A Stream-Aware Compiler for Communication-Exposed Architectures

by

Michael I. Gordon

B.S., Computer Science (2000)
Rutgers University

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

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2002

Author . . . .
Department of Electrical Engineering and Computer Science
August 29, 2002

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

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Stream-Aware Compiler for Communication-Exposed Architectures

by

Michael I. Gordon

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

With the increasing miniaturization of transistors, wire delays are becoming a dominant factor in microprocessor performance. To address this issue, a number of emerging architectures contain replicated processing units with software-exposed communication between one unit and another (e.g., Raw, SmartMemories, TRIPS). However, for their use to be widespread, it will be necessary to develop compiler technology that enables a portable, high-level language to execute efficiently across a range of wire-exposed architectures.

In this thesis, we describe our compiler for StreamIt: a high-level, architecture-independent language for streaming applications. We focus on our backend for the Raw processor. Though StreamIt exposes the parallelism and communication patterns of stream programs, analysis is needed to adapt a stream program to a software-exposed processor. We describe a partitioning algorithm that employs fission and fusion transformations to adjust the granularity of a stream graph, a layout algorithm that maps a stream graph to a given network topology, and a scheduling strategy that generates a fine-grained static communication pattern for each computational element.

We have implemented a fully functional compiler that parallelizes StreamIt applications for Raw, including several load-balancing transformations. Using the cycle-accurate Raw simulator, we demonstrate that the StreamIt compiler can automatically map a high-level stream abstraction to Raw without losing performance. We consider this work to be a first step towards a portable programming model for communication-exposed architectures.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

I am grateful to my advisor Saman Amarasinghe, without his guidance this thesis would not have been possible. I am permanently indebted to William Thies for his work on the partitioning phase of the compiler and many other aspects of the project. His knowledge, patience, attitude, and work are truly amazing. I would like to thank the rest of the StreamIt group: Michal Karczmarek for his work on the scheduler; Jasper Lin for his work on ynchronization removal; Chris Leger for work on the Vocoder application and gathering results; Ali Meli for his work on 3GPP, Filterbank, and other applications; Andrew Lamb for his help gathering results; Jeremy Wong for his work on GSM; and David Maze for his help with the compiler, optimizing the applications, and gathering results. I am thankful to the members of the Raw group, primarily Michael Taylor, Dave Wentzlaf, and Walter Lee, for their helpfulness and enthusiasm. I would like to thank Sam Larsen, Mark Stephenson, Mike Zhang, and Diego Puppin for their comments. Most importantly, I thank my loving and supportive family and friends.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As we approach the billion-transistor era, a number of emerging architectures are addressing the wire delay problem by replicating the basic processing unit and exposing the communication between units to a software layer (*e.g.*, Raw [43], SmartMemories [30], TRIPS [36]). These machines are especially well-suited for streaming applications that have regular communication patterns and widespread parallelism.

However, today's communication-exposed architectures are lacking a portable programming model. If these machines are to be widely used, it is imperative that one be able to write a program once, in a high-level language, and rely on a compiler to produce an efficient executable on any of the candidate targets. For von-Neumann machines, imperative programming languages such as C and FORTRAN served this purpose; they abstracted away the idiosyncratic details between one machine and another (such as the number and type of registers, the ISA, and the memory hierarchies), but encapsulated the common properties (such as a single program counter, arithmetic operations, and a monolithic memory) that are necessary to obtain good performance. However, for wire-exposed targets that contain multiple instruction streams and distributed memory banks, a language such as C is obsolete. Though C can still be used to write efficient programs on these machines, doing so either requires architecture-specific directives or an impossibly smart compiler that can extract the parallelism and communication from the C semantics. Both of these options disqualify C as a portable machine language, since it fails to hide the architectural details from the programmer and it imposes abstractions which are a mismatch for the domain.

In this paper, we describe a compiler for StreamIt [41], a high level stream language that aims to be portable across communication-exposed machines. StreamIt contains basic constructs that expose the parallelism and communication of streaming applications without depending on the topology or granularity of the underlying architecture. Our current backend is for Raw [43], a tiled architecture with fine-grained, programmable communication between processors. However, the compiler employs three general techniques that can be applied to compile StreamIt to machines other than Raw: 1) partitioning, which adjusts the granularity of a stream graph to match that of a given target, 2) layout, which maps a partitioned stream graph to a given network topology, and 3) scheduling, which generates a fine-grained communication pattern for each computational element. We consider this work to be a first step towards a portable programming model for communication-exposed architectures.

1.1 The StreamIt Language

StreamIt is a portable programming language for high-performance signal processing applications. The current version of StreamIt is tailored for static-rate streams: it requires that the input and output rates of each filter are known at compile time. In this section, we provide a brief overview of the syntax and semantics of StreamIt, version 1.1. A more detailed description of the design and rationale for StreamIt can be found in [41], which describes version 1.0; the most up-to-date syntax specification can always be found on our website [4].

Abstractly, the current semantics of the StreamIt language belong to the synchronous dataflow domain [27]. Computation is described by composing processing units into a network. The processing units, called filters, are connected to each other by channels. Data values pass over the channels, in a single direction, to neighboring filters. The term *synchronous* is used to denote the fact that a filter will not fire unless all of its inputs are available. Also, as mentioned above, the input and output

```
float->float filter FIRFilter (float sampleRate, int N) {
  float[N] weights;

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

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

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

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

float->float pipeline Equalizer (float samplingRate, int N) {
  add splitjoin {
    int bottom = 2500;
    int top = 5000;
    split duplicate;
    for (int i=0; i<N; i++, bottom*=2, top*=2) {
      add BandPassFilter(sampleRate, bottom, top);
    }
    join roundrobin;
  }
  add Adder(N);
}

void->void pipeline FMRadio {
  add DataSource();
  add FIRFilter(sampleRate, N);
  add FMDemodulator(sampleRate, maxAmplitude);
  add Equalizer(sampleRate, 4);
  add Speaker();
}
```

Figure 1-1: Parts of an FM Radio in StreamIt.

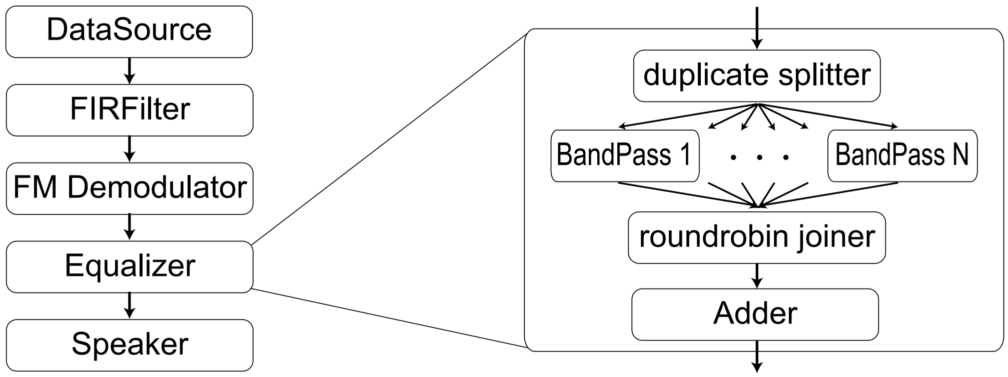


Figure 1-2: Block diagram of the FM Radio.

rates of each filter can be determined statically. In the balance of this thesis we will say that filter A is *downstream* of filter B if there is a path from A to B in the stream graph (following the direction of the channels). We also say that B is *upstream* of A.

1.1.1 Language Constructs

The basic unit of computation in StreamIt is the *filter*. A filter is a single-input, single-output block with a user-defined procedure for translating input items to output items. An example of a filter is the `FIRFilter`, a component of our software radio (see Figure 1-1). Each filter contains an `init` function that is called at initialization time; in this case, the `FIRFilter` calculates `weights`, which represents its impulse response. The `work` function describes the most fine grained execution step of the filter in the steady state. Within the `work` function, the filter can communicate with its neighbors via FIFO queues, using the intuitive operations of `push(value)`, `pop()`, and `peek(index)`, where `peek` returns the value at position `index` without dequeuing the item. The number of items that are pushed, popped, and peeked[1] on each invocation are declared with the `work` function.

In addition to `work`, a filter can contain a `prework` function that is executed exactly once between initialization and the steady-state. Like `work`, `prework` can

[1]We define *peek* as the total number of items read, including the items popped. Thus, we always have that $peek \geq pop$.

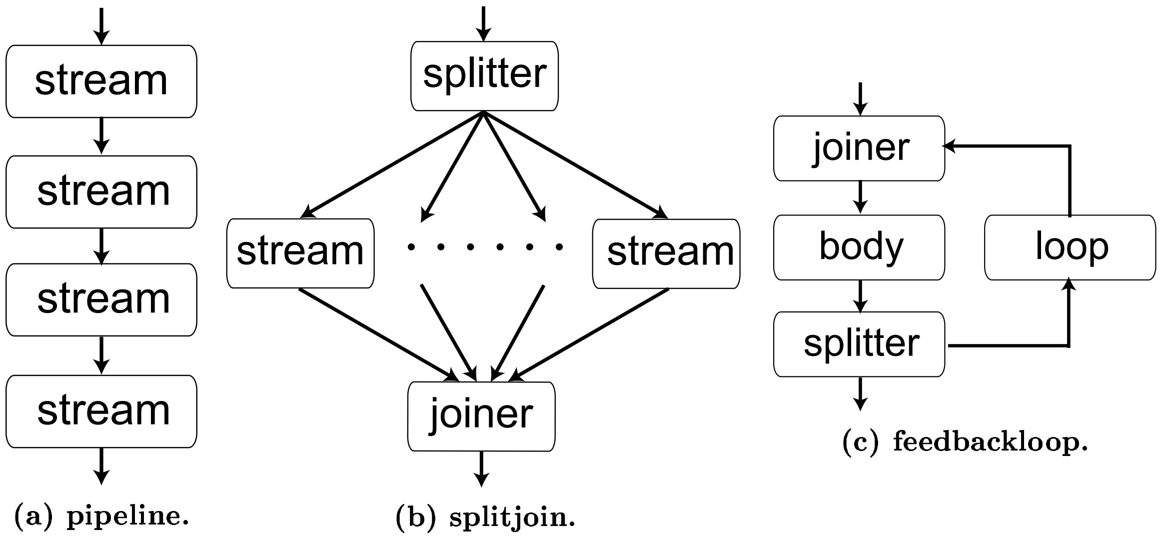


Figure 1-3: Stream structures supported by StreamIt.

access the input and output tapes of the filter; however, the I/O rates of `work` and `prework` can differ. In an `FIRFilter`, a `prework` function is essential for correctly filtering the beginning of the input stream. The user never calls the `init`, `prework`, and `work` functions–they are all called automatically.

The basic construct for composing filters into a communicating network is a *pipeline*, such as the FM Radio in Figure 1-1. A pipeline behaves as the sequential composition of all its child streams, which are specified with successive calls to `add` from within the pipeline. For example, the output of `DataSource` is implicitly connected to the input of `FIRFilter`, who's output is connected to `FMDemodulator`, and so on. The `add` statements can be mixed with regular imperative code to parameterize the construction of the stream graph.

There are two other stream constructs besides pipeline: *splitjoin* and *feedbackloop* (see Figure 1-3). From now on, we use the word *stream* to refer to any instance of a filter, pipeline, splitjoin, or feedbackloop.

A splitjoin is used to specify independent parallel streams that diverge from a common *splitter* and merge into a common *joiner*. There are two kinds of splitters: 1) *duplicate*, which replicates each data item and sends a copy to each parallel stream, and 2) $roundrobin(w_1, \ldots, w_n)$, which sends the first w_1 items to the first stream, the next w_2 items to the second stream, and so on. roundrobin is also the only type of joiner that we support; its function is analogous to a roundrobin splitter. If a roundrobin is written without any weights, we assume that all $w_i = 1$. The splitter and joiner type are specified with the keywords `split` and `join`, respectively (see Figure 1-1); the parallel streams are specified by successive calls to `add`, with the i'th call setting the i'th stream in the splitjoin.

The feedbackloop construct provides a way to create cycles in the stream graph. Each feedbackloop contains: 1) a body stream, which is the block around which a backwards "feedback path" is being created, 2) a loop stream, which can perform some computation along the feedback path, 3) a splitter, which distributes data between the feedback path and the output channel at the bottom of the loop, and 4) a joiner, which merges items between the feedback path and the input channel at the top of the loop. The splitters and joiners can be any of those for splitjoin, except for `null`.

The feedbackloop has special semantics when the stream is first starting to run. Since there are no items on the feedback path at first, the stream instead inputs items from an `initPath` function defined by the feedbackloop construct. Given an index i, `initPath` provides the i^{th} initial input for the feedback joiner. A call to `setDelay`, from within the `init` function specifies how many items should be calculated with `initPath` before the joiner looks for data from the loop.

1.2 The Raw Architecture

In this thesis we show that the StreamIt language is well-suited for wire-exposed architectures. StreamIt aims to be portable across these architectures and also deliver high performance. This thesis describes general compiler phases and transformations to enable portability and performance. We demonstrate this by developing a specific backend for MIT's Raw Microprocessor.

The Raw Microprocessor [12, 43] addresses the wire delay problem [18] by providing direct instruction set architecture (ISA) analogs to three underlying physical resources of the processor: gates, wires and pins. Because ISA primitives exist for

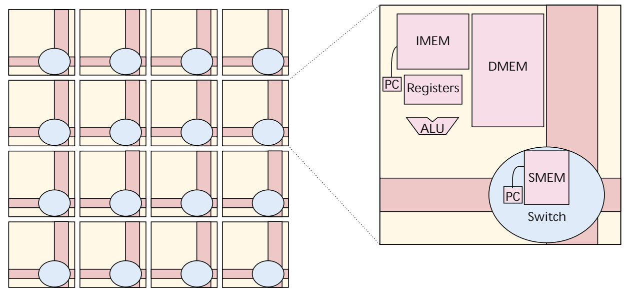


Figure 1-4: Block diagram of the Raw architecture.

these resources, a compiler such as the StreamIt compiler has direct control over both the computation and the communication of values between the functional units of the microprocessor, as well as across the pins of the processor.

The architecture exposes the gate resources as a scalable 2-D array of identical, programmable tiles, that are connected to their immediate neighbors by four on-chip networks. Each network is 32-bit, full-duplex, flow-controlled and point-to-point. On the edges of the array, these networks are connected via logical channels [16] to the pins. Thus, values routed through the networks off of the side of the array appear on the pins, and values placed on the pins by external devices (for example, wide-word A/Ds, DRAMS, video streams and PCI-X buses) will appear on the networks.

Each of the tiles contains a single-issue compute processor, some memory and two types of routers–one static, one dynamic–that control the flow of data over the networks as well as into the compute processor (see Figure 1-4). The compute processor interfaces to the network through a bypassed, register-mapped interface [12] that allows instructions to use the networks and the register files interchangeably. In other words, a single instruction can read up to two values from the networks, compute on them, and send the result out onto the networks, with no penalty. Reads and writes in this fashion are blocking and flow-controlled, which allows for the computation to remain unperturbed by unpredictable timing variations such as cache misses and interrupts.

Each tile's static router has a virtualized instruction memory to control the crossbars of the two static networks. Collectively, the static routers can reconfigure the communication pattern across these networks every cycle. The instruction set of the static router is encoded as a 64-bit VLIW word that includes basic instructions (conditional branch with/without decrement, move, and nop) that operate on values from the network or from the local 4-element register file. Each instruction also has 13 fields that specify the connections between each output of the two crossbars and the network input FIFOs, which store values that have arrived from neighboring tiles or the local compute processor. The input and output possibilities for each crossbar are: North, East, South, West, Processor, to the other crossbar, and into the static router. The FIFOs are typically four or eight elements large.

To route a word from one tile to another, the compiler inserts a route instruction on every intermediate static router [29]. Because the routers are pipelined and compile-time scheduled, they can deliver a value from the ALU of one tile to the ALU of a neighboring tile in 3 cycles, or more generally, 2+N cycles for an inter-tile distance of N hops.

All functional units except the floating point and integer dividers are fully pipelined. The mispredict penalty of the static branch predictor is three cycles. Data memory is single ported and only accessed by the procesor. The load latency is three cycles. The compute processor's pipelined single-precision FPU operations have a latency of 4 cycles, and the integer multiplier has a latency of 2 cycles.

1.3 Compiling StreamIt to Raw

The phases of the StreamIt compiler are described in Table 1.1, and the interaction of the phases is shown in Figure 1-5. The front-end is built on top of KOPI, an open-source compiler infrastructure for Java [15]; we use KOPI as our infrastructure because StreamIt has evolved from a Java-based syntax. We translate the StreamIt syntax into the KOPI syntax tree, and then construct the StreamIt IR (SIR) that encapsulates the hierarchical stream graph. Since the structure of the graph might be parameterized, we propagate constants and expand each stream construct to a static structure of known extent. At this point, we can calculate an execution schedule for the nodes of the stream graph.

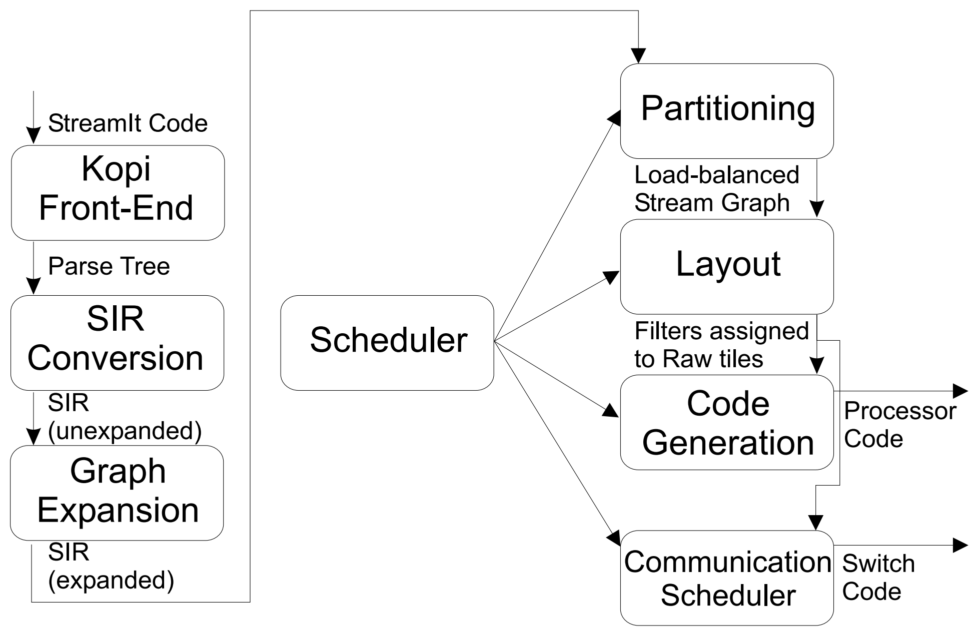


Figure 1-5: The interaction of the compiler phases. Notice that the scheduler is not a separate phase, but is used by multiple phases.

