPEG encoder, and two filters from MPD. Other filters in MPD contain more complex induction variables; an accumulator is reset when a different counter wraps-around to zero. Taken together, there are a total of 9 filters that could become stateless if all induction variables could be converted to a closed form.

There are two approaches for eliminating induction variables from filter state. The first approach is to recognize them automatically in the compiler. While this is straightforward for simple counters, it may prove difficult for nested counters (tracking both row and column) or co-induction variables (periodically reseting one variable based on the value of another). The second approach is to provide a new language primitive that automatically returns the current iteration number of a given filter. This information can easily be maintained by the runtime system without inhibiting parallelization; shifting the burden from the programmer to the compiler would improve both programmability and performance.

The third and final category of state that could potentially be removed is that which results from writing a logically coarse-grained filter at a fine level of granularity. This can result in a filter in which state variables are reset every N executions, corresponding to one coarse-grained execution boundaries. Such filters can be re-written in a stateless manner by moving state variables to local variables in the work function, and scaling up the execution of the work function to represent N fine-grained iterations. Such coarsening would eliminate the state in bubble sort, which is reset at boundaries between data sets, as well as a complex periodic filter (LMaxCalc) in MPD. It would also eliminate many of the induction variables described previously, as they are also periodic. This approach provides a practical solution for eliminating state, and was employed in translating Radar from the original fine-grained version to a coarse-grained alternative (both of which appear in our benchmark suite). The drawbacks of this transformation are the effort required from the programmer and also the increased size of the resulting filter. Coarse-grained filters often incur a larger code footprint, a longer compile time, and a less natural mapping to fine-grained architectures such as FPGAs. While the StreamIt language aims to be agnostic with respect to the granularity of filters, in some cases the tradeoff between writing stateless filters and writing fine-grained filters may need to be iteratively explored to achieve the best performance.

4.4 Feedback loops are uncommon in our benchmarks, but represent significant throughput bottlenecks when present.

While our discussion thus far has focused on stateful filters, six benchmarks also contain explicit feedback loops in the graph structure. Three additional benchmarks (MPEG2 encoder, Mosaic, and FHR) contain implicit loops due to teleport messages that are sent upstream (see [52] for details). Of the explicit feedback loops, three represent significant bottlenecks to parallelization (FHR feedback, H264 subset, CRC), with workloads ranging from 80% to 99% of the overall execution. The loop in GSM is shadowed by a stateful filter; the loop in DToA represents only 0.7% of the runtime; and the loop in Mosaic, while likely a bottleneck, is difficult to quantify due to dynamic rates. Unlike some of the stateful filters, these feedback loops are all intrinsic to the algorithm and are not subject to automatic removal. However, feedback loops can nonetheless afford opportunities for parallelism due to the delay in the loop – that is, if items are enqueued along the feedback path at the start of execution, then they can be processed in parallel. Further analysis of these delays is needed to assess the potential parallelism of feedback loops in our benchmark suite.

5. SCHEDULING CHARACTERISTICS

The semantics of a stream program is that all filters execute concurrently, limited only by the availability of data on the input channels (and the availability of space on the output channels). Thus, it is the role of the compiler to schedule the actual sequence of filter executions on a given target. This section describes the constraints and opportunities of the scheduling process, as informed by our benchmark suite.

5.1 Neighboring filters often have matched I/O rates. This reduces the opportunity and impact of advanced scheduling strategies proposed in the literature.

Many of the advanced scheduling strategies for synchronous dataflow graphs have the highest payoff when the input and output rates of neighboring filters are mismatched. For example, the CD-DAT benchmark (shown in Figure 5) is used in many studies [3, 4, 10, 31, 39, 40, 50]; it converts compact disk auto (sampled at 44.1 khz) to digital audio tape (sampled at 48 khz). Performing this conversion in stages improves efficiency [39]. However, neighboring filters have different communication rates which share no common factors, resulting in a large steady-state schedule[3].

[3]A *steady-state schedule* is a sequence of filter firings that preserves the number of items buffered between each pair of filters. It can be repeated infinitely without risking buffer overflow or underflow.

