Language and Compiler Support for Stream Programs

by

William Thies

Bachelor of Science, Computer Science and Engineering
Massachusetts Institute of Technology, 2001

Bachelor of Science, Mathematics
Massachusetts Institute of Technology, 2002

Master of Engineering, Computer Science and Engineering
Massachusetts Institute of Technology, 2002

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2009

Author . . . .
Department of Electrical Engineering and Computer Science
January 30, 2009

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

Accepted by . . . .
Terry P. Orlando
Chair, Department Committee on Graduate Students

Language and Compiler Support for Stream Programs

by

William Thies

Submitted to the Department of Electrical Engineering and Computer Science
on January 30, 2009, in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

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. Stream programs can be naturally represented as a graph of independent actors that communicate explicitly over data channels. In this work we focus on programs where the input and output rates of actors are known at compile time, enabling aggressive transformations by the compiler; this model is known as synchronous dataflow.

We develop a new programming language, StreamIt, that empowers both programmers and compiler writers to leverage the unique properties of the streaming domain. StreamIt offers several new abstractions, including hierarchical single-input single-output streams, composable primitives for data reordering, and a mechanism called *teleport messaging* that enables precise event handling in a distributed environment. We demonstrate the feasibility of developing applications in StreamIt via a detailed characterization of our 34,000-line benchmark suite, which spans from MPEG-2 encoding/decoding to GMTI radar processing. We also present a novel dynamic analysis for migrating legacy C programs into a streaming representation.

The central premise of stream programming is that it enables the compiler to perform powerful optimizations. We support this premise by presenting a suite of new transformations. We describe the first translation of stream programs into the compressed domain, enabling programs written for uncompressed data formats to automatically operate directly on compressed data formats (based on LZ77). This technique offers a median speedup of 15x on common video editing operations. We also review other optimizations developed in the StreamIt group, including automatic parallelization (offering an 11x mean speedup on the 16-core Raw machine), optimization of linear computations (offering a 5.5x average speedup on a Pentium 4), and cache-aware scheduling (offering a 3.5x mean speedup on a StrongARM 1100). While these transformations are beyond the reach of compilers for traditional languages such as C, they become tractable given the abundant parallelism and regular communication patterns exposed by the stream programming model.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

I would like to start by expressing my deepest gratitude to my advisor, colleague and friend, Saman Amarasinghe. From 4am phone calls in Boston to weeks of one-on-one time in Sri Lanka and India, Saman invested *unfathomable* time and energy into my development as a researcher and as a person. His extreme creativity, energy, and optimism (not to mention mad PowerPoint skills!) have been a constant source of inspiration, and whenever I am at my best, it is usually because I am asking myself: *What would Saman do*? Saman offered unprecedented freedom for me to pursue diverse interests in graduate school – including weeks at a time working with other groups – and served as a fierce champion on my behalf in every possible way. I will forever treasure our deep sense of shared purpose and can only aspire to impact others as much as he has impacted me.

Contributors to this dissertation Many people made direct contributions to the content of this dissertation. The StreamIt project was a fundamentally collaborative undertaking, involving the extended efforts of over 27 people. I feel very lucky to have been part of such an insightful, dedicated, and fun team. Section 1.4 provides a technical overview of the entire project, including the division of labor. In what follows I am listing only a subset of each person's actual contributions. Michael Gordon, my kindred Ph.D. student throughout the entire StreamIt project, led the development of the parallelization algorithms (summarized in Chapter 4), the Raw backend and countless other aspects of the compiler. Rodric Rabbah championed the project in many capacities, including contributions to cache optimizations (summarized in Chapter 4), teleport messaging (Chapter 3), the MPEG2 benchmarks, an Eclipse interface, and the Cell backend. Michal Karczmarek was instrumental in the original language design (Chapter 2) and teleport messaging, and also implemented the StreamIt scheduler and runtime library. David Maze, Jasper Lin, and Allyn Dimock made sweeping contributions to the compiler infrastructure; I will forever admire their skills and tenacity in making everything work.

Central to the StreamIt project is an exceptional array of M.Eng. students, who I feel very privileged to have interacted with over the years. Andrew Lamb, Sitij Agrawal, and Janis Sermulins led the respective development of linear optimizations, linear statespace optimizations, and cache optimizations (all summarized in Chapter 4). Janis also implemented the cluster backend, with support for teleport messaging (providing results for Chapter 3). Matthew Drake implemented the MPEG2 codec in StreamIt, while Jiawen Chen implemented a flexible graphics pipeline and Basier Aziz implemented mosaic imaging. Daviz Zhang developed a lightweight streaming layer for the Cell processor; Kimberly Kuo developed an Eclipse user interface for StreamIt; Juan Reyes developed a graphical editor for stream graphs; and Jeremy Wong modeled the scalability of stream programs. Kunal Agrawal investigated bit-level optimizations in StreamIt. Ceryen Tan is improving StreamIt's multicore backend.

The StreamIt project also benefited from an outstanding set of undergraduate researchers, who taught me many things. Ali Meli, Chris Leger, Satish Ramaswamy, Matt Brown, and Shirley Fung made important contributions to the StreamIt benchmark suite (detailed in Chapter 2). Steve Hall integrated compressed-domain transformations into the StreamIt compiler (providing results for Chapter 5). Qiuyuan Li worked on a StreamIt backends for Cell, while Phil Sung targeted a GPU.

Individuals from other research groups also impacted the StreamIt project. Members of the Raw group offered incredible support for our experiments, including Anant Agarwal, Michael Taylor, Walter Lee, Jason Miller, Ian Bratt, Jonathan Eastep, David Wentzlaff, Ben Greenwald, Hank Hoffmann, Paul Johnson, Jason Kim, Jim Psota, Nathan Schnidman, and Matthew Frank.

Hank Hoffmann, Nathan Schnidman, and Stephanie Seneff also provided valuable expertise on designing and parallelizing signal processing applications. External contributors to the StreamIt benchmark suite include Ola Johnsson, Mani Narayanan, Magnus Stenemo, Jinwoo Suh, Zain ul-Abdin, and Amy Williams. Fabrice Rastello offered key insights for improving our cache optimizations. Weng-Fai Wong offered guidance on several projects during his visit to the group. StreamIt also benefited immensely from regular and insightful conversations with stakeholders from industry, including Peter Mattson, Richard Lethin, John Chapin, Vanu Bose, and Andy Ong.

Outside of the StreamIt project, additional individuals made direct contributions to this dissertation. In developing our tool for extracting stream parallelism (Chapter 6), I am indebted to Vikram Chandrasekhar for months of tenacious hacking and to Stephen McCamant for help with Valgrind. I thank Jason Ansel, Chen Ding, Ronny Krashinsky, Viktor Kuncak, and Alex Salcianu, who provided valuable feedback on manuscripts that were incorporated into this dissertation. I am also grateful to Arvind and Srini Devadas for serving on my committee on very short notice, and to Marek Olszewski for serving as my remote agent of thesis submission!

The rest of the story Throughout my life, I have been extremely fortunate to have had an amazing set of mentors who invested a lot of themselves in my personal growth. I thank Thomas "Doc" Arnold for taking an interest in a nerdy high school kid, and for setting him loose with chemistry equipment in a Norwegian glacier valley – a tactic which cemented my interest in science, especially the kind you can do while remaining dry. I thank Scott Camazine for taking a chance on a high school programmer in my first taste of academic research, an enriching experience which opened up many doors for me in the future. I thank Vanessa Colella and Mitchel Resnick for making my first UROP experience a very special one, as evidenced by my subsequent addiction to the UROP program. I thank Andrew Begel for teaching me many things, not least of which is by demonstration of his staggering commitment, capacity, and all-around coolness in mentoring undergraduates. I'm especially grateful to Brian Silverman, a mentor and valued friend whose unique perspectives on everything from Life in StarLogo to life on Mars have impacted me more than he might know. I thank Markus Zahn for excellent advice and guidance, both as my undergraduate advisor and UROP supervisor. Finally, I'm very grateful to Kath Knobe, who provided unparalleled mentorship during my summers at Compaq and stimulated my first interest in compilers research.

Graduate school brought a new set of mentors. I learned a great deal from authoring papers or proposals with Anant Agarwal, Srini Devadas, Fredo Durand, Michael Ernst, Todd Thorsen, and Frédéric Vivien, each of whom exemplifies the role of a faculty member in nurturing student talent. I am also very grateful to Srini Devadas, Martin Rinard, Michael Ernst, and Arvind for being especially accessible as counselors, showing interest in my work and well-being even in spite of very busy schedules. I could not have imagined a more supportive environment for graduate study.

I thank Charles Leiserson and Piotr Indyk for teaching me about teaching itself. I will always remember riding the T with Charles when a car full of Red Sox fans asked him what he does for a living. Imagining the impressive spectrum of possible replies, I should not have been surprised when Charles said simply, "I teach". Nothing could be more true, and I feel very privileged to have been a TA in his class.

I'd like to thank my collaborators on projects other than StreamIt, for enabling fulfilling and fun pursuits outside of this dissertation. In the microfluidics lab, I thank J.P. Urbanski for many late nights "chilling at the lab", his euphemism for a recurring process whereby he manufactures chips and I destroy them. His knowledge, determination, and overall good nature are truly inspiring. I also learned a great deal from David Craig, Mats Cooper, Todd Thorsen, and Jeremy Gunawardena, who were extremely supportive of our foray into microfluidics. I thank Nada Amin for her insights, skills, and drive in developing our CAD tool, and for being an absolute pleasure to work with.

I'm very thankful to my collaborators in applying technology towards problems in socio-economic development, from whom I've drawn much support. First and foremost is Manish Bhardwaj, whose rare combination of brilliance, determination, and selflessness has been a deep inspiration to me. I also thank Emma Brunskill, who has been a tremendous collaborator on many fronts, as well as Sara Cinnamon, Goutam Reddy, Somani Patnaik and Pallavi Kaushik for being incredibly talented, dedicated, and fun teammates. I am very grateful to Libby Levison for involving me in my first project at the intersection of technology and development, without which I might have gone down a very different path. I also thank Samidh Chakrabarti for being a great officemate and friend, and my first peer with whom I could investigate this space together.

I am indebted to the many students and staff who worked with me on the TEK project, including Marjorie Cheng, Tazeen Mahtab, Genevieve Cuevas, Damon Berry, Saad Shakhshir, Janelle Prevost, Hongfei Tian, Mark Halsey, and Libby Levison. I also thank Pratik Kotkar, Jonathan Birnbaum, and Matt Aasted for their work on the Audio Wiki. I would not have been able to accomplish nearly as much without the insights, dedication, and hard work of all these individuals.

Graduate school would be nothing if not for paper deadlines, and I feel very lucky to have been down in the trenches with such bright, dependable, and entertaining co-authors. Of people not already cited as such, I thank Marten van Dijk, Blaise Gassend, Andrew Lee, Charles W. O'Donnell, Kari Pulli, Christopher Rhodes, Jeffrey Sheldon, David Wentzlaff, Amy Williams, and Matthias Zwicker for some of the best end-to-end research experiences I could imagine.

Many people made the office a very special place to be. Mary McDavitt is an amazing force for good, serving as my rock and foundation throughout many administrative hurricanes; I can't thank her enough for all of her help, advice, and good cheer over the years. I'm also very grateful to Shireen Yadollahpour, Cornelia Colyer, and Jennifer Tucker, whose helpfulness I will never forget. Special thanks to Michael Vezza, system administrator extraordinaire, for his extreme patience and helpfulness in tending to my every question, and fixing everything that I broke.

I thank all the talented members of the Commit group, and especially the Ph.D. students and staff – Jason Ansel, Derek Bruening, Vikram Chandrasekhar, Gleb Chuvpilo, Allyn Dimock, Michael Gordon, David Maze, Michal Karczmarek, Sam Larsen, Marek Olszewski, Diego Puppin, Rodric Rabbah, Mark Stephenson, Jean Yang, and Qin Zhao. On top of tolerating *way* more than their fair share of StreamIt talks, they offered the best meeting, eating, and traveling company ever. I especially thank Michael Gordon, my officemate and trusted friend, for making 32-G890 one of my favorite places – I'm really going to miss our conversations (and productive silences!)

I'd like to extend special thanks to those who supported me in my job search last spring. I feel very grateful for the thoughtful counsel of dozens of people on the interview trail, and especially to a few individuals (you know who you are) who spent many hours talking to me and advocating on my behalf. This meant a great deal to me. I also thank Kentaro Toyama and others at MSR India for being very flexible with my start date, as the submission of this thesis was gradually postponed!

I am extremely fortunate to have had a wonderful support network to sustain me throughout graduate school. To the handful of close friends who joined me for food, walks around town, or kept in touch from a distance: thank you for seeing me through the thick and thin. I'd like to especially call out to David Wentzlaff, Kunal Agrawal, Michael Gordon and Cat Biddle, who held front-row season tickets to my little world and made it so much better by virtue of being there.

Finally, I would like to thank my amazingly loving and supportive parents, who have always been 100% behind me no matter where I am in life. I dedicate this thesis to them.

Relation to Other Publications[1]

This dissertation is one of two Ph.D. theses that emerged out of the StreamIt project. While this manuscript focuses on the language design, domain-specific optimizations and extraction of stream programs from legacy code, Michael Gordon's Ph.D. thesis is the authoritative reference on compiling StreamIt for parallel architectures[2].

This dissertation alternately extends and summarizes prior publications by the author. Chapters 1 and 2 are significantly more detailed than prior descriptions of the StreamIt language [TKA02, TKG+02, AGK+05] and include an in-depth study of the StreamIt benchmark suite that was later published independently[3]. Chapter 3 subsumes the prior description of teleport messaging [TKS+05], including key changes to the semantics and the first uniprocessor scheduling algorithm. Chapter 4 is a condensed summary of prior publications [GTK+02, LTA03, ATA05, STRA05, GTA06], though with new text that often improves the exposition. Chapter 5 subsumes the prior report on compressed-domain processing [THA07], offering enhanced functionality, performance, and readability; this chapter was later published independently[4]. Chapter 6 is very similar to a recent publication [TCA07]. Some aspects of the author's work on StreamIt are not discussed in this dissertation [KTA03, CGT+05].

Independent publications by other members of the StreamIt group are not covered in this dissertation [KRA05, MDH+06, ZLRA08]. In particular, the case study of implementing MPEG2 in StreamIt provides a nice example-driven exposition of the entire language [MDH+06].

Funding Acknowledgment

This work was funded in part by the National Science Foundation (grants EIA-0071841, CNS-0305453, ACI-0325297), DARPA (grants F29601-01-2-0166, F29601-03-2-0065), the DARPA HPCS program, the MIT Oxygen Alliance, the Gigascale Systems Research Center, Nokia, and a Siebel Scholarship.

[1] This section was updated in June, 2010. The rest of the document remains unchanged.

[2] M. Gordon, *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*, Ph.D. Thesis, Massachusetts Institute of Technology, 2010.

[3] W. Thies and S. Amarasinghe, *An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design*, International Conference on Parallel Architectures and Compilation Techniques (PACT), 2010.

[4] W. Thies, S. Hall and S. Amarasinghe, *Manipulating Lossless Video in the Compressed Domain*, ACM Multimedia, 2009.

Contents

List of Figures

Chapter 1

My Thesis

Incorporating streaming abstractions into the programming language can simultaneously improve both programmability and performance. Programmers are unburdened from providing low-level implementation details, while compilers can perform parallelization and optimization tasks that were previously beyond the reach of automation.

1.1 Introduction

A long-term goal of the computer science community has been to automate the optimization of programs via systematic transformations in the compiler. However, even after decades of research, there often remains a large gap between the performance of compiled code and the performance that an expert can achieve by hand. One of the central difficulties is that humans have more information than the compiler, and can thus perform more aggressive transformations. For example, a performance expert may re-write large sections of the application, employing alternative algorithms, data structures, or task decompositions, to produce a version that is functionally equivalent to (but structurally very different from) the original program. In addition, a performance expert may leverage detailed knowledge of the target architecture – such as the type and extent of parallel resources, the communication substrate, and the cache sizes – to match the structure and granularity of the application to that of the underlying machine. To overcome this long-standing problem and make high-performance programming accessible to non-experts, it will likely be necessary to empower the compiler with new information that is not currently embedded in general-purpose programming languages.

One promising approach to automating performance optimization is to embed specific domain knowledge into the language and compiler. By restricting attention to a specific class of programs, common patterns in the applications can be embedded in the language, allowing the compiler to easily recognize and optimize them, rather than having to infer the patterns from complex, general-purpose code. In addition, key transformations that are known only to experts in the domain can be embedded in the compiler, enabling robust performance for a given application class. Tailoring the language to a specific domain can also improve programmers' lives, as functionality that is tedious or unnatural to express in a general-purpose language can be succinctly expressed in a new one. Such "domain-specific" languages and compilers have achieved broad success in the past. Examples include Lex for generating scanners; YACC for generating parsers; SQL for database queries; Verilog and VHDL for hardware design; MATLAB for scientific codes and signal processing;

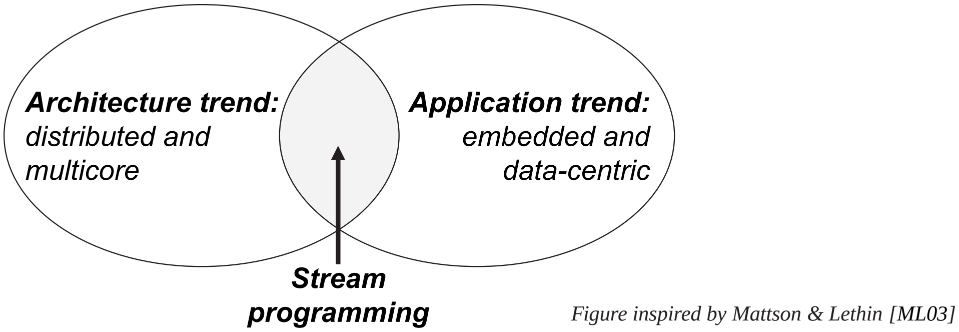


Figure 1-1: Stream programming is motivated by two prominent trends: the trend towards parallel computer architectures, and the trend towards embedded, data-centric computations.

GraphViz for generating graph layouts [EGK+02]; and PADS for processing ad-hoc data [FG05].

If we are taking a domain-specific approach to program optimization, what domain should we focus on to have a long-term impact? We approached this question by considering two prominent trends in computer architectures and applications:

1. **Computer architectures are becoming multicore.** Because single-threaded performance has finally plateaued, computer vendors are investing excess transistors in building more cores on a single chip rather than increasing the performance of a single core. While Moore's Law previously implied a transparent doubling of computer performance every 18 months, in the future it will imply only a doubling of the number of cores on chip. To support this trend, a high-performance programming model needs to expose all of the parallelism in the application, supporting explicit communication between potentially-distributed memories.

2. **Computer applications are becoming embedded and data-centric.** While desktop computers have been a traditional focus of the software industry, the explosion of cell phones is shifting this focus to the embedded space. There are almost four billion cell phones globally, compared to 800 million PCs [Med08]. Also, the compute-intensive nature of scientific and simulation codes is giving way to the data-intensive nature of audio and video processing. Since 2006, YouTube has been streaming over 250 terabytes of video daily [Wat06], and many potential "killer apps" of the future encompass the space of multimedia editing, computer vision, and real-time audio enhancement [ABC+06, CCD+08].

At the intersection of these trends is a broad and interesting space of applications that we term *stream programs*. A stream program is any program that is based around a regular stream of dataflow, as in audio, video, and signal processing applications (see Figure 1-2). Examples include radar tracking, software radios, communication protocols, speech coders, audio beamforming, video processing, cryptographic kernels, and network processing. These programs are rich in parallelism and can be naturally targeted to distributed and multicore architectures. At the same time, they also share common patterns of processing that makes them an ideal target for domain-specific optimizations.

In this dissertation, we develop language support for stream programs that enables non-expert programmers to harness both avenues: parallelism and domain-specific optimizations. Either set

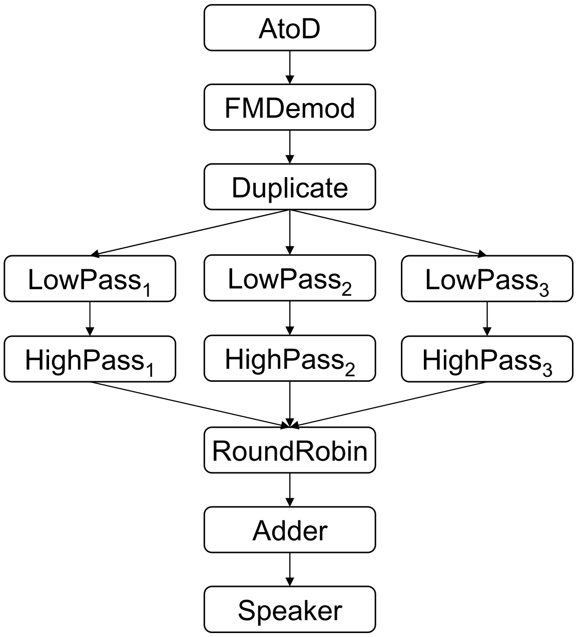


Figure 1-2: Example stream graph for a software radio with equalizer.

of optimizations can yield order-of-magnitude performance improvements. While the techniques used were previously accessible to experts during manual performance tuning, we provide the first general and automatic formulation. This greatly lowers the entry barrier to high-performance stream programming.

In the rest of this chapter, we describe the detailed properties of stream programs, provide a brief history of streaming, give an overview of the StreamIt project, and state the contributions of this dissertation.

1.2 Streaming Application Domain

Based on the examples cited previously, we have observed that stream programs share a number of characteristics. Taken together, they define our conception of the streaming application domain:

1. **Large streams of data.** Perhaps the most fundamental aspect of a stream program is that it operates on a large (or virtually infinite) sequence of data items, hereafter referred to as a *data stream*. Data streams generally enter the program from some external source, and each data item is processed for a limited time before being discarded. This is in contrast to scientific codes, which manipulate a fixed input set with a large degree of data reuse.

2. **Independent stream filters.** Conceptually, a streaming computation represents a sequence of transformations on the data streams in the program. We will refer to the basic unit of this transformation as a *filter*: an operation that – on each execution step – reads one or more items from an input stream, performs some computation, and writes one or more items to an output stream. Filters are generally independent and self-contained, without references to

global variables or other filters. A stream program is the composition of filters into a *stream graph*, in which the outputs of some filters are connected to the inputs of others.

3. **A stable computation pattern.** The structure of the stream graph is generally constant during the steady-state operation of the program. That is, a certain set of filters are repeatedly applied in a regular, predictable order to produce an output stream that is a given function of the input stream.

4. **Sliding window computations.** Each value in a data stream is often inspected by consecutive execution steps of the same filter, a pattern referred to as a *sliding window*. Examples of sliding windows include FIR and IIR filters; moving averages and differences; error correcting codes; biosequence analysis; natural language processing; image processing (sharpen, blur, etc.); motion estimation; and network packet inspection.

5. **Occasional modification of stream structure.** Even though each arrangement of filters is executed for a long time, there are still dynamic modifications to the stream graph that occur on occasion. For instance, if a wireless network interface is experiencing high noise on an input channel, it might react by adding some filters to clean up the signal; or it might re-initialize a sub-graph when the network protocol changes from 802.11 to Bluetooth[1].

6. **Occasional out-of-stream communication.** In addition to the high-volume data streams passing from one filter to another, filters also communicate small amounts of control information on an infrequent and irregular basis. Examples include changing the volume on a cell phone, printing an error message to a screen, or changing a coefficient in an adaptive FIR filter. These messages are often synchronized with some data in the stream–for instance, when a frequency-hopping radio changes frequencies at a specific point of the transmission.

7. **High performance expectations.** Often there are real-time constraints that must be satisfied by stream programs; thus, efficiency (in terms of both latency and throughput) is of primary concern. Additionally, many embedded applications are intended for mobile environments where power consumption, memory requirements, and code size are also important.

While our discussion thus far has emphasized the embedded context for streaming applications, the stream abstraction is equally important in desktop and server-based computing. Examples include XML processing [BCG$^+$03], digital filmmaking, cell phone base stations, and hyperspectral imaging.

1.3 Brief History of Streaming

The concept of a stream has a long history in computer science; see Stephens [Ste97] for a review of its role in programming languages. Figure 1-3 depicts some of the notable events on a timeline, including models of computation and prototyping environments that relate to streams.

The fundamental properties of streaming systems transcend the details of any particular language, and have been explored most deeply as abstract models of computation. These models can

[1] In this dissertation, we do not explore the implications of dynamically modifying the stream structure. As detailed in Chapter 2, we operate on a single stream graph at a time; users may transition between graphs using wrapper code.

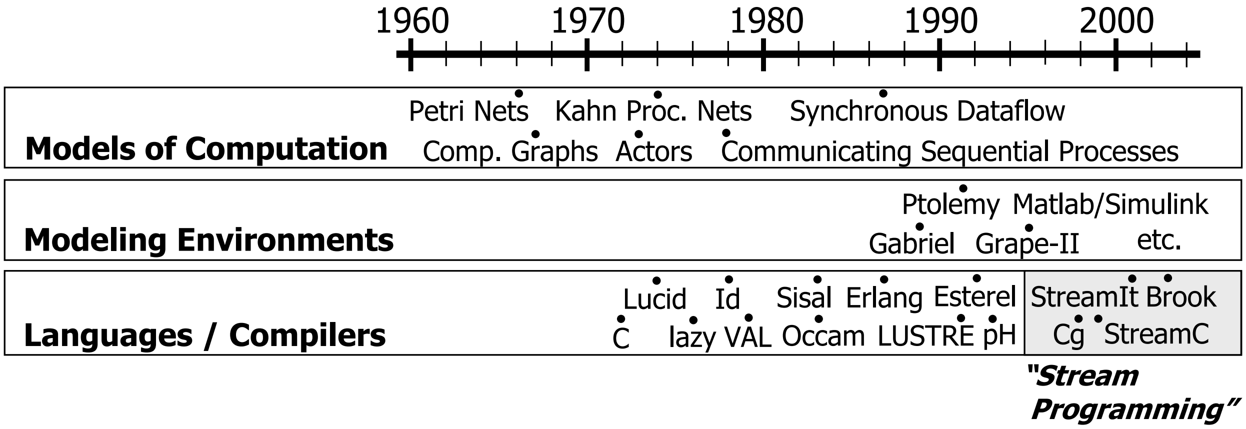


Figure 1-3: Timeline of computer science efforts that have incorporated notions of streams.

generally be considered as graphs, where nodes represent units of computation and edges represent FIFO communication channels. The models differ in the regularity and determinism of the communication pattern, as well as the amount of buffering allowed on the channels. Three of the most prevalent models are Kahn Process Networks [Kah74], Synchronous Dataflow [LM87], and Communicating Sequential Processes [Hoa78]. They are compared in Figure 1-4 and Table 1-5 and are detailed below:

1. **Kahn Process Networks**, also known as *process networks,* are a simple model in which nodes can always enqueue items onto their output channels, but attempts to read from an input channel will block until data is ready (it is not possible to test for the presence or absence of data on input channels). Assuming that each node performs a deterministic computation, these properties imply that the entire network is deterministic; that is, the sequence of data items observed on the output channels is a deterministic function of those submitted to the input channels. However, it is undecidable to statically determine the amount of buffering needed on the channels, or to check whether the computation might deadlock. Process networks is the model of computation adopted by UNIX pipes.

2. **Synchronous Dataflow (SDF)** is a restricted form of process network in which nodes execute atomic steps, and the numbers of data items produced and consumed during each step are constant and known at compile time. Because the input and output rates are known, the compiler can statically check whether a deadlock-free execution exists and, if so, can derive a valid schedule of node executions. The amount of buffering needed can also be determined statically. Many variations of synchronous dataflow have been defined, including cyclo-static dataflow [BELP95, PPL95] and multidimensional synchronous dataflow [ML02]. Due to its potential for static scheduling and optimization, synchronous dataflow provides the starting point for our work.

3. **Communicating Sequential Processes (CSP)** is in some ways more restrictive, and in others more flexible than process networks. The restriction is rendezvous communication: there is no buffering on the communication channels, which means that each send statement blocks

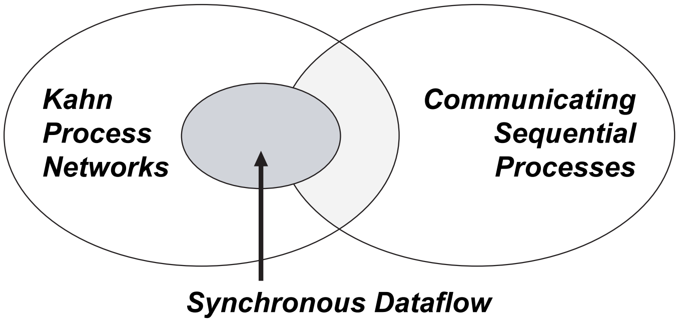


Figure 1-4: Space of behaviors allowed by Kahn process networks, synchronous dataflow (SDF), and communicating sequential processes (CSP). The set of behaviors considered are: buffering data items on a communication channel, making a non-deterministic choice, and deviating from a fixed I/O rate. While these behaviors could always be emulated inside a single Turing-complete node, we focus on the behavior of the overall graph.

	Communication Pattern	**Buffering**	**Notes**
Kahn process networks	Data-dependent, but deterministic	Conceptually unbounded	- UNIX pipes
Synchronous dataflow	Static	Fixed by compiler	- Static scheduling - Deadlock freedom
Communicating Sequential Processes	Data-dependent, allows non-determinism	None (Rendezvous)	- Rich synchronization primitives - Occam language

Table 1-5: Key properties of streaming models of computation.

until being paired with a receive statement (and vice-versa). The flexibility is in the synchronization: a node may make a non-deterministic choice, for example, in reading from any input channel that has data available under the current schedule. This property leads to possibly non-deterministic outputs, and deadlock freedom is undecidable. While CSP itself is an even richer algebra for describing the evolution of concurrent systems, this graph-based interpretation of its capabilities applies to many of its practical instantiations, including the Occam programming language.

While long-running (or infinite) computations are often described using one of the models above, the descriptions of finite systems also rely on other models of computation. *Computation graphs* [KM66] are a generalization of SDF graphs; nodes may stipulate that, in order to execute a step, a threshold number of data items must be available on an input channel (possibly exceeding the number of items consumed by the node). While SDF scheduling results can be adapted to the infinite execution of computation graphs, the original theory of computation graphs focuses on determinacy and termination properties when some of the inputs are finite. *Actors* are a more general model providing asynchronous and unordered message passing between composable, dynamically-created nodes [HBS73, Gre75, Cli81, Agh85]. Petri nets are also a general formalism for modeling many classes of concurrent transition systems [Pet62, Mur89].

In addition to models of computation, prototyping environments have been influential in the history of streaming systems. The role of a prototyping environment is to simulate and validate the design of a complex system. While it has been a long-standing goal to automatically generate efficient and deployable code from the prototype design, in practice most models are re-written by hand in a low-level language such as C in order to gain the full performance and functionality needed. Still, many graph-level optimizations, such as scheduling [BML95, BML96, BSL96, ZTB00, SGB06] and buffer management [ALP97, MB01, GGD02, MB04, GBS05], have been pioneered in the context of prototyping environments. One of the most long-running efforts is Ptolemy, a rich heterogeneous modeling environment that supports diverse models of computation, including the dataflow models described previously as well as continuous- and discrete-event systems [BHLM91, EJL$^+$03]. Other academic environments include Gabriel [LHG$^+$89],Grape-II [LEAP95], and El Greco [BV00], while commercial environments include MATLAB/Simulink (from The MathWorks, Inc.), Cocentric System Studio (from Synposis, Inc.), and System Canvas (from Angeles Design Systems [MCR01]).

The notion of streams has also permeated several programming paradigms over the past half-century. Dataflow languages such as Lucid [AW77], Id [AGP78, Nik91], and VAL [AD79] aim to eliminate traditional control flow in favor of a scheduler that is driven only by data dependences. To expose these dependences, each variable is assigned only once, and each statement is devoid of non-local side effects. These properties overlap streaming in two ways. First, the producer/consumer relationships exposed by dataflow are similar to those in a stream graph, except at a much finer level of granularity. Second, to preserve the single-assignment property within loops, these languages use a *next* keyword to indicate the value of a variable in the succeeding iteration. This construct can be viewed as a regular data stream of values flowing through the loop. Subsequent dataflow languages include Sisal [MSA$^+$85] ("streams and iteration in a single assignment language") and pH [NA01]. More details on dataflow languages are available in review articles [Ste97, JHM04].

Functional languages also have notions of streams, for example, as part of the lazy evaluation of lists [HM76]. It bears noting that there seems to be no fundamental difference between a "functional language" and a "dataflow language". The terminology indicates mainly a difference of community, as dataflow languages were mapped to dataflow machines. In addition, dataflow languages may be more inclined toward an imperative syntax [JHM04]. We do not survey functional languages further in their own right.

Another class of languages is synchronous languages, which offer the abstraction of responding instantly (in synchrony) with their environment [Hal98]. Interpreted as a stream graph, a synchronous program can be thought of as a circuit, where nodes are state machines and edges are wires that carry a single value; in some languages, nodes may specify the logical times at which values are present on the wires. Synchronous programs target the class of reactive systems, such as control circuits, embedded devices, and communication protocols, where the computation is akin to a complex automaton that continually responds to real-time events. Compared to stream programs, which have regular, computation-intensive processing, synchronous programs process irregular events and demand complex control flow. Key synchronous languages include Signal [GBBG86], LUSTRE [CPHP87, HCRP91], Esterel [BG92], Argos [MR01], and Lucid Synchrone [CP95, CHP07]. These languages offer determinism and safety properties, spurring their adoption in industry; Esterel Technologies offers SCADE (based on LUSTRE) as well as Esterel Studio.

There have also been general-purpose languages with built-in support for streams, including Occam [Inm88] and Erlang [AVW93, Arm07]. Occam is an imperative procedural language that is based on communicating sequential processes; it was originally developed for the INMOS transputer, an early multiprocessor. Erlang is a functional language that is based on the actors model; originally developed by Ericsson for distributed applications, it supports very lightweight processes and has found broad application in industry.

Shortcomings of previous languages. It should be evident that previous languages have provided many variations on streams, including many elegant and general ways of expressing the functional essence of streaming computations. However, there remains a critical disconnect in the design flow for streaming systems: while prototyping environments provide rich, high-level analysis and optimization of stream graphs [BML95, BML96, BSL96, ALP97, ZTB00, MB01, GGD02, MB04, GBS05, SGB06], these optimizations have not been automated in any programming language environment and thus remain out-of-reach for the vast majority of developers. The root of the disconnect is the model of computation: previous languages have opted for the flexibility of process networks or communication sequential processes, rather than embracing the restrictive yet widely-applicable model of synchronous dataflow. By focusing attention on a very common case – an unchanging stream graph with fixed communication rates – synchronous dataflow is the only model of computation that exposes the information needed to perform static scheduling and optimization.

Herein lies a unique opportunity to create a language that exposes the inherent regularity in stream programs, and to exploit that regularity to perform deep optimizations. This is the opportunity that we pursue in the StreamIt project.

1.4 The StreamIt Project

StreamIt is a language and compiler for high-performance stream programs. The principal goals of the StreamIt project are three-fold:

1. To expose and exploit the inherent parallelism in stream programs on multicore architectures.

2. To automate domain-specific optimizations known to streaming application specialists.

3. To improve programmer productivity in the streaming domain.

While the first two goals are related to performance improvements, the third relates to improving programmability. We contend that these goals are not in conflict, as high-level programming models contain more information and can be easier to optimize while also being easier to use. However, many languages are designed only from the standpoint of programmability, and often make needless sacrifices in terms of analyzability. Compared to previous efforts, the key leverage of the StreamIt project is a *compiler-conscious language design* that maximizes both analyzability and programmability.

StreamIt is a large systems project, incorporating over 27 people (up to 12 at a given time). The group has made several contributions to the goals highlighted above. In exploiting parallelism, Michael Gordon led the development of the first general algorithm that exploits the task, data, and pipeline parallelism in stream programs [GTA06]. On the 16-core Raw architecture, it achieves an 11x mean speedup for our benchmark suite; this is a 1.8x improvement over our previous approach [GTK$^+$02, Gor02], the performance of which had been modeled by Jeremy Wong [Won04]. Also on the Raw machine, Jiawen Chen led the development of a flexible graphics rendering pipeline in StreamIt, demonstrating that a reconfigurable pipeline can achieve up to twice the throughput of a static one [CGT$^+$05, Che05]. Moving beyond Raw, Janis Sermulins ported StreamIt to multicores and clusters of workstations, and also led the development of cache optimizations that offer a 3.5x improvement over unoptimized StreamIt on embedded processors [STRA05, Ser05]. David Zhang and Qiuyuan Li led the development of a lightweight streaming execution layer that achieves over 88% utilization (ignoring SIMD potential) on the Cell processor [ZLRA08, Zha07]. Michal Karczmarek led the development of phased scheduling, the first hierarchical scheduler for cyclo-static dataflow that also enables a flexible tradeoff between latency, code size and buffer size [KTA03, Kar02]. Phil Sung and Weng-Fai Wong explored the execution of StreamIt on graphics processors.

In the area of domain-specific optimizations, Andrew Lamb automated the optimization of linear nodes, performing coarse-grained algebraic simplification or automatic conversion to the frequency domain [LTA03, Lam03]. These inter-node optimizations eliminate 87% of the floating point operations from code written in a modular style. Sitij Agrawal generalized the analysis to handle linear computations with state, performing optimizations such as algebraic simplification, removal of redundant states, and reducing the number of parameters [ATA05, Agr04]. I led the development of domain-specific optimizations for compressed data formats, allowing certain computations to operate in place on the compressed data without requiring decompression and re-compression [THA07]. This transformation accelerates lossless video editing operations by a median of 15x.

In the area of programmability, I led the definition of the StreamIt language, incorporating the first notions of structured streams as well as language support for hierarchical data reordering [TKA02, TKG$^+$02, AGK$^+$05]. With Michal Karczmarek and Janis Sermulins, I led the development of teleport messaging, a new language construct that uses the flow of data in the stream to provide a deterministic and meaningful timeframe for delivering events between decoupled nodes [TKS$^+$05]. Kimberly Kuo developed an Eclipse development environment and debugger for StreamIt and, with Rodric Rabbah, demonstrated improved programming outcomes in a user study [KRA05, Kuo04]. Juan Reyes also developed a graphical editor for StreamIt [Rey04]. Matthew Drake evaluated StreamIt's suitability for video codecs by implementing an MPEG-2 encoder and decoder [MDH$^+$06, Dra06]. Basier Aziz evaluated StreamIt by implementing image-based motion estimation, including the RANSAC algorithm [Azi07]. I also led the development of dynamic analysis tools to ease the translation of legacy C code into a streaming representation [TCA07].

The StreamIt benchmark suite consists of 67 programs and 34,000 (non-comment, non-blank) lines of code. I provide the first rigorous characterization of the benchmarks as part of this dissertation. In addition to MPEG-2 and image-based motion estimation, the suite includes a ground moving target indicator (GMTI), a feature-aided tracker (FAT), synthetic aperture radar (SAR), a radar array front-end, part of the 3GPP physical layer, a vocoder with speech transformation, a subset of an MP3 decoder, a subset of MPEG-4 decoder, a JPEG encoder and decoder, a GSM decoder, an FM software radio, DES and serpent encryption, matrix multiplication, graphics shaders and rendering algorithms, and various DCTs, FFTs, filterbanks, and sorting algorithms. The StreamIt benchmark suite has been used by outside researchers [KM08]. Some programs are currently restricted for internal use.

Contributors to the benchmark suite include Sitij Agrawal, Basier Aziz, Matt Brown, Jiawen Chen, Matthew Drake, Shirley Fung, Michael Gordon, Hank Hoffmann, Ola Johnsson, Michal Karczmarek, Andrew Lamb, Chris Leger, David Maze, Ali Meli, Mani Narayanan, Rodric Rabbah, Satish Ramaswamy, Janis Sermulins, Magnus Stenemo, Jinwoo Suh, Zain ul-Abdin, Amy Williams, Jeremy Wong, and myself. Individual contributions are detailed in Table 2-10.

While this section was not intended to serve as the acknowledgments, it would be incomplete without noting the deep involvement, guidance, and supervision of Rodric Rabbah and Saman Amarasinghe throughout many of the efforts listed above. The StreamIt infrastructure was also made possible by the tenacious and tireless efforts of David Maze, Jasper Lin, and Allyn Dimock. Additional contributors that were not mentioned previously include Kunal Agrawal (who developed bit-level analyses), Steve Hall (who automated compressed-domain transformations), and Ceryen Tan (who is improving the multicore backend).

The StreamIt compiler (targeting shared-memory multicores, clusters of workstations, and the MIT Raw machine) is publicly available [Stra] and has logged over 850 unique, registered downloads from 300 institutions (as of December, 2008). Researchers at other universities have used StreamIt as a basis for their own work [NY04, Duc04, SLRBE05, JSuA05, And07, So07].

This dissertation does not mark the culmination of the StreamIt project; please consult the StreamIt website [Stra] for subsequent updates.

1.5 Contributions

My role in the StreamIt project has been very collaborative, contributing ideas and implementation support to many aspects of the project. This dissertation focuses on ideas that have not been presented previously in theses by other group members. However, to provide a self-contained view of the breadth and applications of StreamIt, Chapter 4 also provides a survey of others' experience in optimizing the language.

The specific contributions of this dissertation are as follows:

1. **A design rationale and experience report for the StreamIt language, which contains novel constructs to simultaneously improve the programmability and analyzability of stream programs (Chapter 2).** StreamIt is the first language to introduce structured streams, as well as hierarchical, parameterized data reordering. We evaluate the language via a detailed characterization of our 34,000-line benchmark suite, illustrating the impact of each language feature as well as the lessons learned.

2. **A new language construct, termed teleport messaging, that enables precise event handling in a distributed environment (Chapter 3).** Teleport messaging is a general approach that uses the flow of data in the stream to provide a deterministic and meaningful timeframe for delivering events between decoupled nodes. Teleport messaging allows irregular control messages to be integrated into a synchronous dataflow program while preserving static scheduling.

3. **A review of the key results in optimizing StreamIt, spanning parallelization and domain-specific optimization (Chapter 4)**. This chapter validates key concepts of the StreamIt language by highlighting the gains in performance and programmability that have been achieved, including the work of others in the StreamIt group. We focus on automatic parallelization (providing an 11x mean speedup on a 16-core machine), domain-specific optimization of linear computations (providing a 5.5x average speedup on a uniprocessor), and cache optimizations (providing a 3.5x average speedup on an embedded processor).

4. **The first translation of stream programs into the lossless-compressed domain (Chapter 5)**. This domain-specific optimization allows stream programs to operate directly on compressed data formats, rather than requiring conversion to an uncompressed format prior to processing. While previous researchers have focused on compressed-domain techniques for lossy data formats, there are few techniques that apply to lossless formats. We focus on applications in video editing, where our technique supports color adjustment, video compositing, and other operations directly on the Apple Animation format (a variant of LZ77). Speedups are roughly proportional to the compression factor, with a median of 15x and a maximum of 471x.

5. **The first dynamic analysis tool that detects and exploits likely coarse-grained parallelism in C programs (Chapter 6).** To assist programmers in migrating legacy C code into a streaming representation, this tool generates a stream graph depicting dynamic communication between programmer-annotated sections of code. The tool can also generate a parallel version of the program based on the memory dependences observed during training runs. In our experience with six case studies, the extracted stream graphs prove useful and the parallelized versions offer a 2.78x speedup on a 4-core machine.

Related work and future work are presented on a per-chapter basis. We conclude in Chapter 7.

Chapter 2

The StreamIt Language

This chapter provides an overview and experience report on the basics of the StreamIt language. An advanced feature, teleport messaging, is reserved for Chapter 3. For more details on the StreamIt language, please consult the StreamIt language specification [Strc] or the StreamIt cookbook [Strb]. A case study on MPEG-2 also provides excellent examples of the language's capabilities [MDH$^+$06].

2.1 Model of Computation

The model of computation in StreamIt is rooted in (but not equivalent to) synchronous dataflow [LM87]. As described in Chapter 1, synchronous dataflow represents a program as a graph of independent nodes, or *actors*, that communicate over FIFO data channels. Each actor has an atomic execution step that is called repeatedly by the runtime system. The key aspect of synchronous dataflow, as opposed to other models of computation, is that the number of items produced and consumed by an actor on each execution is fixed and known at compile-time. This allows the compiler to perform static scheduling and optimization of the stream graph.

StreamIt differs from synchronous dataflow in five respects:

1. **Multiple execution steps.** Certain pre-defined actors have more than one execution step; the steps are called repeatedly, in a cyclic fashion, by the runtime system. This execution model mirrors cyclo-static dataflow [BELP95, PPL95]. The actors that follow this model are *splitters* and *joiners*, which scatter and gather data across multiple streams. (While the language once supported multiple execution steps for user-programmable actors as well, the benefits did not merit the corresponding confusion experienced by programmers.)

2. **Dynamic rates.** The input and output rates of actors may optionally be declared to be dynamic. A dynamic rate indicates that the actor will produce or consume an unpredictable number of data items that is known only at runtime. Dynamic rates are declared as a range (min, max, and a hint at the average), with any or all of the elements designated as "unknown". While most of our optimizations in StreamIt have focused on groups of static-rate actors, we have runtime support for dynamic rates (as demanded by applications such as MPEG-2 [MDH$^+$06]).

3. **Teleport messaging.** Our support for irregular, out-of-band control messaging falls outside of the traditional synchronous dataflow model. However, it does not impede static scheduling. See Chapter 3 for details.

4. **Peeking.** StreamIt allows actors to "peek" at data items on their input tapes, reading a value without dequeuing it from the channel. Peeking is important for expressing sliding window computations. To support peeking, two stages of scheduling are required: an initialization schedule that grows buffers until they accumulate a threshold number of peeked items, and a steady-state schedule that preserves the size of the buffers over time. While peeking can be represented as edge-wise "delays" in the original nomenclature of synchronous dataflow [LM87], most of the scheduling and optimization research on synchronous dataflow does not consider the implications of these delays.

5. **Communication during initialization.** StreamIt allows actors to input and output a known number of data items during their initialization (as part of the *prework* function). This communication is also incorporated into the initialization schedule.

With the basic computational model in hand, the rest of this section describes how StreamIt specifies the computation within actors as well as the connectivity of the stream graph.

2.2 Filters

The basic programmable unit in StreamIt is called a *filter*. It represents a user-defined actor with a single input channel and single output channel. Each filter has a private and independent address space; all communication between filters is via the input and output channels (and teleport messaging). Filters are also granted read-only access to global constants.

An example filter appears in Figure 2-1. It performs an FIR filter, which is parameterized by a length *N*. Each filter has two stages of execution: initialization and steady state. During initialization, the parameters to a filter are resolved to constants and the *init* function is called. In the case of FIR, the init function initializes an array of weights, which is maintained as state within the filter. During steady state execution, the *work* function is called repeatedly. Inside of work, filters can *push* items to the output channel, *pop* items from the input channel, or *peek* at a given position on the input channel. Filters requiring different behavior on their first execution can declare a *prework* function, which is called once between *init* and *work*.

The work and prework functions declare how many items they will push and pop, and the maximum number of items they might peek, as part of their declarations. To benefit from static scheduling, these expressions must be resolvable to constants at compile time (dynamic rates are declared using a different syntax [Strc]). While a static analysis can infer the input and output rates in most cases, in general the problem is undecidable. Our experience has been that rate declarations provide valuable documentation on the behavior of the filter. In cases where the rates can be inferred, the declarations can be checked by the compiler.

The StreamIt version of the FIR filter is easy to parallelize and optimize. Because there is no mutable state within the filter (that is, the *weights* array is modified only during initialization), the compiler can exploit data parallelism and instantiate many copies of the FIR filter, each operating on different sections of the input tape. Also, due to a lack of pointers in the language, values can easily be traced through arrays from their initialization to their use. This allows the compiler to infer that the FIR filter computes a linear function, subject to aggressive optimization [LTA03]. Also, using a transformation called scalar replacement [STRA05], the *weights* array can be eliminated completely by unrolling loops and propagating constants from the init function to the work function.

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

  init {
    for (int i=0; i<N; i++) {
      weights[i] = calcWeight(i, N);
    }
  }

  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 2-1: FIR filter in StreamIt.

```
void init_FIR(float* weights, int N) {
  int i;

  for (i=0; i<N; i++) {
    weights[i] = calc_weight(i, N);
  }
}

void do_FIR(float* weights, int N,
            int* src, int* dest,
            int* srcIndex, int* destIndex,
            int srcBufferSize, int destBufferSize) {

  float sum = 0.0;
  for (int i = 0; i < N; i++) {
    sum += weights[i] *
           src[(*srcIndex + i) % srcBufferSize];
  }
  dest[*destIndex] = sum;
  *srcIndex = (*srcIndex + 1) % srcBufferSize;
  *destIndex = (*destIndex + 1) % destBufferSize;
}
```

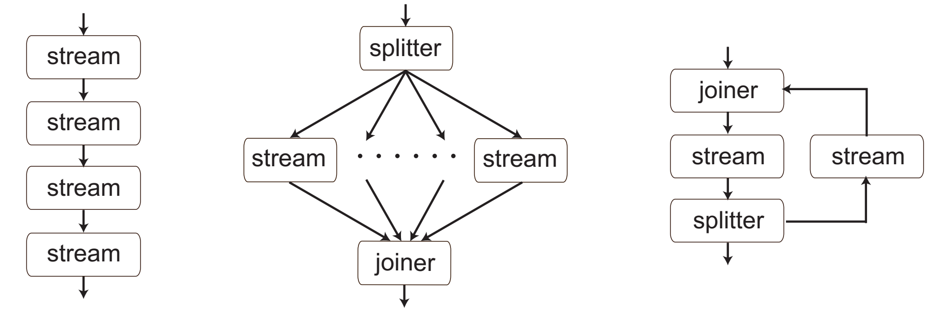
Figure 2-2: FIR filter in C.

A traditional C implementation of an FIR filter (shown in Figure 2-2) resists parallelization and optimization. The sliding-window nature of the FIR computation results in a circular buffer, where elements are addressed using a modulo operation. Modulo operations are very difficult to analyze in a compiler; rather than recognizing the underlying FIFO queue, conservative compilers will regard each read and write as falling anywhere in an array. The problems are confounded by the presence of pointers. To parallelize calls to do_FIR, compilers would need to prove that the *weights* and *src* arrays did not overlap with *dest*, *srcIndex*, or *destIndex*. Similar analysis would be needed to track the values of *weights* from their initialization to their use (in two different procedures). Such precise alias analyses are usually beyond reach. Worse still, it might not even be legal to call *do_FIR* in parallel, depending on the buffer sizes chosen by the programmer. The underlying cause of all these obstacles is that the programmer has over-specified the computation, imposing a scheduling and buffer management policy that is better decided by the compiler.

Despite its simplicity, this example illustrates the potential of improving both the programmability and analyzability of stream programs via a domain-specific language design. In addition to exposing the right information to the compiler, the StreamIt version is also shorter and easier to maintain, representing a win/win situation for both man and machine.

2.3 Stream Graphs

One of the new and experimental ideas in StreamIt is to enforce a *structured* programming model when building stream graphs. Rather than allowing programmers to connect filters into arbitrary graphs, the language provides three hierarchical primitives for building larger graphs out of smaller ones. As illustrated in Figure 2-3, these structures are a pipeline, a splitjoin, and a feedbackloop. Like a filter, each stream structure has a single input channel and single output channel, allowing them to be composed and interchanged freely. We collectively refer to filters and stream structures as *streams*.



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

Figure 2-3: Hierarchical stream structures in StreamIt.

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

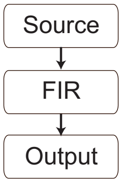
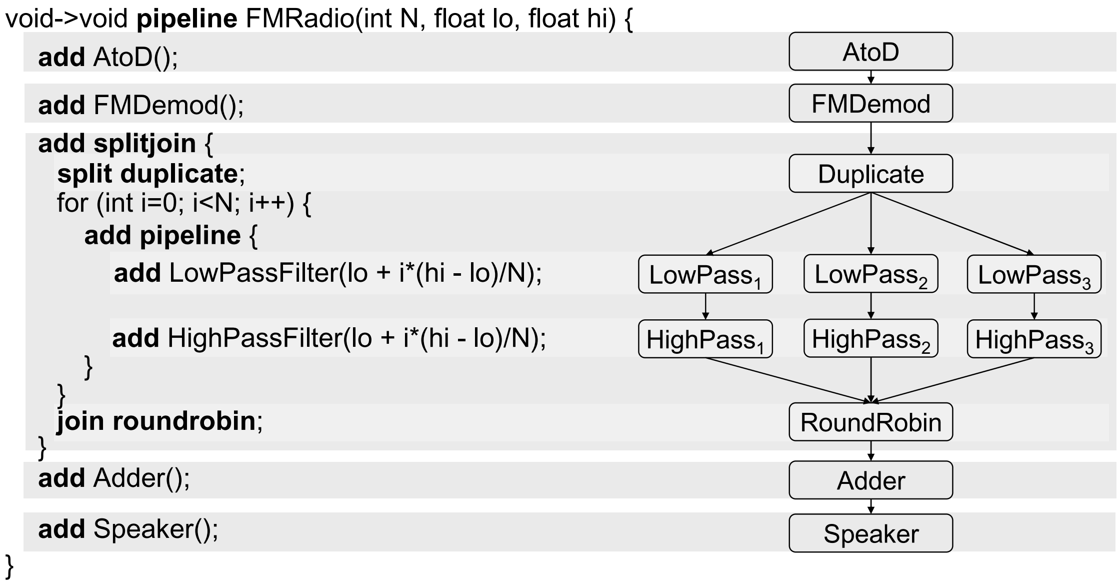


Figure 2-4: Example pipeline with FIR filter.

The pipeline structure represents a serial composition of streams, with the output of one stream flowing to the input of the next. Figure 2-4 illustrates the syntax for pipelines; the *add* keyword indicates that a new stream should be instantiated and appended to the current pipeline. A splitjoin represents a set of parallel and independent streams; a *splitter* distributes data from the input channel to the parallel components, while a *joiner* interleaves the streams' results onto the output channel. In this case, each call to *add* specifies a separate parallel stream (see Figure 2-5). The language provides a fixed set of pre-defined splitters and joiners, encompassing duplication and round-robin behavior (detailed in the next section). Finally, the feedbackloop structure provides a way to induce cycles in the stream graph.

The motivations for introducing structured dataflow in a stream language are analogous to those for introducing structured control flow in an imperative language. While there was once a debate [Dij68] between unstructured control flow (using GOTO statements) and structured control flow (using if/then/else and for loops), in the end structured control flow came to dominate because it allows the programmer to reason locally. Rather than being lost in a sea of "spaghetti code", programmers can recognize common patterns because the language enforces a canonical and hierarchical expression of the control. While skeptics once argued that certain patterns would be more naturally expressed using GOTO statements, over time there emerged structured idioms that were equally recognizable. For example, while a state machine can be written using a GOTO statement for each state transition, it can also be written as a dispatch loop. Structured control flow also benefited compilers, because non-sensical control flow graphs could be ruled out in favor of the common case. The field of loop optimizations would have been much more difficult to develop if researchers had to cope with the full complexity of an unstructured programming model.



```
void->void pipeline FMRadio(int N, float lo, float hi) {
  add AtoD();
  add FMDemod();
  add splitjoin {
    split duplicate;
    for (int i=0; i<N; i++) {
      add pipeline {
        add LowPassFilter(lo + i*(hi - lo)/N);
        add HighPassFilter(lo + i*(hi - lo)/N);
      }
    }
    join roundrobin;
  }
  add Adder();
  add Speaker();
}
```

Figure 2-5: Example of a software radio with equalizer. There is a natural correspondence between the structure of the code and the structure of the graph. In the code, stream structures can be lexically nested to provide a concise description of the application.

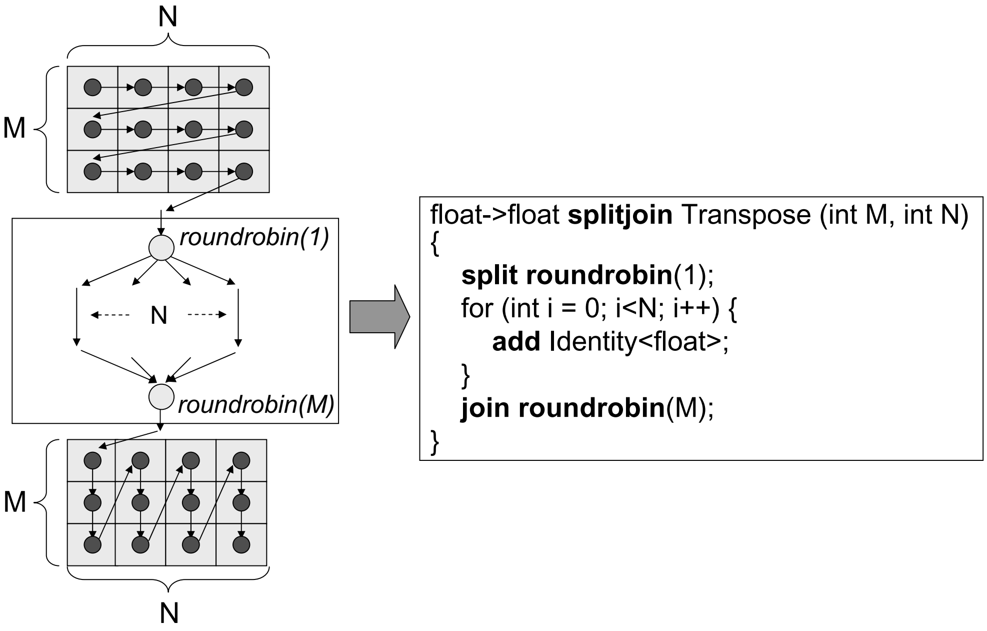
We believe that imposing structure on a stream graph can offer similar benefits. From the programmer's perspective, structured streams offer a disciplined and readable way to describe, parameterize, and compose stream graphs. For example, Figure 2-5 shows the StreamIt code corresponding to a software radio program. There are three things to notice about the figure. First, there is a natural correspondence between the structure of the code and the structure of the stream graph. Rather than reasoning about an ad-hoc set of nodes and edges, the programmer can visualize the graph while reading the code. Second, the graph description is parameterized. The number of parallel streams in the equalizer is dictated by a parameter *N*. Thus, the programmer can easily describe a broad family of stream graphs; the compiler evaluates the values of the parameters to spatially unroll the actual stream structure. Finally, imposing a single-input, single-output discipline on stream programs enables modularity and compositionality. The LowPassFilter and HighPassFilter can be drawn from a common library, without knowing the details of their internal representations.

Enforcing structure in the language can also benefit the compiler. Rather than dealing with the complexity of full graphs, the compiler can focus on a few simple cases. This property helped us to formulate phased scheduling [Kar02, KTA03], linear optimizations [LTA03, Lam03, Agr04, ATA05], and mapping to the compressed domain [THA07].

We give more details on our experience with structure in Section 2.5.

2.4 Data Reordering

Another novelty of the StreamIt language is the provision of flexible, composable, and parameterized language primitives for scattering, gathering, and reordering data. These primitives take the form of pre-defined splitter and joiner nodes, which appear in both splitjoins and feedbackloops.



```
float->float splitjoin Transpose (int M, int N)
{
   split roundrobin(1);
   for (int i = 0; i<N; i++) {
      add Identity<float>;
   }
   join roundrobin(M);
}
```

Figure 2-6: Matrix transpose in StreamIt.

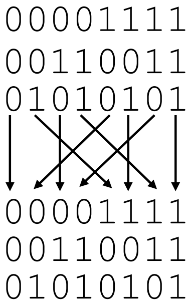
There are two types of splitters. The first splitter, *duplicate*, copies each input item to all of the output channels. The second splitter, *roundrobin*, is parameterized with a set of weights, $w_1 \ldots w_n$, where n is the number of output channels. It sends the first w_1 input items to the first stream, the next w_2 items to the second stream, and so on, repeating in a cyclic fashion. If all of the outputs have the same weight w, the splitter can be written as *roundrobin(w)*; similarly, if all the outputs have weight 1, the programmer can write simply *roundrobin*. Roundrobin is also the only type of joiner available.

By composing these simple primitives – roundrobin splitters, roundrobin joiners, and duplicate splitters – a large number of data distribution and reordering patterns can be elegantly expressed. For example, Figure 2-6 illustrates StreamIt code for a matrix transpose. The reordering needed can be expressed by a single splitjoin. The splitjoin has an empty stream (called an *Identity*) for every column in the matrix; a roundrobin(1) splitter moves the columns into the splitjoin, while a roundrobin(M) joiner moves the rows to the output channel.

Another example is bit-reversed ordering. As illustrated in Figure 2-7, a k-digit bit-reversed ordering is a permutation in which the element at index n (where n has binary digits $b_0 b_1 \ldots b_k$) is reordered to appear at index $b_k b_{k-1} \ldots b_0$. For example, in a 3-digit bit reversal, the item at index one (001) is reordered to index four (100). In a traditional language such as C, the code to perform bit-reversal is very complex; see Figure 2-8 for a standard algorithm [PFTV92]. Given the doubly-nested loops, conditionals, shift expressions, and swap operations, it is unlikely that any compiler will arrive at a sensical representation for the logical reordering performed by this computation. It is equally difficult for humans to comprehend the code.

However, the StreamIt version (Figure 2-9) of bit reversal is far simpler[1]. It represents bit reversal as a recursive reordering. In the base case, there are only two elements and no reordering

[1] Satish Ramaswamy in our group discovered this representation of bit-reversal.



```
j=1;
for (i=1; i<n; i+=2) {
  if (j > i) {
    SWAP(data[j], data[i]);
    SWAP(data[j+1], data[i+1]);
  }
  m=nn;
  while (m >= 2 && j > m) {
    j -= n;
    m >>= 1;
  }
  j += m;
}
```

Figure 2-7: Data movement in a 3-digit bit-reversed ordering.

Figure 2-8: Bit-reversed ordering in an imperative language.

```
complex->complex pipeline BitReverse (int N) {
  if (N==2) {
    add Identity<complex>;
  } else {
    add splitjoin {
      split roundrobin(1);
      add BitReverse(N/2);
      add BitReverse(N/2);
      join roundrobin(N/2);
} } }
```

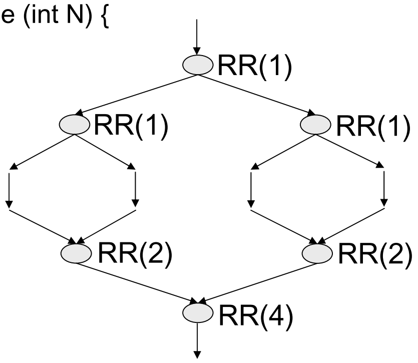


Figure 2-9: Bit-reversed ordering in StreamIt.

is needed (a 1-digit bit reversal is the identity operation). Otherwise, the reordering consists of separating elements into two groups based on the lowest-order bit of the input position, reordering both groups independently, and then joining the groups based on the highest-order bit of the output position. This pattern can be expressed with a roundrobin(1) splitter, a recursive call to BitReverse(N/2), and a roundrobin(N/2) joiner. The intuition is: bit reversal is equivalent to a tree of fine-grained splitting and coarse-grained joining. A graphical depiction of this tree appears in Figure 2-9.

Why bother to represent distribution and reordering operations in an elegant and analyzable way? The reason is that stream programming centers on data movement, and preserving information about exactly where each data item is going enables the compiler to perform more aggressive optimizations. For example, standardized splitters and joiners have enabled us to map reordering operations to a programmable on-chip network [GTK$^+$02] and have enabled certain domain-specific optimizations [LTA03, ATA05, THA07]. Other researchers have also leveraged this representation to automatically generate vector permutation instructions [NY04] and to facilitate program sketching [SLRBE05].

While the reordering primitives we have defined are quite expressive, it should be noted that they are not complete. Because splitters always distribute their first input item to the first output channel (and likewise with joiners), it is impossible to express a general permutation in which the first item is reordered to a different position of the stream. However, this behavior can be emulated by introducing simple computational nodes, such as a filter that decimates some of its inputs. Of course, it could also be rectified by adding programming language support for adjusting the order of items output. To our knowledge, the only benchmark in our suite that could leverage such a primitive is synthetic aperture radar (SAR), in which four matrix quadrants are re-shuffled in preparation for an FFT. We have not found this functionality to be broadly needed.

2.5 Experience Report

Over the past eight years, we have gained considerable experience in developing applications in StreamIt. We reflect on this experience first via a quantitative analysis of our benchmark suite, and then via qualitative impressions from StreamIt programmers.

An overview of the StreamIt benchmark suite appears in Table 2-10. At the time of this writing, the suite consists of 67 programs, including 29 realistic applications, 4 graphics rendering pipelines, 19 libraries and kernels, 8 sorting routines, and 7 toy examples. Benchmarks range in size from 21 lines (Fibonacci) to over 4,000 lines (MPEG2 encoder), with a total of 33,800 non-comment, non-blank lines in the suite[2]. Over 20 people contributed to the suite, including 6 from outside our group; median-pulse compression doppler radar was developed at Halmstad University [JSuA05], TDE was developed at the Information Sciences Insittute, an FFT and bitonic sort were developed at UC Berkeley [NY04], and the graphics pipelines were implemented primarily by the graphics group at MIT [CGT+05]. OFDM was adapted from an internal performance test of Spectrumware [TB96], while Vocoder was implemented with support from Seneff [Sen80]. Other benchmarks were often adapted from a reference implementation in C, Java, or MATLAB.

Graphical depictions of the stream graphs for each benchmark can be found in Appendix B, while the complete source code for a small benchmark (ChannelVocoder) can be found in Appendix A. A subset of the benchmarks have also been prepared for public release on the StreamIt website [Stra]. 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 2-11 details the parameterization of the StreamIt benchmarks (in addition to scheduling statistics, which are discussed later). 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 com-

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

Benchmark	Description	Author	Libraries Used	Lines of Code[1]
Realistic apps (29):				
MPEG2 encoder	MPEG2 video encoder (Drake, 2006)	Matthew Drake	DCT2D, IDCT2D	4041
MPEG2 decoder	MPEG2 vivdeo decoder (Drake, 2006)	Matthew Drake	IDCT2D	3961
GMTI	Ground moving target indicator	Sitij Agrawal	--	2707
Mosaic	Mosaic imaging with RANSAC algorithm (Aziz, 2007)	Basier Aziz	FFT	2367
MP3 subset	MP3 decoder (excluding parsing + huffman coding)	Michal Karczmarek	--	1209
MPD	Median pulse compression doppler radar (Johnsson et al., 2005)	Johnsson et al.[2]	FFT	1027
JPEG decoder	JPEG decoder	Matthew Drake	IDCT2D	1021
JPEG transcoder	JPEG transcoder (decode, then re-encode at higher compression)	Matthew Drake	DCT2D, IDCT2D	978
FAT	Feature-aided tracker[3]	Ali Meli	FFT	865
HDTV	HDTV encoder/decoder[3]	Andrew Lamb	--	845
H264 subset	16x16 intra-prediction stage of H264 encoding	Shirley Fung	--	788
SAR	Synthetic aperture radar	Rodric Rabbah	--	698
GSM	GSM decoder	Jeremy Wong	--	691
802.11a	802.11a transmitter	Sitij Agrawal	FFT	690
DES	DES encryption	Rodric Rabbah	--	567
Serpent	Serpent encryption	Rodric Rabbah	--	550
Vocoder	Phase vocoder, offers independent control over pitch and speed (Seneff, 1980)	Chris Leger	--	513
3GPP	3GPP radio access protocol - physical layer	Ali Meli	--	387
Radar (coarse)	Radar array front end (coarse-grained filters, equivalent functionality)	*multiple*	--	203
Radar (fine)	Radar array front end (fine-grained filters, equivalent functionality)	*multiple*	--	201
Audiobeam	Audio beamformer, steers channels into a single beam	Rodric Rabbah	--	167
FHR (feedback loop)	Frequency hopping radio (using feedback loop for hop signal)	Rodric Rabbah	FFT	161
OFDM	Orthogonal frequency division multiplexer (Tennenhouse and Bose, 1996)	Michael Gordon	--	148
ChannelVocoder	Channel voice coder	Andrew Lamb	--	135
Filterbank	Filter bank for multi-rate signal processing	Andrew Lamb	--	134
TargetDetect	Target detection using matched filters and threshold	Andrew Lamb	--	127
FMRadio	FM radio with equalizer	*multiple*	--	121
FHR (teleport messaging)	Frequency hopping radio (using teleport messaging for hop signal)	Rodric Rabbah	FFT	110
DToA	Audio post-processing and 1-bit D/A converter	Andrew Lamb	Oversampler	100
Graphics Pipelines (4):				
GP - reference version	General-purpose rendering pipeline: 6 vertex shaders, 15 pixel pipes[4]	Jiawen Chen	--	641
GP - phong shading	Phong shading rendering pipeline: 1 vertex shader, 12 two-part pixel pipelines	Jiawen Chen	--	649
GP - shadow volumes	Shadow volumes rendering pipeline: 1 vertex shader, 20 rasterizers	Jiawen Chen	--	460
GP - particle system	Particle system pipeline: 9 vertex shaders, 12 pixel pipelines, split triangle setup	Jiawen Chen	--	631
Libraries / Kernels (19):				
Cholesky	NxN cholesky decomposition	Ali Meli	--	85
CRC	CRC encoder using 32-bit generator polynomial	Jeremy Wong	--	131
DCT (float)	N-point, one-dimensional DCT (floating point)	Ali Meli	--	105
DCT2D (NxM, float)	NxM DCT (floating point)	Ali Meli	--	115
DCT2D (NxN, int, reference)	NxN DCT (IEEE-compliant integral transform, reference version)	Matthew Drake	--	59
IDCT (float)	N-point, one-dimensional IDCT (floating point)	Ali Meli	--	105
IDCT2D (NxM, float)	NxM IDCT (floating point)	Ali Meli	--	115
IDCT2D (NxN, int, reference)	NxN IDCT (IEEE-compliant integral transform, reference version)	Matthew Drake	--	60
IDCT2D (8x8, int, coarse)	8x8 IDCT (IEEE-compliant integral transform, optimized version, coarse-grained)	Matthew Drake	--	139
IDCT2D (8x8, int, fine)	8x8 IDCT (IEEE-compliant integral transform, optimized version, fine-grained)	Matthew Drake	--	146
FFT (coarse - default)	N-point FFT (coarse-grained)	Michal Karczmarek	--	116
FFT (medium)	N-point FFT (medium-grained butterfly, no bit-reverse)	*multiple*	--	53
FFT (fine 1)	N-point FFT (fine-grained butterfly, coarse-grained bit-reverse)	Mani Narayanan	--	139
FFT (fine 2)	N-point FFT (fine-grained butterfly, fine-grained bit-reverse	Satish Ramaswamy	--	90
MatrixMult (fine)	Fine-grained matrix multiply	Michal Karczmarek	--	79
MatrixMult (coarse)	Blocked matrix multiply	Michal Karczmarek	--	120
Oversampler	16x oversampler (found in many CD players)	Andrew Lamb	-	69
RateConvert	Audio down-sampler, converts rate by 2/3	Andrew Lamb	--	58
TDE	Time-delay equalization (convolution in frequency domain)	Jinwoo Suh	FFT	102
Sorting Examples (8):				
BitonicSort (coarse)	Bitonic sort (coarse-grained)	Chris Leger	--	73
BitonicSort (fine, iterative)	Bitonic sort (fine-grained, iterative)	Mani Narayanan	--	121
BitonicSort (fine, recursive)	Bitonic sort (fine-grained, recursive)	Mani Narayanan	--	80
BubbleSort	Bubble sort	Chris Leger	--	61
ComparisonCounting	Compares each element to every other to determine n'th output	Chris Leger	--	67
InsertionSort	Insertion sort	Chris Leger	--	61
MergeSort	Merge sort	Bill Thies	--	66
RadixSort	Radix sort	Chris Leger	--	52
Toy Examples (7):				
Autocor	Produce auto-correlation series	Bill Thies	--	29
Fib	Fibonacci number generator	David Maze	--	21
Lattice	Ten-stage lattice filter	Ali Meli	--	58
RayTracer1	Raytracer (ported from Intel)	Janis Sermulins	--	407
RayTracer2	Raytracer (rudimentary skeleton)	Amy Williams	--	154
SampleTrellis	Trellis encoder/decoder system, decodes blocks of 8 bytes	Andrew Lamb	--	162
VectAdd	Vector-vector addition	*unknown*	--	31

[1] Only non-comment, non-blank lines of code are counted. Line counts do not include libraries used.

[2] The authors of MPD are Ola Johnsson, Magnus Stenemo, and Zain ul-Abdin.

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

[4] The graphics pipelines are described in more detail in Chen et al., 2005.

Table 2-10: Overview of the StreamIt benchmark suite.

Benchmark	Parameters and default values	Parameterized:		Dynamic	Filter Execs per Steady State[1]		
		Graph	I/O Rates	Rate Filters[2]	Min	Mode	Mode Freq.
Realistic apps (29):							
MPEG2 encoder[3]	image size (320x240)	✓	✓	7	1	960	17%
MPEG2 decoder[3]	image size (320x240)	✓	✓	1	1	990	19%
GMTI	over 50 parameters	✓	✓	-	1	1	56%
Mosaic	frame size (320x240)	✓	✓	5	2	2	27%
MP3 subset	--			-	1	18	52%
MPD	FFT size (32); rows (104); cols (32)	✓	✓	-	1	416	25%
JPEG decoder	image size (640x480)		✓	2	1	4800	72%
JPEG transcoder	image size (640x480)			2	1	1	85%
FAT	15 parameters, mostly matrix dimensions	✓	✓	-	1	1	24%
HDTV	trellis encoders (12); interleave depth (5)	✓		-	20	1380	38%
H264 subset	image size (352x288)		✓	-	17	396	26%
SAR	over 30 parameters		✓	-	1	1	95%
GSM	--			-	1	1	65%
802.11a	--			-	1	1	14%
DES	number of rounds (16)	✓		-	1	1	62%
Serpent	number of rounds (32); length of text (128)	✓	✓	-	1	1	40%
Vocoder	pitch & speed adjustments, window sizes	✓	✓	-	1	1	88%
3GPP	matrix dimensions Q, W, N, K (2, 2, 4, 8)	✓	✓	-	1	9	48%
Radar (coarse)	channels (12); beams (4); decimation rates; window sizes	✓	✓	-	1	1	38%
Radar (fine)	channels (12); beams (4); decimation rates; window sizes	✓	✓	-	1	1	49%
Audiobeam	channels (15)	✓	✓	-	1	1	94%
FHR (feedback loop)	window size (256)	✓	✓	-	1	1	26%
OFDM	decimation rates (825, 5); window size (20); demod rate (5)		✓	-	1	1	57%
ChannelVocoder	number of filters (16); pitch window (100); decimation (50)	✓	✓	-	1	50	66%
Filterbank	bands (8); window size (128)	✓	✓	-	1	8	64%
TargetDetect	window size (300)		✓	-	1	1	90%
FM Radio	bands (7); window size (128); decimation (4)	✓	✓	-	1	1	97%
FHR (teleport messaging)	window size (256)	✓	✓	-	1	1	23%
DToA	window size (256)		✓	-	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 (19):							
Cholesky	matrix size (16x16)	✓	✓	-	1	1	70%
CRC	--			-	1	1	98%
DCT (float)	window size (16)	✓	✓	-	1	1	79%
DCT2D (NxM, float)	window size (4x4)	✓	✓	-	1	1	84%
DCT2D (NxN, int, reference)	window size (8x8)	✓	✓	-	1	1	80%
IDCT (float)	window size (16)	✓	✓	-	1	1	85%
IDCT2D (NxM, float)	window size (4x4)	✓	✓	-	1	1	83%
IDCT2D (NxN, int, reference)	window size (8x8)	✓	✓	-	1	1	80%
IDCT2D (8x8, int, coarse)	--			-	1	64	50%
IDCT2D (8x8, int, fine)	--			-	1	1	89%
FFT (coarse - default)	window size (64)	✓	✓	-	1	1	23%
FFT (medium)	window size (64)	✓		-	32	32	92%
FFT (fine 1)	window size (64)	✓	✓	-	1	1	99%
FFT (fine 2)	window size (64)	✓		-	1	1	98%
MatrixMult (fine)	matrix dimensions NxM, MxP (12x12, 9x12)	✓	✓	-	9	108	30%
MatrixMult (coarse)	matrix dimensions NxM, MxP (12x12, 9x12); block cuts (4)	✓	✓	-	9	12	31%
Oversampler	window size (64)		✓	-	1	1	20%
RateConvert	expansion and contraction rates (2, 3); window size (300)		✓	-	2	2	40%
TDE	number of samples (36); FFT size is next power of two	✓	✓	-	1	15	24%
Sorting Examples (8):				-			
BitonicSort (coarse)	number of values to sort (16)	✓	✓	-	1	1	67%
BitonicSort (fine, iterative)	number of values to sort (16)	✓		-	1	1	100%
BitonicSort (fine, recursive)	number of values to sort (16)	✓		-	1	1	95%
BubbleSort	number of values to sort (16)	✓		-	1	1	100%
ComparisonCounting	number of values to sort (16)	✓	✓	-	1	1	85%
InsertionSort	number of values to sort (16)	✓	✓	-	1	1	67%
MergeSort	number of values to sort (16)	✓	✓	-	1	1	88%
RadixSort	number of values to sort (16)	✓	✓	-	1	1	85%
Toy Examples (7):				-			
Autocor	length of vector (32); length of autocor series (8)	✓	✓	-	1	1	80%
Fib	--			-	1	1	100%
Lattice	number of stages (10)	✓		-	1	1	100%
RayTracer1	no parameters, though data read from file			-	1	1	100%
RayTracer2	implicitly parameterized by scene (simple circle)	✓		-	1	1	100%
SampleTrellis	frame size (5)		✓	-	1	40	46%
VectAdd	--			-	1	1	100%

[*] Statistics represent properties of complete programs, in which libraries have been inlined into caller.

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

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

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

Table 2-11: Parameterization and scheduling statistics for StreamIt benchmarks.

	TOTAL FILTERS			PEEKING FILTERS			STATEFUL FILTERS[1]			OTHER CONSTRUCTS		
Benchmark	Types	Instances (non-Iden.)	Instances (Identity)	Types	Instances	Max Work[2]	Types	Instances	Max Work[2]	Splitjoins	Feedback Loops	Work in F. Loop[2]
Realistic apps (29):												
MPEG2 encoder[3]	35	113	30	-	-	-	9	11	N/A	15	-	-
MPEG2 decoder[3]	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	-	-
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	-	-
FM Radio	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 (19):												
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	-	-
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	-	-	-	-	-	-	-	-	-	-
Sorting Examples (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	-	-	-	-	-	-	-	-	-	-
Toy Examples (7):												
Autocor	3	10	-	-	-	-	-	-	-	1	-	-
Fib	2	2	1	1	1	45.5%	-	-	-	-	1	73%
Lattice	4	18	10	-	-	-	-	-	-	9	-	-
RayTracer1	4	4	-	-	-	-	-	-	-	-	-	-
RayTracer2	5	5	-	-	-	-	-	-	-	-	-	-
SampleTrellis	14	12	1	1	1	N/A	2	2	N/A	1	-	-
VectAdd	3	4	-	-	-	-	-	-	-	1	-	-

[*] Statistics represent properties of complete programs, in which libraries have been inlined into caller.

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

[2] 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) for programs containing dynamic rates (MPEG2, Mosaic, Graphics pipelines) or external Java routines (HDTV, SampleTrellis).