1.3.1 Scheduling

The automatic scheduling of the stream graph is one of the primary benefits that StreamIt offers, and the subtleties of scheduling and buffer management are evident throughout all of the following phases of the compiler. The scheduling is complicated by StreamIt's support for the `peek` operation, which implies that some programs require a separate schedule for initialization and for the steady-state. The steady-state schedule must be periodic–that is, its execution must preserve the number of live items on each channel in the graph (since otherwise a buffer would grow without bound.) A separate initialization schedule is needed if there is a filter with $peek > pop$, by the following reasoning. If the initialization schedule were also periodic, then after each firing it would return the graph to its initial configuration, in which there were zero live items on each channel. But a filter with $peek > pop$ leaves $peek - pop$ (a positive number) of items on its input channel after *every* firing, and thus could not be part of this periodic schedule. Therefore, the initialization schedule is separate, and non-periodic.

In the StreamIt compiler, the initialization schedule is constructed via symbolic

Phase	**Function**
KOPI Front-end	Parses syntax into a Java-like abstract syntax tree.
SIR Conversion	Converts the AST to the StreamIt IR (SIR).
Graph Expansion	Expands all parameterized structures in the stream graph.
Scheduling	Calculates initialization and steady-state execution orderings for filter firings.
Partitioning	Performs fission and fusion transformations for load balancing.
Layout	Determines minimum-cost placement of filters on grid of Raw tiles.
Communication Scheduling	Orchestrates fine-grained communication between tiles via simulation of the stream graph.
Code generation	Generates code for the tile and switch processors.

Table 1.1: Phases of the StreamIt compiler.

execution of the stream graph, until each filter has at least $peek - pop$ items on its input channel. For the steady-state schedule, there are many tradeoffs between code size, buffer size, and latency, and we are developing techniques to optimize different metrics [42]. In this thesis, we use a simple hierarchical scheduler that constructs a Single Appearance Schedule (SAS) [8] for each filter. A SAS is a schedule where each node appears exactly once in the loop nest denoting the execution order. We construct one such loop nest for each hierarchical stream construct, such that each component is executed a set number of times for every execution of its parent. In later chapters, we refer to the "multiplicity" of a filter as the number of times that it executes in a complete execution of a schedule.

Following the scheduler, the compiler has stages that are specific for communication-exposed architectures: partitioning, layout, and communication scheduling. The next three chapters of the thesis are devoted to these phases.

This thesis makes the following contributions:

- Filter fusion optimizations that combine both sequential and parallel stream segments, even if there are buffers between nodes.

- A filter fission transformation.

- Graph reordering transformations.
- Synchronization elimination transformations.
- An algorithm for laying out a filter graph onto a tiled architecture.
- A communication scheduling algorithm that manages limited communication and buffer resources.
- An end-to-end implementation of a parallelizing compiler for streaming applications.

The remainder of this thesis is organized as follows. Chapter 2 describes the partitioning phase of the compiler, including the principle enabling transformations. Chapter 3 describes the layout phase and the specific implementation for Raw. Chapter 4 describes the communication scheduler and gives the algorithm for the communication scheduling phase of the Raw backend. Chapter 5 describes code generation for the Raw backend. Chapter 6 gives results for the current implementation of the StreamIt compiler over our benchmark suite. Chapter 7 describes related work. Finally, the appendices give the source code, the layout, the execution trace, and various other items for each application in our benchmark suite.

Chapter 2

Partitioning

StreamIt provides the filter construct as the basic abstract unit of autonomous stream computation. The programmer should decide the boundaries of each filter according to what is most natural for the algorithm under consideration. While one could envision each filter running on a separate machine in a parallel system, StreamIt hides the granularity of the target machine from the programmer. Thus, it is the responsibility of the compiler to adapt the granularity of the stream graph for efficient execution on a particular architecture.

We use the word *partitioning* to refer to the process of dividing a stream program into a set of balanced computation units. Given that a maximum of N computation units can be supported in the hardware, the partitioning stage transforms a stream graph into a set of no more than N filters, each of which performs approximately the same amount of work during the execution of the program. Following this stage, each filter can be run on a separate processor to obtain a load-balanced executable.

Load-balancing is particularly important in the streaming domain, since the throughput of a stream graph is equal to the *minimum* throughput of each of its stages. This is in contrast to scientific programs, which often contain a number of stages which process a given data set; the running time is the *sum* of the running times of the phases, such that a high-performance, parallel phase can partially compensate for an inefficient phase. In mathematical terms, Amdahl's Law captures the maximum realizable speedup for scientific applications. However, for streaming programs, the maximum improvement in throughput is given by the following expression:

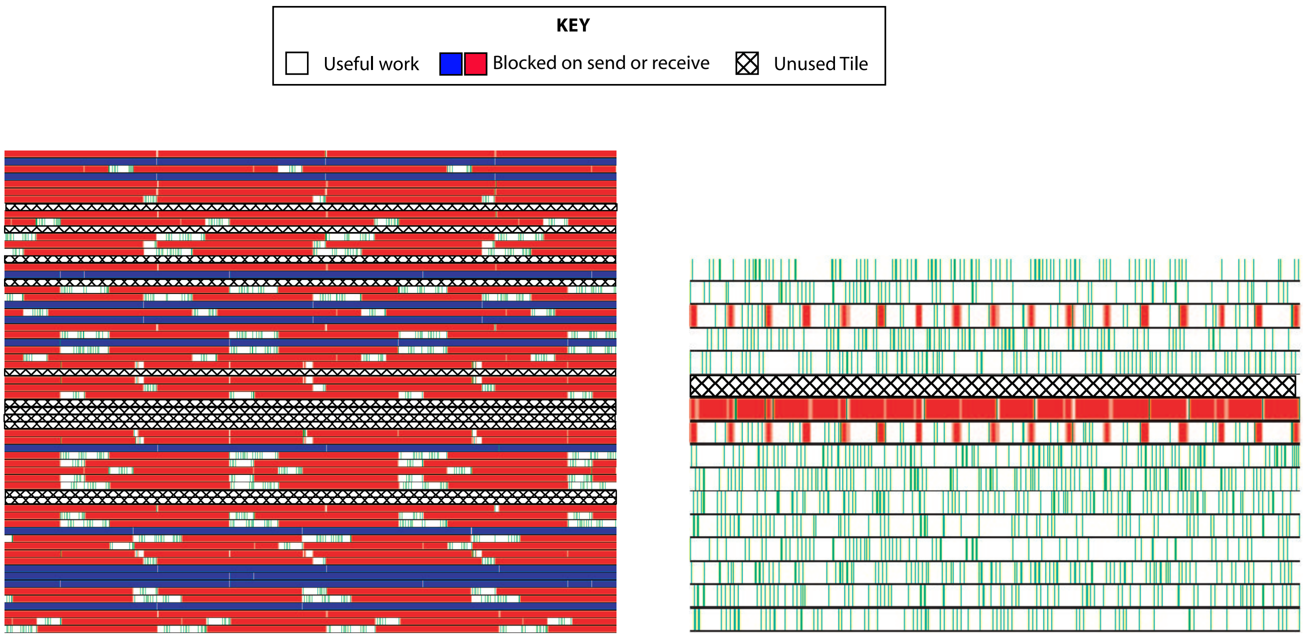
$$Maximum\ speedup(w, c) = \frac{\sum_{i=1}^{N} w_i \cdot c_i}{MAX_i(w_i \cdot c_i)}$$

where $w_1 \dots w_m$ denote the amount of work in each of the N partitions of a program, and c_i denotes the multiplicity of work segment i in the steady-state schedule. Thus, if we double the load of the heaviest node (*i.e.*, the node with the maximum $w_i \cdot c_i$), then the performance could suffer by as much as a factor of two. The impact of load balancing on performance places particular value on the partitioning phase of a stream compiler.

2.1 Overview

Our partitioner employs a set of fusion, fission, and reordering transformations to incrementally adjust the stream graph to the desired granularity. To achieve load balancing, the compiler estimates the number of instructions that are executed by each filter in one steady-state cycle of the entire program; then, computationally intensive filters can be split, and less demanding filters can be fused. Currently, a simple greedy algorithm is used to automatically select the targets of fusion and fission, based on the estimate of the work in each node.

For example, in the case of the Radar application, the original stream graph (Figure 2-2) contains 52 filters. These filters have unbalanced amounts of computation, as evidenced by the execution trace in Figure 2-1(a). The partitioner fuses all of the pipelines in the graph, and then fuses the bottom 4-way splitjoin into a 2-way splitjoin, yielding the stream graph in Figure 2-3. As illustrated by the execution trace in Figure 2-1(b), the partitioned graph has much better load balancing. In the following sections, we describe in more detail the transformations utilized by the partitioner.



(a) **Original (runs on 64 tiles).**

(b) **Partitioned (runs on 16 tiles).**

Figure 2-1: Execution traces for the (a) original and (b) partitioned versions of the Radar application. The x axis denotes time, and the y axis denotes the processor. Dark bands indicate periods where processors are blocked waiting to receive an input or send an output; light regions indicate periods of useful work. The thin stripes in the light regions represent pipeline stalls. Our partitioning algorithm decreases the granularity of the graph from 53 unbalanced tiles (original) to 15 balanced tiles (partitioned). The throughput of the partitioned graph is 11 times higher than the original.

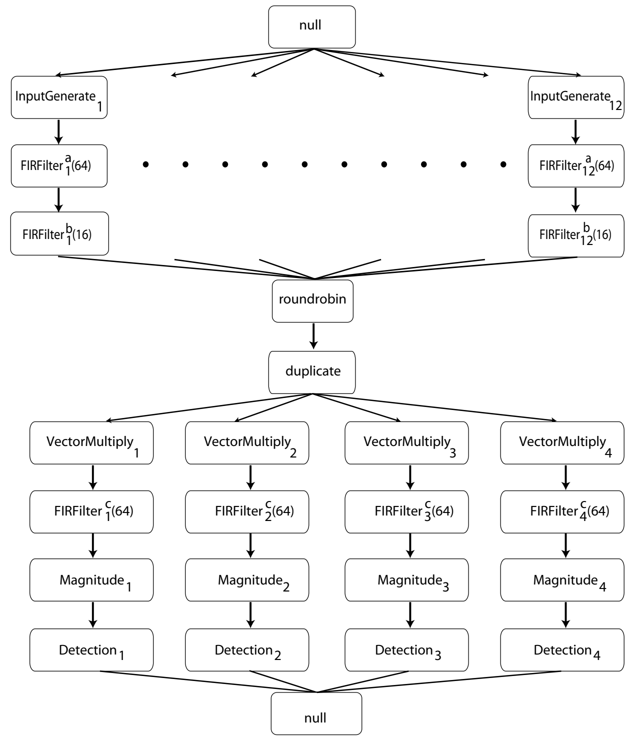


Figure 2-2: Stream graph of the original 12x4 Radar application. The 12x4 Radar application has 12 channels and 4 beams; it is the largest version that fits onto 64 tiles without filter fusion.

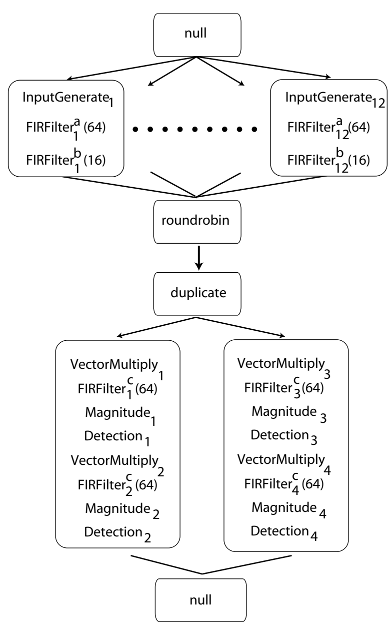


Figure 2-3: Stream graph of the load-balanced 12x4 Radar application. Vertical fusion is applied to collapse each pipeline into a single filter, and horizontal fusion is used to transform the 4-way splitjoin into a 2-way splitjoin. Figure 2-1 shows the benefit of these transformations.

2.2 Fusion Transformations

Filter fusion is a transformation whereby several adjacent filters are combined into one. Fusion can be applied to decrease the granularity of a stream graph so that an application will fit on a given target, or to improve load balancing by merging small filters so that there is space for larger filters to be split. Analogous to loop fusion in the scientific domain, filter fusion can enable other optimizations by merging the control flow graphs of adjacent nodes, thereby shortening the live ranges of variables and allowing independent instructions to be reordered.

2.2.1 Unoptimized Fusion Algorithm

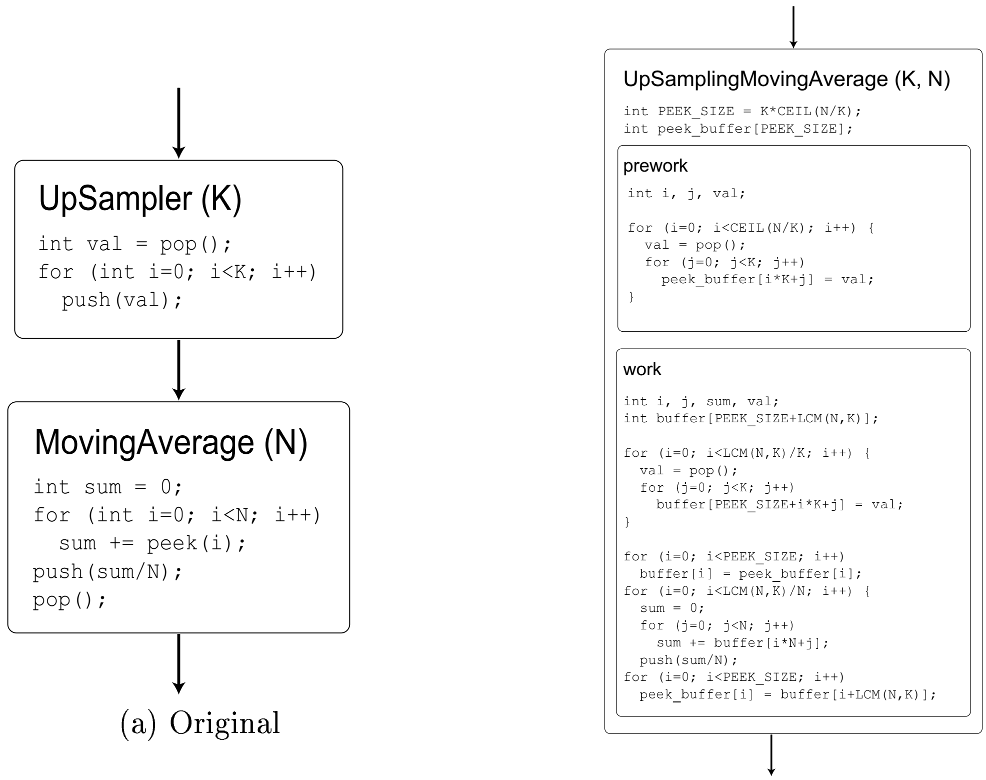
In the domain of structured stream programs, there are two types of fusion that we are interested in: *vertical fusion* for collapsing pipelined filters into a single unit, and *horizontal fusion* for combining the parallel components of a splitjoin. Given that each StreamIt filter has a constant I/O rate, it is possible to implement both vertical and horizontal fusion as a plain compile-time simulation of the execution of the stream graph. A high-level algorithm for doing so is as follows:

1. Calculate a legal initialization and steady-state schedule for the nodes of interest.

2. For each pair of neighboring nodes, introduce a circular buffer that is large enough to hold all the items produced during the initial schedule and one iteration of the steady-state schedule. For each buffer, maintain indices to keep track of the head and tail of the FIFO queue.

3. Simulate the execution of the graph according to the calculated schedules, replacing all push, pop, and peek operations in the fused region with appropriate accesses to the circular buffers.

That is, a naive approach to filter fusion is to simply implement the channel abstraction and to leverage StreamIt's static rates to simulate the execution of the



(a) Original

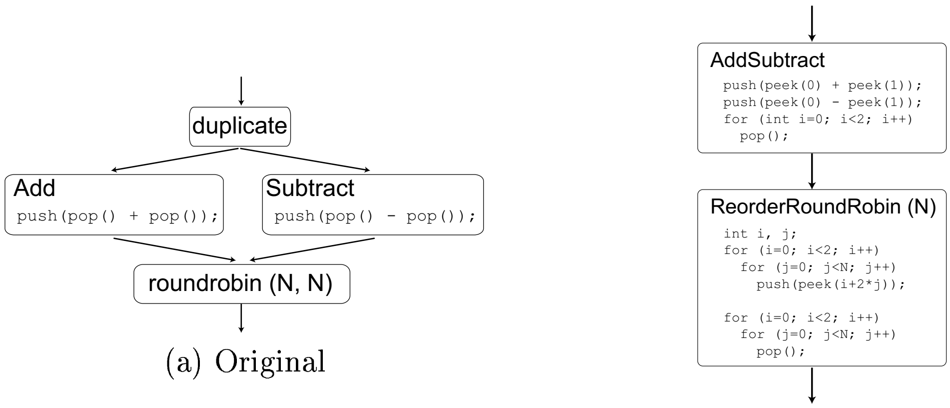
(b) Fused

Figure 2-4: Vertical fusion with buffer localization and modulo-division optimizations.

graph. However, the performance of fusion depends critically on the implementation of channels, and there are several high-level optimizations that the compiler employs to improve upon the performance of a general-purpose buffer implementation. We describe a few of these optimizations in detail in the following sections.

2.2.2 Optimizing Vertical Fusion

Figure 2-4 illustrates two of our optimizations for vertical fusion: the localization of buffers and the elimination of modulo operations. In this example, the UpSampler pushes K items on every step, while the MovingAverage filter peeks at N items but only pops 1. The effect of the optimizations are two-fold. First, buffer localization splits the channel between the filters into a local `buffer` (holding items that are transfered within `work`) and a persistent `peek_buffer` (holding items that are stored between iterations of `work`). Second, modulo elimination arranges copies between



(a) Original

(b) Fused

Figure 2-5: Horizontal fusion of a duplicate splitjoin construct with buffer sharing optimization. To fuse a SplitJoin with a Duplicate plitter, the code of the component filters is inlined into a single filter with repetition according to the steady-state schedule. However, there are some modifications: all `pop` statements are converted to `peek` statements, and the `pop`'s are performed at the end of the fused work function. This allows all the filters to see the data items before they are consumed. Finally, the RoundRobin joiner is simulated by a ReorderRoundRobin filter that re-arranges the output of the fused filter according to the weights of the Joiner.

these two buffers so that all index expressions are known at compile time, preventing the need for a modulo operation to wrap around a circular buffer.