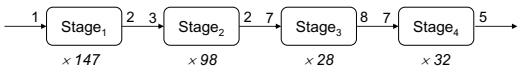


Figure 5: The CD-DAT benchmark [40] exhibits unusually mismatched I/O rates. Nodes are annotated with the number of items pushed and popped per execution, as well as their execution multiplicity in the steady state. Since neighboring filters produce different numbers of items, each filter has a large multiplicity in the steady state. This demands clever scheduling strategies to avoid extremely large buffer sizes.

In our benchmark suite, mismatched communication rates as seen in CD-DAT are rare. The common case is that the entire benchmark is operating on a logical frame of data which is passed through the entire application. Sometimes there are differences in the input and output rates for filters that operate at different levels of granularity; for example, processing one frame at a time, one macroblock at a time, or one pixel at a time. However, these rates have a small common multiple (i.e., the frame size) and can be accommodated without growing the steady state schedule. The JPEG transcoder provides an example of this; Figure 6 illustrates part of the stream graph that operates on a single 8x8 macroblock.

To provide a quantitative assessment of the number of matched rates in our benchmark suite, Figure 7 summarizes the key properties of the steady state schedule derived for each program. We consider the minimal steady state schedule, which executes each filter the minimum number of times so as to consume all of the items produced by other filters in the graph. We count the number of times that each filter executes in this schedule, which we refer to as the *multiplicity* for the filter. The table illustrates, for each benchmark, the minimum multiplicity, the mode multiplicity, and the percentage of filters that have the mode multiplicity (the mode frequency).

The most striking result from the table is that 89% (58 out of 65) of the benchmarks have a minimum filter multiplicity of 1. That is, there exists at least one filter in the program that executes only once in the steady state schedule. This filter defines the logical frame size for the execution; all other filters are simply scaled up to satisfy the input or output requirements of the filter.

The second highlight from the table is that, on average, 63% of the filters in a program share the same multiplicity. For over two-thirds of the benchmarks (44 out of 65), the most common multiplicity is 1; in these benchmarks, an average of 72% of the filters also have a multiplicity of 1. The mode multiplicity can grow higher than 1 in cases where one filter operates at a coarse granularity (e.g., a frame), but the majority of filters operate at a fine granularity (e.g., a pixel). In these benchmarks, 46% of the filters still share the same multiplicity.

The prevalence of matched rates in our benchmark suite also led to unexpected results in some research papers. For example, phased scheduling is an algorithm that reduces the buffer requirements needed to execute a synchronous dataflow graph [29]. The space saved on CD-DAT is over 14x. However, the median savings across our benchmark suite at the time (a subset of the suite presented here) is less than 1.2x. The reason is that the potential savings on most benchmarks is extremely small due to matched input and output rates; simply executing each node once often gives the minimal possible buffering. This result emphasizes the importance of optimizing the common case in realistic programs, rather than restricting attention to small examples.