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

Table 2-12: Properties of filters and other constructs in StreamIt benchmarks.

```
int->int filter DifferenceEncoder_Stateless {

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

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

Figure 2-13: Stateless version of a difference encoder, using peeking and prework.

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

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

Figure 2-14: Stateful version of a difference encoder, using internal state.

prehensibility 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 2-12. In terms of size, benchmarks declare (on average) 11 filter types and instantiate them 63 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.

We organize further discussion of the benchmark suite according to the key outcomes of our survey. 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. 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.

1. **Peeking is widely used for a variety of sliding window computations. Without peeking, such computations would often introduce a stateful bottleneck in the program.** Twenty two benchmarks – and more than half of the realistic applications – contain at least one filter that peeks. (That is, these filters declare a peek rate larger than their pop rate, examining some items that are not dequeued from the input channel until a later execution. We do not count filters that merely call the peek primitive, as those items may be popped during the same execution.) 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 35) of the peeking filter declarations. They are 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 2-13, 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 2-14. Across our benchmark suite, this pattern accounts for more than one quarter (10 out of 35) 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 (10 out of 35) perform various sliding-window functions. For example, MP3 reorders and adds data across large (>1000 item) sliding windows; 802.11 and SampleTrellis 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, Table 2-12 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 8 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.

2. **Prework functions are useful for expressing startup conditions, and for eliminating associated state.** 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 SampleTrellis). Without prework, the delayed items would need to be buffered internally to the filter, introducing state into the computation.

   Other benchmarks use prework for miscellaneous startup conditions. As mentioned previously, the difference encoder in Figure 2-13 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.

3. **Stateful filters are less common than we expected, though are nonetheless required for complete expression of many algorithms. 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 67) 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. While other researchers have noted that stream programs are rich in data parallelism [KRD$^+$03], we nonetheless expected to see a greater proportion of filters that retained mutable state between execution steps. 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 current 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 SampleTrellis) 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 (messaging is described in the next chapter). 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 SampleTrellis 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. **Feedback loops are uncommon in our benchmarks, but represent significant bottlenecks when present.** While our discussion thus far has focused on stateful filters, seven benchmarks also contain explicit feedback loops in the graph structure. Four of these loops (Fib, FHR feedback, H264 subset, CRC) represent significant bottlenecks to parallelization, with workloads ranging from 73% 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. **Splitjoins and Identity filters are very common in the benchmarks.** These two language constructs found broad application across our benchmark suite. Splitjoins appear in over three quarters (53 out of 67) of the benchmarks, with a median of 8 instantiations per benchmark.

Benchmark	Sender(s)	Receiver(s)	Direction	Latency	Purpose
MPEG2 encoder	1	2	Upstream	0	Send back reference frame to forward and backward motion predictors
	1	4	Downstream	0	Send picture number and picture type to processing filters
	1	4	Downstream	0	Broadcast common parameter to adjacent filters, across splitters/joiners
MPEG2 decoder	1	7	Downstream	0	Send video parameters from file parser to decoding stages
Mosaic	1	3	Upstream and Downstream	0	Indicate if RANSAC algorithm should continue to iterate
	1	3	Downstream	0	Broadcast common result (number of inliers) to pipeline of filters
	1	1	Downstream	0	Send result across joiner, with different type than data stream
FHR (teleport messaging)	4	1	Upstream	6	Indicate detection of signal to switch frontend to given frequency

Table 2-15: Use of teleport messaging in StreamIt benchmarks.

Roundrobin splitters accounted for 65% of the instantiations, while the other splitters are of duplicate type. (All joiners are of roundrobin type.) Identity filters were used in half (33 of 67) of the benchmarks, with a median of 13 instantiations per benchmark. Identity filters are recognized by the compiler as a pass-through operation, allowing it to map communication instructions directly to a network fabric.

6. **Teleport messaging and dynamic rates are uncommon in the benchmarks, but provide critical functionality when utilized.** These language features were not fully specified until years after the initial release of the compiler, which contributes to their smaller representation in the benchmark suite.

   As detailed in Table 2-15, teleport messages are utilized by four of the benchmarks (MPEG2 encoder, MPEG2 decoder, Mosaic, and FHR). There are a total of 8 logical messages, often between multiple senders or multiple receivers. Both upstream and downstream messages are utilized; all messages are sent with a latency of zero. While relatively few of the benchmarks use teleport messaging, the functionality provided is essential. As described in the next chapter for the case of FHR, and elsewhere for MPEG2 [Dra06] and Mosaic [Azi07], messaging greatly simplifies and improves the expression of these algorithms in a streaming context.

   Similarly, as illustrated in Table 2-11, dynamic rates are utilized by only 9 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. 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.

7. **Neighboring filters often have matched I/O rates.** 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 2-16) is used in many studies [MBL94, BML95, TZB99, BB00, CBL01, MB04, KSB06]; 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 [MBL94]. However, neighboring filters have different communication rates which share no common factors, resulting in a large steady-state schedule.

   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 difference 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 2-17 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, Table 2-11 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 90% (60 out of 67) 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, 66% of the filters in a program share the same multiplicity. For over two-thirds of the benchmarks (46 out of 67), the most common multiplicity is 1; in these benchmarks, an average of 75% 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 prevalance of matched rates in our benchmark suite also led to unexpected results in some of our papers. For example, in our work on phased scheduling, we developed a new scheduling algorithm that reduces the buffer requirements needed to execute a synchronous dataflow graph [KTA03]. 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 was extremely small due to matched input and output rates; simply executing each node once would often give the minimal possible buffering. This result emphasizes the importance of optimizing the common case in realistic programs, rather than restricting attention to small examples.

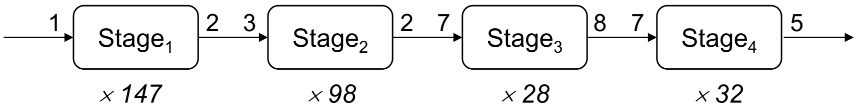


Figure 2-16: The CD-DAT benchmark [MB04] exhibits unusually mis-matched I/O rates. Nodes are annotated with the number of items pushed and popped per execution, as well as their execution multiplicity in the steady state. Since neighboring filters produce different numbers of items, each filter has a large multiplicity in the steady state. This demands clever scheduling strategies to avoid extremely large buffer sizes.

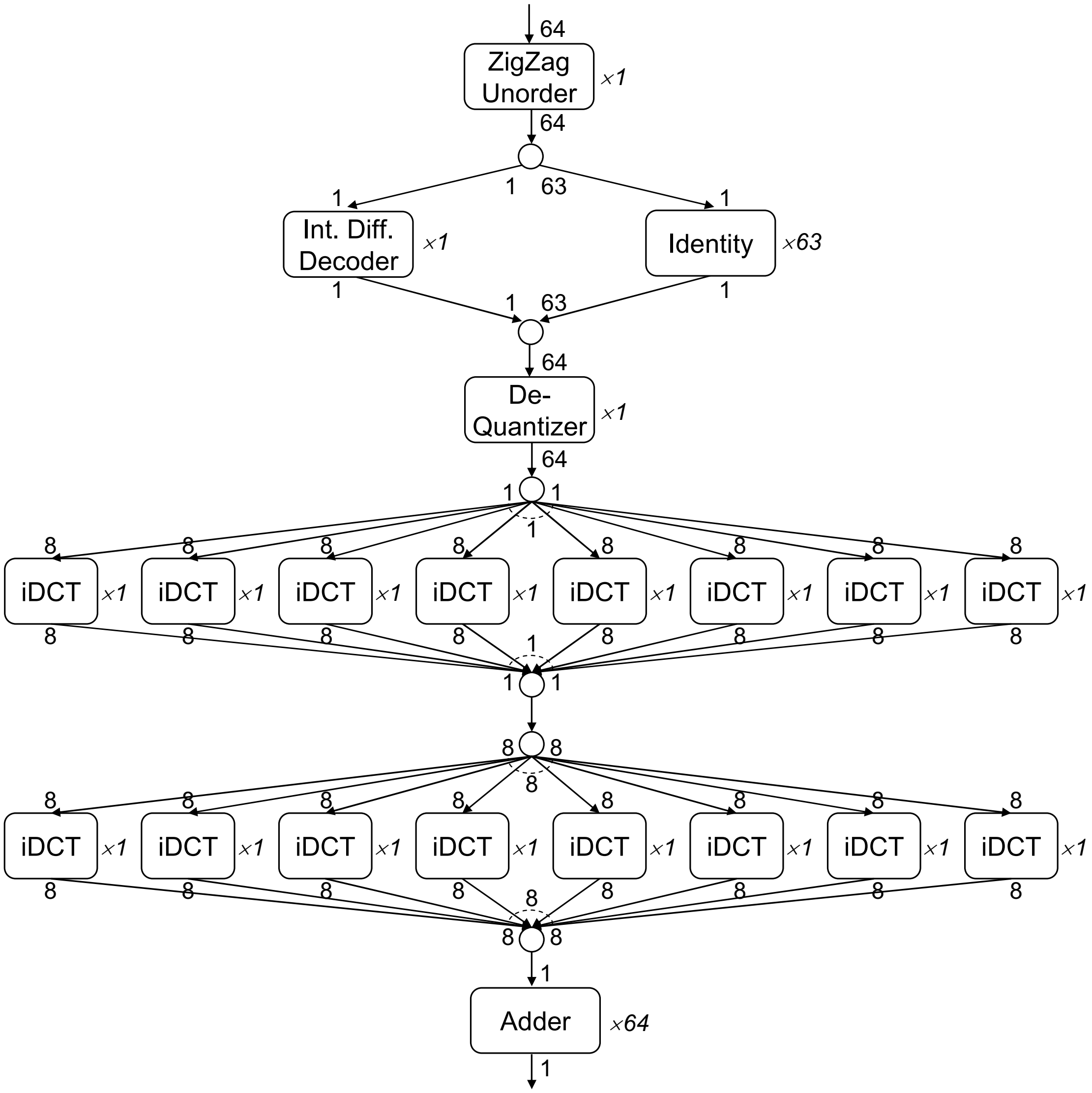
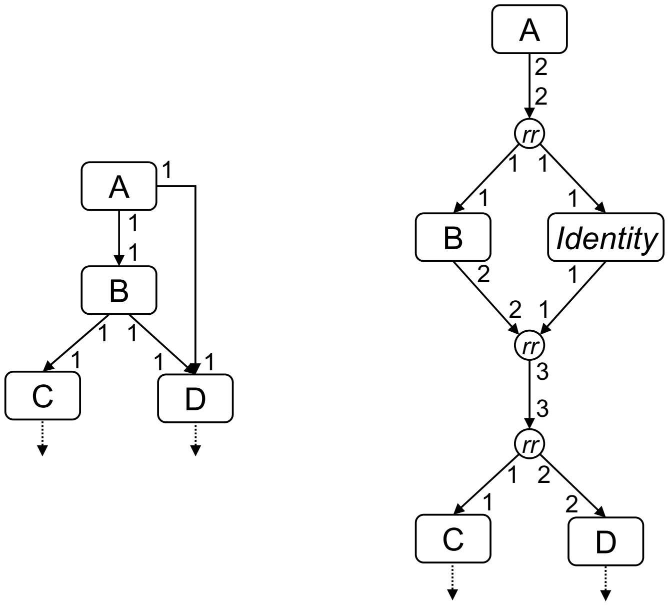


Figure 2-17: This excerpt from the JPEG transcoder illustrates matched I/O rates, as found in many benchmarks. The graph is transforming pixels from an 8x8 macroblock. Nodes are annotated with the number of items pushed and popped per execution, as well as their execution multiplicity in the steady state. 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.



(a) Unstructured (b) Structured

Figure 2-18: Example of refactoring a stream graph to fit a structured programming model. Both graphs achieve equivalent communication between filters.

In addition to our observations about the benchmark characteristics, we also offer some lessons learned from developers' experiences in implementing stream programs. As noted in Table 2-10, 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. We summarize their experience as follows:

1. **Structured streams are a useful and tractable means of writing programs. However, they 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. 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 2-18. 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 2-18b) is needed to perform this interleaving. A more realistic example of the same hazard is shown in Figure 2-19, which corresponds to our 3GPP benchmark.

   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

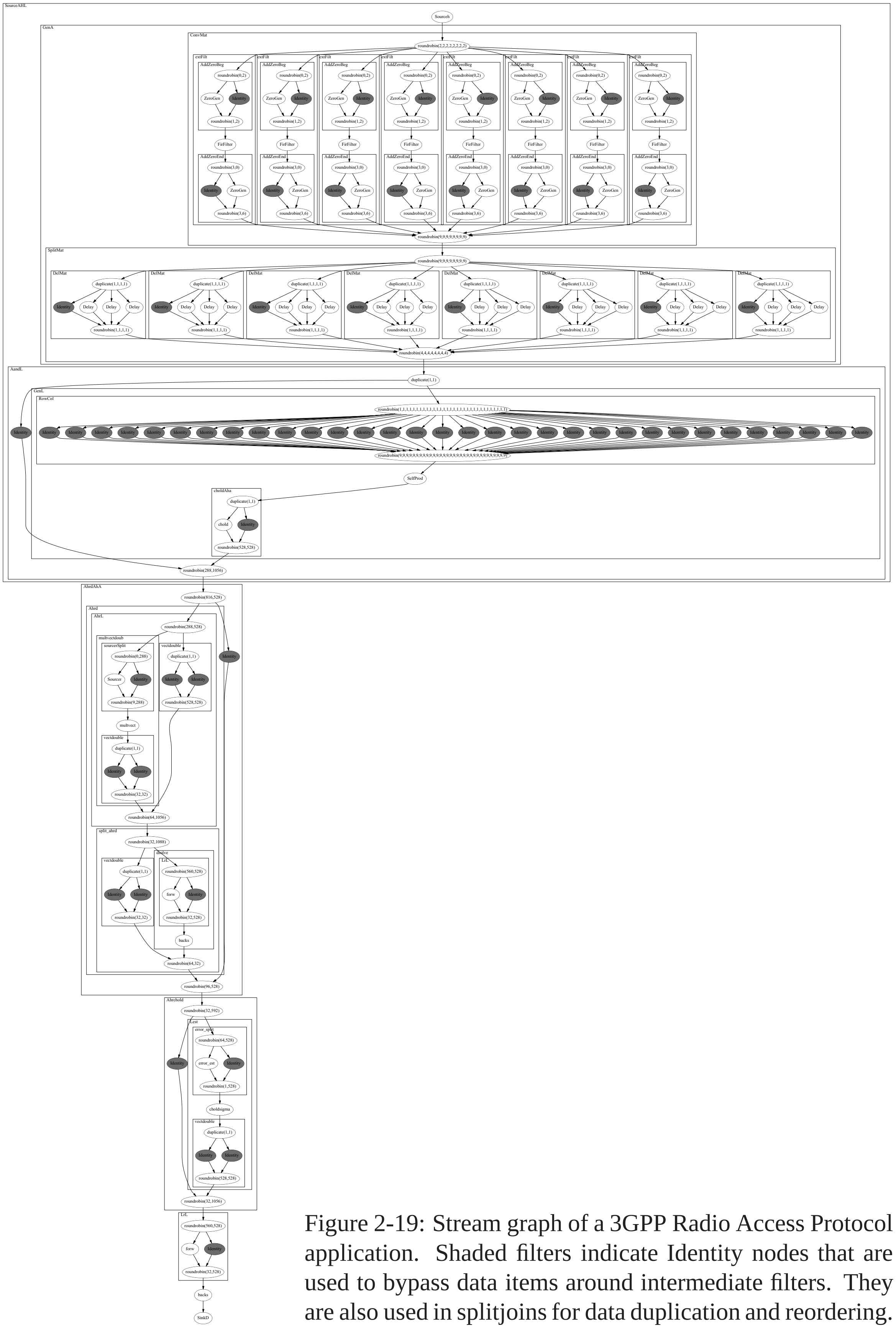


Figure 2-19: Stream graph of a 3GPP Radio Access Protocol application. Shaded filters indicate Identity nodes that are used to bypass data items around intermediate filters. They are also used in splitjoins for data duplication and reordering.

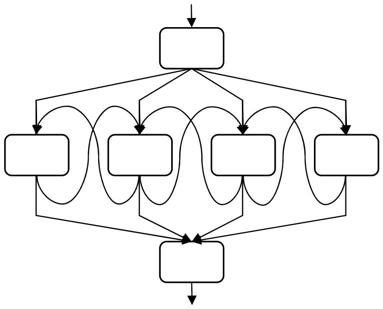


Figure 2-20: A communication pattern unsuitable for structured streams. This pattern can arise in video compression, where each block informs its neighbors of its motion prediction before the next processing step.

.

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 unpackages structures from adjoining data channels. The second approach is to employ teleport messaging; as described in the next chapter, it 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.

Finally, there are rare cases in which the structured primitives in StreamIt have been inadequate for representing a streaming communication pattern. Figure 2-20 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. This case reflects a broader shortcoming, discussed in Section 2.7, that StreamIt is not designed for multidimensional data processing.

2. **Programmers can accidentally introduce unnecessary 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 im-

```
void->int filter SquareWave() {
  work push 2 {
    push(0);
    push(1);
  }
}
```

(a) Stateless

```
void->int filter SquareWave() {
  int x = 0;

  work push 1 {
    push(x);
    x = 1 - x;
  }
}
```

(b) Stateful

Figure 2-21: Programmers can accidentally introduce unnecessary filter state when writing programs. In this example, the intended output is a square wave, emitting alternate values of 0 and 1. Both implementations shown are functionally equivalent. However, the stateless version (a) appears data-parallel to the compiler, while the stateful version (b) appears sequential.

plementations of an algorithm, would sometimes choose the one that includes mutable state. Figure 2-21 gives a pedantic example of this problem, while Figure 2-22 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.

3. **Multi-phase filters confuse programmers and are not necessary.** At one point in the StreamIt project, we embraced the cyclo-static dataflow model [BELP95, PPL95] 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, our experience is that having the option of multiple execution steps is confusing to beginning StreamIt programmers. There is a tendency to interpret multiple execution steps as belonging to multiple distinct filters. It is also difficult to explain to a non-expert why one method should be designated as an execution step, rather than as a plain subroutine call.

   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. However, 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. Apart from splitters and joiners, we did not encounter any scenarios (in our limited benchmark suite) that demanded multiple execution steps in filters.

   Thus, after making a significant investment to support the full generality of cyclo-static dataflow in the StreamIt compiler, we eventually changed course and removed the capability from the language.

4. **Input and output rates can typically be inferred from the code inside a filter. However, it is still worthwhile for the programmer to declare them.** We were surprised how many StreamIt benchmarks contained completely static control flow inside the body of filters. That is,

```
float->float splitjoin
CFARDelayToLMax_Stateless(int rows) {
    split roundrobin;
    add Delay(rows-1);
    add Delay(rows-1);
    add Identity<float>();
    add Delay(rows-1);
    add Delay(rows-1);
    join roundrobin;
}

float->float filter Delay(int N) {
    prework push N {
        for (int i=0; i<N; i++) {
            push(0.0);
        }
    }

    work push 1 pop 1 {
        push(pop());
    }
}
```

(a) Stateless

```
float->float filter
CFARDelayToLMax_Stateful(int rows) {
    float[rows] guardNoise;
    float[rows] thresh;
    float[rows] sumDb;
    float[rows] guardDb;
    int popPos = 0;
    int pushPos = 1;

    work pop 5 push 5 {
        guardNoise[popPos] = pop();
        push(guardNoise[pushPos]);
        thresh[popPos] = pop();
        push(thresh[pushPos]);
        push(pop());
        sumDb[popPos] = pop();
        push(sumDb[pushPos]);
        guardDb[popPos] = pop();
        push(guardDb[pushPos]);

        popPos++;
        pushPos++;

        if(popPos >= rows) {
            popPos = 0;
        }
        if(pushPos >= rows) {
            pushPos = 0;
        }
    }
}
```

(b) Stateful

Figure 2-22: A second example, drawn from MPD, in which a stateless computation was written in a stateful style in the original implementation. The original version (b) performs a complex delay and reordering of interleaved vectors on the input stream, and appears stateful to the compiler. It can be rewritten as a stateless construct (a), which separates the logical streams using a splitjoin and applies a stateless delay to each one.

the path of control taken through the *work* function is often independent of the data values input to the filter. Exceptions to this pattern include sorting algorithms, compression algorithms, and parsing algorithms (e.g., the MPEG-2 bitstream parser).

When the control flow is static, it is often feasible for the compiler to infer the number of items pushed and popped via a static analysis. Such an analysis could save the programmer the trouble of annotating each work function with its input and output rates.

However, we did find that it is valuable for programmers to annotate the input and output rates even when they can be inferred. As is commonly the case with type declarations, these annotations provided documentation to other users regarding the intended behavior of the filter, making it easier to understand and maintain. They also provided a level of redundancy, so that, when possible, the compiler could check the consistency between the declared rates and the actual implementation.

2.6 Related Work

As described in Chapter 1 and elsewhere [Ste97], there is a long history of programming language support for streams in the dataflow, functional, and synchronous language domains. Here we compare to StreamIt's more immediate contemporaries.

The Brook language is architecture-independent and focuses on data parallelism [BFH+04]. Stream kernels are required to be stateless, though there is special support for reducing streams to a single value. Sliding windows are supported via stencils, which indicate how data elements should be replicated across multiple processing instances. While StreamIt supports a single stream graph operating a conceptually infinite stream, Brook supports multiple graphs, embedded in a C program, that operate on finite-length streams. An independent comparison of the two languages by Mattson and Lethin [ML03] aptly summarizes the philosophical difference, in that StreamIt was designed by compiler writers (it is "clean but more constrained") while Brook was driven by application developers and architects, and is "rough but more expressive".

Brook is one of several stream-oriented languages that evolved out of the graphics community. Cg exploits pipeline parallelism and data parallelism, though the programmer must write algorithms to exactly match the two pipeline stages of a graphics processor [MGAK03]. The sH language, subsequently commercialized by RapidMind, is embedded within C++ as a set of macros [MQP02, MTP+04]. Like StreamIt, sH specializes stream kernels to their constant arguments, and fuses pipelined kernels in order to increase their granularity. Unlike StreamIt, sH performs these optimizations dynamically in a Just-In-Time (JIT) compiler, offering increased flexibility. However, StreamIt offers increased expressiveness in that 1) StreamIt can express arbitrary stream graphs, while sH appears to be limited to pipelines, and 2) StreamIt can express kernels with state, while sH kernels must be stateless. Accelerator [TPO06] also employs a Just-In-Time strategy to target GPUs from C#; the system derives parallelism from data-parallel array types rather than explicit stream kernels.

StreamC/KernelC preceded Brook and operates at a lower level of abstraction; kernels written in KernelC are stitched together in StreamC and mapped to the data-parallel Imagine processor [KRD+03]. SPUR adopts a similar decomposition between "microcode" stream kernels and skeleton programs to expose data parallelism [ZLSL05].

StreamIt is not the first language to incorporate the notion of a sliding window. In the Warp project, the AL language [Tse89] had a window operation for use with arrays, and Printz's "signal flow graphs" included nodes that performed a sliding window [Pri91]. The ECOS graphs language allows actors to specify how many items are read but not consumed [HMWZ92]; the Signal language allows access to the window of values that a variable assumed in the past [GBBG86]; and the SA-C language contains a two-dimensional windowing operation [DBH+01]. However, to the best of our knowledge, we are the first to demonstrate the utility of sliding windows in improving parallelism and programmability across a large benchmark suite.

To summarize the differences to other stream languages, StreamIt places more emphasis on exposing task and pipeline parallelism (all the languages expose data parallelism). By adopting the synchronous dataflow model of execution, StreamIt focuses on well-structured and long-running programs that can be aggressively optimized. We are not aware of structured streams or hierarchical mechanisms for data reordering in other stream languages. Spidle [CHR+03] is also a recent stream language that was influenced by StreamIt.

2.7 Future Work

There are many directions in which to expand and refine the StreamIt language. Based on his study of MPEG-2 in StreamIt, Matthew Drake makes a sound case for adding support for programmable splitters and joiners, re-initialization of streams, draining of streams, and dispatch splitjoins [Dra06]. He also discusses extensions to teleport messaging, described in the next chapter. We endorse his recommendations and also highlight the following research directions:

1. **Dynamic changes to stream structure.** A long-time goal of the StreamIt group has been to define and implement support for dynamic changes to the stream graph. For example, an adaptive channel decoder may decide to add or remove filtering stages; an FIR filter may dynamically scale the size of the window it considers; a network router may add or remove streams to represent new logical flows; or an AMPS cellular base station may add and remove streams to support new clients.

   There are several challenges and opportunities in supporting dynamic stream graphs. As described in the next chapter, our basic model for runtime adaptation is to re-evaluate the initialization code for stream structures by sending a teleport message to that stream. The difficulty comes in timing the re-initialization, migrating filter state and buffered data items to the new graph, and maintaining as much static information as possible about the possible configurations of graphs that will be adopted at runtime. Many of these issues arise not from dynamism, but from incorporating a notion of finite streams into the language; as the current language views all streams are conceptually infinite, it does not have to deal with boundary conditions or termination procedures, both of which are prerequisites for dynamic reconfiguration. While we have developed extensive internal notes and proposals on language support for dynamism, we omit them from this dissertation because we have yet to reach consensus on many aspects of the design.

   As an intermediate step towards supporting a fully-reconfigurable stream graph, it would also be interesting to introduce primitives that allow programmers to indicate which parts of code should be evaluated at compile time, versus being evaluated at load time or runtime. The current StreamIt compiler requires the structure and communication rates in the stream graph to be evaluated at compile time, though the StreamIt language could also be interpreted as binding these values at load time (during program initialization). While compile-time evaluation improves optimization opportunities, it is not always permissible by the application. For example, if an external file is used to drive the structure or parameters of the stream graph, then compile-time evaluation is safe if that file is fixed across all executions (e.g., a simulator for a specific processor architecture) but unsafe if it may vary from one execution to the next (e.g., a scene description for a rendering engine). We envision that a simple type modifier, such as a “dynamic” keyword, could be used to distinguish these cases. The type system would guarantee that everything that depends on dynamic data is also declared dynamic. This would allow the compiler to maximally evaluate other sections of the stream graph at compile time.

2. **Multidimensional data.** The current version of StreamIt is a natural fit for handling one-dimensional sequences of data, but falls short in exposing the dependences and flexibility inherent in manipulating multi-dimensional data. When handling sequences of multidimensional data (such as video frames), the programmer is currently left with two alternatives. One option is to take a coarse-grained approach in which filters push and pop entire arrays at a time. However, this results in nested loops within filter code, reducing the problem to a traditional loop analysis without gaining any leverage from the streaming domain. The second option is to take a fine-grained approach, in which individual arrays are split up into columns or blocks and distributed over many filters. However, this mapping forces the programmer to specify a fixed decomposition of the data in the array (row-major, column-major, blocked, etc.) and makes it more difficult for the compiler to infer the underlying dependences and adjust the schedule as needed.

   One possibility for handling multidimensional data could be to add iterators that apply a filter (or entire stream graph) to all of the elements of an array. The Brook language [BFH$^+$04] adopts a similar approach in a construct termed *stencils*. However, stencils generally operate on a single array at a time and are not integrated into a larger stream graph. An opportunity for future work would be to create a unified environment for processing sequences of arrays and data items within arrays, including compiler-friendly "bridge" operators that decompose arrays into data streams and assemble data streams into arrays. Research challenges arise in the specification of boundary conditions on the sides of an array, the dependences and reuse between different parts of an array, and the possibility for carried state across separate arrays. Many of these issues are again rooted in StreamIt's ties to an infinite stream abstraction. Integrated support for finite streams will be needed to effectively handle multidimensional data.

3. **External interfaces.** In practice, it is important for any domain-specific language to have well-defined interfaces for interacting with languages and systems that fall outside of the domain. In the case of streaming, this encompasses interfaces for embedding stream graphs within general purpose languages, as well as for embedding general-purpose computations within stream graphs. While we have developed an internal, ad-hoc interface for interfacing between StreamIt and C, there are interesting research questions in rigorously defining the semantics of such hybrid computational models.

   For example, one characteristic of synchronous dataflow is that data streams are virtually infinite; however, from a general-purpose language, streaming computations can also be gainfully applied to large arrays. Thus, it will be valuable to develop formal notions of draining the stream graph, and perhaps mechanisms to maintain the state of a stream graph from one instantiation to another.

   There are also interesting questions that relate to the memory model of hybrid systems. Synchronous dataflow represents a fully distributed model with no access to shared state; however, other general-purpose programming models often embrace a shared-memory abstraction. As described in the next chapter, one approach to unifying these abstractions could be to allow streaming updates to shared memory so long as they are committed according to a deterministic static schedule.

2.8 Chapter Summary

This chapter describes the design rationale and experienced gained from the StreamIt language, one of the first programming languages that exposes and exploits the inherent regularity of stream programs. StreamIt is rooted in the synchronous dataflow model, with added support for multiple execution steps, dynamic communication rates, teleport messaging, peeking, and communication during initialization. Key novelties of the language are the notion of structured streams – akin to structured control flow in an imperative language – as well as hierarchical and parameterized splitjoins for data reordering. The design of the basic computational node in StreamIt, the filter, also exposes inherent parallelism that is masked by pointer manipulation and modulo operations in a traditional C implementation.

The development of a large-scale benchmark suite in StreamIt led to several insights and surprises. Language support for sliding windows and communication during the initialization stage enabled many filters to be written in a stateless manner, exposing parallelism that would have been masked without these features. We were surprised how few filters contained mutable state; this suggests that many programs can leverage data parallelism, rather than relying on task and pipeline parallelism, to achieve parallel performance. Our benchmarks often contain matched input and output rates, where filters need to execute only a small number of times before satisfying the steady-state data requirements of their neighbors. This property reduces the space of scheduling alternatives as well as the benefit derived (e.g., in buffer space) from complex filter interleavings.

Continuous feedback from StreamIt developers also provided a valuable critique of the StreamIt language. While structured streams were a natural way to represent common programs, in some cases the programmer needed to refactor an unstructured stream graph into a more complex structured representation. An optional mechanism for infrequent unstructured communication may be valuable in future languages. Programmers were also prone to accidentally introduce mutable filter state, impeding parallelization. Future languages should expose this performance cost to the programmer so that they avoid unnecessary serialization. We found that multi-phase filters (as in cyclo-static dataflow) are likely to confuse programmers and are not necessary to express computations in our benchmark suite. Finally, while the input and output rates of most filters could be inferred, it was still worthwhile to declare them from a software engineering standpoint.

There is rich potential for future work in stream languages, including support for dynamically changing the stream structure, support for multidimensional data, and support for external interfaces.

Chapter 3

Teleport Messaging

In this chapter, we develop a new language construct to address one of the pitfalls of parallel programming: precise handling of events across parallel components. The construct, termed *teleport messaging*, uses data dependences between components to provide a common notion of time in a parallel system. We leverage the static properties of synchronous dataflow to compute a stream dependence function, SDEP, that compactly describes the ordering constraints between actor executions.

Teleport messaging utilizes SDEP to provide powerful and precise event handling. For example, an actor A can specify that an event should be processed by a downstream actor B as soon as B sees the "effects" of the current execution of A. We argue that teleport messaging improves readability and robustness over existing practices. We have implemented messaging as part of the StreamIt compiler, with a backend for a cluster of workstations. As teleport messaging exposes optimization opportunities to the compiler, it also results in a 49% performance improvement for a software radio benchmark.

3.1 Introduction

One difficult aspect of stream programming, from both a performance and programmability standpoint, is reconciling regular streaming dataflow with irregular control messages. While the high-bandwidth flow of data is very predictable, realistic applications also include unpredictable, low-bandwidth control messages for adjusting system parameters (e.g., filtering coefficients, frame size, compression ratio, network protocol, etc.). Control messages often have strict timing constraints that are difficult to reason about on parallel systems.

For example, consider a frequency hopping radio (FHR), which mirrors how CDMA-based cell phone technology works. In FHR, a transmitter and a receiver switch between a set of known radio frequencies, and they do so in synchrony with respect to a stream boundary. That is, a receiver must switch its frequency at an exact point in the stream (as indicated by the transmitter) in order to follow the incoming signal. Such a receiver is challenging to implement in a distributed environment because different processors might be responsible for the radio frontend and the frequency hop detection. When a hop is detected, the detector must send a message to the frontend that is timed precisely with respect to the data stream, even though the two components are running on different processors with independent clocks.

Other instances of control messaging have a similar flavor. A component in a communications frontend might detect an invalid checksum for a packet, and send a precisely-timed message downstream to invalidate the effects of what has been processed. Or, a downstream component might detect a high signal-to-noise ratio and send a message to the frontend to increase the amplification. In an adaptive beamformer, a set of filtering coefficients is periodically updated to focus the amplification in the direction of a moving target. Additional examples include: periodic channel characterization; initiating a handoff (e.g., to a new network protocol); marking the end of a large data segment; and responding to user inputs, environmental stimuli, or runtime exceptions.

There are two common implementation strategies for control messages using today's languages and compilers. First, the message can be embedded in the high-bandwidth data flow, perhaps as an extra field in a data structure. Application components check for the presence of messages on every iteration, processing any that are found. This scheme offers precise timing across distributed components, as the control message has a well-defined position with respect to the other data. However, the timing is inflexible: it is impossible for the sender to synchronize the message delivery with a data item that has already been sent, or to send messages upstream, against the flow of data. This approach also adds runtime overhead, as most of the data sent are placeholders to indicate the absence of an infrequent control message. The stream graph may also grow more complex as dedicated channels are added to handle only control messages.

A second implementation strategy is to perform control messaging "out-of-band", via a new low-bandwidth connection or a remote procedure call. While this avoids the complexity of embedding messages in a high-bandwidth data stream, it falls short in terms of timing guarantees. In a distributed environment, each processor has its own clock and is making independent progress on its part of the application. The only common notion of time between processors is the data stream itself. Though extra synchronization can be imposed to keep processors in check, such synchronization is costly and can needlessly suppress parallelism. For example, if the compiler understands the latency of messages between pipelined components, then it can derive the maximal buffering between those components that nonetheless respects the message delivery constraints. This buffering enables aggressive execution reordering, decreased synchronization, and amortized communication overhead. However, with opaque message timing, the runtime must eliminate buffering to maintain a consistent view for all possible schedule of messages, eliminating the optimization potential.

This chapter presents a new language construct and supporting compiler analysis that allows the programmer to declaratively specify control messages. Termed "teleport messaging", this feature offers the simplicity of a method call while maintaining the precision of embedding messages in the data stream. The idea is to treat control messages as an asynchronous method call with no return value. When the sender calls the method, it has the semantics of embedding a placeholder in the sender's output stream. The method is invoked in the receiver when the receiver would have processed the placeholder. We generalize this concept to allow variable latency, in which messages are received at an offset from the placeholder. By using a negative latency, messages can be retroactively attached to data that was previously emitted from a node. We extend this concept further to enable messages to travel opposite the flow of data, timed relative to data that will later be received by the message sender. By exposing the true timing constraints to the compiler, messages can be delivered using whatever mechanism is appropriate for a given architecture. The declarative mechanism also enables the compiler to safely derive whatever buffering is needed and to parallelize and reorder application components so long as messages are delivered on time.

Our formulation of teleport messaging relies on the static input and output rates inherent in the synchronous dataflow model [LM87]. Because the rates are statically known, we can compute the dependences between actors and automatically calculate when a message should be delivered. We develop a stream dependence function, SDEP, that provides an exact, complete, and compact representation of this dependence information; we use SDEP to specify the semantics of teleport messaging.

Teleport messaging is supported in some parts of the StreamIt compiler. The implementation computes SDEP information and automatically targets a cluster of workstations. Based on a case study of a frequency hopping radio, we demonstrate a 49% performance improvement of teleport messaging relative to an explicit feedback loop.

Illustrating Example

Figure 3-1 illustrates a StreamIt version of an FIR (Finite Impulse Response) filter. A common component of digital signal processing applications, FIR filters represent sliding window computations in which a set of coefficients is convolved with the input data. This FIR implementation is very fine-grained; as depicted in Figure 3-4, the stream graph consists of a single pipeline with a `Source`, a `Printer`, and 64 `Multiply` stages – each of which contains a single coefficient (or *weight*) of the FIR filter. Each `Multiply` actor inputs a `Packet` consisting of an input item and a partial sum; the actor increments the sum by the product of a weight and the *previous* input to the actor. Delaying the inputs by one step ensures that each actor adds a different input to the sum. While we typically advocate a more coarse-grained implementation of FIR filters, this formulation provides a simple illustration of our analysis.

The problem addressed in this chapter is as follows. Suppose that the actors in FIR are running in parallel and the `Source` detects that the weights should be adjusted (e.g., to suite the current operating conditions). Further, to guarantee stability, every output from the system must be obtained using either the old weights or the new ones, but not a mixture of the two. This constraint precludes updating all of the weights at the same instant, as the partial sums within the pipeline would retain evidence of the old weights. Rather, the weights must be changed one actor at a time, mirroring the flow of data through the pipeline. What is a simple and efficient way to implement this behavior?

One way to implement this functionality is by manually tagging each data item with a flag, indicating whether or not it marks the transition to a new set of weights. If it does, then the new set of weights is included with the item itself. While this strategy (shown in Figures 3-2 and 3-5) is functional, it complicates the `Packet` structure with two additional fields – a `newWeights` flag and a `weights` array – the latter of which is meaningful only when `newWeights` is true. This scheme muddles steady-state dataflow with event handling by checking the flag on every invocation of `Multiply` (line 41 of Figure 3-2). It is also very inefficient in StreamIt because arrays are passed by value; though it might be possible to compress each `Packet` when the `weights` field is unused, this would require an aggressive compiler analysis and would also jeopardize other optimizations by introducing an unanalyzable communication rate in the stream graph.

This chapter proposes an alternate solution: teleport messaging. The idea behind teleport messaging is for the `Source` to change the weights via an asynchronous method call, where method invocations in the target actors are timed relative to the flow of data in the stream. As shown in Figure 3-3, the `Multiply` actor declares a message handler that adjusts its own weight (lines 40-42). The `Source` actor calls this handler through a *portal* (line 25), which provides a clean interface for

```
struct Packet {
  float sum;
  float val;
}

void->void pipeline FIR {
  int N = 64;

  add Source(N);
  for (int i=0; i<N; i++)
    add Multiply(i);
  add Printer();
}

void->Packet filter Source(int N) {
  work push 1 {
    Packet p;
    p.sum = 0;
    p.val = readNewData();
    push(p);
  }
}

Packet->Packet filter Multiply(int i,
                              int N) {
  float W = initWeight(i, N);
  Packet last;

  work pop 1 push 1 {
    Packet in = pop();
    last.sum = in.sum + last.val * W;
    push(last);
    last = in;
  }
}

Packet->void filter Printer {
  work pop 1 { print(pop().sum); }
}
```

Figure 3.1: FIR code.

```
   struct Packet<N> {
*    boolean newWeights;
*    float[N] weights;
     float sum;
     float val;
   }

   void->void pipeline FIR {
     int N = 64;

     add Source(N);
     for (int i=0; i<N; i++)
       add Multiply(i, N);
     add Printer();
   }

   void->Packet<N> filter Source(int N) {
     work push 1 {
       Packet p;
       p.sum = 0;
       p.val = readNewData();

*      if (newConditions()) {
*        p.newWeights = true;
*        p.weights = calcWeights();
*      } else {
*        p.newWeights = false;
*      }

       push(p);
     }
   }

   Packet<N>->
   Packet<N> filter Multiply(int i, int N) {
     float W = initWeight(i, N);
     Packet<N> last;

     work pop 1 push 1 {
       Packet<N> in = pop();
*      if (in.newWeights) {
*        W = in.weights[i];
*      }
       last.sum = in.sum + last.val * W;
       push(last);
       last = in;
     }
   }

   Packet<N>->void filter Printer {
     work pop 1 { print(pop().sum); }
   }
```

Figure 3.2: FIR code with manual event handling. Modified lines are marked with an asterisk.

```
   struct Packet {
     float sum;
     float val;
   }

   void->void pipeline FIR {
     int N = 64;
*    portal<Multiply> teleport;

*    add Source(N, teleport);
     for (int i=0; i<N; i++)
*      add Multiply(i, N) to teleport;
     add Printer();
   }

   void->Packet filter
   Source(int N, portal<Multiply> teleport) {
     work push 1 {
       Packet p;
       p.sum = 0;
       p.val = readNewData();
       push(p);

*      if (newConditions())
*        teleport.setWeights(calcWeights());
*    }
   }

   Packet->Packet filter Multiply(int i, int N) {
     float W = initWeight(i, N);
     Packet last;

     work pop 1 push 1 {
       Packet in = pop();
       last.sum = in.sum + last.val * W;
       push(last);
       last = in;
     }

*    handler setWeights(float[N] weights) {
*      W = weights[i]
*    }
   }

   Packet->void filter Printer {
     work pop 1 { print(pop().sum); }
   }
```

Figure 3.3: FIR code with teleport messaging. Modified lines are marked with an asterisk.

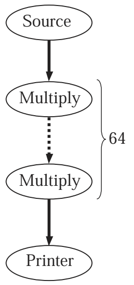


Figure 3.4: FIR stream graph.

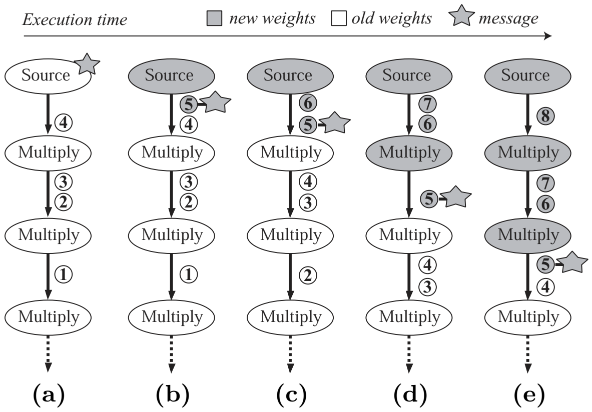


Figure 3.5: Execution snapshots illustrating manual embedding of control messages in FIR. Channels are annotated with data items present on one possible execution; items are numbered in order of production. (a) Source initiates change of weights, (b) weights are attached to data item #5 and embedded in stream, (c)-(e), actors check each input item, adjusting their own weight when they find a tagged item.

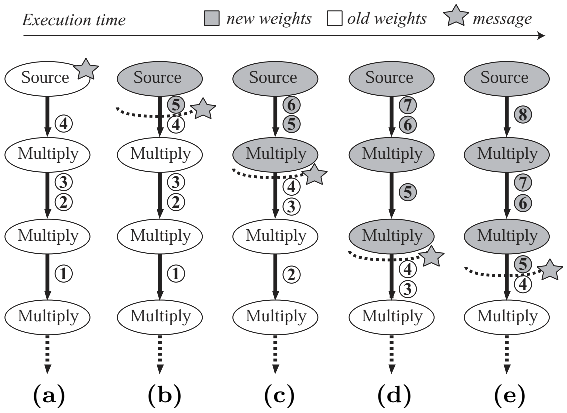


Figure 3.6: Execution snapshots illustrating teleport messaging in FIR. Channels are annotated with data items present on one possible execution; items are numbered in order of production. (a) Source calls a message handler, passing new weights as argument, (b) message boundary is maintained by compiler, (c)-(e), message handler is automatically invoked in actors immediately before the arrival of affected items.

messaging (see Section 3.3). As depicted in Figure 3-6, teleport messaging gives the same result as the manual version, but without corrupting the data structures or control flow used in the steady-state. It also exposes the true information flow, allowing the compiler to deliver the message in the most efficient way for a given architecture. Finally, teleport messaging offers powerful control over timing and latency beyond what is utilized in this example.

The rest of this chapter is devoted to making the above notions more general and more precise. In particular, it is natural to use teleport messaging to send messages upstream – against the flow of data – which is hard to achieve manually. We start by describing a stream dependence function which provides a common timeframe for pairs of dependent actors in the stream graph.

3.2 Stream Dependence Function

This section defines a stream dependence function, SDEP, that describes how one actor depends on the execution of another actor in the stream graph. SDEP is meaningful only for pairs of actors that are connected by a directed path in the stream graph. We say that the *upstream* actor is at the start of the path, while the *downstream* actor is at the end. Dependences between parallel actors (e.g., parallel branches of a splitjoin) currently fall outside the scope of this model but could be addressed in future work (see Section 3.6).

An execution ϕ of a dataflow graph is an ordered sequence of actor firings. Each firing represents the execution of a single phase of the actor. Let $\phi[i]$ denote the ith actor appearing in execution ϕ, and let $|\phi \wedge A|$ denote the number of times that actor A appears in ϕ. An execution is legal if the dataflow requirements are respected; that is, for all i, the sequential firing of actors $\phi[0]$ through $\phi[i-1]$ leaves enough items on the communication channels for $\phi[i]$ to fire its next phase atomically. Let Φ denote the set of legal executions. Note that while Φ is an infinite set, each $\phi \in \Phi$ is a finite sequence.

Informally, $\text{SDEP}_{A \leftarrow B}(n)$ represents the minimum number of times that actor A must execute to make it possible for actor B to execute n times. This dependence is meaningful only if A is upstream of B; otherwise, SDEP assumes a value of zero. Because the I/O rates of each actor are known at compile time, SDEP is a static mapping.

A formal definition of SDEP using the notations introduced above is as follows:

Definition 1. *(SDEP)*

$$\text{SDEP}_{A \leftarrow B}(n) \quad = \quad \min_{\substack{\phi \in \Phi, \\ |\phi \wedge B| = n}} |\phi \wedge A|$$

This equation reads: over all legal executions in which B fires n times, $\text{SDEP}_{A \leftarrow B}(n)$ is the minimum number of times that A fires. Figure 3-8 illustrates an example of SDEP for the stream graph in Figure 3-7.

Calculating SDEP

It is straightforward to calculate $\text{SDEP}_{A \leftarrow B}(n)$ via a fine-grained simulation of the stream graph. Our approach is to construct an execution ϕ that provides the minimum value of $|\phi \wedge A|$ that is selected in Definition 1. We construct ϕ by simulating the stream graph's execution of a *pull schedule* with respect to actor B.

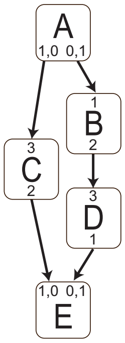
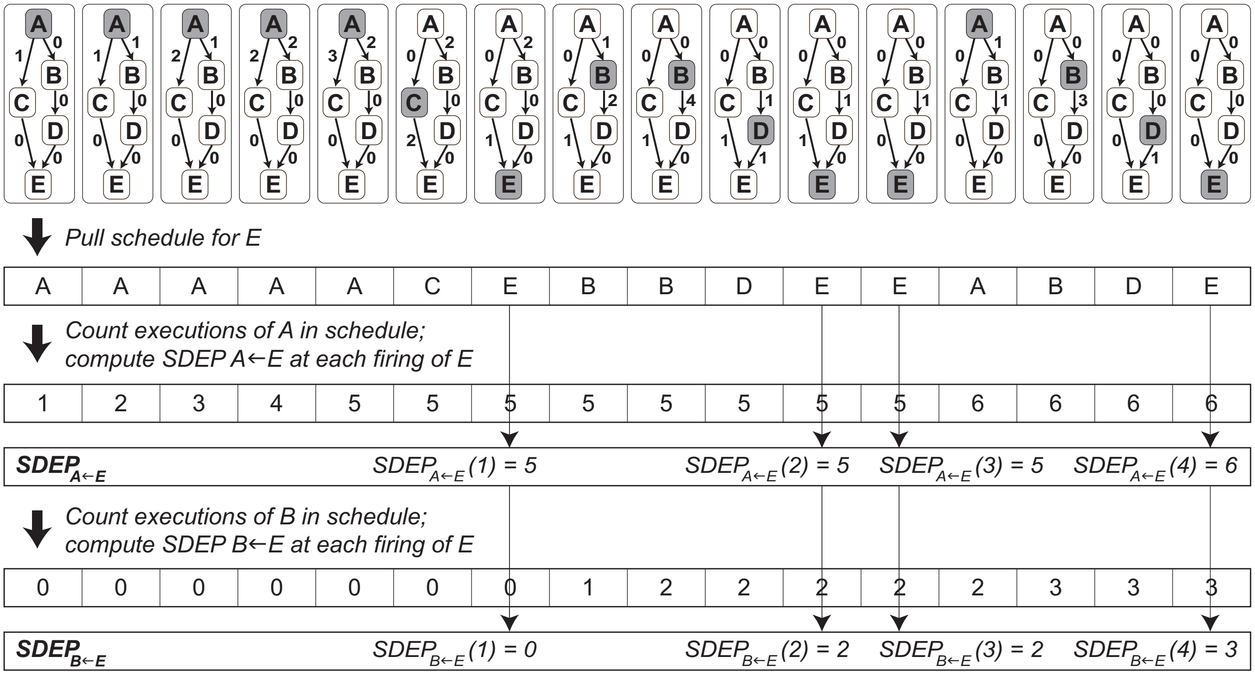


Figure 3-7: Example stream graph. Nodes are annotated with their I/O rates. For example, node C consumes 3 items and produces 2 items on each execution. Node A is a round-robin splitter that produces one item on its left channel during the first phase, and one item on its right channel during the second phase (similarly for Node E).


Pull schedule for E
Count executions of A in schedule; compute SDEP A←E at each firing of E
Count executions of B in schedule; compute SDEP B←E at each firing of E

A	A	A	A	A	C	E	B	B	D	E	E	A	B	D	E
1	2	3	4	5	5	5	5	5	5	5	5	6	6	6	6

SDEP$_{A\leftarrow E}$ — *SDEP*$_{A\leftarrow E}$*(1) = 5* — *SDEP*$_{A\leftarrow E}$*(2) = 5* — *SDEP*$_{A\leftarrow E}$*(3) = 5* — *SDEP*$_{A\leftarrow E}$*(4) = 6*

0	0	0	0	0	0	0	1	2	2	2	2	2	3	3	3

SDEP$_{B\leftarrow E}$ — *SDEP*$_{B\leftarrow E}$*(1) = 0* — *SDEP*$_{B\leftarrow E}$*(2) = 2* — *SDEP*$_{B\leftarrow E}$*(3) = 2* — *SDEP*$_{B\leftarrow E}$*(4) = 3*

Figure 3-8: Example *SDEP* calculation for stream graph in Figure 3-7. The stream graphs illustrate a steady state cycle of a "pull schedule"; execution proceeds from left to right, and channels are annotated with the number of items present. The second line lists the actors that fire in a pull schedule for E. The third line counts the number of times that A executes in the pull schedule, and the fourth line illustrates the computation of *SDEP*$_{A\leftarrow E}(n)$: the number of times that A executes before the nth execution of E. The last two lines illustrate the computation of *SDEP*$_{B\leftarrow E}$.

```
// Returns a pull schedule for $n$ executions of $X$
pullSchedule($X$, $n$) {
  $\phi = \{\}$
  for $i$ = 1 to $n$ {
    // execute predecessors of $X$ until $X$ can execute
    for all input channels $c_i$ of $X$
      while $X$ needs more items on $c_i$ in order to fire
        // extend schedule ($\circ$ denotes concatenation)
        $\phi = \phi \circ$ pullSchedule(source($c_i$), 1)
    // add $X$ to schedule
    $\phi = \phi \circ X$
    // update number of items on I/O channels of $X$
    simulateExecution($X$)
  }
  return $\phi$
}
```

Figure 3-9: Pull scheduling algorithm.

Pull scheduling is defined in Figure 3-9. Intuitively, a pull schedule for X is one that executes other nodes as few times as possible for each firing of X. This is achieved by calculating the demand for data items on the input channels of X, and then propagating the demand back through the stream graph via pull scheduling of the actors connected to X. Pull scheduling results in a fine-grained interleaving of actor firings. Some stream graphs admit multiple pull schedules, as actors might be connected to multiple inputs that can be scheduled in any order; however, the set of actor executions remains constant even as the order changes. The following theorem allows us to use a pull schedule to calculate the SDEP function.

Theorem 1.

$$\text{SDEP}_{A \leftarrow B}(n) = |\mathbf{pullSchedule}(B, n) \wedge A|$$

Proof. By construction, $\mathbf{pullSchedule}(B, n)$ executes each node in the graph as few times as possible for B to fire n times. Thus, there is no execution containing n executions of B where A executes fewer times. The theorem follows from the definition of SDEP. □

Some example SDEP calculations appear in Figure 3-8. The results are summarized in the following table.

n	$\text{SDEP}_{A \leftarrow E}(n)$	$\text{SDEP}_{B \leftarrow E}(n)$
1	5	0
2	5	2
3	5	2
4	6	3

Note that SDEP is non-linear due to mis-matching I/O rates in the stream graph. However, for longer execution traces, there is a pattern in the marginal growth of SDEP (i.e., in $\text{SDEP}(n) - \text{SDEP}(n-1)$); this quantity follows a cyclic pattern and has the same periodicity as the steady state of the stream graph. A steady state $\mathcal{S} \in \Phi$ is an execution that does not change the buffering in the channels – that is, the number of items on each channel after the execution is the same as it was before the execution. Calculating a steady state is well-understood [LM87]. The execution simulated in Figure 3-8 is a steady state, and in this particular example, additional entries of the pull schedule repeat the pattern given in the figure. This means that SDEP also grows in the same pattern, and we can calculate $\text{SDEP}_{A \leftarrow E}(n)$ for $n > 4$ as follows[1]:

$$\begin{aligned} \text{SDEP}_{A \leftarrow E}(n) &= p(n) * |\mathcal{S} \wedge A| + \\ &\qquad \text{SDEP}_{A \leftarrow E}(n - p(n) * |\mathcal{S} \wedge E|) \qquad (1) \\ p(n) &= \lfloor \tfrac{n}{|\mathcal{S} \wedge E|} \rfloor \qquad (2) \end{aligned}$$

where $\mathcal{S}$ is a steady state and $p(n)$ represents the number of steady states that E has completed by iteration n. The first term of Equation 1 gives the total number of times that A has fired in previous steady states, while the second term counts firings of A in the current steady state.

While Equation 1 works for actors A and E, it fails for certain corner cases in stream graphs. For example, for $\text{SDEP}_{A \leftarrow C}(3)$ it detects exactly 3 steady state executions $(p(3) = 3)$ and concludes that each requires 6 executions of A ($|\mathcal{S} \wedge A| = 6$). However, as shown in Figure 3-8, the last firing of C requires only 5 executions of A. C is unusual in that it finishes its steady state before the upstream actor A.

To handle the general case, we need to change the base case in two ways. First, we include the initialization schedule $\mathcal{I}$, which fires prework functions and fills buffers needed by peeking filters; SDEP during initialization is different than SDEP in the steady state. Second, to solve the problem above, we simulate *two* executions of the steady state (rather than one) for the base case of SDEP:

$$\text{SDEP}_{Y \leftarrow X}(n) = \qquad (3)$$
$$\begin{cases} |\textbf{pullSchedule}(X, n) \wedge Y| & \text{if } n \leq |\mathcal{I} \wedge X| + 2 * |\mathcal{S} \wedge X| \\ q(n) * |\mathcal{S} \wedge Y| + & \text{otherwise} \\ \quad \text{SDEP}_{Y \leftarrow X}(n - q(n) * |\mathcal{S} \wedge X|) & \end{cases}$$

$$q(n) = \lfloor \tfrac{n - |\mathcal{I} \wedge X|}{|\mathcal{S} \wedge X|} \rfloor - 1 \qquad (4)$$

In this formulation, the last complete steady state is counted as part of the "current" iteration rather than a "completed" iteration. For example, Equation 3 evaluates $\text{SDEP}_{A \leftarrow C}(3)$ using $q(3) = 2$, yielding $\text{SDEP}_{A \leftarrow C}(3) = 2 * 6 + \text{SDEP}_{A \leftarrow C}(3 - 2 * 1) = 17$ as desired. Moreover, in complex cases[2], the last steady state adds important context to the SDEP lookup for a given execution.

Thus, to calculate $\text{SDEP}_{Y \leftarrow X}(n)$, it is not necessary to simulate a pull schedule for n iterations of X as described in Figure 3-9. Instead, one can simulate $|\mathcal{I} \wedge X| + 2 * |\mathcal{S} \wedge X|$ iterations as a pre-processing step and answer all future SDEP queries in constant time, using Equation 3. In addition, the pull schedule for X can be reused to calculate SDEP from X to any other actor (e.g., $\text{SDEP}_{W \leftarrow X}$ in addition to $\text{SDEP}_{Y \leftarrow X}$).

[1] Note that for any two actors X and Y, $\text{SDEP}_{Y \leftarrow X}(0) = 0$.

[2] For example, if within each steady state, the first firing of X does not depend on the first firing of Y, and the last firing of X does not depend on the last firing of Y.

However, note that the pull schedule for X can *not* be used to calculate SDEP from any actor other than X (e.g., $\text{SDEP}_{W \leftarrow Y}$). The guarantee provided by $\textbf{pullSchedule}(X, n)$ is only with respect to the base actor X. For other pairs of actors in the graph, one actor might execute more than necessary for n executions of the other. For example, consider what happens if one calculates $\text{SDEP}_{A \leftarrow B}$ using the schedule in Figure 3-8 (which is a pull schedule for E). In the schedule, A executes 5 times before the first firing of B, so one would conclude that $\text{SDEP}_{A \leftarrow B}(1) = 5$. However, this is incorrect; since B could have fired after only 2 executions of A, the correct value is $\text{SDEP}_{A \leftarrow B}(1) = 2$. Thus, to calculate $\text{SDEP}_{Y \leftarrow X}$, it is essential to calculate $\textbf{pullSchedule}(X, |\mathcal{S} \wedge X|)$, that is, a steady state cycle of a pull schedule with respect to X.

It is also possible to calculate SDEP using a compositional approach. For example, $\text{SDEP}_{A \leftarrow E}$ from Figure 3-8 can be expressed as follows:

$$\text{SDEP}_{A \leftarrow E}(n) = \max \begin{cases} \text{SDEP}_{A \leftarrow B}(\text{SDEP}_{B \leftarrow E}(n)) \\ \text{SDEP}_{A \leftarrow C}(\text{SDEP}_{C \leftarrow E}(n)) \end{cases}$$

That is, to determine the minimum number of times that A must execute to enable n executions of E, first calculate the minimum number of times each of A's successors in the stream graph must execute for n executions of E. Then A must execute enough to enable all of these children to complete the given number of executions, which translates to the $\max$ operation shown above. Our implementation exploits this compositional property to tabulate SDEP in a hierarchical manner, rather than simulating a pull schedule.

3.3 Semantics of Messaging

Teleport messaging is a language construct that makes use of SDEP to achieve precise timing of control messages. Teleport messaging represents out-of-band communication between two actors, distinct from the high-bandwidth dataflow in the stream graph. Messages are currently supported between any pair of actors with a meaningful SDEP relationship, i.e., wherever there is a directed path in the stream graph from one actor to the other. We say that a *downstream* message travels in the same direction as the steady-state data flow, whereas an *upstream* message travels against it.

Syntax

In order for actor of type A to send a message to actor of type B, the following steps need to be taken:

- B declares a message handler that is invoked when a message arrives. For example:

  ```
  handler increaseGain(float amount) {
    this.gain += amount;
  }
  ```

  Message handlers are akin to normal functions, except that they cannot access the input/output channels and they do not return values.

  For another example, see line 40 of Figure 3-3.

- A parent stream containing instances of A and B declares a variable of type `portal<`B`>` that can forward messages to one or more actors of type B. The parent adds an instance of B to the portal and passes the portal to an instance of A during initialization.

  For example, see lines 8, 10 and 12 of Figure 3-3.

- To send a message, A invokes the handler method on the portal from within its steady-state work function. The handler invocation includes a latency k specifying when the message should be delivered; if no latency is specified, then a default latency of 0 is used. The following illustrates an example.

  ```
  work pop 1 {
    float val = pop();
    if (val < THRESHOLD) {
      portalToB.increaseGain(0.1) @ 2;
    }
  }
  ```

  This code sends an `increaseGain` message to `portalToB` with latency 2.

  For another example, see line 25 of Figure 3-3.

Informal Semantics

The most interesting aspect of teleport messaging is the semantics for the message latency. Because there are many legal orderings of actor executions, there does not exist a notion of "global time" in a stream graph. The only common frame of reference between concurrently executing actors is the series of data items that is passed between them.

Intuitively, the message semantics can be thought of in terms of attaching tags to data items. If A sends a message to downstream actor B with a latency k, then this could be implemented by tagging the items that A outputs k iterations later. These tags propagate through the stream graph; whenever an actor inputs an item that is tagged, all of its subsequent outputs are tagged. Then, the message handler of B is invoked immediately before the first invocation of B that inputs a tagged item. In this sense, the message has the semantics of traveling "with the data" through the stream graph, even though it is not necessarily implemented this way.

The intuition for upstream messages is similar. Consider that B is sending a message with latency k to upstream actor A in the stream graph. This means that A will receive the message immediately after its last invocation that produces an item affecting the output of B's kth firing, counting the current firing as 0. As before, we can also think of this in terms of A tagging items and B observing the tags. In this case, the latency constraint says that B must input a tagged item before it finishes k additional executions. The message is delivered immediately after the latest firing in A during which tagging could start without violating this constraint.

Formal Semantics

The SDEP function captures the data dependences in the graph and provides a natural means of defining a rendezvous point between two actors. The following definition leverages SDEP to give a precise meaning to message timing.

Definition 2. *(Message delivery) Consider that S sends a message to receiver R with latency k. There are two cases*[3]*:*

1. *If R is downstream of S, then the message handler is invoked in R immediately before its mth execution, where m is determined as follows:*

$$m = \min\ m' \ s.t.\ \text{SDEP}_{S \leftarrow R}(m') \geq n + k$$

2. *If R is upstream of S, then the message handler is invoked in R immediately after its mth execution, where m is determined as follows:*

$$m = \text{SDEP}_{R \leftarrow S}(n + k)$$

The first case reads differently than the second case because the SDEP function is neither injective nor surjective. That is, for given values of n and k, there may exist either zero, one, or many values of m for which $\text{SDEP}_{S \leftarrow R}(m) = n+k$. This property is illustrated by example in Figure 3-8. If there does not exist an m for which $\text{SDEP}_{S \leftarrow R}(m) = n + k$, then the message is delivered at the smallest value of m for which $\text{SDEP}_{S \leftarrow R}(m) \geq n + k$. Similarly, if there exist multiple values of m for which $\text{SDEP}_{S \leftarrow R}(m) = n + k$, then the message is delivered before the first satisfying iteration. The formula for upstream message delivery is more simple, because the SDEP function directly provides the unique, latest iteration of the upstream actor that affected iteration $n + k$ of the downstream actor.

As an example of message timing, consider the FIR code in Figure 3-3. On line 25, the `Source` sends a message to the `Multiply` actors with latency zero. Consider that, as illustrated in Figure 3-6, a message is sent during the fifth execution of `Source` ($n = 5$). Because each `Multiply` is downstream of `Source`, we can calculate the delivery time as follows:

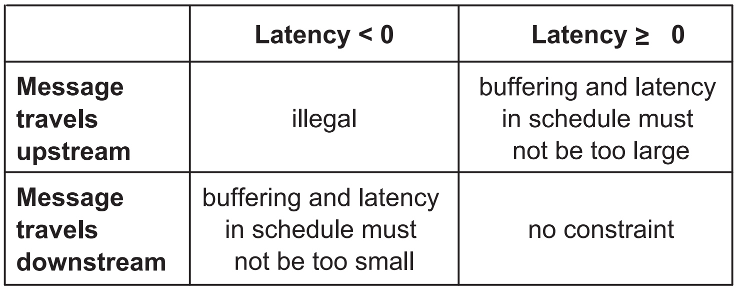
$$m = \min m' \ s.t.\ \text{SDEP}_{Source \leftarrow Multiply}(m') \geq n + k$$
$$m = \min m' \ s.t.\ \text{SDEP}_{Source \leftarrow Multiply}(m') \geq 5$$
$$m = \min m' \ s.t.\ m' \geq 5$$
$$m = 5$$

To calculate $\text{SDEP}_{Source \leftarrow Multiply}$, observe that `Source` produces one item per iteration, while each `Multiply` produces one item and consumes one item. Thus, the `Source` must fire m times before any given `Multiply` can execute m times, and $\text{SDEP}_{Source \leftarrow Multiply}(m) = m$. Substituting into the above equation yields $m = 5$. That is, the message is delivered to each `Multiply` immediately before its fifth execution. This is illustrated in Figures 3-6(c) and 3-6(d) for the first and second `Multiply` in the pipeline, respectively. The message arrives immediately before the fifth data item (which corresponds to the fifth execution).

Constraints on the Schedule

It is important to recognize that messaging can place constraints on the execution schedule. The different categories of constraints are illustrated in Figure 3-10. A negative-latency downstream

[3] In a feedback path, both cases might apply. In this event, we assume the message is being sent upstream.



	Latency < 0	Latency ≥ 0
Message travels upstream	illegal	buffering and latency in schedule must not be too large
Message travels downstream	buffering and latency in schedule must not be too small	no constraint

Figure 3-10: Scheduling constraints imposed by messages.

message has the effect of synchronizing the arrival of the message with some data that was previously output by the sender (e.g., for the checksum example mentioned in the introduction). The latency requires the downstream receiver not to execute too far ahead (i.e., too close to the sender), or else it might process the data before the message arrives. This translates to a constraint on the minimum allowable latency between the sender and receiver actors in the schedule for the program. Intuitively, it also constrains the buffering of data: the data buffers must not grow too small, as otherwise the receiver would be too far ahead.

Similarly, a non-negative-latency upstream message places a constraint on the maximum allowable latency between the sender and receiver. This time the upstream actor must be throttled so that it does not get too far ahead before the message arrives. Intuitively, the amount of data buffered between the actors must not grow too large.

For upstream messages with negative latency, there always exist iterations of the sender during which any messages sent are impossible to deliver. Consider an iteration of the sender that is the first to depend on data propagating from the nth execution of the receiver. A negative-latency message would be delivered immediately after a *previous* iteration of the receiver, but since iteration n has already fired, the message is impossible to deliver. Conversely, a downstream message with positive or zero latency imposes no constraint on the schedule, as the sender has not yet produced the data that is synchronized with the message.

Unsatisfiable Constraints

Messaging constraints can be unsatisfiable – that is, assuming a message is sent on every iteration of the sender's work function, there does not exist a schedule that delivers all of the messages within the desired latency range. Such constraints should result in a compile-time error.

Figure 3-11 illustrates an example of unsatisfiable constraints. Though each messaging constraint is feasible in isolation, the set of constraints together is unsatisfiable. The unsatisfiability is caused by conflicting demands on the buffering between B and C. The message from B to C constrains this buffer to contain at least 10 items, while the message from D to A constrains it to be empty.

It should be noted that downstream messages with negative latency are always unsatisfiable at the beginning of execution, because no buffering has been established in the data channels. Any messages sent during this period are suspended and re-sent as soon as the graph is initialized with buffers that satisfy the message constraints.

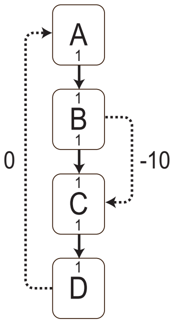


Figure 3-11: Example of unsatisfiable message constraints. Each node is annotated with its input and output rate. Messages are shown by dotted arrows, drawn from sender to receiver with a given latency. The constraints are satisfiable in isolation, but unsatisfiable in combination.

Finding a Schedule

To schedule a stream graph in the presence of messaging constraints, a simple greedy algorithm can be used. As shown in Figure 3-12, this *constrained scheduling* algorithm is a variation on pull scheduling. Like pull scheduling, constrained scheduling also derives a fine-grained schedule with minimal latency. To incorporate the presence of message constraints, two extensions are made. First, before firing an actor, the algorithm ensures that this firing will not cause the actor to miss any messages that were intended for it prior to the given execution. This is done by considering all message senders that target the given actor, and recursively scheduling them until they complete all executions that may send messages with receipt at the current time. Second, at the beginning of the procedure, there is a check for an infinite loop (whereby the current actor is also being scheduled higher on the call stack). This indicates a case of unsatisfiable constraints, as an actor's execution depends on itself.

This algorithm is guaranteed to find a valid schedule if one exists. A filter is fired if and only if that firing is needed to satisfy a data or message dependence. If this fine-grained execution is still too coarse-grained to satisfy message constraints, then there is a cyclic dependence and the constraints are unsatisfiable.

While the algorithm presented derives a single sequence of filter executions, in practice it is desirable to separate that sequence into two parts: an initialization schedule (executed once) and a steady-state schedule (executed repeatedly). This can be achieved by using a standard algorithm to detect a periodicity in the constrained schedule; any repeated pattern of filter firings that preserves the number of items on each data channel is a valid steady state. The initialization schedule consists of whatever is leftover before the first steady-state execution.

In practice, it is also worthwhile to compress the schedule before generating code. Such compression can be achieved easily once the full initialization schedule and steady-state multiplicities have been derived via the above technique. Following initialization, a compressed schedule considers filters in order from top to bottom and executes each filter as long as possible, until either 1) the filter runs out of input items, 2) further execution would violate a message constraint, or 3) the filter meets its steady-state execution multiplicity. This strategy is valid because it is already known that there exists a sequence of filer firings with the given multiplicities (and with the given starting configuration) that satisfies the constraints. Thus execution will be able to make forward progress until all filters have completed their steady state.

```
// Returns a modified pull schedule for 1 execution of $X$, never
// firing a node that would further violate a message constraint.
constrainedSchedule($X$) {
    // check for infinite loop, which indicates unsatisfiable constraints
    if call stack contains call to constrainedSchedule($X$) then
        report that message constraints are unsatisfiable

    $\phi = \{\}$
    // execute predecessors of $X$, based on data dependences
    for all input channels $c_i$ of $X$
        while $X$ needs more items on $c_i$ in order to fire
            // extend schedule ($\circ$ denotes concatenation)
            $\phi = \phi \circ$ constrainedSchedule(source($c_i$))

    // execute predecessors of $X$, based on message dependences
    for all filters $F$ that might, on a future execution, send a
                message to $X$ for delivery prior to next firing of $X$
        $\phi = \phi \circ$ constrainedSchedule($F$)

    // add $X$ to schedule
    $\phi = \phi \ \circ \ X$
    // update number of items on I/O channels of $X$
    simulateExecution($X$)
    return $\phi$
}
```

Figure 3-12: Constrained scheduling algorithm.

Despite its simplicity, we have yet to evaluate this scheduling algorithm in the StreamIt compiler. As described in Section 3.4, our compiler targets a parallel machine in which each sender and receiver executes in its own thread and waits for possible messages at appropriate iterations. This approach does not depend on producing a serial ordering of the actors at compile time.

3.4 Case Study

To illustrate the pros and cons of teleport messaging, we implemented a spread-spectrum frequency hopping radio frontend [HP02] as shown in Figure 3-13. (Matthew Drake also describes the implications of messaging in a case study of MPEG-2 [MDH^{+}06, Dra06].) A frequency hopping radio is one in which the receiver switches between a set of known frequencies whenever it detects certain tones from the transmitter. The frequency hopping is a good match for control messages because the hopping interval is dynamic (based on data in the stream); it spans a large section of the stream graph (there is a Fast Fourier Transform (FFT) with 15 child actors, not shown, between the demodulator and the hop detector); and it requires precise message delivery. The delivery must

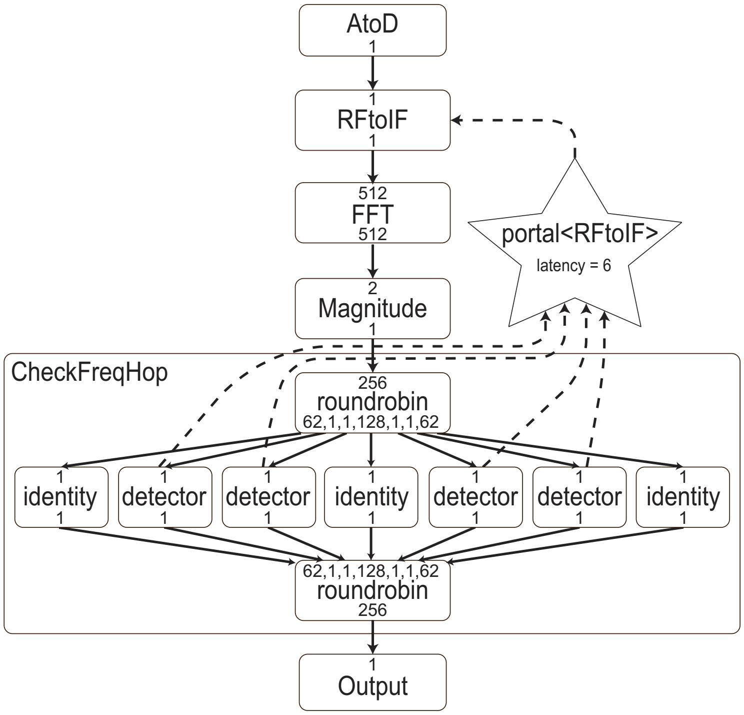


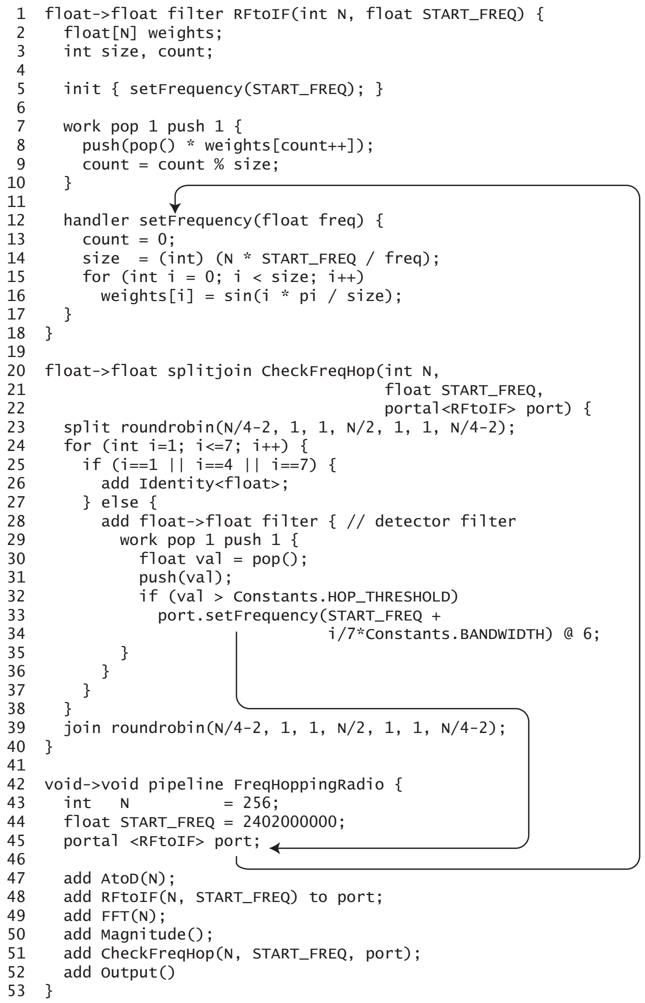
Figure 3-13: Stream graph of frequency hopping radio with teleport messaging. A portal delivers point-to-point latency-constrained messages from the detectors to the RFtoIF stage.

be precise both to meet real-time requirements (as the transmitter will leave the current frequency soon), and to ensure that the message falls at a logical frame boundary; if the frequency change is out of sync with the FFT, then the FFT will muddle the spectrum of the old and new frequency bands.

A StreamIt version of the radio frontend with teleport messaging appears in Figure 3-14. The FreqHoppingRadio pipeline creates a portal and adds the RFtoIF actor as a receiver (lines 45 and 48 respectively). The portal is passed to the CheckFreqHop stage, where four parallel detectors send messages into the portal if they detect a hop in the frequency they are monitoring (lines 32-35). The messages are sent with a latency of 6 to ensure a timely transition. To make sense of the latency, note that $\text{SDEP}_{RFtoIF \leftarrow D}(n) = 512 * n$ for each of the detector actors D. This comes about because the FFT stage consumes and produces 512 items[4]; each detector fires once per set of outputs from the FFT, but RFtoIF fires 512 times to fill the FFT input. Because of this SDEP relationship, messages sent from the detectors to RFtoIF are guaranteed to arrive only at iterations that are a multiple of 512. This satisfies the design criterion that a given FFT stage will not operate on data that were demodulated at two separate frequencies.

Another version of the frequency hopping radio appears in Figures 3-15 and 3-16. This version is functionally equivalent to the first, except that the control messages are implemented manually by embedding them in the data stream and introducing a feedback loop. Because the number of items transfered around the loop must be constant from one iteration to the next, a data item is sent whether or not there is a message as part of the algorithm. The RFtoIF filter checks the values from the loop on every iteration; if the value is non-zero, it is treated as a message (the new frequency),

[4] Though the FFT is 256-way, the real and imaginary parts are interleaved on the tape, leading to an I/O rate of 512.



```
float->float filter RFtoIF(int N, float START_FREQ) {
  float[N] weights;
  int size, count;

  init { setFrequency(START_FREQ); }

  work pop 1 push 1 {
    push(pop() * weights[count++]);
    count = count % size;
  }

  handler setFrequency(float freq) {
    count = 0;
    size  = (int) (N * START_FREQ / freq);
    for (int i = 0; i < size; i++)
      weights[i] = sin(i * pi / size);
  }
}

float->float splitjoin CheckFreqHop(int N,
                                    float START_FREQ,
                                    portal<RFtoIF> port) {
  split roundrobin(N/4-2, 1, 1, N/2, 1, 1, N/4-2);
  for (int i=1; i<=7; i++) {
    if (i==1 || i==4 || i==7) {
      add Identity<float>;
    } else {
      add float->float filter { // detector filter
        work pop 1 push 1 {
          float val = pop();
          push(val);
          if (val > Constants.HOP_THRESHOLD)
            port.setFrequency(START_FREQ +
                              i/7*Constants.BANDWIDTH) @ 6;
        }
      }
    }
  }
  join roundrobin(N/4-2, 1, 1, N/2, 1, 1, N/4-2);
}

void->void pipeline FreqHoppingRadio {
  int   N          = 256;
  float START_FREQ = 2402000000;
  portal <RFtoIF> port;

  add AtoD(N);
  add RFtoIF(N, START_FREQ) to port;
  add FFT(N);
  add Magnitude();
  add CheckFreqHop(N, START_FREQ, port);
  add Output()
}
```

Figure 3-14: Frequency hopping radio with teleport messaging. Arrows depict the path of messages from the sender to the receiver, via a portal declared in the top-level stream.

while a value of zero is ignored (no message). The I/O rate of the RFtoIF filter has been scaled up to ensure that the messaging information is received at intervals of 512 iterations (as in the version with portals). To achieve the desired messaging latency of 6 frames, $6 * 256 = 1536$ items are enqueued on the feedback path prior to execution.

Discussion

Teleport messaging offers several benefits compared to a manual implementation of equivalent functionality. While embedding messages in the data stream is equally precise, it involves several tedious and error-prone changes, not only to the stream graph but also to the steady-state execution code within the actors. In particular, the manual derivation of the loop delay, adjustment of the actor I/O rates, and implicit interleaving of data items with control messages has a negative impact on the readability and maintainability of the code. Teleport messaging provides the same level of precision, but with the simplicity of a method call.

Teleport messaging also has advantages from a compiler standpoint. By separating the data-

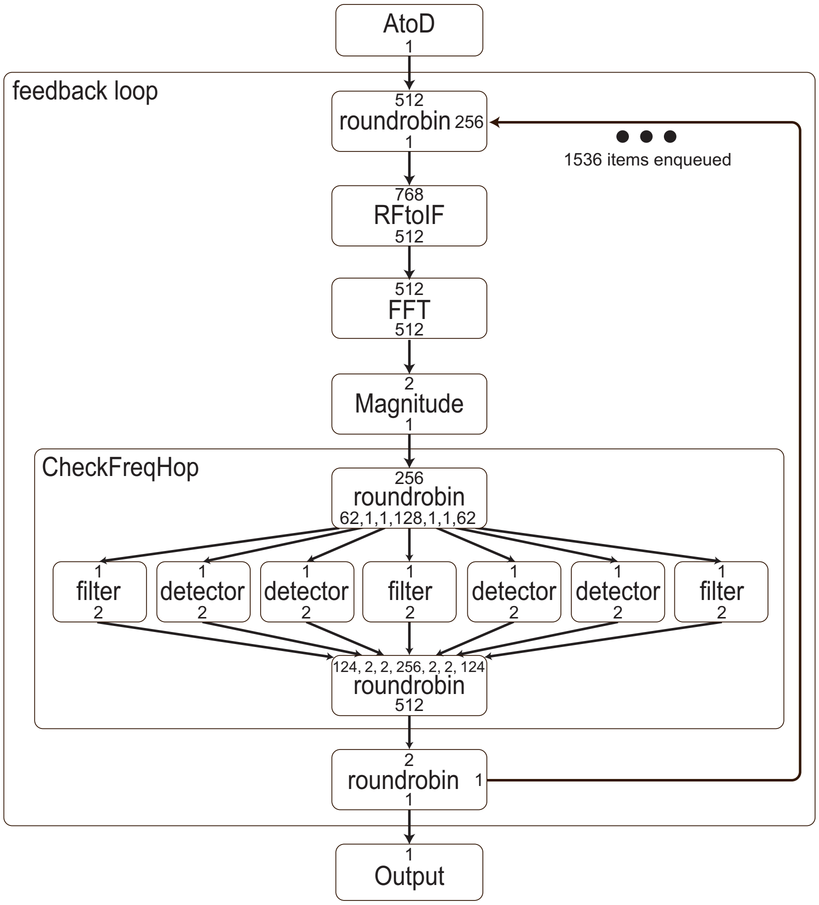


Figure 3-15: Stream graph of frequency hopping radio with control messages implemented manually. A feedback loop connects the detectors with the RFtoIF stage, and an item is sent on every invocation to indicate whether or not a message is present. The latency and periodicity of message delivery are governed by the data rates and the number of items on the feedback path.

intensive code from the control-oriented code, the common case of steady-state execution is not sacrificed for the uncommon case of message processing. There are no “dummy items” serving as placeholders in the static-rate channels. In addition, by exposing the message latency as part of the language, the compiler can infer the true dependences between actor firings and reorder the execution so long as the message constraints are respected. The actual message delivery can be implemented in the most efficient way for a given architecture.

A final benefit of teleport messaging is the clean interface provided by the portals. Since a portal can have multiple receivers, it is straightforward to send a message that is delivered synchronously to two actors in parallel streams. For example, consider a vocoder (an encoder for voice signals) that is separately manipulating the magnitude and phase components of a signal. If something triggers an adjustment to the speech transformation (e.g., the speaker requests a change of pitch) then the mask needs to be updated at the same time relative to data in both parallel streams. A portal that contains both components seamlessly provides this functionality. Finally, portals are useful as an external programming interface; an application can export a portal based on an interface type without exposing the underlying actor implementation.