The execution of the fused filter proceeds as follows. In the `prework` function, which is called only on the first invocation, the `peek_buffer` is filled with initial values from the UpSampler. The steady `work` function implements a steady-state schedule in which $LCM(N, K)$ items are transferred between the two original filters–these items are communicated through a local, temporary `buffer`. Before and after the execution of the MovingAverage code, the contents of the `peek_buffer` are transferred in and out of the `buffer`. If the `peek_buffer` is small, this copying can be eliminated with loop unrolling and copy propagation. Note that the `peek_buffer` is for storing items that are persistent from one firing to the next, while the local `buffer` is just for communicating values during a single firing.

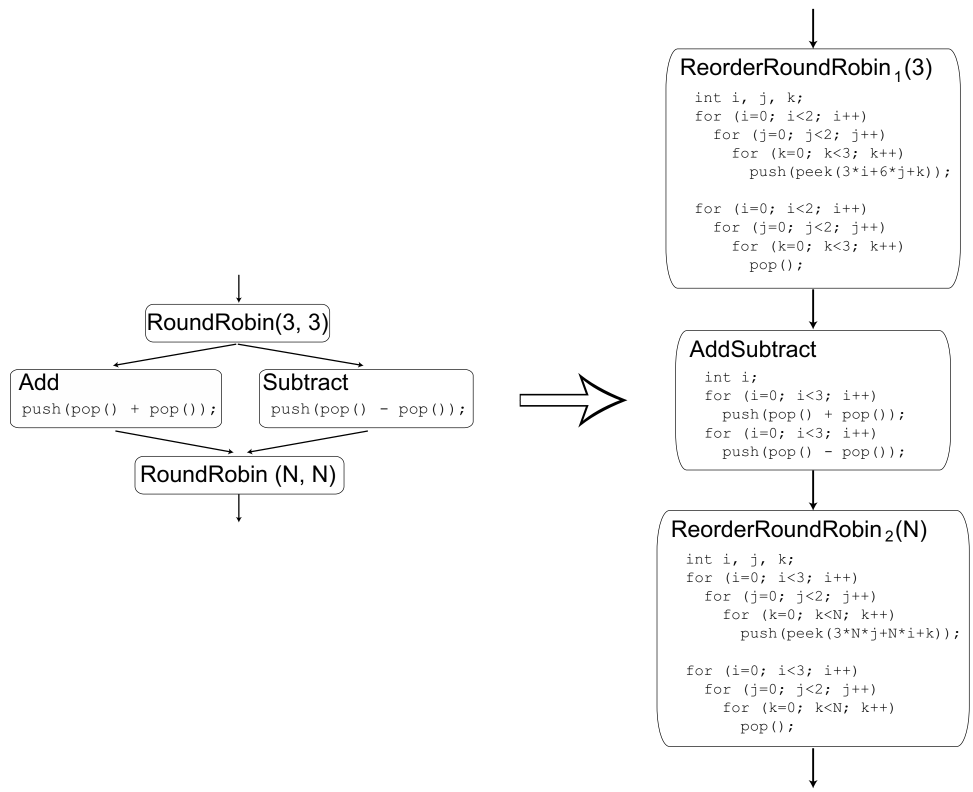


Figure 2-6: Fusion of a roundrobin splitjoin construct. The fusion transformation for splitjoins containing roundrobin splitters is similar to those containing duplicate splitters. One filter simulates the execution of a steady-state cycle in the splitjoin by inlining the code from each filter. This filter is surrounded by ReorderRoundRobin filters that recreate the reordering of the roundrobin nodes. In the above example, differences in the splitter's weights, the filter's I/O rates, and the joiner's weights adds complexity to the reordering.

2.2.3 Optimizing Horizontal Fusion

The naive fusion algorithm maintains a separate input buffer for each parallel stream in a splitjoin. However, in the case of horizontal fusion, the input buffer can be shared between the streams. Our horizontal fusion algorithm inputs a splitjoin where each component is a single filter, and outputs a pipeline of three filters: one to emulate the splitter, one to simulate the execution of the parallel filters, and one to emulate the joiner. The splitters and joiners need to be emulated in case they are roundrobin's that perform some reordering of the data items with respect to the component streams. Generally speaking, the fusion of the parallel components is similar to that of vertical fusion–a sequential steady-state schedule is calculated, and the component work functions are inlined and executed within loops.

The details of our horizontal fusion transformation depend on the type of the splitter in the construct of interest. There are two cases:

1. For **duplicate** splitters, the `pop` expressions from component filters need to be converted to `peek` expressions so that items are not consumed before subsequent filters can read them (see Figure 2-5). Then, at the end of the fused `work` function, the items consumed by an iteration of the splitjoin are popped from the input channel. Also, the splitter itself performs no reordering of the data, so it translates into an Identity filter that can be removed from the stream graph. This fusion transformation is valid even if the component filters peek at items which they do not consume.

2. For **roundrobin** splitters, the `pop` expressions in component filters are left unchanged, and the roundrobin splitter is emulated in order to reorder the data items according to the weights of the splitter and the consumption rates of the component streams (see Figure 2-6). However, this is is invalid if any of the component filters peek at items which it does not consume, since the interleaving of items on the input stream of the fused filter prevents each component from having a continuous view of the items that are intended for it. Thus, we only apply this transformation when all component filters have $peek = pop$.

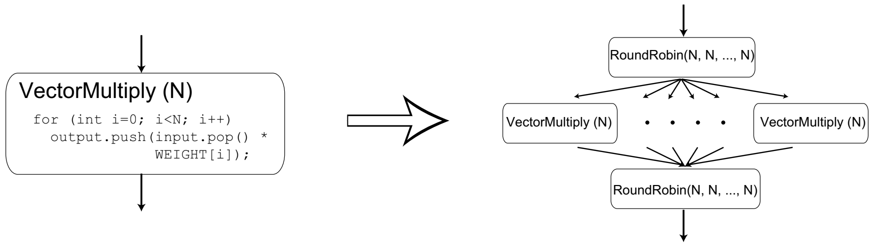


Figure 2-7: Fission of a filter that does not peek. For filters such as a VectorMultiply that consumes every item they look at, horizontal fission consists of embedding copies of the filter in a K-way roundrobin splitjoin. The weights of the splitter and joiner are set to match the *pop* and *push* rates of the filter, respectively.

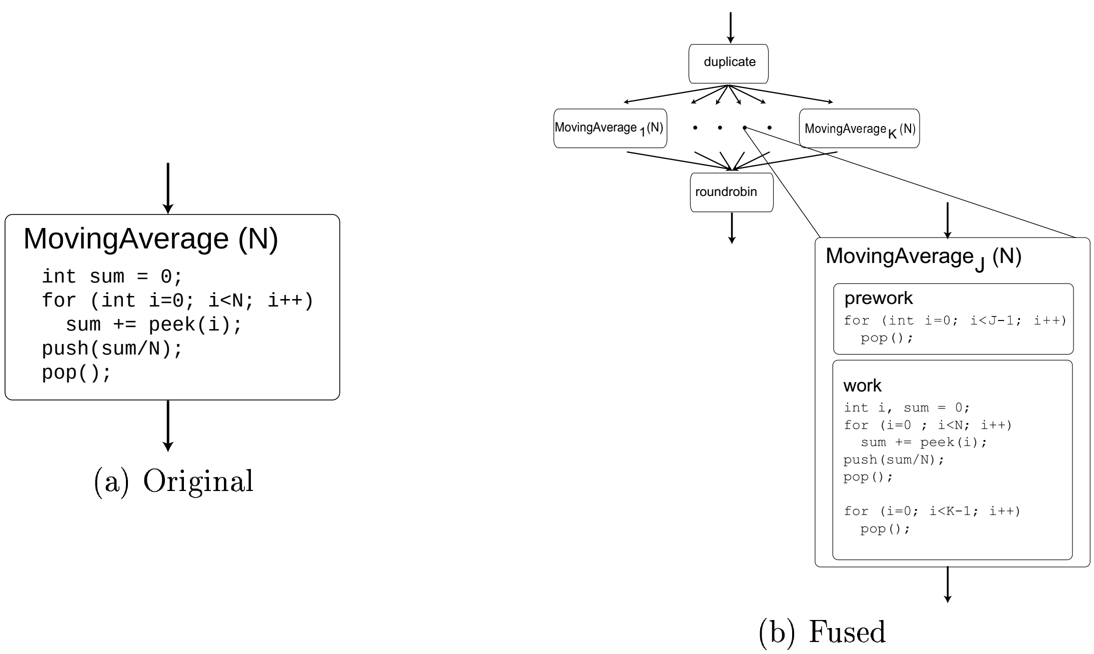


Figure 2-8: Fission of a filter that peeks. Since the MovingAverage filter reads items that it does not consume, the duplicated versions of the filter need to access overlapping portions of the input stream. For this reason, horizontal fission creates a duplicate splitjoin in which each component filter has additional code to filter out items that are irrelevant to a given path. This decimation occurs in two places: once in the `prework` function, to disregard items considered by previous filters on the first iteration of the splitjoin, and once at the end of the steady `work` function, to account for items consumed by other components.

2.3 Fission Transformations

Filter fission is the analog of parallelization in the streaming domain. It can be applied to increase the granularity of a stream graph to utilize unused processor resources, or to break up a computationally intensive node for improved load balancing.

2.3.1 Vertical Fission

Some filters can be split into a pipeline, with each stage performing part of the `work` function. In addition to the original input data, these pipelined stages might need to communicate intermediate results from within `work`, as well as fields within the filter. This scheme could apply to filters with state if all modifications to the state appear at the top of the pipeline (they could be sent over the data channels), or if changes are infrequent (they could be sent via StreamIt's messaging system.) Also, some state can be identified as induction variables, in which case their values can be reconstructed from the `work` function instead of stored as fields. We have yet to automate vertical filter fission in the StreamIt compiler.

2.3.2 Horizontal Fission

We refer to "horizontal fission" as the process of distributing a single filter across the parallel components of a splitjoin. We have implemented this transformation for "stateless" filters–that is, filters that contain no fields that are written on one invocation of `work` and read on later invocations. Let us consider such a filter F with steady-state I/O rates of *peek*, *pop*, and *push*, that is being parallelized into an K-way splitjoin. There are two cases to consider:

1. If $peek = pop$, then F can simply be duplicated K ways in the splitjoin (see Figure 2-7). The splitter is a roundrobin that routes *pop* elements to each copy of F, and the joiner is a roundrobin that reads *push* elements from each component. Since F does not peek at any items which it does not consume, its code does not need to be modified in the component streams–we are just

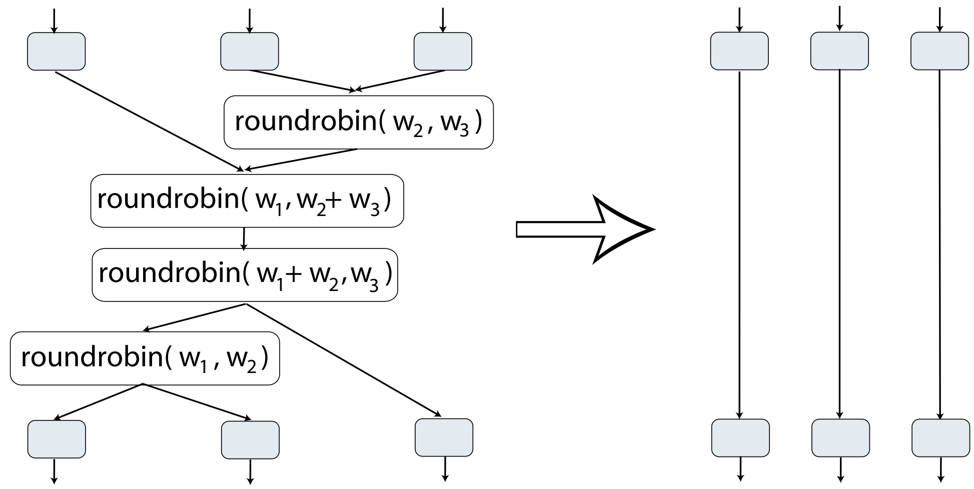


Figure 2-9: Synchronization removal. If there are neighboring splitters and joiners with matching rates, then the nodes can be removed and the component streams can be connected. The example above is drawn from a subgraph of the 3GPP application; the compiler automatically performs this transformation to expose parallelism and improve the partitioning.

distributing the invocations of F.

2. If $peek > pop$, then a different transformation is applied (see Figure 2-8). In this case, the splitter is a duplicate, since the component filters need to examine overlapping parts of the input stream. The i'th component has a steady-state work function that begins with the work function of F, but appends a series of $(K-1) * pop$ `pop` statements in order to account for the data that is consumed by the other components. Also, the i'th filter has a `prework` function that pops $(i-1) * pop$ items from the input stream, to account for the consumption of previous filters on the first iteration of the splitjoin. As before, the joiner is a roundrobin that has a weight of $push$ for each stream.

2.4 Reordering Transformations

There are a multitude of ways to reorder the elements of a stream graph so as to facilitate fission and fusion transformations. For instance, in synchronization removal, neighboring splitters and joiners with matching weights can be eliminated (Figure 2-9). Synchronization removal is especially valuable in the context of libraries–many

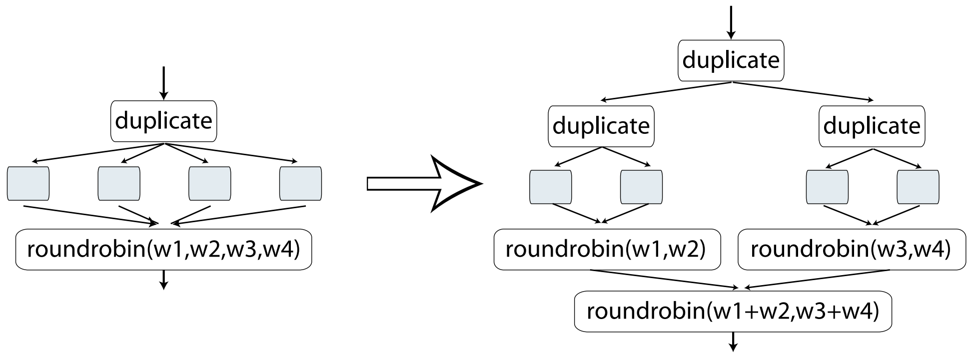


Figure 2-10: Breaking a splitjoin into hierarchical units. Though our horizontal fusion algorithms work on the granularity of an entire splitjoin, it is straightforward to transform a large splitjoin into a number of smaller pieces, as shown here. Following this transformation, the fusion algorithms can be applied to obtain an intermediate level of granularity. This technique was employed to help load-balance the Radar application (see Chapter 6).

distinct components can employ splitjoins for processing interleaved data streams, and the modules can be composed without having to synchronize all the streams at each boundary. A splitjoin construct can be divided into a hierarchical set of splitjoins to enable a finer granularity of fusion (Figure 2-10); and identical stateless filters can be pushed through a splitter or joiner node if the weights are adjusted accordingly. (Figure 2-11). A detailed anaylsis of our reordering transformations is beyond the scope of this thesis.

2.5 Automatic Partitioning

In order to drive the partitioning process, we have implemented a simple greedy algorithm that performs well on most applications. The algorithm analyzes the `work` function of each filter and estimates the number of cycles required to execute it. The current work estimation implementation is rather naive and we believe that a more accurate work estimator will increase performance.

2.5.1 Greedy Algorithm

In the case where there are fewer filters than tiles, the partitioner considers the filters in decreasing order of their computational requirements and attempts to split them

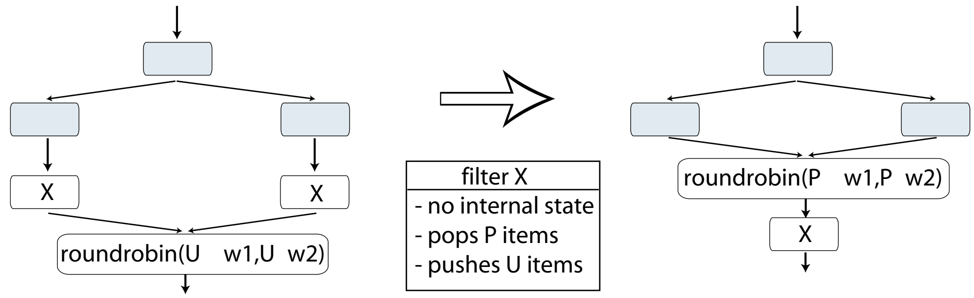


Figure 2-11: Filter hoisting. This transformation allows a stateless filter to be moved across a joiner node if its *push* value evenly divides the weights of the joiner.

using the filter fission algorithm described above. Fission proceeds until there are enough filters to occupy the available machine resources, or until the heaviest node in the graph is not amenable to a fission transformation. Generally, it is not beneficial to split nodes other than the heaviest one, as this would introduce more synchronization without alleviating the bottleneck in the graph.

If the stream graph contains more nodes than the target architecture, then the partitioner works in the opposite direction and repeatedly fuses the least demanding stream construct until the graph will fit on the target. The work estimates of the filters are tabulated hierarchically and each construct (*i.e.,* pipeline, splitjoin, and feedbackloop) is ranked according to the sum of its children's computational requirements. At each step of the algorithm, an entire stream construct is collapsed into a single filter. The only exception is the final fusion operation, which only collapses to the extent necessary to fit on the target; for instance, a 4-element pipeline could be fused into two 2-element pipelines if no more collapsing was necessary.

Despite its simplicity, this greedy strategy works well in practice because most applications have many more filters than can fit on the target architecture; since there is a long sequence of fusion operations, it is easy to compensate from a short-sighted greedy decision. However, we can construct cases in which a greedy strategy will fail. For instance, graphs with wildly unbalanced filters will require fission of some components and fusion of others; also, some graphs have complex symmetries where fusion or fission will not be beneficial unless applied uniformly to each component of the graph. We are working on improved partitioning algorithms that take these measures into account.

2.6 Summary

In this chapter we discussed the partitioning phase of the StreamIt compiler. The goal of partitioning is to transform the stream graph into a set of load-balanced computational units. If there are N computation nodes in the target architecture, the partitioning stage will adjust the stream graph such that there are no more than N filters that are approximately load-balanced. To facilitate partitioning, we employ both fusion and fission transformations. The fusion transformation merges streams into a single filter and the fission transformation splits a stream into multiple, parallel filters. Finally, we described the current version of the algorithm that drives the partitioning decisions. In the next phase of the StreamIt compiler, layout, the filters of the load-balanced, partitioned stream graph are assigned to Raw tiles.