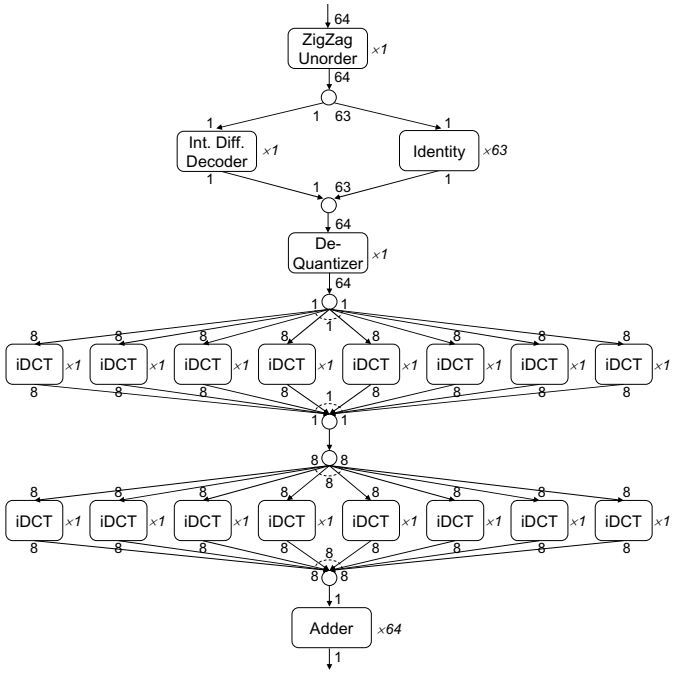


Figure 6: This excerpt from the JPEG transcoder illustrates matched I/O rates, as found in many benchmarks. The graph is transforming pixels from an 8x8 macroblock. Nodes are annotated with the number of items pushed and popped per execution, as well as their execution multiplicity in the steady state. Since neighboring filters often produce the same number of items on each execution, all filters except for Identity and Adder execute exactly once in the steady state. This offers less flexibility to optimize the schedule, and affords less benefit from doing so.

5.2 It is not useful for filters to divide their atomic execution step into a cycle of smaller steps. However, multiple execution steps are important for scatter/gather stages.

At one point in the StreamIt project, we embraced the cyclo-static dataflow model [6, 42] for all filters. Under this model, the programmer can define multiple work functions that are executed under a specified pattern. By dividing execution into more fine-grained units, cyclo-static dataflow can offer lower latency than synchronous dataflow, and can also avoid deadlock in tightly constrained loops.

However, for general filters, we did not find any compelling application of cyclic execution steps across our benchmark suite. The concept of multiple execution steps was confusing to programmers, who often interpreted the steps as belonging to different filters, or as being interchangeable with a normal function call. For this reason, we eventually changed course and removed cyclo-static dataflow from the StreamIt language.

Multiple execution steps did prove to be important to the semantics of splitters and joiners, which would have an unreasonably large granularity if they were forced to transfer a full cycle of data at a single time. For example, a feedback loop in the GSM benchmark requires fine-grained splitting and joining to avoid deadlock [29]. Because StreamIt relies on a few built-in primitives for splitting and joining, the subtlety of this execution semantics could be hidden from the programmer.

Benchmark	Dynamic Rate Filters[a]	Filter Execs per Steady State[b]		
		Min	Mode	Mode Freq.
Realistic Apps (30):				
MPEG2 encoder[c]	7	1	960	17%
MPEG2 decoder[c]	1	1	990	19%
GMTI	-	1	1	56%
Mosaic	5	2	2	27%
MP3 subset	-	1	18	52%
MPD	-	1	416	25%
JPEG decoder	2	1	4800	72%
JPEG transcoder	2	1	1	85%
FAT	-	1	1	24%
HDTV	-	20	1380	38%
H264 subset	-	17	396	26%
SAR	-	1	1	95%
GSM	-	1	1	65%
802.11a	-	1	1	14%
DES	-	1	1	62%
Serpent	-	1	1	40%
Vocoder	-	1	1	88%
RayTracer	-	1	1	100%
3GPP	-	1	9	48%
Radar (coarse)	-	1	1	38%
Radar (fine)	-	1	1	49%
Audiobeam	-	1	1	94%
FHR (feedback loop)	-	1	1	26%
OFDM	-	1	1	57%
ChannelVocoder	-	1	50	66%
Filterbank	-	1	8	64%
TargetDetect	-	1	1	90%
FMRadio	-	1	1	97%
FHR (teleport messaging)	-	1	1	23%
DToA	-	1	16	43%
Graphics Pipelines (4):				
GP - reference version	15	1	2	85%
GP - phong shading	12	1	1	94%
GP - shadow volumes	20	1	1	93%
GP - particle system	12	1	36	70%
Libraries / Kernels (23):				
Autocor	-	1	1	80%
Cholesky	-	1	1	70%
CRC	-	1	1	98%
DCT (float)	-	1	1	79%
DCT2D (NxM, float)	-	1	1	84%
DCT2D (NxN, int, reference)	-	1	1	80%
IDCT (float)	-	1	1	85%
IDCT2D (NxM, float)	-	1	1	83%
IDCT2D (NxN, int, reference)	-	1	1	80%
IDCT2D (8x8, int, coarse)	-	1	64	50%
IDCT2D (8x8, int, fine)	-	1	1	89%
FFT (coarse - default)	-	1	1	23%
FFT (medium)	-	32	32	92%
FFT (fine 1)	-	1	1	99%
FFT (fine 2)	-	1	1	98%
Lattice	-	1	1	100%
MatrixMult (fine)	-	9	108	30%
MatrixMult (coarse)	-	9	12	31%
Oversampler	-	1	1	20%
RateConvert	-	2	2	40%
TDE	-	1	15	24%
Trellis	-	1	40	46%
VectAdd	-	1	1	100%
Sorting Routines (8):	-			
BitonicSort (coarse)	-	1	1	67%
BitonicSort (fine, iterative)	-	1	1	100%
BitonicSort (fine, recursive)	-	1	1	95%
BubbleSort	-	1	1	100%
ComparisonCounting	-	1	1	85%
InsertionSort	-	1	1	67%
MergeSort	-	1	1	88%
RadixSort	-	1	1	85%