```
 float->float filter RFtoIF(int N, float START_FREQ) {
   float[N] weights;
   int size, count;

   init { setFrequency(START_FREQ); }

*  work pop 3*N push 2*N {
*    // manual loop to 2*N.  Factor of N because messages
*    // for given time slice come in groups of N; factor
*    // of 2 for data-rate conversion of Magnitude filter
*    for (int i=0; i<2*N; i++) {
*      push(pop() * weights[count++]);
*      count = count % size;
*    }
*    // manually check for messages;
*    // special value of 0 encodes no message
*    for (int i=0; i<N; i++) {
*      float freqHop = pop();
*      if (freqHop!=0)
*        setFrequency(freqHop);
*    }
*  }

   handler setFrequency(float freq) {
     count  = 0;
     size   = (int) (N * START_FREQ / freq);
     for (int i = 0; i < size; i++)
       weights[i] = sin(i * pi / size);
   }
 }

 float->float splitjoin CheckFreqHop(int N,
                                     float START_FREQ) {
   split roundrobin(N/4-2, 1, 1, N/2, 1, 1, N/4-2);
   for (int i=1; i<=7; i++) {
     if (i==1 || i==4 || i==7) {
       add float->float filter {
*        work pop 1 push 2 {
           push(pop());
*          push(0);
         }
       }
     } else {
       add float->float filter { // detector filter
*        work pop 1 push 2 {
           float val = pop();
           push(val);
*          if (val > Constants.HOP_THRESHOLD) {
*            push(START_FREQ + i/7*Constants.BANDWIDTH);
*          } else {
*            push(0);
*          }
         }
       }
     }
   }
*  join roundrobin(2*(N/4-2), 2, 2, 2*(N/2), 2, 2, 2*(N/4-2));
 }

 void->void pipeline FreqHoppingRadio {
   int   N             = 256;
   float START_FREQ    = 2402000000;

   add AtoD(N);
*  add float->float feedbackloop {
*    // adjust joiner rates to match data rates in loop
*    join roundrobin(2*N,N);
*    body pipeline {
*      add RFtoIF(N, START_FREQ);
*      add FFT(N);
*      add Magnitude();
*      add CheckFreqHop(N, START_FREQ);
*    }
*    split roundrobin();
*    // number of items on loop path = latency * N
*    for (int i=0; i<6*N; i++)
*      enqueue(0);
*  }
   add Output()
 }
```

Figure 3-16: Frequency hopping radio with manual feedback loop for event handling. Lines that differ from Figure 3-14 are marked with an asterisk.

One aspect of teleport messaging might be considered unusual: the granularity of message delivery can be affected by changes in granularity elsewhere in the stream graph. This is evident in the frequency hopping radio, as the I/O rate of 512 on the FFT implies that the RFToIF stage will receive messages from CheckFreqHop at most once every 512 iterations. (If the FFT were coarsened to 1024-way, the granularity of messages in RFToIF would increase accordingly.) In this case the behavior is desirable, as messages should not interrupt frame boundaries. It seems that in many cases, the I/O rates are meaningful aspects of the program and their influence on message granularity is appropriate. Nonetheless, this non-local influence might come as a surprise to programmers. If the FFT granularity is scaled up for a different reason (e.g., caching behavior), the effects on message granularity might be unwanted.

This suggests that it might be worthwhile, in future work, to investigate additional mechanisms for programmers to specify the messaging contract independently of the declared I/O rates. For example, a parent stream could override the I/O rates of a child for the sake of a given SDEP calculation. The scheduler would deliver messages according to the parent's expectation of SDEP, or report an error if such delivery is incompatible with the actual I/O rates.

Experimental Evaluation

We have implemented teleport messaging in the StreamIt compiler infrastructure, with a backend that targets a cluster of workstations. A StreamIt program is compiled to a set of parallel threads; if two threads are allocated to different machines, they communicate via dedicated TCP/IP connections. Messages are supported via auxiliary communication channels that transmit two kinds of signals from senders to receivers: 1) the contents of a control message, or 2) a *credit* that indicates the receiver can execute some number of iterations before checking for a message again.

Each actor alternates between normal execution and checking for the exchange of credits. This serves to throttle the message receiver in accordance with the constraints (Section 3.3), as an actor will block waiting for credits until the sender has reached a given point in its execution. The compiler calculates the SDEP information and schedules the exchange of credits to make sure that the timing constraints are respected. When a message is sent, it is tagged with the iteration number during which the receiver should process it; this is also calculated using SDEP in the compiler.

We chose a cluster-based evaluation for two reasons. First, many streaming applications run on the server side (e.g., cell phone base stations, radar processing, HDTV editing) and require large computational resources. Second, clusters provide a simple abstraction for distributed and parallel computing – multiple program counters, and distributed memories – which is at the heart of emerging multicore architectures for embedded, desktop, and server computing.

The teleport implementation of the frequency hopping radio was compiled into 29 threads whereas the alternate version using a feedback loop results in 33 threads. Each thread corresponds to a single actor (there are more threads than appear in Figures 3-13 and 3-15 because the FFT stage is a pipeline composed of several actors). The thread mapping is done using a dynamic programming algorithm that aims to reduce the overall bottleneck, thereby maximizing throughput (outputs per unit time). Threads are assigned to one of sixteen 750Mhz Pentium III workstations, each with a 256Kb cache. The machines are interconnected using a fully switched 100Mb network.

Figure 3-17 shows the measured throughput (y-axis) for various cluster sizes. Note that due to the limited parallelism in the two implementations of the frequency hopper, cluster configurations with more than five workstations lead to negligible performance gains. From the data, we

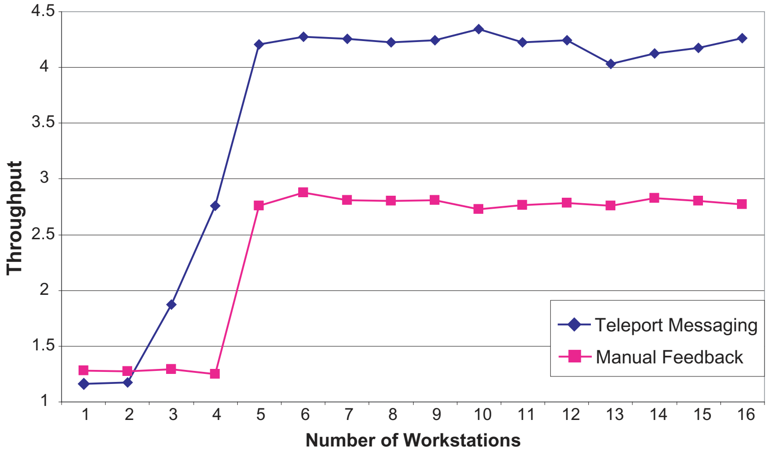


Figure 3-17: Parallel performance of teleport messaging and manual event handling.

can observe that teleport messaging achieves a maximal throughput that is 49% better than its counterpart. We attribute this speedup primarily to reduced communication overhead. A detailed analysis of the results indicates that teleport messaging reduces the number of items communicated by 35%. While the feedback loop version sends a message placeholder on every iteration, teleport messaging uses credits to allow the receiver to execute several iterations at a time without checking for messages. The amount of communications savings is dictated by the message latency, as larger latencies allow for a less frequent exchange of credits.

3.5 Related Work

The work most closely related to teleport messaging comes from the fields of heterogeneous modeling, program slicing, and domain-specific languages.

As part of the Ptolemy project [EJL$^+$03], Lee et al. have developed hybrid models that incorporate dynamic dataflow (in which the I/O rates of actors are fully dynamic). Boolean dataflow [HL97] is a compromise between these two extremes; it computes a parameterized schedule of the graph at compile time, and substitutes runtime conditions to decide which paths are taken. The performance is nearly that of synchronous dataflow while keeping some flexibility of dynamic dataflow.

Teleport messaging shares the motivation of boolean dataflow, but is different in its approach. We believe that control messages represent a distinct and well-behaved class of dynamic communication in which a parameter is “pushed” into the receiving actor in an asynchronous way. Because the message handlers do not access the I/O channels of the receiving actor, their irregular invocations do not interfere with a given static schedule. Instead, the schedule is constrained only by the latency of control messages; if a message does not show up in the allotted window, then the receiving actor can go ahead with its high-bandwidth schedule. This is the distinction in the computational model. In addition, the static/dynamic integration offered by our system is integrated with language features that support the model.

Program slicing identifies the set of statements in a program that a given statement might depend on. There is a rich history of work in program slicing; see Tip [Tip95] for a comprehensive review. Many program slicing techniques rely on the Program Dependence Graph as described by Horwitz et al. [HRB88]. Program slicing has been applied for debugging, testing, and program analysis. In many respects, SDEP analysis can be thought of as a slicing technique for synchronous dataflow graphs. Because the input domain is restricted (in particular, because of the absence of control flow and recursion), the SDEP calculation can make stronger guarantees than slicing analyses for general procedural languages; SDEP is decidable, exact, and admits a compact representation in terms of the steady state schedule.

Pugh and Rosser present an iteration-based slicing algorithm [PR97] to identify the dynamic instances of statements (in terms of their loop iteration) that effect a given value. This bears some similarity to stream dependence analysis, as $\text{SDEP}_{A \leftarrow B}(n)$ represents the last iteration of actor A that affected the nth iteration of actor B. However, [PR97] focuses on the problem of computing the transitive closure of dependences in loops, in which some iterations do not depend on others. We are not interested in this question, as we assume that all actor invocations depend on their previous invocations; SDEP addresses the question of finding only the most recent invocation that is relevant. Moreover, our motivation differs from the slicing community, as we apply SDEP to enrich the semantics of language features. To the best of our knowledge, slicing has not been applied in this way before.

3.6 Future Work

There are some limitations in the current study that are fertile grounds for future research. First, our formulation of SDEP requires a directed path in the stream graph between the actors in question. We are generalizing SDEP to actors that run in parallel by leveraging their data dependences with common predecessors (upstream) or successors (downstream). Second, in the current model only actors can send and receive messages. We are extending this into a hierarchical model where stream containers (such as pipelines) can also receive events and dispatch them precisely to other streams. This capability is critical for enabling dynamic modifications to the stream graph, as messages will indicate to stream containers that they should re-initialize their part of the stream graph. Finally, our approach relies on the static communication rates present in synchronous dataflow. It is interesting to consider teleport messaging in a more dynamic context; for example, downstream non-negative latency messages could be supported by embedding messages in data items, while other messages might require speculative delivery or modified timing contracts.

Our basic approach to all of the above questions is to reframe SDEP in terms of a more general and intuitive concept called a *canonical schedule*. The abstraction presented to the programmer is that filters do *not* execute in parallel, but rather in a predefined order called the canonical schedule. A pull schedule will serve as the basis of the canonical schedule, though the canonical schedule will also impose a strict ordering on parallel branches of a splitjoin (e.g., always executing from left to right). Given this abstraction, teleport messages with latency zero can be considered to be delivered immediately to the receiver. Messages sent on iteration n with latency k are simply equivalent to messages sent on iteration $n + k$ with latency 0. This model becomes powerful when stream containers are also incorporated in the canonical schedule; for example, a pipeline could be considered to execute whenever one of its children executes, or, for “atomic” pipelines, when all of its children have executed. This provides a precise timeframe for delivering re-initialization events, as well as for flexible forwarding and delegation of messages throughout multiple layers of the stream hierarchy. Of course, just like teleport messaging, the key benefit of the approach is that the canonical schedule is not actually implemented at runtime; rather, the compiler analyzes the dependences inherent in the messages and orchestrates a parallel execution that is indistinguishable from the canonical schedule with respect to message delivery.

Another interesting direction is to leverage the timing properties of teleport messaging to exchange information other than control messages. For example, while StreamIt currently prohibits filters from writing to a global address space, this decision was motivated by the desire to eliminate problematic dependences and aliasing relationships between parallel components. If all reads and writes to shared memory were made via teleport messages to named scalar variables, the compiler could again understand the exact dependences and orchestrate a parallel execution that respects them. Interestingly, such an execution would remain deterministic even though parallel components are writing to shared memory; this determinism comes because the steady-state dataflow provides a canonical ordering for their memory accesses. The compiler could convert shared-memory locations to queues, thereby decoupling the progress of communicating filters. In addition to this application, Matthew Drake proposes a compelling scenario in which messages are used to simultaneously switch the routing behavior at opposite ends of a splitjoin while maintaining analyzability by the compiler [Dra06].

3.7 Chapter Summary

This chapter makes two contributions. First, it introduces teleport messaging: a powerful language construct enabling precise message delivery between nodes of a distributed stream program. In comparison with other methods to implement messaging functionality in a synchronous dataflow model, teleport messaging is arguably more readable, more robust, and easier to maintain. In addition, our implementation of teleport messaging in the StreamIt compiler results in a 49% performance improvement for a frequency hopping radio running on a cluster of workstations. Like several other declarative language constructs, teleport messaging improves performance by exposing the true dependences to the compiler and allowing it to optimize the communication.

Second, this chapter formulates SDEP, a natural and useful dependence representation for the streaming domain. While we apply SDEP to a new language construct, we envision other applications as well. For example, SDEP could be used in a debugger to identify which iterations of one actor are affecting a given iteration of another. In a software-based speculation system [Fra03a], SDEP could be applied to trace the effects of a failed prediction and to roll back the appropriate actor executions. Analogous to representations such as dependence levels [AK82], direction vectors [Wol82], and dependence polyhedra [IT88] for scientific programs, SDEP provides dependence information that could be used to test or verify program transformations. Also, SDEP offers a new method for measuring latency in a stream graph.

Our work can be viewed as integrating dynamic behavior into a static dataflow language. Our insight is that there is a class of control messages that only adjust parameters in the target actor; they do not otherwise affect the input or output channels upon delivery. This model enables a hybrid scheduling scheme in which the steady-state dataflow is exactly orchestrated at compile time, but there are windows in which a message could adjust an internal field of an actor between its execution steps. We consider this to be a promising avenue for creating a unified development environment that captures all aspects of stream application development without sacrificing either performance or programmability.

Chapter 4

Optimizing Stream Programs

This chapter validates the premise that stream programming enables new and powerful optimizations that are outside the reach of a traditional compiler. To do so, we summarize three optimization projects conducted in collaboration with many others in the StreamIt group. Unlike the other chapters in this thesis, the primary description of each of these projects appears in a different thesis [Lam03, Agr04, Ser05] or paper [GTA06].

The key results are as follows:

1. **Parallelization.** We demonstrate an end-to-end stream compiler that attains robust multicore performance in the face of varying application characteristics. As benchmarks exhibit different amounts of task, data, and pipeline parallelism, we exploit all types of parallelism in a unified manner in order to achieve this generality. Our compiler, which maps from the StreamIt language to the 16-core Raw architecture, attains an 11x mean speedup and an 18x maximum speedup over a single-core baseline.

2. **Optimizing Linear Computations**. We demonstrate that several algorithmic transformations, traditionally hand-tuned by DSP experts, can be completely automated by the compiler. We focus on linear filters, where each output is an affine combination of the inputs. We present several optimizations of linear filters, including algebraic simplification of adjacent filters and automatic translation to the frequency domain. These transformations offer an average speedup of 5.5x and a maximum speedup of 8.0x over unoptimized StreamIt on a Pentium 4.

3. **Cache Optimizations**. We formulate a set of cache aware optimizations that automatically improve instruction and data locality. We highlight two techniques: 1) cache aware fusion, which combines adjacent filters while respecting instruction cache constraints, and 2) cache aware scaling, which improves instruction locality while respecting data cache constraints. Our implementation of cache aware optimizations in the StreamIt compiler yields a 3.49x average speedup and an 88x maximum speedup over unoptimized StreamIt on a StrongARM 1110 processor.

These projects are described in more detail in each of the following sections.

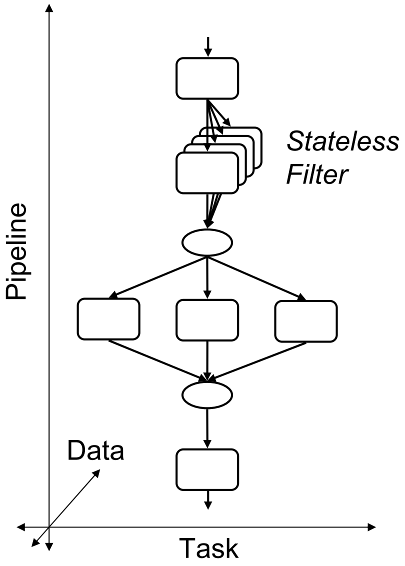


Figure 4-1: Types of parallelism in stream programs. Task parallelism exists between filters in a common splitjoin; pipeline parallelism exists between filters in a producer/consumer relationship; and data parallelism exists between separate instances of a stateless filter.

4.1 Parallelization

Despite the abundance of parallelism in stream programs, it is nonetheless a challenging problem to obtain an efficient mapping to a multicore architecture. Often the gains from parallel execution can be overshadowed by the costs of communication and synchronization. In addition, not all parallelism has equal benefits, as there is sometimes a critical path that can only be reduced by running certain actors in parallel. Due to these concerns, it is critical to leverage the right combination of task, data, and pipeline parallelism while avoiding the hazards associated with each.

Task parallelism refers to pairs of actors that are on different parallel branches of the original stream graph, as written by the programmer. That is, the output of each actor never reaches the input of the other (see Figure 4-1). In stream programs, task parallelism reflects logical parallelism in the underlying algorithm. It is easy to exploit by mapping each task to an independent processor and splitting or joining the data stream at the endpoints. The hazards associated with task parallelism are the communication and synchronization associated with the splits and joins. Also, as the granularity of task parallelism depends on the application (and the programmer), it is not sufficient as the only source of parallelism.

Data parallelism refers to any actor that has no dependences between one execution and the next. Such “stateless” actors[1] offer unlimited data parallelism, as different instances of the actor can be spread across any number of computation units. However, while data parallelism is well-suited to vector machines, on coarse-grained multicore architectures it can introduce excessive communication overhead. Previous data-parallel streaming architectures have focused on designing a special memory hierarchy to support this communication [KRD^{+}03]. However, data parallelism has the hazard of increasing buffering and latency, and the limitation of being unable to parallelize actors with state.

[1] A stateless actor may still have read-only state.

Pipeline parallelism applies to chains of producers and consumers that are directly connected in the stream graph. In our previous work [GTK$^+$02], we exploited pipeline parallelism by mapping clusters of producers and consumers to different cores and using an on-chip network for direct communication between actors. Compared to data parallelism, this approach offers reduced latency, reduced buffering, and good locality. It does not introduce any extraneous communication, and it provides the ability to execute any pair of stateful actors in parallel. However, this form of pipelining introduces extra synchronization, as producers and consumers must stay tightly coupled in their execution. In addition, effective load balancing is critical, as the throughput of the stream graph is equal to the minimum throughput across all of the processors.

In this section, we describe a robust compiler system that leverages the right combination of task, data, and pipeline parallelism to achieve good multicore performance across a wide range of input programs. Because no single type of parallelism is a perfect fit for all situations, a unified approach is needed to obtain consistent results. Using StreamIt as our input and targeting the 16-core Raw architecture, our compiler demonstrates a mean speedup of 11.2x over a single-core baseline. This also represents a 1.84x improvement over our original approach [GTK$^+$02].

Parallelization Algorithm

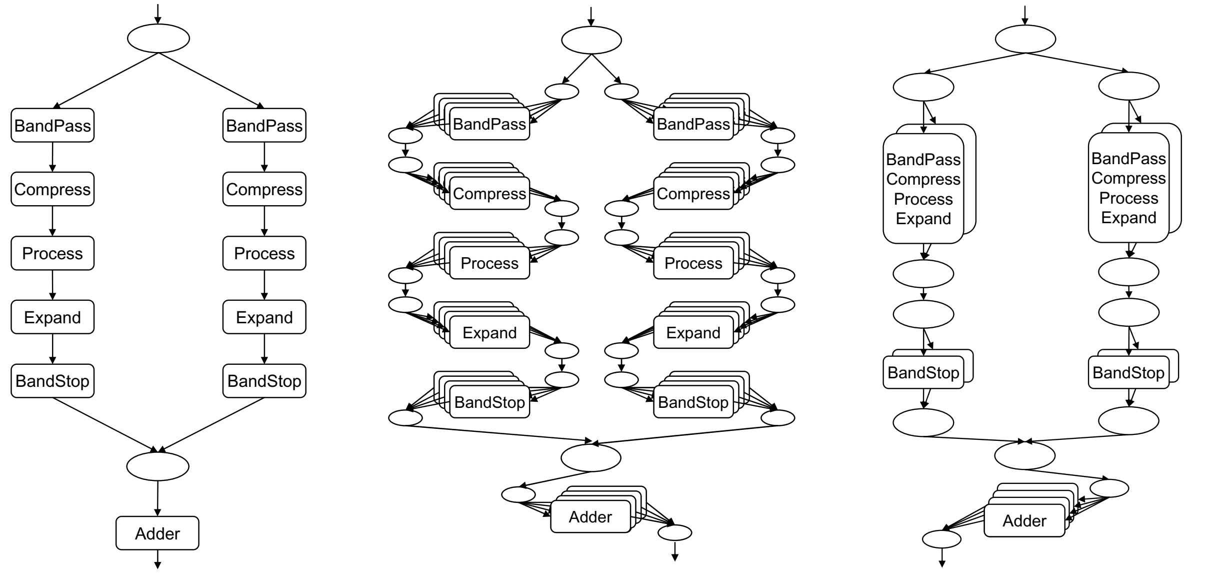
We illustrate our technique by way of an example: we discuss how to map a simplified version of our FilterBank benchmark (see Figure 4-2a) to a four-core machine. The complete details of our algorithm are available elsewhere [GTA06].

Previous Practice: Fine-Grained Data Parallelism Perhaps the most common approach to parallelization is to identify loops that can be run in a data-parallel (DOALL) style. Such loops can be annotated by the programmer using OpenMP; they are also the most common parallelization target of production compilers. For example, the Intel C Compiler includes an optional flag to detect and parallelize data-parallel loops. In the case of FilterBank, this may seem like a promising approach, as all the filters are stateless and the implicit loops surrounding them can be run in a data-parallel manner. Figure 4-2b illustrates such a mapping.

Unfortunately, on a coarse-grained multicore architecture, it is hardly profitable to parallelize each individual filter due to the communication and synchronization overheads incurred. When we target the 16-core Raw architecture, this approach offers only a 1.4x mean speedup over a single core. This represents an upper bound on the speedups attainable using standard techniques. In practice, for reasons explained in Section 2.2, a production C compiler would achieve even smaller speedups due to the inherent difficulties of proving that filters are data-parallel.

First Innovation: Coarse-Grained Data Parallelism The overheads of fine-grained data parallelism can be drastically reduced by performing two novel transformations. First, the granularity of the stream graph is coarsened via filter *fusion*, a transformation in which two neighboring filters are statically scheduled and inlined into a single filter [GTK$^+$02, STRA05]. We fuse neighboring stateless filters as much as possible so long as the resulting filter remains stateless, ensuring that it is still amenable to data parallelism.

Second, we data-parallelize the coarsened filters, but only by the amount necessary to complement existing task parallelism in the stream graph. That is, for filters that are already embedded



(a) Original (b) Fine-grained data parallelism (c) Coarse-grained data parallelism

Figure 4-2: Mapping a simplified version of the FilterBank benchmark for execution on four cores. The original stream graph is shown in (a), while a conventional mapping is shown in (b). Our technique coarsens the graph and introduces the minimal parallelism needed, as shown in (c).

in a splitjoin, we parallelize each filter so that the total splitjoin width covers all of the cores, rather than data-parallelizing each branch of the splitjoin to cover all of the cores. By reducing the width of the scatter and gather stages, we reduce the communication and synchronization overhead incurred by data parallelism.

Figure 4-2c shows an example of our transformations on the FilterBank benchmark. The coarsening stage fuses all of the pipelines together with the exception of the BandStop filter, which is not fused because it performs peeking on its input channel. Communication with peeking represents a case where some data items are reused between successive firings of a filter, which would translate to internal state if the buffer were to be inlined into a fused filter. Following coarsening, the parallelization stage replicates the Adder filter across all four of the target cores. However, the other filters are split only two ways, due to the presence of task parallelism between alternate branches of the splitjoin. Applying this strategy across our benchmark suite offers a speedup of 9.9x relative to a single core.

These transformations are out-of-reach of traditional compilers. In an imperative language, the analog of graph coarsening is to selectively fuse loops so long as no new loop-carried dependences are introduced. The analog of task-conscious data parallelism is to analyze the entire program for other threads that might be running concurrently, and to introduce only as much parallelism as is needed to complement the other threads. We rely on the properties of the stream programming model to make these transformations tractable.

Second Innovation: Coarse-Grained Software Pipelining While coarse-grained data parallelism is effective for parallelizing stateless computations, it does nothing to help with computations that retain state, either within filters or within feedbackloops. For example, the Vocoder

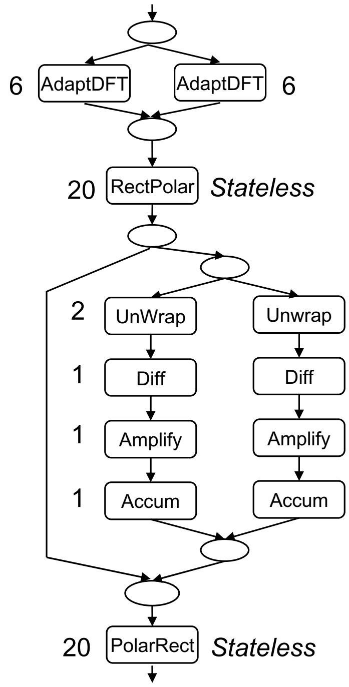


Figure 4-3: Simplified subset of the Vocoder benchmark. Nodes are annotated with the amount of work that they perform per steady state.

benchmark (simplified subset shown in Figure 4-3) contains a significant fraction of stateful filters. While two of the filters can be data-parallelized, there remain large gaps in the execution schedule (see Figure 4-4).

To run stateful computations in parallel with each other, we exploit pipeline parallelism. We take the concept of software pipelining, well-understood in the context of instruction scheduling, and apply it in the context of an entire stream graph. As illustrated in Figure 4-5, this technique involves unrolling the execution of the graph into two stages. In the first stage, a prologue schedule establishes buffering in the data channels. Then, in the steady state, the filters are decoupled and can execute in any order, writing intermediate results to the buffers. Compared to exploiting only coarse-grained data parallelism, this technique offers large gains for our stateful benchmarks (1.7x for Vocoder, 1.9x for Radar). Together with coarse-grained data parallelism, it offers an 11.2x speedup over a single core across our benchmark suite.

Coarse-grained software pipelining is also beyond the reach of traditional compilers. Rather than pipelining individual instructions, it represents the pipelining of entire procedures. This involves reordering large pieces of the program. The stream programming model makes such a transformation feasible by exposing the stable flows of data between long-running actors.

Experimental Evaluation

We target the Raw microprocessor [TKM$^+$02, WTS$^+$97], a tiled array of 16 cores with a programmable mesh interconnect. Though Raw has been implemented in silicon, we generate results with the btl simulator, augmented with 16 streaming DRAM controllers (providing enough band-

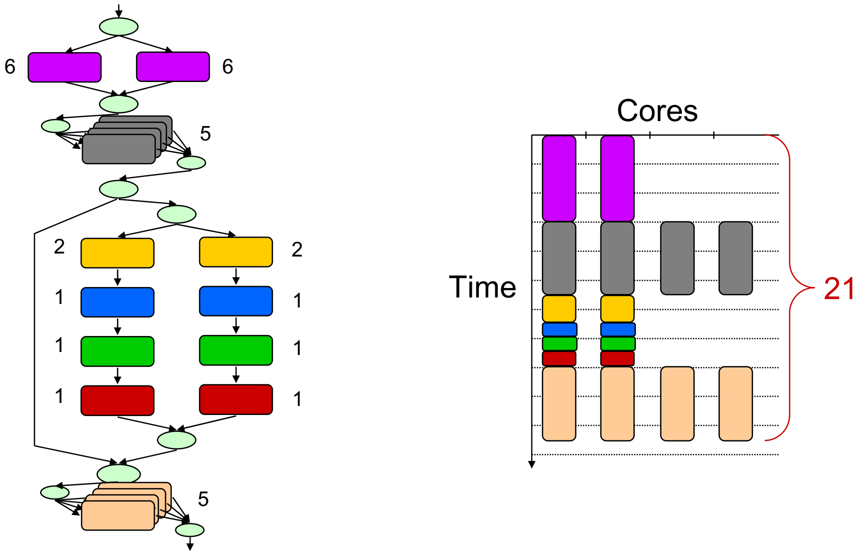


Figure 4-4: Simplified vocoder mapped with data parallelism. In the stream graph (left), stateless nodes are replicated but stateful nodes remain untouched. An execution trace (right) requires 21 units per steady state.

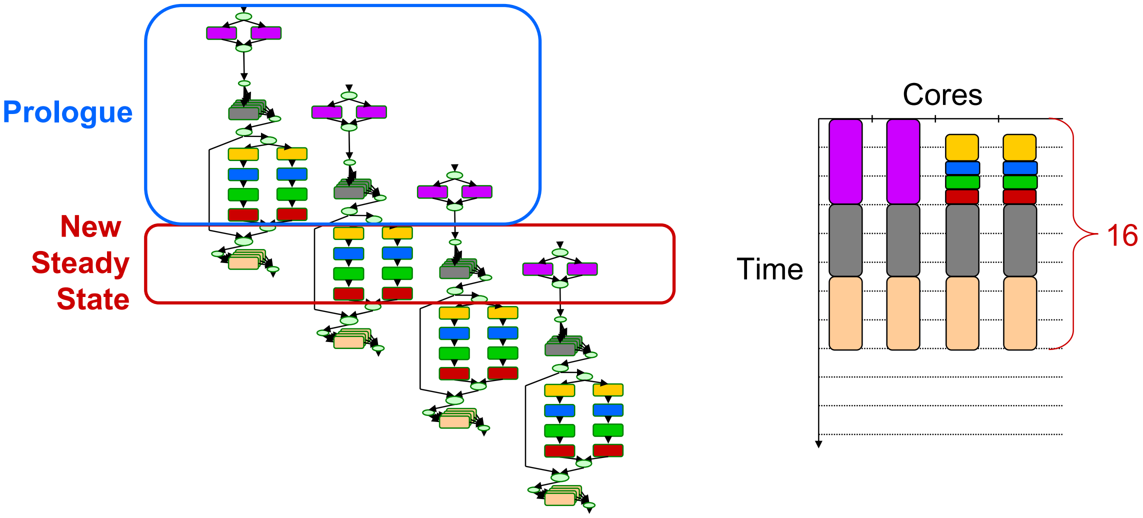


Figure 4-5: Simplified vocoder mapped with coarse-grained software pipelining. By unrolling multiple executions of the stream graph (left), stateful nodes can run in parallel with other nodes during the steady state. An execution trace (right) requires 16 units per steady state, an improvement over plain data parallelism.

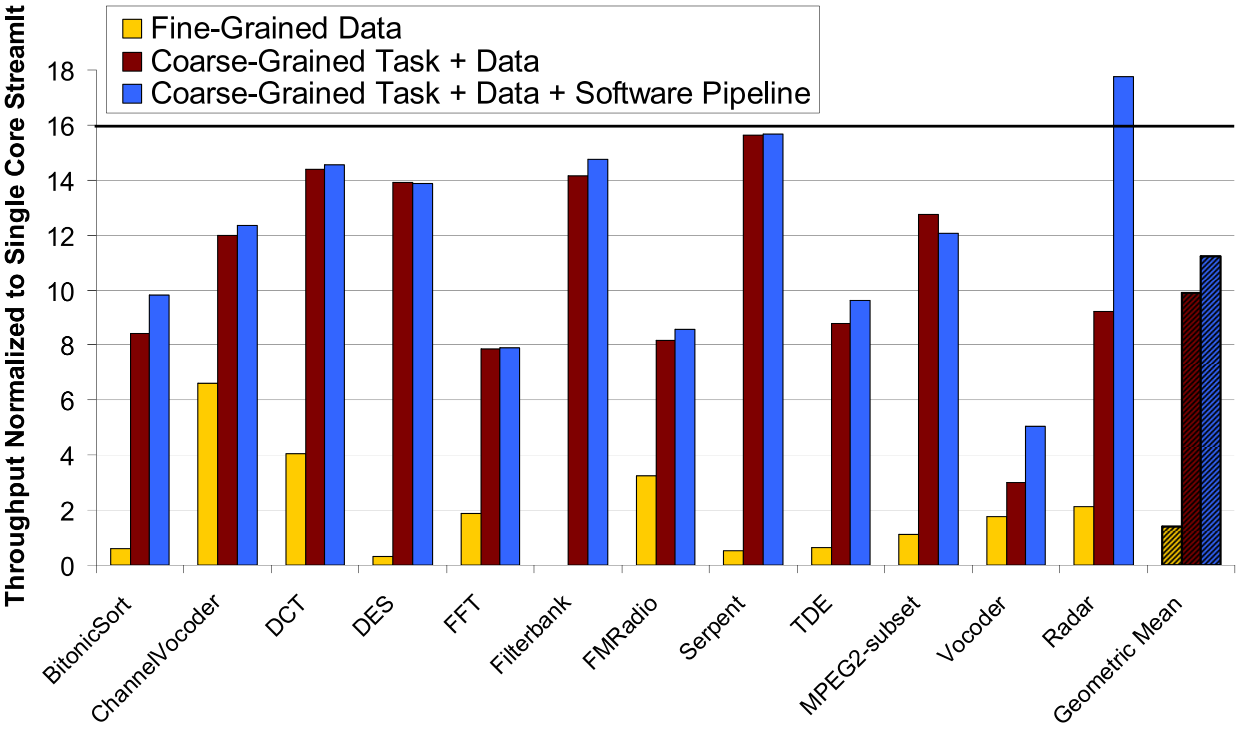


Figure 4-6: Parallelization results on the 16-core Raw processor[1].

width to saturate both directions of a Raw port). In this configuration, one can obtain higher throughput in streaming data from the off-chip memory than from a core's local data cache. Thus, our implementation elects to buffer all streaming data off-chip. However, when targeting an architecture with more modest off-chip memory bandwidth, the stream buffers could reside completely in on-chip memory.

A summary of our results appears in Figure 4-6. We show the speedup offered by the three techniques mentioned: fine-grained data parallelism, the previous standard; coarse-grained data parallelism, which also leverages the existing task parallelism in the stream graph; and coarse-grained software pipelining, which runs as a post-pass to coarse-grained data parallelism. Our baseline is StreamIt executing on a single core, which (in the case of Raw) has been shown to outperform hand-written C implementations on a single core [TLM+04]. While coarse-grained data parallelism performs well (attaining a mean speedup of 9.9x), the most robust performance comes by adding coarse-grained software pipelining (which attains a mean speedup of 11.2x). As expected, software pipelining mostly benefits the stateful benchmarks, Vocoder and Radar. There is a super-linear speedup in Radar because reordering operations are moved from the compute core to the network.

[1] The benchmarks used here were sometimes parameterized differently than the ones described in Chapter 2. Details on the benchmark configurations are available elsewhere [GTA06].

4.2 Optimizing Linear Computations

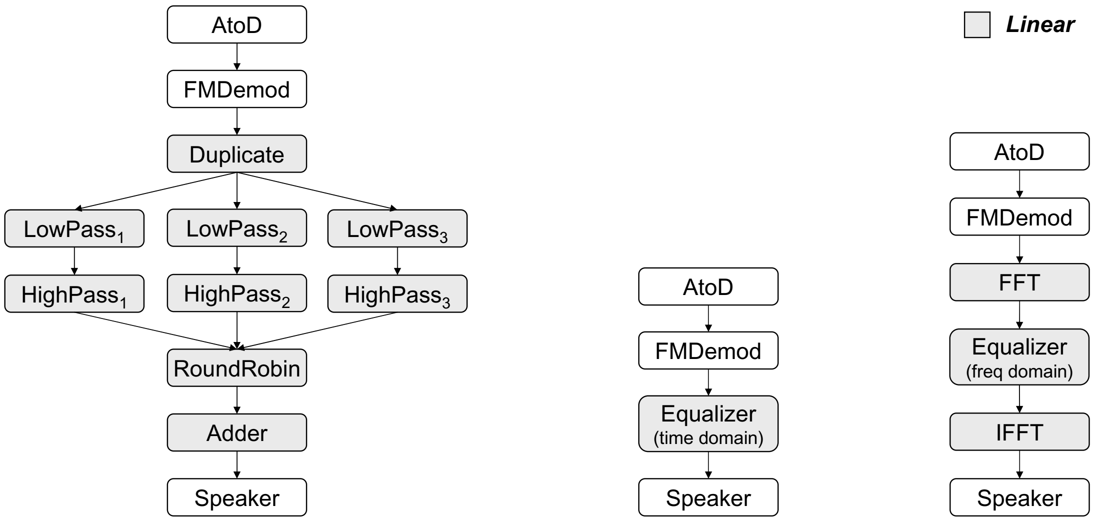
The design flow for digital signal processing applications typically contains three steps. First, application designers specify a block diagram of the computation, drawing on rich software libraries to prototype its behavior in an environment such as MATLAB. Once the the functionality has been fixed, the second step is performed by digital signal processing (DSP) experts, who inspect the global structure of application and perform many domain-specific optimizations to reduce the overall processing requirements while preserving the basic input/output relationship. Finally, once the mathematical algorithms have been determined, a software engineer implements those algorithms in a low-level language such as C to obtain the final product.

In order to reduce the cost of this development process, a long-term goal of the computer science community has been to generate efficient and deployable code from a high-level, functional specification of the program. In order to achieve this goal, the expertise of DSP experts must be encapsulated into the tool. While library generators such as Spiral [PMJ$^+$05], FFTW [FJ05], and ATLAS [WPD01, DDE$^+$05] can automatically derive and optimize specific classes of DSP kernels, programmers must integrate these libraries into their development process rather than having the compiler automatically recognize and transform the original code. Our goal is to invent and adapt domain-specific optimizations in the context of the StreamIt language, so as to provide a unified development environment that can express the full functionality of the application while automatically applying deep optimizations to the specific code sections where they apply.

Our focus in the current work is the optimization of *linear* computations, which are the most common target of DSP experts. Linear filters are those in which each output is an affine combination of the inputs. Examples include finite impulse response (FIR) filters, compressors, expanders and signal processing transforms such as the discrete Fourier transform (DFT) and discrete cosine transformation (DCT). We also describe the optimization of linear statespace filters, a generalization of linear filters that maintain internal states. In a linear statespace filter, each each output is an affine combination of the states and the inputs, and each state is also updated in an affine fashion. An infinite impulse response (IIR) filter is an example of a linear statespace filter.

Figure 4-7 illustrates an example of linear optimizations as applied to our software radio benchmark. The radio contains an equalizer, which was specified by the designer in a simple but inefficient manner. Each frequency band is processed in a separate branch of a splitjoin, and each branch contains a successive high-pass and low-pass filter to accomplish a band-pass functionality. While this representation of the algorithm allows it to be easily understood and maintained, it performs many redundant computations. In practice, because all of the components of the equalizer are linear, they can be collapsed into a single filter that performs far fewer computations. Furthermore, as that filter is performing a sliding window computation, it can be converted into the frequency domain to reduce the asymptotic processing requirements from $O(n^2)$ to $O(n \log n)$. Both of these transformations require deep inter-procedural analysis and are far beyond the reach of traditional compilers. However, using a stream programming model, we can robustly automate both steps of the optimization process.

In the rest of this section, we provide an overview of our linear optimization techniques. We describe how to extract a linear representation from the code in a StreamIt filter, how to algebraically simplify adjacent linear filters, and how to translate linear filters into the frequency domain. We also describe optimizations for linear statespace filters, including removal of redundant states and reduction of the number of parameters. We give a procedure for determining which optimizations



(a) Software FM radio with Equalizer (b) After linear combination (c) After translation to the frequency domain

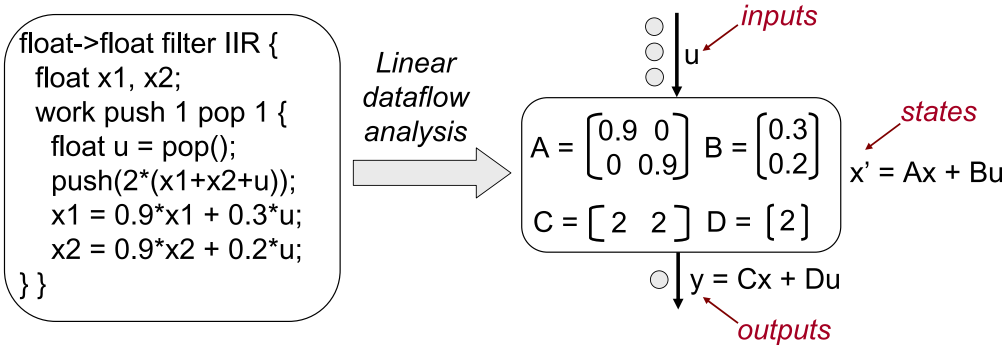
Figure 4-7: Example optimization of linear filters. Our software FM radio benchmark contains an equalizer in which all filters are linear. These filters can be algebraically simplified into a single filter and then translated into the frequency domain.

to apply to a given program, and we evaluate the optimizations in the StreamIt compiler. The average speedup obtained is 4.5x, with a maximum of 8.0x.

Extracting a Linear Representation

Rather than directly manipulating the code inside a filter's work function, our linear optimizations rely on an abstract representation in which linear filters are represented by a set of matrices. Figure 4-8 gives an example of this representation for an IIR filter. The StreamIt compiler automatically extracts this representation using a symbolic execution of the filter's work function. The basic idea is to execute a complete invocation of the function just like a normal interpreter, except that instead of assigning values to the states and input items, these quantities are left as free variables and tracked throughout the execution. If the interpreter encounters any branches or conditionals that depend on free variables, then the analysis is aborted and the node is deemed non-linear. Otherwise, when execution completes, a symbolic expression has been established for every state variable and every value pushed to the output tape. If all of these expressions are an affine function of the free variables, then linear analysis has succeeded and the linear representation is built. A more precise description of this analysis, including support for innocuous branches that do not affect the linear representation, is described elsewhere [LTA03].

Of course, it would also be possible for programmers to specify the linear representation directly rather than relying on the compiler to extract it from the code. If programmers prefer this approach, then they could develop a generic linear filter in StreamIt and call it as a library. However, we believe that it is valuable to support automatic recognition of optimization targets, as otherwise the programmer needs to be knowledgeable of every potential optimization and annotate the code accordingly.



```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```

Linear dataflow analysis
inputs
u
states
$A = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix}$
$B = \begin{bmatrix} 0.3 \\ 0.2 \end{bmatrix}$
$C = \begin{bmatrix} 2 & 2 \end{bmatrix}$
$D = \begin{bmatrix} 2 \end{bmatrix}$
x' = Ax + Bu
y = Cx + Du
outputs

Figure 4-8: Extracting a linear representation from the code inside a filter's work function.

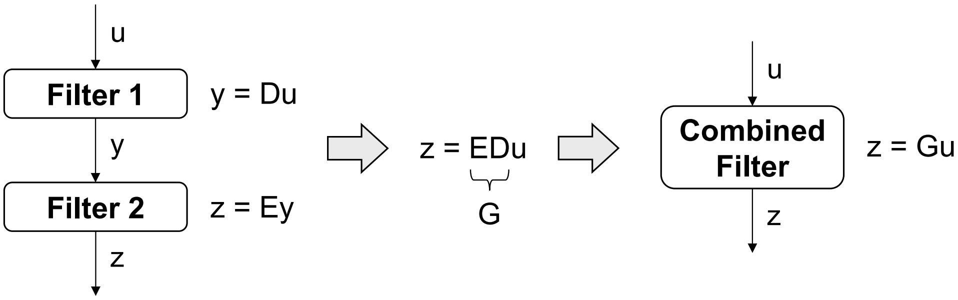


Figure 4-9: Algebraic simplification of adjacent linear filters. For simplicity, we omit filter states from this diagram.

Algebraic Simplification of Adjacent Linear Filters

If neighboring filters in the stream graph both perform a linear computation, then that section of the stream graph can be collapsed into a single linear filter. The most simple case of this transformation is illustrated in Figure 4-9, where two stateless filters are communicating in a pipeline. Given the computation matrix for each filter, the output of the entire pipeline can be represented as a matrix product. Because each of the matrices is known at compile time, the matrix product can also be evaluated at compile time. This offers the potential for large performance gains. For example, if both matrices are square (representing filters that read the same number of items as they write) and there is no peeking involved, then the output matrix will be the same size as each of the input matrices, reducing the computation by a factor of two. Larger gains are possible if the communication rate between the filters is larger than that of the end-to-end pipeline. Conversely, if the communication rate between the filters is lower than the overall pipeline, it is possible for this transformation to to increase the computation requirements; as described later, this hazard is avoided by our automatic selection algorithm. In our experimental evaluation, combining filers wherever possible (even when detrimental) leads to a 2.1x average performance improvement, with a maximum improvement of 5.0x.

We have extended the simple idea of algebraic simplification to handle the general case, hiding many complexities from the user [ATA05]. To perform the matrix multiplication in Figure 4-9,

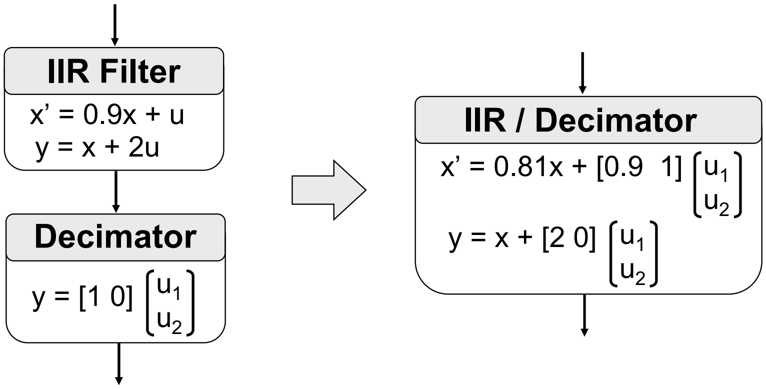

IIR Filter
x' = 0.9x + u
y = x + 2u
Decimator
y = [1 0] $\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$
IIR / Decimator
x' = 0.81x + [0.9 1] $\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$
y = x + [2 0] $\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$

Figure 4-10: Example simplification of an IIR filter and a decimator. The original pair of filters requires 6 FLOPs per output, while the combined version requires only 4 FLOPs per output.

the output rate of the first filter must match the input rate of the second filter. In cases where this is not true in the program, the analysis expands each linear representation to encompass multiple executions of the original filter. In addition to collapsing pipelines, we have also developed complete combination rules to handle the other StreamIt language constructs: splitjoins and feedbackloops. Splitjoins introduce complexity due to the reordering in the splitters and joiners, as well as implicit buffering that may be involved due to mis-matched I/O rates along alternate branches. Feedbackloops introduce complexity because of the initial items enqueued on the backward path of the feedback loop; in addition, the periodicity of the entire feedbackloop may be coarser than the periodicity of its components, requiring further expansion and analysis by the compiler. The presence of sliding window operations (or peeking) also adds complexity to all of the combination rules; in our general formulation, the peeked data items are converted into states in the filter.

By automating the combination of linear filters, we allow the programmer to maintain a natural expression of the algorithm. Figure 4-10 illustrates an example combination of an IIR filter with a decimator, reducing the total number of operations by 25%. This optimization opportunity is not obvious to non-experts due to the state retained by the IIR filter. Also, even when the programmer understands that linear combination is possible, it may be intractable to manage of all of the details and to maintain the code following the transformation. This effect is especially important in the context of software libraries, where the final application may contain filters that were authored by many different developers. The compiler can perform this analysis across module boundaries, synthesizing an efficient implementation while preserving a modular development style.

Optimization of a Single Linear Filter

In addition to optimizing groups of linear filters, it is possible to improve the execution of a single linear filter at a time. Stateless filters can be mapped into the frequency domain, while stateful filters are subject to state removal and parameter reduction. These transformations are generally applied after algebraic simplification.

Mapping into the Frequency Domain Filters that perform a sliding window computation, such as FIR filters, are equivalent to a convolution of the filter coefficients with the input tape. This

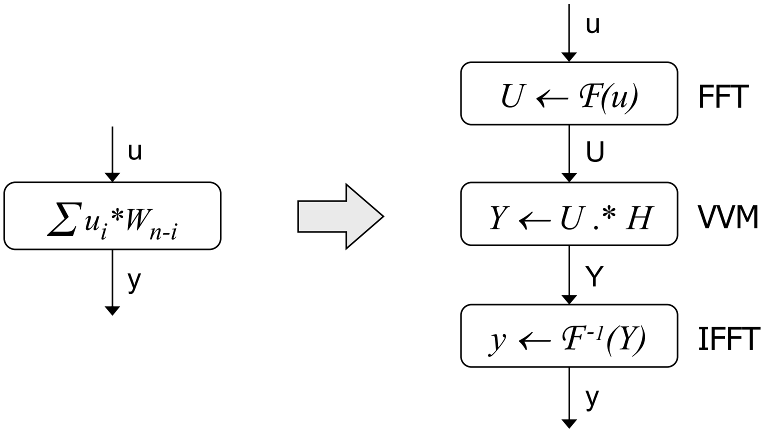

u
$\sum u_i * W_{n-i}$
y
u
$U \leftarrow \mathcal{F}(u)$
FFT
U
$Y \leftarrow U .* H$
VVM
Y
$y \leftarrow \mathcal{F}^{-1}(Y)$
IFFT
y

Figure 4-11: Mapping linear filters into the frequency domain.

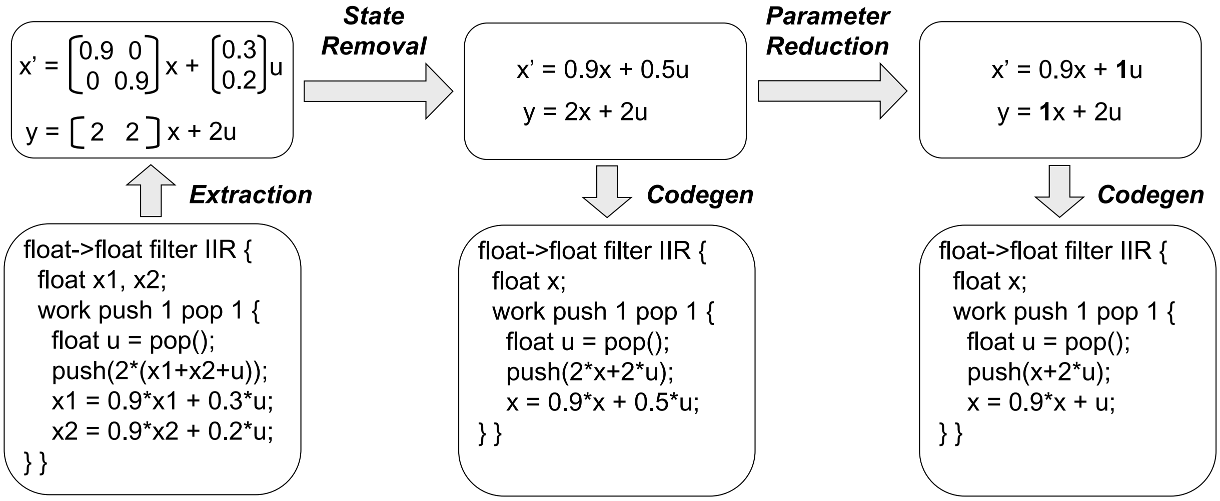
means that they are amenable to a classic transformation in single processing, whereby the computation is mapped from the time domain into the frequency domain. As illustrated in Figure 4-11, this consists of wrapping the filter in an FFT and inverse FFT, and changing the convolution into a vector-vector multiply. Asymptotically, this reduces the computation requirements from $O(n^2)$ to $O(n \log n)$, where n is the size of FFT (which can be set by the compiler). In our experiments, translating each filter into the frequency domain (even when detrimental) leads to an average speedup of 3.8x and a maximum speedup of 8.0x.

While this transformation is well-understood and can also be done by hand, there are benefits to automating it in the compiler. The size of the FFT can be automatically selected and complex startup conditions can be handled automatically. Also, there are cases where it is not profitable to translate to the frequency domain (for example, if the peek window is too small, or if the filter decimates items in addition to peeking), or where conversion is profitable only following linear combination. By coupling the translation algorithm with our optimization selection algorithm (described later), the programmer does not need to worry about when to apply the transformation.

Removing States Linear statespace filters maintain and update a set of internal states on each time step. However, especially following combination with adjacent nodes, it is possible that some of these states could be redundant; that is, their values could be fully derived from other states in the filter. It is beneficial to remove any redundant states, both for memory savings and to eliminate redundant computations that update the states.

We have adapted a simple algorithm that guarantees to identify and remove all of the redundant states in a filter [May73]. While this algorithm was previously known by the signal processing community, to our knowledge this is its first application in an optimizing compiler. The algorithm works by constructing augmented matrices from the filter's representation (Figure 4-8), and by reducing these matrices to a special row-echelon form.

An example of state removal appears in Figure 4-12. The analysis detects that the two states `x1` and `x2` are always scaled proportionately, so they can be combined into a single state `x`. This reduced the computational requirements of the filter from 9 FLOPs per execution to 5 FLOPs per execution.



$$x' = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix} x + \begin{bmatrix} 0.3 \\ 0.2 \end{bmatrix} u$$

$$y = \begin{bmatrix} 2 & 2 \end{bmatrix} x + 2u$$

```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```

$$x' = 0.9x + 0.5u$$

$$y = 2x + 2u$$

```
float->float filter IIR {
  float x;
  work push 1 pop 1 {
    float u = pop();
    push(2*x+2*u);
    x = 0.9*x + 0.5*u;
} }
```

$$x' = 0.9x + \mathbf{1}u$$

$$y = \mathbf{1}x + 2u$$

```
float->float filter IIR {
  float x;
  work push 1 pop 1 {
    float u = pop();
    push(x+2*u);
    x = 0.9*x + u;
} }
```

(a) Original filter:
9 FLOPs per output

(b) After state removal:
6 FLOPs per output

(c) After parameter reduction:
4 FLOPs per output

Figure 4-12: Example of state removal and parameter reduction.

Reducing the Number of Parameters After removing as many states as possible, additional computations can be eliminated by transforming the filter's linear representation into one with fewer non-zero, non-one entries (termed parameters). Each coefficient that is converted to a zero serves to eliminate a multiplication and addition operation per execution of the filter, while each coefficient that is converted to a one serves to eliminate a multiplication.

We automated parameter reduction by starting with a known signal processing technique [AB71] and reformulating it in the context of StreamIt. As with the state removal algorithm, the number of parameters in a linear statespace filter can be reduced using a systematic sequence of matrix operations. However, compared to state removal, there are looser guarantees on the optimality of the final system [ATA05].

An example of parameter reduction is illustrated in Figure 4-12. Following the transformation, the state variable x assumes a value that is twice as large as the original (at any given point of execution). However, this change does not affect the output of the filter, as the other coefficients are compensated accordingly. The transformation enables two coefficients two change to a value of 1, thereby eliminating two multiplication operations and reducing the total cost to 4 FLOPs per execution.

Optimization Selection

As mentioned previously, many of the described transformations have the potential to decrease the performance of the program. Linear combination can bloat the processing requirements depending on the communication rates of the filters, and translation to the frequency domain can introduce overhead for filters with high pop rates or low peek rates. Instead of applying the transformations blindly, they should be guided by a selection algorithm that matches the behavior of DSP experts.

We have developed a general and robust optimization selection algorithm that considers a large space of candidate transformations. To prevent an exponential explosion of candidate transformations on different parts of the stream graph, the algorithm leverages overlapping sub-problems and uses dynamic programming to arrive at an efficient solution.

The algorithm works by estimating the minimum cost for each structure (filter, pipeline, splitjoin, and feedbackloop) 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. (This algorithm does not consider state removal and parameter reduction, which were invented subsequently.) The cost functions for the collapsed cases are guided by profiler feedback, performed once during the development of the compiler. For the uncollapsed case, the cost is the sum of each child's minimum cost.

A key aspect of the algorithm is that it considers many possible boundaries for the structures in the stream graph. For example, while the programmer might have constructed the graph as a specific hierarchy of pipelines, the compiler flattens the hierarchy into a single pipeline and then considers linear optimizations for each contiguous region within that pipeline. A similar decomposition applies to splitjoins, where any number of adjacent branches and any contiguous sequence of streams in those branches is considered for transformation. In this sense, the algorithm determines not only the best transformations to apply, but also the best way to refactor the stream graph into a form that is amenable to optimization.

An example of optimization selection for the Radar benchmark is shown in Figure 4-13. Radar[2] contains many linear filters. However, performing maximal linear combination results in a 3.2x slowdown, and translating to the frequency domain worsens performance by an additional 12x. The problem with linear combination is 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). The optimization selection algorithm avoids combining BeamForm with its successor, and avoids the costly frequency translation. Applying only selective transformations causes 55% of the FLOPs to be eliminated. However, the final speedup is only 5%, mostly due to unrelated data and code size issues that could be addressed independently (each filter is very coarse-grained).

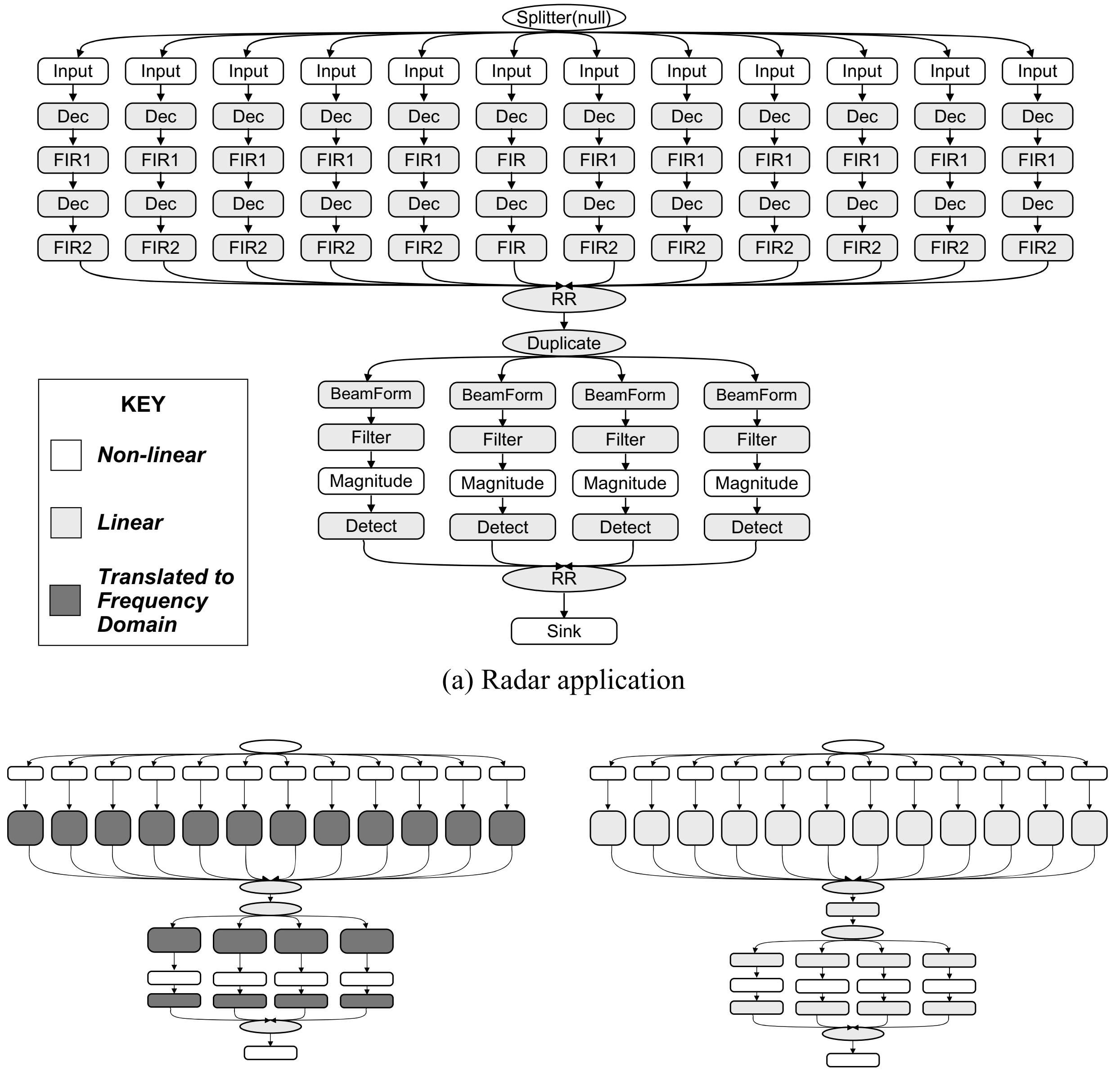
Experimental Evaluation

We have implemented linear optimizations in the StreamIt compiler. Here we present results for stateless linear nodes, though we have also shown that linear statespace analysis offers improved generality [ATA05]. For more detailed results, stream graphs, and source code, please visit http://cag.lcs.mit.edu/linear/ or see the accompanying thesis [Lam03].

We evaluate linear optimizations on a uniprocessor. Our measurement platform is a Dual Intel Pentium 4 Xeon system with 2GB of memory running GNU/Linux. To measure the number of floating point operations, we use an instruction counting DynamoRIO [BDB99] client.

Figure 4-14 indicates the number of floating point operations (FLOPs) removed from the program. The removal of FLOPs represents fundamental computation savings that is independent of the streaming runtime system and other (FLOPs-preserving) optimizations in the compiler. We

[2] This version of the Radar benchmark is different from the one used in the parallelization section. It is rewritten to be extremely coarse-grained, eliminating the internal state and exposing the linear relationships.


Splitter(null)
Input Input Input Input Input Input Input Input Input Input Input Input
Dec Dec Dec Dec Dec Dec Dec Dec Dec Dec Dec Dec
FIR1 FIR1 FIR1 FIR1 FIR1 FIR FIR1 FIR1 FIR1 FIR1 FIR1 FIR1
Dec Dec Dec Dec Dec Dec Dec Dec Dec Dec Dec Dec
FIR2 FIR2 FIR2 FIR2 FIR2 FIR FIR2 FIR2 FIR2 FIR2 FIR2 FIR2
RR
Duplicate
BeamForm BeamForm BeamForm BeamForm
Filter Filter Filter Filter
Magnitude Magnitude Magnitude Magnitude
Detect Detect Detect Detect
RR
Sink
KEY
Non-linear
Linear
Translated to Frequency Domain

(a) Radar application

(b) Radar after maximal linear combination and translation to the frequency domain (33x more FLOPs; 40x slowdown)

(c) Radar after optimization selection (55% fewer FLOPs; 5% speedup)

Figure 4-13: Optimization selection for the Radar benchmark. Performing maximal linear combination and translation to the frequency domain results in a large slowdown. The selection procedure avoids this hazard by blocking the vertical combination of the BeamForm filter and by preventing translation to the frequency domain.

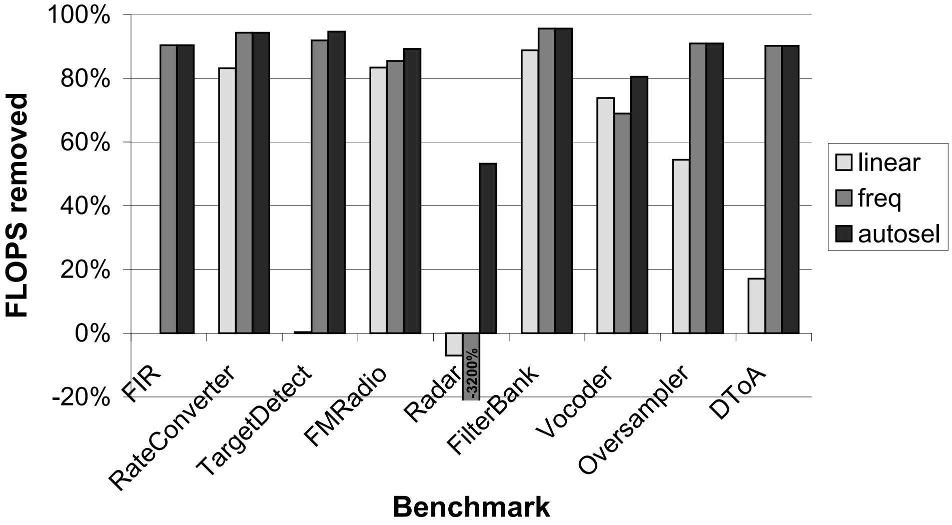


Figure 4-14: Elimination of floating point operations by maximal linear replacement, maximal frequency replacement, and automatic optimization selection[1].

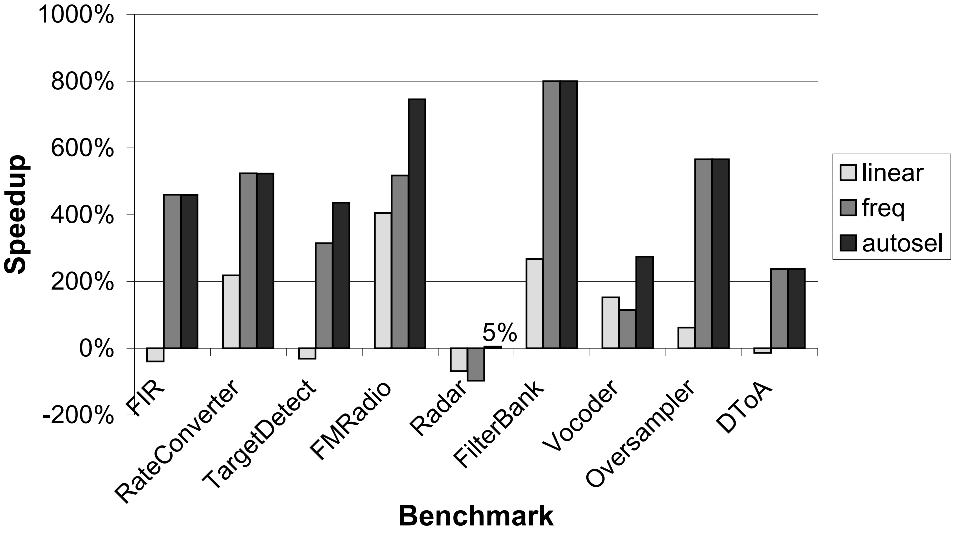


Figure 4-15: Execution speedup for maximal linear replacement, maximal frequency replacement, and automatic optimization selection[1].

evaluate three strategies: maximal combination of linear nodes, maximal translation to the frequency domain (following maximal combination), and automatic optimization selection. The automatic selection routing removes an average of 87% of the FLOPs from our benchmarks, with a maximum of 96% (Vocoder). The automatic selection option eliminates more FLOPS than either of the other options for TargetDetect, FMRadio, Radar, and Vocoder. Automatic selection always performs at least as well as the other two options.

[1] The benchmarks used here were sometimes parameterized differently than the ones described in Chapter 2. Details on the benchmark configurations are available elsewhere [Lam03].

Execution speedups are shown in Figure 4-15. With automatic selection, our benchmarks speed up an average factor of 5.5x and by a factor of 8.0x in the best case (FilterBank). 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.

While these results represent radical improvements relative to most compiler optimizations, we emphasize that the same transformations would likely be done by hand in a production system. Our contribution is to enable a modular programming environment by automatically performing the transformations from a high-level description.

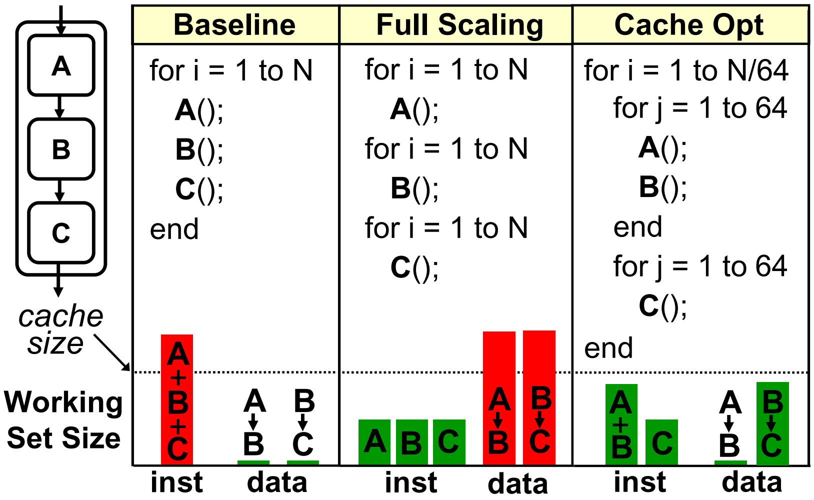
4.3 Cache Optimizations

An important part of achieving high performance is to maximize the utilization of the cache. This is especially important on embedded processors, which often lack an L2 cache. In tandem with this need for high cache utilization, there is also a unique opportunity in the streaming domain to reorder filter executions so as to improve the cache behavior. Memory accesses are extremely regular due to the explicit producer-consumer relationships between filters, allowing the compiler to anticipate and optimize the cache usage.

We have developed a set of cache optimizations that simultaneously consider data and instruction locality while scheduling stream programs. An overview of our optimizations are illustrated in Figure 4-16. In scheduling a pipeline of filters, the executions can be interleaved in any order so long as data is produced before it is consumed. In the baseline configuration, there is a fine-grained interleaving of filters; each filter fires once per execution of the outer loop. While this results in a very small data working set (data is consumed immediately following its production), the instruction working set is large because all filters are accessed frequently. The opposite of this scheduling strategy, termed "full scaling", wraps each filter in its own loop, buffering all of the data before the next filter executes. While this shrinks the instruction working set size (since only one actor is accessed at a time), the data working set could grow very large due to the buffering between actors.

Our optimized scheduling strategy, illustrated on the right of Figure 4-16, can be considered as a tradeoff between these two extremes. First, we employ a heuristic called *cache aware fusion* that fuses executions of the inner loop as much as possible without overflowing the instruction cache. In this case, filters A and B can be fused, but filter C remains separate. Then, we employ a technique called *cache aware scaling* that sets the inner loop bounds as high as possible without overflowing the data cache. In this case, a bound of 64 ensures that the communication between B and C stays within the cache. This technique offers joint improvement of instruction and data locality without risking the penalty of cache overflow.

In the rest of this section, we provide more details on cache aware fusion, cache aware scaling, and present an experimental evaluation. Our full report on this subject contains further details, including optimized buffer management strategies [STRA05, Ser05].