Chapter 3

Layout

The goal of the layout phase is to assign nodes in the stream graph to computation nodes in the target architecture while minimizing the communication and synchronization present in the final layout. The layout phase assigns exactly one node in the stream graph to one computation node in the target. This phase assumes that the given stream graph will fit onto the computation fabric of the target and that the filters are load balanced. These requirements are satisfied by the partitioning phase described above.

Classically, layout (or placement) algorithms have fallen into two categories: constructive initial placement and iterative improvement [25]. Both try to minimize a predetermined cost function. In constructive initial placement, the algorithm calculates a solution from scratch, using the first complete placement encountered. Iterative improvement starts with an initial random layout and repeatedly perturbs the placement in order to minimize the cost function.

The layout phase of the StreamIt compiler is implemented using a modified version of the simulated annealing algorithm[23], a type of iterative improvement. We will explain the modifications below. Simulated annealing is a form of stochastic hill-climbing. Unlike most other methods for cost function minimization, simulated annealing is suitable for problems where there are many local minima. Simulated annealing achieves its success by allowing the system to go uphill with some probability as it searches for the global minimum. As the simulation proceeds, the probability of climbing uphill decreases.

We selected simulated annealing for its combination of performance and flexibility. To adapt the layout phase for a given architecture, we supply the simulated annealing algorithm with three architecture-specific parameters: a cost function, a perturbation function, and the set of legal layouts. To change the compiler to target one tiled architecture instead of another, these parameters should require only minor modifications.

The cost function should accurately measure the added communication and synchronization generated by mapping the stream graph to the communication model of the target. Due to the static qualities of StreamIt, the compiler can provide the layout phase with exact knowledge of the communication properties of the stream graph. The terms of the cost function can include the counts of how many items travel over each channel during an execution of the steady-state. Furthermore, with knowledge of the routing algorithm, the cost function can infer the intermediate hops for each channel. For architectures with non-uniform communication, the cost of certain hops might be weighted more than others. In general, the cost function can be tailored to suit a given architecture.

Note that it is impractical to perform an exhaustive search of all the possible layouts for a 16 tile Raw configuration. For 16 tiles, we would have to examine approximately $2 * 10^{13}$ possible layouts. We would have to perform some kind of cost analysis of each layout. Even if the cost analysis consumed only one cycle, on a 1 GHz machine the search would require 5 1/2 hours. For the simulated annealing algorithm we describe below, on average 5000 layouts are examined, a more reasonably number.

We also could have formulated the layout problem as 0/1 integer programming problem. 0/1 integer programming would give us an optimal solution to the layout problem, but has exponential worst-case complexity. As we will show, our modified simulated annealing implementation performs quite well for our benchmark suite and we feel that there is no reason to consider an optimal solution framework. Furthermore, a 0/1 integer programming implementation would lack the retargetability of simulated annealing.

3.1 Layout for Raw

For Raw, the layout phase maps nodes in the stream graph to the tile processors. Each filter is assigned to exactly one tile, and no tile holds more than one filter. However, the ends of a splitjoin construct are treated differently; each splitter node is folded into its upstream neighbor, and neighboring joiner nodes are collapsed into a single tile (see Section 4.1). Thus, joiners occupy their own tile, but splitters are integrated into the tile of their upstream filter or joiner.

Due to the properties of the static network and the communication scheduler (see Section 4.1), the layout phase does not have to worry about deadlock. All assignments of nodes to tiles are legal. This gives simulated annealing the flexibility to search many possibilities and simplifies the layout phase. The perturbation function used in simulated annealing simply swaps the assignment of two randomly chosen tile processors.

3.1.1 Cost Function

After some experimentation, we arrived at the following cost function to guide the layout on Raw. We let *channels* denote the pairs of nodes $\{(src_1, dst_1)\ldots(src_N, dst_N)\}$ that are connected by a channel in the stream graph; $layout(n)$ denote the placement of node n on the Raw grid; and $route(src, dst)$ denote the path of tiles through which a data item is routed in traveling from tile src to tile dst. In our implementation, the *route* function is a simple dimension-ordered router that traces the path from src to dst by first routing in the X dimension and then routing in the Y dimension. Given fixed values of *channels* and *route*, our cost function evaluates a given layout of the stream graph:

$$cost(layout) = \sum_{(src,dst)\,\in\, channels} \mathbf{items}(src, dst) \cdot (\mathbf{hops}(path) + 10 \cdot \mathbf{sync}(path))$$

$$\text{where } path = route(layout(src), layout(dst))$$

In this equation, $\mathbf{items}(src, dst)$ gives the number of data words that are transfered from *src* to *dst* during each steady state execution, $\mathbf{hops}(p)$ gives the number of intermediate tiles traversed on the path p, and $\mathbf{sync}(p)$ estimates the cost of the synchronization imposed by the path p. We calculate $\mathbf{sync}(p)$ as the number of tiles along the route that are assigned a stream node plus the number of tiles along the route that are involved in routing *other* channels.

With the above cost function, we heavily weigh the added synchronization imposed by the layout. For Raw, this metric is far more important than the length of the route because neighbor communication over the static network is cheap. If a tile that is assigned a filter must route data items through it, then it must synchronize the routing of these items with the execution of its `work` function. Also, a tile that is involved in the routing of many channels must serialize the routes running through it. Both limit the amount of parallelism in the layout and need to be avoided.

Initially we used a slightly different cost function than the function given above. Our first cost function cubed $\mathbf{sync}(p)$, and in the calculation of $\mathbf{sync}(p)$ weighted more heavily the cost of tiles assigned to filters along the route (versus non-assigned tiles). Our intuition was that the synchronization added from routing through assigned tiles is by far the most important factor. After some analysis, we came to the conclusion that this initial cost function was not smooth enough. More precisely, small changes in the layout could lead to an enormous change in the cost function. This prevented the algorithm from backing out of local minima due to the large cost difference.

In contrast, the current cost function does not have such a large delta between a local minimum and its peak. This allows the simulated annealing algorithm to climb out and explore other layout options. The current cost function still weights $\mathbf{sync}(p)$ heavily, but has been scaled down to an appropriate level.

3.1.2 Modifications to Simulated Annealing

The simulated annealing implementation used in the StreamIt compiler was adopted from [44] and includes some important modifications. First, the initial layout is not entirely random. We found that a random initial layout could lead the algorithm to wallow in local minima. This was especially the case for long pipelines that have a zero-cost layout on Raw. Instead, for the initial layout we place a depth-first traversal of the stream graph along the raw tiles, starting at the top-left tile and snaking across rows (see Algorithm 2 and Figure 3-2(a)). In this way, pipelines are placed perfectly by the initial layout.

Additionally, we found that in rare cases simulated annealing did not always finish with the best layout. It sometimes found the layout with the minimum cost early in the search and backed out of it to settle on a different, higher-cost, local minimum. To prevent this, we cache the layout with the minimum cost that was encountered during the simulated annealing search and use it as the final layout. The algorithm ends if a layout with zero cost is found.

Most importantly, we found that the layout problem was sometimes too constrained for the simulated annealing algorithm. It was difficult for the algorithm to back out of a local minimum late in the simulation. Conceptually, local minima are spaced too far apart for the simulated annealing algorithm to back out of late in the algorithm. Simply changing the temperature multiplier did not help the situation. The problem was that it took too long for the annealing algorithm to decide which minimum it would descend. The first half of the algorithm was spend oscillating between minima, with no significant drop in cost. By the time it settled on a path to descend, it was too late to reverse the decision.

We found that running multiple, separate iterations of the simulated annealing algorithm solved the problem. In this case, the final layout of the previous iteration becomes the initial layout for the new iteration. We cache the minimum layout over all the iterations and use it as the final layout. Now, each iteration has the chance to settle on a different (possibly local) minimum because when restarting the annealing we use the high temperature to search for a minimum. After experimentation, we found that running two annealing iterations for a 16 tile Raw configuration produced excellent layouts for all our benchmarks and test programs. Although this doubled the running time of the layout phase, the layout time for a 16 tile Raw configuration is under 10 seconds.

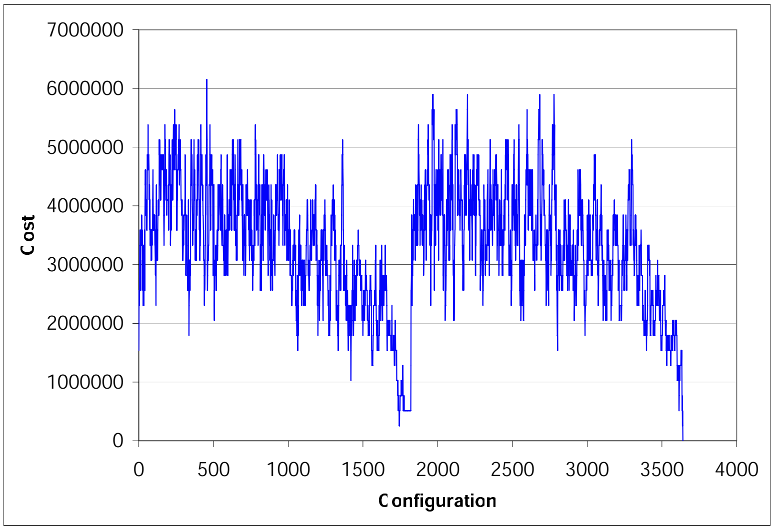


Figure 3-1: Estimated cost for successive accepted configurations of the load-balanced FFT layout as evaluated by the simulated annealing algorithm.

The complete, modified algorithm is given in Algorithms 1-4. All constants in the code were initially set to the value given in [44] and adjusted based on the results of the algorithm. Most constants did not change. For the decay rate (.9), the number of perturbations per temperature (100), and the temperature limits (90% and 1%), we found that the constants given in [44] gave the best results.

Figure 3-1 illustrates how the cost metric varies over time during a run of the simulated annealing algorithm for the FFT application. The figure illustrates that the cost converges to 0, causing layout to stop. In the figure one can clearly see two iterations of the simulated annealing algorithm. At the start of the second iteration, the cost increases rapidly as the algorithm accepts perturbations of higher cost. This breaks out of the local minima reached by the first iteration, allowing the algorithm to reach the zero-cost layout. Notice also that each iteration spends a significant amount of time searching for a minimum to descend.

Figure 3-2 shows the initial layout of the FFT application on the left and the final, zero-cost layout on the right. Figure E-2 gives the stream graph after partitioning. For the FFT application, the layout determined by our algorithm has a throughput that exceeds that of the initial layout by a factor of 10. The remaining applications in our benchmark suite obtain similar performance improvements from the layout

Algorithm 1 Layout Algorithm on Raw

SimulatedAnealingAssign(G, M) assigns the filters and coalesced joiners of the stream graph G to Raw tiles. Each is assigned to exactly one tile. M describes the Raw configuration. $E(C)$ denotes the cost function applied to the layout C.

```
   Let C_init ← InitialPlacement(G, M) (see Algorithm 2).
   Let C_old ← C_init.
   if E(C_init) = 0 then
     return C_init.
   end if
   Let T ← InitialTemp(C_init). (see Algorithm 3)
   Let T_f ← FinalTemp(C_init). (see Algorithm 4)
   Let E_min ← 0.
   Let C_min ← C_init.
   for i = 1 to 2 do
     repeat
**     for j = to 100 do
         Let C_new be C_old with the assignment of a pair of tiles swapped.
         if E(C_new) = 0 then
           return C_new.
         end if
         if E(C_new) < E_min then
           E_min ← E(C_new).
           C_min ← C_new.
         end if
         if E(C_new) < E(C_old) then
           P = 1.
         else
           P = e^((E(C_old) − E(C_new)) / T).
         end if
         Randomly choose a number 0.0 ≤ R ≤ 1.0.
         if R < P then
           C_old ← C_new.
         end if
       end for
       Set T ← 9/10 * T.
     until T < T_f
   end for
   Return C_min.
```

Algorithm 2 Initial Placement

InitialPlacement(G, M). Given the stream graph G and the Raw configuration M, return the initial placement of G on M.

- Let D be a sequence of filters and coalesced joiners in G ordered by a depth-first traversal of G.
- Let R be the number of rows in M.
- Let C be the number of columns in M.

```
for r = 0 to R − 1 do
  if r is even then
    for c = 0 to C − 1 do
      assign the next node of D to tile r, c
    end for
  else
    for c = C − 1 downto 0 do
      assign the next node of D to tile r, c
    end for
  end if
end for
```

Algorithm 3 Calculation of Initial Temperature

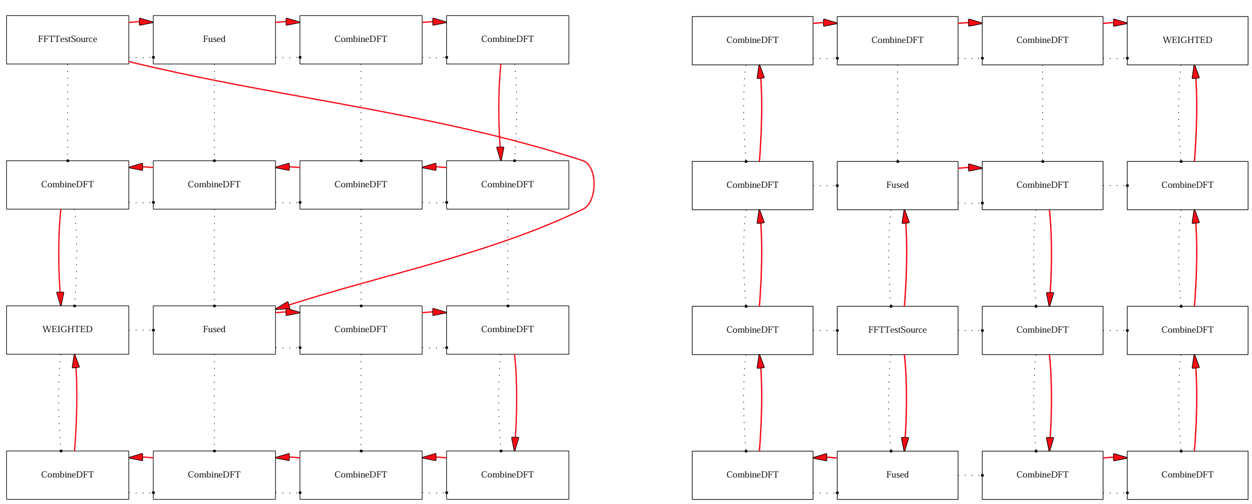
InitialTemp(C_{init}) determines the initial temperature of the algorithm, adapted from [44].

1. Set $T \leftarrow 1.0$.
2. Repeat the following until the at least 90% of new configurations are accepted in step 2c or the steps have been repeated 200 times.
   (a) Set $T \leftarrow 2 * T$.
   (b) Set $C_{old} \leftarrow C_{init}$.
   (c) Perform the **for** loop of **SimulatedAnealingAssign** noted with a $**$.
3. Return T.

Algorithm 4 Calculation of Final Temperature

FinalTemp(C_{init}) determines the termination temperature for the algorithm, adapted from [44].

1. Set $T \leftarrow 1.0$.
2. Repeat the following until the at most 1% of new configurations are accepted in step 2c or the steps have been repeated 200 times.
   (a) Set $T \leftarrow \frac{1}{2} * T$.
   (b) Set $C_{old} \leftarrow C_{init}$.
   (c) Perform the **for** loop of **SimulatedAnealingAssign** noted with a $**$.
3. Return T.



(a) **Initial Layout.**

(b) **Final Layout.**

Figure 3-2: (a) The initial layout. Notice the source is located at the top-left and connected filters of a single pipeline are placed on neighboring tiles. (b) The final layout, with zero cost.

algorithm.

3.2 Summary

In this chapter we presented the layout phase of the StreamIt compiler. It's goal is to assign nodes of the stream graph to computation units of the target architecture. We use simulated annealing to drive the layout phase. We were attracted to simulated annealing by both its retargetablility and its performance. We described the cost function used by simulated annealing for the Raw backend. Finally, this chapter describes the modifications we were forced to make to the core simulated annealing algorithm and gave pseudo-code of the modified algorithm.

Chapter 4

Communication Scheduler

With the nodes of the stream graph assigned to computation nodes of the target, the next phase of the compiler must map the communication explicit in the stream graph to the interconnect of the target. This is the task of the communication scheduler. The communication scheduler maps the infinite FIFO abstraction of the stream channels to the limited resources of the target. Its goal is to avoid deadlock and starvation while utilizing the parallelism explicit in the stream graph.

The exact implementation of the communication scheduler is tied to the communication model of the target. The simplest mapping would occur for hardware with support for an end-to-end, infinite FIFO abstraction. The scheduler need only determine the sender and receiver of each data item. This information is easily calculated from the weights of the splitters and joiners. As the communication model becomes more constrained, the communication scheduler becomes more complex, requiring analysis of the stream graph. For targets implementing a finite, blocking nearest-neighbor communication model, the exact ordering of tile execution must be specified.

Due to the static nature of StreamIt, the compiler can statically orchestrate the communication resources. As described in Section 1.3, we create an initialization schedule and a steady-state schedule that fully describe the execution of the stream graph. The schedules can give us an order for execution of the graph if necessary. One can generate orderings to minimize buffer length, maximize parallelism, or minimize latency.

Deadlock must be carefully avoided in the communication scheduler. Each architecture requires a different deadlock avoidance mechanism. A detailed discussion of deadlock is beyond the scope of this thesis. In general, deadlock occurs when there is a circular dependence on resources. A circular dependence can surface in the stream graph or in the routing pattern of the layout. If the architecture does not provide sufficient buffering, the scheduler must serialize all potentially deadlocking dependencies.

4.1 Communication Scheduler for Raw

The communication scheduling phase of the StreamIt compiler maps StreamIt's channel abstraction to Raw's static network. As mentioned in Section 1.2, Raw's static network provides optimized, nearest neighbor communication. Tiles communicate using buffered, blocking sends and receives. It is the compiler's responsibility to statically orchestrate the explicit communication of the stream graph while preventing deadlock.

To statically orchestrate the communication of the stream graph, the communication scheduler simulates the firing of nodes in the stream graph, recording the communication as it simulates. The simulation does not model the code inside each filter; instead it assumes that each filter fires instantaneously. This relaxation is possible because of the flow control of the static network–since sends block when a channel is full and receives block when a channel is empty, the compiler needs only to determine the ordering of the sends and receives rather than arranging for a precise rendezvous between sender and receiver.