Figure 7: Scheduling statistics for StreamIt benchmarks.

5.3 Dynamic I/O rates are necessary for expressing several applications.

Support for dynamic rates was introduced years after the initial StreamIt release, leading to less widespread usage within our benchmarks than might be expected for the streaming domain as a whole.

As illustrated in Figure 7, dynamic rates are utilized by only 9 of our benchmarks, but are absolutely necessary to express these benchmarks in StreamIt. Though there are a total of 76 dynamic-rate filters instantiated across the benchmarks, these instantiations correspond to only 14 filter types that perform a set of related functions. In JPEG and MPEG2, dynamic-rate filters are needed to parse and also create both the BMP and MPEG formats. MPEG2 encoder also requires a dynamic-rate filter to reorder pictures (putting B frames in the appropriate place). All of these filters have unbounded push, pop, and peek rates, though in JPEG and MPEG2 decoder there is a minimum rate specified.

In Mosaic, dynamic rates are used to implement a feedback loop (in the RANSAC algorithm) that iterates an unpredictable number of times; the signal to stop iteration is driven by a teleport message [1]. The entry to the loop pops either 0 or 1 items, while the exit from the loop pushes either zero or one items. Mosaic also contains three parameterized filters, in which the input and output rates are governed by the number of points of interest as determined by the algorithm. The count is established via a teleport message, thus fixing the input and output rates prior to a given iteration.

In the graphics pipelines, the only dynamic-rate filters are the rasterizers, which expand each triangle into an unknown number of pixels.

6. PROGRAMMING STYLE

In addition to our observations about the benchmark characteristics, we also offer some lessons learned from developers' experiences in implementing stream programs. The StreamIt benchmarks were developed by 22 different people; all but one of them were students, and half of them were undergraduates or M.Eng students at MIT. As the developers were newcomers to the StreamIt language, we expect that their experience would reflect that of a broader user population; their coding style was not influenced by the intent of the original language designers.

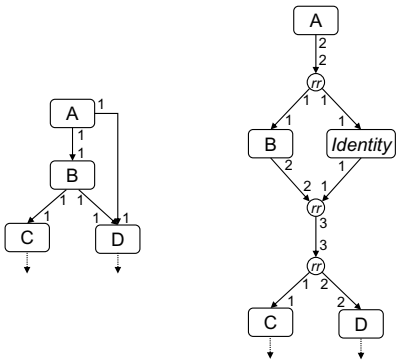
6.1 It is useful and tractable to write programs using structured streams, in which all modules have a single input and a single output. However, structured streams are occasionally unnatural and, in rare cases, insufficient.

Overall, we found structured streams – the hierarchical composition of pipelines, splitjoins, and feedbackloops – to be a good match for the applications in our benchmark suite. Splitjoins appear in over three quarters (51 out of 65) of the benchmarks, with a median of 8 instantiations per benchmark. Roundrobin splitters

[a] Figures represent the number of runtime instances of dynamic-rate filters. Number of corresponding static filter types are provided in the text.

[b] Dynamic rate filters are replaced with push 1, pop 1 filters for calculation of the steady state. Splitters and joiners are not included in the counts.