```
for i = 1 to N
  A();
  B();
  C();
end
```

```
for i = 1 to N
  A();
for i = 1 to N
  B();
for i = 1 to N
  C();
```

```
for i = 1 to N/64
  for j = 1 to 64
    A();
    B();
  end
  for j = 1 to 64
    C();
end
```

Figure 4-16: Overview of cache optimizations. While execution scaling can improve instruction locality relative to the baseline, it loses the benefits of filter fusion and has the hazard of overflowing the data cache. Our cache optimizations fuse filters as much as possible without exceeding the instruction cache, then scale as much as possible without exceeding the data cache.

Cache Aware Fusion

As mentioned previously, filter fusion is a transformation in which two filters are tightly scheduled and inlined into the same filter. Fusion offers many benefits, including reduced method call overhead and improved producer-consumer locality. It also allows traditional compiler optimizations to span across filter boundaries; in particular, results that were previously buffered in memory can now be allocated to registers in an optimization known as scalar replacement. For our benchmark suite, fusing all of the filters in the program improves performance by an average of 1.3x on an embedded processor.

However, the hazard of excessive fusion is that the combined instruction and data footprint of the filters will overflow the caches, thereby hampering performance. The scalar replacement optimization also benefits from aggressive loop unrolling, which causes code bloat and increases the risk of cache overflow. To address this hazard, our cache aware fusion algorithm greedily fuses neighboring filters so long as the instruction and data working sets fit within the respective caches. In addition, a fraction of the data cache is reserved for input and output items. Compared to a full fusion strategy, cache aware fusion improves performance by an additional 1.4x on an embedded processor.

Cache Aware Scaling

It is advantageous to execute a filter multiple times at once, because the first execution will incur cache misses that can be amortized over subsequent executions. We user the term *scaling* to refer to the process of increasing a filter's execution multiplicity. While scaling can improve performance by amortizing cold misses of the filter's instructions and state, excessive scaling will worsen performance because the filter's input and output buffers will eventually overflow the cache. This effect is illustrated empirically in Figure 4-17. To achieve the highest performance, the compiler

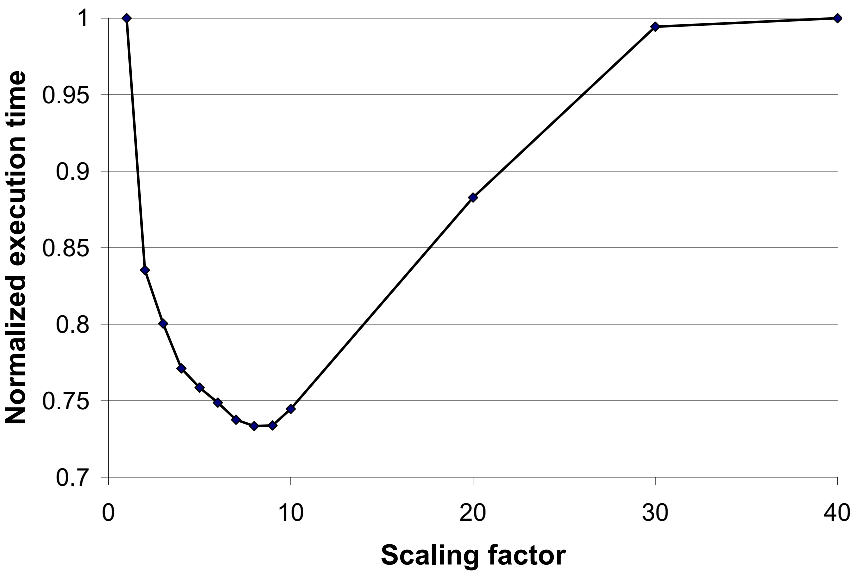


Figure 4-17: Effect of execution scaling on performance. Moderate scaling improves performance by improving the locality of instructions and local filter data. However, excessive scaling worsens performance as the input and output buffers overflow the cache.

needs to select an intermediate scaling factor that represents a tradeoff between the filter's static footprint (instructions and local data) and its dynamic footprint (input and output items).

We have developed a cache aware scaling heuristic that is effective in addressing this problem. The heuristic scales the execution of every filter in the graph by the same amount. The scaling factor is set as high as possible so long as 90% of the filters can fit both their static and dynamic footprints in the cache. This means that 10% of the filters may overflow the cache with their dynamic data, but these overflows are compensated by improved reuse of static data in other filters. In the case of FFT (characterized in Figure 4-17, the heuristic arrives at a scaling factor of 5, which yields performance that is within 5% of the optimum. For our benchmark suite, cache aware scaling gives a further improvement of 1.9x over cache aware fusion alone.

Experimental Evaluation

We implemented cache aware fusion and cache aware scaling in the StreamIt compiler, and evaluate its performance on three different architectures: a 137 MHz StrongARM 1110, a 600 MHz Pentium 3 and a 1.3 GHz Itanium 2. The StrongARM results reflect performance for an embedded target; it has a 16 Kb L1 instruction cache, an 8 Kb L1 data cache, and no L2 cache. The StrongARM also has a separate 512-byte minicache (not targeted by our optimizations). The Pentium 3 and Itanium 2 reflect desktop performance; they have a 16 Kb L1 instruction cache, 16 Kb L1 data cache, and 256 Kb shared L2 cache.

In addition to cache optimizations, we enable aggressive loop unrolling (by a factor of 128) to facilitate scalar replacement. The StreamIt compiler outputs a functionally equivalent C program that is compiled with `gcc` (v3.4, -O3) for the StrongARM and for the Pentium 3 and with `ecc` (v7.0, -O3) for the Itanium 2.

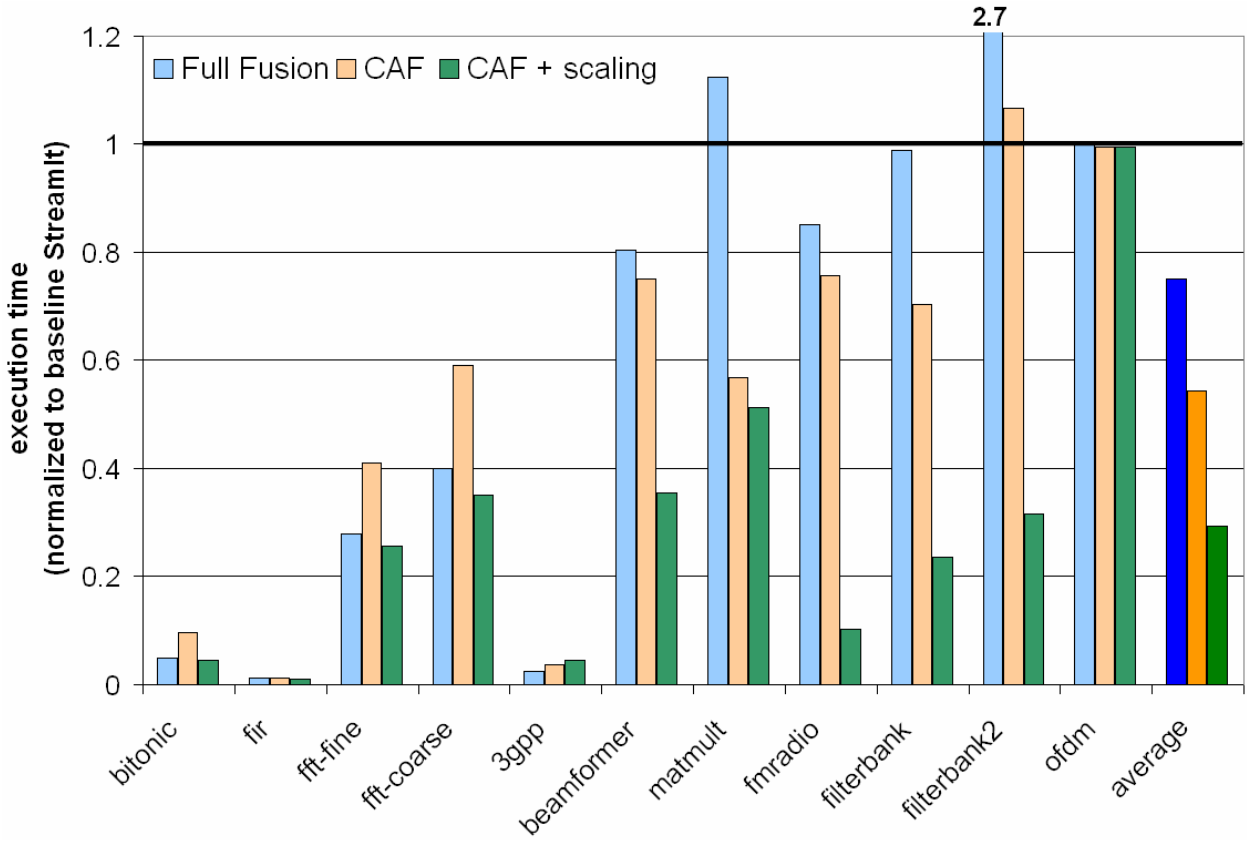


Figure 4-18: Performance of cache optimizations on the StrongARM processor (CAF stands for cache aware fusion)[1].

The performance of our techniques on the StrongARM processor is illustrated in Figure 4-18. The graph illustrates the performance of full fusion, cache aware fusion, and cache aware fusion with cache aware scaling. Performance is normalized to unoptimized StreamIt, in which no actors are fused (but there is still unrolling by 128). On average, our cache optimizations offer a 3.49x speedup over the baseline and a 2.62x average speedup over full fusion. Cache optimizations always perform better than the baseline, and they perform better than full fusion in all cases except for `3gpp`, where they yield a 45% slowdown. This slowdown is due to conservative code size estimation: the compiler predicts that the fused version of `3gpp` will not fit into the instruction cache, thereby preventing fusion. However, due to optimizations by `gcc`, the final code size is smaller than expected and does fit within the cache. While such inaccuracies could be improved by adding feedback between the output of `gcc` and our code estimation, each fusion possibility would need to be evaluated separately as the fusion boundary affects the impact of low-level optimizations (and thus the final code size).

The speedups offered by cache optimizations over a full fusion strategy are more modest for the desktop processors: 1.34x average speedup on Pentium 3 and essentially zero speedup (6% by the arithmetic mean, -8% by the geometric mean) on Itanium 2 (Figure 4-19). Performance on

[1] The benchmarks used here were sometimes parameterized differently than the ones described in Chapter 2. Details on the benchmark configurations are available elsewhere [Ser05].

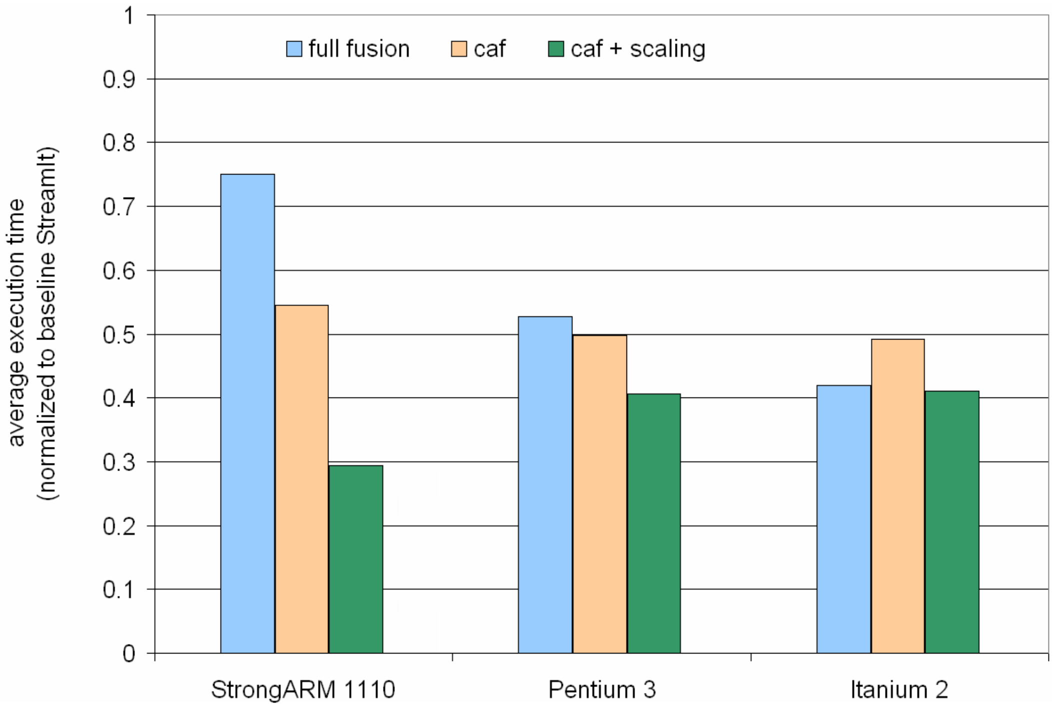


Figure 4-19: Summary of cache optimizations on the StrongARM, Pentium 3 and Itanium 2 processors (CAF stands for cache aware fusion).

any architecture is a tradeoff between two factors: 1) the benefit of data and instruction locality, and 2) the benefit of fusion, which reduces memory accesses due to improved register allocation across actor boundaries. Compared to the StrongARM, the Pentium 3 and Itanium 2 offer an L2 cache (as well as a larger L1 data cache), thereby lessening the impact of locality-enhancing cache optimizations. However, the fusion benefit remains a significant factor; for example, using Intel VTune on the Pentium 3, we measured that full fusion offers a 50% reduction in memory accesses over the cache-optimized version. This effect may be pronounced on the Itanium 2 due to the larger number of registers on that architecture (128 general, 128 floating point). While fusion benefits are also present on the StrongARM, cache optimizations are more important on that processor due to the large penalty for cache misses.

In summary, cache optimizations prove to be a valuable asset to the compiler, especially when targeting embedded processors. Via simple scheduling heuristics, they improve performance by 3.49x. These gains are out of the reach of compilers for traditional languages such as C, in which it is intractable to infer the buffers between filters and to grow or shrink them to match the schedule. The stream programming model exposes the information needed to transform the program and attain the desired performance.

4.4 Related Work

Parallelization Printz also compiles synchronous dataflow graphs (including support for sliding window operations and stateful filters) to the distributed-memory Warp machine [Pri91]. In order to support applications with hard real-time constraints, he minimizes the latency of a single steady state rather than maximizing the overall throughput. Both coarse-grained data parallelism and coarse-grained software pipelining have the potential to increase the latency of the program, and thus fall outside the scope of his model. Also, Printz does not attempt to minimize communication when assigning tasks to processors; data-parallel actors are assigned to processors in any topological order and are always spread across as many processors as possible. As he acknowledges, this sometimes introduces unnecessary communication [Pri91, p.155]. Our technique exploits only as much data parallelism as is needed to complement existing task parallelism in the program. It also matches producers and consumers prior to data parallelization by fusing stateless actors as much as possible. Still, it is difficult to compare our techniques directly because they focus on different things. Our focus is on stateless actors with limitless data parallelism, while Printz focuses on actors with a fixed amount of internal data parallelism. He also has built-in support for systolic computations (pipelines of balanced filters) which we do not model explicitly.

The ASSIGN parallel program generator also maps coarse-grained stream graphs to Warp [O'H91]. However, it relies solely on task and pipeline parallelism; the stream graph is partitioned into contiguous groups for execution on the processors. While we previously adopted a similar approach [GTK+02], it proved difficult to balance the load and to utilize all of the processors without exploiting data parallelism. The approach described in this chapter offers a 1.84x improvement over our previous work.

Liao et al. map Brook to multicore processors by leveraging the affine partitioning model [LDWL06]. While affine partitioning is a powerful technique for parameterized loop-based programs, in StreamIt we simplify the problem by fully resolving the program structure at compile time. This allows us to schedule a single steady state using flexible, non-affine techniques (e.g., simulated annealing) and to repeat the found schedule for an indefinite period at runtime. Gummaraju and Rosenblum map stream programs to a general-purpose hyperthreaded processor [GR05]. Such techniques could be integrated with our spatial partitioning to optimize per-core performance. Gu et al. expose data and pipeline parallelism in a Java-like language and use a compiler analysis to efficiently extract coarse-grained filter boundaries [DFA05]. Ottoni et al. also extract decoupled threads from sequential code, using hardware-based software pipelining to distribute the resulting threads across cores [ORSA05]. By embedding pipeline-parallel filters in the programming model, we focus on the mapping step.

Previous work in scheduling computation graphs to parallel targets has focused on partitioning and scheduling techniques that exploit task and pipeline parallelism [EM87, MSK87, PL95, PBL95, KA99]. Application of loop-conscious transformations to coarse-grained dataflow graphs has been investigated. Unrolling (or "unfolding" in this domain) is employed for synchronous dataflow (SDF) graphs to reduce the initiation interval but they do not evaluate mappings to actual architectures [PM91, CS97]. Software pipelining techniques have been applied to SDF graphs onto various embedded and DSP targets [BG99, CV02], but has required programmer knowledge of both the application and the architecture. To our knowledge, none of these systems automatically exploit the combination of task, data, and pipeline parallelism. Furthermore, these systems do not provide a robust end-to-end path for application parallelization from a high-level, portable programming language.

Optimizing Linear Computations Several other groups have developed automated frameworks for optimizing linear signal processing kernels. SPIRAL is a system that generates libraries for signal processing algorithms [PMJ$^+$05]. Using a feedback-directed search process, DSP transforms are optimized for the underlying architecture. The input language to SPIRAL is SPL [XJJP01, Xio01], 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, due to StreamIt's support for sliding windows. We also support optimizations on linear statespace filters, which are not handled in SPIRAL. In the future, SPIRAL could be integrated with StreamIt to optimize a matrix factorization for a given architecture.

The FFTW system [FJ05] generates platform-optimized FFT libraries using a dynamic programming algorithm and profile feedback to match the recursive FFT formulation to a given memory hierarchy. ATLAS [WPD01, DDE$^+$05] produces platform-specific linear algebra routines by searching over blocking strategies and other parameters; Sparsity [DDE$^+$05, IYV04] applies a similar approach to sparse matrices. StreamIt is again complementary to these packages: it allows programmers to interface with them using general user-level code. It also supports linear statespace filters.

A variety of tools have been developed for specifying and deriving DSP algorithms [ON92]. The SMART project aims to develop an algebraic theory of signal processing, providing a unified framework for deriving, explaining, and classifying fast transform algorithms [PM03]. ADE (A Design Environment) provides a predefined set of composable signal transforms, as well as a rule-based system that searches for improved algorithms using extensible rewriting rules [Cov89]. Janssen et al. automatically derive low-cost hardware implementations of signal flow graphs using algebraic transformations and hill-climbing search [JCM94]. Our work shares the vision of automatically deriving optimized algorithms from a high-level description, though we start from a general-purpose, imperative stream language rather than a mathematical formalism.

Karr [Kar76] and Cousot and Halbwachs [CH78] 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.

Potkonjak and Rabaey describe optimized hardware synthesis for linear and "feedback linear" computations [PR00]. Linear state space systems correspond to "constant feedback linear computations" in the authors' terminology. For linear and linear feedback systems, their technique offers 1) a maximally fast implementation under latency constraints, 2) an arbitrarily fast implementation, and 3) an implementation reducing the number of arithmetic operations. In reducing arithmetic operations, they perform common subexpression elimination (CSE) in a manner that resembles our state removal optimization. However, the general state removal transformation cannot be achieved by CSE alone (or by the Potkonjak and Rabaey algorithm). We are unaware of any sequence of traditional compiler optimizations that achieves the same effect as state removal (and likewise for parameter reduction).

Also note that the "linear data flow analysis" of Ryan [Rya92] is completely unrelated to our work; it aims to do program analysis in linear time.

Cache Optimizations There is a large body of literature on scheduling synchronous dataflow (SDF) graphs to optimize various metrics [BML96, BML99]. The work most closely related to ours is a recent study by Kohli [Koh04] on cache aware scheduling of SDF graphs, implemented as part of the Ptolemy framework for simulating heterogeneous embedded systems [Lee03]. Kohli develops a Cache Aware Scheduling (CAS) heuristic for an embedded target with a software-managed scratchpad instruction cache. His algorithm greedily decides how many times to execute a given actor based on estimates of the data cache and instruction cache penalties associated with switching to the next actor. In contrast, our algorithm considers the buffering requirements of all filters in a given container and increases the multiplicity so long as 90% of buffers are contained within the data cache. The evaluation is limited to one 6-filter pipeline and an assortment of random SDF graphs. An empirical comparison of our heuristics on a common architectural target would be an interesting direction for future work.

It is recognized that there is a tradeoff between code size and buffer size when determining an SDF schedule. Most techniques to date have focused on "single appearance schedules" in which each filter appears at only one position in the loop nest denoting the schedule. Such schedules guarantee minimal code size and facilitate the inlining of filters. There are a number of approaches to minimizing the buffer requirements for single-appearance schedules (see Bhattacharyya [BML99] for a review). While it has been shown that obtaining the minimal memory requirements for general graphs is NP-complete [BML97], there are two complimentary heuristics, APGAN (Pairwise Grouping of Adjacent Nodes) and RPMC (Recursive Partitioning by Minimum Cuts), that have been shown to be effective when applied together [BML97]. Buffer merging[MB99, MB04] represents another technique for decreasing buffer sizes, which could be integrated with our approach in the future.

Govindarajan et al. develop a linear programming framework for determining the "rate-optimal schedule" with the minimal memory requirement [GGD94]. A rate-optimal schedule is one that takes advantage of parallel resources to execute the graph with the maximal throughput. However, the technique is specific to rate-optimal schedules and can result in a code size explosion, as the same node is potentially executed in many different contexts.

The work described above is related to ours in that minimizing buffer requirements can also improve caching behavior. However, our goal is different in that we aim to improve spatial and temporal locality instead of simply decreasing the size of the live data set. In fact, our scaling transformation actually *increases* the size of the data buffers, leading to higher performance across our benchmark suite. Our transformations also take into account the size of the instruction and data caches to select an appropriate scaling and partitioning for the stream graph.

Proebsting and Watterson [PW96] give a fusion algorithm that interleaves the control flow graphs of adjacent filters. Their algorithm supports synchronous `put` and `get` operations (analogous to `push` and `pop`), but would have to be extended to deal with peeking.

4.5 Future Work

Our current parallelization algorithm does not support the full generality of the StreamIt language; it omits support for teleport messages and dynamic rates. Messaging may constrain the latency of certain parts of the stream graph, preventing the compiler from exploiting data parallelism. Also, static rates are important for estimating the work performed by pipeline-parallel filters. In the software pipelining stage, static load balancing would be difficult in the presence of dynamic rates. Incorporating these language features into the parallelization process is fertile grounds for future research.

While our implementation targets Raw, the techniques developed should be applicable to other multicore architectures. As Raw has a relatively high communication bandwidth, coarsening the granularity of data parallelism may benefit other multicores even more. In porting this transformation to a new architecture, one may need to adjust the threshold computation-to-communication ratio that justifies data parallelism. As for coarse-grained software pipelining, the scheduling freedom afforded should benefit many multicore systems. One should consider the most efficient location for intermediate buffers (local memory, shared memory, FIFOs, etc.) as well as the best mechanism for shuffling data (DMA, on-chip network, etc.). The basic algorithms for coarsening granularity, introducing data parallelism, and software pipelining are largely architecture-independent. At the time of this writing, others in the StreamIt group are porting the algorithm to target the Tilera TILE64 chip, the Cell processor, commodity multicores, and a cluster of workstations.

A limitation of the linear optimizations is that they currently do not support linear filters that send or receive teleport messages. This scenario is meaningful because messages are often sent to adjust filtering coefficients in linear nodes. Supporting messages in tandem with linear optimizations represents an interesting research opportunity. If messages update the state of a linear filter, than those state updates should be propagated through any combined or optimized nodes. Alternately, it may be simpler to maintain two versions of the filter at runtime: one which is optimized (and potentially combined with other filters), and one which is unoptimized (and stands alone). The runtime system could speculatively execute the optimized node until messages demand a temporary switch to the unoptimized node.

Our experience with linear optimizations also revealed an opportunity to develop an interesting and useful optimization which we term *decimation propagation*. Our current analysis will automatically propagate any decimation of the output through the linear portions of the stream graph, eliminating any computations that do not affect the final output. However, it would be possible to generalize and extend this analysis to handle non-linear filters as well. By performing a dependence analysis within each filter, one can trace which input items are decimated due to decimations on the output. Comparable to an inter-procedural version of dead code elimination, this optimization could prune unnecessary operations far more effectively than traditional compilers.

The cache aware scaling heuristic applies the same scaling factor to all parts of the stream graph. We have been working (with Fabrice Rastello) on generalizing this approach to use different scaling factors for different sections of the stream graph. This approach has the potential to strike a more flexible tradeoff between the static data footprint and the dynamic data footprint in the cache.

Finally, a broad limitation of the current StreamIt compiler is that it performs all of its optimizations at compile time. To embed streaming concepts in a general-purpose programming language, it will likely be desirable to provide API support for constructing and initializing the stream graph at runtime. In this context, all of the optimizations described should be migrated to a Just-In-Time

(JIT) compiler, where they can be invoked once at the start of steady-state execution. As part of this change, it will be important to evaluate and improve the execution time of all of the compiler analyses.

4.6 Chapter Summary

This chapter presents three aggressive transformations that utilize the abundant parallelism and regular communication patterns of stream programs to achieve automatic performance improvements that are beyond the reach of traditional compilers.

In parallelizing stream programs, we leverage the task, data, and pipeline parallelism that is exposed in the programming model to attain robust performance on a multicore architecture. The key aspect of our work is in exploiting parallelism at a coarse level of granularity. To bolster the benefits of data parallelism on a multicore architecture, we build coarse-grained data-parallel units that are duplicated as few times as needed. And to leverage the benefits of pipeline parallelism, we employ software pipelining techniques—traditionally applied at the instruction level—to coarse-grained filters in the program. The combination of these techniques achieves an 11.2x mean speedup on the 16-core Raw machine.

In optimizing linear computations, we demonstrate how the compiler can mirror the actions of a DSP expert in performing algorithmic transformations on the stream graph. We automatically extract a linear representation from the code in a filter's work function, and manipulate that representation to perform algebraic simplification of adjacent filters, translation of filters into the frequency domain, removal of redundant states, and reduction of the number of parameters. We develop an optimization selection algorithm that uses dynamic programming to choose the most profitable transformations out of a large array of possibilities. The combination of these techniques eliminates an average of 87% of the FLOPs and offers an average speedup of 5.5x across our benchmark suite.

In performing cache optimizations, we derive a schedule of filter executions that improves the instruction and data locality. In order to gain the benefits of fusing nodes together without the hazard of exceeding the instruction cache, a cache aware fusion algorithm fuses only so long as the cache limit is respected. And to amortize the cost of cold misses upon loading a filter's instructions and data into the cache, a cache aware scaling algorithm repeatedly executes each filter many times – but only so long as the resulting communication buffers are unlikely to exceed the cache. The combination of these techniques is especially useful in the context of embedded processors; we achieve a 3.49x average improvement over unoptimized StreamIt on a StrongARM 1100 processor.

There were many aspects of the stream programming model that enabled the optimizations described in this chapter. The properties of the synchronous dataflow model – that is, the separation of filters into independent units with atomic execution steps and known communication rates – was essential for almost all of the analyses described. In addition, the StreamIt construct of peeking was uniformly useful for facilitating analysis of sliding window operations.

Structured streams also found some applications within the compiler, though fewer than we had originally anticipated. The presence of structure was essential for formulating the optimization selection algorithm for linear filters; without the regular structure, there would have been no prescription for finding overlapping subproblems in the stream graph. We believe that structure also simplified the development and presentation of the linear optimizations, as they only have to handle a fixed number of cases rather than dealing with the full generality of an arbitrary stream graph. The splitjoin construct also facilitated the detection of task-parallel components during the introduction of data parallelism. Though we did not discuss it here, the single-input, single-output property of structured streams was integral to the development of phased scheduling [KTA03, Kar02]. Finally, though the StreamIt compiler completely unrolls the stream graph, structure may prove useful in facilitating a parameterized graph representation in future work.

Chapter 5

Translating Stream Programs into the Compressed Domain

This chapter presents a new domain-specific optimization for stream programs: translation to the compressed domain. This transformation allows programs to operate directly on compressed data, accelerating common video editing operations by a median of 15x. Unlike the optimization of linear nodes in StreamIt, this represents a domain-specific optimization that was previously unknown (it was not performed even by experts). We define the transformation in general terms and also evaluate it experimentally in the context of StreamIt.

5.1 Introduction

Stream programs often operate on huge volumes of data. For example, each frame of a digital film requires approximately 2 megabytes, implying that a fully-edited 90-minute video demands about 300 gigabytes of data for the imagery alone [DM05]. Industrial Light and Magic reports that, in 2003, their processing pipeline output 13.7 million frames and their internal network processed 9 petabytes of data [Ben]. The U.S. Geological Survey had archived over 13 million frames of photographic data by the end of 2004, and estimates that 5 years is needed to digitize 8.6 million additional images [U.S04]. In all of these situations, the data is highly compressed to reduce storage costs. At the same time, extensive post-processing is often required for adding captions, watermarking, resizing, compositing, adjusting colors, converting formats, and so on. As such processing logically operates on the uncompressed format, the usual practice is to decompress and re-compress the data whenever it needs to be modified.

In order to accelerate the process of editing compressed data, researchers have identified specific transformations that can be mapped into the compressed domain – that is, they can operate directly on the compressed data format rather than on the uncompressed format [Cha95, Smi95, MIP99, WSA02]. In addition to avoiding the cost of the decompression and re-compression, such techniques greatly reduce the total volume of data processed, thereby offering large savings in both execution time and memory footprint. However, existing techniques for operating directly on compressed data are largely limited to lossy compression formats such as JPEG [SS96b, SS96a, SR96, SS98, DA01, MM02, FJ03] and MPEG [AS98, Vas98, DRB00, NKH00, WSA02]. While these formats are used pervasively in the distribution of image and video content, they are rarely used during the production of such content. Instead, professional artists and filmmakers rely on lossless compression formats (BMP, PNG, Apple Animation) to avoid accumulating artifacts during the editing process. Given the computational intensity of professional video editing, there is a large demand for new techniques that could accelerate operations on lossless formats.

In this chapter, we present a technique for translating stream programs to operate directly on losslessly-compressed data. We consider compression formats that are based on LZ77, a compression algorithm that is utilized by ZIP and fully encapsulates common formats such as Apple Animation, Microsoft RLE, and Targa. The transformation is most efficient when each element of a stream is transformed in a uniform way (e.g., adjusting the brightness of each pixel). However, it also applies to cases in which multiple items are processed at once (e.g., averaging pixels) or in which multiple streams are split or combined (e.g., compositing frames).

The key idea behind our technique can be understood in simple terms. In LZ77, compression is achieved by indicating that a given part of the data stream is a repeat of a previous part of the stream. If a program is transforming each element of the stream in the same way, then any repetitions in the input will necessarily be present in the output as well. Thus, while new data sequences need to be processed as usual, any repeats of those sequences do not need to be transformed again. Rather, *the repetitions in the input stream can be directly copied to the output stream*, thereby referencing the previously-computed values. This preserves the compression in the stream while avoiding the cost of decompression, re-compression, and computing on the uncompressed data.

In this work, we extend this simple idea to encompass a broad class of programs that can be expressed in StreamIt. We have implemented a subset of our general technique in the StreamIt compiler. The end result is a fully-automatic system in which the user writes programs that operate on uncompressed data, and our compiler emits an optimized program that operates directly on compressed data. Our compiler generates plugins for two popular video editing tools (MEncoder and Blender), allowing the optimized transformations to be used as part of a standard video editing process.

Using a suite of 12 videos (screencasts, animations, and stock footage) in Apple Animation format, our transformation offers a speedup roughly proportional to the compression factor. For transformations that adjust a single video (brightness, contrast, color inversion), speedups range from 2.5x to 471x, with a median of 17x. For transformations that combine two videos (overlays and mattes), speedups range from 1.1x to 32x, with a median of 6.6x. We believe this is the first demonstration of compressed-domain techniques for losslessly compressed video content.

In the general case, compressed processing techniques may need to partially decompress the input data to support the behavior of certain programs. Even if no decompression is performed, the output may benefit from an additional re-compression step if new redundancy is introduced during the processing (for example, increasing image brightness can whiteout parts of the image). This effect turns out to be minor in the case of our experiments. For pixel transformations, output sizes are within 0.1% of input sizes and often (more than half the time) are within 5% of a full re-compression. For video compositing, output files maintain a sizable compression ratio of 8.8x (median) while full re-compression results in a ratio of 13x (median).

In the remainder of the chapter, we give an overview of LZ77 compression before describing our technique and its compatibility with popular file formats. We then present an experimental evaluation before closing with related work and a chapter summary.

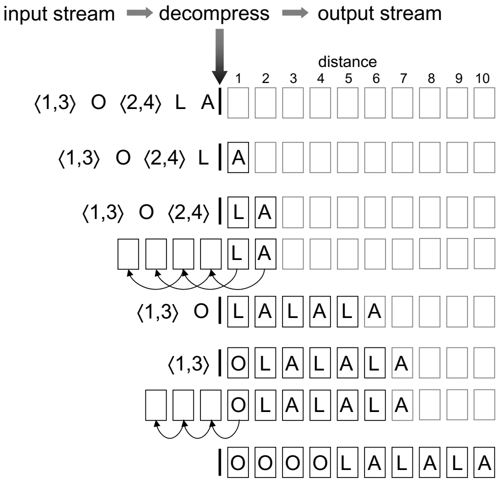


Figure 5-1: Example of LZ77 decompression.

LZ77 Compression

Our technique supports compressed data formats that are based on LZ77. LZ77 is a lossless, dictionary-based compression algorithm that is asymptotically optimal [WZ94]. LZ77 forms the basis for many popular compression formats, including ZIP, GZIP and PNG, and also serves as a generalization of simpler encodings such as Apple Animation, Microsoft RLE, and Targa.

The basic idea behind LZ77 is to utilize a sliding window of recently encoded values as the dictionary for the compression algorithm. In the compressed data stream, there are two types of tokens: *values* and *repeats*. A value indicates a token that should be copied directly to the output of the decoded stream. A repeat $\langle d, c\rangle$ contains two parts: a distance d and a count c. It indicates that the decoder should start at offset d from the end of the decoded stream and copy a sequence of c values to the output. It is important to note that the count may exceed the distance, in which case some of the values produced by a repeat operation are also copied by that operation. For example, a value A followed by a repeat $\langle 1, 3\rangle$ results in an output of “A A A”. An additional example is given in Figure 5-1.

5.2 Mapping into the Compressed Domain

Our technique allows any cyclo-static dataflow program to operate directly on LZ77-compressed data. Rather than modifying the code within the actors, our transformation treats actors as black boxes and wraps them in a new execution layer. The transformation attempts to preserve as much compression as possible without ever performing an explicit re-compression step. While there exist cases in which the output data will not be as compressed as possible, under certain conditions the output is guaranteed to be fully compressed (relative to the compression of the input). We quantify this issue later.

To describe the mapping into the compressed domain, we consider each StreamIt construct in turn. An alternate formulation of our technique, in terms of an operational semantics, is available in a technical report [THA07].

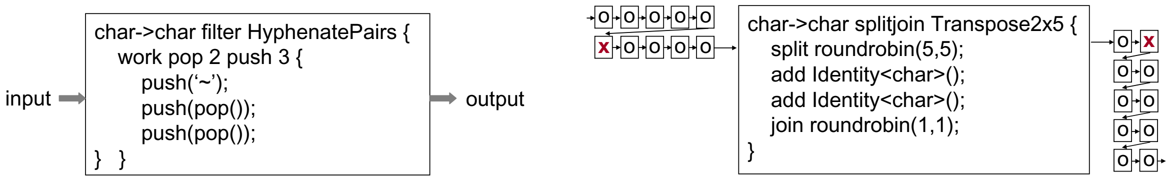
```
Execute a filter in the compressed domain, given that it consumes $n$
items and produces $m$ items on each execution.
EXECUTE-COMPRESSED-FILTER (int $n$, int $m$) {
  while true {
    /* pass-uncompressed */
    if input endswith $n$ values then
      execute one call to uncompressed filter

    /* pass-compressed */
    else if input endswith $\langle d, c \rangle$ and $d\%n = 0$ and $c \geq n$ then
      replace $\langle d, c \rangle$ with $\langle d, c\%n \rangle$ on input
      push $\langle m\ d/n, m\ (c - c\%n)/n \rangle$ to output

    else
      let $\langle d, c \rangle =$ last repeat on input

      /* coarsen-repeat */
      let $L =$ LCM$(d, n)$
      if $d < L < c$ then
        replace $\langle d, c \rangle$ with $\langle c - (L - d) \rangle, \langle d, L - d \rangle$ on input

      /* expand */
      else if $c > 0$ then
        decode $\langle d, c \rangle$ into $\langle d, c - 1 \rangle, V$ on input

      /* prune */
      else /* c = 0 */
        remove $\langle d, c \rangle$ from input
  }
}
```

Figure 5-2: Translation of filters into the compressed domain. We use $\%$ to denote a modulo operation.

Filters

The procedure for translating a filter into the compressed domain is given in Figure 5-2, and an example appears in Figure 5-4. The behavior of the compressed-domain filter can be considered in two pieces. The first piece consists of the simple case (annotated *pass-uncompressed* in the code) in which the upcoming inputs to the filter are uncompressed values. In this case, the original filter is called with those inputs, transforming n input items to m output items. The rest of the code deals with repeat tokens, attempting to translate them across the filter with the minimum decompression needed.

The **key idea behind our technique** is encapsulated in the *pass-compressed* case in Figure 5-2. This case specifies how to translate a repeat token directly from a filter's input tape to a filter's



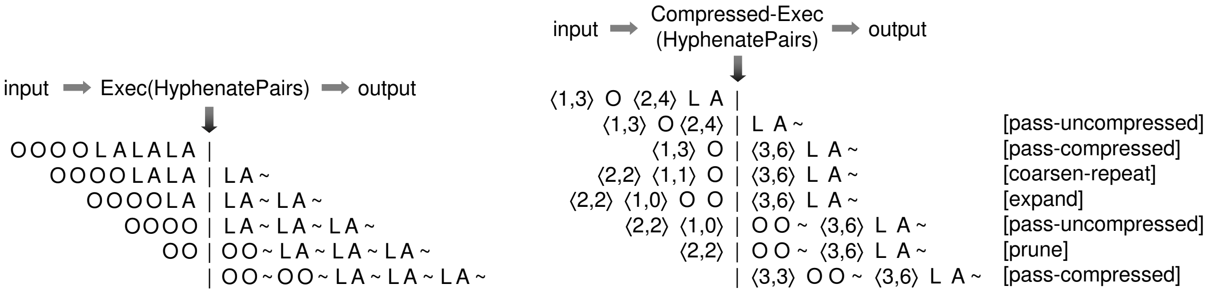
```
char->char filter HyphenatePairs {
   work pop 2 push 3 {
      push('~');
      push(pop());
      push(pop());
}  }
```

(a) Example filter

```
char->char splitjoin Transpose2x5 {
   split roundrobin(5,5);
   add Identity<char>();
   add Identity<char>();
   join roundrobin(1,1);
}
```

(b) Example splitjoin

Figure 5-3: Example filter, splitter, and joiner in StreamIt. The splitter and joiner combine to form a Transpose. Translation to the compressed domain is illustrated in Figures 5-4 and 5-8.



```
OOOOLALALA |
    OOOOLALA | LA~
      OOOOLA | LA~LA~
        OOOO | LA~LA~LA~
          OO | OO~LA~LA~LA~
             | OO~OO~LA~LA~LA~
```

(a) Normal execution

```
⟨1,3⟩ O ⟨2,4⟩ L A |
      ⟨1,3⟩ O ⟨2,4⟩ | L A ~                            [pass-uncompressed]
            ⟨1,3⟩ O | ⟨3,6⟩ L A ~                      [pass-compressed]
      ⟨2,2⟩ ⟨1,1⟩ O | ⟨3,6⟩ L A ~                      [coarsen-repeat]
    ⟨2,2⟩ ⟨1,0⟩ O O | ⟨3,6⟩ L A ~                      [expand]
        ⟨2,2⟩ ⟨1,0⟩ | O O ~ ⟨3,6⟩ L A ~                [pass-uncompressed]
              ⟨2,2⟩ | O O ~ ⟨3,6⟩ L A ~                [prune]
                    | ⟨3,3⟩ O O ~ ⟨3,6⟩ L A ~          [pass-compressed]
```

(b) Compressed-domain execution

Figure 5-4: Example execution of a filter in the uncompressed and compressed domains. See Figure 5-3(a) for the source filter.

output tape without invoking the filter's computation. This translation is possible whenever the repeat distance d is a multiple of the filter's input rate n. In other words, the repeat is aligned with the execution boundaries of the actor, so invoking the actor would produce the same results as before. In transferring the repeat token to the output tape, two adjustments are made: 1) the distance and count are scaled by a factor of m/n to reflect possible differences between the output (m) and input (n) rates of the actor, and 2) if the count is not an even multiple of the input rate, then some leftover items ($c\%n$, where $\%$ represents the modulo operation) are left on the input tape.

In cases where the repeat distance does not match the granularity of the actor, the distance can sometimes be adjusted to allow compressed-domain processing. The *coarsen-repeat* logic in Figure 5-2 represents such an adjustment. Consider that a filter inputs two items at a time, but encounters a long repeat with distance three and count 100. That is, the input stream contains a regular pattern of values with periodicity three. Though consecutive executions of the filter are aligned at different offsets in this pattern, every third filter execution (spanning six values) falls at the same alignment. In general, a repeat with distance d can be exploited by a filter with input rate n by expanding the distance to LCM(d, n). In order to perform this expansion, the count must be greater than the distance, as otherwise the repeat references old data that may have no periodicity. Also, the stream needs to be padded with LCM $- d$ values before the coarsened repeat can begin; this padding takes the form of a shorter repeat using the original distance.

A second way to adjust a repeat token for compressed-domain processing is by changing its count rather than its distance (case *expand* in Figure 5-2). This case applies if a repeat has a count less than n, if it is unaligned with the boundaries of an actor's execution, or if its distance is not a multiple of n (and cannot be coarsened appropriately). The expand logic decodes a single value from a repeat token, thereby decreasing its count by one; the rest of the repeat may become aligned later. If the count of a repeat reaches zero, it is eliminated in the *prune* case.

Note that the *expand* logic requires partial decompression of the data stream. In order to perform this decompression, it may be necessary to maintain an auxiliary data structure–separate from the filter's input stream–that holds a complete window of decompressed data. This auxiliary structure is needed because the sliding-window dictionary of LZ77 makes it difficult to decode one element without decoding others. However, even if the stream is fully decompressed in parallel with the main computation, our technique retains many benefits because the filters still operate on the compressed stream; the volume of data processed is reduced, and the cost of re-compression is averted. For general algorithms such as gzip, compression can be up to 10x slower than decompression [ZSdMNBY00].

Splitters

Duplicate splitters are trivial to transform to the compressed domain, as all input tokens (both values and repeats) are copied directly to the output streams. For roundrobin splitters, the central concern is that a repeat token can only be transferred to a given output tape if the items referenced are also on that tape. If the items referenced by the repeat token were distributed to another tape, then the repeat must be decompressed.

The rest of this section focuses on roundrobin splitters. To simplify the presentation, we consider a splitter with only two output streams, distributing m_1 and m_2 items to each respective stream. This case captures all of the fundamental ideas; extension to additional streams is straightforward. In addition, we use the following notations:

- Splitters adopt a fine-grained cyclo-static execution model, in which each execution step transfers only one item from an input tape to an output tape. That is, a roundrobin(m_1, m_2) splitter has $m_1 + m_2$ distinct execution steps. We refer to every group of $m_1 + m_2$ steps as an *execution cycle*.

- The pseudocode for our algorithm assumes, without loss of generality, that the next execution step of the splitter will write to the first output stream (output1).

- We use *pos* to denote the number of items (in terms of the uncompressed domain) that have already been written to the current output stream (output1) in the current execution cycle. For brevity, the pseudocode does not maintain the value of *pos*, though it is straightforward to do so.

The procedure for executing a roundrobin splitter in the compressed domain appears in Figure 5-5, while an example appears in Figure 5-8. As mentioned previously, a repeat token can be transferred to an output tape so long as the items referenced also appear on that tape. However, the repeat may need to be fragmented (into several repeats of a lesser count), depending on the repeat distance. There are two cases to consider.

```
Execute a roundrobin splitter in the compressed domain, given that it outputs
$m_1$ items to output1 and $m_2$ items to output2 on each execution cycle.
Execute-Compressed-Splitter (int $m_1$, int $m_2$) {
    while true {
        /* pass-uncompressed */
        if input endswith value then
            transfer value from input to output1

        else
            let $\langle d, c\rangle$ = end of input
            let offset = $d\%(m_1 + m_2)$

            /* pass-compressed-long */
            if offset $= 0$ then
                let $(L_1, L_2)$ = Split-To-Both-Streams($c$)
                pop $\langle d, c\rangle$ from input
                push $\langle dm_1/(m_1 + m_2), L_1\rangle$ to output1
                push $\langle dm_2/(m_1 + m_2), L_2\rangle$ to output2

            /* pass-compressed-short */
            else if Split-To-One-Stream($d, c$) $> 0$ then
                let offset' = if offset $\leq pos$ then offset else offset $- m_2$
                let $L$ = Split-To-One-Stream$(d, c)$
                replace $\langle d, c\rangle$ with $\langle d, c - L\rangle$ on input
                push $\langle m_1$ floor$(d/(m_1 + m_2))$ + offset', $L\rangle$ to output1

            /* expand */
            else /* Split-To-One-Stream($d, c$) = 0 */
                decode $\langle d, c\rangle$ into $\langle d, c - 1\rangle, V$ on input

            /* prune */
            if input endswith $\langle d, 0\rangle$ then
                pop $\langle d, 0\rangle$ from input
    }
}
```

Figure 5-5: Translation of splitters into the compressed domain.

The first case, called *pass-compressed-long* in Figure 5-5, distributes an entire repeat token to both output tapes without any fragmentation. This is only possible when the repeat can be cleanly separated into two independent sequences, one offset by m_1 and the next offset by m_2. In other words, the repeat distance must be a multiple of $m_1 + m_2$. In this case, the repeat token is moved to the output streams. The repeat distance is scaled down to match the weight of each stream, and the count is divided according to the current position of the splitter (a simple but tedious calculation implemented by Split-To-Both-Streams in Figure 5-6).

Given that c items are available on input stream of a splitter, returns the number of items that can be written to each output stream before the input is exhausted. Assumes that the splitter is currently writing to the first output stream, to which pos items have previously been written in the current execution cycle.

```
Split-To-Both-Streams (int $c$) returns (int, int) {
  // the number of complete splitter cycles, and the leftover
  let total_cycles = floor(min$(c_1/n_1, c_2/n_2)$)
  let total_leftover = $c\%(m_1 + m_2)$

  // the last partial cycle may end in three regions:
  if total_leftover $\leq m_1 -$ pos then
    // 1. in writing to the first output stream
    $L_1 =$ total_leftover
    $L_2 = 0$
  else if total_leftover $\leq m_1 -$ pos $+ m_2$ then
    // 2. in subsequent writing to the second output stream
    $L_1 = m_1 -$ pos
    $L_2 =$ total_leftover $- m_1 -$ pos
  else
    // 3. in wrap-around writing to the first output stream
    $L_1 =$ total_leftover $- m_2$
    $L_2 = m_2$

  return $(m_1 *$ total_cycles $+ L_1, m_2 *$ total_cycles $+ L_2)$
}
```

Figure 5-6: The Split-To-Both-Streams function is called during compressed splitter execution. In the case where an input token can be split across both output streams, it calculates the maximum numbers of items that can be written to the outputs before the input is exhausted.

The second case, called *pass-compressed-short*, is when the repeat distance is mis-aligned with the splitter's execution cycle, and thus the repeat (if it is long enough) eventually references items that are distributed to a different output tape. Nonetheless, part of the repeat may be eligible to pass through, so long as the items referenced refer to the current output tape. This judgment is performed by Split-To-One-Stream (Figure 5-7) by comparing the repeat distance to the current position in the output stream. If one or more of the repeated values are in range, the valid segment of the repeat (of length *actual_repeat*) is moved to the output tape. As before, the repeat distance needs to be scaled according to the weights of the splitter, and an extra offset is needed if the repeat distance wraps around to reference the end of a previous cycle.

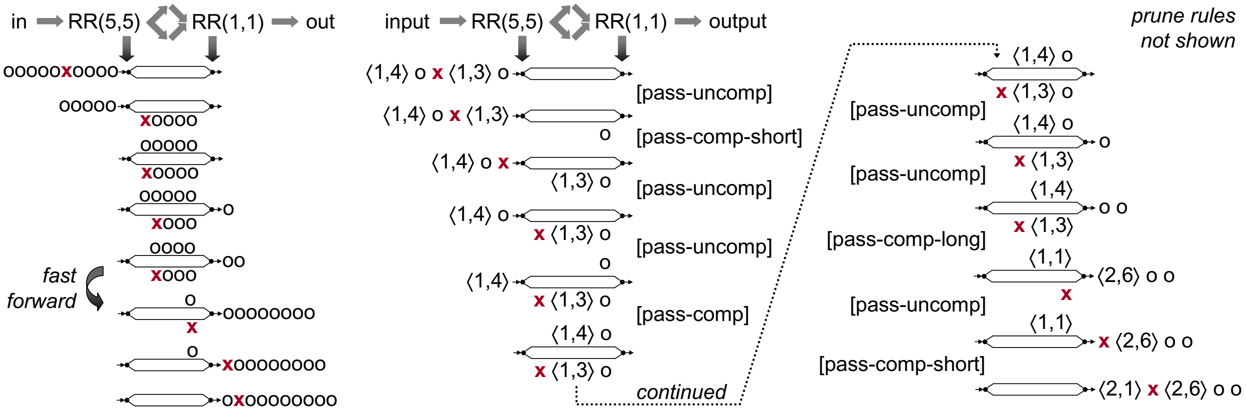
If neither of the above transfers apply, then the input stream needs to be partially decompressed (according to the *expand* case) because the current repeat token references items that will be sent to the wrong output tape. The *prune* case is also needed to clear empty repeats generated by *expand*.

As future work, it would also be desirable to derive an analog of the *coarsen-repeat* logic (Figure 5-2) to preserve even more compression across a splitter. The intuition is that, by increas-

Given a repeat token with distance d and count c that is input to a splitter, and that is not possible to divide across both output streams of the splitter, returns the maximum count of a repeat token that could safely be emitted to the current output stream of the splitter.

```
SPLIT-TO-ONE-STREAM (int $d$, int $c$) returns int {
  let offset $= d\%(m_1 + m_2)$
  if offset $\leq$ pos then
    // repeat for remainder of this execution cycle
    return $\min(c, m_1 - pos)$
  else if offset $> m_2 + pos$ then
    // repeat until referenced data goes out of range
    return $\min(c, \text{offset} - (m_2 + pos))$
  else
    // referenced data is on the other output stream
    return $0$
}
```

Figure 5-7: The SPLIT-TO-ONE-STREAM function is called during compressed splitter execution. In the case where an input token cannot be split across both output streams, it calculates the maximum number of items that can be passed to a single output stream.



(a) Normal execution

(b) Compressed-domain execution of splitter (left) and joiner (right)

Figure 5-8: Example execution of splitters and joiners in the compressed domain. As illustrated by the input/output pairs in Figure 5-3(b), the example performs a transpose of a 2x5 matrix. When the matrix is linearized as shown here, the input stream traverses the elements row-wise while the output stream traverses column-wise. Due to redundancy in the matrix, this reordering can be done largely in the compressed domain.

ing certain repeat distances, the splitter's output tapes can become more independent (referencing themselves rather than each other). This would enable a compressed rule to fire in place of an expansion step.

```
*Execute a roundrobin joiner in the compressed domain, given that it inputs*
*$n_1$ items from input1 and $n_2$ items from input2 on each execution cycle.*
EXECUTE-COMPRESSED-JOINER (int $n_1$, int $n_2$) {
    **while true** {
        /* *pass-uncompressed* */
        **if** input1 **endswith** value **then**
            **transfer** value from input1 to output

        /* *pass-compressed-long* */
        **else if** input1 **endswith** $\langle d_1, c_1 \rangle$ **and** $d_1 \% n_1 = 0$
              **and** input2 **endswith** $\langle d_2, c_2 \rangle$ **and** $d_2 \% n_2 = 0$
              **and** $d_1/n_1 = d_2/n_2$ **then**
            **let** $(L_1, L_2)$ = JOIN-FROM-BOTH-STREAMS($c_1$, $c_2$)
            **replace** $\langle d_1, c_1 \rangle$ with $\langle d_1, c_1 - L_1 \rangle$ on input1
            **replace** $\langle d_1, c_2 \rangle$ with $\langle d_2, c_2 - L_2 \rangle$ on input2
            **push** $\langle d_1(n_1 + n_2)/n_1, L_1 + L_2 \rangle$ to output

        /* *pass-compressed-short* */
        **else** /* *input1 endswith $\langle d, c \rangle$ and $c > 0$* */
            **let** offset = **if** $d \% n_1 \leq$ *pos* **then** *pos* **else** $d \% n_1 + n_2$
            **let** L = Join-From-One-Stream($d$, $c$))
            **replace** $\langle d, c \rangle$ with $\langle d, c - L \rangle$ on input1
            **push** $\langle (n_1 + n_2)$ floor$(d/n_1)$ + offset$, L \rangle$ to output

        /* *prune* */
        **if** input1 **endswith** $\langle d, 0 \rangle$ **then**
            **pop** $\langle d, 0 \rangle$ from input1
        **if** input2 **endswith** $\langle d, 0 \rangle$ **then**
            **pop** $\langle d, 0 \rangle$ from input2
    }
}
```

Figure 5-9: Translation of joiners into the compressed domain.

Joiners

The procedure for executing a joiner in the compressed domain appears in Figure 5-9, while an example appears in Figure 5-8. Analogously to splitters, we consider a roundrobin joiner with only two input streams, collecting n_1 and n_2 items from each respective stream. We also use the following notations:

- Joiners adopt a fine-grained cyclo-static execution model, in which each execution step transfers only one item from an input tape to an output tape. That is, a roundrobin(n_1, n_2) joiner has $n_1 + n_2$ distinct execution steps. We refer to every group of $n_1 + n_2$ steps as an *execution cycle*.

Given that c_1 and c_2 compressed items are available on the first and second input streams of a joiner, returns the number of items that can be read from each input before one of them is exhausted. Assumes that the joiner is currently reading from the first input stream, from which pos items have previously been consumed in the current execution cycle.

```
JOIN-FROM-BOTH-STREAMS (int $c_1$, int $c_2$) returns (int, int) {
  // the number of complete joiner cycles, and the leftovers
  let total_cycles $=$ floor$(c/(n_1 + n_2))$
  let leftover$_1 = c_1 -$ total_cycles $* n_1$
  let leftover$_2 = c_2 -$ total_cycles $* n_2$

  // the last partial cycle may end in three regions:
  if leftover$_1 \leq n_1 -$ pos then
    // 1. in reading from the first input stream
    $L_1 =$ leftover$_1$
    $L_2 = 0$
  else if leftover$_2 \leq n_2$ then
    // 2. in subsequent reading from the second input stream
    $L_1 = n_1 -$ pos
    $L_2 =$ leftover$_2$
  else
    // 3. in wrap-around reading from the first input stream
    $L_1 =$ leftover$_1$
    $L_2 = n_2$

  return $(n_1 *$ total_cycles $+ L_1, n_2 *$ total_cycles $+ L_2)$
}
```

Figure 5-10: The JOIN-FROM-BOTH-STREAMS function is called during compressed joiner execution. In the case where the input tokens to the joiner have compatible repeat distances, it calculates the maximum repeat lengths that can be passed to the output.

- The pseudocode in Figure 5-9 assumes, without loss of generality, that the next execution step of the joiner will read from the first input stream (input1).

- We use *pos* to denote the number of items (in terms of the uncompressed domain) that have already been read from the current input stream (input1) in the current execution cycle. For brevity, the pseudocode does not maintain the value of *pos*, though it is straightforward to do so.

There are two ways to pass repeat tokens through a joiner. If the input streams contain compatible repeat tokens, then they can be combined into a long repeat that spans multiple execution cycles; otherwise, a shorter repeat is extracted from only one of the streams.

The first and most powerful way to execute joiners in the compressed domain is to combine repeat tokens from both input streams (case *pass-compressed-long* in Figure 5-9). For this to be

```
Given a repeat token with distance $d$ and count $c$ on the current input
stream of a joiner, and that cannot be combined with a token on the other
input of the joiner, returns the maximum count of a repeat token that could
safely be emitted to the output stream.
Join-From-One-Stream (int $d$, int $c$) returns int {
   let offset $= d\%n_1$
   if offset $\leq$ pos then
      // repeat for remainder of this execution cycle
      return $\min(c, n_1 - pos)$
   else
      // repeat until referenced data goes out of range
      return $\min(c, \text{offset} - pos)$
}
```

Figure 5-11: The Join-From-One-Stream function is called during compressed joiner execution. In the case where the input tokens to the joiner have incompatible repeat distances, it calculates the maximum length of the current token that may be passable to the output.

possible, both repeat distances must be the same multiple of their respective joiner weight (n_1 or n_2); the combined token has a repeat distance that is a multiple of n_1+n_2. The Join-From-Both-Streams routine (detailed in Figure 5-10) calculates the maximum repeat length depending on the current position of the joiner and the repeat lengths of the inputs.

The second mode of compressed joiner execution (*pass-compressed-short* in Figure 5-9) inputs only a single repeat token, extracting the maximum length that can safely move to the output. The Join-From-One-Stream routine (detailed in Figure 5-11) determines how much of the repeat can be moved to the output before the data referenced would have originated from a different input stream.

As in the case of splitters, further compression gains are possible by adding rules to coarsen the repeat distance or shift the distance to align with other streams. We leave this for future work.

5.3 Supported File Formats

As LZ77 refers to a compression algorithm rather than a complete compression format, there are additional factors to consider in mapping computations to real-world image and video codecs. Some codecs are a subset of LZ77, utilizing only run-length encoding or a fixed window size; these are supported very efficiently by our technique. Others are a superset of LZ77, incorporating additional techniques such as delta coding or Huffman coding; these may incur additional processing overhead. In the following sections, we describe the practical considerations involved in targeting various compression formats with our technique. Formats are ordered by approximate goodness of the achievable mapping.

High-Efficiency Mappings

All of the formats in this category can be considered to be subsets of LZ77.

1. **Apple Animation.** The Apple Animation codec (which forms the basis for our experimental evaluation) is supported as part of the Quicktime MOV container format. It serves as an industry standard for exchanging computer animations and digital video content before they are rendered to lossy formats for final distribution [Ado06, p. 106][HMG04, p. 284] [LS02, p. 367][Pog03, p. 280].

   The Animation codec represents a restricted form of LZ77 in which repeat distances are limited to two values: a full frame or a single pixel. A repeat across frames indicates that a stretch of pixels did not change from one frame to the next, while a repeat across pixels indicates that a stretch of pixels has the same color within a frame.

2. **Flic Video.** Flic Video files (FLI/FLC) were originally produced by Autodesk Animator and are still supported by many animation packages today. Their compression of frame data is almost identical to Apple Animation.

3. **Microsoft RLE.** Microsoft RLE compression can appear in both BMP images and AVI animations. Apart from bit-depth and formatting details, its capabilities are identical to Apple Animation; it can perform run-length encoding within a frame, and can skip over pixels to exploit inter-frame redundancy.

4. **Targa.** The Truevision Targa (TGA) format is a simple image format that is widely used to render frame sequences in the computer animation and video industries. The format includes an optional RLE compression stage, making it a good target for our technique.

5. **PXY.** The pxy format is a research-based image format designed to support efficient transpose and rotation of black-and-white images [Sho95]. It consists of the series of (x, y) coordinates at which the image changes color during a horizontal scan. As this information can be converted to a run-length encoding, it can also be targeted by our technique.

Medium-Efficiency Mappings

While the formats in this category utilize an encoding that is compatible with LZ77, they incur extra overhead because the data is reorganized prior to the compression stage.

1. **Planar RGB.** The Planar RGB video format is supported by Apple Quicktime files. It utilizes run-length encoding for pixels within a frame, with partial support for expressing inter-frame repeats (only the end of lines can be skipped). The red, green, and blue planes are encoded separately in order to increase compression. For user transformations that need to process red, green, and blue values together, this introduces additional alignment overhead when applying our technique.

2. **OpenEXR.** OpenEXR is an emerging image format (backed by Industrial Light and Magic) for use in digital film. It offers several compression options, including run-length encoding, zip, and wavelet-based compression. However, in run-length encoding mode, the low and high bytes of the pixels are separated and encoded as separate run-length sequences; this enables pixels with variations in the low bytes to nonetheless benefit from compression of the high bytes. As most user transformations would utilize the entire bit-width of the pixel, our technique suffers additional alignment overhead in processing these files.

Low-Efficiency Mappings

The formats in this category are supersets of LZ77. While our technique could offer some gains in exploiting the LZ77 compression, it would have to undo any compression sitting on top of LZ77 and offers limited benefit for filters (as in PNG) applied underneath LZ77.

1. **DEFLATE.** DEFLATE is a general-purpose algorithm that provides all of the compression for popular formats such as ZIP and GZIP. The algorithm consists of a full LZ77 encoder followed by Huffman coding, which resizes the symbols in the stream to match their usage frequencies. In targeting ZIP or GZIP with our transformations, we would first have to undo the Huffman coding (unless the application simply reordered data, in which case the coding could remain intact). Though Huffman decoding is a lightweight lookup operation, it would also increase the memory footprint. In addition, as DEFLATE's LZ77 algorithm operates on individual bytes, there may be an exaggerated alignment cost if the application operates on a larger word size.

2. **TSCC.** The TechSmith Screen Capture Codec is very similar to Microsoft RLE, except that the final output is further compressed using DEFLATE. Thus, any overheads incurred by our technique on DEFLATE also extend to TSCC.

3. **PNG.** The PNG image format also relies on DEFLATE to compress the pixels in the image. However, before running DEFLATE, the pixels are usually filtered with a delta encoding; each pixel is replaced with the difference between its value and a predicted value, where the prediction is a linear combination of neighboring pixels. While program segments that compute a linear function [LTA03] could perhaps be mapped to this compressed format, our current technique only applies if the delta encoding is turned off. Even in this scenario, there is a large amount of overhead due to the Huffman coding in DEFLATE.

5.4 Experimental Evaluation

As an initial demonstration of the potential benefits of mapping into the compressed domain, we implemented a core subset of our transformations as part of the StreamIt compiler. Our current implementation supports two computational patterns: 1) transforming each individual element of a stream (via a pop-1, push-1 filter), and 2) combining the elements of two streams (via a roundrobin(1,1) joiner and a pop-2, push-1 filter). The program can contain any number of filters that perform arbitrary computations, so long as the I/O rates match these patterns. While we look forward to performing a broader implementation in future work, these two building blocks are sufficient to express a number of useful programs and to characterize the performance of the technique.

Our evaluation focuses on applications in digital video editing. Given StreamIt source code that operates on pixels from each frame of a video, the StreamIt compiler maps the computation into the compressed domain and emits executable plugins for two popular video editing tools, MEncoder and Blender. The plugins are written for the Apple Animation format (see Section 5.3).

Our benchmarks fall into two categories: 1) pixel transformations, such as brightness, contrast, and color inversion, which adjust pixels within a single video, and 2) video compositing, in which one video is combined with another as an overlay or mask.

	VIDEO	DESCRIPTION	SOURCE	DIMENSIONS	FRAMES	SIZE (MB)	COMPRESSION FACTOR
Internet Video	screencast-demo	Online demo of an authentication generator	Software website	691 x 518	10621	38	404.8
	screencast-ppt	Powerpoint presentation screencast	Self-made	691 x 518	13200	26	722.1
	logo-head	Animated logo of a small rotating head	Digital Juice	691 x 518	10800	330	46.8
	logo-globe	Animated logo of a small rotating globe	Digital Juice	691 x 518	10800	219	70.7
Computer Animation	anim-scene1	Rendered indoor scene	Elephant's Dream	720 x 480	1616	10	213.8
	anim-scene2	Rendered outdoor scene	Elephant's Dream	720 x 480	1616	65	34.2
	anim-character1	Rendered toy character	Elephant's Dream	720 x 480	1600	161	13.7
	anim-character2	Rendered human characters	Elephant's Dream	720 x 480	1600	108	20.6
Digital Television	digvid-background1	Full-screen background with lateral animation	Digital Juice	720 x 576	300	441	1.1
	digvid-background2	Full-screen background with spiral animation	Digital Juice	720 x 576	300	476	1.0
	digvid-matte-frame	Animated matte for creating new frame overlays	Digital Juice	720 x 576	300	106	4.7
	digvid-matte-third	Animated matte for creating new lower-third overlays	Digital Juice	720 x 576	300	51	9.7

Table 5-12: Characteristics of the video workloads.

The main results of our evaluation are:

- Operating directly on compressed data offers a speedup roughly proportional to the compression factor in the resulting video.
- For pixel transformations, speedups range from 2.5x to 471x, with a median of 17x. Output sizes are within 0.1% of input sizes and about 5% larger (median) than a full re-compression.
- For video compositing, speedups range from 1.1x to 32x, with a median of 6.6x. Output files retain a sizable compression ratio (1.0x to 44x) and are about 52% larger (median) than a full re-compression.

The following sections provide more details on our video workloads, the evaluation of pixel transformations, and the evaluation of video compositing.

Video Workloads

Our evaluation utilizes a suite of 12 video workloads that are described in Table 5-12; some of the videos are also pictured in Figure 5-16. The suite represents three common usage scenarios for lossless video formats: Internet screencasts, computer animation, and digital television production. While videos in each area are often rendered to a lossy format for final distribution, lossless codecs are preferred during the editing process to avoid accumulating compression artifacts. All of our source videos are in the Apple Animation format (described in Section 5.3), which is widely used by video editing professionals [Ado06, p. 106] [HMG04, p. 284] [LS02, p. 367] [Pog03, p. 280]. The Apple Animation format is also popular for capturing video from the screen or camera, as the encoder is relatively fast.

Our suite of videos is assembled from a variety of realistic and industry-standard sources. The first screencast is an online demo of an authentication generator for rails [Aut]; the second is a PowerPoint presentation (including animations), captured using Camtasia Studio. As Internet content is often watermarked with a logo or advertisement, we include two animated logos in the "Internet video" category. These logos are taken from Digital Juice [Dig06], a standard source for professional animations, and rendered to Apple Animation format using their software. The animated logos are rendered full-frame (with the logo in the corner) because compositing operations in our testbed (Blender) are done on equal-sized videos.

The computer animation clips are derived from Elephant's Dream, a short film with entirely open-source content [Ele]; our videos are rendered from source using Blender. Finally, the digital television content is also taken from a Digital Juice library [Dig06]. The backgrounds represent high-resolution, rotating backdrops as might appear in the introduction to a program. The mattes are black-and-white animations that can be used to synthesize a smaller overlay (such as a frame or a "lower third", often used for text) from a full animated background (see Figure 5-16b for an example).

The videos exhibit a wide range of compression factors. The screencasts have very high compression (∼400x-700x) because only a small part of the screen (e.g., a mouse, menu, or PowerPoint bullet) is changing on any given frame; the Apple Animation format compresses the inter-frame redundancy. The compression for `anim-scene1` is also in excess of 200x because motion is limited to a small animated character. The animated logos are the next most compressed (∼50-70x), influenced largely by the constant blank region outside the logo. The computer animation content (∼10-30x compression) has a high level of detail but benefits from both inter-frame and intra-frame redundancy, as some rendered regions have constant color. Next are the digital video mattes (∼5-10x compression), which have fine-grained motion in some sections. Finally, the digital video backgrounds offer almost no compression gains (1.0-1.1x) under Apple Animation, as they have pervasive motion and detail across the entire frame.

The Apple Animation format supports various bit depths. All of our source videos use 32 bits per pixel, allocating a single byte for each of the red, green, blue, and alpha channels.

Pixel Transformations

The pixel transformations adjust the color of each pixel in a uniform way. We evaluated three transformations:

- Brightness adjustment, which increases each RGB value by a value of 20 (saturating at 255).
- Contrast adjustment, which moves each RGB value away from the center (128) by a factor of 1.2 (saturating at 0 and 255).
- Color inversion, which subtracts each RGB value from 255 (useful for improving the readability of screencasts or for reversing the effect of video mattes).

We implemented each transformation as a single StreamIt filter that transforms one pixel to another. Because the filter has a pop rate of one, it does not incur any alignment overhead.

Setup The pixel transformations were compiled into plugins for MEncoder, a popular command-line tool (bundled with MPlayer) for video decoding, encoding, and filtering. MEncoder relies on the FFMPEG library to decode the Apple Animation format; as FFMPEG lacked an encoder for this format, the authors implemented one. Additionally, as MEncoder lacks an interface for toggling only brightness or contrast, the baseline configuration was implemented by the authors.

The baseline configuration performs decompression, pixel transformations, then re-compression. Because the main video frame is updated incrementally by the decoder, the pixel transformations are unable to modify the frame in place (otherwise pixels present across frames would be transformed multiple times). Thus, the baseline transformation writes to a separate location in memory. The optimized configuration performs pixel transformations directly on the compressed data,

		SPEEDUP			OUTPUT SIZE / INPUT SIZE (Compute on Compressed Data)			OUPUT SIZE / INPUT SIZE (Uncompress, Compute, Re-Compress)		
	VIDEO	Brightness	Contrast	Inverse	Brightness	Contrast	Inverse	Brightness	Contrast	Inverse
Internet Video	screencast-demo	137.8x	242.3x	154.7x	1.00	1.00	1.00	0.90	0.90	1.00
	screencast-ppt	201.1x	470.6x	185.1x	1.00	1.00	1.00	0.75	0.74	1.00
	logo-head	27.0x	29.2x	25.2x	1.00	1.00	1.00	0.87	0.86	1.00
	logo-globe	35.7x	46.4x	36.6x	1.00	1.00	1.00	1.00	0.64	1.00
Computer Animation	anim-scene1	66.4x	124.3x	58.5x	1.00	0.98	1.00	0.99	0.92	1.00
	anim-scene2	19.3x	27.9x	20.5x	1.00	1.00	1.00	0.99	0.85	1.00
	anim-character1	11.5x	12.2x	11.2x	1.00	1.00	1.00	0.96	0.90	1.00
	anim-character2	15.6x	15.3x	14.8x	1.00	1.00	1.00	0.95	0.88	1.00
Digital Television	digvid-background1	4.6x	2.6x	4.6x	1.00	1.00	1.00	1.00	0.88	1.00
	digvid-background2	4.1x	2.5x	4.7x	1.00	1.00	1.00	0.92	0.91	1.00
	digvid-matte-frame	6.3x	5.3x	6.5x	1.00	1.00	1.00	0.98	0.64	1.00
	digvid-matte-third	7.5x	6.9x	8.9x	1.00	1.00	1.00	0.83	0.35	1.00

Table 5-13: Results for pixel transformations.

avoiding data expansion implied by decompression and multiple frame buffers, before copying the data to the output file.

Our evaluation platform is a dual-processor Intel Xeon (2.2 GHz) with 2 GB of RAM. As all of our applications are single-threaded, the second processor is not utilized. For the timing measurements, we execute each program five times and report the median user time.

Results Detailed results for the pixel transformations appear in Table 5-13. Figure 5-14 illustrates the speedups, which range from 2.5x to 471x. As illustrated in Figure 5-15, the speedups are closely correlated with the compression factor in the original video. For the highly-compressed screencasts and `anim-scene1`, speedups range from 58x to 471x. For the medium-compression computer animations (including the animated logos), speedups range from 11x to 46x. And for the low-compression digital television content, speedups range from 2.5x to 8.9x.

There are two distinct reasons for the speedups observed. First, by avoiding the decompression stage, computing on compressed data reduces the volume of data that needs to be stored, manipulated, and transformed. This savings is directly related to the compression factor and is responsible for the upwards slope of the graph in Figure 5-15. Second, computing on compressed data eliminates the algorithmic complexity of re-compression. For the Apple Animation format, the cost of compressing a given frame does not increase with the compression factor (if anything, it decreases as fewer pixels need a fine-grained encoding). Thus, the baseline devotes roughly constant runtime to re-compressing each video, which explains the positive intercept in the graph of Figure 5-15.

The impact of re-compression is especially evident in the digital television examples. Despite a compression factor of 1.0 on `digvid-background2`, our technique offers a 4.7x speedup on color inversion. Application profiling confirms that 73% of the baseline runtime is spent in the encoder; as this stage is absent from the optimized version, it accounts for $1/(1-0.73) = 3.7$x of the speedup. The remaining speedup in this case is due to the extra frame buffer (and associated memory operations) in the decompression stage of the baseline configuration.

Another important aspect of the results is the size of the output files produced. Apart from the first frame of a video[1], performing pixel transformations directly on compressed data will never

[1] In the Apple Animation format, the first frame is encoded as if the previous frame was black. Thus, adjusting the color of black pixels in the first frame may increase the size of the file, as it removes inter-frame redundancy.

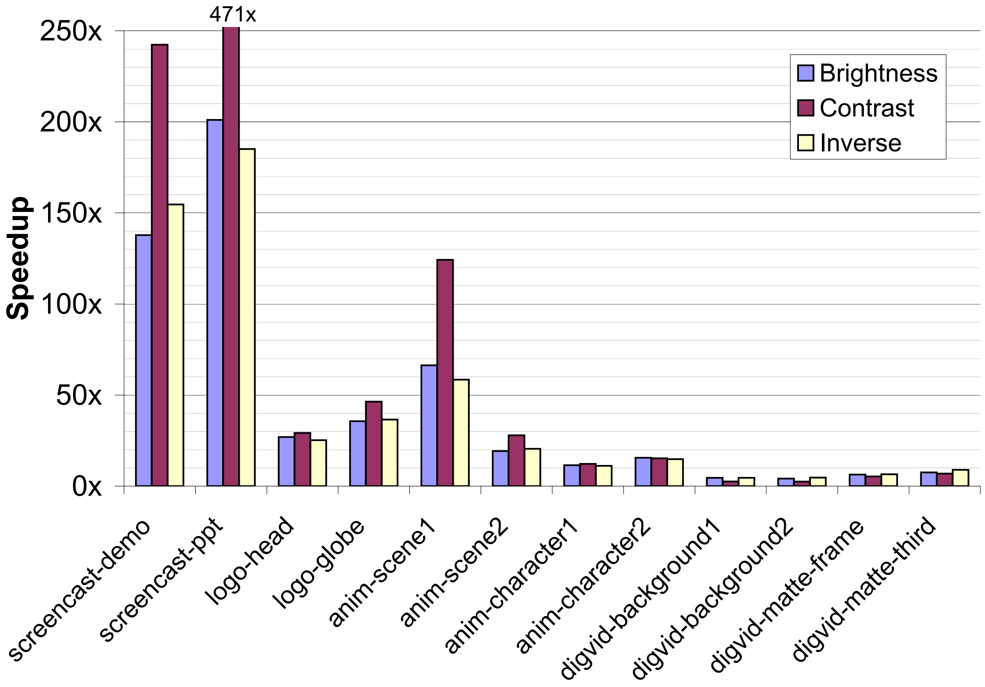


Figure 5-14: Speedup on pixel transformations.

increase the size of the file. This is illustrated in the middle columns of Table 5-13, in which the output sizes are mostly equal to the input sizes (up to 2 decimal places). The only exception is contrast adjustment on `anim-scene1`, in which the output is 2% smaller than the input due to variations in the first frame; for the same reason, some cases experience a 0.1% increase in size (not visisble in the table).

Though computing on compressed data has virtually no effect on the file size, there are some cases in which the pixel transformation increases the redundancy in the video and an additional re-compression step could compress the output even further than the original input. This potential benefit is illustrated in the last three columns of Table 5-13, which track the output size of the baseline configuration (including a re-compression stage) versus the original input. For the inverse transformation, no additional compression is possible because inverse is a 1-to-1 transform: two pixels have equal values in the output file if and only if they have equal values in the input file. However, the brightness and contrast transformations may map distinct input values to the same output value, due to the saturating arithmetic. In such cases, the re-compression stage can shrink the file to as low as 0.75x (brightness) and 0.35x (contrast) its original size. These are extreme cases in which many pixels are close to the saturating point; the median re-compression (across brightness and contrast) is only 10%.

To achieve the minimal file size whenever possible, future work will explore integrating a lightweight re-compression stage into the compressed processing technique. Because most of the compression is already in place, it should be possible to improve the compression ratio without running the full encoder (e.g., run-length encoded regions can be extended without being rediscovered).

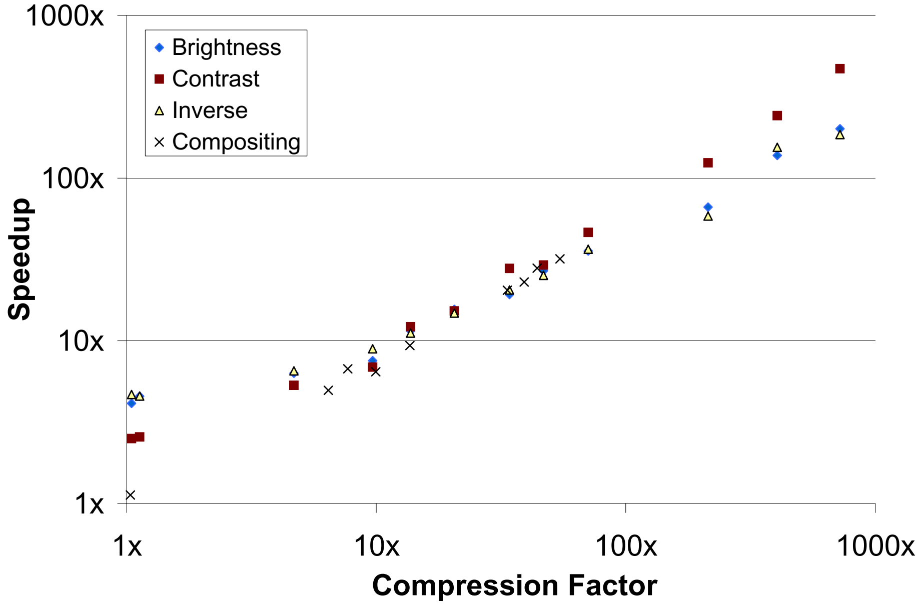


Figure 5-15: Speedup vs. compression factor for all transformations.

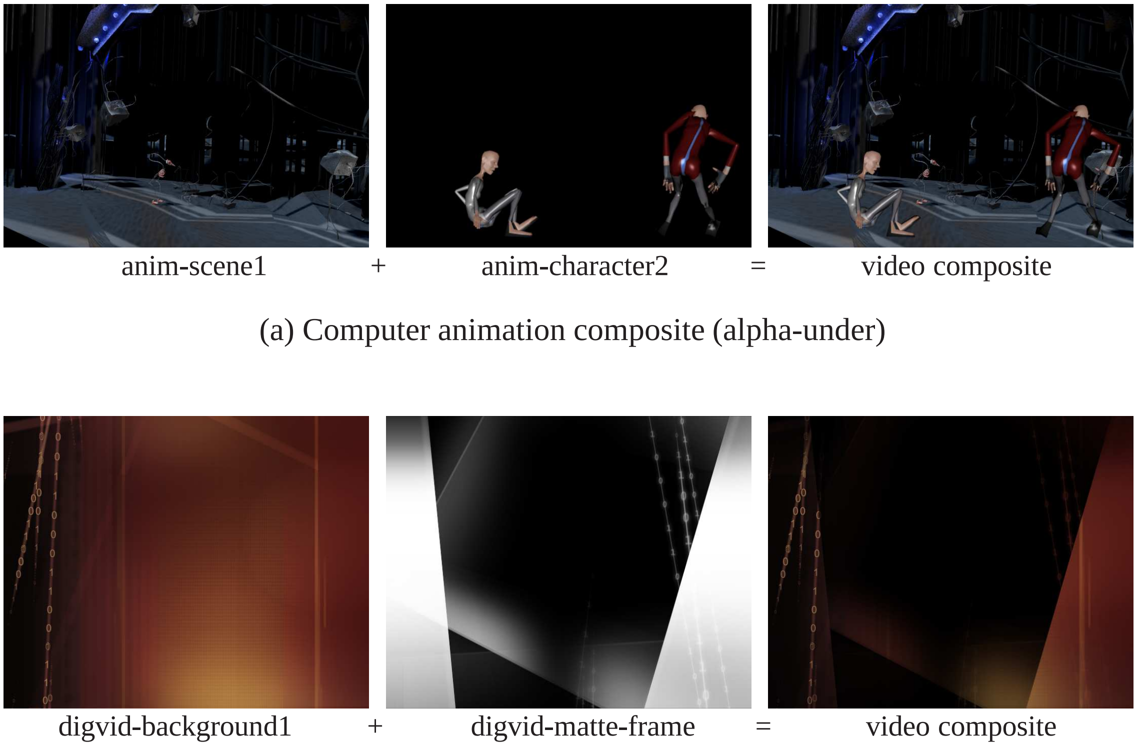
Video Compositing

In video compositing, two videos are combined using a specific function to derive each output pixel from a pair of input pixels (see Figure 5-16). In the case of subtitling, animated logos, and computer graphics, an alpha-under transformation is common; it overlays one video on top of another using the transparency information in the alpha channel. In applying an animated matte, the videos are combined with a multiply operation, thereby masking the output according to the brightness of the matte. For our experiments, we generated composites using each foreground/background pair within a given application area, yielding a total of 12 composites.

In StreamIt, we implemented each compositing operation as a roundrobin(1,1) joiner (to interleave the streams) followed by a filter (to combine the pixel values). The intuition of the compressed-domain execution is that if both streams have the same kind of repeat (inter-frame or intra-frame), then the repeat is copied directly to the output. If they have different kinds of repeats, or if one stream is uncompressed, then both streams are uncompressed.

Setup The compositing operations were compiled into plugins for Blender, a popular tool for modeling, rendering, and post-processing 3-D animations. Blender has logged 1.8 million downloads in the last year [Ble06b] and was used in the production of Spiderman 2 [Ble06a]. Like MEncoder, Blender relies on the FFMPEG library for video coding, so we utilize the same Apple Animation decoder/encoder as in the pixel transformations.

As Blender already includes support for video compositing, we use its implementation as our baseline. The compositing operations have already been hand-tuned for performance; the implementation of alpha-under includes multiple shortcuts, unrolled loops, and the following comment: “this complex optimalisation is because the ’skybuf’ can be crossed in”. We further improved the baseline performance by patching other parts of the Blender source base, which were designed



(a) Computer animation composite (alpha-under)

(b) Digital television composite (multiply)

Figure 5-16: Examples of video compositing operations.

around 3-D rendering and are more general than needed for video editing. We removed two redundant vertical flips for each frame, two redundant BGRA-RGBA conversions, and redundant memory allocation/deallocation for each frame.

Our optimized configuration operates in the compressed domain. Outside of the auto-generated plugin, we patched three frame-copy operations in the Blender source code to copy only the compressed frame data rather than the full frame dimensions.

Results Full results for the compositing operations appear in Table 5-17. Figure 5-18 illustrates the speedups, which range from 1.1x to 32x. As in the case of the pixel transformations, the speedups are closely correlated with the compression factor of the resulting videos, a relationship depicted in Figure 5-15. The highly-compressed screencasts enjoy the largest speedups (20x-32x), the computer animations have intermediate speedups (5x-9x), while the digital television content has negligible speedups (1.1x-1.4x). Overall, the speedups on video compositing (median = 6.6x) are lower than the pixel transformations (median = 17x); this is because the compression achieved on composite videos is roughly proportional to the minimum compression across the two input files.

As for the pixel transformations, the composite videos produced by the compressed processing technique would sometimes benefit from an additional re-compression stage. The last three columns in Table 5-17 quantify this benefit by comparing the compression factors achieved by compressed processing and normal processing (including a re-compression step). For screencasts

				COMPRESSION FACTOR		
	VIDEO COMPOSITE	EFFECT	SPEEDUP	Compute on Compressed Data	Uncompress, Compute, Re-Compress	Ratio
Internet Video	screencast-demo + logo-head	alpha-under	20.46x	34	52	1.55
	screencast-demo + logo-globe	alpha-under	27.96x	44	61	1.39
	screencast-ppt + logo-head	alpha-under	22.99x	39	54	1.38
	screencast-ppt + logo-globe	alpha-under	31.88x	55	64	1.18
Computer Animation	anim-scene1 + anim-character1	alpha-under	6.72x	7.7	12	1.57
	anim-scene1 + anim-character2	alpha-under	9.35x	14	19	1.39
	anim-scene2 + anim-character1	alpha-under	4.96x	6.4	10	1.49
	anim-scene2 + anim-character2	alpha-under	6.45x	10	13	1.32
Digital Television	digvid-background1 + digvid-matte-frame	mul	1.23x	1.0	2.2	2.28
	digvid-background2 + digvid-matte-third	mul	1.13x	1.0	5.6	5.42
	digvid-background2 + digvid-matte-frame	mul	1.38x	1.0	1.8	1.84
	digvid-background2 + digvid-matte-third	mul	1.16x	1.0	4.8	4.91

Table 5-17: Results for composite transformations.

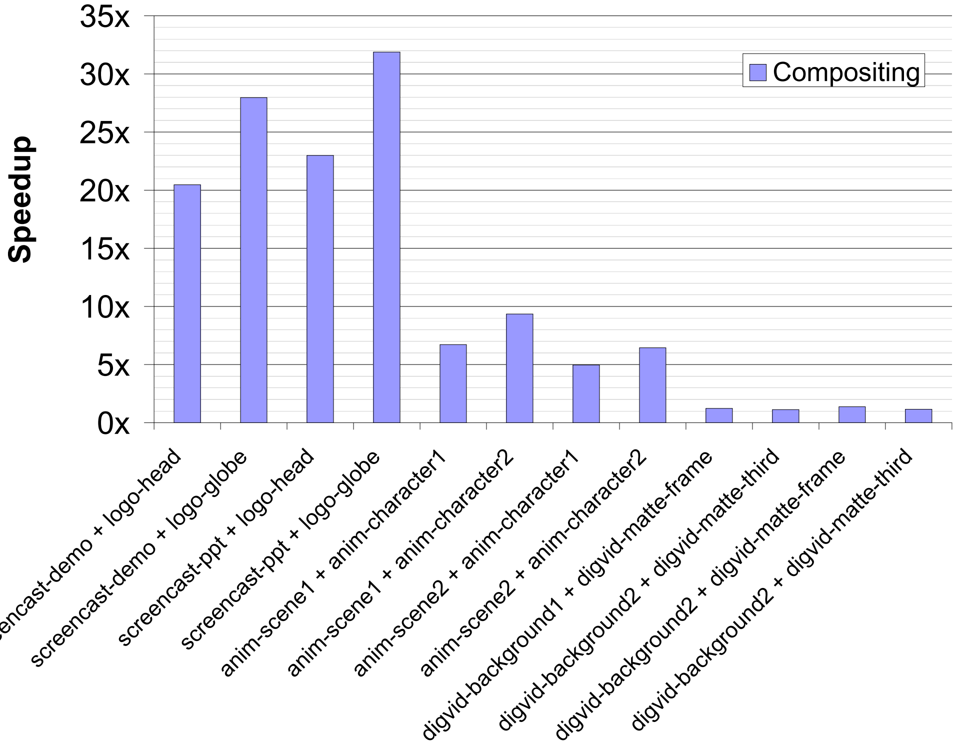


Figure 5-18: Speedup on composite transformations.

and computer animations, compressed processing preserves a sizable compression factor (7.7x-44x), though the full re-compression can further reduce file sizes by 1.2x to 1.6x. For digital television, the matting operations introduce a large amount of redundancy (black regions), thereby enabling the re-compression stage to shrink the file by 1.8x to 5.4x over the compressed processing technique.

Even if a composite transformation does not introduce any new redundancy in the video, the compressed processing technique may increase file sizes by ignoring a specific kind of redundancy in the inputs. Suppose that in the first frame, both inputs are 100% black, while in the second frame, one input is 100% black and the other is 100% white. If the inputs are averaged, the second frame of output will be 100% gray and can be run-length encoded within the frame. However, because the inputs have different kinds of redundancy on the second frame (one is inter-frame, the other is intra-frame), the technique is unable to detect the intra-frame redundancy in the output and will instead produce N distinct pixels (all of them gray). We believe that this effect is small in practice, though we have yet to quantify its impact in relation to the new redundancy introduced by a transformation. Future work will explore alternate data structures for the compressed processing technique that may be able to preserve this redundancy with low overhead.

5.5 Related Work

Several other researchers have pursued the idea of operating directly on compressed data formats. The novelty of our work is two-fold: first, in its focus on lossless compression formats, and second, in its ability to map a flexible stream program, rather than a single predefined operation, into the compressed domain.

Most of the previous work on mapping algorithms into the compressed domain has focused on formats such as JPEG that utilize a Discrete Cosine Transform (DCT) to achieve spatial compression [SS96b, SS96a, SR96, AS98, SS98, Vas98, NKH00, DRB00, DA01, MM02, FJ03]. This task requires a different analysis, with particular attention given to details such as the blocked decomposition of the image, quantization of DCT coefficients, zig-zag ordering, and so-on. Because there is also a run-length encoding stage in JPEG, our current technique might find some application there; however, it appears that techniques designed for JPEG have limited application to formats such as LZ77.

There has been some interest in performing compressed processing on lossless encodings of black-and-white images. Shoji presents the pxy format for performing transpose and other affine operations [Sho95]; the memory behavior of the technique was later improved by Misra et al. [MAC99]. The pxy format lists the (x, y) coordinate pairs at which a black-and-white image changes color during a horizontal scan. As illustrated in Figure 5-8, our technique can also preserve a certain amount of compression during a transpose, though we may achieve lesser compression than the pxy format due to our one-dimensional view of the data.

Researchers have also considered the problem of pattern matching on compressed text. A randomized algorithm has been developed for LZ77 [FT98] while deterministic strategies exist for LZ78 and LZW [Nav03, NT05]. These solutions are specialized to searching text; they do not apply to our transformations, and our technique does not apply to theirs.

In the realm of programming languages, Swartz and Smith present RIVL, a Resolution Independent Video Language [SS95]. The language is used to describe a sequence of image transformations; this allows the compiler to analyze the sequence and, via lazy evaluation, to eliminate any operations that do not effect the final output. Such a technique is complementary to ours and could also be implemented using StreamIt as the source language.

5.6 Future Work

There remain rich areas for future work in computing on compressed data. First, the compressed processing technique can be applied far beyond the current focus. In its current form, the technique could be evaluated on video operations such as thresholding, color depth reduction, sepia toning, saturation adjustment, and color replacement. With minor extensions, the technique can support video operations such as cropping, padding, histograms, image flipping, sharpening, and blurring. The technique may also have applications in an embedded setting, where it could offer power savings – for example, in processing the RAW data format within digital cameras. It may even be possible to do sparse matrix operations using the technique; in addition to compressing the locations of the zero elements, LZ77 would also compress repetitive patterns in the non-zero elements.

Research is also underway to apply a similar technique to lossy, DCT-based compression formats. Because these formats represent a linear encoding, they are subject to the linear optimizations described in the previous chapter. That is, a JPEG transcoder typically performs an iDCT (during decompression), followed by the user's transformation, followed by a DCT (during compression). If the user's transformation is also linear (e.g., color inversion) then all three stages can be automatically collapsed, thereby eliminating the decompression and re-compression steps. Preliminary experiments in this direction indicate speedups upwards of 10x. Additional research will be needed to support piecewise linear transformations, such as brightness adjustment with saturation at the maximum brightness level. By extending the framework to multiple compression formats, users will be able to write their transformations once, in a high-level language, and rely on the compiler to map the computations to each of the compressed domains.

While we formulated our transformation in terms of a streaming model, the techniques can be applied within other functional and general-purpose languages so long as the right information is available and certain constraints are satisfied. The transformation relies on a regular pattern of data access; we use a streaming abstraction, but structured iteration over arrays could also suffice. We rely on static data rates in actors, which could also be expressed as functions with a fixed number of arguments and return values. Actors (functions) must be pure, without side effects or unresolvable dependences on potentially mutable data. While these properties are intrinsic to a language such as StreamIt, they also come naturally in most functional languages and may be adaptable to general-purpose languages in the form of a runtime library with a restricted API.

5.7 Chapter Summary

In order to accelerate operations on compressible data, this chapter presents a general technique for translating stream programs into the compressed domain. Given a natural program that operates on uncompressed data, our transformation outputs a program that directly operates on the compressed data format. We support lossless compression formats based on LZ77. In the general case, the transformed program may need to partially decompress the data to perform the computation, though this decompression is minimized throughout the process and significant compression ratios are preserved without resorting to an explicit re-compression step.

We implemented some of our transformations in the StreamIt compiler and demonstrated excellent speedups. Across a suite of 12 videos in Apple Animation format, computing directly on compressed data offers a speedup roughly proportional to the compression ratio. For pixel transformations (brightness, contrast, inverse) speedups range from 2.5x to 471x, with a median of 17x; for video compositing operations (overlays and mattes) speedups range from 1.1x to 32x, with a median of 6.6x. While previous researchers have used special-purpose compressed processing techniques to obtain speedups on lossy, DCT-based codecs, we are unaware of a comparable demonstration for lossless video compression. As digital films and animated features have embraced lossless formats for the editing process, the speedups obtained may have practical value.

Chapter 6

Migrating Legacy C Programs to a Streaming Representation

This chapter stands independently of the StreamIt project. Rather than starting with a stream programming language, we consider the problem of starting with a legacy C application and migrating the code to a streaming representation. To address this problem, we equip the programmer with a simple set of annotations (indicating possible filter boundaries) and a dynamic analysis that tracks all communication across those boundaries. Our analysis outputs a stream graph of the application as well as a set of macros for (unsoundly) parallelizing the program and communicating the data needed.

Our analysis is unsound because it is based on a fixed set of dynamic traces, rather than a conservative static analysis. However, we argue that this unsoundness enables us to provide programmers with more information, that is ultimately more useful, than can be expected from a static analysis. We apply our methodology to six case studies, including MPEG-2 decoding, MP3 decoding, GMTI radar processing, and three SPEC benchmarks. Our analysis extracts a useful block diagram for each application, facilitating a translation to StreamIt and other stream languages. In addition, the parallelized versions run correctly (given appropriate training inputs) and offer a 2.78x mean speedup on a 4-core machine.

6.1 Introduction

While adopting a stream programming model is an attractive approach for improving the performance of future applications, one of the drawbacks of relying on a new programming model is that it does not immediately address the vast quantities of legacy code that have already been written in other languages. There are 310 billion lines of legacy code in industry today, and 75-80% of the typical IT budget is spent maintaining legacy systems [HH06]. While much of this code is amenable to streaming, the process of migrating the code to a streaming representation is an arduous and time-consuming process. The most important resources that could help with the translation – such as the original author of the code, or the high-level design documents that guided its implementation – are often unavailable. Thus, a fresh programmer is left with the daunting task of obtaining an in-depth understanding of all the program modules, the dependences between them, and the possibilities for safely extracting parallelism.

While there have been many efforts to automatically parallelize legacy codes, few of them have focused on the pipeline parallelism that is characteristic of the streaming domain. It is even difficult to express pipeline parallelism in a traditional programming model. This is in stark contrast to task parallelism, which is naturally supported by threads, as well as data parallelism, which is supported by dialects such as OpenMP. The only efforts to exploit pipeline parallelism in C programs have been very fine-grained, partitioning individual instructions across processing cores [ORSA05]. Such fine-grained communication is inefficient on commodity machines and demands new hardware support [RVVA04, ORSA05]. While a coarse-grained partitioning is more desirable, it is difficult to achieve at compile time due to the obscured data dependences in C; constructs such as pointer arithmetic, function pointers, and circular buffers (with modulo operations) make it nearly impossible to extract coarse-grained parallelism from realistic C programs.

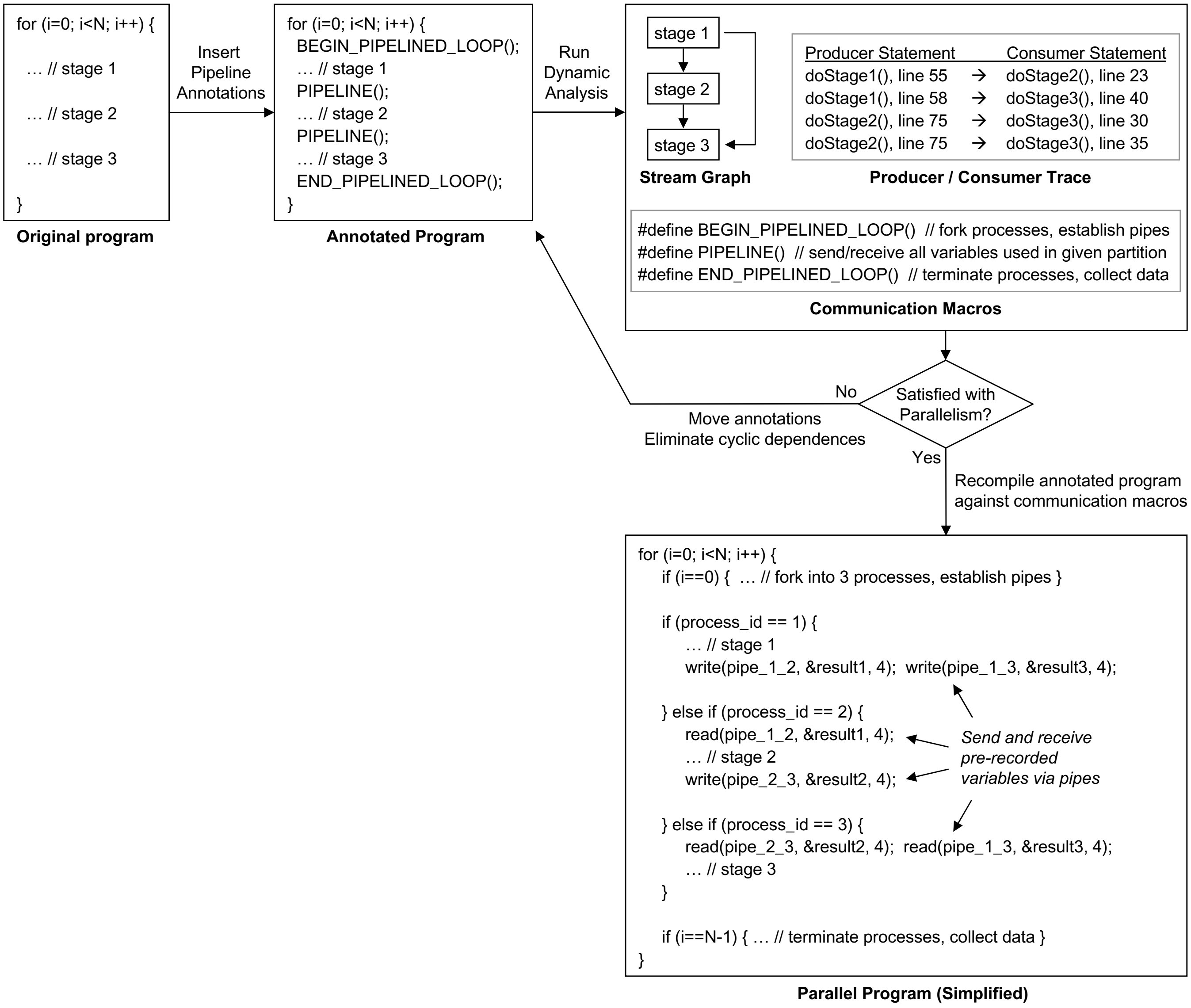
In this chapter, we overcome the traditional barriers in exploiting coarse-grained pipeline parallelism by embracing an *unsound* program transformation. Our key insight is that, for stream programs, the data communicated across pipeline-parallel stages is stable throughout the lifetime of the program. No matter how obfuscated the C implementation appears, the heart of the algorithm is following a regular communication pattern. For this reason, it is unnecessary to undertake a heroic static analysis; we need only observe the communication pattern at the beginning of execution, and then "safely" infer that it will remain constant throughout the rest of execution (and perhaps other executions).

As depicted in Figure 6-1, our analysis does exactly that. We allow the programmer to naturally specify the boundaries of pipeline partitions, and then we record all communication across those boundaries during a training run. The communication trace is emitted as a stream graph that reflects the high-level structure of the algorithm (aiding a possible translation to StreamIt), as well as a list of producer/consumer statements that can be used to trace down problematic dependences. The programmer never needs to worry about providing a "correct" partitioning; if there is no parallelism between the suggested partitions, it will result in cycles in the stream graph. If the programmer is satisfied with the parallelism in the graph, he recompiles the annotated program against a set of macros that are emitted by our analysis tool. These macros serve to fork each partition into its own process and to communicate the recorded locations using pipes between processes.

Though our transformation is grossly unsound, we argue that it is quite practical within the domain of streaming applications. Because pipeline parallelism is deterministic, any incorrect transformations incurred by our technique can be identified via traditional testing methods, and failed tests can be fixed by adding the corresponding input to our training set. Further, the communication trace provided by our analysis is useful in aiding manual parallelization of the code – a process which, after all, is only sound insofar as the programmer's understanding of the system. By improving the programmer's understanding, we are also improving the soundness of the current best-practice for parallelizing legacy C applications.

We have applied our methodology to six case studies: MPEG-2 decoding, MP3 decoding, GMTI radar processing, and three SPEC benchmarks. Our tool was effective at parallelizing the programs, providing a mean speedup of 2.78x on a four-core architecture. Despite the potential unsoundness of the tool, our transformations correctly decoded ten popular videos from YouTube, ten audio tracks from MP3.com, and the complete test inputs for GMTI and SPEC benchmarks. At the same time, we did identify specific combinations of training and testing data (for MP3) that lead to erroneous results. Thus, it is important to maximize the coverage of the training set and to apply the technique in concert with a rigorous testing framework.

The remainder of this chapter is organized as follows. In Section 6.2, we show that stream


Insert Pipeline Annotations
Run Dynamic Analysis
stage 1
stage 2
stage 3
Stream Graph
Producer / Consumer Trace
Communication Macros
No
Satisfied with Parallelism?
Move annotations
Eliminate cyclic dependences
Yes
Recompile annotated program against communication macros
Send and receive pre-recorded variables via pipes