In the current implementation we simulate the execution of the stream graph using a *push* schedule. We define a push schedule [22] as a schedule that always fires the node that is the furthest downstream in the stream graph at any given time. A node can only fire if it has enough items in its incoming buffer. Initially, we planned to simulate a single-appearance schedule. However, when using a SAS we calculated that for some of our applications, the incoming buffer size would be too large to fit in the data cache of a Raw tile. So, we chose to simulate a push schedule because the incoming buffer of each node is much smaller than that of a single appearance schedule. In fact, with a push schedule we obtain the minimal size for the incoming buffer of each filter [22]. The incoming buffer size of each filter is approximately equal to the number of items peeked by the filter (see Section 5.1.2 for the exact equation).

4.1.1 Joiner Deadlock Avoidance

Special care is required in the communication scheduler to avoid deadlock in splitjoin constructs. Figure 4-1 illustrates a case where the naive implementation of a splitjoin would cause deadlock in Raw's static network. The fundamental problem is that some splitjoins require a buffer of values at the joiner node–that is, the joiner outputs values in a different order than it receives them. This can cause deadlock on Raw because the buffers between channels can hold only four elements; once a channel is full, the sender will block when it tries to write to the channel. If this blocking propagates the whole way from the joiner to the splitter, then the entire splitjoin is blocked and can make no progress.

To avoid this problem, the communication scheduler implements internal buffers in the joiner node instead of exposing the buffers on the Raw network (see Figure 4-2). As the execution of the stream graph is simulated, the scheduler records the order in which items arrive at the joiner, and the joiner is programmed to fill its internal buffers accordingly. At the same time, the joiner outputs items according to the ordering given by the weights of the roundrobin. That is, the sending code is interleaved with the receiving code in the joiner; no additional items are input if a buffered item can be written to the output stream. To facilitate code generation (Chapter 5), the maximum buffer size of each internal buffer is recorded during the simulation.

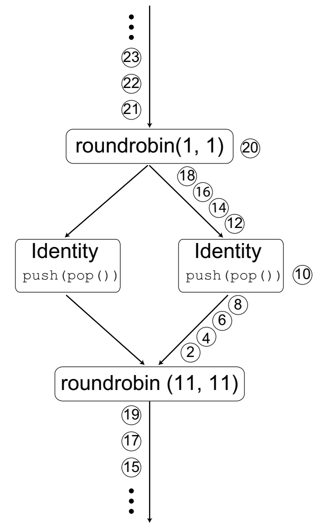


Figure 4-1: Example of deadlock in a splitjoin. As the joiner is reading items from the stream on the left, items accumulate in the channels on the right. On Raw, senders will block once a channel has four items in it. Thus, once 10 items have passed through the joiner, the system is deadlocked, as the joiner is trying to read from the left, but the stream on the right is blocked.

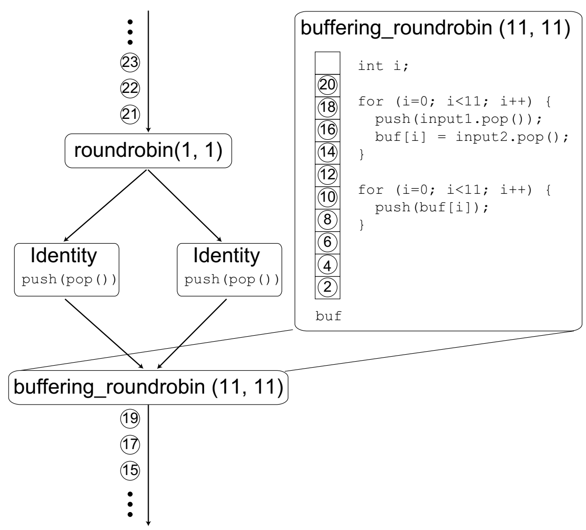


Figure 4-2: Fixing the deadlock with a buffering joiner. The buffering_roundrobin is an internal StreamIt construct (it is not part of the language) which reads items from its input channels in the order in which they arrive, rather than in the order specified by its weights. The order of arrival is determined by a simulation of the stream graph's execution; thus, the system is guaranteed to be deadlock-free, as the order given by the simulation is feasible for execution on Raw. To preserve the semantics of the joiner, the items are written to the output channel from the internal buffers in the order specified by the joiner's weights. The ordered items are sent to the output as soon as they become available.

4.2 Implementation

In this section we will give a rigorous and near-complete implementation for the communication scheduling phase of the Raw backend. This description lacks some features of the actual implementation used in the StreamIt compiler. For clarity, we will not describe joiner coalescing and the `Identity` filter removal optimization. Each is tightly ingrained in the implementation of the communication scheduler and would complicate the discussion. We further restrict that only scalars are being passed over the channels. Finally, we neglect to represent all the state that is recorded for later phases of the compiler. For example, we do not describe the recording of the maximum size of a joiner's internal buffer.

4.2.1 Switch Instruction Format

Raw's switch instructions consist of a processor component and a list of routes for the static networks. In the implementation, we are only using the first static network. At this time, we neglect the processor component of the switch instruction. We are using the switch to route data only. In the implementation of switch code compression (Section 5.1.2), we use the processor component of the instruction, but for now it is a `nop`.

The combination of processor and route components of a single instruction are subject to the following constraints [39]:

- The source of the processor component can be a register or a switch port, but the destination must be a register.
- The source of a route can be a register or a switch port but the destination must always be a switch port.
- Two values cannot be routed to the same location.
- If there are multiple reads from the register file, they must use the same register. This is because there is only one read port.

`$csto` is the FIFO buffer from the compute processor to the switch, and `$csti` is the FIFO buffer from the switch to the compute processor. `$cNi`, `$cEi`, `$cSi`, and `$cWi` are the FIFO buffers *from* the switch's north, east, south, and west neighbors, respectively. `$cNo`, `$cEo`, `$cSo`, and `$cWo` are the FIFO buffers *to* the switch's north, east, south, and west neighbors, respectively.

4.2.2 Preliminaries

- G represents the stream graph of the application.
- *root* denotes the root of G, because of the structure of StreamIt programs, all non-null programs have a unique root.
- L is a set containing the nodes of G mapped to raw tiles by the layout algorithm. In this case, everything but splitter nodes.
- T is a set representing the tiles of the Raw configuration.
- *tile*(a) where $a \in L$, returns $t \in T$ where a is mapped to t by the layout phase.
- Assume that the `prework` function always appears in the initialization schedule even if it is empty for a given filter.
- For each filter n, we define the fields, *n.prePeek*, *n.prePop*, *n.prePush*, *n.peek*, *n.pop*, and *n.push*.
- For each splitter s we define the field *s.type* to denote the type of the splitter, either *duplicate* or *roundrobin*.
- $C_i[n]$ holds the multiplicity (see Section 1.3.1) for the initialization schedule for node n in the stream graph as computed by the scheduler for a push schedule [22].
- $C_s[n]$ holds the multiplicity for the initialization schedule *and* the steady state for node n in the stream graph as computed by the scheduler for a push schedule.

- Let *downstream*(n) and *upstream*(n) denote the set containing all downstream neighbors of n in G and all upstream neighbors of n in G, respectively.

- For nodes a and b where $a, b \in G$ and $a, b \in L$, let *getRoute*(a, b) return the route from a to b in the *layout*. It returns a sequence of intermediate hops of the form $\langle s, t \rangle$, where s sending to t is an intermediate hop of the route and $s, t \in T$.

- For neighboring tiles a and b where $a, b \in T$, let *getDirection*(a, b) return a string representing the direction from a to b, either `N`, `E`, `S`, or `W`.

- Let the operator $+$ denote string concatenation.

- The *global* state of the simulation is stored in the following structures:

  - *Buffer*$[f]$ stores number of items currently in the buffer for each filter of the stream graph. *Buffer*$[f]$ will return the number of items in the incoming buffer for filter f.

  - *SwitchCode*$[t]$ holds the switch code schedule for tile $t \in T$ We add entries to the end of the schedule during the simulation with the @ operator.

  - Associated with each joiner is a set of internal buffers, one for each incoming channel of the buffer. When referring to a buffer we will describe it by the endpoints of the channel it represents. *JoinBuf*$[j][b]$ denotes an integer representing the number of items in buffer b of joiner j.

  - For each joiner j we create an *internal buffer schedule*, *JBufSch*$[j]$. This schedule is used by the code generation phase to produce the code for joiners. For each joiner the schedule is represented as a sequence. Initially empty, we add entries to the end of the schedule during the simulation with the @ operator. The entries can take the following form:

    * *fire*(b), the joiner sends downstream one item from buffer b.

    * *receive*(b), the joiner receives one data item into the buffer b.

* *initPath*(b, i), the joiner calls `initPath(i)` and places the result in buffer b.

– Given the static, incoming channel weights of each joiner, we build a static *sending schedule* for each joiner. This schedule is represented as a sequence of nodes in G. The schedule gives the order in which to send items from the joiner's internal buffers. We call this sending schedule *StaticJoinSch*$[j]$ for joiner j. The schedule is computed prior to the algorithm described below and its calculation is trivial. Each schedule is represented as a circular list, with the last node pointing to the first. We will use *head*(*StaticJoinSch*$[j]$) to retrieve joiner j's current buffer that it should send from (the current node in the schedule), and *step*(*StaticJoinSch*$[j]$) to step the schedule to the next node.

– Given the static, outgoing channel weights of each roundrobin splitter, we build a static *sending schedule* for each roundrobin splitter. This schedule, represented as a list of nodes in G, gives the order in which to send items from the splitter to its downstream neighbors. The schedule is computed prior to the algorithm described below and its calculation is trivial. Since splitters are not mapped to the tiles of Raw, we only use the schedule when we are calculating the destination of a data item as it passes through the splitter on its way to a filter or joiner. Call this schedule *StaticSplitSch*$[s]$ for splitter s. It has the same properties and accessors as *StaticJoinSch*.

– *FireCount*$[n]$ stores the number of times node n has fired.

4.2.3 Pseudo-Code

The entry point of the communication scheduler is the `start()` function, most of the state of the simulation global, so `start()` does not take any arguments. We begin by placing the results of the `initPath()` calls of any feedbackloops in the appropriate internal buffer of the feedback joiner. This is accomplished by `setupInitPath()`. We then simulate the initialization schedule followed by the steady-state schedule. Each simulation starts with a call to the `simulate(`C`, `*prev*`)` function. This function determines which node should fire next in the simulation by calling `whoShouldFire(`n`, `C`)`. After finding a node to fire, `simulate(`C`, `*prev*`)` updates the state of the simulation by calling the `fire(`f`, `C`)` function for each item produced. After the state is updated, `simulate(`C`, `*prev*`)` creates the switch instructions for the item produced, accomplished by `generateSwitchCode(`f`, `D`)`. Finally, `simulate(`C`, `*prev*`)` recursively calls itself to determine if any node downstream of the previously fired node can fire.

The function `canFire?(`n`, `C`)` is called by `whoShouldFire(`n`, `C`)` to determine if node n can fire. The function `getDestination(`p`, `c`)` is called by `simulate(`C`, `*prev*`)` to determine the destinations of the of the current item. It is a recursive function that traverses G to find the destinations.

Note that C is an argument to some of the above methods. This argument holds the multiplicity for each node in G for the schedule we are simulating. For the initialization phase $C \leftarrow C_i$. For the steady-state phase $C \leftarrow C_s$, where $C_s[n]$ is equal to the multiplicity of n for both the initialization *and* the steady-state schedule.

`start()`. The entry point of the simulation. We first place the return values of `initPath()` in the appropriate joiner buffer. The initialization schedule is simulated next, followed by the simulation of the steady-state. The simulation returns the joiner buffer schedules and the switch code schedules for both the initialization and steady-state schedule. After the initialization schedule simulation we save the computed switch code and the joiner internal-buffer schedules.

Initialize all element of *Buffer*, *JoinBuf*, and *FireCount* to 0.
Initialize all sequences in *SwitchCode* and *JBufSch* to $\emptyset$.
`setupInitPath(`S`)`.
`simulate(`C_i`, `*root*`)`.
Let $JBufSch_{init} \leftarrow S.JBufSch$.
Let $SwitchCode_{init} \leftarrow S.SwitchCode$.
Set all sequences in *SwitchCode* and *JBufSch* to $\emptyset$.
`simulate(`C_s`, `*root*`)`.
return $\langle JBufSch_{init}, SwitchCode_{init}, JBufSch, SwitchCode \rangle$.

`setupInitPath()`. For feedbackloops, add the results of the `initPath()` function to the appropriate internal buffer of the joiner of the feedbackloop. Let J be the set of all joiners directly contained in a feedbackloop in G. For joiner node j, let $j.delay$ denote the delay of the feedbackloop (see Section 1.1.1).

for all $j \in J$ **do**
  for $i = 0$ to $j.delay - 1$ **do**
    Let b denote the internal joiner buffer representing the channel from the loop of the feedbackloop to the joiner.
    $JBufSch[j] \leftarrow JBufSch[j]@initPath(b, i)$.
    $JoinBuf[j][b] \leftarrow JoinBuf[j][b] + 1$.
  end for
end for

`simulate(`C`, `$prev$`)`. Given the multiplicities C and the previous node we fired *prev*, simulate the firing of each data item. We first find a filter to fire, one that is downstream of *prev*. Next, we update the simulation and generate the switch code. After each *item* produced, we see if any downstream filter can fire by recursively calling `simulate()`.

while *true* **do**
  $f \leftarrow$ `whoShouldFire(`$prev$`, `C`)`.
  if f = NIL **then**
    return.
  end if
  Let $I \leftarrow$ `fire(`f`, `C`)`.
  for $i \leftarrow 0$ to I **do**
    `generateSwitchCode(`f`, getDestination(`$downstream(f)$`))`.
    `simulate(`C`, `f`)`.
  end for
end while

`whoShouldFire(`n`, `C`)`. Given the multiplicities C and the previous node we fired n, this function determines which node in the stream graph should fire next. It follows a breadth-first traversal from the node n and calls `canFire?()` for each node, returning the latest node for which `canFire?()` returns *true*.

Let Q be a FIFO queue, initially empty
Let V be a set, initially empty.
Let $m \leftarrow$ NIL.
$enqueue(Q, n)$.
while $Q \neq \emptyset$ **do**
  Let $h \leftarrow dequeue(Q)$.
  $V \leftarrow V \cup \{h\}$.
  if `canFire?(`h`, `C`)` **then**
    $m \leftarrow h$.
  end if
  for all $c \in downstream(h)$ **do**
    if $c \notin V$ **then**
      enqueue(Q,c).
    end if
  end for
end while
return m.

`canFire?(`n`, `C`)`. Given the multiplicities C and node n, this function determines if node n can fire at this time in the simulation. For filters, it determines if the incoming buffer has enough items and if the filter has fired fewer times than the multiplicity given in C. For joiners, it determines if the internal joiner buffer at the head of the joiner send schedule has at least one item.

if n is a filter **then**
  Let $i \leftarrow 0$.
  if $FireCount[n] = 0$ **then**
    $i \leftarrow n.prePeek$.
  else
    $i \leftarrow n.peek$.
  end if
  if $FireCount[n] < C[n]$ **and** $Buffer[n] \geq i$ **then**
    return *true*.
  end if
else if n is a joiner **then**
  if $JoinBuf[n][head(StaticJoinSch[n])] > 0$ **then**
    return *true*.
  end if
end if
return *false*.

`fire(`f`, `C`)`. Given the multiplicities C and the node we are firing f, this function updates the state of the simulation for the firing of f, returning the number of items f produces. If f is a joiner, we add an entry to the joiner buffer schedule to fire the joiner, retrieving the data from the buffer at the head of the joiner send schedule. Then we step the joiner send schedule.

Let $\mathit{produced} = 0$.
if f is a filter **then**
  Let $\mathit{consumed} = 0$.
  if $\mathit{FireCount}[f] = 0$ **then**
    $\mathit{consumed} \leftarrow f.\mathit{prePop}$.
    $\mathit{produced} \leftarrow f.\mathit{prePush}$.
  else
    $\mathit{consumed} \leftarrow f.\mathit{Pop}$.
    $\mathit{produced} \leftarrow f.\mathit{Push}$.
  end if
  $\mathit{Buffer}[f] \leftarrow \mathit{Buffer}[f] - \mathit{consumed}$.
  $\mathit{FireCount}[f] \leftarrow \mathit{FireCount}[f] + 1$.
else if f is a joiner **then**
  $\mathit{produced} \leftarrow 1$.
  $\mathit{FireCount}[f] \leftarrow \mathit{FireCount}[f] + 1$.
  $\mathit{JBufSch}[f] \leftarrow \mathit{JBufSch}[f] @ \mathit{fire}(\mathit{head}(\mathit{StaticJoinSch}[f]))$.
  $\mathit{JoinBuf}[f][\mathit{head}(\mathit{StaticJoinSch}[f])] \leftarrow \mathit{JoinBuf}[f][\mathit{head}(\mathit{StaticJoinSch}[f])] - 1$.
  $\mathit{step}(\mathit{StaticJoinSch}[f])$.
end if
return $\mathit{produced}$.

`getDestination(`p`, `c`)`. Given the previous node we visited p and the current node we are visiting c, this function returns a set of the destinations. If c is a joiner, we receive the item into the internal buffer representing the channel connecting p to c. If c is a duplicate splitter, we must build a set of the destination nodes, as there is more than one destination for the item. If c is a roundrobin splitter then the item passes through the splitter to the downstream node given by the splitter's static sending schedule.

if c is a filter **then**
    $Buffer[c] \leftarrow Buffer[c] + 1$.
    return $\{c\}$.
else if c is a joiner **then**
    Let b denote the internal buffer of c representing the channel connecting p to c.
    $JoinBuf[c][b] \leftarrow JoinBuf[c][b] + 1$.
    $JBufSch[c] \leftarrow JBufSch[c]@receive(b)$.
    return $\{c\}$.
else if c is a splitter **then**
    if $c.type$ = duplicate **then**
        Let $Z = \emptyset$
        for all $n \in downstream(c)$ **do**
            $Z = Z \cup$ `getDestination(`c`, `n`)`.
        end for
        return Z.
    else if $c.type$ = roundrobin **then**
        Let $n = head(StaticSplitSch[c])$.
        $step(StaticSplitSch[c])$.
        return `getDestination(`c`, `n`)`.
    end if
end if

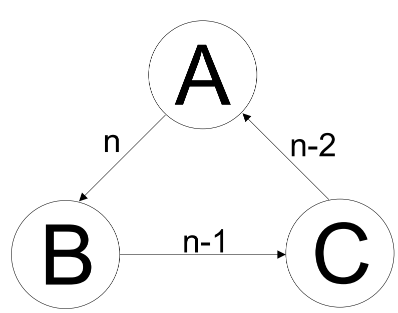
`generateSwitchCode(`f`, `D`)`. Given the node that is firing f and the set of destinations, D, this function generates switch code for the firing of one item. We place the switch instructions in *SwitchCode*. Remember, multiple reads from the same port within a single switch instruction reads the same value. It is not until the completion if the instruction that the item is dequeued from the port. We should also mention that the routes of a single item form a tree. This is a consequence of dimension-ordered routing.