[c] Due to the large size of MPEG2, splitjoins replicating a single filter are automatically collapsed by the compiler prior to gathering statistics.



(a) Unstructured (b) Structured

Figure 8: Example of refactoring a stream graph to fit a structured programming model. Both graphs achieve equivalent communication between filters.

account for 65% of the instantiations, while the other splitters are of duplicate type. (All joiners are of roundrobin type.) While the developer sometimes had to refactor an unstructured block diagram into structured components, the result was nonetheless a viable way to represent the application.

One shortcoming of structure is that it can force programmers to multiplex and demultiplex conceptually-distinct data streams into a single channel. The underlying cause of this hazard is illustrated in Figure 8. Because filters C and D are running in parallel, their input streams must converge at a common splitter under a structured programming model. However, this implies that the auxiliary communication from A to D must also pass through the splitter, in a manner that is interleaved with the output of B. An extra splitjoin (at the top of Figure 8b) is needed to perform this interleaving. The 3GPP benchmark represents a more realistic example of this hazard; see our full report for details [52, p.48].

Needless to say, this pattern of multiplexing and demultiplexing adds considerable complexity to the development process. It requires the programmer to maintain an unwritten contract regarding the logical interleaving of data streams on each physical channel. Moreover, the addition of a new communication edge in the stream graph may require modification to many intermediate stages.

While there is no perfect solution to this problem, we have sometimes embraced two imperfect workarounds. First, the data items in the multiplexed streams can be changed from a primitive type to a structure type, allowing each logical stream to carry its own name. This approach would benefit from a new kind of splitter and joiner which automatically packages and un-packages structures from adjoining data channels. The second approach is to employ teleport messaging, which allows point-to-point communication and avoids interleaving stream data. However, since it is designed for irregular control messages, it does not expose information about the steady-state dataflow to the compiler.

In practice, we have chosen to tolerate the occasional complexity of stream multiplexing rather than to fall back on an unstructured programming model. However, it may be valuable to consider a natural syntax for unstructured components of the stream graph – the analog of break and continue statements (or even a rare GOTO statement) in structured control flow. It is important to note, however, that there is no overhead introduced by adding splitters and joiners to the stream graph; the StreamIt compiler analyzes the communication (via an analysis known as *synchronization removal*) to recover the original unstructured communication [19].

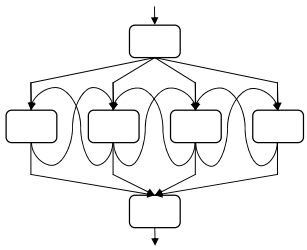


Figure 9: A communication pattern unsuitable for structured streams. This pattern can arise in video compression, where each block informs its neighbors of its motion prediction before the next processing step.

Finally, there are rare cases in which the structured primitives in StreamIt have been inadequate for representing a streaming communication pattern. Figure 9 illustrates an example from video compression, where each parallel filter performs a motion prediction for a fixed area of the screen. Between successive frames, each filters shares its prediction with its neighbors on either side. While this could be represented with a feedback loop around the entire computation, there would be complicated interleaving involved.

6.2 Programmers can accidentally introduce unnecessary sequential bottlenecks (mutable state) in filters.

Filters that have no mutable state are attractive because they can be run in a data-parallel fashion. Unfortunately, the performance cost of introducing state is not exposed in the current StreamIt language. Thus, we found that several programmers, when faced with two alternative implementations of an algorithm, would sometimes choose the one that includes mutable state. Figure 4 gives a pedantic example of this problem, while our accompanying report [52, p.51] illustrates a realistic case from MPD. Prior to conducting our performance evaluations, we examined all stateful filters in the benchmarks and rewrote them as stateless filters when it was natural to do so. In future stream languages, it may be desirable to require an extra type modifier on stateful filters, such as a *stateful* keyword in their declaration, to force programmers to be cognizant of any added state and to avoid it when possible.