```
for (i=0; i<N; i++) {

  … // stage 1

  … // stage 2

  … // stage 3

}
```

Original program

```
for (i=0; i<N; i++) {
  BEGIN_PIPELINED_LOOP();
  … // stage 1
  PIPELINE();
  … // stage 2
  PIPELINE();
  … // stage 3
  END_PIPELINED_LOOP();
}
```

Annotated Program

Producer Statement		Consumer Statement
doStage1(), line 55	→	doStage2(), line 23
doStage1(), line 58	→	doStage3(), line 40
doStage2(), line 75	→	doStage3(), line 30
doStage2(), line 75	→	doStage3(), line 35

```
#define BEGIN_PIPELINED_LOOP()  // fork processes, establish pipes
#define PIPELINE()  // send/receive all variables used in given partition
#define END_PIPELINED_LOOP()  // terminate processes, collect data
```

```
for (i=0; i<N; i++) {
    if (i==0) {  … // fork into 3 processes, establish pipes }

    if (process_id == 1) {
        … // stage 1
        write(pipe_1_2, &result1, 4);  write(pipe_1_3, &result3, 4);

    } else if (process_id == 2) {
        read(pipe_1_2, &result1, 4);
        … // stage 2
        write(pipe_2_3, &result2, 4);

    } else if (process_id == 3) {
        read(pipe_2_3, &result2, 4);  read(pipe_1_3, &result3, 4);
        … // stage 3
    }

    if (i==N-1) { … // terminate processes, collect data }
}
```

Parallel Program (Simplified)

Figure 6-1: Overview of our approach.

programs have a stable communication pattern. Communication observed at the start of execution is often preserved throughout the program lifetime, as well as other executions. Section 6.3 describes our dynamic analysis tool and programmer methodology to iteratively extract stream parallelism from C programs. Section 6.4 describes the implementation of the tool using the Valgrind infrastructure. Section 6.5 is devoted to our case studies, including performance results and the experience gained during parallelization. The remaining sections present related work, future work, and a chapter summary.

6.2 Stability of Stream Programs

A dynamic analysis is most useful when the observed behavior is likely to continue, both throughout the remainder of the current execution as well as other executions (with other inputs). Our hypothesis is that stream programs exhibit very stable flows of data, enhancing the reliability of dynamic analyses toward the point where they can be trusted to validate otherwise-unsafe program transformations. For the purpose of our analysis, we consider a program to be *stable* if there is a predictable set of memory dependences between pipeline stages. The boundaries between stages are specified by the programmer using a simple set of annotations; the boundaries used for the experiments in this section are illustrated by the stream graphs that appear later (Figure 6-10).

Stability Within a Single Execution

Our first experiment explores the stability of memory dependences within a single program execution. We profiled MPEG-2 and MP3 decoding using the most popular content from YouTube[1] and MP3.com; results appear in Figures 6-2 and 6-3. These graphs plot the cumulative number of unique addresses that are passed between program partitions as execution proceeds. The figures show that after a few frames, the program has already performed a communication for most of the addresses it will ever send between pipeline stages.

In the case of MPEG-2, all of the address traces remain constant after 50 frames, and 8 out of 10 traces remain constant after 20 frames. The videos converge at different rates in the beginning due to varying parameters and frame types; for example, video 10 contains an intra-coded frame where all other videos have a predictive-coded frame, thereby delaying the use of predictive buffers in video 10. Video 1 communicates more addresses than the others because it has a larger frame size.

MP3 exhibits a similar stability property, though convergence is slower for some audio tracks. While half of the tracks exhibit their complete communication pattern in the first 35 frames, the remaining tracks exhibit a variable delay (up to 420 frames) in making the final jump to the common communication envelope. These jumps correspond to elements of two parameter structures which are toggled only upon encountering certain frame types. Track 10 is an outlier because it starts with a few layer-1 frames, thus delaying the primary (layer-3) communication and resulting in a higher overall communication footprint. The only other file to contain layer-1 frames is track 9, resulting in a small address jump at iteration 17,900 (not illustrated).

It is important to note that there does exist a dynamic component to these applications; however, the dynamism is contained within a single pipeline stage. For example, in MP3, there is a Huffman decoding step that relies on a dynamically-allocated lookup tree. Throughout the program, the shape of the tree grows and shrinks and is manipulated on the heap. Using a static analysis, it is difficult to contain the effects of such dynamic data structures; a conservative pointer or shape analysis may conclude that the dynamism extends throughout the entire program. However, using a dynamic analysis, we are able to observe the actual flow of data, ignoring the intra-node communication and extracting the regular patterns that exist between partitions.

Stability Across Different Executions

The communication patterns observed while decoding one input file can often extend to other inputs as well. Figures 6-4 and 6-5 illustrate the minimum number iterations (i.e., frames) that need to be profiled from one file in order to enable correct parallel decoding of the other files. In

[1] YouTube videos were converted from Flash to MPEG-2 using ffmpeg and vixy.net.

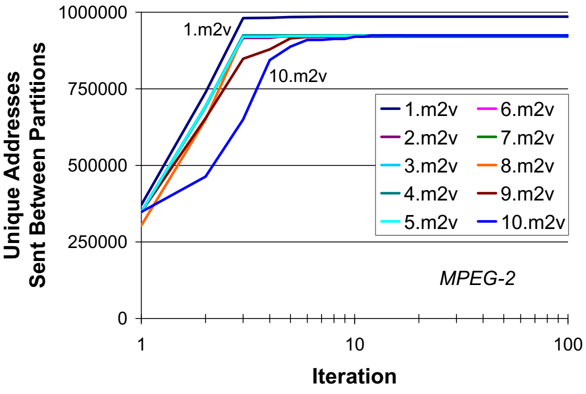


Figure 6-2: Stability of streaming communication patterns for MPEG-2 decoding. The decoder was monitored while processing the top 10 short videos from YouTube. See Figure 6-10a for a stream graph of the application.

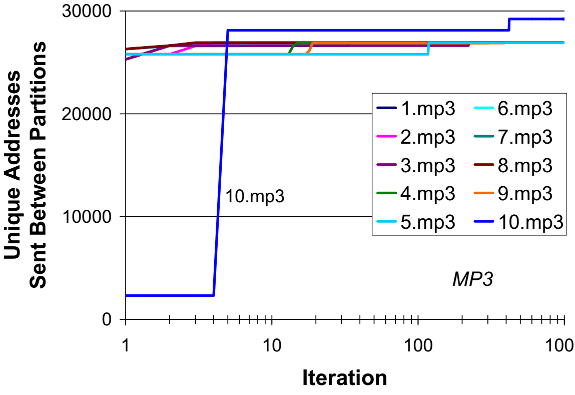


Figure 6-3: Stability of streaming communication patterns for MP3 decoding. The decoder was monitored while processing the top 10 tracks from MP3.com. See Figure 6-10b for a stream graph of the application.

MPEG-2		Testing File									
		1.m2v	2.m2v	3.m2v	4.m2v	5.m2v	6.m2v	7.m2v	8.m2v	9.m2v	10.m2v
Training File	1.m2v	3	3	3	3	3	3	3	3	3	3
	2.m2v	3	3	3	3	3	3	3	3	3	3
	3.m2v	5	5	5	5	5	5	5	5	5	5
	4.m2v	3	3	3	3	3	3	3	3	3	3
	5.m2v	3	3	3	3	3	3	3	3	3	3
	6.m2v	3	3	3	3	3	3	3	3	3	3
	7.m2v	3	3	3	3	3	3	3	3	3	3
	8.m2v	3	3	3	3	3	3	3	3	3	3
	9.m2v	3	3	3	3	3	3	3	3	3	3
	10.m2v	4	4	4	4	4	4	4	4	4	4

Figure 6-4: Minimum number of training iterations (frames) needed on each video in order to correctly decode the other videos.

MP3		Testing File									
		1.mp3	2.mp3	3.mp3	4.mp3	5.mp3	6.mp3	7.mp3	8.mp3	9.mp3	10.mp3
Training File	1.mp3	1	1	1	1	1	1	1	1	—	—
	2.mp3	1	1	1	1	1	1	1	1	—	—
	3.mp3	1	1	1	1	1	1	1	1	—	—
	4.mp3	1	1	1	1	1	1	1	1	—	—
	5.mp3	1	1	1	1	1	1	1	1	—	—
	6.mp3	1	1	1	1	1	1	1	1	—	—
	7.mp3	1	1	1	1	1	1	1	1	—	—
	8.mp3	1	1	1	1	1	1	1	1	—	—
	9.mp3	1	1	1	1	1	1	1	1	17900	—
	10.mp3	5	5	5	5	5	5	5	5	5	5

Figure 6-5: Minimum number of training iterations (frames) needed on each track in order to correctly decode the other tracks.

most cases, a training set of five loop iterations is sufficient to infer an address trace that correctly decodes the other inputs in their entirety. The exceptions are tracks 9 and 10 of MP3 decoding, which are the only two files containing layer-1 frames; because they execute code that is never reached by the other files, training on the other files is insufficient to expose the full communication trace. In addition, track 9 is insufficient training for track 10, as the latter contains an early CRC error that triggers a unique recovery procedure. As each of these hazards is caused by executing code that is untouched by the training set, the runtime system could easily detect such cases (using guards around untrained code) and revert to a sequential execution for the iterations in question. Rigorous testing practices that incorporate code coverage metrics would also help to reduce the risk of encountering unfamiliar code at runtime.

The ability to generalize short training runs across multiple executions relies on two aspects of our methodology. First, as described later, we require the user to supply a symbolic size for each dynamically-allocated variable; this allows MPEG-2 address traces to apply across different frame sizes. Second, we coarsen the granularity of the trace to treat structure types and dynamically-allocated segments as atomic units. That is, whenever a single element of such a structure is communicated between partitions, the rest of the structure is communicated as well (so long as it does not conflict with a local change in the target partition). Such coarsening increases the tolerance to small element-wise changes as observed in later iterations of MPEG-2 and MP3. However, it does not trivialize the overall result, as coarsening is only needed for a small fraction of communicated addresses (15% for MP3 and dependent on frame size for MPEG-2).

While we have focused on MPEG-2 and MP3 in this section, we observe similar stability across our other benchmarks (GMTI, bzip2, parser, and hmmer). As described in Section 6.5, we profile five iterations of a training file and (with minimal programmer intervention) apply the trace to correctly execute a test file.

6.3 Migration Methodology

We introduce a dynamic analysis tool that empowers the programmer in migrating legacy C applications to a streaming representation. Using this tool, the programmer follows the workflow illustrated in Figure 6-1. The first step is to identify the main loop in the application, which is typically iterating over frames, packets, or another long-running data source. The programmer annotates the start and end of this loop, as well as the boundaries between the desired pipeline-parallel partitions. The tool reports the percentage of execution time spent in each pipeline stage in order to help guide the placement of pipeline boundaries.

In our current implementation, there are some restrictions on the placement of the partition boundaries. All boundaries must appear within the loop body itself, rather than within a nested loop, within nested control flow, or as part of another function (this is an artifact of using macros to implement the parallelism). The programmer may work around these restrictions by performing loop distribution or function inlining. Also, though both `for` loops and `while` loops are supported, there cannot be any `break` or `continue` statements within the loop; such statements implicitly alter the control flow in all of the partitions, an effect that is difficult to trace in our dynamic analysis. If such statements appear in the original code, the programmer needs to convert them to a series of `if` statements, which our tool will properly handle.

Once a loop has been annotated with partition boundaries, the programmer selects a set of training inputs and runs our dynamic analysis to trace the communication pattern. The tool outputs a stream graph, a list of producer/consumer statements, and a set of communication macros for automatically running the code in parallel.

An example stream graph for GMTI radar processing appears in Figure 6-6. The graph extracted by our tool is very similar to the block diagram from the GMTI specification, which appears in Figure 6-7. Our graph contains some additional edges that are not depicted in the specification; these represent communication of minor flags rather than the steady-state dataflow. Edges flowing from a node back unto itself (e.g., in Setup, Beamformer, and Tracker) indicate mutable state that is retained across iterations of the main loop. Nodes without such dependences are stateless with respect to the main loop, and the programmer may choose to execute them in a data-parallel manner (see below). Overall, the tight correspondence between our extracted stream graph and the original specification demonstrates that the tool can effectively capture the underlying communication patterns, assisting the programmer in understanding the opportunities and constraints for parallelization.

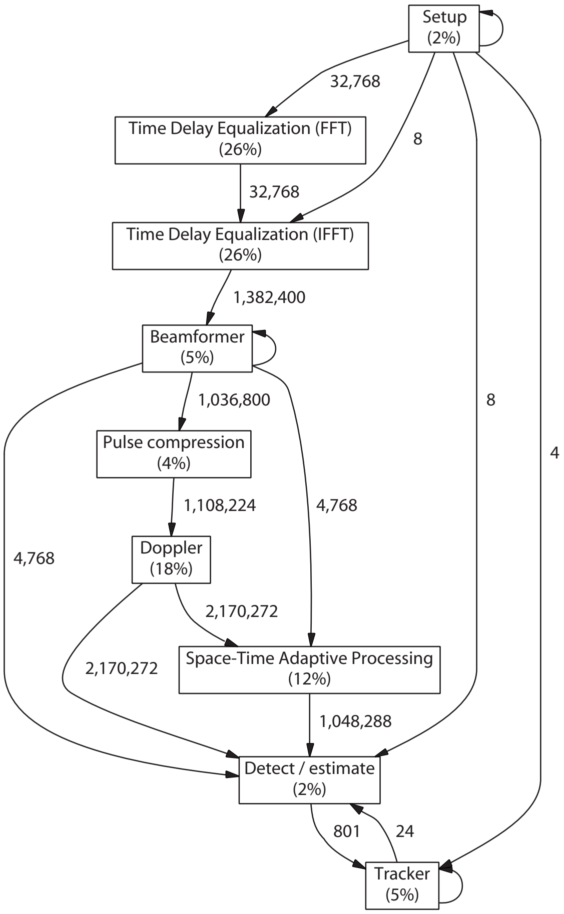


Figure 6-6: Stream graph for GMTI, as extracted using our tool. Nodes are annotated with their computation requirements, and edges are labeled with the number of bytes transferred per steady-state iteration.

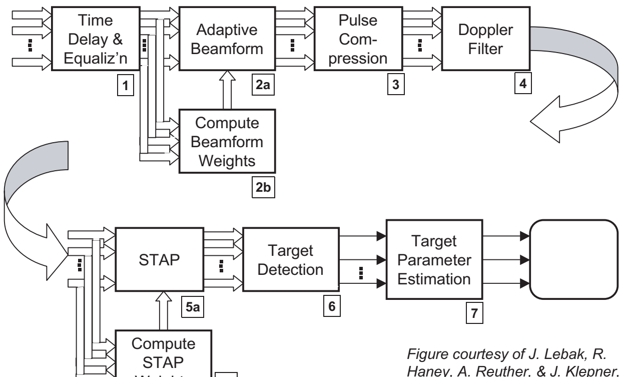


Figure courtesy of J. Lebak, R. Haney, A. Reuther, & J. Klepner, MIT Lincoln Laboratories

Figure 6-7: Stream graph for GMTI, as it appears in the GMTI specification [Reu03].

```
for (i=0; i<N; i++) {
   BEGIN_PIPELINED_LOOP();
   … // stage 1
   PIPELINE(W);
   … // stage 2
   PIPELINE();
   … // stage 3
   END_PIPELINED_LOOP();
}
```

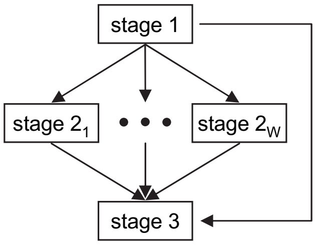


Figure 6-8: Programmers can specify data parallelism by passing an extra argument to the pipeline annotation. In this case, the runtime system executes W parallel copies of stage 2.

Many nodes in a streaming application are suitable to data parallelism, in which multiple loop iterations are processed in parallel by separate instances of the node. Such nodes are immediately visible in the stream graph, as they lack a carried dependence[2] (i.e., a self-directed edge). Our tool offers natural support for exploiting data parallelism: the user simply provides an extra argument to the `PIPELINE` annotation, specifying the number of ways that the following stage should be replicated (see Figure 6-8). While this annotation does not affect the profiler output, it is incorporated by the runtime system to implement the intended parallelism.

Depending on the parallelism evident in the stream graph, it may be desirable to iterate the parallelization process by adjusting the pipeline partitions as well as the program itself. The partitions can execute in a pipeline-parallel manner so long as there are no cyclic dependences between them. If there are any strongly connected components in the stream graph, they will execute sequentially; the programmer can reduce the overhead by collapsing such partitions into one. Alternately, the programmer may be able to verify that certain dependences can safely be ignored, in which case our analysis tool will filter them out of future reports. For example, successive calls to malloc result in a data dependence that was originally reported by our tool; however, this dependence (which stems from an update of a memory allocation map) does not prohibit parallelism because the calls can safely execute in any order. Additional examples of non-binding dependences include legacy debugging information such as timers, counters, etc. that are not observable in the program output. Sometimes, dependences can also be removed by eliminating the reuse of certain storage locations (see Section 6.5 for details).

Once the programmer is satisfied with the parallelism in the stream graph, the code can automatically be executed in a pipeline-parallel fashion using the communication macros emitted by the tool. In most cases, the macros communicate items from one partition to another using the corresponding variable name (and potential offset, in the case of arrays) from the program. However, a current limitation is in the case of dynamically-allocated data, where we have yet to automate the discovery of variable name given the absolute addresses that are communicated dynamically. Thus, if the tool detects any communication of dynamically-allocated data, it alerts the user and indicates the line of the program that is performing the communication. The user needs to supply a symbolic expression for the name and size of the allocated region. Only two of our six benchmarks (MPEG-2 and bzip2) communicate dynamically-allocated data across partition boundaries.

[2] In some cases, nodes with carried dependences on an outer loop can still be data-parallelized on an inner loop. We perform such a transformation in MP3, though it is not fully automatic.

6.4 Implementation

Dynamic Analysis Tool

Our tool is built on top of Valgrind, a robust framework for dynamic binary instrumentation [NS07]. Our analysis interprets every instruction of the program and (by tracing the line number in the annotated loop) recognizes which partition it belongs to. The analysis maintains a table that indicates, for each memory location, the identity of the partition (if any) that last wrote to that location. On encountering a store instruction, the analysis records which partition is writing to the location. Likewise, on every load instruction, the analysis does a table lookup to determine the partition that produced the value being consumed by the load. Every unique producer-consumer relationship is recorded in a list that is output at the end of the program, along with the stream graph and communication macros.

There are some interesting consequences of tracking dependence information in terms of load and store instructions. In order to track the flow of data through local variables, we disable register allocation and other optimizations when preparing the application for profiling. However, as we do not model the dataflow through the registers, the tool is unable to detect cases in which loaded values are never used (and thus no dependence exists). This pattern often occurs for short or unaligned datatypes; even writes to such variables can involve loads of neighboring bytes, as the entire word is loaded for modification in the registers. Our tool filters out such dependences when they occur in parallel stack frames, i.e., a spurious dependence between local variables of two neighboring function calls. Future work could further improve the precision of our reported dependences by also tracking dependences through registers (in the style of Redux [NM03]).

As the dynamic analysis traces communication in terms of absolute memory locations, some engineering was required to translate these addresses to variable names in the generated macros. (While absolute addresses could also be used in the macros, they would not be robust to changes in stack layout or in the face of re-compilation.) We accomplish this mapping using a set of gdb scripts[3], which provide the absolute location of every global variable as well as the relative location of every local variable (we insert a known local variable and print its location as a reference point). In generating the communication code, we express every address as an offset from the first variable allocated at or below the given location. In the case of dynamically-allocated data, the mapping from memory location to variable name is not yet automated and requires programmer assistance (as described in the previous section).

Parallel Runtime System

The primary challenge in implementing pipeline parallelism is the need to buffer data between execution stages. In the sequential version of the program, a given producer and consumer takes turns in accessing the shared variables used for communication. However, in the parallel version, the producer is writing a given output while the producer is still reading the previous one. This demands that the producer and consumer each have a private copy of the communicated data, so that they can progress independently on different iterations of the original loop. Such a transformation is commonly referred to as “double-buffering”, though we may wish to buffer more than two copies to reduce the synchronization between pipeline stages.

[3] Our scripts rely on having compiled with debug information.

Benchmark	Description	Source	Lines of Code
MPEG-2	MPEG-2 video decoder	MediaBench [LPMS97]	10,000
MP3	MP3 audio decoder	Fraunhofer IIS [Fra03b]	5,000
GMTI	Ground Moving Target Indicator	MIT Lincoln Laboratory [Reu03]	37,000
197.parser	Grammatical parser of English language	SPECINT 2000	11,000
256.bzip2	bzip2 compression and decompression	SPECINT 2000	5,000
456.hmmer	Calibrating HMMs for biosequence analysis	SPECCPU 2006	36,000

Table 6-9: Benchmark characteristics.

There are two broad approaches for establishing a buffer between pipeline stages: either explicitly modify the code to do the buffering, or implicitly wrap the existing code in a virtual environment that performs the buffering automatically. The first approach utilizes a shared address space and modifies the code for the producer or consumer so that they access different locations; values are copied from one location to the other at synchronization points. Unfortunately, this approach requires a deep program analysis in order to infer all of the variables and pointer references that need to be remapped to shift the produced or consumed data to a new location. Such an analysis seems largely intractable for a language such as C.

The second approach, and the one that we adopt, avoids the complexities of modifying the code by simply forking the original program into multiple processes. The memory spaces of the processes are isolated from one another, yet the processes share the exact same data layout so no pointers or instructions need to be adjusted. A standard inter-process communication mechanism (such as pipes) is used to send and buffer data from one process to another; a producer sends its latest value for a given location, and the consumer reads that value into the same location in its private address space. At the end of the loop's execution, all of the processes copy their modified data (as recorded by our tool during the profiling stage) into a single process that continues after the loop. Our analysis also verifies that there is no overlap in the addresses that are sent to a given pipeline stage; such an overlap would render the program non-deterministic and would likely lead to incorrect outputs.

6.5 Case Studies

To evaluate our approach, we applied our tool and methodology to six realistic programs. Three of these are traditional stream programs (MPEG-2 decoding, MP3 decoding, GMTI radar processing) while three are SPEC benchmarks (parser, bzip2, hmmer) that also exhibit regular flows of data. As illustrated in Table 6-9, the size of these benchmarks ranges from 5 KLOC to 37 KLOC. Each program processes a conceptually-unbounded stream of input data; our technique adds pipeline parallelism to the toplevel loop of each application, which is responsible for 100% of the steady-state runtime. (For bzip2, there are two toplevel loops, one for compression and one for decompression.)

In the rest of this section, we first describe our experience in parallelizing the benchmarks before presenting performance results.

Parallelization Experience

During the parallelization process, the programmer relied heavily on the stream graphs extracted by our tool. The final graphs for each benchmark appear in Figures 6-10 and 6-11. In the graphs, node labels are gleaned from function names and comments in the code, rather than from any domain-specific knowledge of the algorithm. Nodes are also annotated with the amount of work they perform, while edges are labeled with the number of bytes communicated per steady-state iteration. Nodes that were data-parallelized are annotated with their multiplicity; for example, the Dequantize stage in MP3 (Figure 6-10b) is replicated twice.

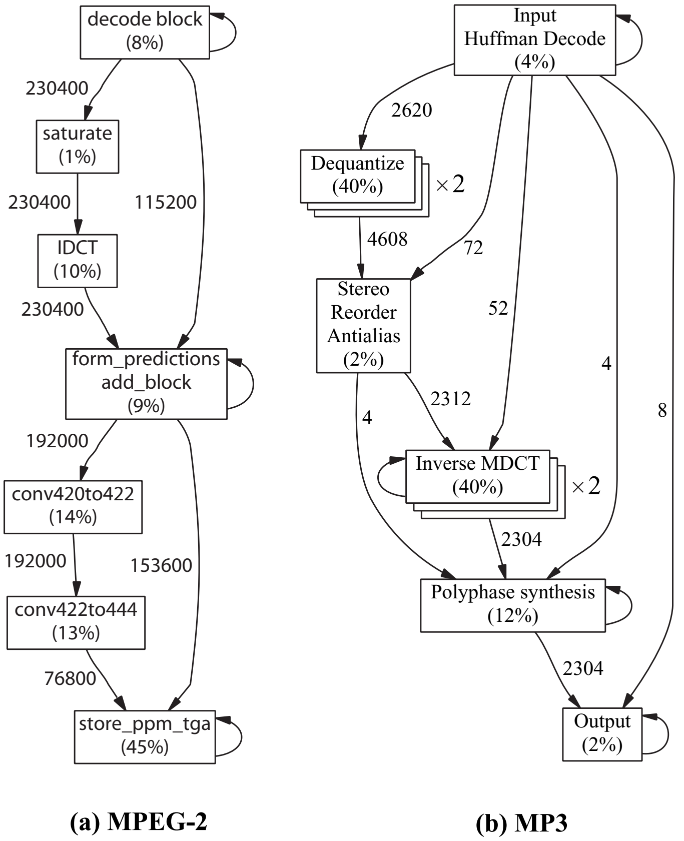
As described in Section 6.3, our tool relies on some programmer assistance to parallelize the code. The manual steps required for each benchmark are summarized in Figure 6-12 and detailed in the following sections.

MPEG-2 Decoding To obtain the stream graph for MPEG-2 (Figure 6-10a), the programmer iteratively refined the program with the help of the dynamic analysis tool. Because the desired partition boundaries fell in distinct functions, those functions were inlined into the main loop. Early return statements in these functions led to unstructured control flow after inlining; the programmer converted the control flow to if/else blocks as required by our analysis. The tool exposed an unintended data dependence that was inhibiting parallelism: a global variable (progressive_frame) was being re-used as a temporary variable in one module. The programmer introduced a unique temporary variable for this module, thereby restoring the parallelism. In addition, the updates to some counters in the main loop were reordered so as to place them in the same pipeline stage that the counters were utilized.

In generating the parallel version, our tool required two interventions from the programmer. First, as the pipeline boundaries spanned multiple loop nests, the communication code (auto-generated for a single loop nest) was patched to ensure that matching send and receive instructions executed the same number of times. Second, as described in Section 6.3, the programmer supplied the name and size of dynamically-allocated variables (in this case, frame buffers) that were sent between partitions.

MP3 Decoding The extracted stream graph for MP3 decoding appears in Figure 6-10b. In the process of placing the pipeline boundaries, the programmer inlined functions, unrolled two loops, and distributed a loop. Four dynamically-allocated arrays (of fixed size) were changed to use static allocation, so that our tool could manage the communication automatically. As profiling indicated that the dequantization and inverse MDCT stages were consuming most of the runtime, they were each data-parallelized two ways.

In analyzing the parallelism of MP3, the programmer made three deductions. First, the initial iteration of the loop was found to exhibit many excess dependences due to one-time initialization of coefficient arrays; thus, the profiling and parallelization was postponed to the second iteration. Second, though the tool reports a carried dependence in the inverse MDCT stage, the programmer found that this dependence is on an outer loop and that it is safe to data-parallelize the stage on an inner loop. Finally, the programmer judged the execution to be insensitive to the ordering of diagnostic print statements, allowing the dependences between statements to be ignored for the sake of parallelization. (With some additional effort, the original ordering of print statements can always be preserved by extracting the print function into its own pipeline stage.)



(a) MPEG-2 **(b) MP3**

Figure 6-10: Extracted stream graphs for MPEG-2 and MP3 decoding.

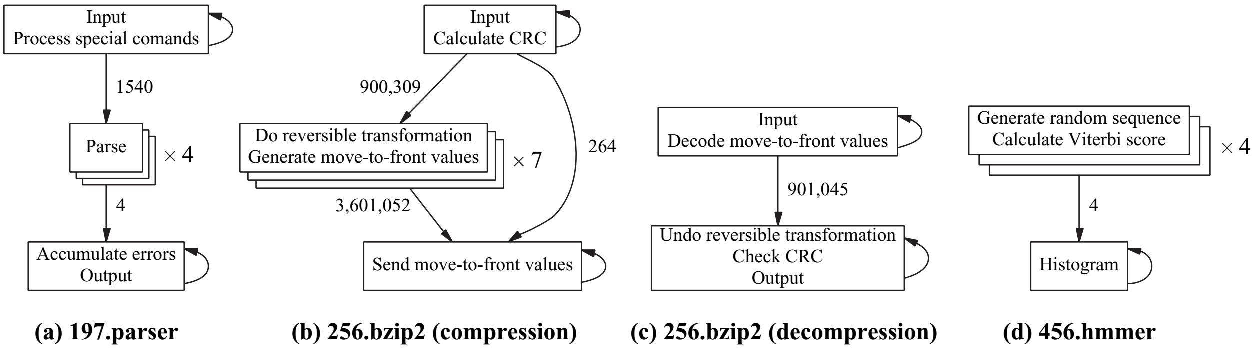
As in the case of MPEG-2, the programmer also patched the generated communication code to handle nested loops.

GMTI Radar Processing The Ground Moving Target Indicator (GMTI) is a radar processing application that extracts targets from raw radar data [Reu03]. The stream graph extracted by our tool (Figure 6-6) is very similar to the one that appears in the GMTI specification (Figure 6-7).

In analyzing GMTI, the programmer made minor changes to the original application. The programmer inlined two functions, removed the application's self-timers, and scaled down an FFT window from 4096 to 512 during the profiling phase (the resulting communication code was patched to transfer all 4096 elements during parallel execution).

As print statements were judged to be independent of ordering, the tool was instructed to ignore the corresponding dependences. Dependences between calls to memory allocation functions (malloc/free) were also disregarded so as to allow pipeline stages to manage their local memories in parallel. The programmer verified that regions allocated within a stage remained private to that stage, thus ensuring that the parallelism introduced could not cause any memory hazards.

Our tool reported an address trace that was gradually increasing over time; closer inspection revealed that an array was being read in a sparse pattern that was gradually encompassing the entire data space. The programmer directed the tool to patch the parallel version so that the entire array was communicated at once.



(a) 197.parser **(b) 256.bzip2 (compression)** **(c) 256.bzip2 (decompression)** **(d) 456.hmmer**

Figure 6-11: Extracted stream graphs for parser, bzip2 (compression and decompression) and hmmer.

Parser The stream graph for 197.parser appears in Figure 6-11a. Each steady-state iteration of the graph parses a single sentence; the benchmark runs in batch mode, repeatedly parsing all of the sentences in a file. As indicated in the graph, the cyclic dependences in the benchmark are limited to the input stage (which performs file reading and adjusts the configuration of the parser) and the output stage (which accumulates an error count). The parsing stage itself (which represents most of the computation) retains no mutable state from one sentence to the next, and can thus be replicated to operate on many sentences in parallel. In our optimized version, the parsing stage is replicated four times.

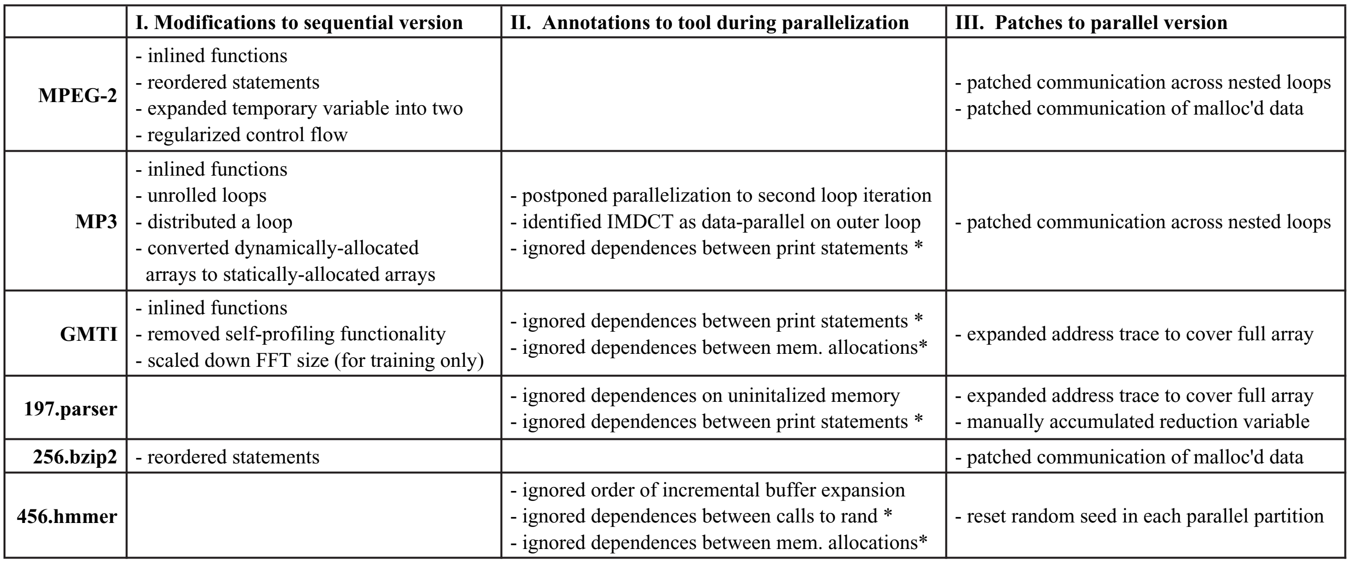
During the iterative parallelization process, the programmer made three adjustments to the program. Our tool reported a number of loop-carried dependences due to the program's implicit use of uninitialized memory locations; the program allocates space for a struct and later copies the struct (by value) before all of the elements have been initialized. This causes our tool to report a dependence on the previous write to the uninitialized locations, even though such writes were modifying a different data structure that has since been de-allocated. The programmer eliminated these dependence reports by initializing all elements to a dummy value at the time of allocation.

The programmer also made two adjustments to the communication trace emitted by our tool. One block of addresses was expanding gradually over the first few iterations of the program. Closer inspection revealed that that sentences of increasing length were being passed between partitions. The programmer patched the trace to always communicate the complete sentence buffer. Also, the programmer observed that in the case of errors, the parser's error count needs to be communicated to the output stage and accumulated there. As none of our training or testing samples elicited errors, our trace did not detect this dependence.

Our data-parallel version of the program may reorder the program's print statements. If desired, the print statements can be serialized by moving them to the output stage.

Bzip2 The stream graphs for 256.bzip2 appear in Figures 6-11b and 6-11c. The benchmark includes both a compression and decompression stage, which were parallelized separately.

Because bzip2 compresses blocks of fixed size, the main compression routine is completely data-parallel. The only cyclic dependences in the compressor are at the input stage (file reading, CRC calculation) and output stage (file writing). The programmer replicated the compression stage seven ways to match the four-core machine; this allows three cores to handle two compression



	I. Modifications to sequential version	II. Annotations to tool during parallelization	III. Patches to parallel version
MPEG-2	- inlined functions - reordered statements - expanded temporary variable into two - regularized control flow		- patched communication across nested loops - patched communication of malloc'd data
MP3	- inlined functions - unrolled loops - distributed a loop - converted dynamically-allocated arrays to statically-allocated arrays	- postponed parallelization to second loop iteration - identified IMDCT as data-parallel on outer loop - ignored dependences between print statements *	- patched communication across nested loops
GMTI	- inlined functions - removed self-profiling functionality - scaled down FFT size (for training only)	- ignored dependences between print statements * - ignored dependences between mem. allocations*	- expanded address trace to cover full array
197.parser		- ignored dependences on uninitalized memory - ignored dependences between print statements *	- expanded address trace to cover full array - manually accumulated reduction variable
256.bzip2	- reordered statements		- patched communication of malloc'd data
456.hmmer		- ignored order of incremental buffer expansion - ignored dependences between calls to rand * - ignored dependences between mem. allocations*	- reset random seed in each parallel partition

Figure 6-12: Steps taken by the programmer to assist in parallelizing each benchmark. Assistance may be needed to expose parallelism in the original code, to verify parallelism using the tool, or to handle special cases in the parallelized code. Steps annotated with an asterisk (*) may change the observable behavior of the program[1].

stages each, while one core handles a single compression stage as well as the input and output stages. The decompression step lacks data-parallelism because the boundaries of the compressed blocks are unknown; however, it can be split into a pipeline of two stages.

In parallelizing bzip2, the programmer reordered some statements to improve the pipeline partitioning (the call to `generateMTFValues` moved from the output stage to the compute stage). The programmer also supplied the name and size of two dynamically-allocated arrays.

Hmmer In 456.hmmer, a Hidden Markov Model is loaded at initialization time, and then a series of random sequences are used to calibrate the model. Figure 6-11d shows the extracted stream graph for this benchmark. The calibration is completely data-parallel except for a histogram at the end of the loop, which must be handled with pipeline parallelism. In our experiments, the programmer replicated the data-parallel stage four ways to utilize the four-core machine.

Our tool reports three parallelism-limiting dependences for hmmer. The first is due to random number generation: each iteration generates a new random sample and modifies the random seed. The programmer chose to ignore this dependence, causing the output of our parallel version to differ from the original version by 0.01%. Also, the programmer made an important patch to the parallel code: after forking from the original process, each parallel partition needs to set its random seed to a different value. Otherwise each partition would follow an identical sequence of random values, and the parallel program would sample only a fraction of the input space as the original program.

The second problematic dependence is due to an incremental resizing of an array to fit the

[1] Reordering calls to malloc (or reordering calls to free) will only change the program's behavior if one of the calls fails.

Benchmark	**Pipeline Depths**	**Data-Parallel Widths**	**Speedup**
GMTI	9	—	3.03x
MPEG-2	7	—	2.03x
MP3	6	2,2	2.48x
197.parser	3	4	2.95x
256.bzip2	3,2	7	2.66x
456.hmmer	2	4	3.89x
GeoMean			**2.78x**

Table 6-13: Characteristics of the parallel stream graphs and performance results on a 4-core machine. Data-parallel width refers to the number of ways any data-parallel stage was replicated.

length of the input sequence. Since each parallel partition can expand its own private array, this dependence is safely ignored. Finally, as in the case of GMTI, dependences between memory allocation functions were relaxed for the sake of the parallelization.

Performance Results

Following parallelization with our tool, all of the benchmarks obtain the correct results on their training and testing sets. For MPEG-2 and MP3, we train using five iterations of input files 1 and 10, respectively (see Section 6.2). For GMTI, we only have access to a single input trace, so we use five iterations for training and the rest (300 iterations) for testing. For the SPEC benchmarks, we train on five iterations of the provided training set and test on the provided testing set.

Our evaluation platform contains two AMD Opteron 270 dual-core processors (for a total of 4 cores) with 1 MB L2 cache per processor and 8 GB of RAM. We measure the speedup of the parallel version, which uses up to 4 cores, versus the original sequential version, which uses 1 core. We generate one process per stage of the stream graph, and rely on the operating system to distribute the processes across cores (we do not provide an explicit mapping from threads to cores). All speedups reflect total (wall clock) execution time.

Our performance results appear in Table 6-13. Speedups range from 2.03x (MPEG-2) to 3.89x (hmmer), with a geometric mean of 2.78x. While these results are good, there is some room for improvement. Some benchmarks (MPEG-2, decompression stage of bzip2) suffer from load imbalance that is difficult to amend without rewriting parts of the program. The imperfect speedups in other benchmarks may reflect synchronization overheads between threads, as the operating system would need to interleave executions in a specific ratio to avoid excessive blocking in any one process. The volume of communication does not appear to be a significant bottleneck; for example, duplicating all communication instructions in MP3 results in only a 1.07x slowdown. Ongoing work will focus on improving the runtime scheduling of the processes, as well as exploring other inter-process communication mechanisms (e.g., using shared memory).

6.6 Related Work

Static Analysis

The work most closely related to ours is that of Bridges et al. [BVZ$^+$07], which was developed concurrently to our first publication of this research [TCA07]. They exploit pipeline parallelism using the techniques of Decoupled Software Pipelining [RVVA04, ORSA05]. In addition, they employ thread-level speculation to speculatively execute multiple loop iterations in parallel. Both of our systems require some assistance from the programmer in parallelizing legacy applications. Whereas we annotate spurious dependences within our tool, they annotate the original source code with a new function modifier (called "commutative") to indicate that successive calls to the function can be freely reordered. Such source-level annotations are attractive (e.g., for malloc/free) and could be integrated with our approach. However, our transformations rely on a different property of these functions, as we call them in parallel from isolated address spaces rather than reordering the calls in a single address space.

Once parallelism has been exposed, their compiler automatically places the pipeline boundaries and generates a parallel runtime, whereas we rely on the programmer to place pipeline boundaries and to provide some assistance in generating the parallel version (see Section 6.3). Our approaches arrive at equivalent decompositions of 197.parser and 256.bzip2. However, our runtime systems differ. Rather than forking multiple processes that communicate via pipes, they rely on a proposed "versioned memory" system [VRR$^+$07] that maintains multiple versions of each memory location. This allows threads to communicate via shared memory, with the version history serving as buffers between threads. Their evaluation platform also includes a specialized hardware construct termed the synchronization array [RVVA04]. In comparison, our technique runs on commodity hardware.

Dai et al. presents an algorithm for automatically partitioning sequential packet-processing applications for pipeline-parallel execution on network processors [DHLH05]. Their static analysis targets fine-grained instruction sequences within a single procedure, while our dynamic analysis is coarse-grained and inter-procedural. Du et al. describes a system for pipeline-parallel execution of Java programs [DFA05]. The programmer declares parallel regions, while the compiler automatically places pipeline boundaries and infers the communicated variables using an inter-procedural static analysis. Unlike our system, the compiler does not check if the declared regions are actually parallel.

Dynamic Analysis

The dynamic analysis most similar to ours is that of Rul et al. [RVDB06], which also tracks producer/consumer relationships between functions and uses the information gleaned to assist the programmer in parallelizing the program. They use bzip2 as a case study and report speedups comparable to ours. However, it appears that their system requires the programmer to determine which variables should be communicated between threads and to modify the original program to insert new buffers and coordinate thread synchronization.

Karkowski and Corporaal also utilize dynamic information to uncover precise dependences for parallelization of C programs [KC97]. Their runtime system utilizes a data-parallel mapping rather than a pipeline-parallel mapping, and they place less emphasis on the programmer interface and visualization tools.

Redux is a tool that traces instruction-level producer/consumer relationships for program comprehension and debugging [NM03]. Unlike our tool, Redux tracks dataflow through registers in addition to memory locations. (We avoid the need for such tracking by profiling an unoptimized binary, generated with gcc -O0, that stores all intermediate values to memory.) Because it generates a distinct graph node for every value produced, the authors note that the visualization becomes unwieldy and does not scale to realistic programs. We address this issue by coarsening the program partitions.

A style of parallelism that is closely related to pipeline parallelism is DOACROSS parallelism [PKL80, Cyt86]. Rather than devoting a processor to a single pipeline stage, DOACROSS parallelism assigns a processor to execute complete loop iterations, spanning all of the stages. In order to support dependences between iterations, communication is inserted at pipeline boundaries to pass the loop-carried state between processors. While DOACROSS parallelism has been exploited dynamically using inspector/executor models (see Rauchwerger [Rau98] for a survey), they lack the generality needed for arbitrary C programs. The parallelism and communication patterns inferred by our tool could be used to generate a DOACROSS-style mapping; such a mapping could offer improved load balancing, at the possible expense of degrading instruction locality and adding communication latency to the critical path.

Giacomoni et al. describe a toolchain for pipeline-parallel programming [GMP$^+$07], including BDD-based compression of dependence traces [PV06]. Such techniques could extend our stream graph visualization to a much finer granularity. DDgraph is a dynamic analysis for Lisp that offers a visualization of call graphs and data dependence graphs for the sake of program understanding and correctness checking [BWCA05]. It is implemented as part of a Lisp interpreter and has been applied to an AI Blocks World program, which exhibits less regular streams of data than our target applications. Malton and Pahelvan also use a dynamic analysis (built on gdb) to identify control flow between "pivotal functions" that are likely to aid in program understanding [MP05]. They do not extract streams of data flow.

Program slicing is a technique that aims to identify the set of program statements that may influence a given statement in the program. Slicing is a rich research area with many static and dynamic approaches developed to date; see Tip [Tip95] for a review. The problem that we consider is more coarse-grained than slicing; we divide the program into partitions and ask which partitions affect a given partition. Also, we identify a list of memory locations that are sufficient to convey all the information needed between partitions. Finally, we are interested only in direct dependences between partitions, rather than the transitive dependences reported by slicing tools.

6.7 Future Work

There are rich opportunities for future work in enhancing the soundness and automation of our tool. If the runtime system encounters code that was not visited during training, it could execute the corresponding loop iteration in a sequential manner (such a policy would have fixed the only unsoundness we observed). A static analysis could also lessen the programmer's involvement, e.g., by automatically handling nested loops or automatically placing the pipeline partitions. Many of the optimizations implemented in StreamIt could be targeted to the extracted stream graphs, as they follow the synchronous dataflow model. It could also be interesting to develop systematic testing techniques to exhibit control flow paths that were not covered during training.

More broadly, the observations in Section 6.5 suggest that many of the memory dependences that constrain automatic parallelizers can be safely ignored without affecting the ultimate program outcome. It would be interesting to build a testing tool that explores this opportunity more deeply, perhaps by systematically violating each individual memory dependence in the program and reporting those that do not affect the program outcome. While such an analysis would be very slow if only one dependence is broken per run, perhaps an optimized version can be built by speculatively "pooling" many tests into a single run, or by detecting that an intermediate program state is correct without finishing the complete execution. Testing tools could also be useful for inferring likely high-level properties, such as commutativity between methods in a library. This would save the user the trouble of indicating the potential for such reordering to our tool.

6.8 Chapter Summary

This work represents one of the first systematic techniques to extract a coarse-grained streaming representation from C programs. Rather than extracting streams from small instruction sequences or inner loops, we extract pipeline stages from the outermost toplevel loop of a streaming application, which encapsulates 100% of the steady-state runtime. Our approach is applicable both to legacy codes, in which the user has little or no knowledge about the structure of the program, as well as new applications, in which programmers can utilize our annotations to easily express the desired pipelining.

The key observation underlying our technique is that for the domain of streaming applications, the steady-state communication pattern is regular and stable, even if the program is written in a language such as C that resists static analysis. To exploit this pattern, we employ a dynamic analysis to trace the memory locations communicated between program partitions at runtime. Partition boundaries are defined by the programmer using a simple set of annotations; the partitions can be iteratively refined to improve the parallelism and load balance. Our tool uses the communication trace to construct a stream graph for the application as well as a detailed list of producer-consumer instruction pairs, both of which aid program understanding and help to track down any problematic dependences. Our dynamic analysis tool also outputs a set of macros to automatically parallelize the program and communicate the needed data between partitions. We applied our tool to six realistic case studies; the parallel programs produced the correct output and offered a mean speedup of 2.78x on a 4-core machine.

Our technique gains both its leverage and its liabilities from the fact that it is unsound. By tracing the dynamic flow of data, it can detect communication patterns that are beyond the reach of static analyses. However, because the analysis samples only a fraction of program executions, the observed communication may offer an incomplete picture of the behavior of other runs. The unsound data remains useful for program understanding, as it helps programmers understand the common-case behavior of the application. In the context of automatic parallelization, the unsoundness is more problematic because untested executions may lead to incorrect results when parallelized.

This risk can be minimized by employing a rigorous quality assurance (QA) policy, in which the tool observes all distinct modes of execution prior to parallelization. Such QA practices are already deeply ingrained in the industry. Given that industrial testing strategies have proven effective for eliminating a myriad of human-induced bugs, it does not seem implausible that they could be applied to eliminate potential bugs introduced by our tool. Unlike a programmer, our analysis can automatically fix any bugs found by including the buggy input in its training set. Perhaps by treating tools more like programmers – as intelligent but unsound beings whose output must be subjected to rigorous testing – we can overcome some of the traditional limitations of automatic parallelization.

Chapter 7

Conclusions

My thesis is that incorporating streaming abstractions into the programming language can simultaneously improve both programmability and performance. Programmers are unburdened from providing low-level implementation details, while compilers can perform parallelization and optimization tasks that were previously beyond the reach of automation. This dissertation supports this thesis with the following contributions:

1. We define the StreamIt language, one of the first programming languages to embrace synchronous dataflow as a model of computation. StreamIt contains novel language constructs, including structured streams, parameterized data reordering, and teleport messaging, that improve both the programmability and the analyzability of stream programs. Teleport messaging addresses a long-standing problem in synchronizing events across decoupled modules, and represents the first general framework for delivering messages with respect to the regular dataflow in the stream. By providing a modular and composable syntax, the StreamIt language becomes accessible to non-expert programmers. Simultaneously, the language preserves the rich static properties of the streaming domain, exposing them to the compiler for the sake of optimization.

2. We demonstrate that it is tractable to develop large-scale applications in a stream programming model. Our experience comes on two fronts. First, we describe the development of the 34,000-line StreamIt benchmark suite, consisting of large applications (such as MPEG-2 encoding/decoding and GMTI radar processing) that were written by programmers who were previously unfamiliar with StreamIt. Our survey of these benchmarks provides the first rigorous characterization of the streaming domain, as well as new insights into the utility and usability of various language features. Second, we develop a tool for migrating legacy C programs into a streaming representation. It is the first tool to use a dynamic analysis to expose coarse-grained parallelism in C programs. We show that this tool is effective at extracting a synchronous dataflow graph from large C applications, spanning MPEG-2, GMTI, MP3, and others.

3. We develop a new optimization for the streaming domain, allowing programmers to accelerate common video editing operations by a median of 15x and a maximum of 471x. This transformation maps stream programs into the compressed domain, allowing them to operate directly on compressed data formats rather than requiring a costly decompression and re-compression on either side of processing. Our technique is the first to support compressed-domain processing of LZ77-compressed data. We apply our technique to accelerate transformations such as

color adjustment and video compositing on the Apple Animation format. Performance gains are proportional to the compression factor.

4. We review the key optimization results in the StreamIt project, enabling programmers to obtain large speedups on many tasks. Targeting a 16-core architecture, our compiler leverages a novel combination of task, data, and pipeline parallelism to obtain a robust speedup of over 11x (relative to a single core). In optimizing linear computations, our compiler mirrors the behavior of a DSP expert, automatically combining linear nodes, translating them to the frequency domain, and selecting the most profitable series of transformations. Linear optimizations yield an average performance improvement of 5.5x, and a maximum improvement of 8.0x. Finally, we offer a set of cache optimizations that adjusts the schedule of filter executions so as to improve the data and instruction locality. It offers an average benefit of 3.5x when targeting an embedded processor.

Several of the transformations that are automated in the StreamIt compiler are already accessible to expert programmers. For example, the optimization of linear nodes is a standard part of the DSP design flow. Cache optimizations similar to ours are routinely performed during the manual tuning of an embedded system. Streaming applications can be parallelized in other languages with significant help from the programmer. However, the key benefit of StreamIt is that all of these transformations become accessible to non-experts. Using a stream programming model, the compiler can leverage new information to automate transformations that were previously reserved for technology wizards.

Perhaps the biggest limitation of the techniques described in this dissertation is that they apply primarily to static-rate programs, in which the input and output rates of actors are known at compile time. The compiler depends on static rates for load-balancing task- and pipeline-parallel actors; for optimizing linear filters (which are implicitly static-rate); for cache optimizations; and for teleport messaging (though Section 3.6 describes how to extend messaging to handle dynamism). Some of the techniques described do extend to dynamic rates, including the language's support for hierarchical streams and parameterized data reordering, as well as the compiler's support for coarse-grained data parallelism, translation to the compressed domain (with minor modifications), and our dynamic analysis for extracting parallelism from C programs.

In the long term, we envision that our optimizations of static-rate graphs would have maximum impact when those graphs are embedded in a more flexible and hybrid programming model. By analogy to instruction scheduling, one could consider our optimizations as focusing on the basic block: a simple yet pervasive construct that can be stitched together to perform a broad array of tasks. While support for complex control flow is important for functionality, aggressive optimization within the basic block is essential for high performance. Our benchmark suite supports the premise that dynamic rates often occur at only a few points in an application; out of our 29 most realistic applications, 24 are completely static-rate, and out of those with any dynamic rates, only 3% of the user-defined filters have a dynamic rate. Thus, the compiler can focus on optimizing each static-rate subgraph, while relying on the runtime system to orchestrate the dynamic-rate boundaries. In addition to supporting dynamism, we envision that a hybrid programming model would provide support for many models of computation (transactions, event-driven programs, scientific codes, etc.) with high-performance streaming playing only one part. These models could likely be embedded in a general-purpose programming language, using a separate library and runtime system for each one. Integrating such models of computation into a unified authoring environment is an interesting direction for further research.

What is the future of stream programming? Well, the goal of academic programming language research is frequently misunderstood. While it would not be objectionable to have a language like StreamIt take over the world, this goal is rarely realistic or sustainable. Rather, the primary goal of our research is to influence the direction of future languages. There is broad precedent for such influence; for example, Bjarne Stroustrup traces the detailed evolution of the C++ language, tracing the impact of many previous languages on its structure and feature set [Str94]. While many of these languages are also well known (Fortran, C, Ada, Simula), there are also important influences from lesser-known languages, many of which are of academic origin (CPL, BCPL, ML, Clu). Given the trend towards multicore processors, and the increasing prevalence of streaming applications, we anticipate the emergence of languages and libraries with integrated support for efficient stream processing. Already the StreamIt system (which is open-source and available online [Stra]) has been reused and extended by multiple research groups, including UC Berkeley [NY04, SLRBE05], IBM Research [HHBR08, HKM+08], University of Michigan [KM08, HKM+08], Halmstad University [JSuA05, And07], Johns Hopkins University [Duc04], and North Carolina State University [So07]. It is our hope that the abstractions, optimizations, and lessons learned as part of this dissertation will serve to inform and inspire this research as well as the upcoming generation of mainstream programming languages.

Bibliography

[AB71] J. Achermann and R. Bucy, *Canonical minimal realization of a matrix of impulse response sequences*, Information and Control (1971), 224–231.

[ABC+06] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, *The landscape of parallel computing research: A view from Berkeley*, Tech. Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[AD79] W. Ackerman and J. Dennis, *VAL – A value-oriented algorithmic language*, Tech. Report MIT-LCS-TR-218, Masscahusetts Institute of Technology, 1979.

[Ado06] *About digital video editing*, Adobe online education materials, 2006, `http://www.adobe.com/education/pdf/cib/pre65_cib/pre65_cib02.pdf`.

[Agh85] G. Agha, *Actors: A model of concurrent computation in distributed systems*, Ph.D. Thesis, Massachusetts Insitute of Technology, 1985.

[AGK+05] S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies, *Language and compiler design for streaming applications*, International Journal of Parallel Programming, 2005.

[AGP78] Arvind, K. Gostelow, and W. Plouffe, *An asynchronous programming language and computing machine*, Tech. Report TR 114a, University of California, Irvine, 1978.

[Agr04] S. Agrawal, *Linear state-space analysis and optimization of StreamIt programs*, M.Eng. Thesis, Massachusetts Institute of Technology, 2004.

[AK82] J. Allen and K. Kennedy, *PFC: A program to convert Fortran to parallel form*, IBM Conference on Parallel Computing and Scientific Computations (1982).

[ALP97] M. Adé, R. Lauwereins, and J. A. Peperstraete, *Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets*, Design Automation Conference (DAC), 1997.

[And07] J. Andersson, *Modelling and evaluating the StreamBits language*, Master's thesis, Halmstad University, 2007.

[Arm07] J. Armstrong, *A history of Erlang*, Conference on History of Programming Languages (HOPL), ACM, 2007.

[AS98] S. Acharya and B. Smith, *Compressed domain transcoding of MPEG*, International Conference on Multimedia Computing and Systems (1998), 295–304.

[ATA05] S. Agrawal, W. Thies, and S. Amarasinghe, *Optimizing stream programs using linear state space analysis*, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2005.

[Aut] *Authentication generator demo*, Online screencast, `http://penso.info/auth_generator`.

[AVW93] J. Armstrong, R. Virding, and M. Williams, *Concurrent programming in Erlang*, Prentice Hall, 1993.

[AW77] E. A. Ashcroft and W. W. Wadge, *Lucid, a nonprocedural language with iteration*, Communications of the ACM **20** (1977), no. 7, 519–526.

[Azi07] A. Aziz, *Image-based motion estimation in a stream programming language*, M.Eng. Thesis, Massachusetts Institute of Technology, 2007.

[BB00] B. Bhattacharya and S. S. Bhattacharyya, *Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems*, International Workshop on Rapid System Prototyping (RSP), 2000, p. 84.

[BCG$^+$03] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski, *Streaming XPath processing with orward and backward axes*, International Conference on Data Engineering, 2003.

[BDB99] V. Bala, E. Duesterwald, and S. Banerjia, *Dynamo: A transparent dynamic optimization system*, Conference on Programming Language Design and Implementation (PLDI), 1999.

[BELP95] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, *Cyclo-static data flow*, International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 1995.

[Ben] S. Benza, *Interview transcript*, Computer Graphics Society, `http://forums.cgsociety.org/showthread.php?s=&threadid=115293`.

[BFH$^+$04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, *Brook for GPUs: Stream computing on graphics hardware*, SIGGRAPH, 2004.

[BG92] G. Berry and G. Gonthier, *The ESTEREL synchronous programming language: Design, semantics, implementation*, Science of Computer Programming **19** (1992), no. 2, 87–152.

[BG99] S. Bakshi and D. D. Gajski, *Partitioning and pipelining for performance-constrained hardware/software systems*, IEEE Transactions on Very Large Scale Integrated Systems **7** (1999), no. 4, 419–432.

[BHLM91] J. Buck, S. Ha, E. Lee, and D. Messerschmitt, *Multirate signal processing in Ptolemy*, International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 1991.

[Ble06a] *Blender*, Wikipedia, The Free Encyclopedia, November 2006, `http://en.wikipedia.org/wiki/Blender_software`.

[Ble06b] *Blender.org: website statistics*, Blender Foundation, 2006.

[BML95] S. Bhattacharyya, P. Murthy, and E. Lee, *Optimal parenthesization of lexical orderings for DSP block diagrams*, International Workshop on VLSI Signal Processing, 1995.

[BML96] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software synthesis from dataflow graphs*, Kluwer Academic Publishers, 1996.

[BML97] C. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations*, Journal of Design Automation for Embedded Systems (1997), 33–60.

[BML99] S. Bhattacharyya, P. Murthy, and E. Lee, *Synthesis of embedded software from synchronous dataflow specifications*, Journal of VLSI Signal Processing Systems **21** (1999), no. 2.

[BSL96] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, *Self-timed resynchronization: A post-optimization for static multiprocessor schedules*, International Parallel Processing Symposium, 1996, pp. 199–205.

[BV00] J. Buck and R. Vaidyanathan, *Heterogeneous modeling and simulation of embedded systems in El Greco*, International Workshop on Hardware/Software Codesign (CODES), 2000.

[BVZ$^+$07] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August, *Revisiting the sequential programming model for multi-core*, MICRO, 2007.

[BWCA05] F. Balmas, H. Wertz, R. Chaabane, and L. Artificielle, *DDgraph: A tool to visualize dynamic dependences*, Workshop on Program Comprehension through Dynamic Analysis, 2005.

[CBL01] N. Chandrachoodan, S. Bhattacharyaa, and K. Liu, *An efficient timing model for hardware implementation of multirate dataflow graphs*, International Conference on Acoustics, Speech, and Signal Processing (ICASSP), vol. 2, 2001.

[CCD$^+$08] Y.-K. Chen, J. Chhugani, P. Dubey, C. Hughes, D. Kim, S. Kumar, V. Lee, A. Nguyen, and M. Smelyanskiy, *Convergence of recognition, mining, and synthesis workloads and its implications*, Proceedings of the IEEE **96** (2008), no. 5, 790–807.

[CGT$^+$05] J. Chen, M. I. Gordon, W. Thies, M. Zwicker, K. Pulli, and F. Durand, *A reconfigurable architecture for load-balanced rendering*, SIGGRAPH / Eurographics Workshop on Graphics Hardware, 2005.

[CH78] P. Cousot and N. Halbwachs, *Automatic discovery of linear restraints among variables of a program*, Symposium on Principles of Programming Languages (POPL), 1978, pp. 84–97.

[Cha95] S. Chang, *Compressed-domain techniques for image/video indexing and manipulation*, International Conference on Image Processing (1995).

[Che05] J. Chen, *Load-balanced rendering on a general-purpose tiled architecture*, M.Eng. Thesis, Massachusetts Insitute of Technology, 2005.

[CHP07] P. Caspi, G. Hamon, and M. Pouzet, *Real-time systems: Models and verification – Theory and tools*, ch. Synchronous Functional Programming with Lucid Synchrone, ISTE, 2007.

[CHR$^+$03] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu, *Spidle: A DSL approach to specifying streaming applications*, International Conference on Generative Programming and Component Engineering, 2003.

[Cli81] W. D. Clinger, *Foundations of actor semantics*, Ph.D. Thesis, Massachusetts Insitute of Technology, 1981.

[Cov89] M. M. Covell, *An algorithm design environment for signal processing*, Ph.D. Thesis, MIT, 1989.

[CP95] P. Caspi and M. Pouzet, *A functional extension to Lustre*, International Symposium on Languages for Intentional Programming, 1995.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, *LUSTRE: A declarative language for real-time programming*, International Symposium on Principles of Programming Languages (POPL), 1987.

[CS97] L.-F. Chao and E. H.-M. Sha, *Scheduling data-flow graphs via retiming and unfolding*, IEEE Transactions on Parallel and Distributed Systems **08** (1997), no. 12, 1259–1267.

[CV02] K. S. Chatha and R. Vemuri, *Hardware-software partitioning and pipelined scheduling of transformative applications*, IEEE Transactions on Very Large Scale Integrated Systems **10** (2002), no. 3, 193–208.

[Cyt86] R. Cytron, *DOACROSS: Beyond vectorization for multiprocessors*, International Conference on Parallel Processing, 1986.

[DA01] R. Dugad and N. Ahuja, *A fast scheme for image size change in the compressed domain*, IEEE Transactions on Circuits and Systems for Video Technology **11** (2001), no. 4, 461–474.

[DBH$^+$01] B. A. Draper, A. P. W. Böhm, J. Hammes, W. A. Najjar, J. R. Beveridge, C. Ross, M. Chawathe, M. Desai, and J. Bins, *Compiling SA-C programs to FPGAs: Performance results*, International Workshop on Computer Vision Systems, 2001.

[DDE$^+$05] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick, *Self adapting linear algebra algorithms and software*, Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation **93** (2005), no. 2.

[DFA05] W. Du, R. Ferreira, and G. Agrawal, *Compiler support for exploiting coarse-grained pipelined parallelism*, International Conference on Supercomputing, 2005.

[DHLH05] J. Dai, B. Huang, L. Li, and L. Harrison, *Automatically partitioning packet processing applications for pipelined architectures*, Conference on Programming Language Design and Implementation (PLDI), 2005.

[Dig06] *Digital Juice, Editor's toolkit 4: High tech tools*, Digital Juice, 2006, `http://www.digitaljuice.com/products/products.asp?pid=119`.

[Dij68] E. W. Dijkstra, *Go to statement considered harmful*, Communications of the ACM **11** (1968), no. 3, 147–148.

[DM05] G. Dolbier and V. Megler, *Building an animation and special effects studio from the ground up*, IBM Report, 2005.

[Dra06] M. Drake, *Stream programming for image and video compression*, M.Eng. Thesis, Massachusetts Institute of Technology, 2006.

[DRB00] C. Dorai, N. Ratha, and R. Bolle, *Detecting dynamic behavior in compressed fingerprint videos: Distortion*, Conference on Computer Vision and Pattern Recognition (CVPR), 2000.

[Duc04] N. Duca, *Applications and execution of stream graphs*, Senior Undergraduate Thesis, Johns Hopkins University, 2004.

[EGK$^+$02] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull, *Graphviz – Open source graph drawing tools*, Graph Drawing (2002), 594–597.

[EJL$^+$03] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, *Taming heterogeneity – the Ptolemy approach*, Proceedings of the IEEE **91** (2003), no. 1, 127–144.

[Ele] *Elephant's Dream*, `http://orange.blender.org/`.

[EM87] E. and D. Messerschmitt, *Pipeline interleaved programmable DSP's: Synchronous data flow programming*, IEEE Transactions on Signal Processing **35** (1987), no. 9.

[FG05] K. Fisher and R. Gruber, *PADS: A domain-specific language for processing ad hoc data*, Conference on Programming Language Design and Implementation (PLDI), 2005.

[FJ03] G. Feng and J. Jiang, *Image segmentation in compressed domain*, Journal of Electronic Imaging **12** (2003), no. 3, 390–397.

[FJ05] M. Frigo and S. G. Johnson, *The design and implementation of FFTW3*, Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation **93** (2005), no. 2.

[Fra03a] M. Frank, *SUDS: Automatic parallelization for Raw processors*, Ph.D. Thesis, MIT, 2003.

[Fra03b] Fraunhofer Institute, *MP3 reference implementation*, `http://www.mpeg1.de/util/dos/mpeg1iis/`, 2003.

[FT98] M. Farach and M. Thorup, *String matching in Lempel-Ziv compressed strings*, Algorithmica **20** (1998).

[GBBG86] P. L. Guernic, A. Benveniste, P. Bournai, and T. Gautier, *Signal – A data flow-oriented language for signal processing*, IEEE Transactions on Acoustics, Speech and Signal Processing **34** (1986), no. 2, 362–374.

[GBS05] M. Geilen, T. Basten, and S. Stuijk, *Minimising buffer requirements of synchronous dataflow graphs with model checking*, Design Automation Conference (DAC) (Anaheim, California, USA), ACM, 2005, pp. 819–824.

[GGD94] R. Govindarajan, G. Gao, and P. Desai, *Minimizing memory requirements in rate-optimal schedules*, International Conference on Application Specific Array Processors (ASAP), 1994, pp. 75–86.

[GGD02] R. Govindarajan, G. R. Gao, and P. Desai, *Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks*, Journal of VLSI Signal Processing **31** (2002), no. 3, 207–229.

[GMP+07] J. Giacomoni, T. Moseley, G. Price, B. Bushnell, M. Vachharajani, and D. Grunwald, *Toward a toolchain for pipeline parallel programming on CMPs*, Workshop on Software Tools for Multi-Core Systems (STMCS), 2007.

[Gor02] M. Gordon, *A stream-aware compiler for communication-exposed architectures*, S.M. Thesis, Massachusetts Institute of Technology, 2002.

[GR05] J. Gummaraju and M. Rosenblum, *Stream programming on general-purpose processors*, MICRO, 2005.

[Gre75] I. Greif, *Semantics of communicating parallel processes*, Ph.D. Thesis, Massachusetts Insitute of Technology, 1975.

[GTA06] M. I. Gordon, W. Thies, and S. Amarasinghe, *Exploiting coarse-grained task, data, pipeline parallelism in stream programs*, International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2006.

[GTK+02] 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*, International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2002.

[Hal98] N. Halbwachs, *Synchronous programming of reactive systems*, International Conference on Computer Aided Verification, Springer-Verlag, 1998, pp. 1–16.

[HBS73] C. Hewitt, P. Bishop, and R. Steiger, *A universal modular ACTOR formalism for artificial intelligence*, International Joint Conferences on Artificial Intelligence (IJCAI), 1973.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, *The synchronous data flow programming language LUSTRE*, Proceedings of the IEEE **79** (1991), no. 9, 1305–1320.

[HH06] C. Howard and P. Hoke, *Legacy migrations: Turning legacy systems into gold–the planning process*, T&I Supplement (2006), `http://www.ateras.com/PDF/bij_Article.pdf`.

[HHBR08] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah, *Liquid Metal: Object-oriented programming across the hardware/software boundary*, European Conference on Object-Oriented Programming (ECOOP) (Paphos, Cypress), Springer-Verlag, 2008, pp. 76–103.

[HKM+08] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, *Optimus: Efficient realization of streaming applications on FPGAs*, International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES) (Atlanta, GA, USA), ACM, 2008, pp. 41–50.

[HL97] S. Ha and E. A. Lee, *Compile-time scheduling of dynamic constructs in dataflow program graphs*, IEEE Transactions on Computers **6** (1997).

[HM76] P. Henderson and J. H. Morris, *A lazy evaluator*, Symposium on Principles of Programming Languages (POPL), 1976.

[HMG04] R. Harrington, R. Max, and M. Geduld, *After effects on the spot: Time-saving tips and shortcuts from the pros*, Focal Press, 2004.

[HMWZ92] J. C. Huang, J. Muñoz, H. Watt, and G. Zvara, *ECOS graphs: A dataflow programming language*, Symposium on Applied Computing, 1992.

[Hoa78] C. A. R. Hoare, *Communicating sequential processes*, Communications of the ACM **21** (1978), no. 8, 666–677.

[HP02] H. Harada and R. Prasad, *Simulation and software radio for mobile communications*, Artech House, 2002.

[HRB88] S. Horwitz, T. Reps, and D. Binkley, *Interprocedural slicing using dependence graphs*, Conference on Programming Language Design and Implementation (PLDI), 1988.

[Inm88] Inmos Corporation, *Occam 2 reference manual*, Prentice Hall, 1988.

[IT88] F. Irigoin and R. Triolet, *Supernode partitioning*, Symposium on Principles of Programming Languages (POPL), 1988.

[IYV04] E. Im, K. A. Yelick, and R. Vuduc, *SPARSITY: An optimization framework for sparse matrix kernels*, International Journal of High Performance Computing Applications **18** (2004), no. 1, 135–158.

[JCM94] M. Janssen, F. Catthoor, and H. D. Man, *A specification invariant technique for operation cost minimisation in flow-graphs*, International Symposium on High-level Synthesis, 1994, pp. 146–151.

[JHM04] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, *Advances in dataflow programming languages*, ACM Compututing Surveys **36** (2004), no. 1, 1–34.

[JSuA05] O. Johnsson, M. Stenemo, and Z. ul Abdin, *Programming and implementation of streaming applications*, Tech. Report IDE0405, Halmstad University, 2005.

[KA99] Y.-K. Kwok and I. Ahmad, *Static scheduling algorithms for allocating directed task graphs to multiprocessors*, ACM Computing Surveys **31** (1999), no. 4, 406–471.

[Kah74] G. Kahn, *The semantics of a simple language for parallel programming*, Information Processing (1974), 471–475.

[Kar76] M. Karr, *Affine relationships among variables of a program*, Acta Informatica **6** (1976), 133–155.

[Kar02] M. A. Karczmarek, *Constrained and phased scheduling of synchronous data flow graphs for the StreamIt language*, Master's thesis, Massachusetts Institute of Technology, 2002.

[KC97] I. Karkowski and H. Corporaal, *Overcoming the limitations of the traditional loop parallelization*, HPCN Europe, 1997.

[KM66] R. M. Karp and R. E. Miller, *Properties of a model for parallel computations: Determinacy, termination, queueing*, SIAM Journal on Applied Mathematics **14** (1966), no. 6, 1390–1411.

[KM08] M. Kudlur and S. Mahlke, *Orchestrating the execution of stream programs on multicore platforms*, Conference on Programming Language Design and Implementation (PLDI), 2008.

[Koh04] S. Kohli, *Cache aware scheduling of synchronous dataflow programs*, Master's Report Technical Memorandum UCB/URL M04/03, UC Berkeley, 2004.

[KRA05] K. Kuo, R. Rabbah, and S. Amarasinghe, *A productive programming environment for stream computing*, Workshop on Productivity and Performance in High-End Computing, 2005.

[KRD$^+$03] 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).

[KSB06] M.-Y. Ko, C.-C. Shen, and S. S. Bhattacharyya, *Memory-constrained block processing optimization for synthesis of DSP software*, International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, 2006, pp. 137–143.

[KTA03] M. Karczmarek, W. Thies, and S. Amarasinghe, *Phased scheduling of stream programs*, Conference on Languages, Compilers, Tools for Embedded Systems (LCTES), 2003.

[Kuo04] K. Kuo, *The StreamIt development tool: A programming environment for StreamIt*, M.Eng. Thesis, Massachusetts Institute of Technology, 2004.

[Lam03] A. A. Lamb, *Linear analysis and optimization of stream programs*, M.Eng. Thesis, Massachusetts Institute of Technology, 2003.

[LDWL06] S. Liao, Z. Du, G. Wu, and G. Lueh, *Data and computation transformations for Brook streaming applications on multiprocessors*, International Symposium on Code Generation and Optimization (CGO), 2006.

[LEAP95] R. Lauwereins, M. Engels, M. Ade, and J. Peperstraete, *Grape-II: A system-level prototyping environment for DSP applications*, IEEE Computer **28** (1995), no. 2, 35–43.

[Lee03] E. A. Lee, *Overview of the Ptolemy project*, Tech. report, Tech Memo UCB/ERL M03/25, UC Berkeley, 2003.

[LHG$^+$89] E. Lee, W.-H. Ho, E. Goei, J. Bier, and S. Bhattacharyya, *Gabriel: A design environment for DSP*, IEEE Transactions on Acoustics, Speech and Signal Processing **37** (1989), no. 11, 1751–1762.

[LM87] E. A. Lee and D. G. Messerschmitt, *Static scheduling of synchronous data flow programs for digital signal processing*, IEEE Transactions on Computing **36** (1987), no. 1, 24–35.

[LPMS97] C. Lee, M. Potkonjak, and W. Mangione-Smith, *MediaBench: A tool for evaluating and synthesizing multimedia andcommunications systems*, IEEE MICRO, 1997.

[LS02] B. Long and S. Schenk, *Digital filmmaking handbook*, Charles River Media, 2002.

[LTA03] A. A. Lamb, W. Thies, and S. Amarasinghe, *Linear analysis and optimization of stream programs*, Conference on Programming Language Design and Implementation (PLDI), 2003.

[MAC99] V. Misra, J. Arias, and A. Chhabra, *A memory efficient method for fast transposing run-length encoded images*, International Conference on Document Analysis and Recognition (1999), 161–164.

[May73] D. Q. Mayne, *An elementary derivation of Rosenbrock's minimal realization algorithm*, IEEE Transactions on Automatic Control (1973), 306–307.

[MB99] P. K. Murthy and S. S. Bhattacharyya, *A buffer merging technique for reducing memory requirements of synchronous dataflow specifications*, Proceedings of the International Symposium on System Synthesis (San Jose, California), 1999, pp. 78–84.

[MB01] P. Murthy and S. Bhattacharyya, *Shared buffer implementations of signal processing systems using lifetime analysis techniques*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **20** (2001), no. 2, 177–198.

[MB04] 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** (2004), no. 2, 212–237.

[MBL94] P. Murthy, S. Bhattacharyya, and E. Lee, *Minimizing memory requirements for chain-structured synchronous dataflow programs*, International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 1994.

[MCR01] P. K. Murthy, E. G. Cohen, and S. Rowland, *System canvas: A new design environment for embedded DSP and telecommunication systems*, International Symposium on Hardware/Software Codesign (CODES), 2001.

[MDH$^+$06] Matthew, Drake, H. Hoffman, R. Rabbah, and S. Amarasinghe, *MPEG-2 decoding in a stream programming language*, International Parallel and Distributed Processing Symposium (IPDPS), 2006.

[Med08] C. Medford, *Microsoft/Yahoo is mobile equal of Google*, Red Herring (2008).

[MGAK03] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, *Cg: A system for programming graphics hardware in a C-like language*, SIGGRAPH, 2003.

[MIP99] M. Mandal, F. Idris, and S. Panchanathan, *A critical evaluation of image and video indexing techniques in the compressed domain*, Image and Vision Computing **17** (1999), no. 7, 513–529.

[ML02] P. Murthy and E. Lee, *Multidimensional synchronous dataflow*, IEEE Transactions on Signal Processing **50** (2002), no. 8, 2064–2079.

[ML03] P. Mattson and R. Lethin, *"Streaming" as a pattern*, August 2003.

[MM02] J. Mukherjee and S. Mitra, *Image resizing in the compressed domain using subband DCT*, IEEE Transactions on Circuits and Systems for Video Technology **12** (2002), no. 7, 620–627.

[MP05] A. Malton and A. Pahelvan, *Enhancing static architectural design recovery by lightweight dynamic analysis*, Workshop on Program Comprehension through Dynamic Analysis, 2005.

[MQP02] M. D. McCool, Z. Qin, and T. S. Popa, *Shader metaprogramming*, SIGGRAPH, 2002.

[MR01] F. Maraninchi and Y. Remond, *Argos: An automaton-based synchronous language*, Computer Languages **27** (2001), no. 1-3, 61–92.

[MSA$^+$85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldhoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas, *SISAL: Streams and iteration in a single assignment language*, Language reference manual, version 1.2, Lawrence Livermore National Laboratory, 1985.

[MSK87] D. May, R. Shepherd, and C. Keane, *Communicating process architecture: Transputers and Occam*, Future Parallel Computers: An Advanced Course (Lecture Notes in Computer Science) **272** (1987), 35–81.

[MTP$^+$04] M. McCool, S. D. Toit, T. Popa, B. Chan, and K. Moule, *Shader algebra*, SIGGRAPH, 2004.

[Mur89] T. Murata, *Petri nets: Properties, analysis and applications*, Proceedings of the IEEE **77** (1989), no. 4, 541–580.

[NA01] R. Nikhil and Arvind, *Implicit parallel programming in pH*, 1st ed., Morgan Kaufmann, 2001.

[Nav03] G. Navarro, *Regular expression searching on compressed text*, Journal of Discrete Algorithms **1** (2003).

[Nik91] R. S. Nikhil, *ID language reference manual, version 90.1*, Computation Structures Group Memo 284-2, Massachusetts Institute of Technology, 1991.

[NKH00] J. Nang, O. Kwon, and S. Hong, *Caption processing for MPEG video in MC-DCT compressed domain*, ACM Multimedia (2000), 211–218.

[NM03] N. Nethercote and A. Mycroft, *Redux: A dynamic dataflow tracer*, Workshop on Runtime Verification, 2003.

[NS07] N. Nethercote and J. Seward, *Valgrind: A framework for heavyweight dynamic binary instrumentation*, Conference on Programming Language Design and Implementation (PLDI), 2007.

[NT05] G. Navarro and J. Tarhio, *LZgrep: A Boyer-Moore string matching tool for Ziv-Lempel compressed text*, Software: Practice and Experience **35** (2005).

[NY04] M. Narayanan and K. Yelick, *Generating permutation instructions from a high-level description*, Workshop on Media and Streaming Processors (MSP), 2004.

[O'H91] D. O'Hallaron, *The Assign parallel program generator*, Distributed Memory Computing Conferenc, 1991.

[ON92] A. V. Oppenheim and S. H. Nawab (eds.), *Symbolic and knowledge-based signal processing*, Prentice Hall, 1992.

[ORSA05] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, *Automatic thread extraction with decoupled software pipelining*, International Symposium on Microarchitecture (MICRO), 2005.

[PBL95] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, *A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs*, Technical Report UCB/ERL M95/36, University of California, Berkeley, 1995.

[Pet62] C. Petri, *Communication with automata*, Ph.D. Thesis, Darmstadt Institue of Technology, 1962.

[PFTV92] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical recipes in C: The art of scientific computing*, 2nd ed., Cambridge University Press, October 1992.

[PKL80] D. Padua, D. Kuck, and D. Lawrie, *High-speed multiprocessors and compilation techniques*, Transactions on Computers **C-29** (1980), no. 9.

[PL95] J. L. Pino and E. A. Lee, *Hierarchical static scheduling of dataflow graphs onto multiple processors*, International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 1995.

[PM91] K. Parhi and D. Messerschmitt, *Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding*, IEEE Transactions on Computers **40** (1991), no. 2, 178–195.

[PM03] M. Püschel and J. Moura, *The algebraic approach to the discrete cosine and sine transforms and their fast algorithms*, SIAM Journal of Computing **32** (2003), no. 5, 1280–1316.

[PMJ$^+$05] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, *SPIRAL: Code generation for DSP transforms*, Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation **93** (2005), no. 2.

[Pog03] D. Pogue, *Imovie 3 & IDVD: The missing manual*, O'Reilly, 2003.

[PPL95] T. M. Parks, J. L. Pino, and E. A. Lee, *A comparison of synchronous and cycle-static dataflow*, Asilomar Conference on Signals, Systems, and Computers, 1995.

[PR97] W. Pugh and E. Rosser, *Iteration based slicing and its application to communication optimization*, International Conference on Supercomputing, 1997.

[PR00] M. Potkonjak and J. M. Rabaey, *Maximally and arbitrarily fast implementation of linear and feedback linear computations*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **19** (2000), no. 1, 30–43.

[Pri91] H. Printz, *Automatic mapping of large signal processing systems to a parallel machine*, Ph.D. Thesis, Carnegie Mellon University, 1991.

[PV06] G. D. Price and M. Vachharajani, *A case for compressing traces with BDDs*, Computer Architecture Letters **5** (2006), no. 2.

[PW96] T. A. Proebsting and S. A. Watterson, *Filter fusion*, Symposium on the Principles of Programming Languages (POPL), 1996.

[Rau98] L. Rauchwerger, *Run-time parallelization: Its time has come*, Parallel Computing **24** (1998), no. 3-4.

[Reu03] A. Reuther, *Preliminary design review: GMTI narrowband for the basic PCA integrated radar-tracker application*, Tech. Report ESC-TR-2003-076, MIT Lincoln Laboratory, 2003.

[Rey04] J. C. Reyes, *A graph editing framework for the StreamIt language*, M.Eng. Thesis, Massachusetts Institute of Technology, 2004.

[RVDB06] S. Rul, H. Vandierendonck, and K. De Bosschere, *Function level parallelism driven by data dependencies*, Workshop on Design, Architecture and Simulation of Chip Multi-Processors, 2006.

[RVVA04] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August, *Decoupled software pipelining with the synchronization array*, International Conference on Parallel Architectures and Compilation Techniques (PACT), 2004.

[Rya92] S. Ryan, *Linear data flow analysis*, ACM SIGPLAN Notices **27** (1992), no. 4, 59–67.

[Sen80] S. Seneff, *Speech transformation system (spectrum and/or excitation) without pitch extraction*, Master's thesis, Massachussetts Institute of Technology, 1980.

[Ser05] J. Sermulins, *Cache Optimizations for Stream Programs*, M.Eng. Thesis, Massachusetts Institute of Technology, 2005.

[SGB06] S. Stuijk, M. Geilen, and T. Basten, *Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs*, Design Automation Conference (DAC), 2006.

[Sho95] K. Shoji, *An algorithm for affine transformation of binary images stored in pxy tables by run format*, Systems and Computers in Japan **26** (1995), no. 7, 69–78.

[SLRBE05] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu, *Programming by sketching for bit-streaming programs*, Conference on Programming Language Design and Implementation (PLDI), 2005.

[Smi95] B. Smith, *A survey of compressed domain processing techniques*, Cornell University, 1995.

[So07] W. So, *Software thread integration for instruction level parallelism*, Ph.D. Thesis, North Carolina State University, 2007.

[SR96] B. Smith and L. Rowe, *Compressed domain processing of JPEG-encoded images*, Real-Time Imaging **2** (1996), no. 2, 3–17.

[SS95] J. Swartz and B. Smith, *RIVL: A resolution independent video language*, Proceedings of the Tcl/TK Workshop, 1995.

[SS96a] B. Shen and I. Sethi, *Convolution-based edge detection for image/video in block DCT domain*, Journal of Visual Communication and Image Representation **7** (1996), no. 4, 411–423.

[SS96b] ______, *Direct feature extraction from compressed images*, SPIE Storage & Retrieval for Image and Video Databases IV **2670** (1996).

[SS98] ______, *Block-based manipulations on transform-compressed images and videos*, Multimedia Systems **6** (1998), no. 2, 113–124.

[Ste97] R. Stephens, *A survey of stream processing*, Acta Informatica **34** (1997), no. 7, 491–541.

[Stra] *StreamIt homepage*, `http://cag.csail.mit.edu/streamit`.

[Strb] *StreamIt cookbook*,
`http://cag.csail.mit.edu/streamit/papers/streamit-cookbook.pdf`.

[Strc] *StreamIt language specification*, `http://cag.csail.mit.edu/streamit/papers/streamit-lang-spec.pdf`.

[Str94] B. Stroustrup, *The Design and Evolution of C++*, Addison-Wesley Professional, 1994.

[STRA05] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe, *Cache aware optimization of stream programs*, Conference on Languages, Compilers, Tools for Embedded Systems (LCTES), 2005.

[TB96] D. L. Tennenhouse and V. G. Bose, *The SpectrumWare approach to wireless signal processing*, Wireless Networks **2** (1996), no. 1, 1–12.

[TCA07] W. Thies, V. Chandrasekhar, and S. Amarasinghe, *A practical approach to exploiting coarse-grained pipeline parallelism in C programs*, International Symposium on Microarchitecture (MICRO), 2007.

[THA07] W. Thies, S. Hall, and S. Amarasinghe, *Mapping stream programs into the compressed domain*, Tech. Report MIT-CSAIL-TR-2007-055, Massachusetts Institute of Technology, 2007, `http://hdl.handle.net/1721.1/39651`.

[Tip95] F. Tip, *A survey of program slicing techniques*, Journal of Programming Languages **3** (1995), no. 3.

[TKA02] W. Thies, M. Karczmarek, and S. Amarasinghe, *StreamIt: A language for streaming applications*, International Conference on Compiler Construction (CC), 2002.

[TKG$^+$02] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe, *A common machine language for grid-based architectures*, ACM SIGARCH Computer Architecture News, 2002.

[TKM$^+$02] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, J.-W. Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, *The Raw microprocessor: A computational fabric for software circuits and general purpose programs*, IEEE MICRO **22** (2002).

[TKS$^+$05] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, *Teleport messaging for distributed stream programs*, Symposium on Principles and Practice of Parallel Programming (PPoPP), 2005.

[TLM$^+$04] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff., I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, *Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams*, International Symposium on Computer Architecture (ISCA), 2004.

[TPO06] D. Tarditi, S. Puri, and J. Oglesby, *Accelerator: Using data parallelism to program GPUs for general-purpose uses*, International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2006.

[Tse89] P.-S. Tseng, *A parallelizing compiler for disributed memory parallel computers*, Ph.D. thesis, Carnegie Mellon University, 1989.

[TZB99] J. Teich, E. Zitzler, and S. S. Bhattacharyya, *3D exploration of software schedules for DSP algorithms*, International Workshop on Hardware/Software Codesign (CODES), 1999.

[U.S04] U.S. Geological Survey, *Annual report of data sales, distribution, and archiving*, National Center for EROS, 2004, `http://edc.usgs.gov/about/reports/sales2004.pdf`.

[Vas98] B. Vasudev, *Compressed-domain reverse play of MPEG video streams*, SPIE Conference on Multimedia Systems and Applications (1998), 237–248.

[VRR$^+$07] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, *Speculative decoupled software pipelining*, Parallel Architectures and Compilation Techniques (PACT), 2007.

[Wat06] D. Watkins, *Mash Hits*, The Guardian (2006).

[Wol82] M. Wolfe, *Optimizing supercompilers for supercomputers*, Ph.D. Thesis, University of Illinois, Urbana-Champaign, 1982.

[Won04] J. Wong, *Modeling the scalability of acyclic stream programs*, M.Eng. Thesis, Massachusetts Institute of Technology, 2004.

[WPD01] R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated empirical optimizations of software and the ATLAS project*, Parallel Computing **27** (2001), no. 1–2, 3–35.

[WSA02] S. Wee, B. Shen, and J. Apostolopoulos, *Compressed-domain video processing*, HP Labs Technical Report, HPL-2002 **282** (2002).

[WTS$^+$97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, et al., *Baring it all to software: Raw machines*, IEEE Computer **30** (1997), no. 9, 86–93.

[WZ94] A. Wyner and J. Ziv, *The sliding-window Lempel-Ziv algorithm is asymptotically optimal*, Proceedings of the IEEE **82** (1994), no. 6.

[Xio01] J. Xiong, *Automatic optimization of DSP algorithms*, Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, 2001.

[XJJP01] J. Xiong, J. Johnson, R. W. Johnson, and D. A. Padua, *SPL: A language and compiler for DSP algorithms*, Conference on Programming Language Design and Implementation (PLDI), 2001, pp. 298–308.

[Zha07] X. D. Zhang, *A streaming computation framework for the Cell processor*, M.Eng. Thesis, Massachusetts Institute of Technology, Cambridge, MA, 2007.

[ZLRA08] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe, *A lightweight streaming layer for multicore execution*, Workshop on Design, Architecture, and Simulation of Chip Multi-Processors (dasCMP), 2008.

[ZLSL05] D. Zhang, Z.-Z. Li, H. Song, and L. Liu, *A programming model for an embedded media processing architecture*, International Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS), 2005.

[ZSdMNBY00] N. Ziviani, E. Silva de Moura, G. Navarro, and R. Baeza-Yates, *Compression: A key for next-generation text retrieval systems*, IEEE Computer **33** (2000), no. 11.

[ZTB00] E. Zitzler, J. Teich, and S. S. Bhattacharyya, *Multidimensional exploration of software implementations for DSP algorithms*, Journal of VLSI Signal Processing **24** (2000), no. 1, 83–98.

Appendix A

Example StreamIt Program

This appendix provides a complete program listing for a small StreamIt benchmark, ChannelVocoder. The stream graph for this benchmark can be found in Figure B-9.