Let $nextHop[t] \leftarrow \emptyset, \forall t \in T$.
Let $prevHop[t] \leftarrow$ NIL $, \forall t \in T$.
for all $d \in D$ **do**
  for all $\langle a, b \rangle \in getRoute(f, d)$ **do**
    $nextHop[a] \leftarrow nextHop[a] \cup \{b\}$.
    $prevHop[b] \leftarrow a$.
  end for
end for
for all $a \in L$ **do**
  Let $t \leftarrow tile(a)$.
  Let $ins =$ "`route `".
  if $a \in D$ **then**
    $ins = ins$ + "`$`" + $getDirection(t, prevHop[t])$ + "`i->$csti `".
    $SwitchCode[t] = SwitchCode[t]@ins$.
  end if
  if $nextHop[t] = \emptyset$ **then**
    continue.
  end if
  for all $b \in nextHop[t]$ **do**
    if $a = f$ **then**
      $ins = ins$ + "`$csto->$c`" + $getDirection(t, b)$ + "`o `".
    else
      $ins = ins$ + "`$`" + $getDirection(t, prevHop[t])$ + "`i->$c`" + $getDirection(t, b)$ + "`o `".
    end if
  end for
  $SwitchCode[t] = SwitchCode[t]@ins$
end for

4.3 Deadlock Avoidance

We will now give a brief, intuitive deadlock avoidance proof for the implementation described above. At this time we are in the process of formulating a rigorous proof. Assume that we are given a *valid* schedule. By *valid*, we mean a schedule that does not deadlock given infinite resources. With the above algorithm, we are mapping a valid schedule to the finite resources of a Raw machine, so we must show that this mapping does cannot lead to deadlock. The above algorithm operates at the granularity of a single data item. So for each data item, we route the item from the source to the destination, where it is consumed. Since the switch processor executes an ordered sequence of route instructions, we guarantee that if routes cross at a tile, the interleaving of the routes will be ordered by the switch [29]. Intuitively, for each data item we bring up a communication channel from the source to the destination and then rip it down when the item reaches the destination.

In the simulation, no data item is left on a communication channel so it cannot possibly deadlock. But on Raw this may not be the case, a switch may not be ready to receive an item when another switch is trying to send to it. The flow-control of the static network will block the sending tile until the receiving switch is ready to accept the item. In this case, we will not run into a circular dependency because such a dependency would lead to a contradiction. The existence of the dependency would mean a switch has passed over an routing instruction without actually performing the route. In the simulation we essentially order all the item produced during the simulation. This order is maintained by the sequence of instructions on each switch. A switch cannot route item n_1, without routing all previous items $n_2 < n_1$.



For example, look at the figure above. Let A, B, and C represent Raw tiles. We are in a deadlocking situation where A is trying to send to B, B is trying to send to C, and C is trying to send to A. Lets say A is trying to send item number n to B. For B to not have received item n, it must be busy sending an item of order less than n, say $n - 1$. In the same way, for C to not receive $n - 1$ from B it must be sending an item of order less than $n - 1$, say $n - 2$. But we have a contradiction. For A to be sending n it must have already sent $n - 2$, so C could not possibly be waiting to send this item. In this way a more rigorous proof can be constructed to show that all deadlocking situations are avoided.

4.4 Summary

In this chapter we started by describing the general function of the communication scheduling phase of the StreamIt compiler. Next, we gave the specific implementation for the Raw backend of the StreamIt compiler and attempted to explain any non-obvious parts of the implementation. Finally, we gave a short correctness proof for the implementation.

Chapter 5

Code Generation

The final phase in the flow of the StreamIt compiler is code generation. The code generation phase must use the results of each of the previous phases to generate the complete program text. The results of the partitioning and layout phases are used to generate the computation code that executes on a computation node of the target. The communication code of the program is generated from the schedules produced by the communication scheduler.

5.1 Code Generation for Raw

The code generation phase of the Raw backend generates code for both the tile processor and the switch processor. For the switch processor, we generate assembly code directly. For the tile processor, we generate C code that is compiled using Raw's GCC port.

5.1.1 Switch Code

To generate the instructions for the switch processor, we directly translate the switch schedules computed by the communication scheduler (see Section 4.1). The initialization switch schedule is followed by the steady state switch schedule, with the steady state schedule looping infinitely.

For larger applications, we sometimes overflowed the switch's instruction memory. In the simulator the switch instruction memory can store 8K instructions. To overcome this problem we compress the switch schedule generated by the communication scheduler. Repeating non-overlapping sequences of switch instruction are identified and placed in a loop. At this time, we only compress the three largest non-overlapping instructions sequences found in the switch schedule. This is because the switch processor has only four registers and no local memory. Thus, any constants must be explicitly loaded from the compute processor. We use three of the registers to store the repetition count for the sequences. We use the fourth register as a working register. It is used as the counter variable of the loop.

5.1.2 Tile Code

Filter Code

In the translation, each filter collects the data necessary to fire in an internal buffer. Before each filter is allowed to fire, it must have *peek* items in its internal buffer. The buffer is managed circularly with items received at the end of the buffer. The size of the buffer is calculated by the following expression. Assume that filter A is connected to Filter B and we are calculating the buffer size for B:

$$\begin{aligned} \mathit{BufferSize(B)} = & (init_A * maxpush_A) - \\ & ((init_B - 1) * minpop_B + minPeek_B) + \\ & maxpeek_B \end{aligned}$$

where:

- $init_X$ is the number of executions of filter X in the initialization schedule.
- $maxpush_A$ is the maximum of the push rate of `prework()` and the push rate of `work()` for filter A.

- $minpop_B$ is the minimum of the pop rate of `prework()` and the pop rate of `work()` for filter B.

- $minpeek_B$ is the minimum of the peek rate of `prework()` and the peek rate of `work()` for filter B.

- $maxpeek_B$ is the maximum of the peek rate of `prework()` and the peek rate of `work()` for filter X.

If A is a joiner or a splitter, $maxpush_A$ equals 1. The equation becomes a bit more complicated if A is a roundrobin splitter. We cannot use $(init_A * maxpush_A)$ to calculate the number of items A sends B in the initialization schedule. We must multiply this term by the ratio of the weight on the edge from A to B by the total of the outgoing splitter weights of A. For example. if x is a roundrobin splitter and it sends to m and n with weights 1 and 2, respectively. To calculate the items sent from x to m in the initialization schedule we multiply $init_A$ by 1/3.

Intuitively, in the above equation we are setting the buffer size to be equal to the peek rate of the filter. But we need to add any data that is produced by the upstream neighbor during the initialization schedule that is not consumed by the filter during the initialization schedule. The buffer size is actually set to the next greater power of 2 so we can replace expensive modulo operations by a bit mask operation.

In the code, `peek(index)` and `pop()` are translated into accesses of the buffer, with `pop()` adjusting the start of the buffer, and `peek(index)` accessing the $index^{th}$ element from the start of the buffer. `push(value)` is translated directly into a send from the tile processor to the switch processor. The switch processors are then responsible for routing the data item.

Filter execution starts with the `begin()` function. The code for the `begin()` function is given in Figure 5-1. It starts with a call to the function `raw_init()` which first loads the repetition counts for the switch code compression into switch registers (if necessary). Next in `raw_init()`, if the switch neighbors a file device (see Section 5.1.3), we send a dummy item to the device to start it up. Finally, it sets the switch PC to the start of the initialization schedule. The next two statements of `begin()`

```
void begin(void) {
  raw_init();
  __FLOAT_HEADER_WORD__ =
    construct_dyn_hdr(3, 1, 0, 0, 0, 3, 0);
  __INT_HEADER_WORD__ =
    construct_dyn_hdr(3, 1, 1, 0, 0, 3, 0);
  init(/* Args to init */);
  preWork();
  work();
}
```

Figure 5-1: The entry function for a filter.

initialize the dynamic message headers for print messages (see Section 5.1.3). `begin()` then calls the `init(...)` function, the `prework` function, and the `work` function. It never returns from the call to the work function.

In the `prework` function we first receive *prePeek* items, where *prePeek* is the peek rate of the `prework` function. We then translate the body of `prework()`. The translation of `work` is a bit more complicated because of the `prework` function and the initialization schedule. First, lets define the following:

- We assume that the `prework()` call always appears in the initialization schedule, even if it is not defined by the filter.
- Let $prePeek_X$, $prePop_X$, and $prePush_X$ be the *peek*, *pop*, and *push* rate of the `prework` function of filter X, respectively.
- Let $peek_X$, pop_X, and $push_X$ be the *peek*, *pop*, and *push* rate of the `work` function of filter X, respectively.
- Let $init_X$ be the number of executions of filter X in the initialization schedule.
- Let $bottomPeek_X$ equal the numbers of items that must be received after `prework()` has been called in order to execute the first call of `work()` in the initialization schedule. If filter X fires at least once in the initialization schedule, let:

$$bottomPeek_X \leftarrow \max(peek_X - (prePeek_X - prePop_X), 0)$$

  Otherwise, if X filter does not fire in the initialization schedule, let $bottomPeek_X \leftarrow 0$

- Let $remaining_B$ equal the number of items that filter B must receive into its buffer after its executes the `work()` calls in the initialization schedule. These are items that were produced by the upstream neighbor of B and must be received into B's buffer before the steady-state schedule is run. These items cannot just be left on in the network. If A is connected to filter B, Let:

$$\begin{aligned} remaining_B \leftarrow & (prePush_A + ((init_A - 1) * push_A)) - \\ & (prePeek_B + bottomPeek_B + \\ & (\max((init_B - 2), 0) * pop_B)) \end{aligned}$$

Algorithm 5 Algorithm to translate the `work` function for filter A.

```
if (init_A − 1) > 0 then
  if bottomPeek_A > 0 then
    Generate code to receive bottomPeek_A items into the buffer.
  end if
  Generate code to run work() (init_A − 1) number of times. Before each invocation
  of work() except the first, we receive pop_A items into the buffer.
end if
if remaining_A > 0 then
  Generate code to receive remaining_A items into the buffer.
end if
Generate code to infinitely execute work(). Before each invocation, receive pop_A
items into the buffer.
```

The translation of the `work` function is given in Algorithm 5. After the execution of the initialization schedule, we are guaranteed that at least *peek* - *pop* items appear in the buffer [22]. So, for each steady-state execution of the `work` function, the filter has to receive *pop* items before it can fire. All calls to `work()` in Algorithm 5 are inlined. See Figure 5-2 for an example of the `work` function translation.

The filter code does not interleave send instructions with receive instructions. The filter must receive all of the data necessary to fire before it can execute its work function. This is an overly conservative approach that prevents deadlock for certain situations, but limits parallelism. For example, this technique prevents feedbackloops from deadlocking by serializing the loop and the body. The loop and the body cannot

```
void work() {
  /* bottompeek = 1 */
  for (__EXEINDEX__ = 0; __EXEINDEX__ < 1; __EXEINDEX__++)
      //static_receive into buffer
  for (__EXEINDEX__ = 0; __EXEINDEX__ < 32; __EXEINDEX__++) {
    /* do not receive on the first invocation, taken care of
       by bottompeek */
    if (__EXEINDEX__ != 0) {
      /* receive pop items before each work function execution */
      for (__EXEINDEX__1__ = 0; __EXEINDEX__1__ < 1; __EXEINDEX__1__++)
        //static_receive into buffer
    }
    //work function
  }
  /*no remaining items to receive*/
  while (1) {
    /* receive pop items before each work function execution */
    for (__EXEINDEX__ = 0; __EXEINDEX__ < 1; __EXEINDEX__++)
      //static_receive into buffer

    //work function
  }
}
```

Figure 5-2: An example of the `work` function translation taken from the Filterbank application. In this example, the filter executes 33 times in the initialization schedule. $prePeek = prePop = 31$, $peek = pop = push = 1$. The upstream neighbor of this filter produces 63 items during the initialization schedule. So $bottompeek = 1 - (31 - 31) = 0$, and $remaining = 63 - (31 + 1 + (31 * 1)) = 0$.

execute in parallel. We are investigating methods for relaxing the serialization.

Joiner Code

As described in Section 4.1, the communication scheduler computes an internal buffer schedule for each collapsed joiner node. This schedule exactly describes the order in which to send and receive data items from within the joiner. The schedule is annotated with the destination buffer of the receive instruction and the source buffer of the send instruction. Also, the communication scheduler calculates the maximum size of each buffer. With this information the code generation phase can produce the code necessary to realize the internal buffer schedule on the tile processor.

Each internal buffer of the collapsed joiner is managed circularly. Send instructions send from the start of a buffer and receive instructions receive to the end of a buffer. The size of each buffer set to the next power of two greater than the buffer size calculated by the communication scheduler to allow for bit masking instead of expensive modulo operations.

A receive instruction is translated into the assembly instruction: `sw $csti, 0($r)`. Store the contents of `$csti` into the address `$r`, where `$csti` is the FIFO from the switch to the compute processor and `$r` is the address of the current index of the buffer we are receiving into. A send instruction is translated into: `lw $csto, 0($r)`. Load the word at address `$r` into the register `$csto`, where `$r` is the address of the current index of the buffer we are sending from and `$csto` is the FIFO from compute procssor to the switch. We can directly translate send and receive into one instruction memory operations because we know that they operate on a buffer that is stored in memory.

In Section 4.1 we state that the `initPath` function of a feedbackloop is placed in the joiner of the feedbackloop (see Figure 1-3). We place the return value of the `initPath()` call in the appropriate buffer, so the joiner can send it on to the body of the feedbackloop. The schedule node for an `initPath()` call includes the argument value and buffer to place the return value. So we simply assign the return value of the `initPath()` call to the end of the corresponding buffer.

As in the case of the switch code, we also found that for larger applications the joiner code overflowed the instruction memory of the compute processor. We therefore compress the schedule for each joiner node. Non-overlapping sequences of instructions are placed in a loop.

5.1.3 I/O

We currently support simple print statements and simple file manipulation filters. Print statements are translated into a dynamic network instruction that sends the value to be printed (and a header) to a print server sitting on the outside of the chip. This print server waits on the dynamic network and prints to the screen the data items it receives.

StreamIt includes two file access filters, the `FileReader` and the `FileWriter`. Semantically, they act just like normal filters where a `FileReader` pushes 1 item per execution and a `FileWriter` pops 1 item per execution. Each takes as an argument a string designating the file name and the type of data to be read or written.

To implement `FileReader`s and `FileWriter`s, we use the static network. We map a file device to the I/O ports on the right side of the chip for each `FileReader` or `FileWriter`. Thus these filters do not get mapped to tiles. We support as many open files as there are rows in the raw configuration. The file device requires one data word to be received before it starts executing. It is the responsibility of the neighboring switch to send the device a dummy item. The communication scheduler knows of the `FileReader`s and `FileWriter`s, and it correctly generates the routes for these filters.

5.2 Summary

In this chapter we covered the code generation phase of the Raw backend. We discussed the translation of the switch code and the tile code. We covered in detail the generation of tile code for joiner nodes and filter nodes. Finally, we discussed I/O primitives in StreamIt and their translation.

Chapter 6

Results

Our current implementation of StreamIt supports fully automatic compilation through the Raw backend. We have also implemented the optimizations that we have described: synchronization removal (Section 2.4), modulo expression elimination (vertical fusion, Section 2.2.2), buffer localization (vertical fusion), and buffer sharing (horizontal fusion, Section 2.2.3).

The results of this thesis were generated using btl, a cycle-accurate simulator that models arrays of Raw tiles identical to those in the .15 micron 16-tile Raw prototype ASIC chip. With a target clock rate of 250 MHz, the tile employs as compute processor an 8-stage, single issue, in-order MIPS-style pipeline that has a 32 KB data cache, 32 KB of instruction memory, and 64 KB of static router memory.

We evaluate the StreamIt compiler for the set of applications shown in Table 6.1. Table 6.2 gives some static measures of the benchmarks. For each benchmark, we show the number of lines of StreamIt code, the occurrence of each stream construct, and the number of nodes required to execute the expanded graph on Raw. Table 6.3 gives the performance results for the applications running on the 16 tile, 250 Mhz Raw simulator. For each benchmark, we show MFLOPS (which is not available for integer applications), processor utilization (the percentage of time that an *occupied tile* is not blocked on a send or receive, see Figure 6-1), and throughput. In Table 6.4 we compare the results of our implementation. For each application, we compare the throughput of StreamIt with a hand-written C program, running the latter on either

Benchmark	Description
FIR	64 tap FIR
Radar	Radar array front-end[26]
Radio	FM Radio with an equalizer
Sort	32 element Bitonic Sort
FFT	64 element FFT
Filterbank	8 channel Filterbank
GSM	GSM Decoder
Vocoder	28 channel Vocoder [37]
3GPP	3GPP Radio Access Protocol [6]

Table 6.1: Application Description.

a single tile of Raw or on a Pentium IV. For Radio, GSM, and Vocoder, the C source code was obtained from a third party; in other cases, we wrote a C implementation following a reference algorithm. We show the performance of the C code, which is not available for C programs that did not fit onto a single Raw tile (Radar, GSM, and Vocoder). Figures 6-2 and 6-3 illustrate the speedups obtained by StreamIt compared to the C implementations[1].

The results are encouraging. In many cases, the StreamIt compiler obtains good processor utilization–over 60% for four benchmarks and over 40% for two additional ones. For GSM, parallelism is limited by a feedbackloop that sequentializes much of the application. Vocoder is hindered by our work estimation phase, which has yet to accurately model the cost of library calls such as `sin` and `tan`; this impacts the partitioning algorithm and thus the load balancing. 3GPP also has difficulties with load balancing, in part because our current implementation fuses all the children of a stream construct at once.

StreamIt performs respectably compared to the C implementations, although there is room for improvement. The aim of StreamIt is to provide a higher level of abstraction than C without sacrificing performance. Our current implementation has taken a large step towards this goal. For instance, the synchronization removal optimization improves the throughput of 3GPP by a factor of 1.8 on 16 tiles (and by a factor of 2.5 on 64 tiles.) Also, our partitioner can be very effective–as illustrated in Figure 2-1, partitioning the Radar application improves performance by a factor

[1]FFT and Filterbank perform better on a Raw tile than on the Pentium 4. This could be because Raw's single-issue processor has a larger data cache and a shorter processor pipeline.