7. RELATED WORK

While other researchers have characterized benchmark suites that overlap the domain of stream programs [2, 5, 15, 22, 32, 34, 38, 46, 48, 57], our work differs in two key respects: 1) we examine programs written in a stream programming language, rather than a general purpose language, and 2) we characterize high-level programming patterns and their implications for programming language design, rather than low-level statistics (instruction mix, branch behavior, etc.) that influence architectural design. Concurrently to this publication, Gordon also published an in-depth analysis of the "StreamIt Core benchmarks", a set of 12 programs that has served as the focus of performance optimizations in the StreamIt group [19]. Separate publications are also devoted to the StreamIt implementations of MPEG2 [16], Mosaic [1], MPD [27], and FHR [54].

The MediaBench suite [32] consists of multimedia applications that (with the exception of GhostScript) can also be considered as stream programs. Three of the applications (JPEG, MPEG, and GSM) overlap with our benchmarks. MediaBench (which is written in C) has been shown to have different caching behavior than

SPECint [32]. Researchers have further characterized MediaBench along architectural axes such as instruction mix, branch prediction rates, working set sizes, locality, and instruction level parallelism [7, 18]. Recently, MediaBench II has been released with a focus on video workloads; the authors utilize similar characterization metrics [17]. We are unaware of any characterization of multimedia applications that focuses on high-level programming patterns or uses a stream programming language.

The STREAM benchmark is a small kernel designed to assess memory bandwidth [37]. Available in C and FORTRAN, it measures performance on four long vector operations. VersaBench [43] includes embedded streaming benchmarks in addition to four other domains; the stream programs were drawn from the StreamIt benchmark suite, including FM Radio and Radar as described in this paper. VersaBench has been characterized according to instruction mix, temporal and spatial locality, and instruction level parallelism [43].

There are a number of stream programming languages in addition to StreamIt, including Brook [9], StreamC/KernelC [28], Cg [36], Baker [12], SPUR [58] and Spidle [14]. The authors of Brook and StreamC/KernelC point to data parallelism, arithmetic intensity, and (in the case of StreamC/KernelC) lack of data reuse as defining aspects of the streaming domain [9, 30], though we are unaware of a quantitative characterization.

Perhaps most closely related to StreamIt is the Brook language, which provides an architecture-independent representation and has been mapped to graphics processors [9]. A set of Brook programs are available as part of a public release [8]. While we would expect most of our findings to extend to these programs, some differences would be evident due to differences in the languages. Brook programs would exhibit higher data parallelism, as stateful filters are prohibited by the language. Brook programs would also show a higher prevalence of dynamic rates, which were incorporated earlier into Brook than into StreamIt. The presence of explicit I/O rates and startup conditions in StreamIt makes it easier to study scheduling characteristics in StreamIt than in Brook. Also, primitives such as structured streams and teleport messaging are unique to StreamIt. It would be interesting to undertake a deeper comparison of the Brook and StreamIt benchmark suites in future work.

8. CONCLUSIONS

To the best of our knowledge, this paper represents the first characterization of a benchmark suite that was authored in a stream programming language. By starting from a streaming representation, we can focus on the underlying properties of the streaming algorithms, rather than fighting to extract those algorithms from a general-purpose programming language.

The characterization described in this paper has had a significant impact on the direction of the StreamIt project. Lessons learned from the benchmark suite caused us to do the following:

1. Optimize parallelization of sliding windows [19], based on their prevalence in our suite (Section 4.1).
2. Introduce prework functions to support observed startup behaviors (Section 4.2).
3. Migrate our parallelization algorithm from one that favors pipeline parallelism [21] to one that favors data parallelism [20], based on the observed scarcity of stateful filters (Section 4.3).
4. Curtail pursuit of sophisticated scheduling optimizations that apply to large steady-state schedules [29], given that most programs have small steady states in practice (Section 5.1).
5. Remove support for multi-phase filters from the language and compiler, given that they were unnecessary and confusing (Section 5.2).

In addition, the characterization highlights several opportunities for further improving the StreamIt language. Based on lessons learned in this paper, in the future we hope to:

1. Exploit parallelism that is hidden behind benign stateful dependences, including message handlers, induction variables, and artifacts of filter granularity (Section 4.3).
2. Provide constructs for programmers to simplify expression of unstructured data flows in a structured programming model (Section 6.1).
3. Introduce mechanisms (such as a *stateful* annotation) to prevent programmers from accidentally introducing mutable state (Section 6.2).