```
/**
 * Author: Andrew Lamb
 *
 * This is a channel vocoder as described in MIT 6.555 Lab 2. Its salient features
 * are a filter bank, each of which contains a decimator after a bandpass filter.
 *
 * First the signal is conditioned using a lowpass filter with cutoff at 5000
 * Hz. Then the signal is “center clipped” which basically means that very high and
 * very low values are removed. The sampling rate is 8000 Hz.
 *
 * Then, the signal is sent both to a pitch detector and to a filter bank with 200 Hz
 * wide windows (16 overall).
 *
 * Thus, each output is the combination of 16 band envelope values from the filter
 * bank and a single pitch detector value. This value is either the pitch if the
 * sound was voiced or 0 if the sound was unvoiced.
 **/
void->void pipeline ChannelVocoder {
    int PITCH_WINDOW = 100; // the number of samples to base the pitch detection on
    int DECIMATION = 50; // decimation factor
    int NUM_FILTERS = 16;

    add FileReader<float>("input.dat");
    // low pass filter to filter out high freq noise
    add LowPassFilter(1, (2*pi*5000)/8000, 64);
    add float->float splitjoin {
        split duplicate;
        add PitchDetector(PITCH_WINDOW, DECIMATION);
        add VocoderFilterBank(NUM_FILTERS, DECIMATION);
        join roundrobin(1, NUM_FILTERS);
    }
    add FileWriter<float>("output.dat");
}
```

```
/**
 * Pitch detector.
 **/
float->float pipeline PitchDetector(int winsize, int decimation) {
    add CenterClip();
    add CorrPeak(winsize, decimation);
}

/**
 * The channel vocoder filterbank.
 **/
float->float splitjoin VocoderFilterBank(int N, int decimation) {
    split duplicate;
    for (int i=0; i<N; i++) {
        add FilterDecimate(i, decimation);
    }
    join roundrobin;
}

/**
 * A channel of the vocoder filter bank – has a band pass filter centered at i*200
 * Hz followed by a decimator with decimation rate of decimation.
 **/
float->float pipeline FilterDecimate(int i, int decimation) {
    add BandPassFilter(2, 400*i, 400*(i+1), 64);
    add Compressor(decimation);
}

/**
 * This filter “center clips” the input value so that it is always within the range
 * of -.75 to .75
 **/
float->float filter CenterClip {
    float MIN = -0.75;
    float MAX = 0.75;
    work pop 1 push 1 {
        float t = pop();
        if (t<MIN) {
            push(MIN);
        } else if (t>MAX) {
            push(MAX);
        } else {
            push(t);
        }
    }
}
```

```
/**
 * This filter calculates the autocorrelation of the next winsize elements and then
 * chooses the max peak. If the max peak is under a threshold we output a zero. If
 * the max peak is above the threshold, we simply output its value.
 **/
float->float filter CorrPeak(int winsize, int decimation) {
    float THRESHOLD = 0.07;
    work peek winsize push 1 pop decimation {
        float[winsize] autocorr; // auto correlation
        for (int i=0; i<winsize; i++) {
            float sum = 0;
            for (int j=i; j<winsize; j++) {
                sum += peek(i)*peek(j);
            }
            autocorr[i] = sum/winsize;
        }

        // armed with the auto correlation, find the max peak in a real vocoder, we
        // would restrict our attention to the first few values of the auto corr to
        // catch the initial peak due to the fundamental frequency.
        float maxpeak = 0;
        for (int i=0; i<winsize; i++) {
            if (autocorr[i]>maxpeak) {
                maxpeak = autocorr[i];
            }
        }

        // output the max peak if it is above the threshold.
        // otherwise output zero.
        if (maxpeak > THRESHOLD) {
            push(maxpeak);
        } else {
            push(0);
        }
        for (int i=0; i<decimation; i++) {
            pop();
        }
    }
}
```

```
/**
 * A simple adder which takes in N items and pushes out the sum of them.
 **/
float->float filter Adder(int N) {
    work pop N push 1 {
        float sum = 0;
        for (int i=0; i<N; i++) {
            sum += pop();
        }
        push(sum);
    }
}

/**
 * This is a bandpass filter with the rather simple implementation of a low pass
 * filter cascaded with a high pass filter. The relevant parameters are: end of
 * stopband=ws and end of passband=wp, such that 0<=ws<=wp<=pi gain of passband and
 * size of window for both filters. Note that the high pass and low pass filters
 * currently use a rectangular window.
 **/
float->float pipeline BandPassFilter(float gain, float ws, float wp, int numSamples) {
    add LowPassFilter(1, wp, numSamples);
    add HighPassFilter(gain, ws, numSamples);
}

/**
 * This filter compresses the signal at its input by a factor M.
 * Eg it inputs M samples, and only outputs the first sample.
 **/
float->float filter Compressor(int M) {
    work peek M pop M push 1 {
        push(pop());
        for (int i=0; i<(M-1); i++) {
            pop();
        }
    }
}
```

```
/**
 * Simple FIR high pass filter with gain=g, stopband ws(in radians) and N samples.
 *
 * Eg
 *                  ^ H(e^jw)
 *                  |
 *     _____        |     _____
 *    |     |     |     |     |
 *    |     |     |     |     |
 *   <-----------------------> w
 *                pi-wc pi pi+wc
 *
 * This implementation is a FIR filter is a rectangularly windowed sinc function (eg
 * sin(x)/x) multiplied by e^(j*pi*n)=(-1)^n, which is the optimal FIR high pass
 * filter in mean square error terms.
 *
 * Specifically, h[n] has N samples from n=0 to (N-1)
 * such that h[n] = (-1)^(n-N/2) * sin(cutoffFreq*pi*(n-N/2))/(pi*(n-N/2)).
 * where cutoffFreq is pi-ws
 * and the field h holds h[-n].
 */
float->float filter HighPassFilter(float g, float ws, int N) {
    float[N] h;

    /* since the impulse response is symmetric, I don't worry about reversing h[n]. */
    init {
        int OFFSET = N/2;
        float cutoffFreq = pi - ws;
        for (int i=0; i<N; i++) {
            int idx = i + 1;
            /* flip signs every other sample (done this way so that it gets array destroyed) */
            int sign = ((i%2) == 0) ? 1 : -1;
            // generate real part
            if (idx == OFFSET)
                /* take care of div by 0 error (lim x->oo of sin(x)/x actually equals 1)*/
                h[i] = sign * g * cutoffFreq / pi;
            else
                h[i] = sign * g * sin(cutoffFreq * (idx-OFFSET)) / (pi*(idx-OFFSET));
        }
    }

    /* implement the FIR filtering operation as the convolution sum. */
    work peek N pop 1 push 1 {
        float sum = 0;
        for (int i=0; i<N; i++) {
            sum += h[i]*peek(i);
        }
        push(sum);
        pop();
    }
}
```

```
/**
 * Simple FIR low pass filter with gain=g, wc=cutoffFreq(in radians) and N samples.
 * Eg:
 *                ^ H(e^jw)
 *                |
 *         ---------------
 *         |      |      |
 *         |      |      |
 *    <----------------------> w
 *       -wc            wc
 *
 * This implementation is a FIR filter is a rectangularly windowed sinc function (eg
 * sin(x)/x), which is the optimal FIR low pass filter in mean square error terms.
 *
 * Specifically, h[n] has N samples from n=0 to (N-1)
 * such that h[n] = sin(cutoffFreq*pi*(n-N/2))/(pi*(n-N/2)).
 * and the field h holds h[-n].
 */
float->float filter LowPassFilter(float g, float cutoffFreq, int N) {
    float[N] h;

    /* since the impulse response is symmetric, I don't worry about reversing h[n]. */
    init {
        int OFFSET = N/2;
        for (int i=0; i<N; i++) {
            int idx = i + 1;
            // generate real part
            if (idx == OFFSET)
                /* take care of div by 0 error (lim x->oo of sin(x)/x actually equals 1)*/
                h[i] = g * cutoffFreq / pi;
            else
                h[i] = g * sin(cutoffFreq * (idx-OFFSET)) / (pi*(idx-OFFSET));
        }
    }