Benchmark	lines of code	# of constructs in the program				# of filters in the expanded graph
		filters	pipelines	splitjoins	feedbackloops	
FIR	125	5	1	0	0	132
Radar	549	8	3	6	0	52
Radio	525	14	6	4	0	26
Sort	419	4	5	6	0	242
FFT	200	3	3	2	0	24
Filterbank	650	9	3	1	1	51
GSM	2261	26	11	7	2	46
Vocoder	1964	55	8	12	1	101
3GPP	1087	16	10	18	0	48

Table 6.2: Application Characteristics.

of 11 even though it executes on less than one third of the tiles.

As mentioned in Section 1.2, the clock rate of the Raw simulator is 250MHz. With 16 tiles, the chip supports 16 floating point operation per cycle and 4.0 giga-floating point operations per second (GFLOPS). None of our benchmarks come close to approaching this number. In the following sections we will explain what needs to be done to achieve better performance on the Raw chip.

6.1 Communication and Synchronization

Looking at the execution traces given in the appendices, we notice that some of applications spend a significant number of cycles blocking. The compute processor of the tile becomes blocked when it tries to send to data to the switch but the FIFO from processor to the switch is full. Alternatively, the compute processor becomes blocked when it tries to receive data from the switch but the FIFO from the switch to the processor is empty. In this section we will explain some of the high-level causes of blocking.

The most prominent cause for blocking occurs when a tile assigned to a filter is an *intermediate hop* for a channel. By this we mean that the tile must route items through itself, the items being from a channel where it is not an endpoint. The tile must synchronize the routing of data items with the execution of its `work` function. For example, Figure I-3 gives the layout for the 3GPP application. It is far from a perfect layout. Many of the channels in this layout must route through tiles assigned

Benchmark	StreamIt on a 16 tile, 250 MHz Raw processor			
	Utilization	# of tiles used	MFLOPS	Throughput (per 10^5 cycles)
FIR	84%	14	815	1188.1
Radar	79%	16	1,231	0.52
Radio	73%	16	421	53.9
Sort	64%	16	N/A	2,664.4
FFT	42%	16	182	2,141.9
Filterbank	41%	16	644	256.4
GSM	23%	16	N/A	80.9
Vocoder	17%	15	118	8.74
3GPP	18%	16	44	119.6

Table 6.3: Raw Performance Results.

to filters. In fact, there does not exist a layout without intersecting routes for this partitioning. The resulting execution trace (Figure I-4) graphically shows the blocking caused by the crossed routes.

To lessen the effect of added synchronization, we could implement a more complex partitioning algorithm. As stated in Section 2.5 we use a simple greedy algorithm for partitioning and load balancing the stream graph. This greedy algorithm does not take into account the topology of the target architecture. For example, on Raw the partitioning algorithm should avoid numerous or large-way splitjoins in the final stream graph because Raw implements near-neighbor communication and each tile can only communicate directly with its neighbors.

Another solution to this problem is to interleave the routing task with the execution of the filter's work function. As mentioned in Section 4.1, we currently do not aggressively interleave routing instructions on the switch processor with computation of the filter's work function.

Unbalanced computation load between communicating filters is another source of blocking. The implementation of the work estimation algorithm is far from perfect and sometimes grossly mis-estimates the computation of a filter.

6.1.1 Limits Study on the Impact of Communication

To quantify the effect of the added synchronization imposed by the layout, we modified the filters of each application to execute in a decoupled manner. More specifically,

Benchmark	250 MHz Raw processor		C on a 2.2 GHz Intel Pentium IV
	StreamIt on 16 tiles	C on a single tile	
	Throughput (per 10^5 cycles)	Throughput (per 10^5 cycles)	Throughput (per 10^5 cycles)
FIR	1188.1	293.5	445.6
Radar	0.52	*app. too large*	0.041
Radio	53.9	8.85	14.1
Sort	2,664.4	225.6	239.4
FFT	2,141.9	468.9	448.5
Filterbank	256.4	8.9	7.0
GSM	80.9	*app. too large*	7.76
Vocoder	8.74	*app. too large*	3.35
3GPP	119.6	17.3	65.7

Table 6.4: Performance Comparison.

in the experiment each filter runs totally separate from its neighbors. The switch feeds the compute processor with dummy values when the compute process is receiving. Also, the switch disregards any data item sent to it by the compute processor. Obviously, this does not maintain correctness, but is a limit study on the cost of communication and synchronization imposed by mapping StreamIt's channel abstraction to Raw. But this simulates more than just instantaneous communication. Since the filters execute in a completely decoupled manner, filters with little work are not throttled by filters with a heavy work load. So load-balancing is not an issue. Finally, the joiner's task of data-reorganization is neglected. In the end, this experiment will give us an indication of how well the generated tile code is performing.

The results of decoupled execution are given in Table 6.5 and Figure 6-4. FFT and Radar show the largest and smallest performance gain, respectively. A detailed examination of each application is given in the next sections. Across the benchmarks we see only a 22% increase in MFLOPS. We interpret this as meaning that performance is not being limited by the communication implementation. The generated tile code is preventing us from achieving higher utilization of Raw's floating point units. Reasons for the low MFLOPS rate of the tile code include:

- Raw's port of gcc is not producing good code. We run gcc with optimization level 3 but looking at the assembly code generated we see much room for improvement.

- As mentioned in Section 5.1.2, the implementation of a filter uses a circular

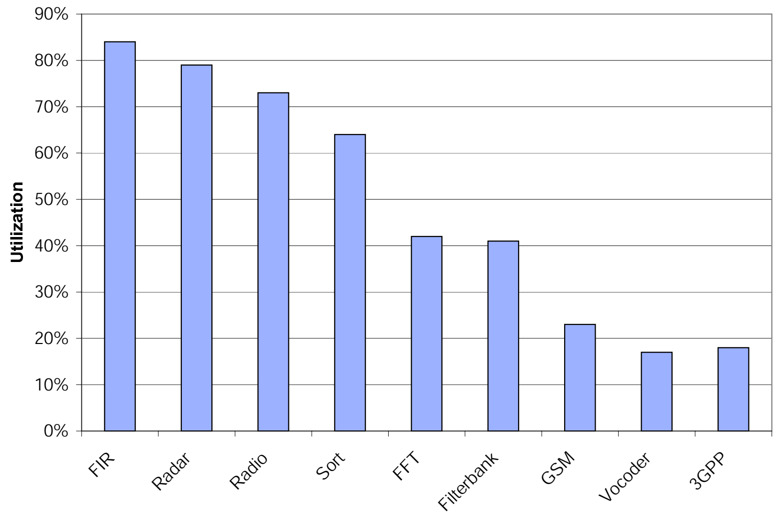


Figure 6-1: Processor utilization of StreamIt code running on 16 tiles.

buffer. The access and management time of this buffer could be killing us on certain apps.

- Currently, we have not fully optimized the partitioning implementation. It will be interesting to see if we can reduce the overhead of fusion and fission.

- The in-order, 8-stage Raw pipeline is not a good match for the code we are generating. The single-ported data cache is limiting the performance of our memory-intensive generated code.

6.2 A Closer Look

6.2.1 Radar Application

The application with the highest FLOPS rate is the Radar application. In this section, we will explain why we do not obtain better FPU utilization for the Radar application. For the Radar application we assign filters to 14 of the Raw tiles (see B-3). Tile 5 is unassigned and there is joiner tile that does not contain floating point operations. Thus, at most we can achieve 3500 MFLOPS. Looking at the execution trace for the Radar application (Figure B-4), we can see that the communication overhead is minimal. There is very little blocking noticeable for the tiles assigned to

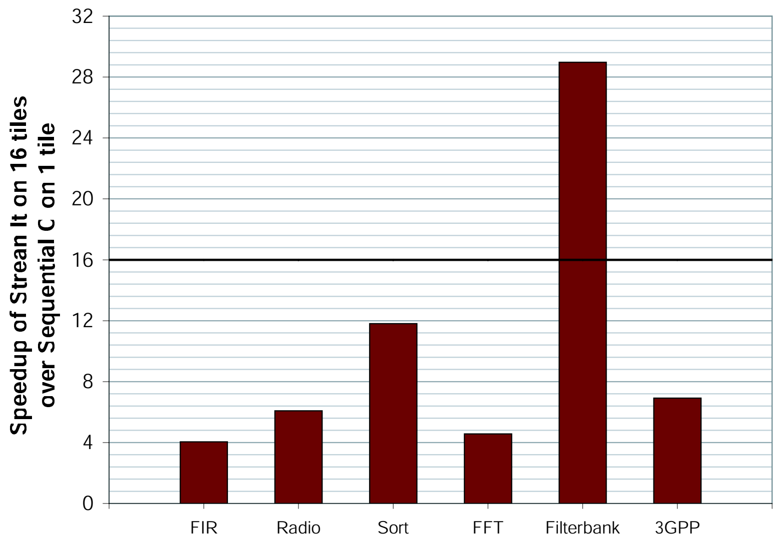


Figure 6-2: StreamIt throughput on a 16-tile Raw machine, normalized to throughput of hand-written C running on a single Raw tile.

filters. Looking at Table 6.5, we can see that decoupled execution of the filters only increased the MFLOPS count by 10. As we suspected, communication and added synchronization does not hurt us in the Radar application. This is due to the fact that the fused filters in the top splitjoin (see Figure B-2) push only two items and the filters of the bottom splitjoin neighbor the joiner. Other applications, such as 3GPP, do have a high synchronization overhead.

Our next inclination was that the joiner could be hurting us by throttling its upstream filters and not feeding the downstream filters fast enough. But the decouple execution experiment neglects the cost of the joiner and there is only a 10 MFLOPS increase from the experiment. The joiner is not hurting the performance significantly; it has plenty of time to perform the data-reorganization while the filter tiles are computing.

Looking at the C code generated by the StreamIt compiler for the Radar application, we see that the code implementing the fusion transformations occupy about 40% of the source text for each tile. It is impossible to remove this fusion code and maintain correctness, so we cannot calculate an exact overhead for the partitioning code. But it is safe to say that a large percentage of the cycles time for the Radar application is spend reordering and copying data inside the fusion code. This is nec-

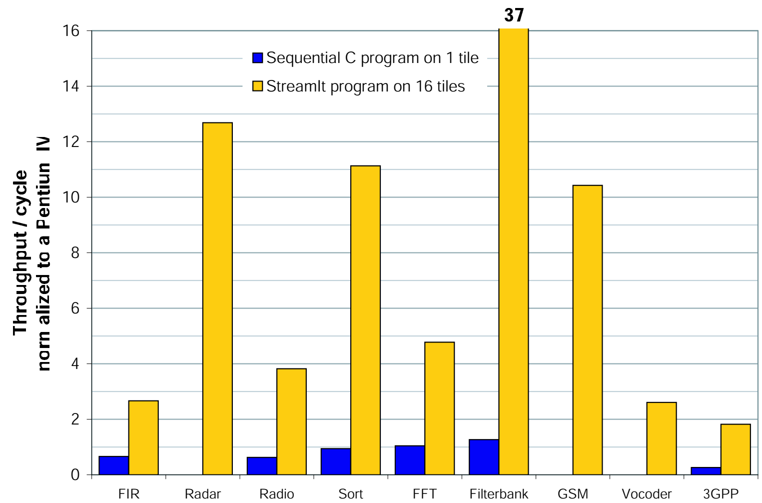


Figure 6-3: Throughput of StreamIt code running on 16 tiles and C code running on a single tile, normalized to throughput of C code on a Pentium IV.

essary for correctness, but is not reflected in the MFLOPS number. The current partitioning code is our first attempt at a correct implementation. We need to spend considerable time optimizing the generated fusion and fission code for Raw's memory hierarchy to reduce the overhead.

Examining the computational cores for the fused filters, we noticed that all of the floating point operations required at least one of the operands to be fetched from the data cache. On Raw, data cache access is pipelined but it has a one cycle occupancy. This occupancy is costly on a single issue machine. These operands correspond to a large array of weights required for a beamforming operation and an FIR operation. If the size of these weight arrays could be reduced to fit within the register file, we would see a large increase in MFLOPS. In the future, we hope that the vertical fission transformation (see Section 2.3.1) will be able to help this problem. With vertical fission, we could split a filter into a pipeline of smaller filters, where the state of each resulting filter could fit in the register file.

As discussed in Section 5.1, before a filter can fire, we receive *pop* items into a circular buffer. This circular buffer is stored in the data cache, so for each item we must read it from the network into the cache. Also, each `pop` instruction is translated into a read from this buffer, causing another cache access. So, for any floating point

Benchmark	**MFLOPS for Normal Communication**	**MFLOPS for Decoupled Execution**	**% increase**
FIR	815	1025	26%
Radar	1,231	1241	1%
Radio	421	519	23%
FFT	182	359	97%
Filterbank	644	787	22%
Vocoder	118	214	81%
3GPP	44	61.5	40%

Table 6.5: Decoupled Execution.

operation in the original code with a `pop` expression as one of its operands, we must read that operand from the data cache. So each `pop` expression that appeared as operand to a floating point operation in the original StreamIt code is translated into three instructions. One instruction for the floating point operation and two instructions for the data cache access associated with the circular buffer.

We should also note that we perform aggressive loop unrolling to amortize the cost of the loop overhead. Tight loops are common this application and across our benchmark suite. Without this simple optimization our results would have been far worse.

6.2.2 FFT

In Figure E-4 we have the execution trace for the FFT application. We can see that there is a significant amount of blocking in the trace even though the application is partitioned down to a single pipeline (Figure E-2) with a perfect raw layout (Figure E-3). If we refer to Table 6.5 we can see that decoupled execution increase MFLOPS by 97%.

The main problem is that there is a significant amount of blocking due to the mismatching computation rates of neighboring filters. Some filters are blocked trying to send or receive while their neighbor is executing code.

Another problem is the joiner node mapped to tile 3. Since the filters of FFT perform a relatively small amount of work, the joiner tile throttles the upstream filters and does not feed the downstream filters fast enough. Interestingly, the joiner node is not needed for this application. It sends data in the same order it receives

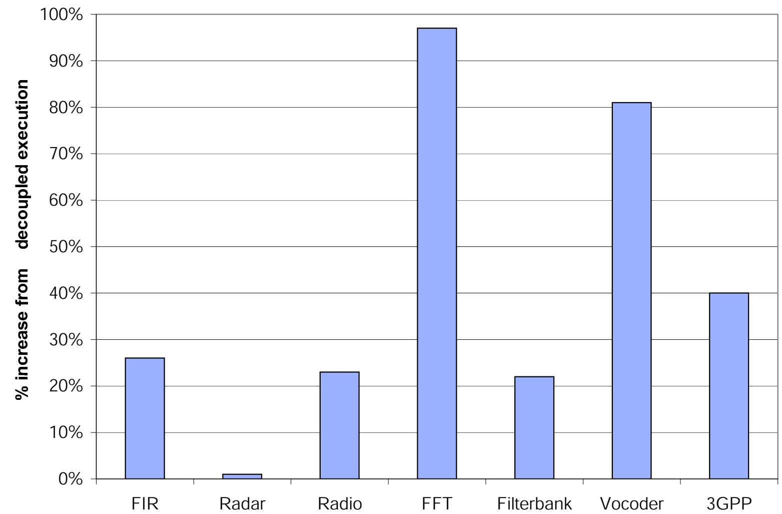


Figure 6-4: Percentage increase in MFLOPS for decoupled execution over normal execution.

it. No data-reorganization is needed. Removing this joiner increases the performance of FFT by 33%. We should note that this joiner optimization was rather specific and did not appear in any other application (and thus was not automated). We are investigating ways to further optimize joiners.

6.3 Summary

In this chapter we presented the performance results for the Raw backend of the StreamIt compiler. The StreamIt optimization framework is far from complete, and the results presented here represent a first step rather than an upper bound on our performance. We are actively implementing aggressive inter-node optimizations and more sophisticated partitioning strategies that will allow us to better utilize the abundant resources of Raw.

Chapter 7

Related Work

In this chapter we will present related work. The related work includes other stream languages and how they compare to StreamIt. We will describe some other communication-exposed architectures. Also, we will describe other stream architectures and discuss the primary programming language used for each architecture.

The Transputer architecture [5] is an array of processors, where neighboring processors are connected with unbuffered point-to-point channels. The Transputer does not include a separate communication switch, and the processor must get involved to route messages. The programming language used for the Transputer is Occam [20]: a streaming language similar to CSP [19]. However, unlike StreamIt filters, Occam concurrent processes are not statically load-balanced, scheduled and bound to a processor. Occam processes are run off a very efficient runtime scheduler implemented in microcode [31].

DSPL is a language with simple filters interconnected in a flat acyclic graph using unbuffered channels [32]. Unlike the Occam compiler for the Transputer, the DSPL compiler automatically maps the graph into the available resources of the Transputer. The DSPL language does not expose a cyclic schedule, thus the compiler models the possible executions of each filter to determine the possible cost of execution and the volume of communication. It uses a search technique to map multiple filters onto a single processor for load balancing and communication reduction.

The Imagine architecture is specifically designed for the streaming application domain [35]. It operates on streams by applying a computation kernel to multiple data items off the stream register file. The compute kernels are written in Kernel-C while the applications stitching the kernels are written in Stream-C. Unlike StreamIt, with Imagine the user has to manually extract the computation kernels that fit the machine resources in order to get good steady state performance for the execution of the kernel [21]. On the other hand, StreamIt uses fission and fusion transformations to create load-balanced computation units and filters are replicated to create more data parallelism when needed. Furthermore, the StreamIt compiler is able to use global knowledge of the program for layout and transformations at compile-time while Stream-C interprets each basic block at runtime and performs local optimizations such as stream register allocation in order to map the current set of stream computations onto Imagine.

At this time we would like to point out the difference between space-division and time-division stream multiplexing. In space-division multiplexing the different filters of the stream program execute on separate computational units, each unit executing one filter. The Raw backend of the StreamIt compiler uses space-division multiplexing as each filter is assigned to a different Raw tile with each running concurrently. The fusion transformations allow us to fit a stream graph of arbitrary size on tiles of Raw. The Imagine architecture uses time-division multiplexing, where over time different filters of the stream program run on a single computational unit. Imagine swaps computation kernels of the stream program in and out of the processor as the program executes. A detailed comparison of time and space division stream multiplexing is beyond the scope of this paper. We are currently researching ways to combine both time and space division multiplexing in the StreamIt compiler.