It is important to emphasize that one must exercise care in generalizing our results, as all of our benchmarks are written in a single stream programming language (StreamIt) and the choice of programming language can often influence the expression of an algorithm. For example, as described in Section 4.3, some of the state observed in StreamIt filters could be eliminated via different language constructs and compiler analyses. However, we also believe that most of the patterns described in this paper are fundamental to streaming algorithms; for example, sliding windows, startup vs. steady state, and I/O rates are inseparable from the underlying algorithm, and would be present in one form or another in any programming language. Many of these properties could actually be inferred from a block diagram of the algorithm (e.g., the MPEG2 specification); however, we gain precision and completeness by analyzing a real implementation. We hope that future researchers will also undertake characterizations of stream programs in other stream languages, to better understand the benefits and drawbacks of various points in the design space.

9. ACKNOWLEDGEMENTS

This work would not have been possible without the hard work of the entire StreamIt team [49]. We are particularly grateful to those who contributed to the StreamIt benchmark suite [52, p.37]: Sitij Agrawal, Basier Aziz, Jiawen Chen, Matthew Drake, Shirley Fung, Michael Gordon, Ola Johnsson, Andrew Lamb, Chris Leger, Michal Karczmarek, David Maze, Ali Meli, Mani Narayanan, Satish Ramaswamy, Rodric Rabbah, Janis Sermulins, Magnus Stenemo, Jinwoo Suh, Zain ul-Abdin, Amy Williams, and Jeremy Wong. This work was funded in part by the National Science Foundation (grants 0325297, 0832997, and 0811696).

10. REFERENCES

[1] A. Aziz. Image-based motion estimation in a stream programming language. M.Eng. Thesis, MIT, 2007.

[2] M. D. Berry, D. Chen, P. Koss, and D. Kuck. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. Technical Report CSRD 827, UIUC, 1998.

[3] B. Bhattacharya and S. S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. In *Rapid System Prototyping*, 2000.

[4] S. Bhattacharyya, P. Murthy, and E. Lee. Optimal parenthesization of lexical orderings for DSP block diagrams. In *International Workshop on VLSI Signal Processing*, 1995.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*, 2008.

[6] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static data flow. In *ICASSP*, 1995.

[7] B. Bishop, T. Kelliher, and M. Irwin. A detailed analysis of MediaBench. In *IEEE Workshop on Signal Processing Systems*, 1999.

[8] Brook online release. `http://sourceforge.net/projects/brook/files/`.

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH*, 2004.

[10] N. Chandrachoodan, S. Bhattacharyaa, and K. Liu. An efficient timing model for hardware implementation of multirate dataflow graphs. In *ICASSP*, volume 2, 2001.

[11] J. Chen, M. I. Gordon, W. Thies, M. Zwicker, K. Pulli, and F. Durand. A reconfigurable architecture for load-balanced rendering. In *Graphics Hardware*, 2005.

[12] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju. Shangri-La: Achieving high performance from compiled network applications while enabling ease of programming. In *PLDI*, 2005.

[13] Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge. Stream compilation for real-time embedded multicore systems. In *CGO*, 2009.

[14] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu. Spidle: A DSL approach to specifying streaming applications. In *Generative Programming and Component Engineering*, 2003.

[15] P. J. de Langen, B. Juurlink, and S. Vassiliadis. HandBench: A Benchmarking Suite for Processors Embedded in Handheld Devices. In *Workshop on Circuits, Systems and Signal Processing*, 2004.

[16] M. Drake. Stream programming for image and video compression. M.Eng. Thesis, MIT, 2006.

[17] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. MediaBench II video: Expediting the next generation of video systems research. *Microprocessors & Microsystems*, 33(4), 2009.

[18] J. E. Fritts, W. H. Wolf, and B. Liu. Understanding multimedia application characteristics for designing programmable media processors. In *Multimedia Hardware Architectures*. SPIE, 1999.

[19] M. I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. PhD thesis, MIT, 2010.