    /* Implement the FIR filtering operation as the convolution sum. */
    work peek N pop 1 push 1 {
        float sum = 0;
        for (int i=0; i<N; i++) {
            sum += h[i]*peek(i);
        }
        push(sum);
        pop();
    }
}
```

Appendix B

Graphs of StreamIt Benchmarks

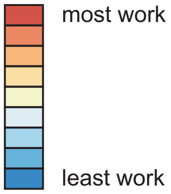
This appendix contains stream graphs for the StreamIt benchmark suite (detailed in Table 2-10). As described in Table 2-11, many of the graphs are parameterized, and we often assign small values to the parameters in order to facilitate visualization and comprehension of the graph. Graphs with different sizes, shapes, and work distributions can be obtained by varying the parameters.

The stream graphs reflect the structure of the original input program, prior to any transformations by the compiler. In practice, the compiler canonicalizes each graph by removing redundant synchronization points, flattening nested pipelines, and collapsing data-parallel splitjoins. With the exception of GMTI and MPEG2[1], this canonicalization is disabled to illustrate the programmer's original intent.

In the stream graphs, each filter is annotated with the following information:

- The filter name.
- The number of items[2] pushed and popped per execution of the filter.
- The estimated work (number of cycles) per execution of the filter.
- Peeking filters are annotated with the number of items peeked (but not popped) per execution.
- Stateful filters are annotated as such.

Filters are also colored to indicate their approximate amount of work relative to other filters in the same program. The heaviest and lightest filters in a program are assigned fixed colors, and intermediate filters are colored on a linear scale between the two:



Work estimates are gathered statically and may differ by 2x or more from actual runtime values. Work estimates are not available in some programs due to dynamic rates or Java subroutines. Also, some individual filters are marked as having "unknown" work in cases where the work estimator is known to perform poorly (while loops, recursive functions, etc.)

[1] The stream graphs for GMTI and MPEG2 are canonicalized by the compiler, in order to reduce their size and improve the visualization.

[2] Some items may contain multiple values. For example, if a data channel carries items of array type, then the graphs illustrate the number of arrays pushed and popped per execution.

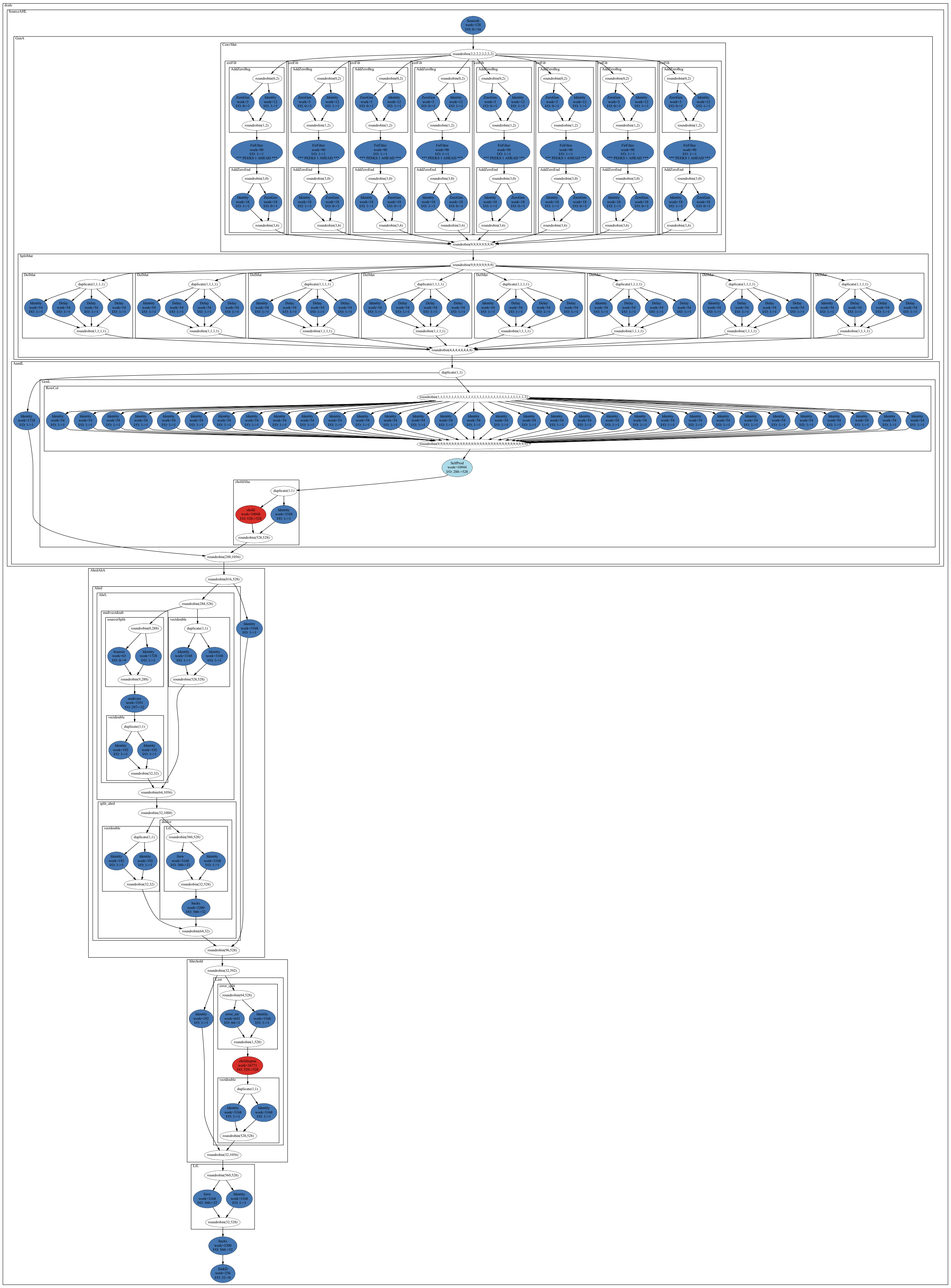


Figure B-1: Stream graph for 3GPP.

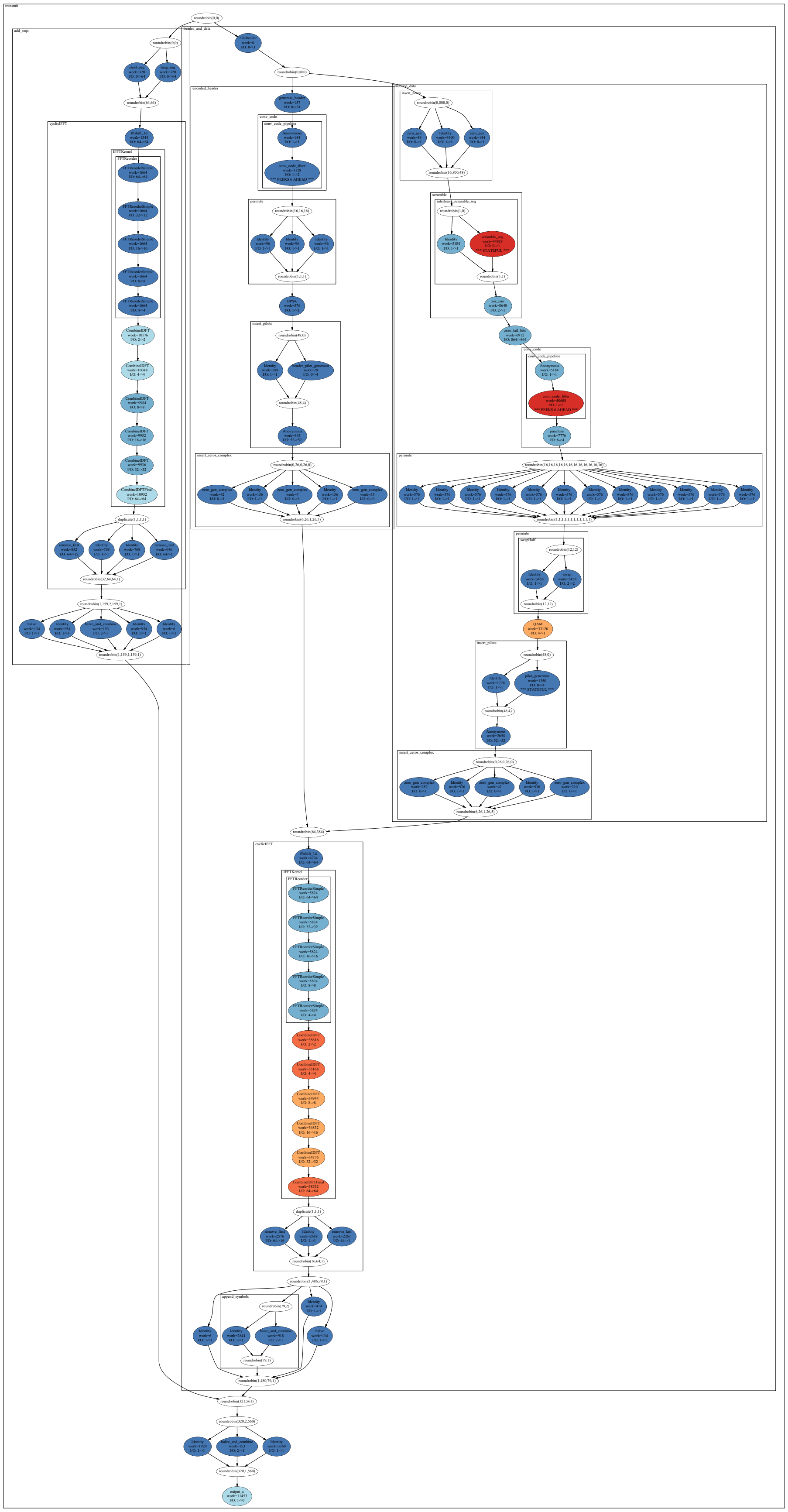


Figure B-2: Stream graph for 802.11a.

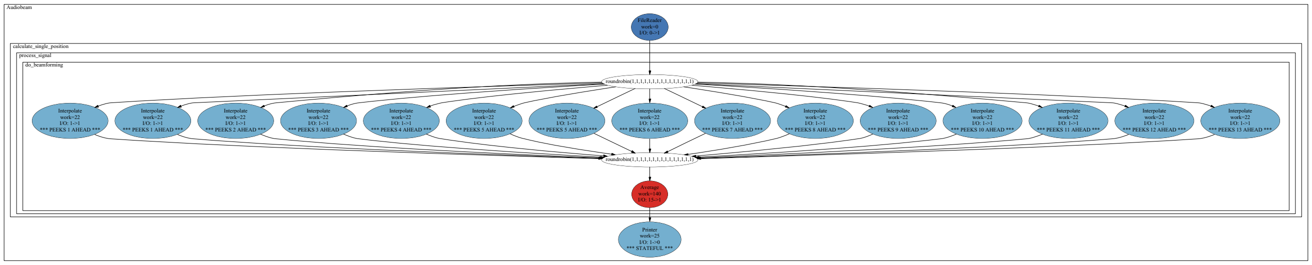


Figure B-3: Stream graph for Audiobeam.

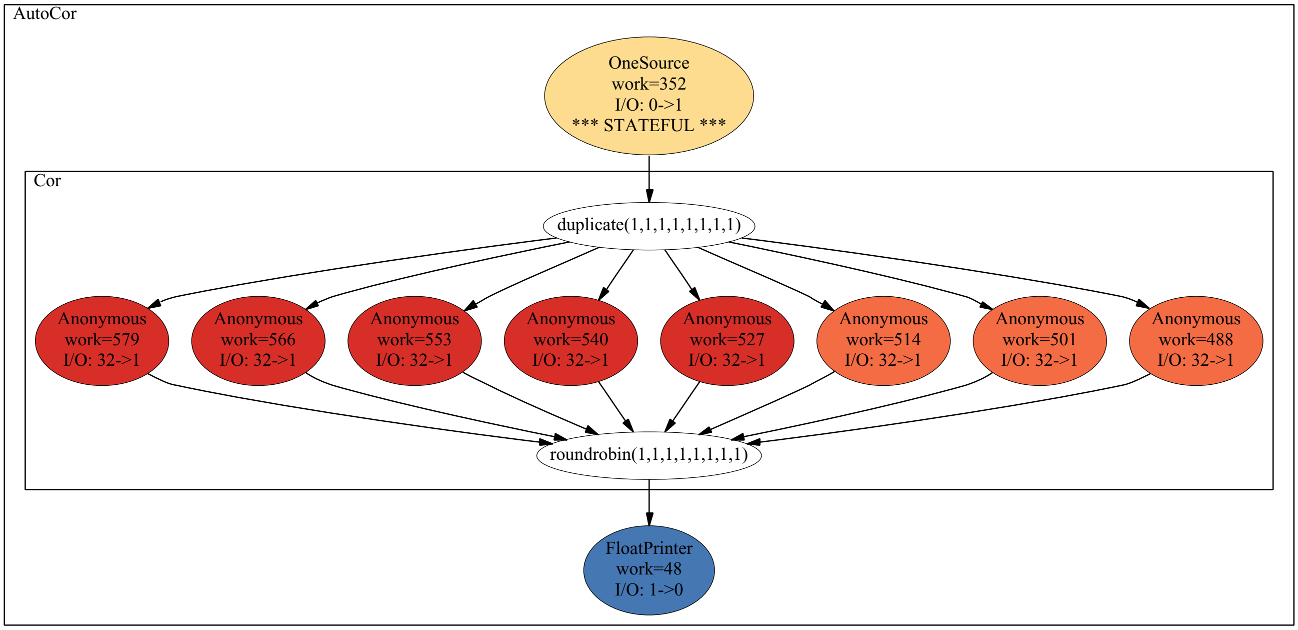


Figure B-4: Stream graph for Autocor.

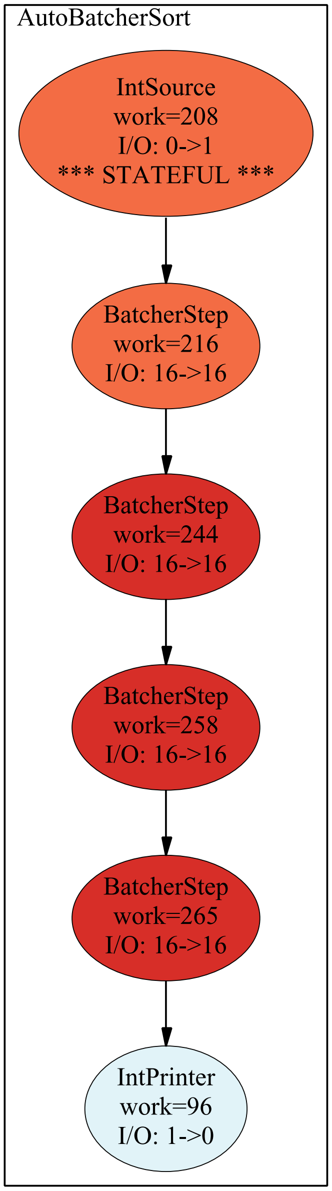


Figure B-5: Stream graph for BitonicSort (coarse).

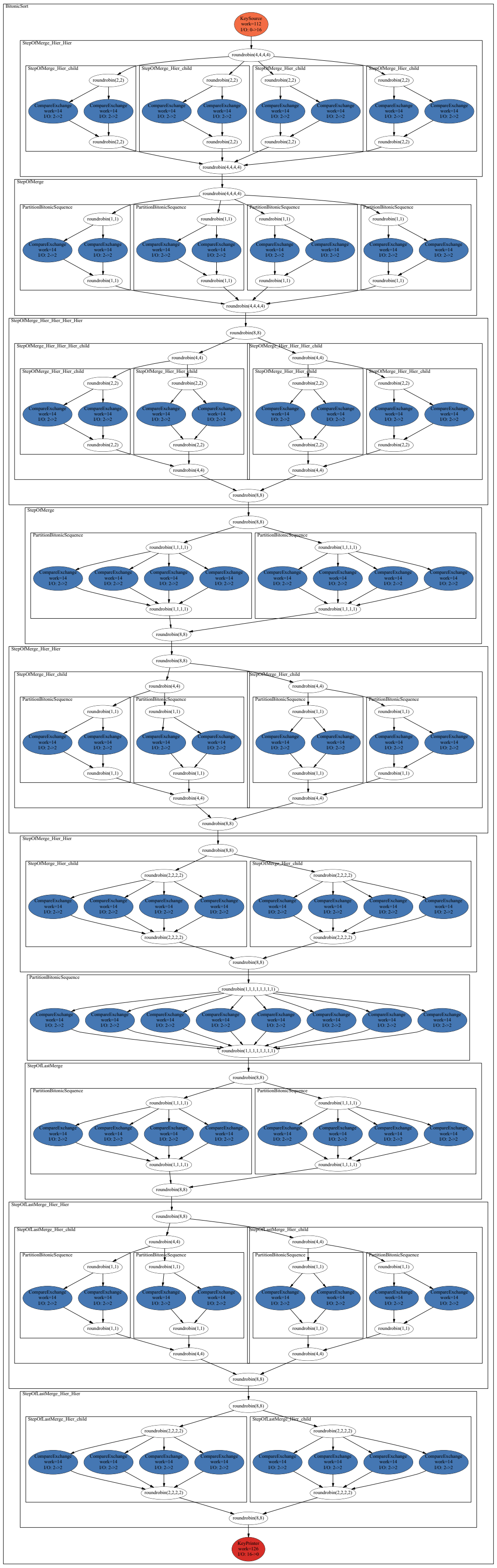
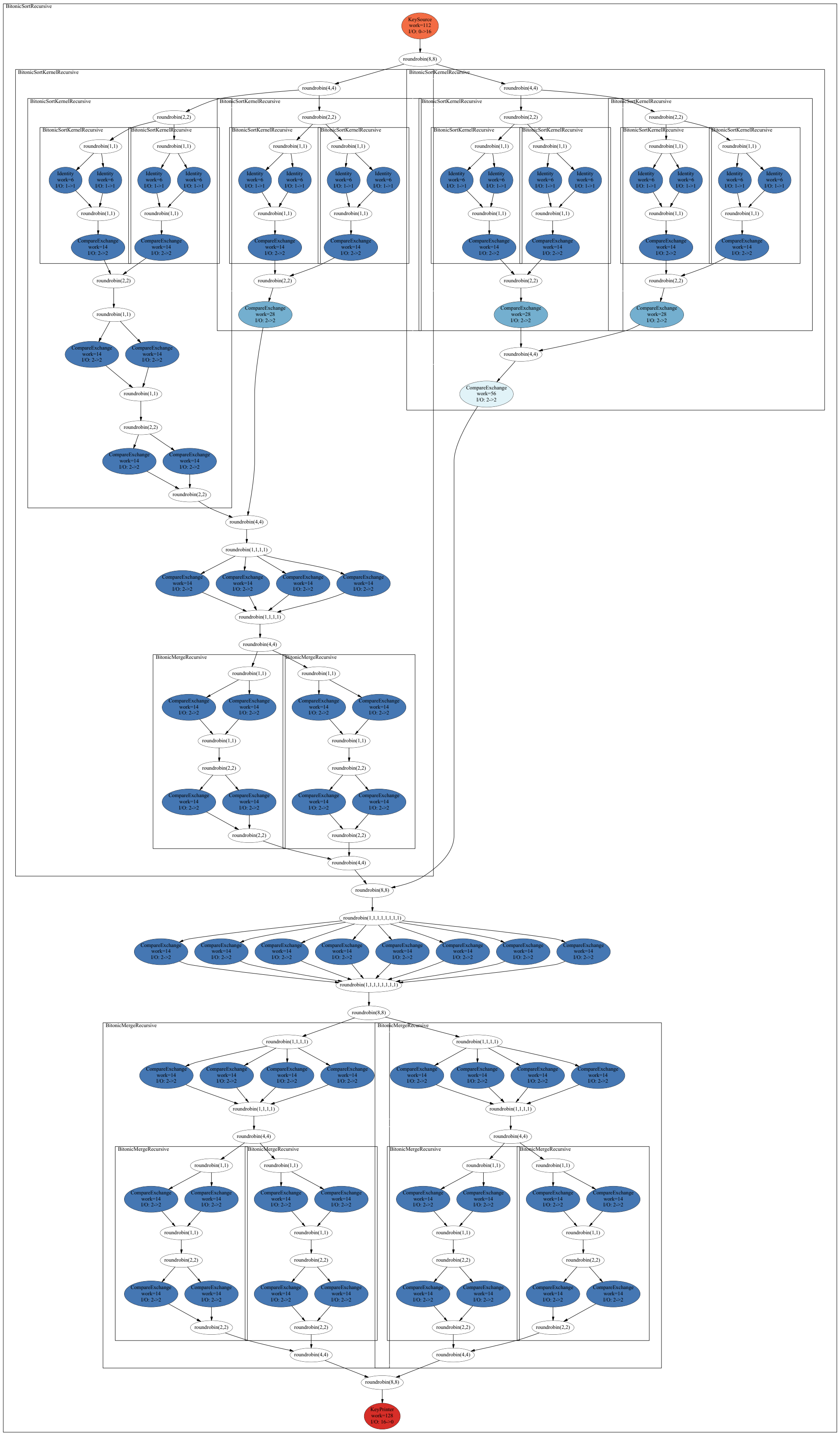


Figure B-6: Stream graph for BitonicSort (fine, iterative).



BitonicSortRecursive
KeySource work=112 I/O: 0->16
roundrobin(8,8)
BitonicSortKernelRecursive
roundrobin(4,4)
BitonicSortKernelRecursive
roundrobin(2,2)
BitonicSortKernelRecursive
roundrobin(1,1)
Identity work=6 I/O: 1->1
Identity work=6 I/O: 1->1
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
BitonicSortKernelRecursive
roundrobin(1,1)
Identity work=6 I/O: 1->1
Identity work=6 I/O: 1->1
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
roundrobin(2,2)
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(1,1)
roundrobin(2,2)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(2,2)
BitonicSortKernelRecursive
roundrobin(2,2)
BitonicSortKernelRecursive
roundrobin(1,1)
Identity work=6 I/O: 1->1
Identity work=6 I/O: 1->1
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
BitonicSortKernelRecursive
roundrobin(1,1)
Identity work=6 I/O: 1->1
Identity work=6 I/O: 1->1
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
roundrobin(2,2)
CompareExchange work=28 I/O: 2->2
roundrobin(4,4)
roundrobin(1,1,1,1)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(1,1,1,1)
roundrobin(4,4)
BitonicMergeRecursive
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(1,1)
roundrobin(2,2)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(2,2)
BitonicMergeRecursive
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(1,1)
roundrobin(2,2)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(2,2)
roundrobin(4,4)
BitonicSortKernelRecursive
roundrobin(4,4)
BitonicSortKernelRecursive
roundrobin(2,2)
BitonicSortKernelRecursive
roundrobin(1,1)
Identity work=6 I/O: 1->1
Identity work=6 I/O: 1->1
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
BitonicSortKernelRecursive
roundrobin(1,1)
Identity work=6 I/O: 1->1
Identity work=6 I/O: 1->1
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
roundrobin(2,2)
CompareExchange work=28 I/O: 2->2
BitonicSortKernelRecursive
roundrobin(2,2)
BitonicSortKernelRecursive
roundrobin(1,1)
Identity work=6 I/O: 1->1
Identity work=6 I/O: 1->1
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
BitonicSortKernelRecursive
roundrobin(1,1)
Identity work=6 I/O: 1->1
Identity work=6 I/O: 1->1
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
roundrobin(2,2)
CompareExchange work=28 I/O: 2->2
roundrobin(4,4)
CompareExchange work=56 I/O: 2->2
roundrobin(8,8)
roundrobin(1,1,1,1,1,1,1,1)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(1,1,1,1,1,1,1,1)
roundrobin(8,8)
BitonicMergeRecursive
roundrobin(1,1,1,1)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(1,1,1,1)
roundrobin(4,4)
BitonicMergeRecursive
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(1,1)
roundrobin(2,2)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(2,2)
BitonicMergeRecursive
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(1,1)
roundrobin(2,2)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(2,2)
roundrobin(4,4)
BitonicMergeRecursive
roundrobin(1,1,1,1)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(1,1,1,1)
roundrobin(4,4)
BitonicMergeRecursive
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(1,1)
roundrobin(2,2)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(2,2)
BitonicMergeRecursive
roundrobin(1,1)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(1,1)
roundrobin(2,2)
CompareExchange work=14 I/O: 2->2
CompareExchange work=14 I/O: 2->2
roundrobin(2,2)
roundrobin(4,4)
roundrobin(8,8)
KeyPrinter work=128 I/O: 16->0

Figure B-7: Stream graph for BitonicSort (fine, recursive).

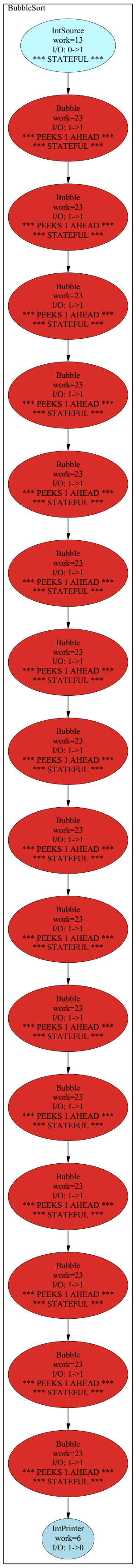


Figure B-8: Stream graph for BubbleSort.

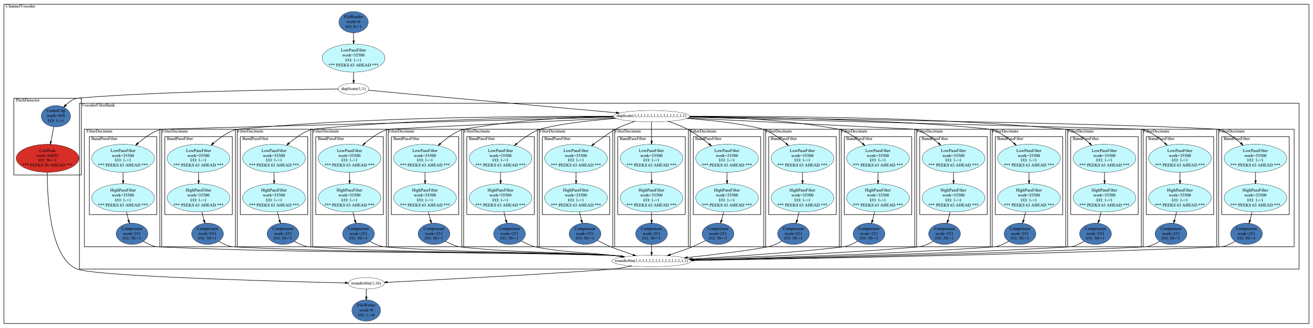


Figure B-9: Stream graph for ChannelVocoder.

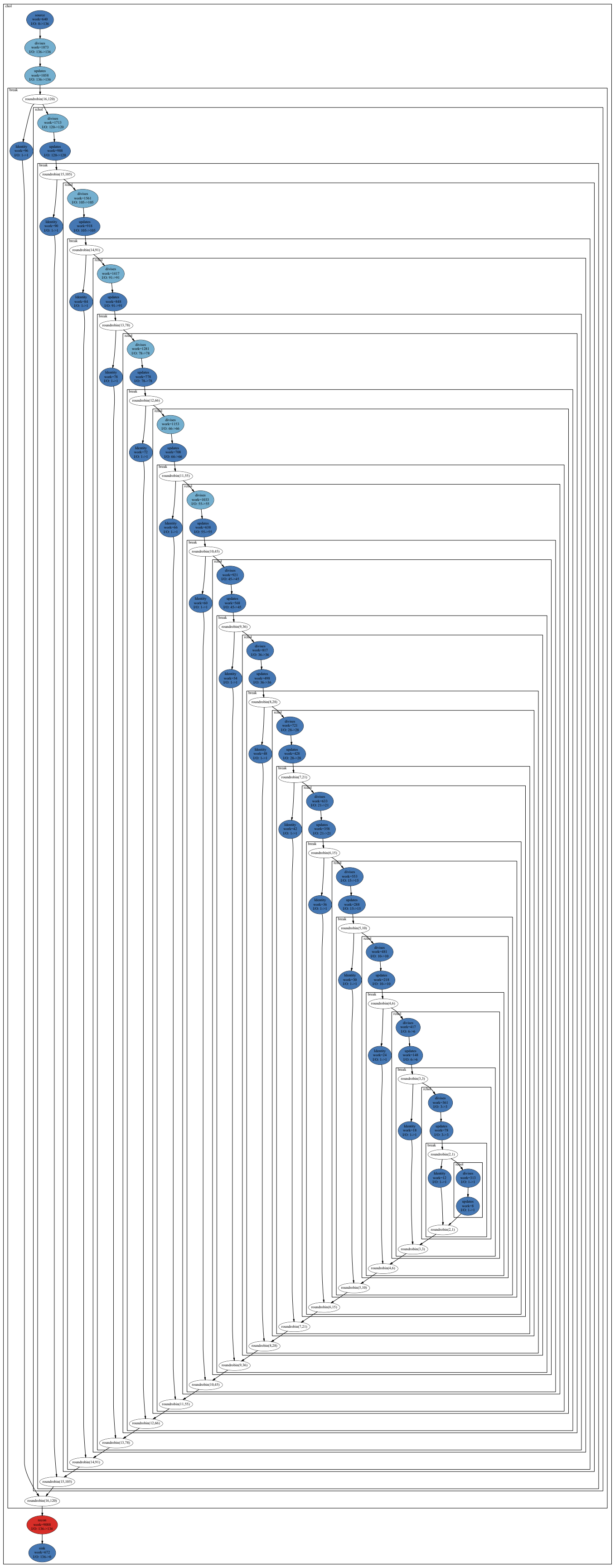


Figure B-10: Stream graph for Cholesky.

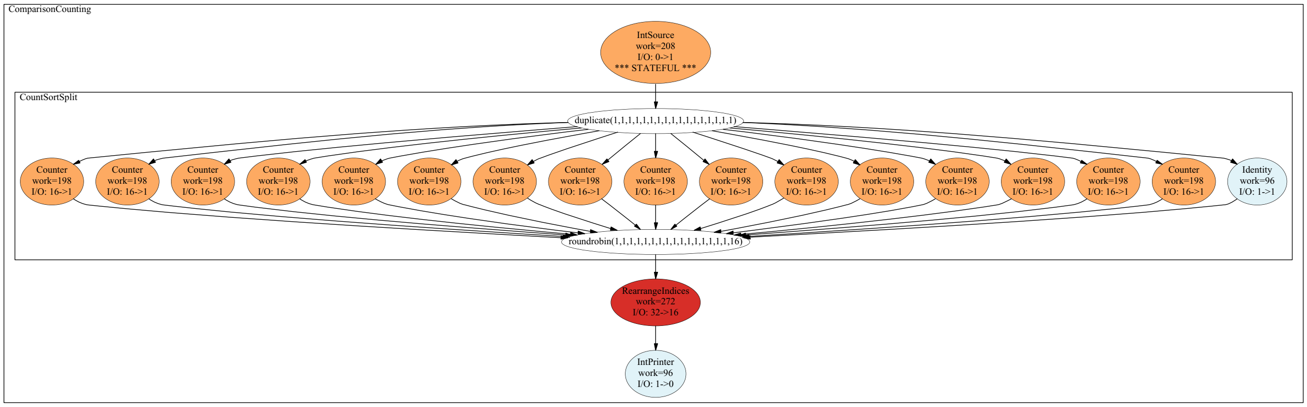


Figure B-11: Stream graph for ComparisonCounting.

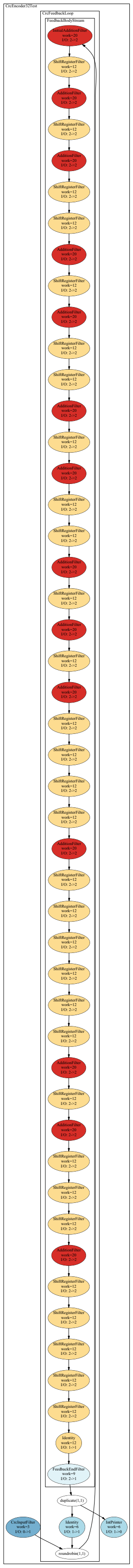

CrcEncoder32Test
CrcFeedbackLoop
FeedbackBodyStream
InitialAdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
AdditionFilter work=20 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
ShiftRegisterFilter work=12 I/O: 2->2
Identity work=12 I/O: 1->1
FeedbackEndFilter work=9 I/O: 2->1
duplicate(1,1)
CrcInputFilter work=3 I/O: 0->1
Identity work=6 I/O: 1->1
IntPrinter work=6 I/O: 1->0
roundrobin(1,1)

Figure B-12: Stream graph for CRC.

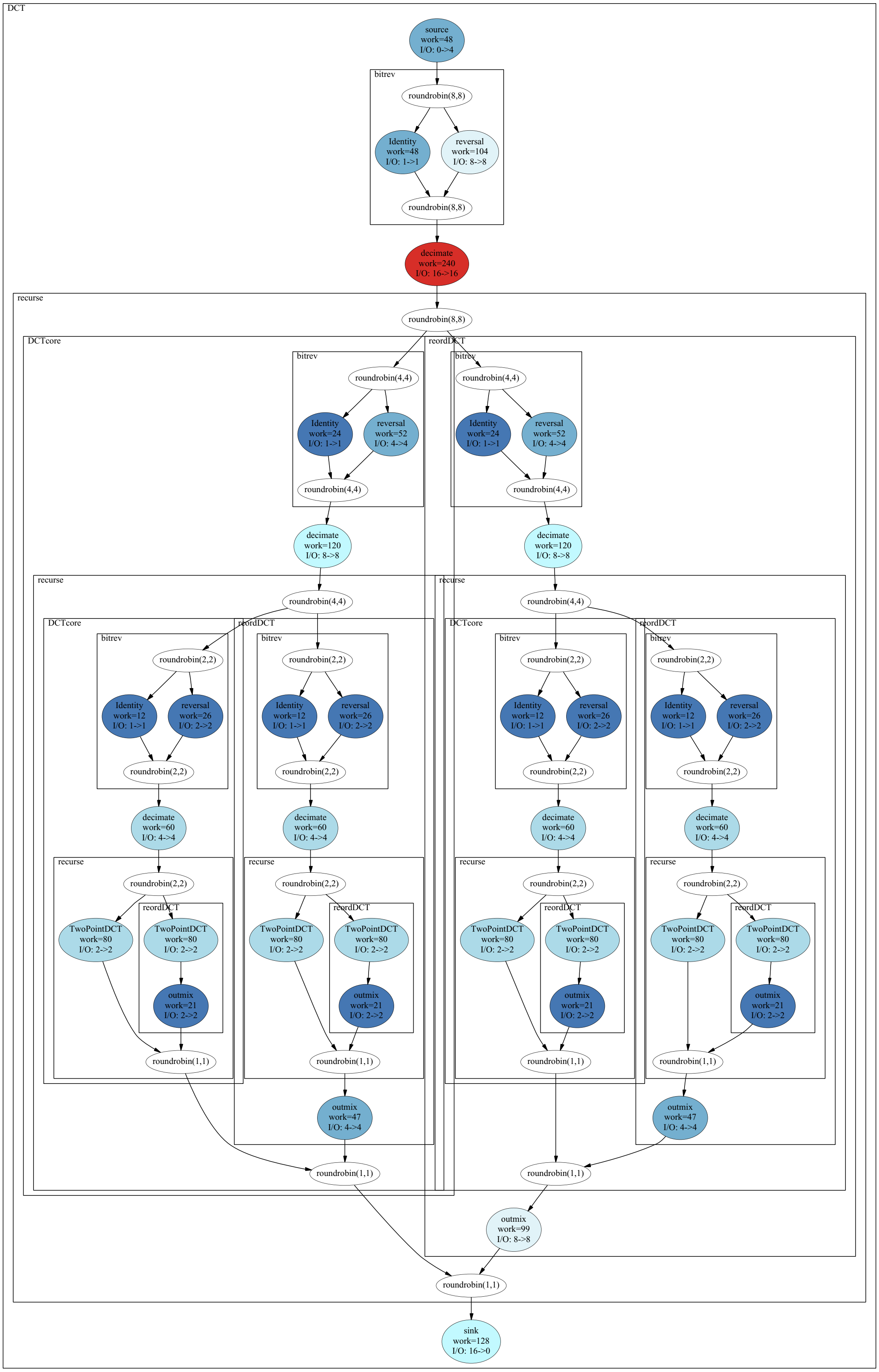


Figure B-13: Stream graph for DCT (float).

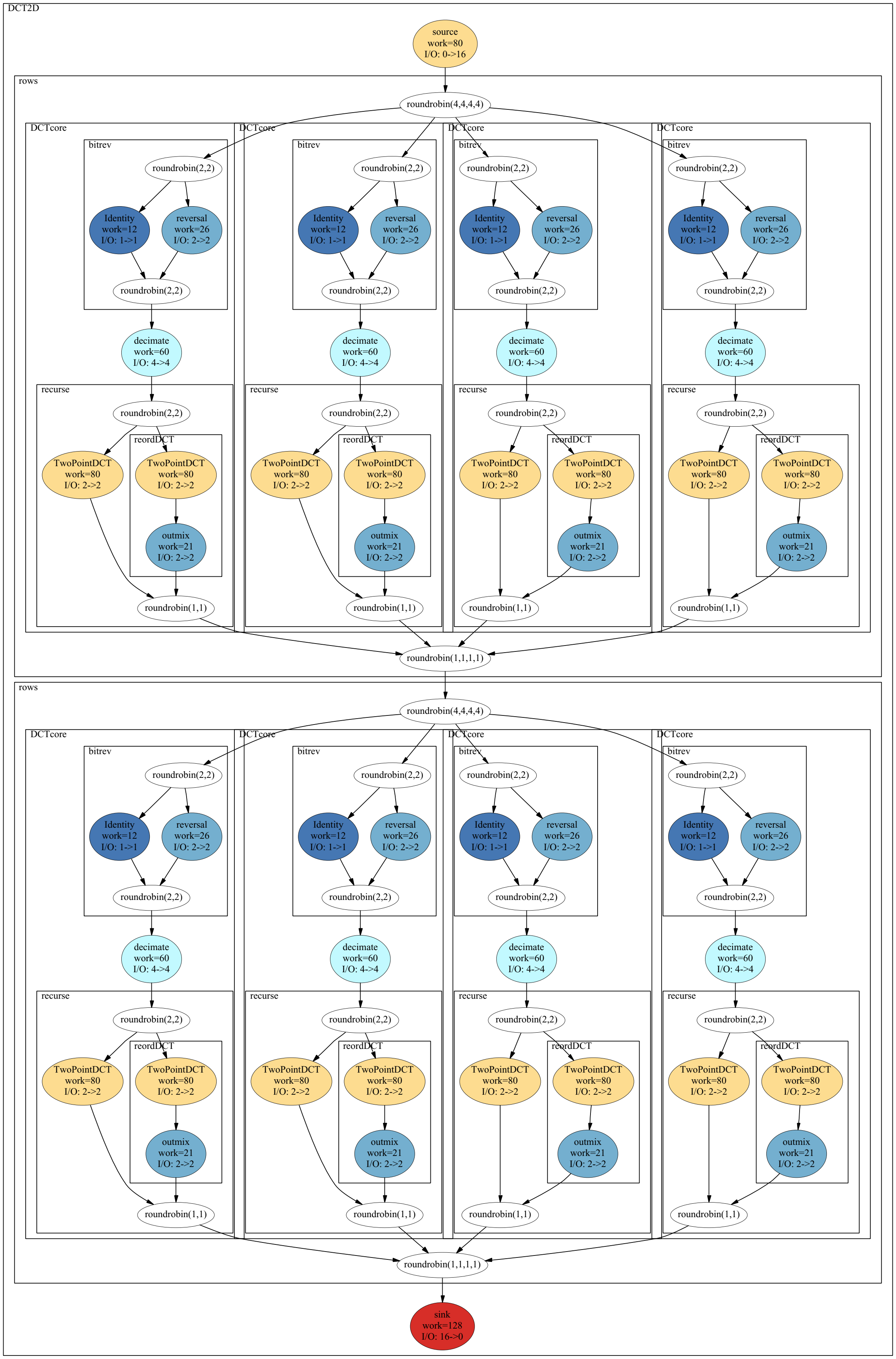

DCT2D
source work=80 I/O: 0->16
rows
roundrobin(4,4,4,4)
DCTcore
bitrev
roundrobin(2,2)
Identity work=12 I/O: 1->1
reversal work=26 I/O: 2->2
roundrobin(2,2)
decimate work=60 I/O: 4->4
recurse
roundrobin(2,2)
reordDCT
TwoPointDCT work=80 I/O: 2->2
TwoPointDCT work=80 I/O: 2->2
outmix work=21 I/O: 2->2
roundrobin(1,1)
DCTcore
bitrev
roundrobin(2,2)
Identity work=12 I/O: 1->1
reversal work=26 I/O: 2->2
roundrobin(2,2)
decimate work=60 I/O: 4->4
recurse
roundrobin(2,2)
reordDCT
TwoPointDCT work=80 I/O: 2->2
TwoPointDCT work=80 I/O: 2->2
outmix work=21 I/O: 2->2
roundrobin(1,1)
DCTcore
bitrev
roundrobin(2,2)
Identity work=12 I/O: 1->1
reversal work=26 I/O: 2->2
roundrobin(2,2)
decimate work=60 I/O: 4->4
recurse
roundrobin(2,2)
reordDCT
TwoPointDCT work=80 I/O: 2->2
TwoPointDCT work=80 I/O: 2->2
outmix work=21 I/O: 2->2
roundrobin(1,1)
DCTcore
bitrev
roundrobin(2,2)
Identity work=12 I/O: 1->1
reversal work=26 I/O: 2->2
roundrobin(2,2)
decimate work=60 I/O: 4->4
recurse
roundrobin(2,2)
reordDCT
TwoPointDCT work=80 I/O: 2->2
TwoPointDCT work=80 I/O: 2->2
outmix work=21 I/O: 2->2
roundrobin(1,1)
roundrobin(1,1,1,1)
rows
roundrobin(4,4,4,4)
DCTcore
bitrev
roundrobin(2,2)
Identity work=12 I/O: 1->1
reversal work=26 I/O: 2->2
roundrobin(2,2)
decimate work=60 I/O: 4->4
recurse
roundrobin(2,2)
reordDCT
TwoPointDCT work=80 I/O: 2->2
TwoPointDCT work=80 I/O: 2->2
outmix work=21 I/O: 2->2
roundrobin(1,1)
DCTcore
bitrev
roundrobin(2,2)
Identity work=12 I/O: 1->1
reversal work=26 I/O: 2->2
roundrobin(2,2)
decimate work=60 I/O: 4->4
recurse
roundrobin(2,2)
reordDCT
TwoPointDCT work=80 I/O: 2->2
TwoPointDCT work=80 I/O: 2->2
outmix work=21 I/O: 2->2
roundrobin(1,1)
DCTcore
bitrev
roundrobin(2,2)
Identity work=12 I/O: 1->1
reversal work=26 I/O: 2->2
roundrobin(2,2)
decimate work=60 I/O: 4->4
recurse
roundrobin(2,2)
reordDCT
TwoPointDCT work=80 I/O: 2->2
TwoPointDCT work=80 I/O: 2->2
outmix work=21 I/O: 2->2
roundrobin(1,1)
DCTcore
bitrev
roundrobin(2,2)
Identity work=12 I/O: 1->1
reversal work=26 I/O: 2->2
roundrobin(2,2)
decimate work=60 I/O: 4->4
recurse
roundrobin(2,2)
reordDCT
TwoPointDCT work=80 I/O: 2->2
TwoPointDCT work=80 I/O: 2->2
outmix work=21 I/O: 2->2
roundrobin(1,1)
roundrobin(1,1,1,1)
sink work=128 I/O: 16->0

Figure B-14: Stream graph for DCT2D (NxM, float).

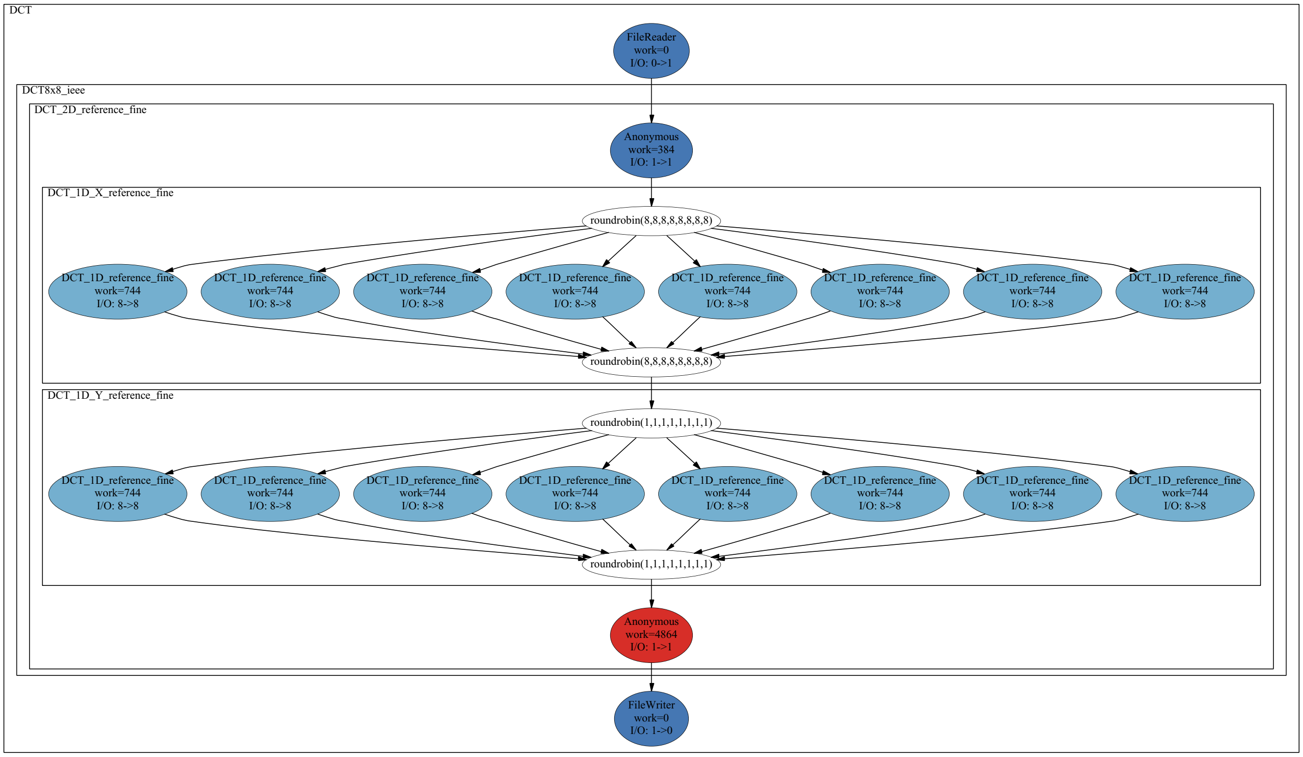


Figure B-15: Stream graph for DCT2D (NxN, int, reference).

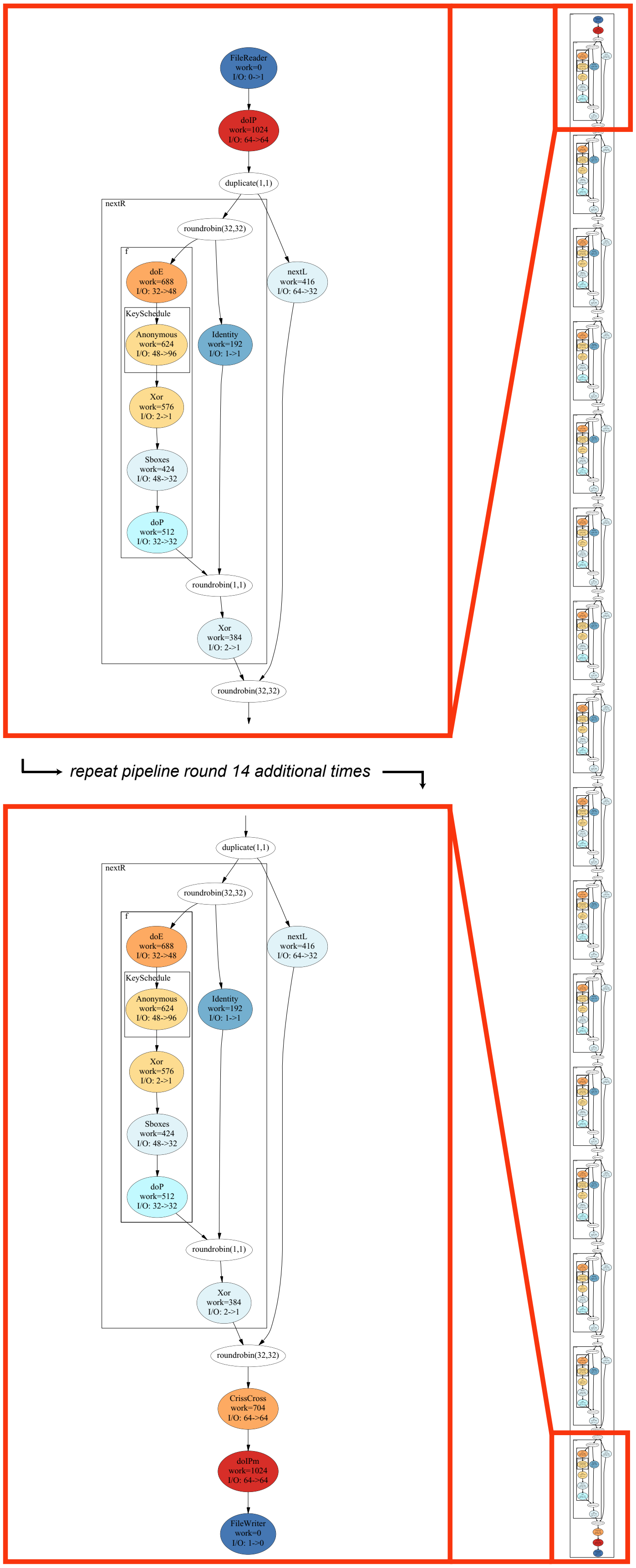


Figure B-16: Stream graph for DES.

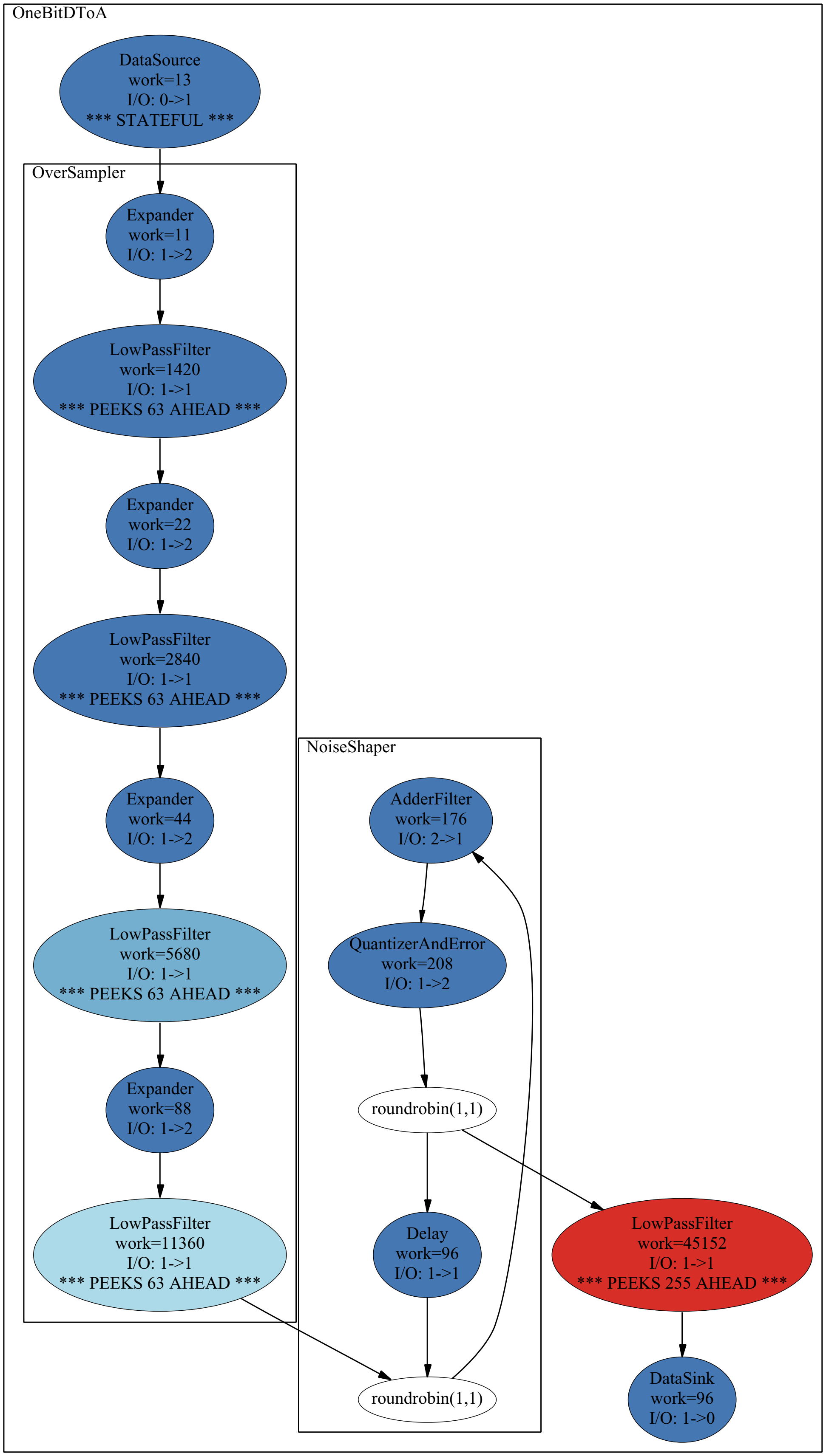

OneBitDToA
DataSource work=13 I/O: 0->1 *** STATEFUL ***
OverSampler
Expander work=11 I/O: 1->2
LowPassFilter work=1420 I/O: 1->1 *** PEEKS 63 AHEAD ***
Expander work=22 I/O: 1->2
LowPassFilter work=2840 I/O: 1->1 *** PEEKS 63 AHEAD ***
Expander work=44 I/O: 1->2
LowPassFilter work=5680 I/O: 1->1 *** PEEKS 63 AHEAD ***
Expander work=88 I/O: 1->2
LowPassFilter work=11360 I/O: 1->1 *** PEEKS 63 AHEAD ***
NoiseShaper
AdderFilter work=176 I/O: 2->1
QuantizerAndError work=208 I/O: 1->2
roundrobin(1,1)
Delay work=96 I/O: 1->1
roundrobin(1,1)
LowPassFilter work=45152 I/O: 1->1 *** PEEKS 255 AHEAD ***
DataSink work=96 I/O: 1->0

Figure B-17: Stream graph for DToA.

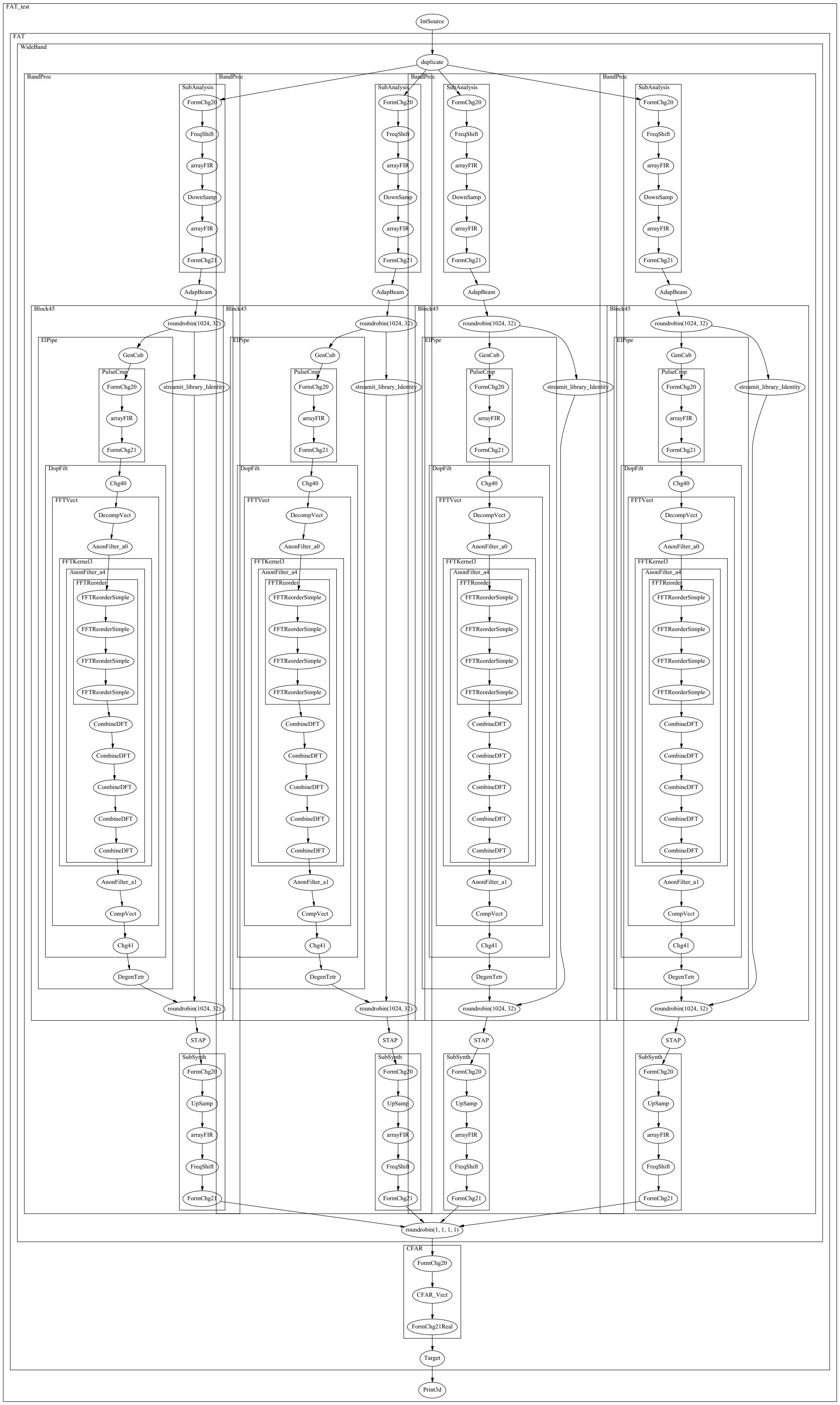


Figure B-18: Stream graph for FAT.

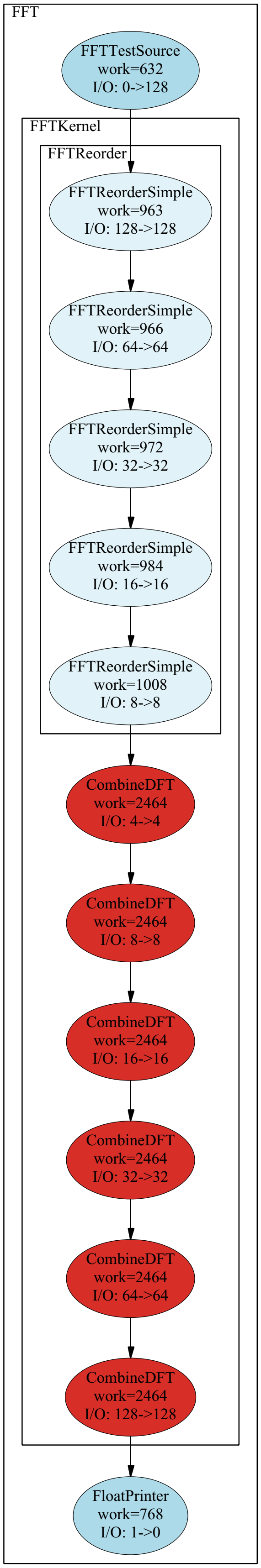


Figure B-19: Stream graph for FFT (coarse, default).

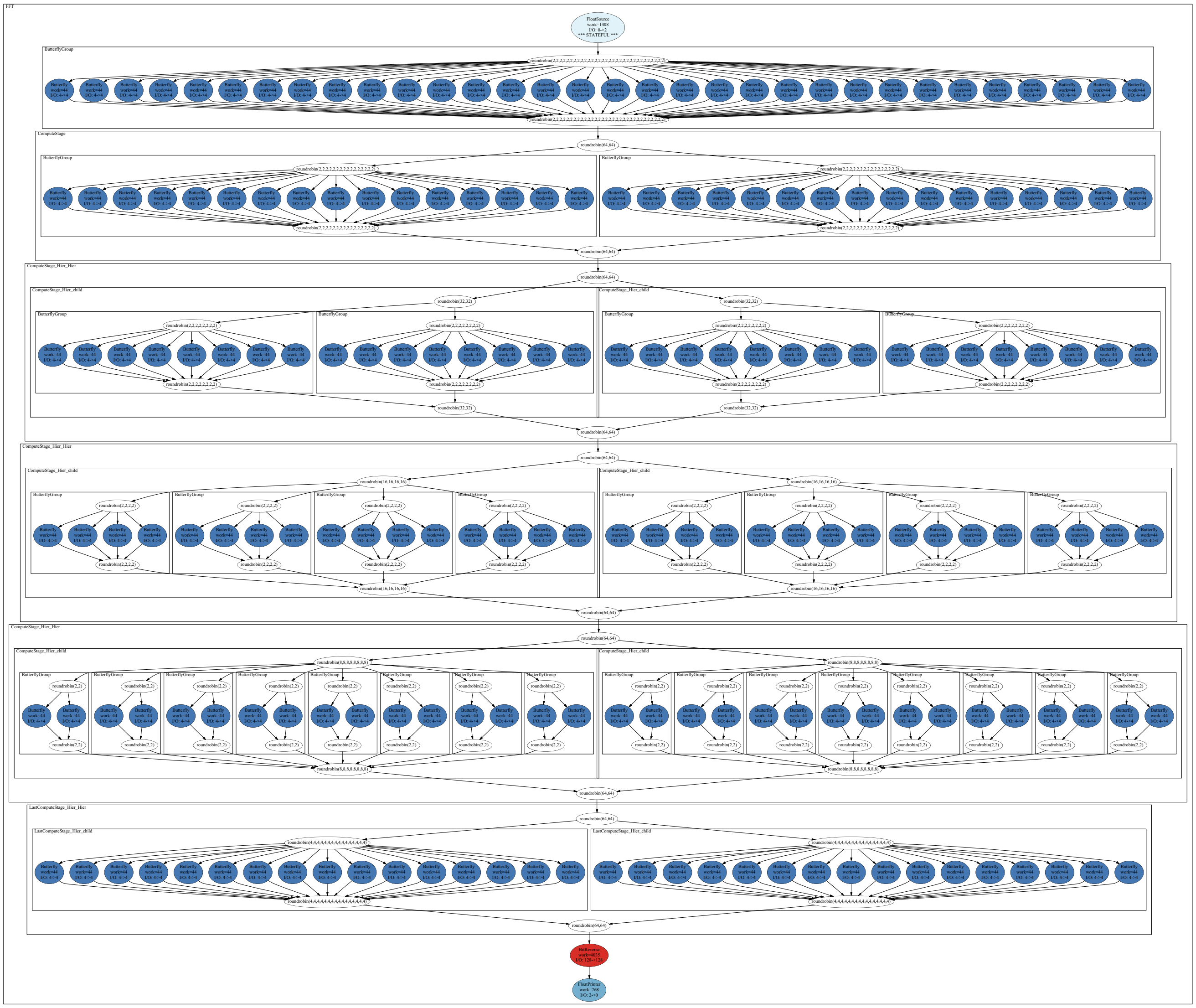


Figure B-20: Stream graph for FFT (fine 1).

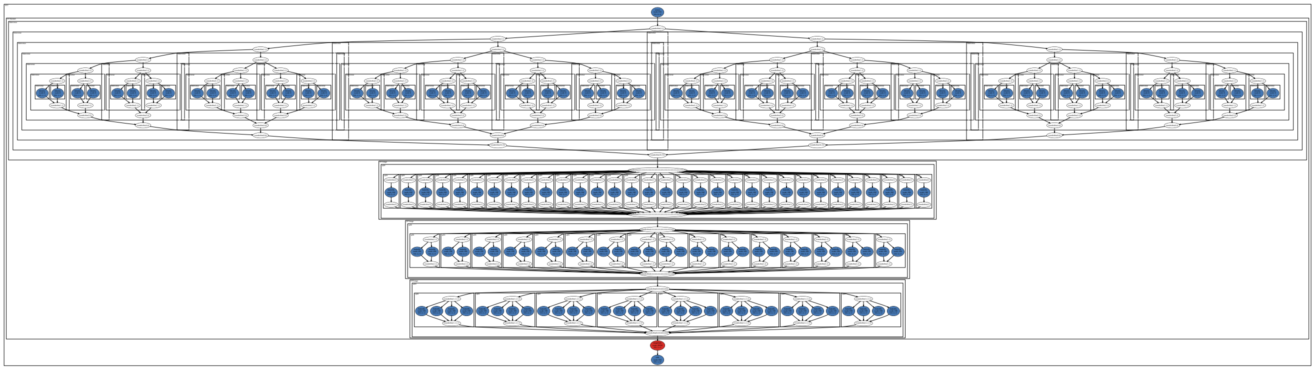


Figure B-21: Stream graph for FFT (fine 2).

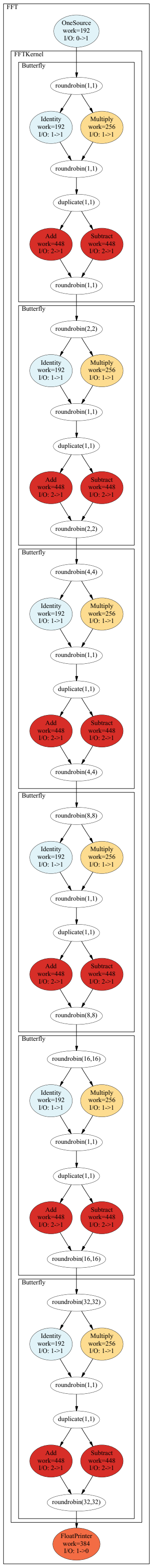


Figure B-22: Stream graph for FFT (medium).

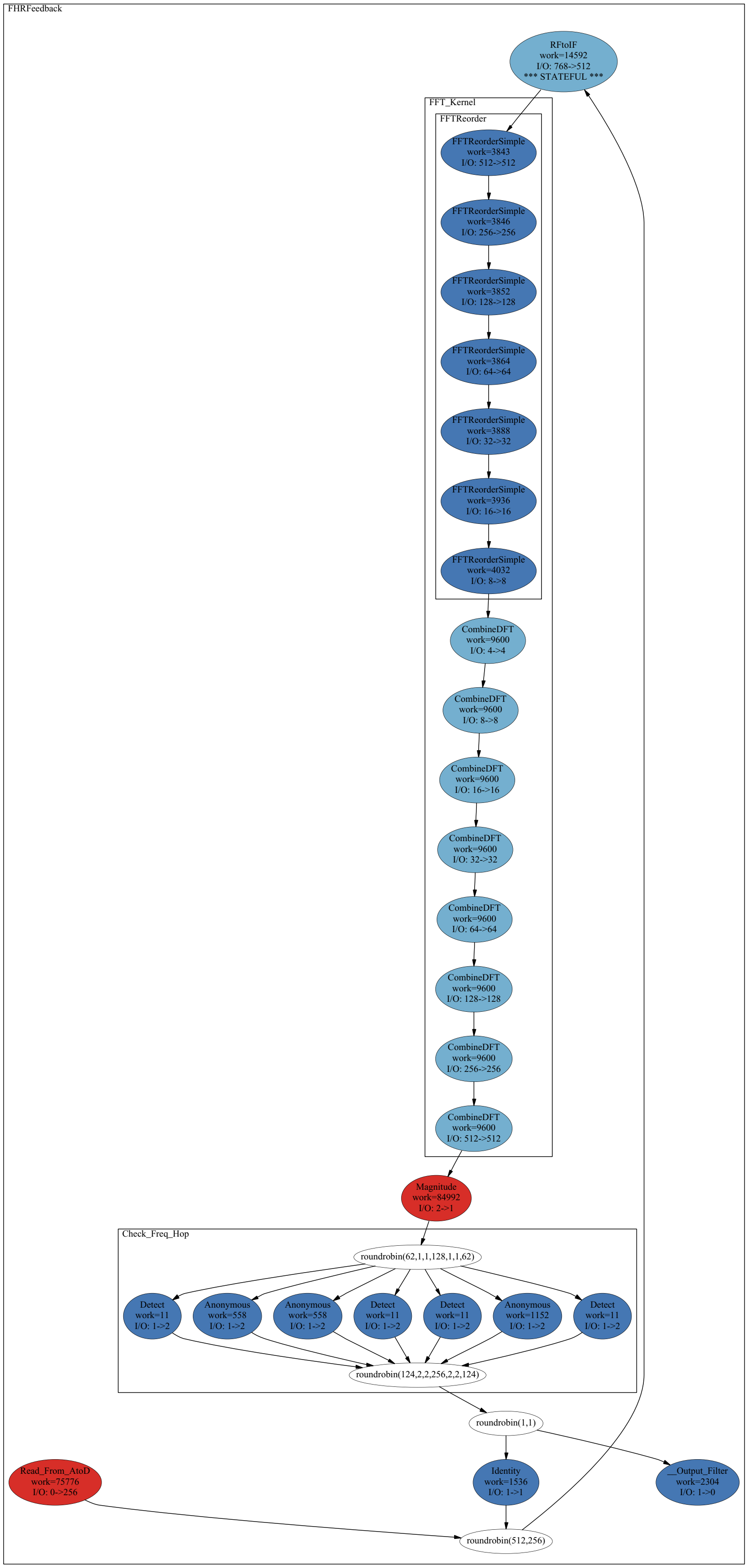


Figure B-23: Stream graph for FHR (feedback loop).

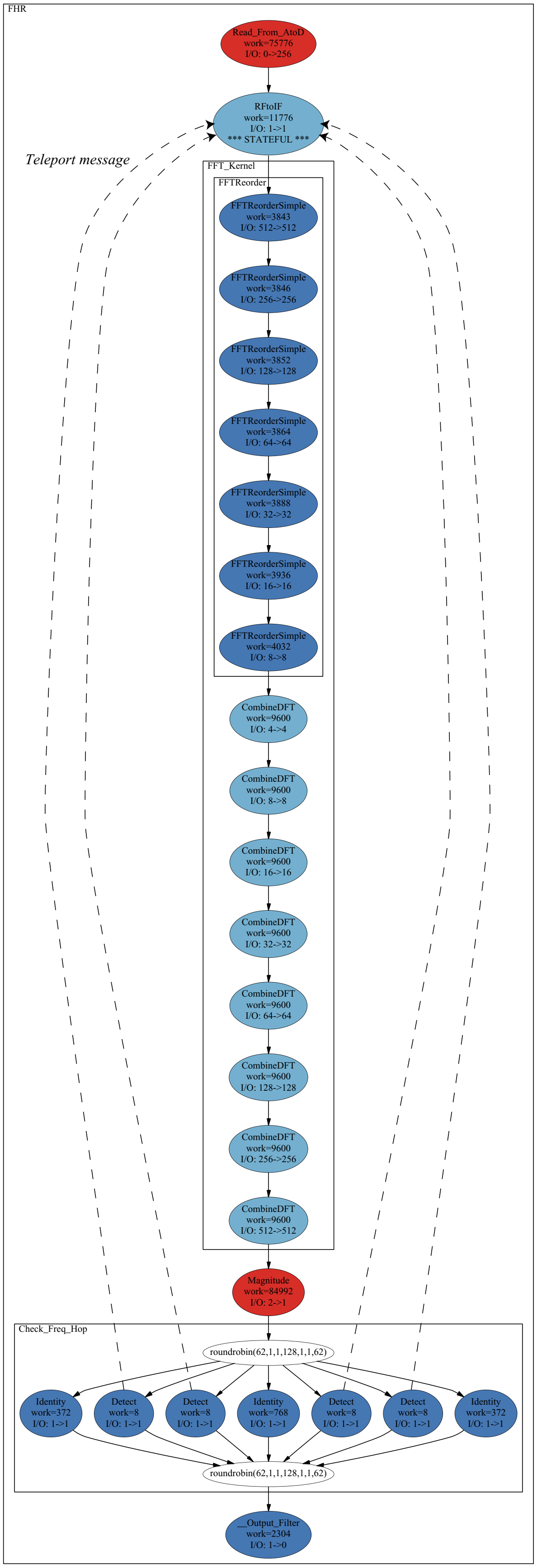


Figure B-24: Stream graph for FHR (teleport messaging).

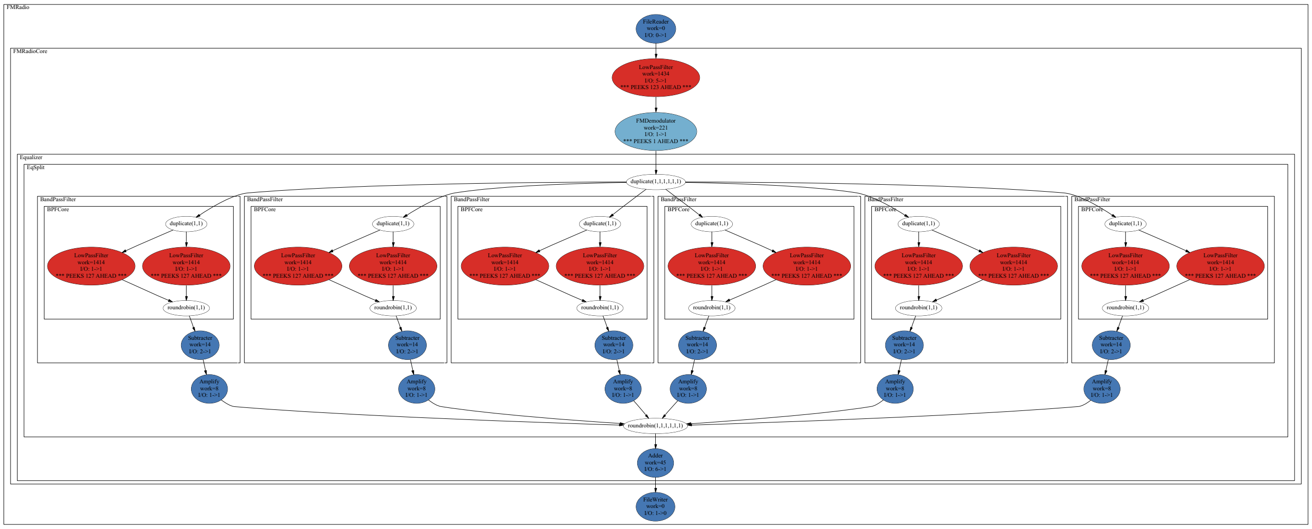


Figure B-25: Stream graph for FMRadio.

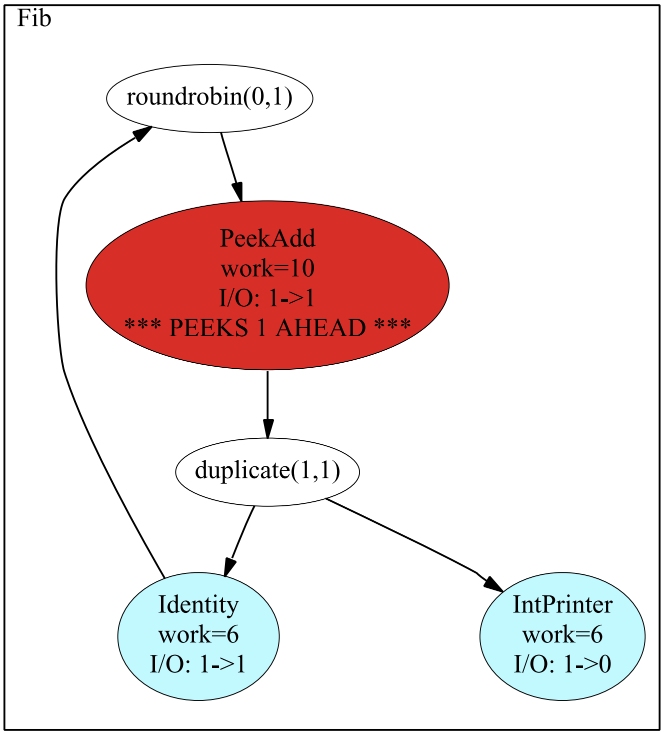


Figure B-26: Stream graph for Fib.

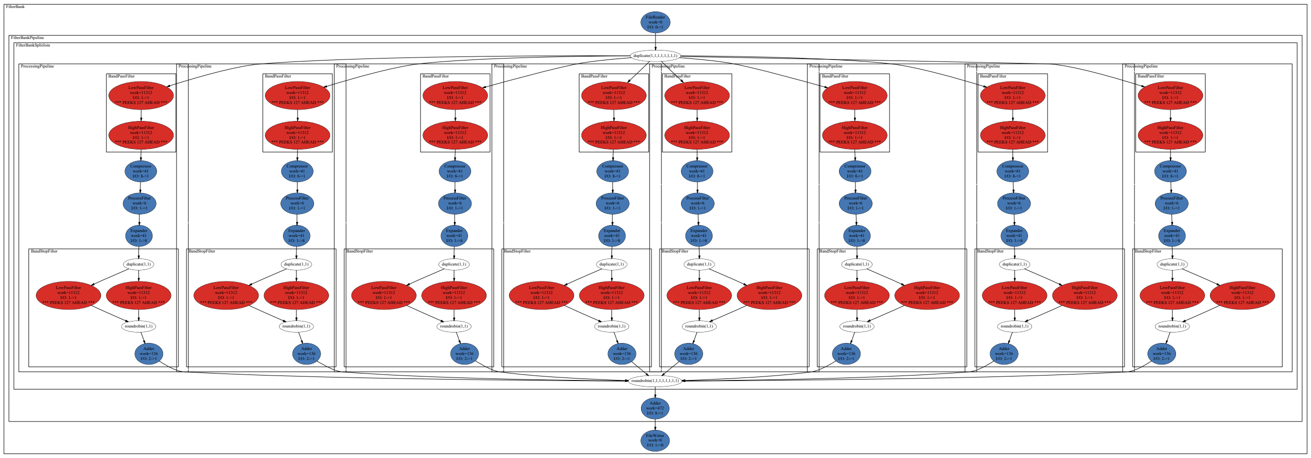


Figure B-27: Stream graph for FilterBank.

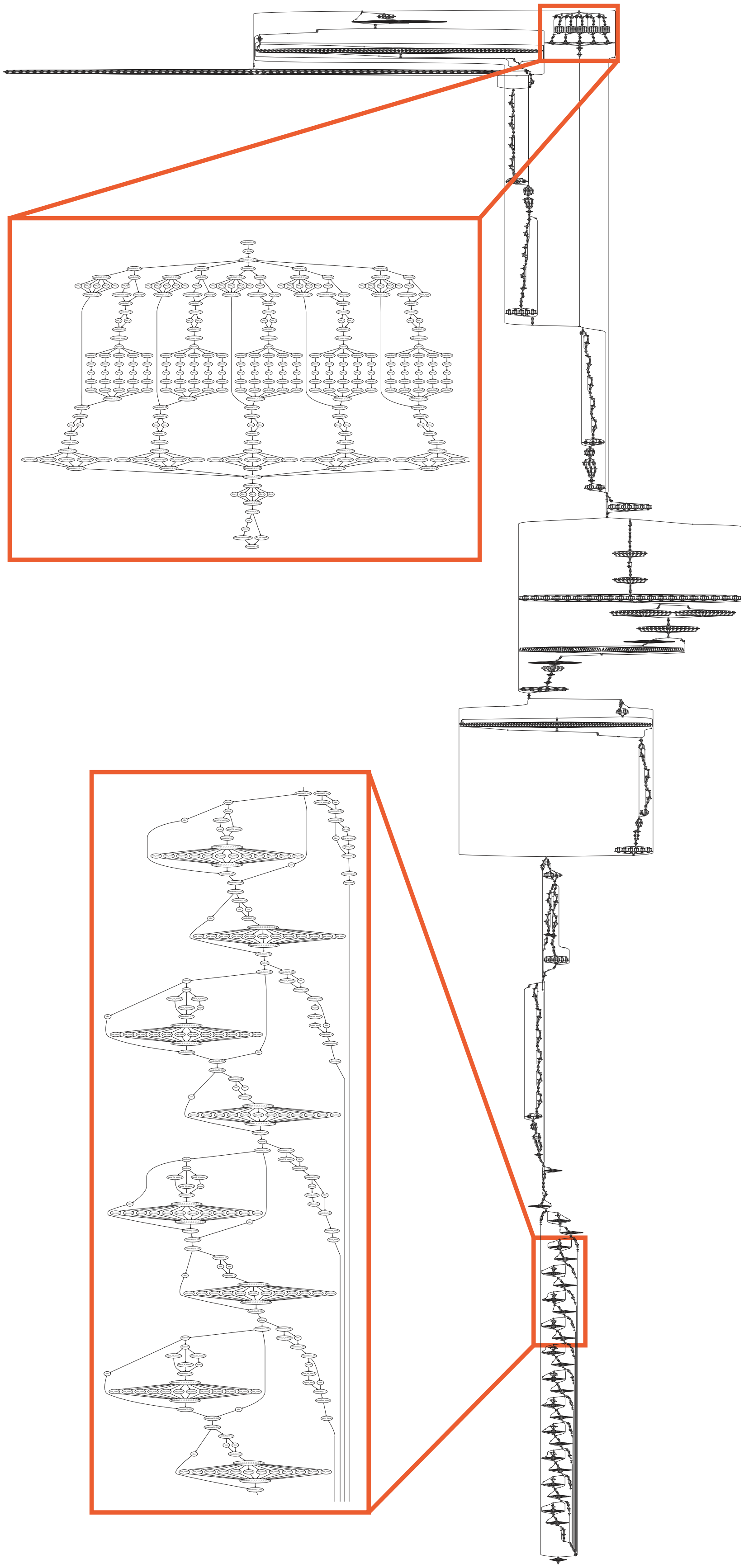


Figure B-28: Stream graph for GMTI.

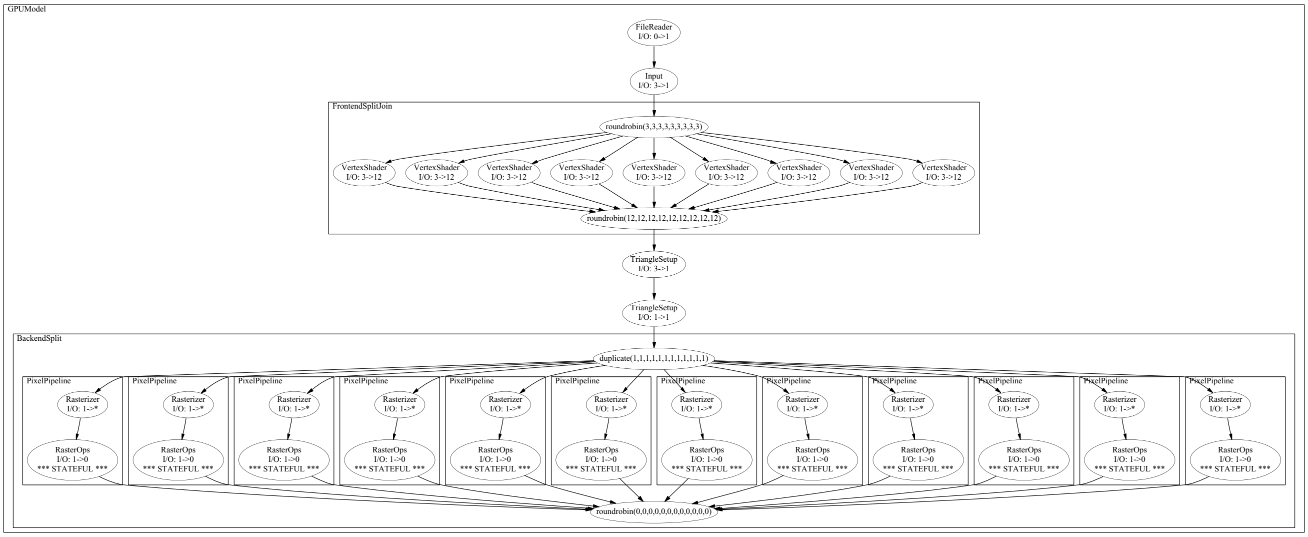


Figure B-29: Stream graph for GP - particle-system.

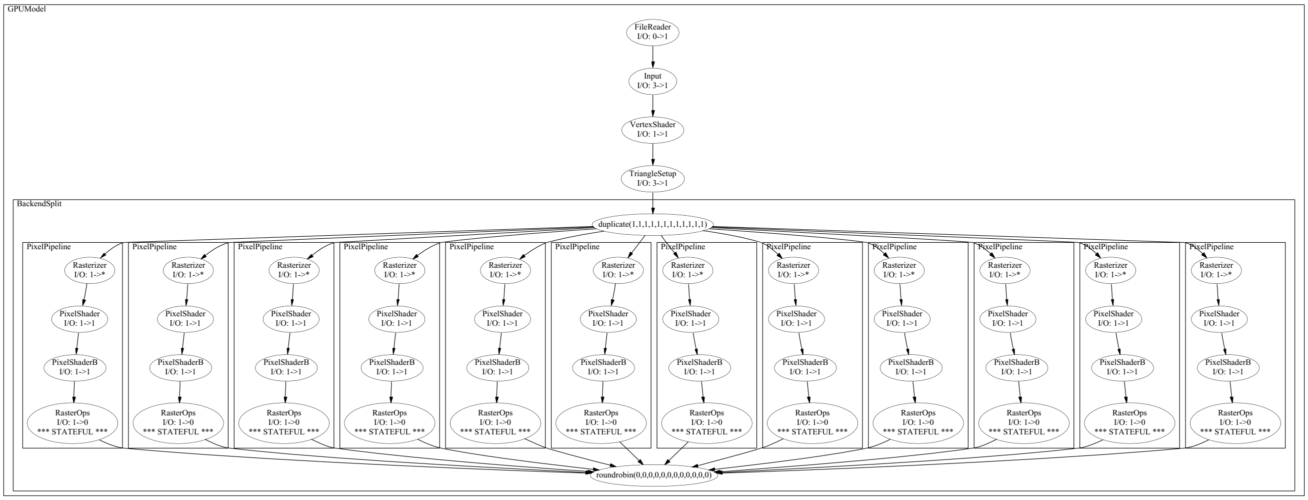


Figure B-30: Stream graph for GP - phong-shading.

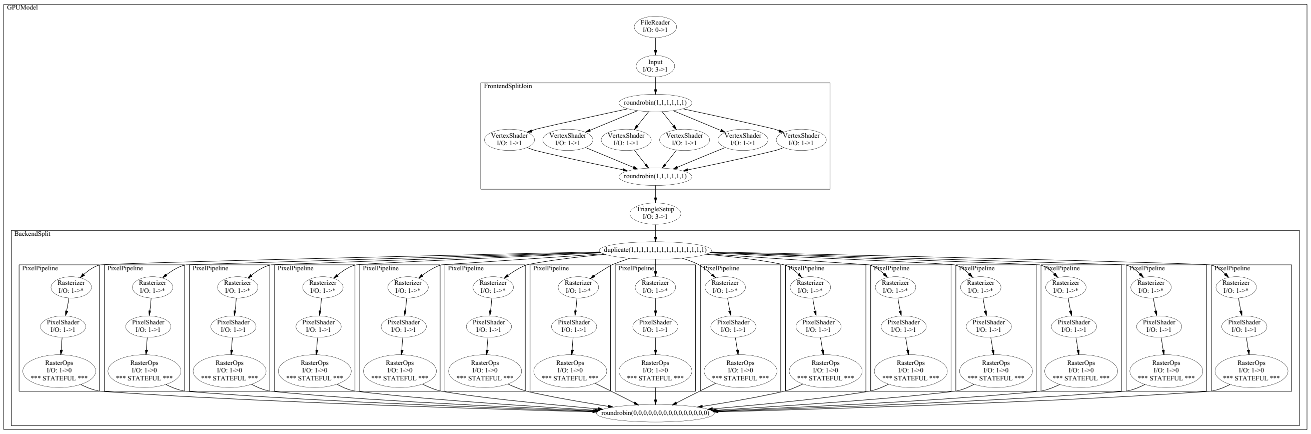


Figure B-31: Stream graph for GP - reference-version.

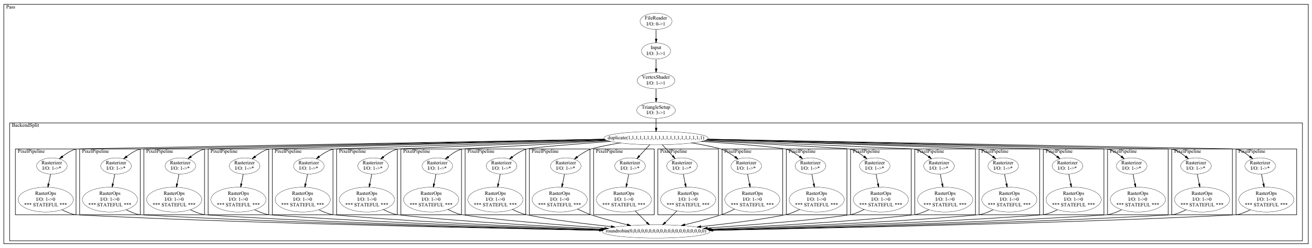


Figure B-32: Stream graph for GP - shadow-volumes.

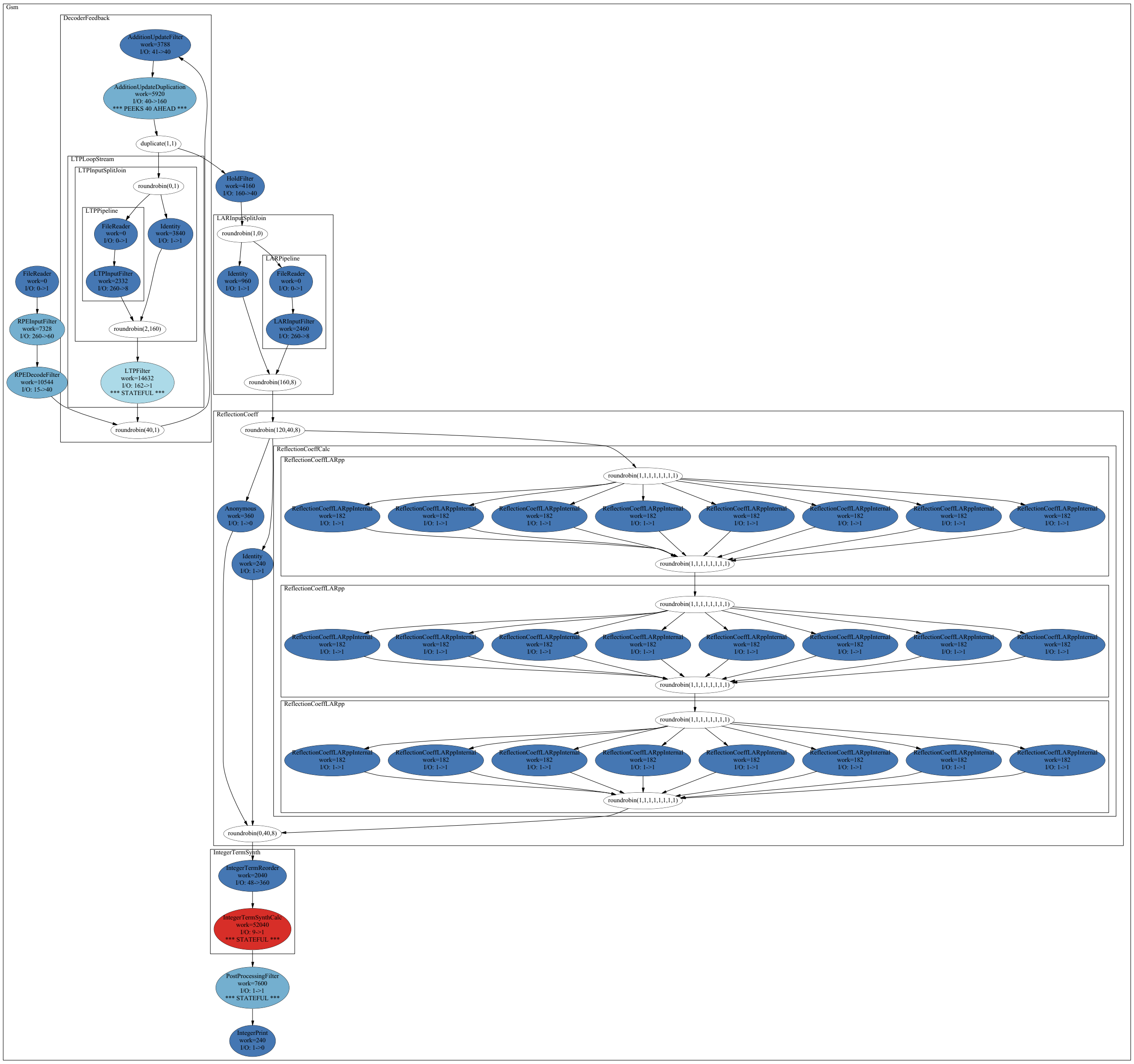

Gsm
DecoderFeedback
AdditionUpdateFilter work=3788 I/O: 41->40
AdditionUpdateDuplication work=5920 I/O: 40->160 *** PEEKS 40 AHEAD ***
duplicate(1,1)
LTPLoopStream
LTPInputSplitJoin
roundrobin(0,1)
LTPPipeline
FileReader work=0 I/O: 0->1
Identity work=3840 I/O: 1->1
LTPInputFilter work=2332 I/O: 260->8
roundrobin(2,160)
LTPFilter work=14632 I/O: 162->1 *** STATEFUL ***
roundrobin(40,1)
FileReader work=0 I/O: 0->1
RPEInputFilter work=7328 I/O: 260->60
RPEDecodeFilter work=10544 I/O: 15->40
HoldFilter work=4160 I/O: 160->40
LARInputSplitJoin
roundrobin(1,0)
Identity work=960 I/O: 1->1
LARPipeline
FileReader work=0 I/O: 0->1
LARInputFilter work=2460 I/O: 260->8
roundrobin(160,8)
ReflectionCoeff
roundrobin(120,40,8)
ReflectionCoeffCalc
ReflectionCoeffLARpp
roundrobin(1,1,1,1,1,1,1,1)
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
roundrobin(1,1,1,1,1,1,1,1)
Anonymous work=360 I/O: 1->0
Identity work=240 I/O: 1->1
ReflectionCoeffLARpp
roundrobin(1,1,1,1,1,1,1,1)
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
roundrobin(1,1,1,1,1,1,1,1)
ReflectionCoeffLARpp
roundrobin(1,1,1,1,1,1,1,1)
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
ReflectionCoeffLARppInternal work=182 I/O: 1->1
roundrobin(1,1,1,1,1,1,1,1)
roundrobin(0,40,8)
IntegerTermSynth
IntegerTermReorder work=2040 I/O: 48->360
IntegerTermSynthCalc work=52040 I/O: 9->1 *** STATEFUL ***
PostProcessingFilter work=7600 I/O: 1->1 *** STATEFUL ***
IntegerPrint work=240 I/O: 1->0

Figure B-33: Stream graph for GSM.

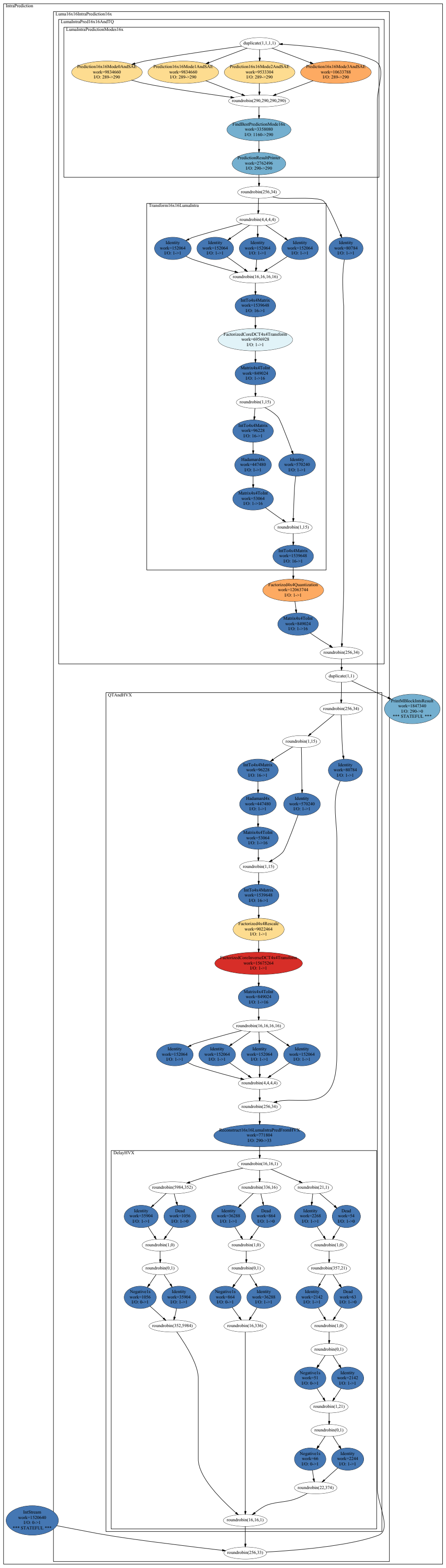

IntraPrediction
Luma16x16IntraPrediction16x
LumaIntraPred16x16AndTQ
LumaIntraPredictionModes16x
duplicate(1,1,1,1)
Prediction16x16Mode0AndSAE work=9834660 I/O: 289->290
Prediction16x16Mode1AndSAE work=9834660 I/O: 289->290
Prediction16x16Mode2AndSAE work=9533304 I/O: 289->290
Prediction16x16Mode3AndSAE work=10633788 I/O: 289->290
roundrobin(290,290,290,290)
FindBestPredictionMode16x work=3358080 I/O: 1160->290
PredictionResultPrinter work=2762496 I/O: 290->290
roundrobin(256,34)
Transform16x16LumaIntra
roundrobin(4,4,4,4)
Identity work=152064 I/O: 1->1
Identity work=152064 I/O: 1->1
Identity work=152064 I/O: 1->1
Identity work=152064 I/O: 1->1
Identity work=80784 I/O: 1->1
roundrobin(16,16,16,16)
IntTo4x4Matrix work=1539648 I/O: 16->1
FactorizedCoreDCT4x4Transform work=6956928 I/O: 1->1
Matrix4x4ToInt work=849024 I/O: 1->16
roundrobin(1,15)
IntTo4x4Matrix work=96228 I/O: 16->1
Hadamard4x work=447480 I/O: 1->1
Identity work=570240 I/O: 1->1
Matrix4x4ToInt work=53064 I/O: 1->16
roundrobin(1,15)
IntTo4x4Matrix work=1539648 I/O: 16->1
Factorized4x4Quantization work=12063744 I/O: 1->1
Matrix4x4ToInt work=849024 I/O: 1->16
roundrobin(256,34)
duplicate(1,1)
PrintMBlockIntsResult work=1847340 I/O: 290->0 *** STATEFUL ***
QTAndHVX
roundrobin(256,34)
roundrobin(1,15)
IntTo4x4Matrix work=96228 I/O: 16->1
Identity work=80784 I/O: 1->1
Hadamard4x work=447480 I/O: 1->1
Identity work=570240 I/O: 1->1
Matrix4x4ToInt work=53064 I/O: 1->16
roundrobin(1,15)
IntTo4x4Matrix work=1539648 I/O: 16->1
Factorized4x4Rescale work=9022464 I/O: 1->1
FactorizedCoreInverseDCT4x4Transform work=15675264 I/O: 1->1
Matrix4x4ToInt work=849024 I/O: 1->16
roundrobin(16,16,16,16)
Identity work=152064 I/O: 1->1
Identity work=152064 I/O: 1->1
Identity work=152064 I/O: 1->1
Identity work=152064 I/O: 1->1
roundrobin(4,4,4,4)
roundrobin(256,34)
Reconstruct16x16LumaIntraPredFromHVX work=771804 I/O: 290->33
DelayHVX
roundrobin(16,16,1)
roundrobin(5984,352)
roundrobin(336,16)
roundrobin(21,1)
Identity work=35904 I/O: 1->1
Dead work=1056 I/O: 1->0
Identity work=36288 I/O: 1->1
Dead work=864 I/O: 1->0
Identity work=2268 I/O: 1->1
Dead work=54 I/O: 1->0
roundrobin(1,0)
roundrobin(1,0)
roundrobin(1,0)
roundrobin(0,1)
roundrobin(0,1)
roundrobin(357,21)
Negative1s work=1056 I/O: 0->1
Identity work=35904 I/O: 1->1
Negative1s work=864 I/O: 0->1
Identity work=36288 I/O: 1->1
Identity work=2142 I/O: 1->1
Dead work=63 I/O: 1->0
roundrobin(352,5984)
roundrobin(16,336)
roundrobin(1,0)
roundrobin(0,1)
Negative1s work=51 I/O: 0->1
Identity work=2142 I/O: 1->1
roundrobin(1,21)
roundrobin(0,1)
Negative1s work=66 I/O: 0->1
Identity work=2244 I/O: 1->1
roundrobin(22,374)
IntStream work=1520640 I/O: 0->1 *** STATEFUL ***
roundrobin(16,16,1)