[20] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, pipeline parallelism in stream programs. In *ASPLOS*, 2006.

[21] M. I. 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. In *ASPLOS*, 2002.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workshop on Workload Characterization*, 2001.

[23] A. Hagiescu, W. Wong, D. Bacon, and R. Rabbah. A computing origami: Folding streams in FPGAs. In *DAC*, 2009.

[24] A. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT*, 2009.

[25] A. Hormati, Y. Choi, M. Woh, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Macross: Macro-simdization of streaming applications. In *ASPLOS*, 2010.

[26] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah. Optimus: Efficient realization of streaming applications on FPGAs. In *CASES*, 2008.

[27] O. Johnsson, M. Stenemo, and Z. ul Abdin. Programming and implementation of streaming applications. Technical Report IDE0405, Halmstad University, 2005.

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

[29] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. In *LCTES*, 2003.

[30] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang. Imagine: Media Processing with Streams. *IEEE Micro*, 2001.

[31] M.-Y. Ko, C.-C. Shen, and S. S. Bhattachryya. Memory-constrained block processing for dsp software optimization. *Journal of Signal Processing Systems*, 50(2), 2008.

[32] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia andcommunications systems. In *IEEE MICRO*, 1997.

[33] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computing*, 36(1), 1987.

[34] M. L. Li, R. Sasanka, S. Adve, Y. K. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *International Symposium on Workload Characterization*, 2005.

[35] M. Ludlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI*, 2008.

[36] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *SIGGRAPH*, 2003.

[37] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec. 1995.

[38] G. Memik, W. H. Mangione-Smith, and W. Hu. Netbench: A benchmarking suite for network processors. In *ICCAD*, 2001.

[39] P. Murthy, S. Bhattacharyya, and E. Lee. Minimizing memory requirements for chain-structured synchronous dataflow programs. In *ICASSP*, 1994.

[40] P. K. Murthy and S. S. Bhattacharyya. Buffer merging – a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transactions on Design Automation for Electronic Systems*, 9(2), 2004.

[41] M. Narayanan and K. Yelick. Generating permutation instructions from a high-level description. In *Workshop on Media and Streaming Processors*, 2004.

[42] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cycle-static dataflow. In *Asilomar Conference on Signals, Systems, and Computers*, 1995.

[43] R. M. Rabbah, I. Bratt, K. Asanovic, and A. Agarwal. Versatility and Versabench: A New Metric and a Benchmark Suite for Flexible Architectures. Technical Report MIT-LCS-TM-646, MIT, 2004.

[44] S. Seneff. Speech transformation system (spectrum and/or excitation) without pitch extraction. Master's thesis, MIT, 1980.

[45] D. Seo and M. Thottethodi. Disjoint-path routing: Efficient communication for streaming applications. In *IPDPS*, 2009.

[46] N. Slingerland and A. Smith. Design and characterization of the Berkeley multimedia workload. *Multimedia Systems*, 8(4), 2000.

[47] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[48] SPEC Benchmarks. `http://www.spec.org/benchmarks.html`.

[49] StreamIt homepage. `http://cag.csail.mit.edu/streamit`.

[50] J. Teich, E. Zitzler, and S. S. Bhattacharyya. 3D exploration of software schedules for DSP algorithms. In *CODES*, 1999.

[51] D. L. Tennenhouse and V. G. Bose. The SpectrumWare approach to wireless signal processing. *Wireless Networks*, 2(1):1–12, 1996.

[52] W. Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, MIT, 2009.

[53] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *CC*, 2002.

[54] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport messaging for distributed stream programs. In *Principles and Practice of Parallel Programming*, 2005.

[55] A. Udupa, R. Govindarajan, and M. Thazhuthaveetil. Software Pipelined Execution of Stream Programs on GPUs. In *CGO*, 2009.

[56] A. Udupa, R. Govindarajan, and M. Thazhuthaveetil. Synergistic execution of stream programs on multicores with accelerators. In *LCTES*, 2009.

[57] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[58] D. Zhang, Z.-Z. Li, H. Song, and L. Liu. A programming model for an embedded media processing architecture. In *SAMOS*, 2005.