roundrobin(256,33)

Figure B-34: Stream graph for H264 subset.

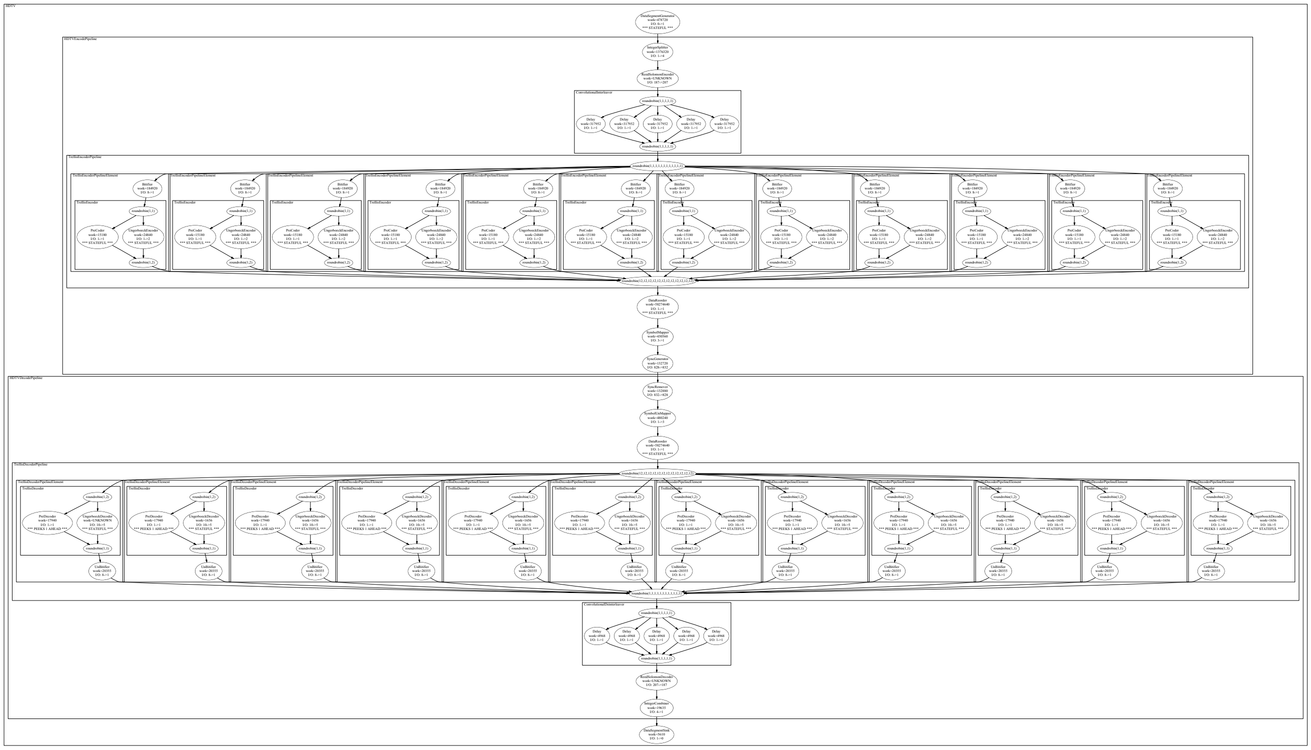

HDTV
DataSegmentGenerator work=478720 I/O: 0->1 *** STATEFUL ***
HDTVEncodePipeline
IntegerSplitter work=1376320 I/O: 1->4
ReedSolomonEncoder work=UNKNOWN I/O: 187->207
ConvolutionalInterleaver
roundrobin(1,1,1,1,1)
Delay work=317952 I/O: 1->1
roundrobin(1,1,1,1,1)
TrellisEncoderPipeline
roundrobin(1,1,1,1,1,1,1,1,1,1,1,1)
TrellisEncoderPipelineElement
Bitifier work=184920 I/O: 8->1
TrellisEncoder
roundrobin(1,1)
PreCoder work=15180 I/O: 1->1 *** STATEFUL ***
UngerboeckEncoder work=24840 I/O: 1->2 *** STATEFUL ***
roundrobin(1,2)
roundrobin(12,12,12,12,12,12,12,12,12,12,12,12)
DataReorder work=58274640 I/O: 1->1 *** STATEFUL ***
SymbolMapper work=430560 I/O: 3->1
SyncGenerator work=132720 I/O: 828->832
HDTVDecodePipeline
SyncRemover work=132880 I/O: 832->828
SymbolUnMapper work=480240 I/O: 1->3
DataReorder work=58274640 I/O: 1->1 *** STATEFUL ***
TrellisDecoderPipeline
roundrobin(12,12,12,12,12,12,12,12,12,12,12,12)
TrellisDecoderPipelineElement
TrellisDecoder
roundrobin(1,2)
PreDecoder work=17940 I/O: 1->1 *** PEEKS 1 AHEAD ***
UngerboeckDecoder work=UNKNOWN I/O: 10->5 *** STATEFUL ***
UngerboeckDecoder work=1656 I/O: 10->5 *** STATEFUL ***
roundrobin(1,1)
UnBitifier work=20355 I/O: 8->1
roundrobin(1,1,1,1,1,1,1,1,1,1,1,1)
ConvolutionalDeinterleaver
roundrobin(1,1,1,1,1)
Delay work=4968 I/O: 1->1
roundrobin(1,1,1,1,1)
ReedSolomonDecoder work=UNKNOWN I/O: 207->187
IntegerCombiner work=19635 I/O: 4->1
DataSegmentSink work=5610 I/O: 1->0

Figure B-35: Stream graph for HDTV.

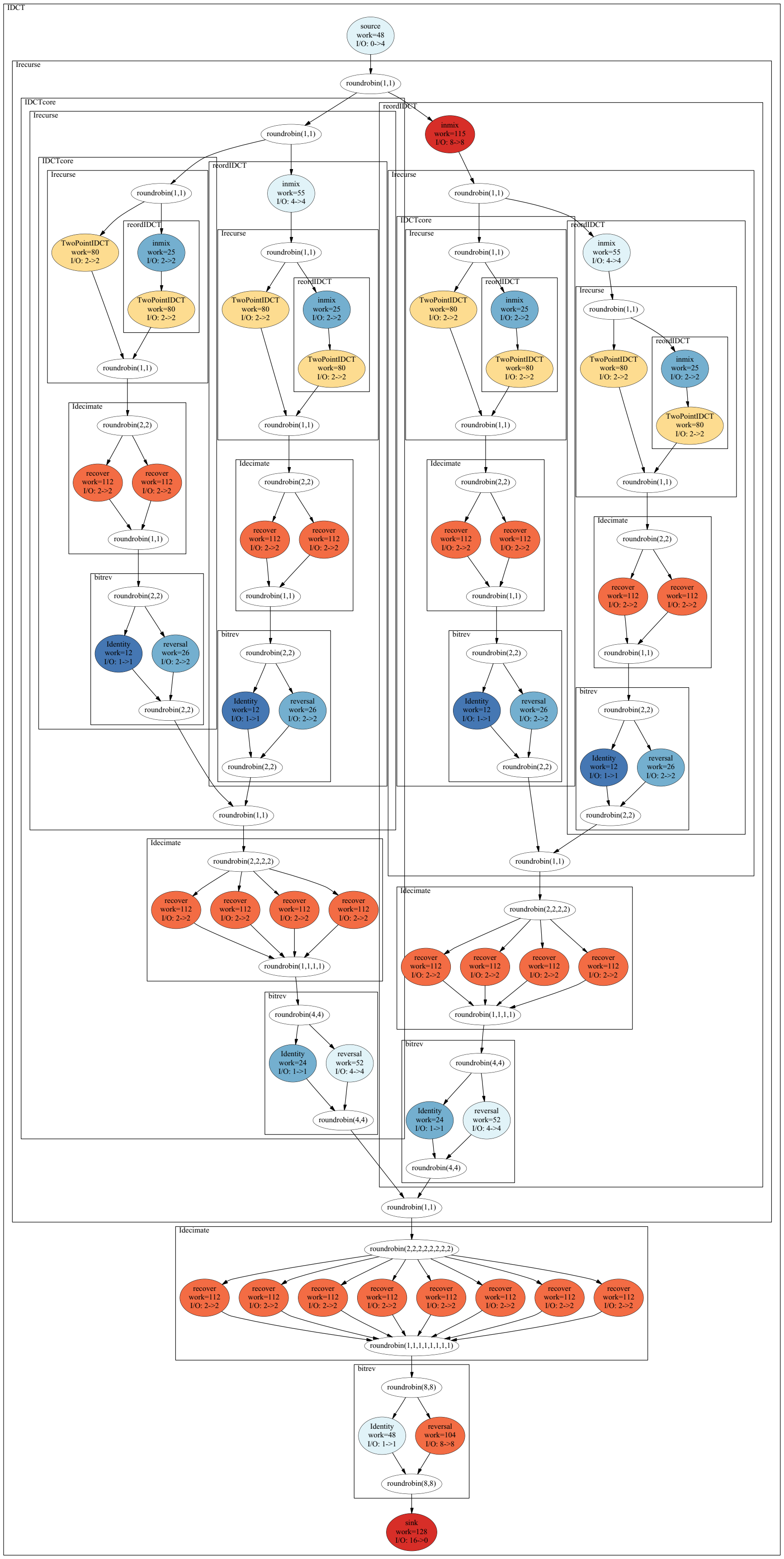

IDCT
source work=48 I/O: 0->4
Irecurse
roundrobin(1,1)
IDCTcore
Irecurse
roundrobin(1,1)
reordIDCT
inmix work=115 I/O: 8->8
IDCTcore
Irecurse
roundrobin(1,1)
TwoPointIDCT work=80 I/O: 2->2
reordIDCT
inmix work=25 I/O: 2->2
TwoPointIDCT work=80 I/O: 2->2
roundrobin(1,1)
Idecimate
roundrobin(2,2)
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
roundrobin(1,1)
bitrev
roundrobin(2,2)
Identity work=12 I/O: 1->1
reversal work=26 I/O: 2->2
roundrobin(2,2)
reordIDCT
inmix work=55 I/O: 4->4
Irecurse
roundrobin(1,1)
TwoPointIDCT work=80 I/O: 2->2
reordIDCT
inmix work=25 I/O: 2->2
TwoPointIDCT work=80 I/O: 2->2
roundrobin(1,1)
Idecimate
roundrobin(2,2)
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
roundrobin(1,1)
bitrev
roundrobin(2,2)
Identity work=12 I/O: 1->1
reversal work=26 I/O: 2->2
roundrobin(2,2)
roundrobin(1,1)
Idecimate
roundrobin(2,2,2,2)
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
roundrobin(1,1,1,1)
bitrev
roundrobin(4,4)
Identity work=24 I/O: 1->1
reversal work=52 I/O: 4->4
roundrobin(4,4)
Irecurse
roundrobin(1,1)
IDCTcore
Irecurse
roundrobin(1,1)
TwoPointIDCT work=80 I/O: 2->2
reordIDCT
inmix work=25 I/O: 2->2
TwoPointIDCT work=80 I/O: 2->2
roundrobin(1,1)
Idecimate
roundrobin(2,2)
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
roundrobin(1,1)
bitrev
roundrobin(2,2)
Identity work=12 I/O: 1->1
reversal work=26 I/O: 2->2
roundrobin(2,2)
reordIDCT
inmix work=55 I/O: 4->4
Irecurse
roundrobin(1,1)
TwoPointIDCT work=80 I/O: 2->2
reordIDCT
inmix work=25 I/O: 2->2
TwoPointIDCT work=80 I/O: 2->2
roundrobin(1,1)
Idecimate
roundrobin(2,2)
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
roundrobin(1,1)
bitrev
roundrobin(2,2)
Identity work=12 I/O: 1->1
reversal work=26 I/O: 2->2
roundrobin(2,2)
roundrobin(1,1)
Idecimate
roundrobin(2,2,2,2)
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
roundrobin(1,1,1,1)
bitrev
roundrobin(4,4)
Identity work=24 I/O: 1->1
reversal work=52 I/O: 4->4
roundrobin(4,4)
roundrobin(1,1)
Idecimate
roundrobin(2,2,2,2,2,2,2,2)
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
recover work=112 I/O: 2->2
roundrobin(1,1,1,1,1,1,1,1)
bitrev
roundrobin(8,8)
Identity work=48 I/O: 1->1
reversal work=104 I/O: 8->8
roundrobin(8,8)
sink work=128 I/O: 16->0

Figure B-36: Stream graph for IDCT (float).

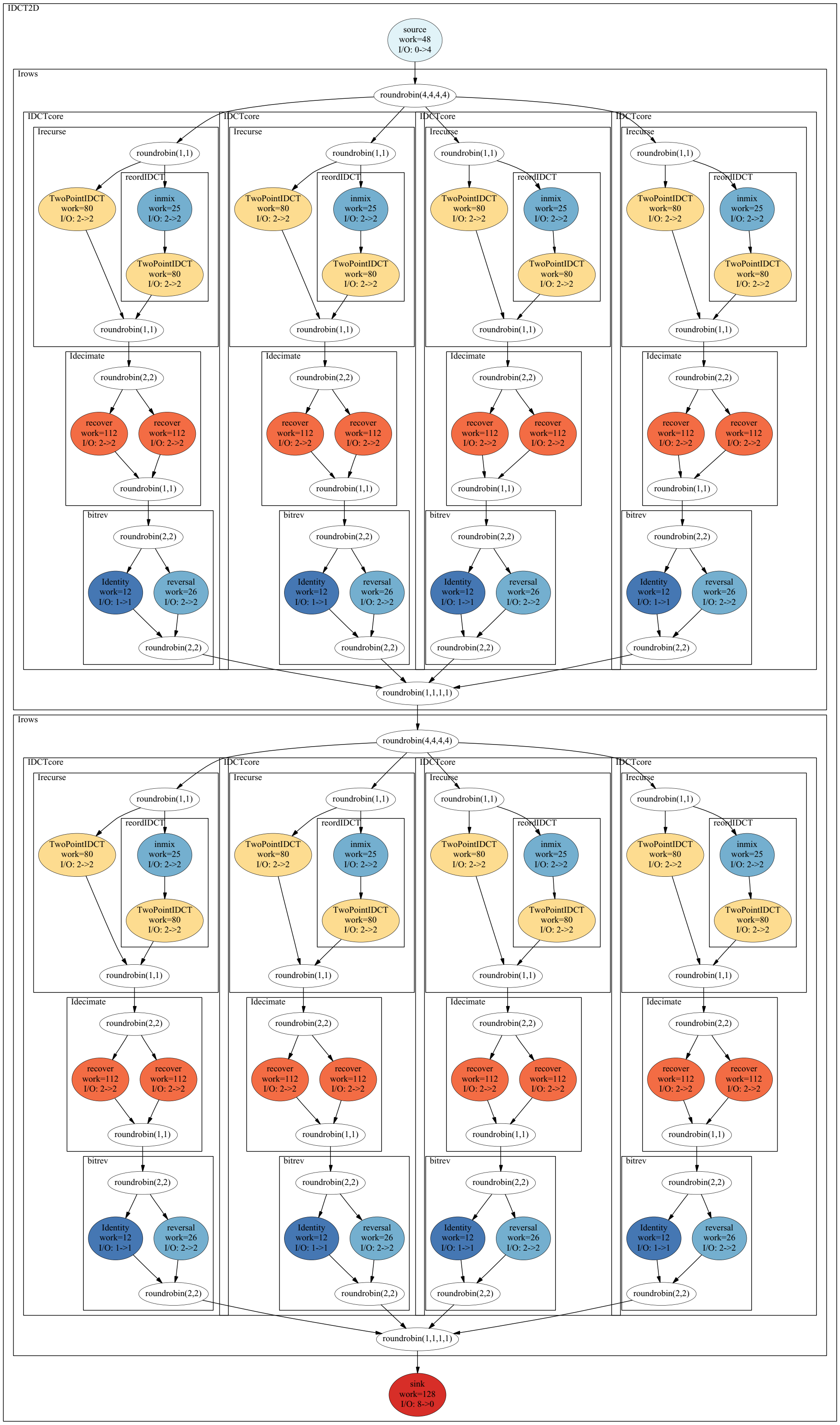


Figure B-37: Stream graph for IDCT2D (NxM-float).

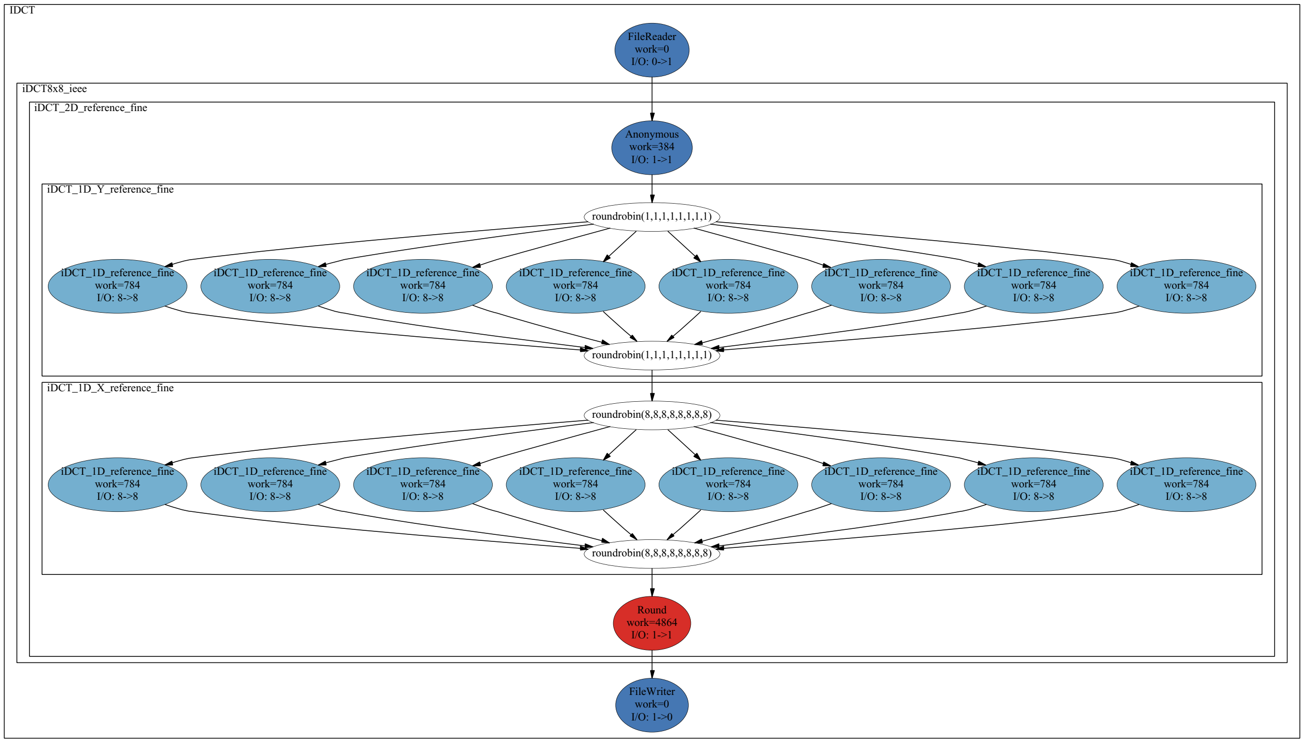

IDCT
FileReader work=0 I/O: 0->1
iDCT8x8_ieee
iDCT_2D_reference_fine
Anonymous work=384 I/O: 1->1
iDCT_1D_Y_reference_fine
roundrobin(1,1,1,1,1,1,1,1)
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
roundrobin(1,1,1,1,1,1,1,1)
iDCT_1D_X_reference_fine
roundrobin(8,8,8,8,8,8,8,8)
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
iDCT_1D_reference_fine work=784 I/O: 8->8
roundrobin(8,8,8,8,8,8,8,8)
Round work=4864 I/O: 1->1
FileWriter work=0 I/O: 1->0

Figure B-38: Stream graph for IDCT2D (NxN, int, reference).

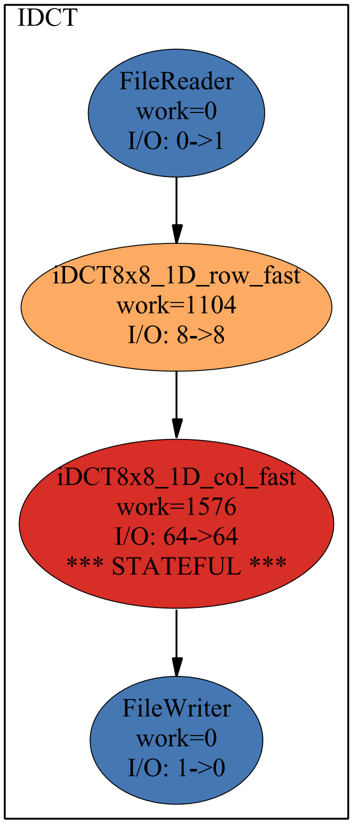


Figure B-39: Stream graph for IDCT2D (8x8, int, coarse).

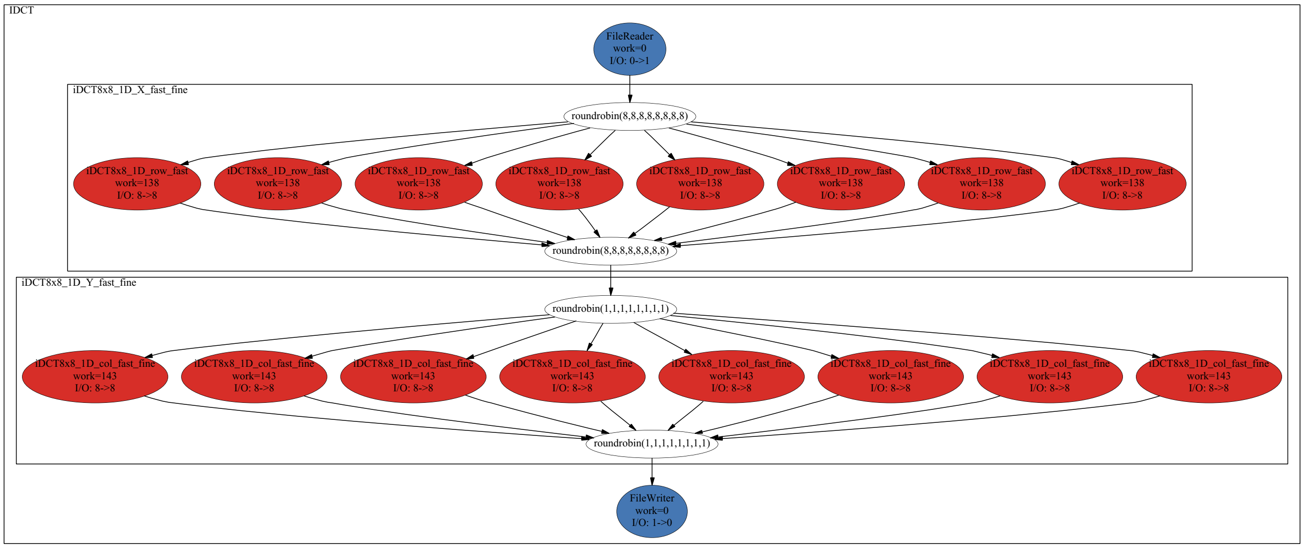


Figure B-40: Stream graph for IDCT2D (8x8, int, fine).

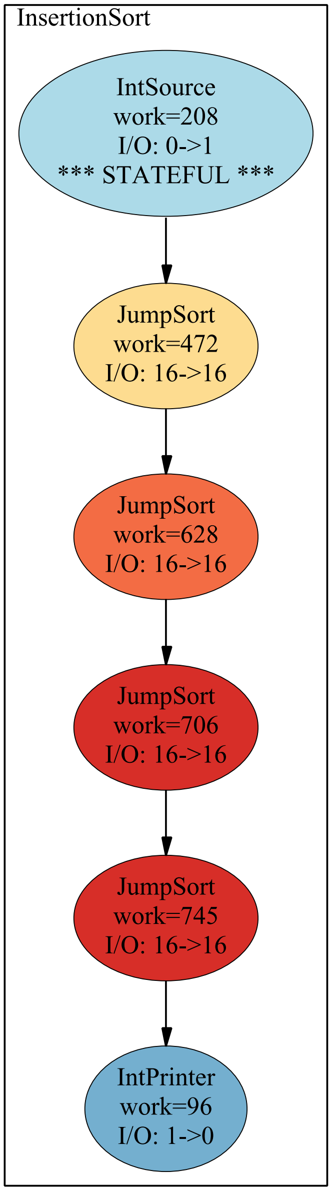


Figure B-41: Stream graph for InsertionSort.

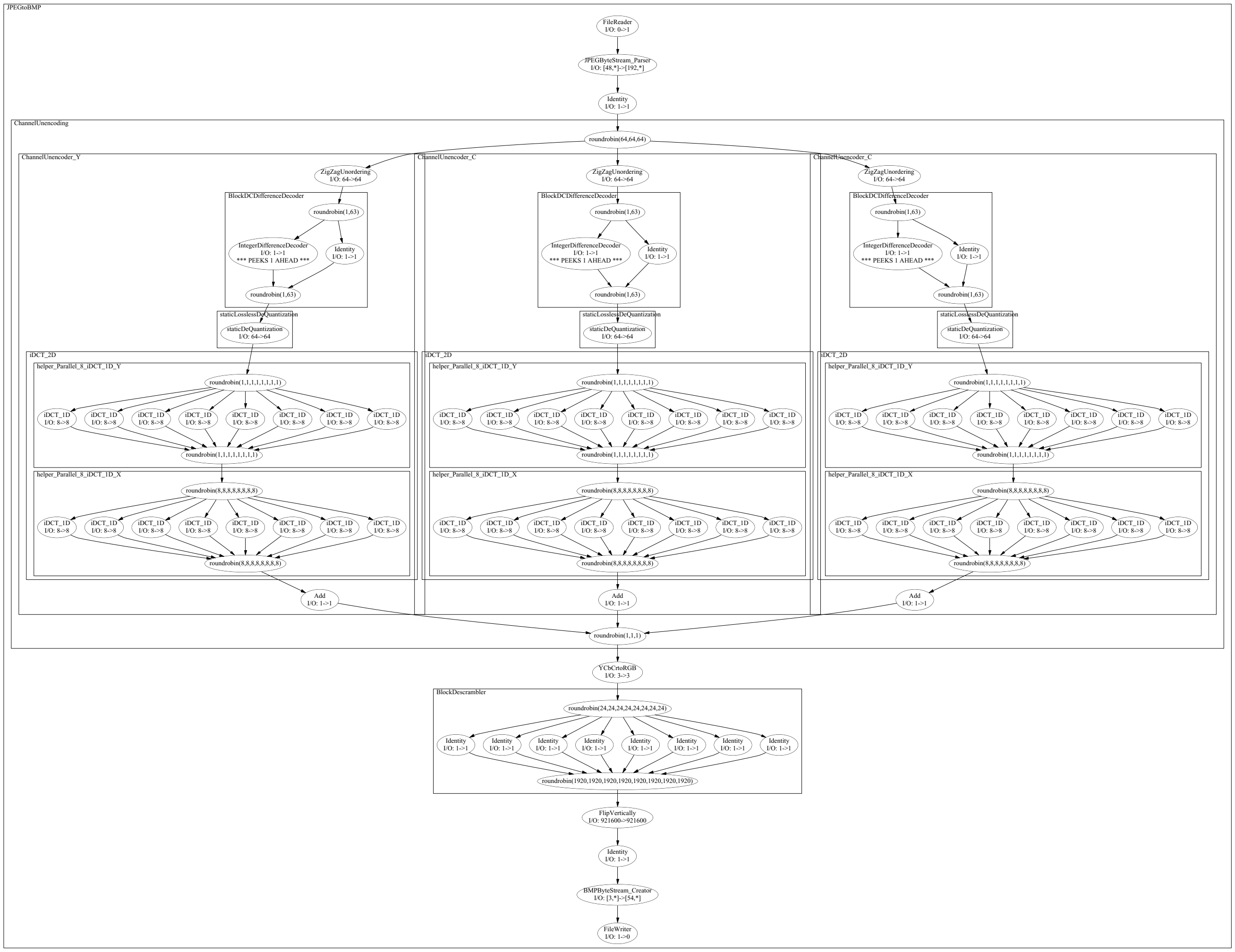

JPEGtoBMP
FileReader I/O: 0->1
JPEGByteStream_Parser I/O: [48,*]->[192,*]
Identity I/O: 1->1
ChannelUnencoding
roundrobin(64,64,64)
ChannelUnencoder_Y
ChannelUnencoder_C
ChannelUnencoder_C
ZigZagUnordering I/O: 64->64
BlockDCDifferenceDecoder
roundrobin(1,63)
IntegerDifferenceDecoder I/O: 1->1 *** PEEKS 1 AHEAD ***
Identity I/O: 1->1
roundrobin(1,63)
staticLosslessDeQuantization
staticDeQuantization I/O: 64->64
iDCT_2D
helper_Parallel_8_iDCT_1D_Y
roundrobin(1,1,1,1,1,1,1,1)
iDCT_1D I/O: 8->8
roundrobin(1,1,1,1,1,1,1,1)
helper_Parallel_8_iDCT_1D_X
roundrobin(8,8,8,8,8,8,8,8)
iDCT_1D I/O: 8->8
roundrobin(8,8,8,8,8,8,8,8)
Add I/O: 1->1
roundrobin(1,1,1)
YCbCrtoRGB I/O: 3->3
BlockDescrambler
roundrobin(24,24,24,24,24,24,24,24)
Identity I/O: 1->1
roundrobin(1920,1920,1920,1920,1920,1920,1920,1920)
FlipVertically I/O: 921600->921600
Identity I/O: 1->1
BMPByteStream_Creator I/O: [3,*]->[54,*]
FileWriter I/O: 1->0

Figure B-42: Stream graph for JPEG decoder.

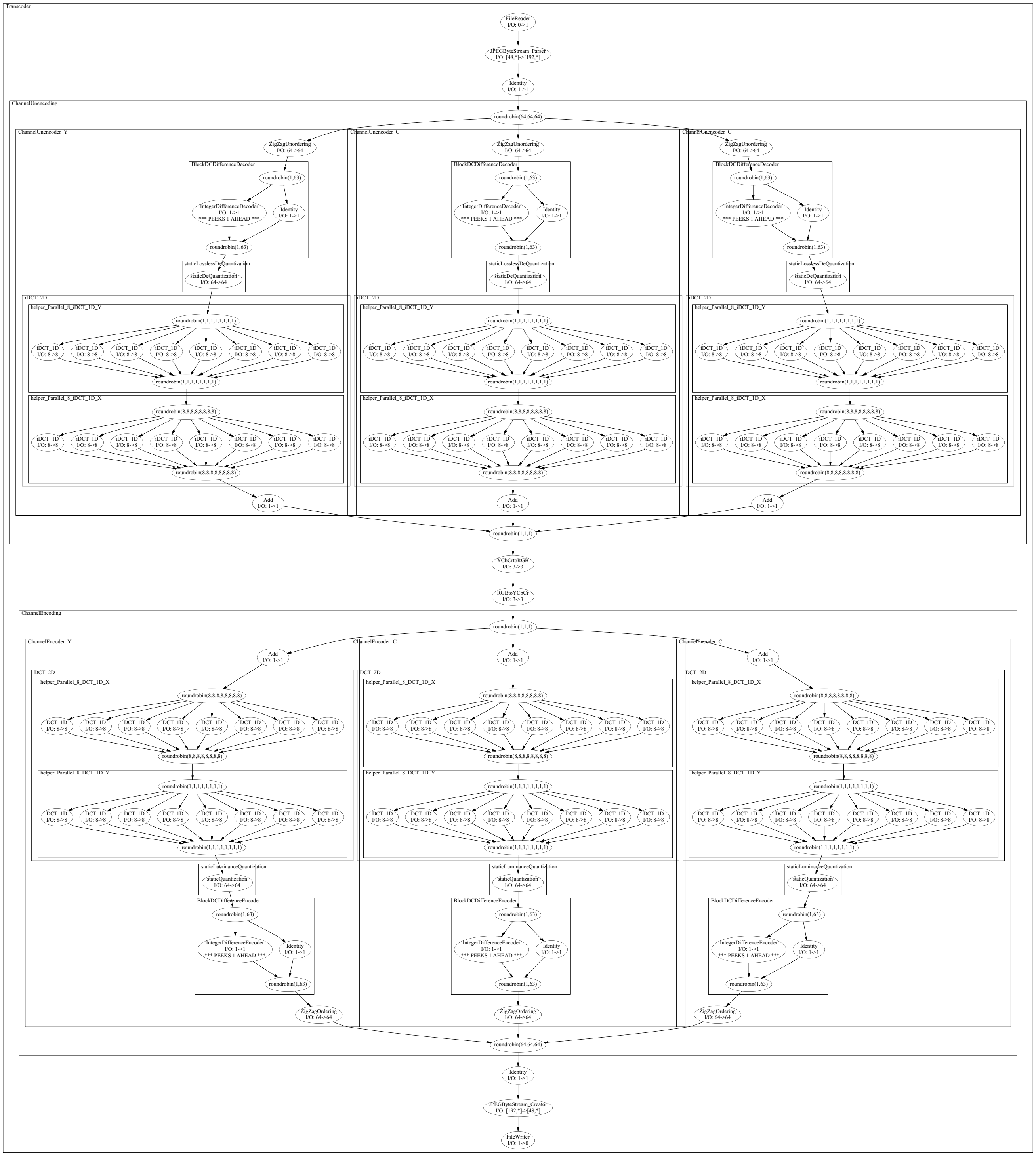


Figure B-43: Stream graph for JPEG transcoder.

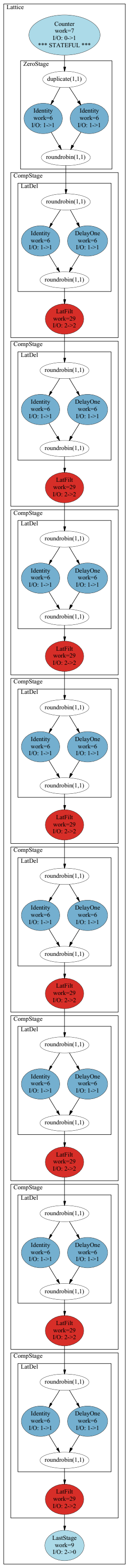


Figure B-44: Stream graph for Lattice.

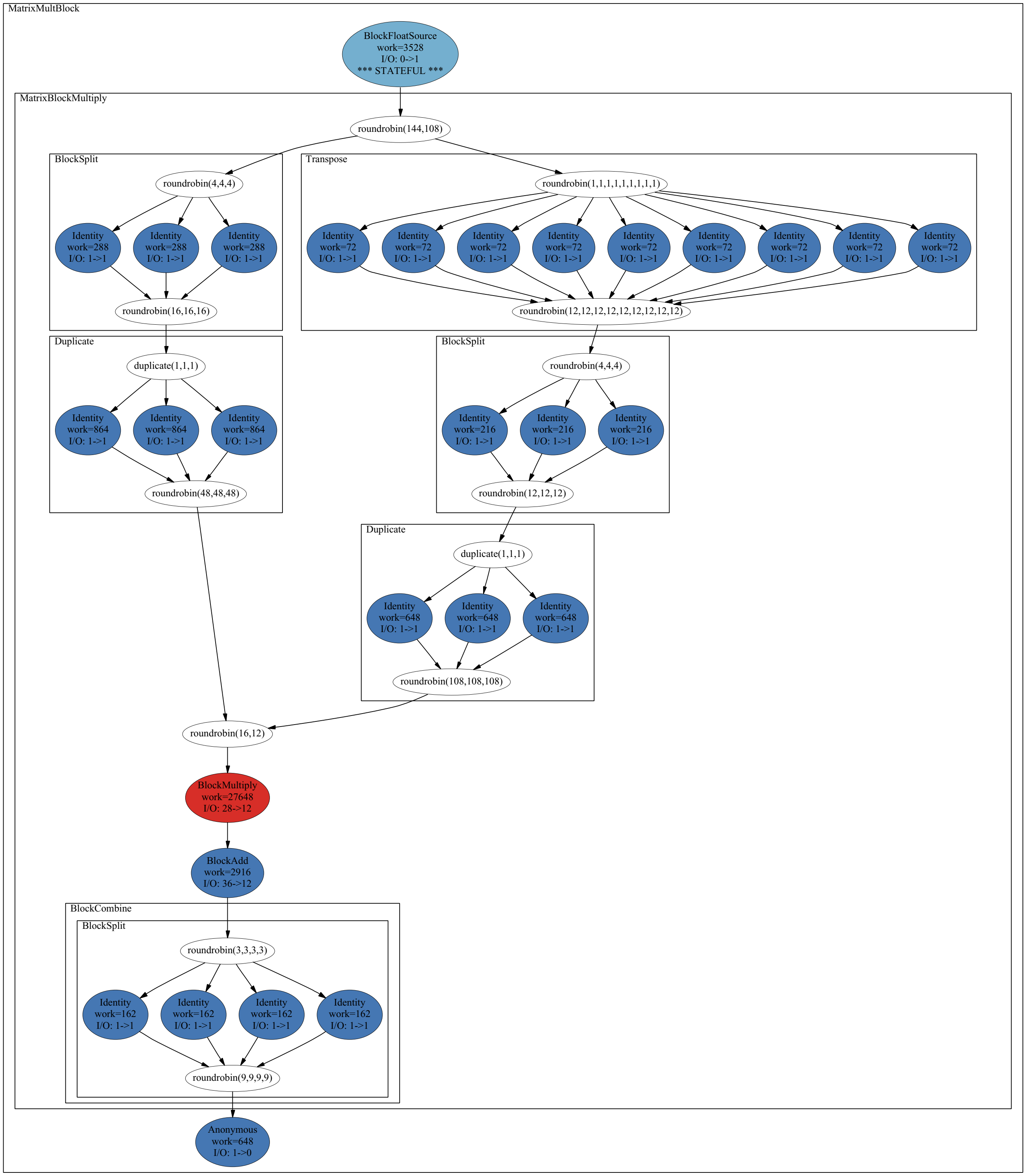


Figure B-45: Stream graph for MatrixMult (coarse).

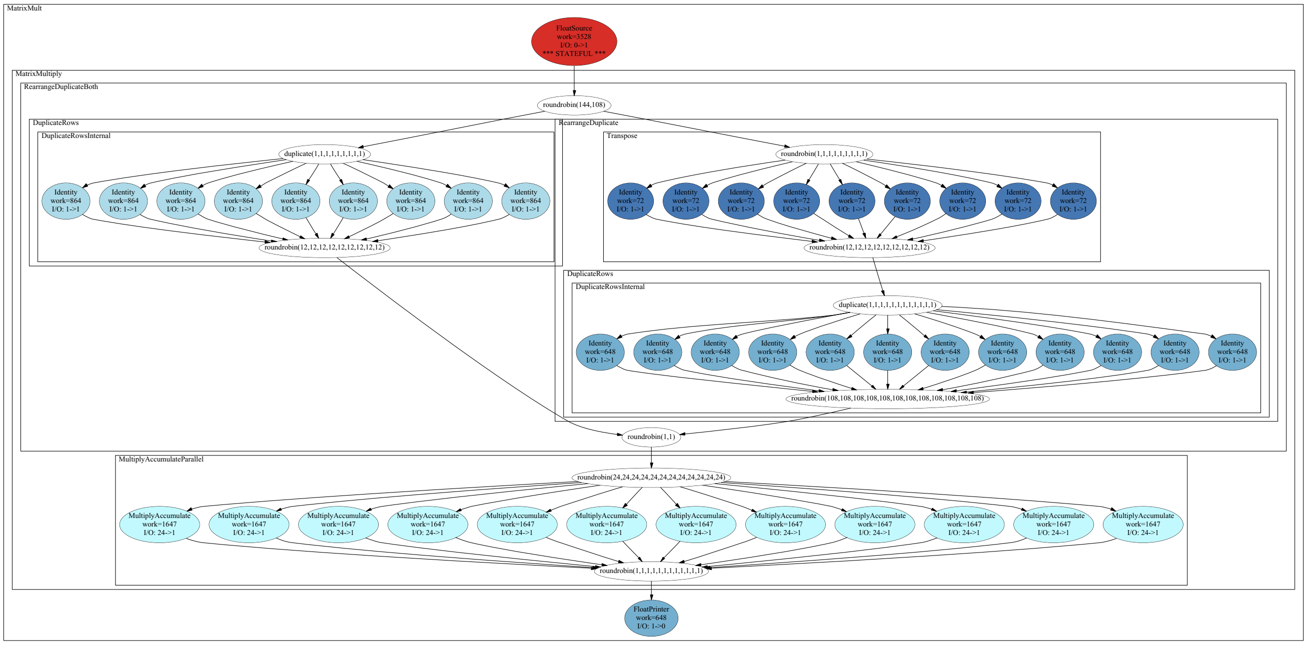

MatrixMult
FloatSource work=3528 I/O: 0->1 *** STATEFUL ***
MatrixMultiply
RearrangeDuplicateBoth
roundrobin(144,108)
DuplicateRows
DuplicateRowsInternal
duplicate(1,1,1,1,1,1,1,1,1)
Identity work=864 I/O: 1->1
Identity work=864 I/O: 1->1
Identity work=864 I/O: 1->1
Identity work=864 I/O: 1->1
Identity work=864 I/O: 1->1
Identity work=864 I/O: 1->1
Identity work=864 I/O: 1->1
Identity work=864 I/O: 1->1
Identity work=864 I/O: 1->1
roundrobin(12,12,12,12,12,12,12,12,12)
RearrangeDuplicate
Transpose
roundrobin(1,1,1,1,1,1,1,1,1)
Identity work=72 I/O: 1->1
Identity work=72 I/O: 1->1
Identity work=72 I/O: 1->1
Identity work=72 I/O: 1->1
Identity work=72 I/O: 1->1
Identity work=72 I/O: 1->1
Identity work=72 I/O: 1->1
Identity work=72 I/O: 1->1
Identity work=72 I/O: 1->1
roundrobin(12,12,12,12,12,12,12,12,12)
DuplicateRows
DuplicateRowsInternal
duplicate(1,1,1,1,1,1,1,1,1,1,1,1)
Identity work=648 I/O: 1->1
Identity work=648 I/O: 1->1
Identity work=648 I/O: 1->1
Identity work=648 I/O: 1->1
Identity work=648 I/O: 1->1
Identity work=648 I/O: 1->1
Identity work=648 I/O: 1->1
Identity work=648 I/O: 1->1
Identity work=648 I/O: 1->1
Identity work=648 I/O: 1->1
Identity work=648 I/O: 1->1
Identity work=648 I/O: 1->1
roundrobin(108,108,108,108,108,108,108,108,108,108,108,108)
roundrobin(1,1)
MultiplyAccumulateParallel
roundrobin(24,24,24,24,24,24,24,24,24,24,24,24)
MultiplyAccumulate work=1647 I/O: 24->1
MultiplyAccumulate work=1647 I/O: 24->1
MultiplyAccumulate work=1647 I/O: 24->1
MultiplyAccumulate work=1647 I/O: 24->1
MultiplyAccumulate work=1647 I/O: 24->1
MultiplyAccumulate work=1647 I/O: 24->1
MultiplyAccumulate work=1647 I/O: 24->1
MultiplyAccumulate work=1647 I/O: 24->1
MultiplyAccumulate work=1647 I/O: 24->1
MultiplyAccumulate work=1647 I/O: 24->1
MultiplyAccumulate work=1647 I/O: 24->1
MultiplyAccumulate work=1647 I/O: 24->1
roundrobin(1,1,1,1,1,1,1,1,1,1,1,1)
FloatPrinter work=648 I/O: 1->0

Figure B-46: Stream graph for MatrixMult (fine).

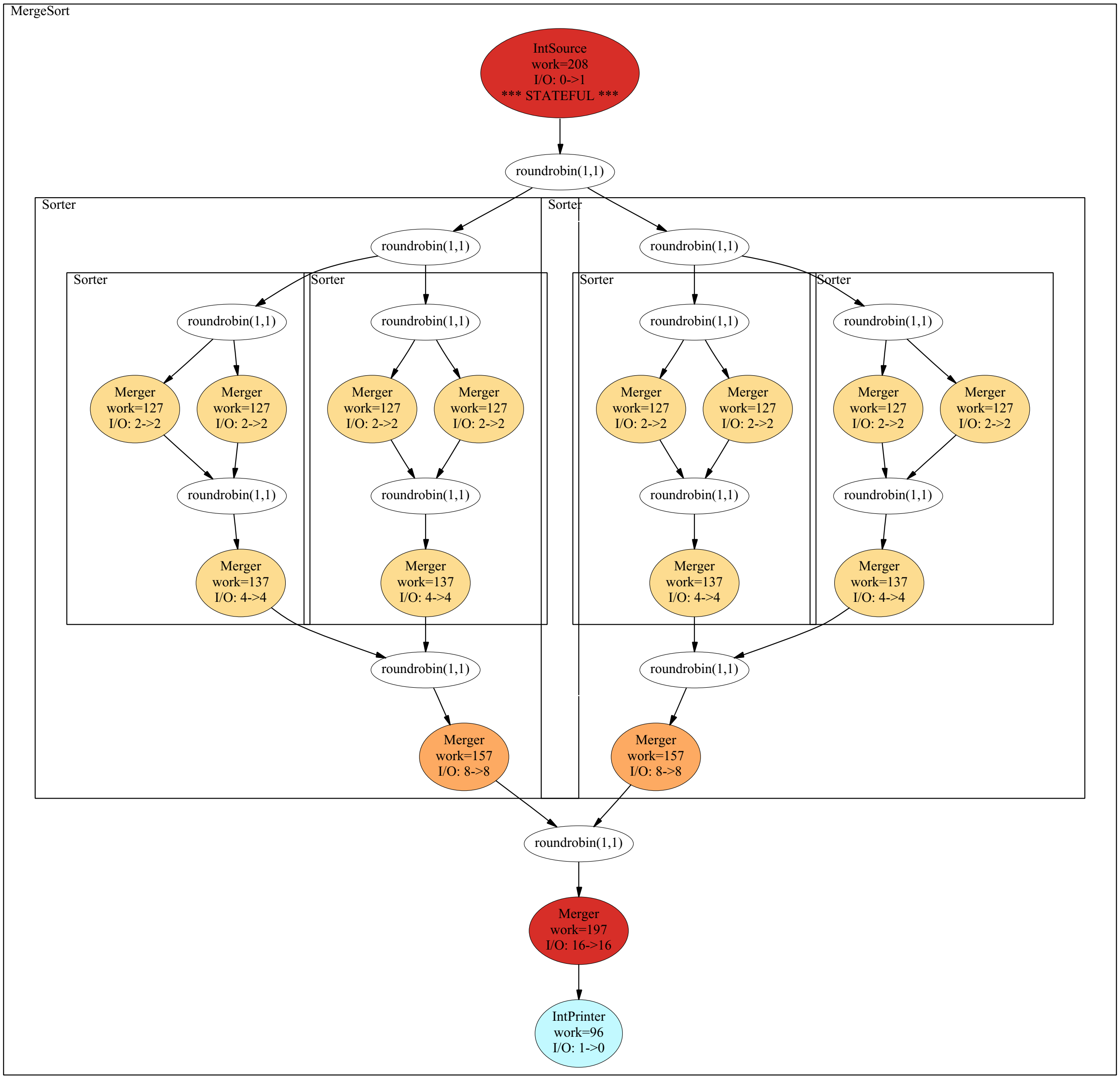


Figure B-47: Stream graph for MergeSort.

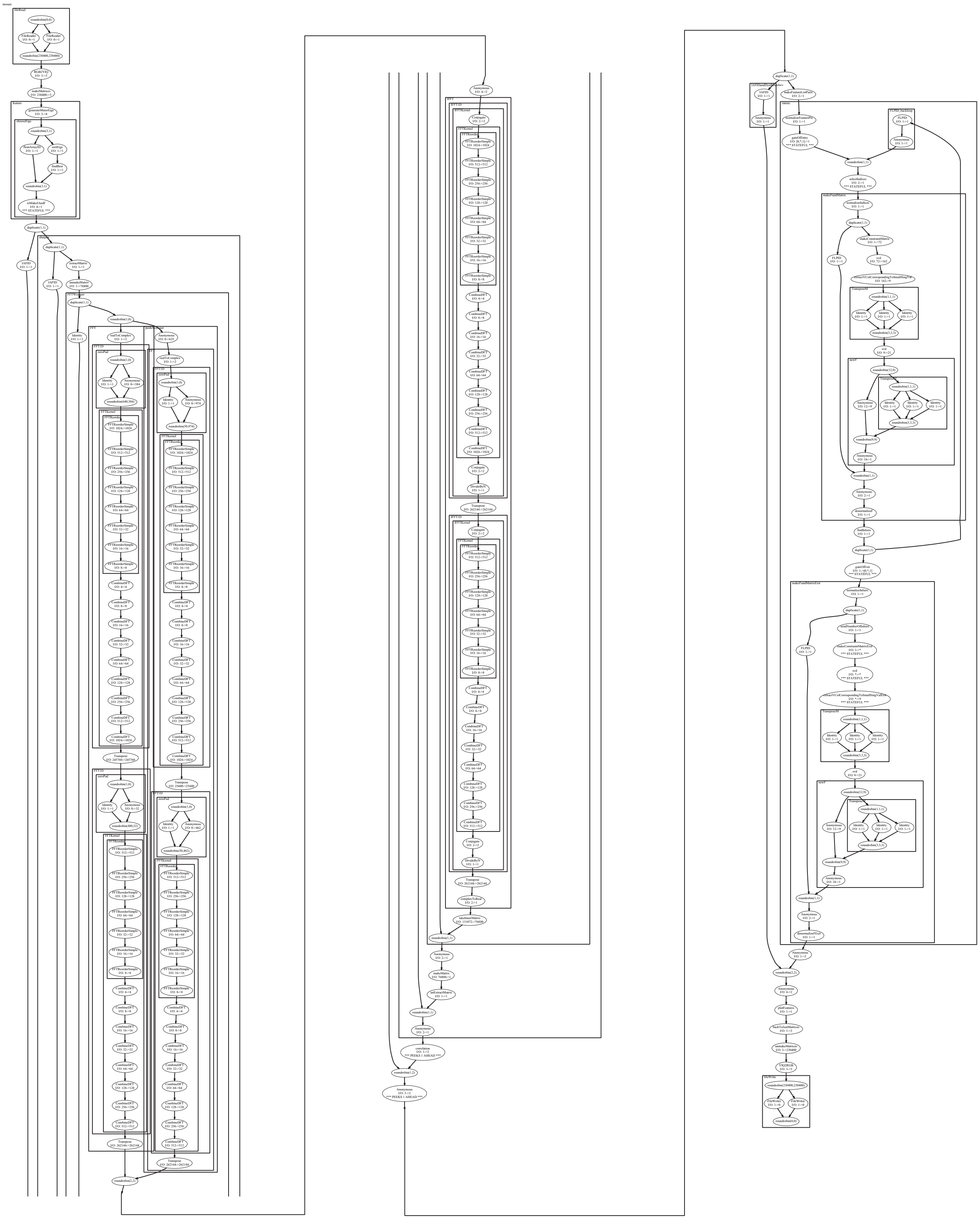


Figure B-48: Stream graph for Mosaic (teleport messages not shown).

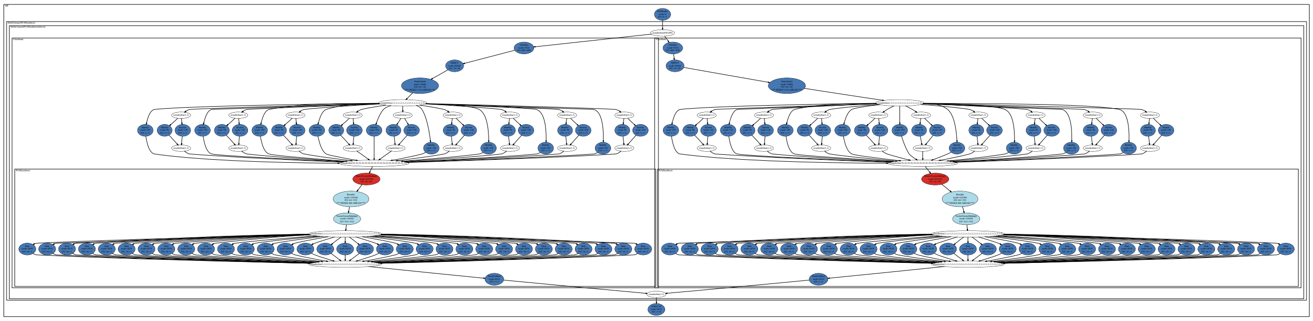


Figure B-49: Stream graph for MP3.

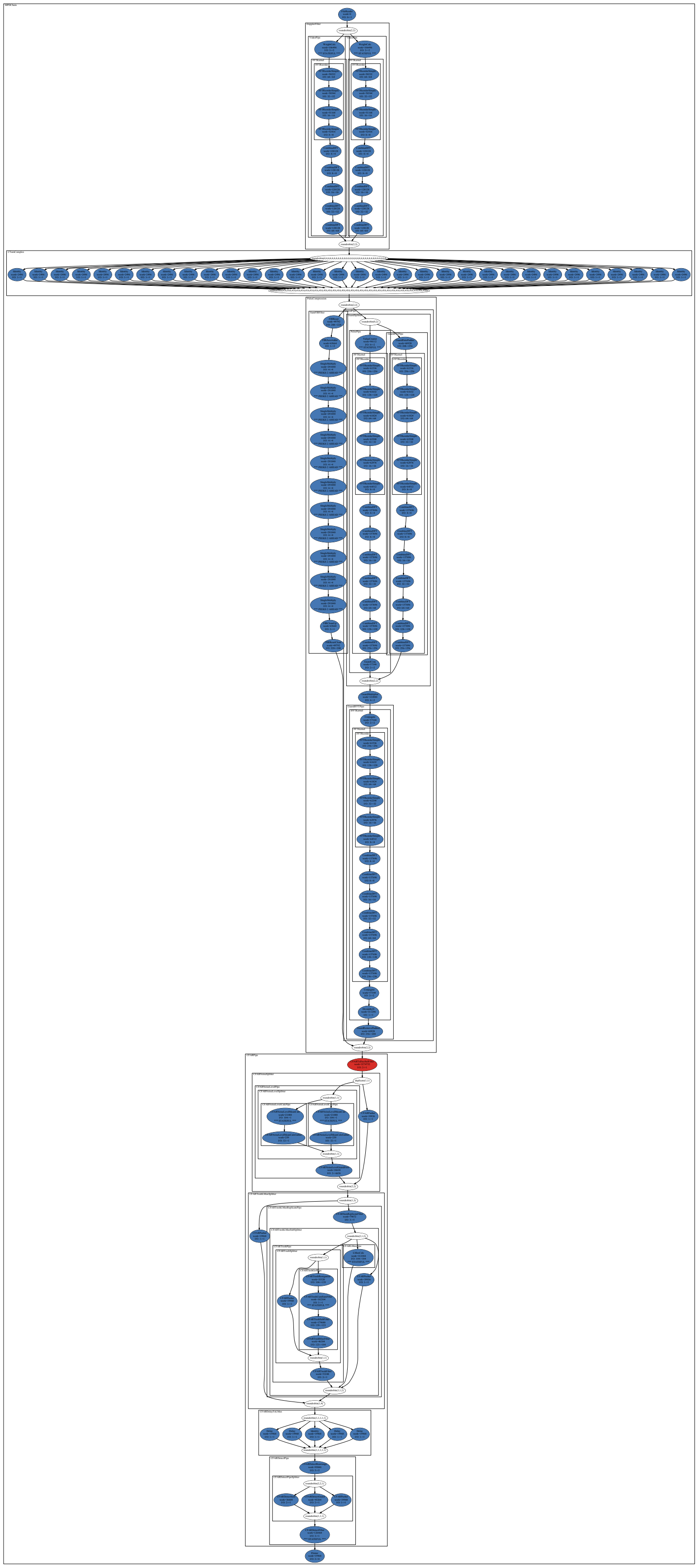


Figure B-50: Stream graph for MPD.

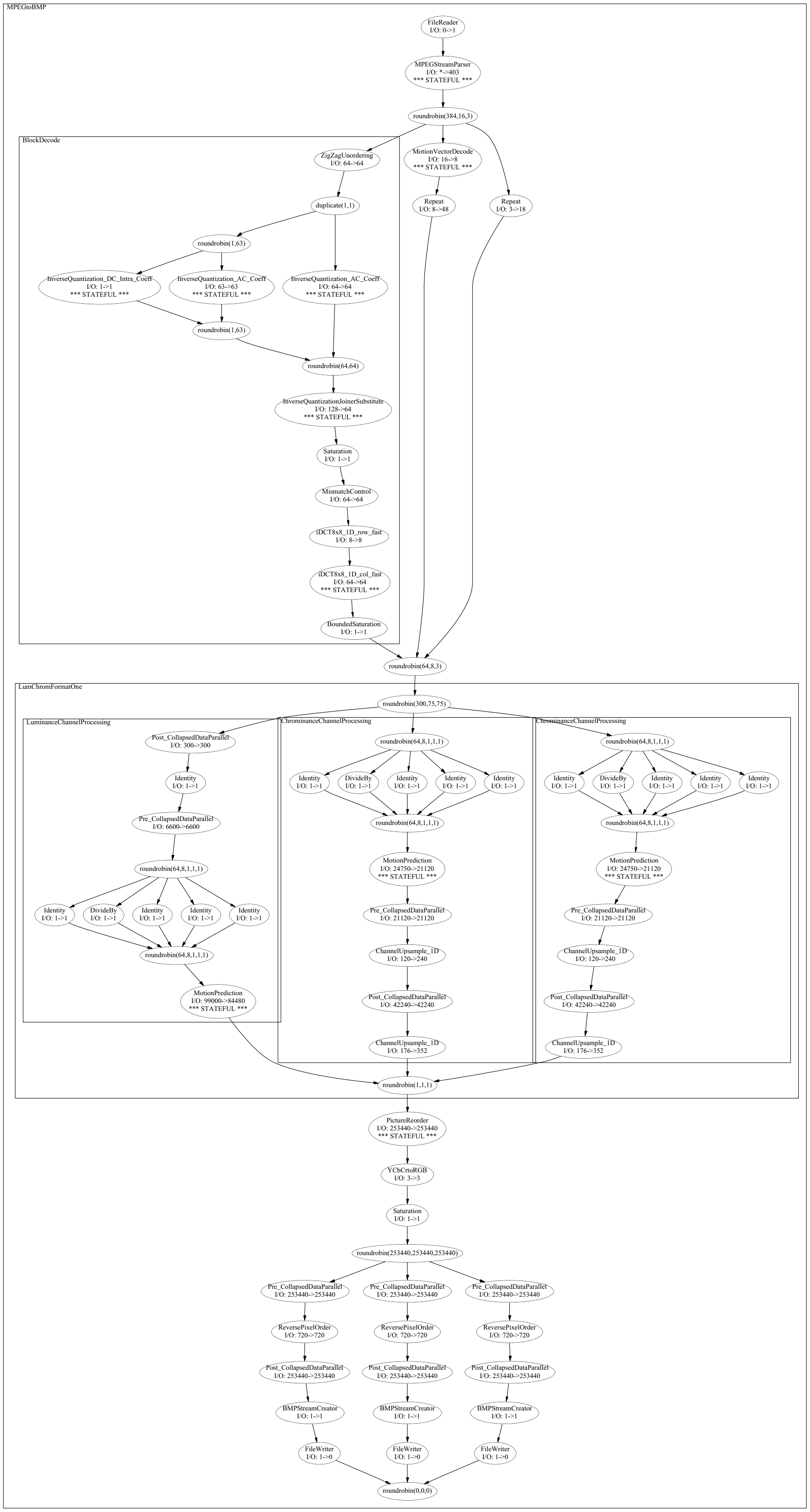


Figure B-51: Stream graph for MPEG2 decoder (teleport messages not shown).

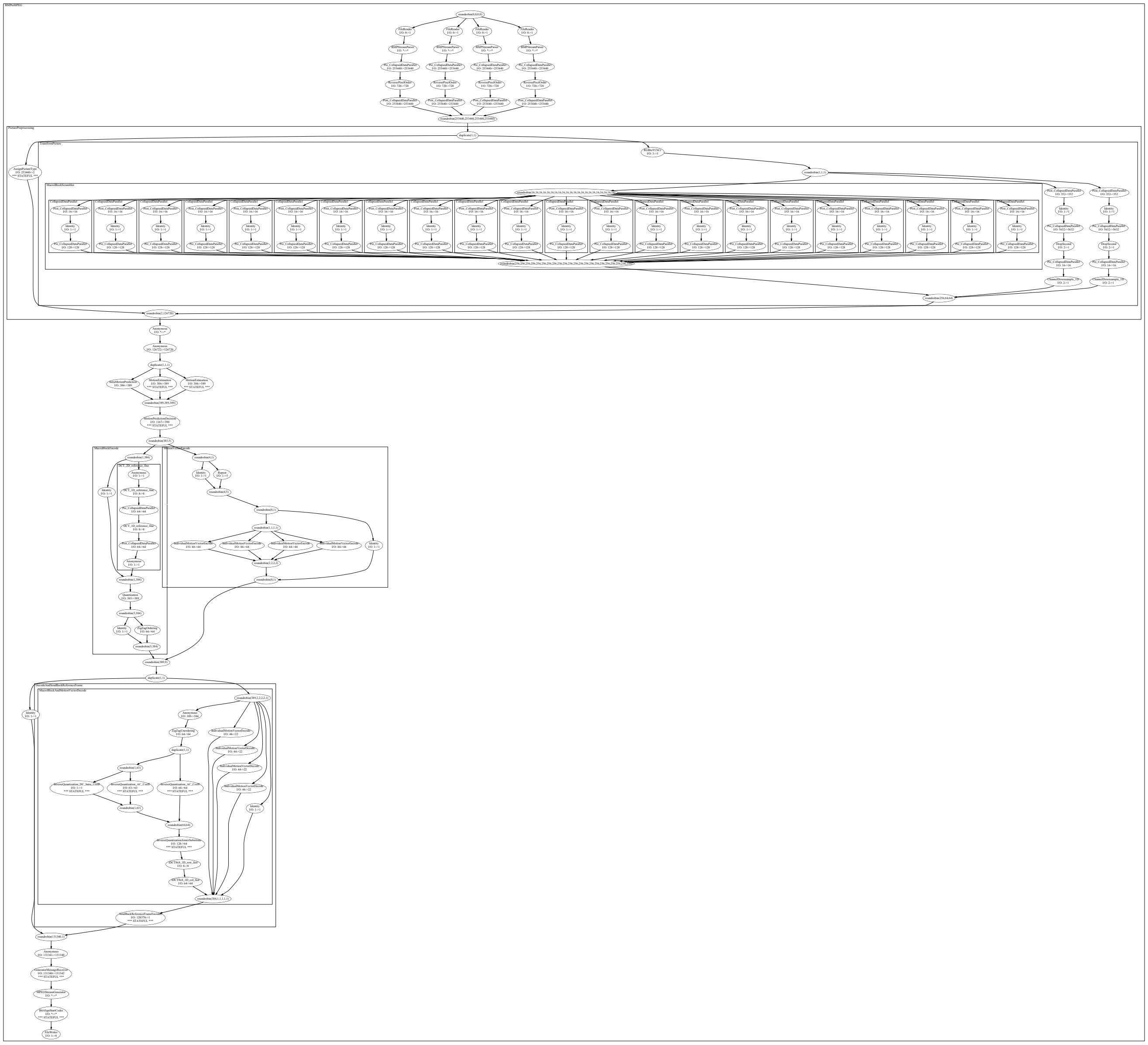

BMPtoMPEG
roundrobin(0,0,0,0)
FileReader I/O: 0->1
BMPStreamParser I/O: *->*
Pre_CollapsedDataParallel I/O: 253440->253440
ReversePixelOrder I/O: 720->720
Post_CollapsedDataParallel I/O: 253440->253440
roundrobin(253440,253440,253440,253440)
PicturePreprocessing
duplicate(1,1)
AssignPictureType I/O: 253440->2 *** STATEFUL ***
TransformPicture
RGBtoYCbCr I/O: 3->3
roundrobin(1,1,1)
MacroBlockScrambler
roundrobin(16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16)
CollapsedDataParallel
Post_CollapsedDataParallel I/O: 16->16
Identity I/O: 1->1
Pre_CollapsedDataParallel I/O: 128->128
roundrobin(256,256,256,256,256,256,256,256,256,256,256,256,256,256,256,256,256,256,256,256,256,256)
Post_CollapsedDataParallel I/O: 352->352
Identity I/O: 1->1
Pre_CollapsedDataParallel I/O: 5632->5632
DropSecond I/O: 2->1
Pre_CollapsedDataParallel I/O: 16->16
ChannelDownsample_1D I/O: 2->1
roundrobin(256,64,64)
roundrobin(2,126720)
Anonymous I/O: *->*
Anonymous I/O: 126722->126720
duplicate(1,1,1)
IntraMotionPrediction I/O: 384->389
MotionEstimation I/O: 384->389 *** STATEFUL ***
MotionEstimation I/O: 384->389 *** STATEFUL ***
roundrobin(389,389,389)
MotionPredictionDecision I/O: 1167->390 *** STATEFUL ***
roundrobin(385,5)
MacroBlockEncode
roundrobin(1,384)
DCT_2D_reference_fine
Anonymous I/O: 1->1
DCT_1D_reference_fine I/O: 8->8
Pre_CollapsedDataParallel I/O: 64->64
DCT_1D_reference_fine I/O: 8->8
Post_CollapsedDataParallel I/O: 64->64
Anonymous I/O: 1->1
Identity I/O: 1->1
roundrobin(1,384)
Quantisation I/O: 385->389
roundrobin(5,384)
Identity I/O: 1->1
ZigZagOrdering I/O: 64->64
roundrobin(5,384)
MotionVectorEncode
roundrobin(4,1)
Identity I/O: 1->1
Repeat I/O: 1->5
roundrobin(4,5)
roundrobin(8,1)
roundrobin(1,1,1,1)
IndividualMotionVectorEncode I/O: 44->44
Identity I/O: 1->1
roundrobin(2,2,2,2)
roundrobin(8,1)
roundrobin(389,9)
duplicate(1,1)
DecodeAndSendBackReferenceFrame
MacroBlockAndMotionVectorDecode
roundrobin(389,2,2,2,2,1)
Identity I/O: 1->1
Anonymous I/O: 389->384
ZigZagUnordering I/O: 64->64
duplicate(1,1)
roundrobin(1,63)
InverseQuantization_DC_Intra_Coeff I/O: 1->1 *** STATEFUL ***
InverseQuantization_AC_Coeff I/O: 63->63 *** STATEFUL ***
InverseQuantization_AC_Coeff I/O: 64->64 *** STATEFUL ***
roundrobin(1,63)
roundrobin(64,64)
InverseQuantizationJoinerSubstitute I/O: 128->64 *** STATEFUL ***
iDCT8x8_1D_row_fast I/O: 8->8
iDCT8x8_1D_col_fast I/O: 64->64
IndividualMotionVectorDecode I/O: 44->22
Identity I/O: 1->1
roundrobin(384,1,1,1,1,1)
SendBackReferenceFrameEncoder I/O: 128370->1 *** STATEFUL ***
roundrobin(131340,1)
Anonymous I/O: 131341->131340
GeneratorMessageReceiver I/O: 131340->131342 *** STATEFUL ***
MPEGStreamGenerator I/O: *->*
BitAlignStartCodes I/O: *->* *** STATEFUL ***
FileWriter I/O: 1->0

Figure B-52: Stream graph for MPEG2 encoder (teleport messages not shown).

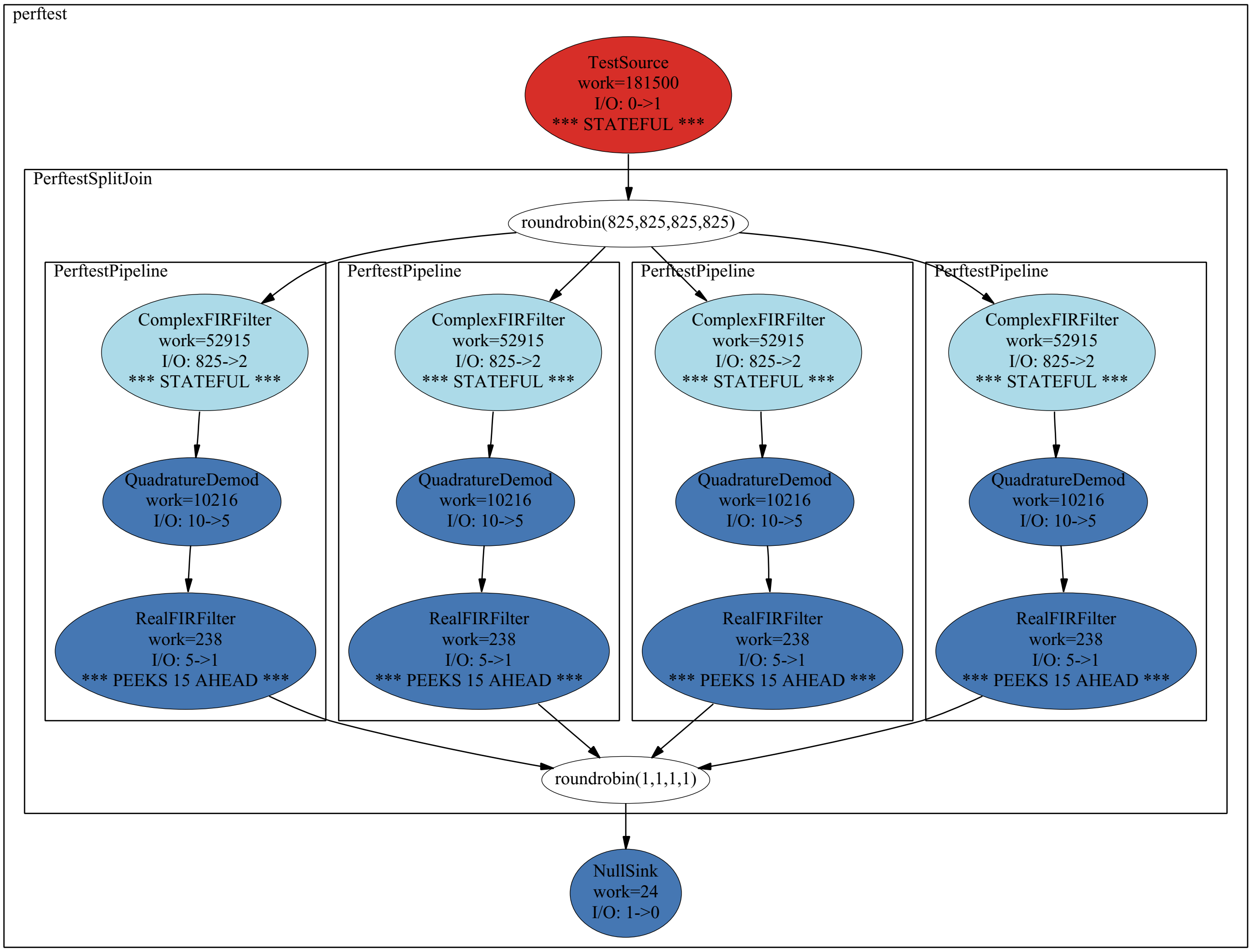


Figure B-53: Stream graph for OFDM.

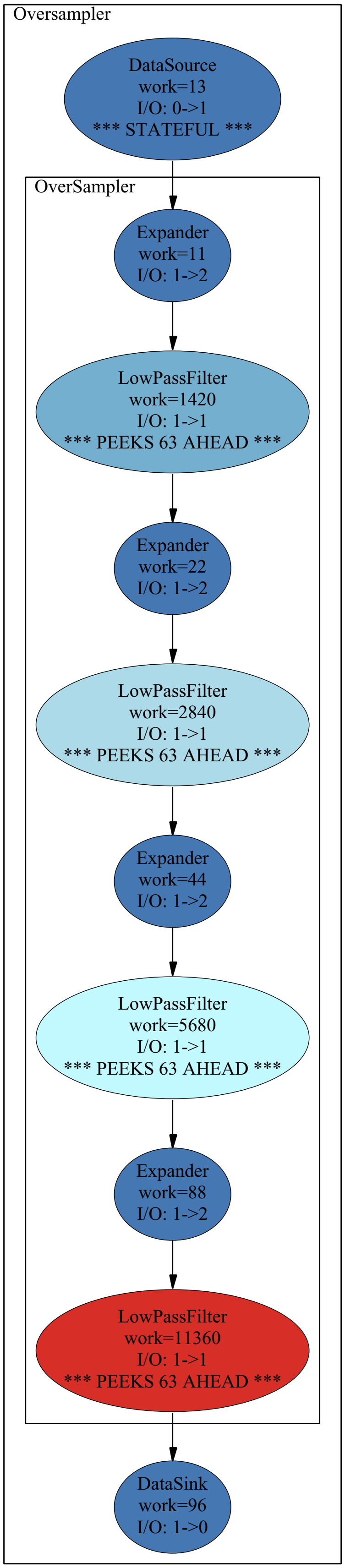


Figure B-54: Stream graph for Oversampler.

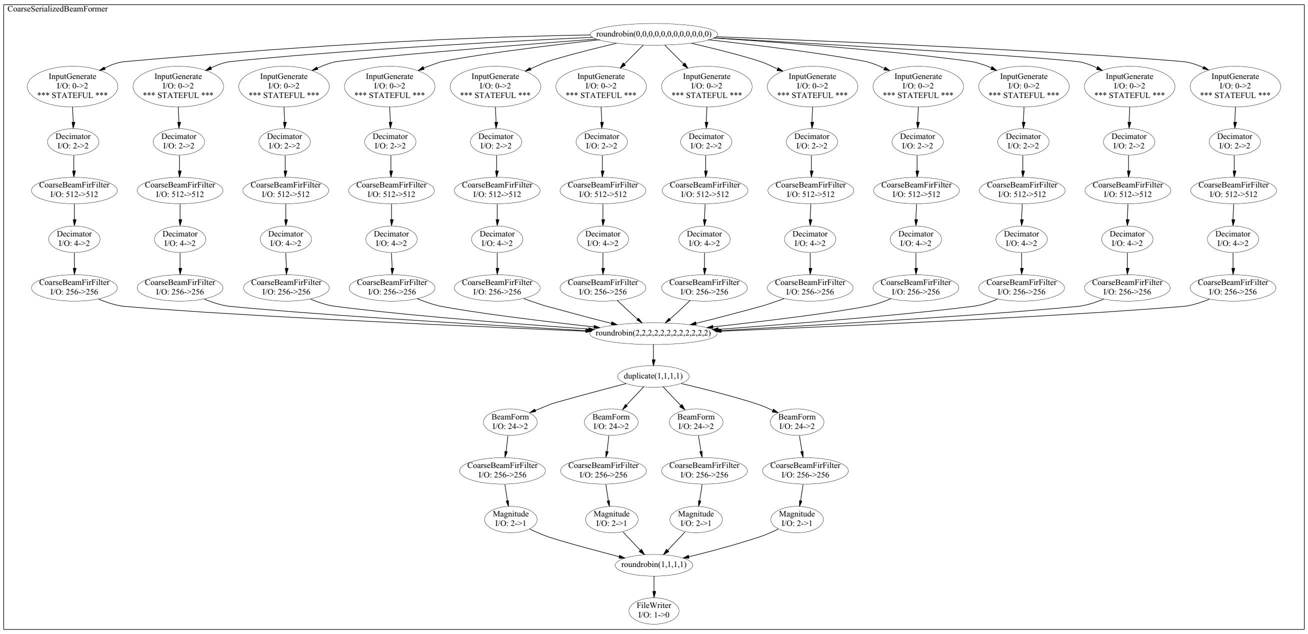

CoarseSerializedBeamFormer
roundrobin(0,0,0,0,0,0,0,0,0,0,0,0)
InputGenerate I/O: 0->2 *** STATEFUL ***
InputGenerate I/O: 0->2 *** STATEFUL ***
InputGenerate I/O: 0->2 *** STATEFUL ***
InputGenerate I/O: 0->2 *** STATEFUL ***
InputGenerate I/O: 0->2 *** STATEFUL ***
InputGenerate I/O: 0->2 *** STATEFUL ***
InputGenerate I/O: 0->2 *** STATEFUL ***
InputGenerate I/O: 0->2 *** STATEFUL ***
InputGenerate I/O: 0->2 *** STATEFUL ***
InputGenerate I/O: 0->2 *** STATEFUL ***
InputGenerate I/O: 0->2 *** STATEFUL ***
InputGenerate I/O: 0->2 *** STATEFUL ***
Decimator I/O: 2->2
Decimator I/O: 2->2
Decimator I/O: 2->2
Decimator I/O: 2->2
Decimator I/O: 2->2
Decimator I/O: 2->2
Decimator I/O: 2->2
Decimator I/O: 2->2
Decimator I/O: 2->2
Decimator I/O: 2->2
Decimator I/O: 2->2
Decimator I/O: 2->2
CoarseBeamFirFilter I/O: 512->512
CoarseBeamFirFilter I/O: 512->512
CoarseBeamFirFilter I/O: 512->512
CoarseBeamFirFilter I/O: 512->512
CoarseBeamFirFilter I/O: 512->512
CoarseBeamFirFilter I/O: 512->512
CoarseBeamFirFilter I/O: 512->512
CoarseBeamFirFilter I/O: 512->512
CoarseBeamFirFilter I/O: 512->512
CoarseBeamFirFilter I/O: 512->512
CoarseBeamFirFilter I/O: 512->512
CoarseBeamFirFilter I/O: 512->512
Decimator I/O: 4->2
Decimator I/O: 4->2
Decimator I/O: 4->2
Decimator I/O: 4->2
Decimator I/O: 4->2
Decimator I/O: 4->2
Decimator I/O: 4->2
Decimator I/O: 4->2
Decimator I/O: 4->2
Decimator I/O: 4->2
Decimator I/O: 4->2
Decimator I/O: 4->2
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
roundrobin(2,2,2,2,2,2,2,2,2,2,2,2)
duplicate(1,1,1,1)
BeamForm I/O: 24->2
BeamForm I/O: 24->2
BeamForm I/O: 24->2
BeamForm I/O: 24->2
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
CoarseBeamFirFilter I/O: 256->256
Magnitude I/O: 2->1
Magnitude I/O: 2->1
Magnitude I/O: 2->1
Magnitude I/O: 2->1
roundrobin(1,1,1,1)
FileWriter I/O: 1->0

Figure B-55: Stream graph for Radar (coarse).

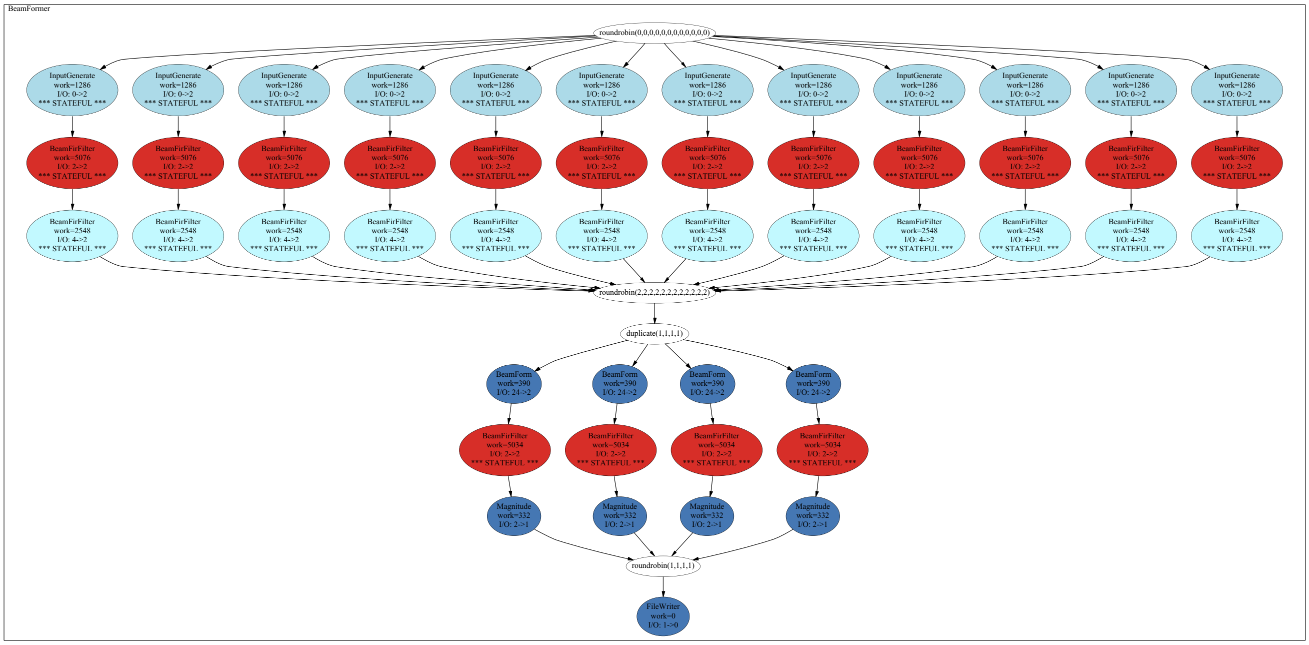


Figure B-56: Stream graph for Radar (fine).

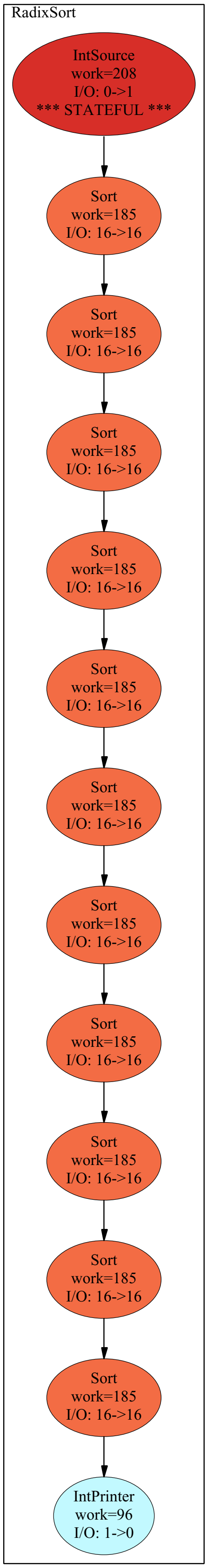


Figure B-57: Stream graph for RadixSort.

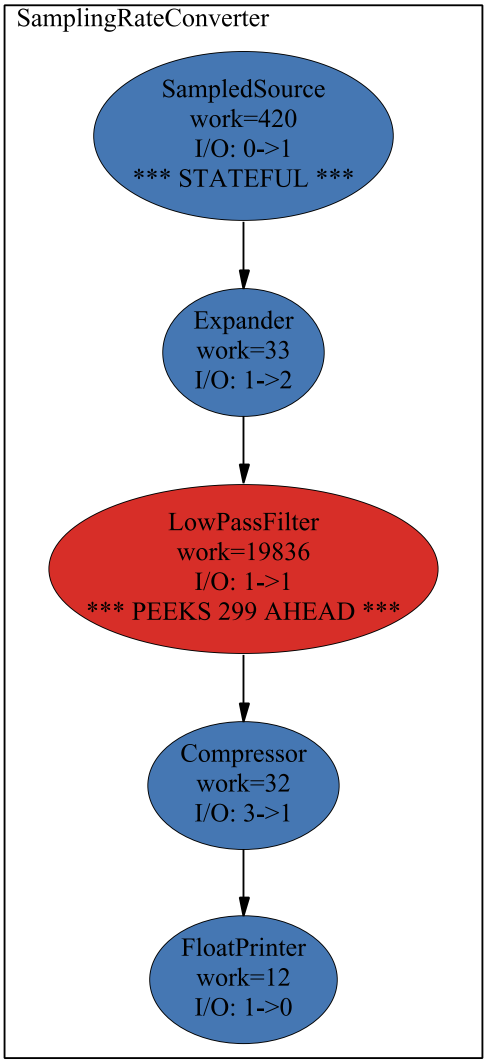


Figure B-58: Stream graph for RateConvert.

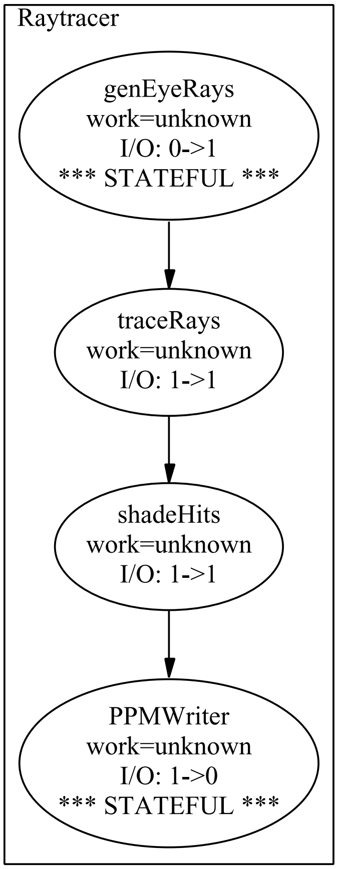


Figure B-59: Stream graph for Raytracer1.

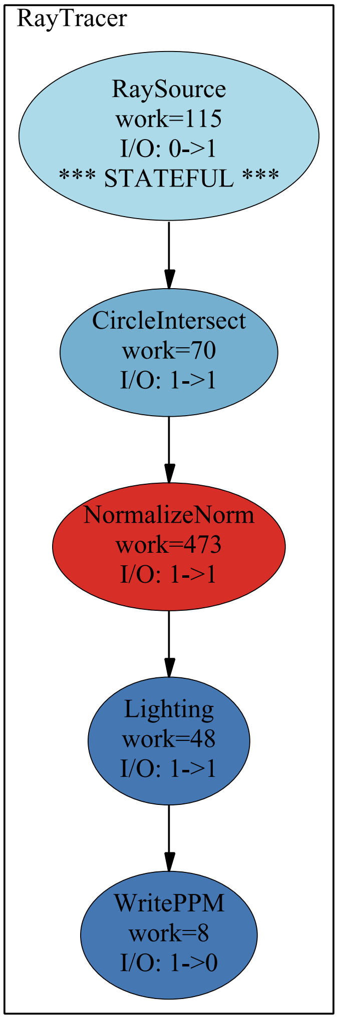


Figure B-60: Stream graph for RayTracer2.

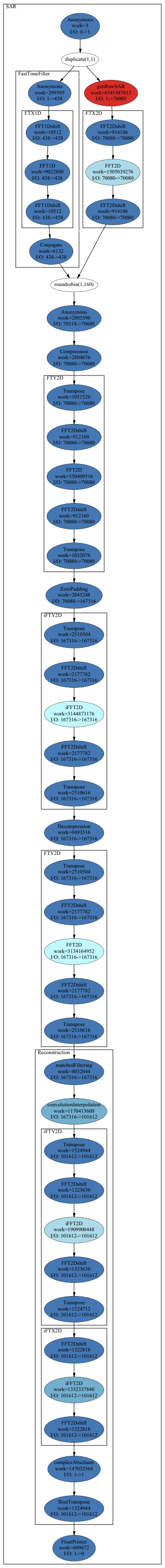


Figure B-61: Stream graph for SAR.

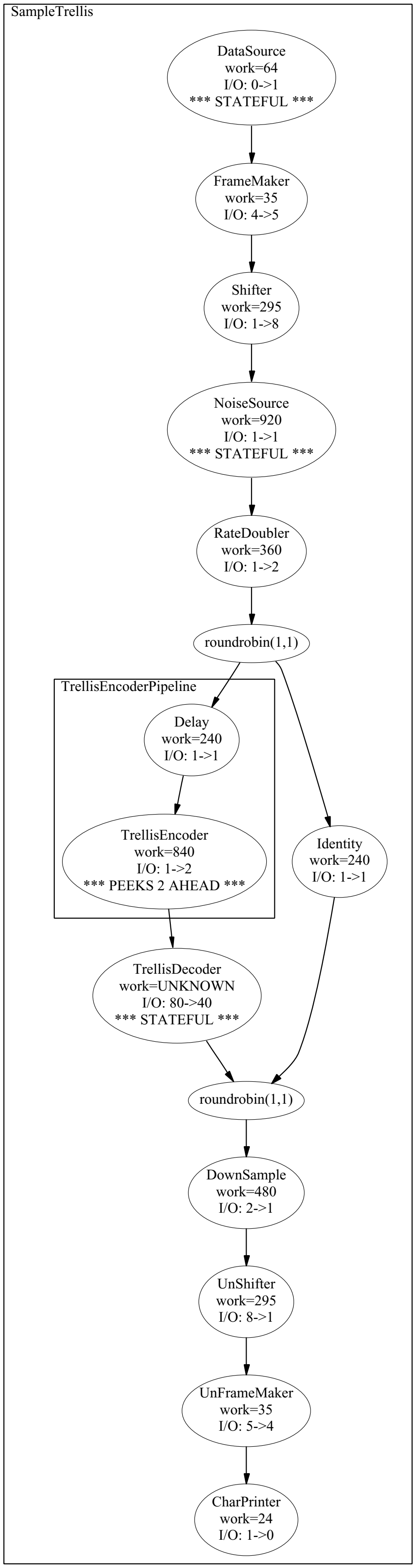
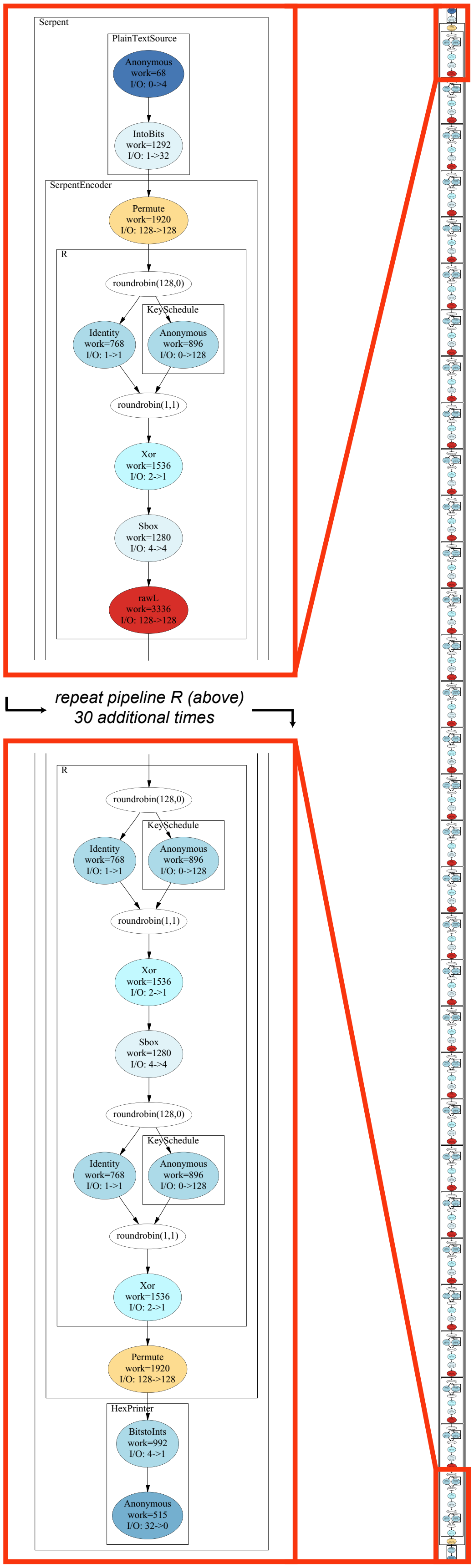


Figure B-62: Stream graph for SampleTrellis.



*repeat pipeline R (above)
30 additional times*

Figure B-63: Stream graph for Serpent.

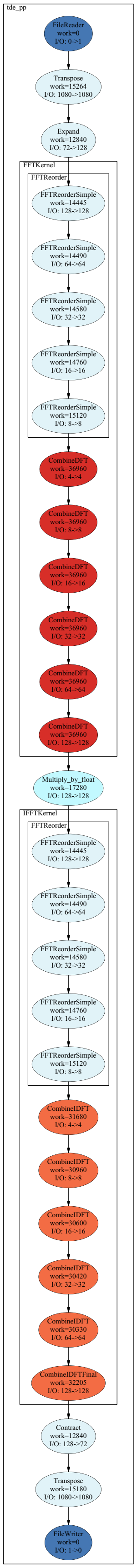


Figure B-64: Stream graph for TDE.

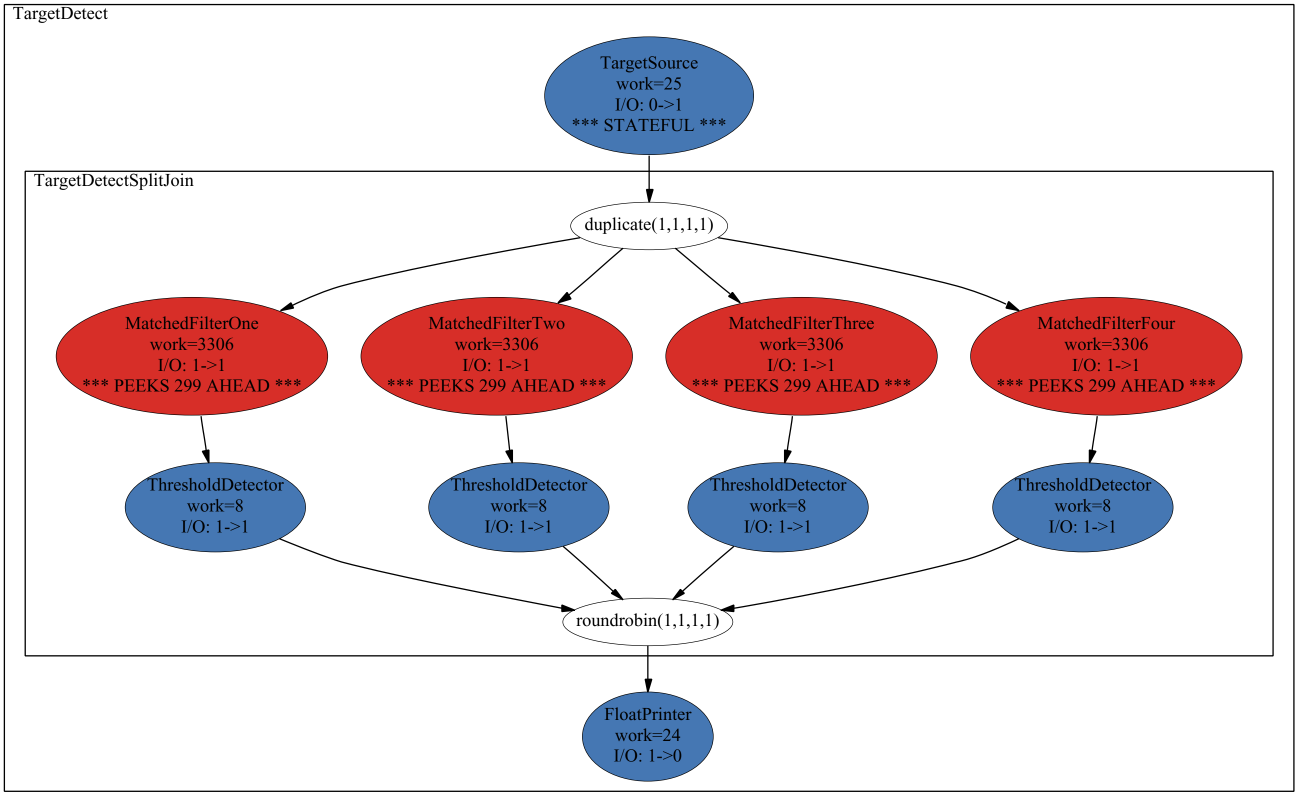


Figure B-65: Stream graph for TargetDetect.

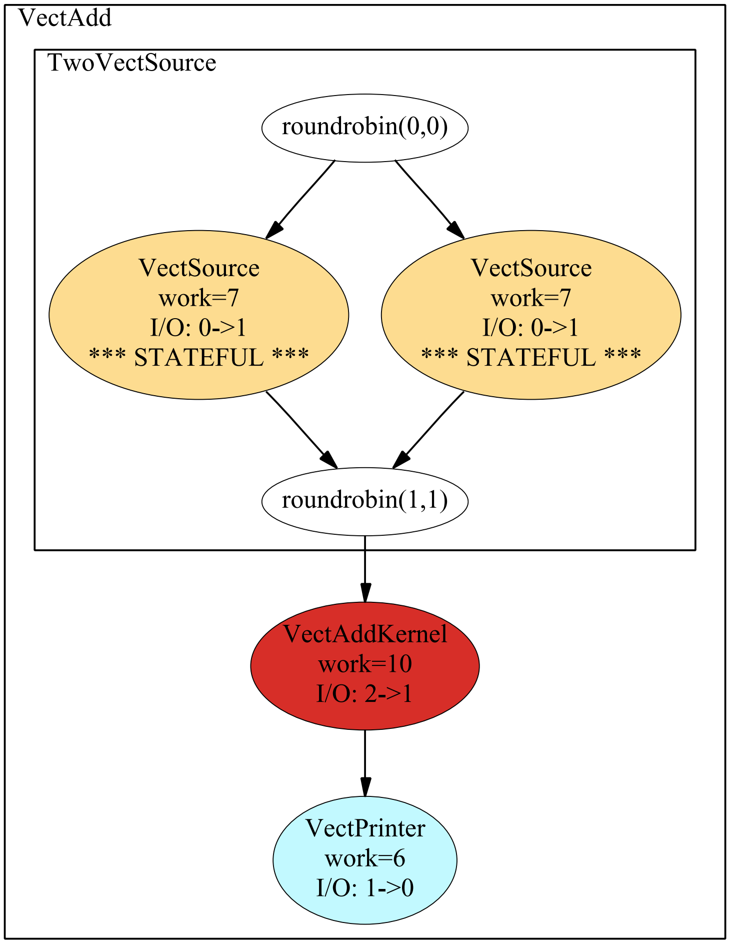


Figure B-66: Stream graph for VectAdd.

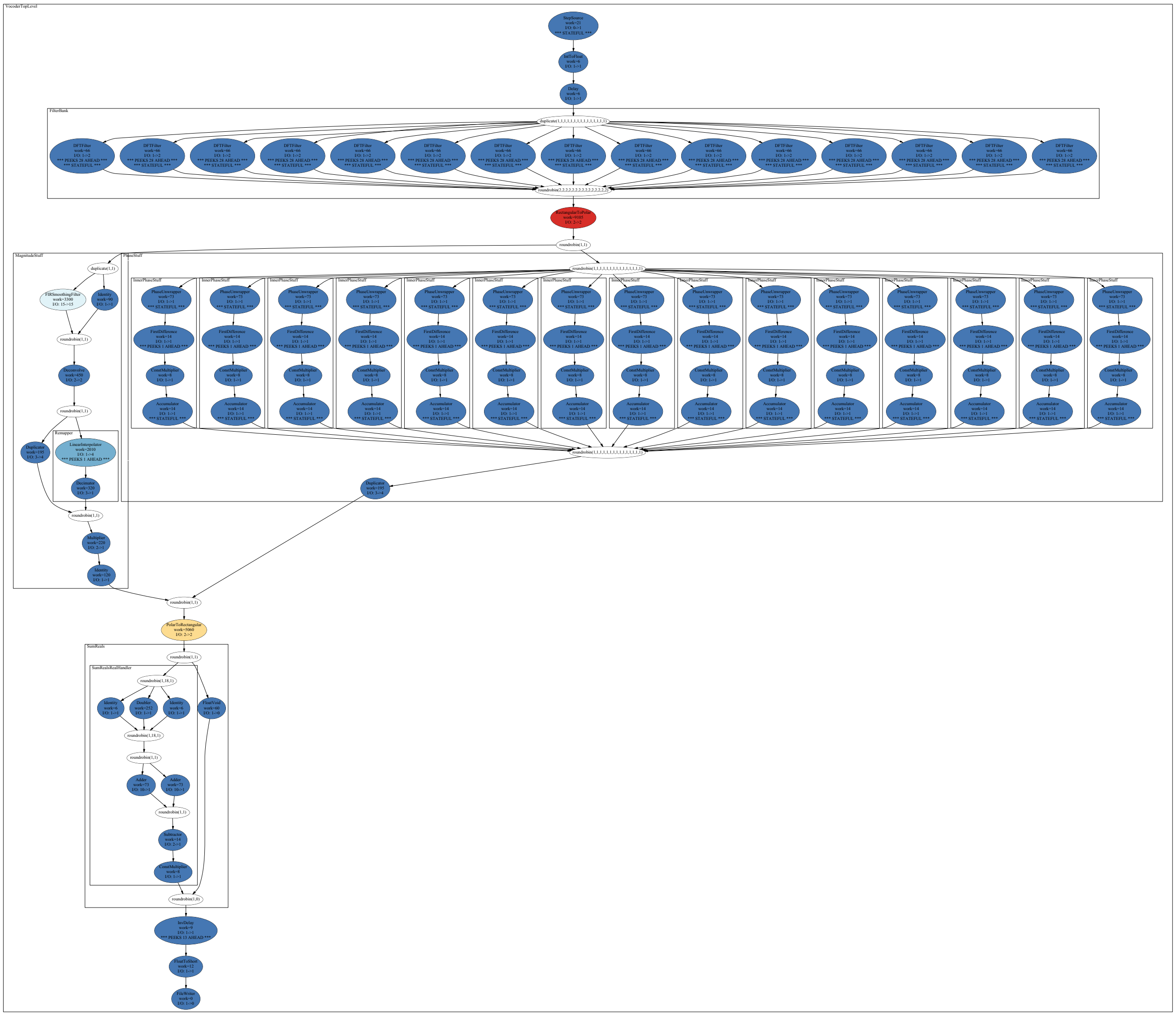

VocoderTopLevel
StepSource work=21 I/O: 0->1 *** STATEFUL ***
IntToFloat work=6 I/O: 1->1
Delay work=6 I/O: 1->1
FilterBank
duplicate(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
DFTFilter work=66 I/O: 1->2 *** PEEKS 28 AHEAD *** *** STATEFUL ***
roundrobin(2,2,2,2,2,2,2,2,2,2,2,2,2,2,2)
RectangularToPolar work=9105 I/O: 2->2
roundrobin(1,1)
MagnitudeStuff
duplicate(1,1)
FIRSmoothingFilter work=3300 I/O: 15->15
Identity work=90 I/O: 1->1
roundrobin(1,1)
Deconvolve work=450 I/O: 2->2
roundrobin(1,1)
Duplicator work=195 I/O: 3->4
Remapper
LinearInterpolator work=2010 I/O: 1->4 *** PEEKS 1 AHEAD ***
Decimator work=320 I/O: 3->1
roundrobin(1,1)
Multiplier work=220 I/O: 2->1
Identity work=120 I/O: 1->1
PhaseStuff
roundrobin(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
InnerPhaseStuff
PhaseUnwrapper work=73 I/O: 1->1 *** STATEFUL ***
FirstDifference work=14 I/O: 1->1 *** PEEKS 1 AHEAD ***
ConstMultiplier work=8 I/O: 1->1
Accumulator work=14 I/O: 1->1 *** STATEFUL ***
roundrobin(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)
Duplicator work=195 I/O: 3->4
roundrobin(1,1)
PolarToRectangular work=5060 I/O: 2->2
SumReals
roundrobin(1,1)
SumRealsRealHandler
roundrobin(1,18,1)
Identity work=6 I/O: 1->1
Doubler work=252 I/O: 1->1
Identity work=6 I/O: 1->1
FloatVoid work=60 I/O: 1->0
roundrobin(1,18,1)
roundrobin(1,1)
Adder work=73 I/O: 10->1
Adder work=73 I/O: 10->1
roundrobin(1,1)
Subtractor work=14 I/O: 2->1
ConstMultiplier work=8 I/O: 1->1
roundrobin(1,0)
InvDelay work=9 I/O: 1->1 *** PEEKS 13 AHEAD ***
FloatToShort work=12 I/O: 1->1
FileWriter work=0 I/O: 1->0

Figure B-67: Stream graph for Vocoder.