The iWarp system [10] is a scalable multiprocessor with configurable communication between nodes. In iWarp, one can set up a few FIFO channels for communicating between non-neighboring nodes. However, reconfiguring the communication channels is more coarse-grained and has a higher cost than on Raw, where the network routing patterns can be reconfigured on a cycle-by-cycle basis [40]. ASSIGN [33] is a tool for building large-scale applications on multiprocessors, especially iWarp. ASSIGN starts with a coarse-grained flow graph that is written as fragments of C code. Like StreamIt, it performs partitioning, placement, and routing of the nodes in the graph. However, ASSIGN is implemented as a runtime system instead of a full language and compiler such as StreamIt. Consequently, it has fewer opportunities for global transformations such as fission and reordering.

SCORE (Stream Computations Organized for Reconfigurable Execution) is a stream-oriented computational model for virtualizing the resources of a reconfigurable architecture [11]. Like StreamIt, SCORE aims to improve portability across reconfigurable machines, but it takes a dynamic approach of time-multiplexing computations (divided into "compute pages") from within the operating system, rather than statically scheduling a program within the compiler.

Ptolemy [28] is a simulation environment for heterogeneous embedded systems, including the domain of Synchronous Dataflow (SDF) that is similar to the static-rate stream graphs of StreamIt. While there are many well-established scheduling techniques for SDF [8], the round-robin nodes in our stream graph require the more general model of Cyclo-Static Dataflow (CSDF) [9] for which there are fewer results. Even CSDF does not have a notion of an initialization phase, filters that peek, or a dynamic messaging system as supported in StreamIt. In all, the StreamIt compiler differs from Ptolemy in its focus on optimized code generation for the nodes in the graph, rather than high-level modeling and design.

Proebsting and Watterson [34] present a filter fusion algorithm that interleaves the control flow graphs of adjacent nodes. However, they assume that nodes communicate via synchronous `get` and `put` operations; StreamIt's asynchronous peek operations and implicit buffer management fall outside the scope of their model.

A large number of programming languages have included a concept of a stream; see [38] for a survey. Synchronous languages such as LUSTRE [17], Esterel [7], and Signal [14] also target the embedded domain, but they are more control-oriented than StreamIt and are not aggressively optimized for performance. Sisal (Stream and Iteration in a Single Assignment Language) is a high-performance, implicitly parallel functional language [13]. The Distributed Optimizing Sisal Compiler [13] considers compiling Sisal to distributed memory machines, although it is implemented as a coarse-grained master/slave runtime system instead of a fine-grained static schedule.

Chapter 8

Conclusion

In this thesis, we describe the StreamIt compiler and a backend for the Raw architecture. Unlike other streaming languages, StreamIt enforces a structure on the stream graph that allows a systematic approach to optimization and parallelization. The structure enables us to define multiple transformations and to compose them in a hierarchical manner.

We introduce a collection of optimizations–vertical and horizontal filter fusion, vertical and horizontal filter fission, and filter reordering–that can be used to restructure stream graphs. We show that by applying these transformations, the compiler can automatically convert a high-level stream program, written to reflect the composition of the application, into a load-balanced executable for Raw.

The stream graph of a StreamIt program exposes the data communication pattern to the compiler, and the lack of global synchronization frees the compiler to reorganize the program for efficient execution on the underlying architecture. The StreamIt compiler demonstrates the power of this flexibility by partitioning large programs for execution on Raw. However, many of the techniques we describe are not limited to Raw; in fact, we believe that the explicit parallelism and communication in StreamIt is essential for obtaining high performance on other communication-exposed architectures. In this sense, we consider the techniques described in this thesis to be a first step towards establishing a portable programming model for communication-exposed machines.

Appendix A

FIR Application

A.1 Description

This benchmark models a Finite Impulse Response system on an input of 128 points.

A.2 Code

```
    import streamit.*;
import streamit.io.*;

public class FIRfine extends StreamIt
{
    public static void main (String [] args)
    {
        new FIRfine ().run (args);
    }

    public void init ()
    {
        add (new FloatSource (10000));
        add (new FIR (128));
        add (new FileWriter("output.dat", Float.TYPE));
        //add (new FloatPrinter (10000));
    }
}

class FIR extends Pipeline
{
    FIR (int N)
    {

    }

    public void init (final int N)
    {
        int i;
        add(new Filter() {
                public void init() {
                    this.input =
                        new Channel(Float.TYPE, 1, 1);
                    this.output =
                        new Channel(Float.TYPE, 2);
                }
                public void work() {
                    this.output.pushFloat(0);
                    this.output.pushFloat
                        (this.input.popFloat());
                }
            });
        for(i=0; i<N; i++)
            add (new SingleMultiply(i));
        add(new Filter() {
                public void init() {
                    this.input =
                        new Channel(Float.TYPE, 2, 2);
                    this.output =
                        new Channel(Float.TYPE, 1);
                }
                public void work() {
                    this.output.pushFloat
                        (this.input.popFloat());
                    this.input.popFloat();
                }
            });
    }
}

class SingleMultiply extends Filter
{
    SingleMultiply(int i)
    {

    }

    float W;
    float last;
    public void init(final int i) {
        last = 0;
        W = 2*i*i/((float)i+1);
        this.input = new Channel(Float.TYPE, 12);
        this.output = new Channel(Float.TYPE, 12);
    }

    public void work() {
        for (int i=0; i<6;i++) {
            float s;
            s = this.input.popFloat();
            this.output.pushFloat(s+last*W);
            this.output.pushFloat(last);
            last = this.input.popFloat();
        }
    }
}

class FloatSource extends Filter
{
    FloatSource (float maxNum)
    {

    }

    float num;
    float maxNum;

    public void init (float maxNum2)
    {
        output = new Channel (Float.TYPE, 1);
        this.maxNum = maxNum2;
```

```
        this.num = 0;
    }

    public void work ()
    {
        output.pushFloat (num);
        num++;
        if (num == maxNum) num = 0;
    }
}

class FloatPrinter extends Filter
{
    FloatPrinter (int x)
    {
    }

    int x;
    public void init (int x2)
    {
        input = new Channel (Float.TYPE, 1);
        this.x = x2;

    }
    public void work ()
    {
        System.out.println(input.popFloat ());
    }
}
```

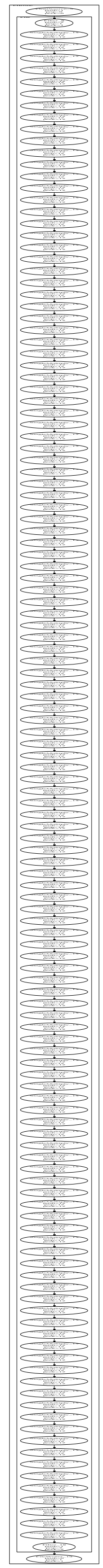


Figure A-1: FIR before partitioning.

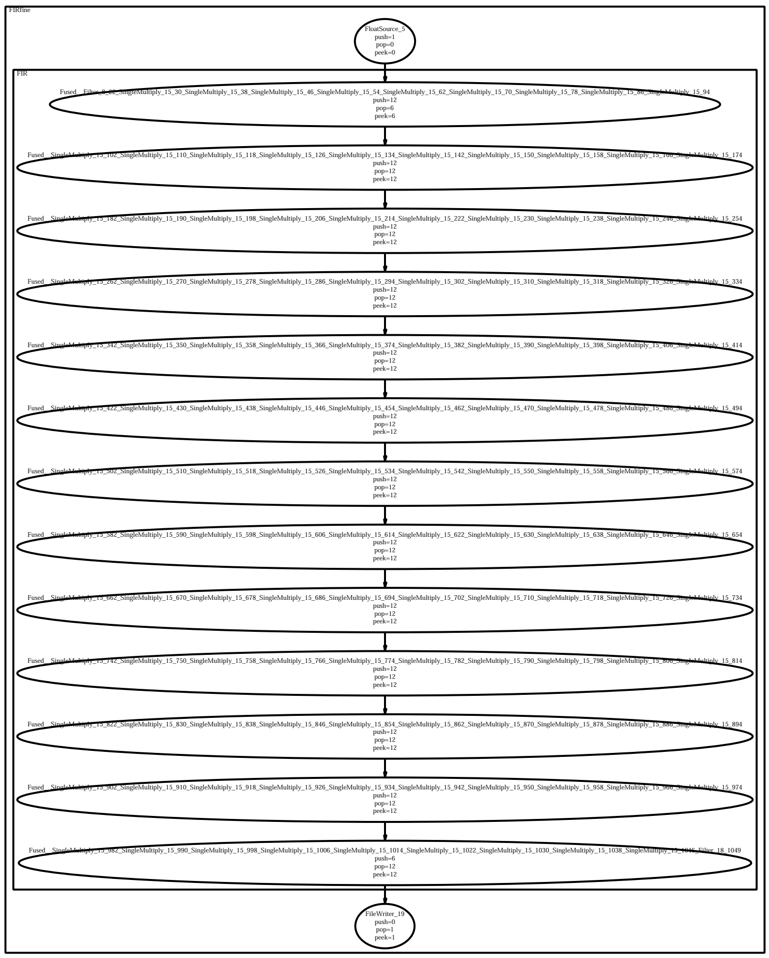


Figure A-2: FIR after partitioning.

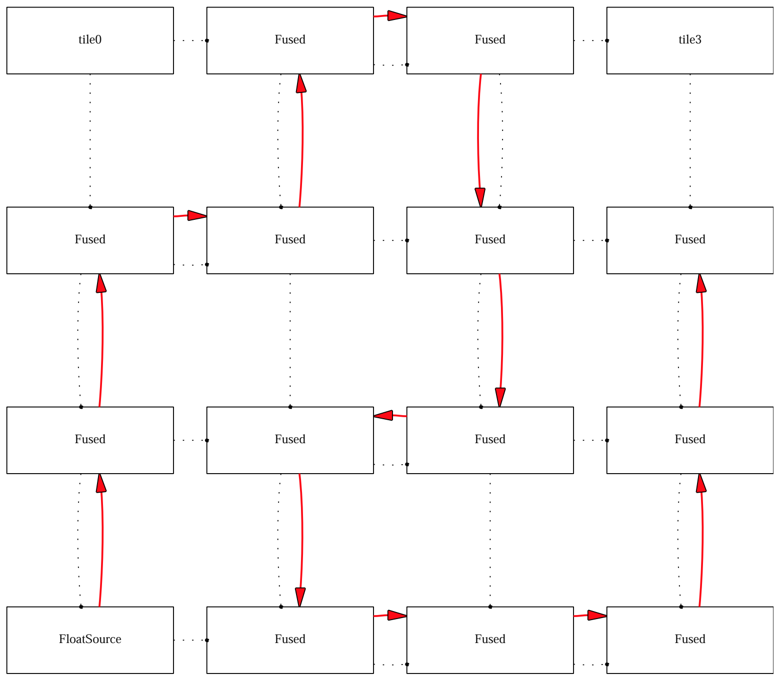


Figure A-3: FIR layout.

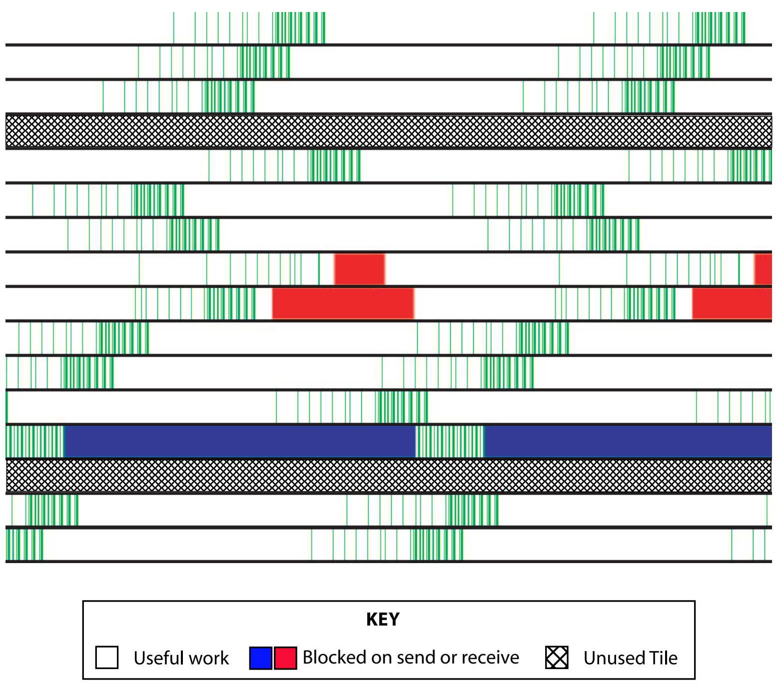


Figure A-4: FIR execution trace.

Appendix B

Radar Application

B.1 Description

The application consists of four stages: pulse compression, Doppler filtering, beamforming, and detection. The first two stages perform preliminary processing on the data similar to the low-pass filtering stage of the multi-stage application. The beamforming stage transforms the filtered data to allow detection of signals coming from a particular set of directions of interest, just as in the multi-stage application. The detection stage determines whether targets are actually present in the beamformed data and performs simple grouping and parameter estimation operations on those targets. The previous description was taken from [26].

B.2 Code

```
import streamit.*;

public class BeamFormer extends StreamIt
{
  static public void main(String[] t)
  {
    BeamFormer test = new BeamFormer();
    test.run(t);
  }

  public void init()
  {
    // how many streams per hierarchical
    // splitjoin in the detect phase
    final int GENERATE_BLOCKING     = 1;
    // how many streams per hierarchical
    // splitjoin in the detect phase
    final int DETECT_BLOCKING       = 2;

    final int numChannels           = 12;//48;
    final int numSamples            = 64;//4096;
    final int numBeams              = 4;//16;
    final int numCoarseFilterTaps   = 64;//
    final int numFineFilterTaps     = 64;//
    final int coarseDecimationRatio = 1;
    final int fineDecimationRatio   = 2;
    final int numSegments           = 1;
    final int numPostDec1
      = numSamples/coarseDecimationRatio;
    final int numPostDec2
      = numPostDec1/fineDecimationRatio;
    final int mfSize
      = numSegments*numPostDec2;
    final int pulseSize             = numPostDec2/2;
    final int predecPulseSize       = pulseSize*
      coarseDecimationRatio*fineDecimationRatio;
    final int targetBeam            = numBeams/4;
    final int targetSample          = numSamples/4;
    // targetSamplePostDec used to
    //  have a 1 added to it, but that
    // seemed to do the wrong thing --bft
    final int targetSamplePostDec   =
      targetSample/coarseDecimationRatio /
      fineDecimationRatio;
    final float dOverLambda         = 0.5f;
    final float cfarThreshold       = 0.95f *
      dOverLambda*numChannels *
      (0.5f*pulseSize);

    add(new SplitJoin() {
        public void init() {
          int i;
          setSplitter(NULL());
          for(i=0; i<numChannels;
              i+=GENERATE_BLOCKING) {
            add(new SplitJoin(i) {
                SplitJoin(int i)
                {super(i); }
                public void init(int i) {
                  setSplitter(NULL());
                  for (int k=0; k<GENERATE_BLOCKING;
                       k++) {
                    add(new Pipeline(i+k) {
                        Pipeline(int i) {super(i);}
                        public void init(int i) {
                          add(new InputGenerate
                              (i, numSamples,
                               targetBeam,targetSample,
                               cfarThreshold));
                          add(new BeamFirFilter
                              (numCoarseFilterTaps,
                               numSamples,
                               coarseDecimationRatio));
                          add(new BeamFirFilter
```

```
                              (numFineFilterTaps,
                               numPostDec1,
                               fineDecimationRatio));
                        }
                      });
                  }
                  setJoiner(ROUND_ROBIN(2));
                }
              });
          }
          setJoiner(ROUND_ROBIN(2*GENERATE_BLOCKING));
        }
      });

    add(new SplitJoin() {
        public void init() {
          int i;
          setSplitter(DUPLICATE());
          for(i=0; i<numBeams; i+=DETECT_BLOCKING) {
            add(new SplitJoin(i) {
                public SplitJoin(int i) { super(i); }
                public void init(int i) {
                  setSplitter(DUPLICATE());
                  for (int k=0; k<DETECT_BLOCKING; k++){
                    add (new Pipeline(i+k) {
                        public Pipeline(int i)
                          { super(i); }
                        public void init(int i) {
                          add(new Beamform
                              (i, numChannels));
// Need to replace this fir with
//fft -> elWiseMult -> ifft
                          add(new BeamFirFilter
                              (mfSize,
                               numPostDec2,
                               1));
                          add(new Magnitude());
// with a more sophisticated detector, we need
// someplace to store the data until we can find
                          // the targets...
                          add(new Detector
                              (i,
                               numPostDec2,
                               targetBeam,
                               targetSamplePostDec,
                               cfarThreshold));
                        }
                      });
                  }
                  setJoiner(NULL());
                }
              });
            setJoiner(NULL());
          }
        }
      });
  }
}

class InputGenerate extends Filter
{ // class InputGenerate

  int curSample;
  int numberOfSamples;
  boolean holdsTarget;
  int targetSample;
  int myChannel;
  float thresh;
  //    int i2;

  public InputGenerate(int i, int n, int t1,
                       int t2, float c) {
    super(i, n, t1, t2, c);
  }

  public void init(int i,
                   int nSamples,
                   int tarBeam,
                   int tarSample,
                   float cfarThresh)
  { // InputGenerate::init()

    curSample = 0;
    numberOfSamples = nSamples;
    holdsTarget = (tarBeam == i);
    targetSample = tarSample;
    myChannel = i;

    thresh = cfarThresh;
    //      i2 = 0;
    output = new Channel(Float.TYPE, 2);
  }

  public void work()
  { // InputGenerate::work()
    if( holdsTarget && (curSample == targetSample) )
      {
        // real
        output.pushFloat((float)Math.sqrt(thresh));
        // imag
        output.pushFloat(0);
      }
    else
      {
        // real
        output.pushFloat(0);
        // imag
        output.pushFloat(0);
      }

    //      System.out.println(i2++);

    curSample++;

    if( curSample >= numberOfSamples )
      {
        curSample = 0;
      }
  }
}

/**
 * This filter just outputs a stream of zeros.
 */
class ZeroSource extends Filter {

  public ZeroSource() {
    super();
  }

  public void init() {
    output = new Channel(Float.TYPE, 1);
  }

  public void work() {
    output.pushFloat(0);
  }
}

class DummySink extends Filter {
  public DummySink() {
    super();
  }

  public void init() {
    input = new Channel(Float.TYPE, 1);
  }

  public void work() {
    input.popFloat();
  }
}

class BeamFirFilter extends Filter
{ // class FirFilter...

  float[] real_weight;
  float[] imag_weight;
  int numTaps;
  int inputLength;
  int decimationRatio;
  float[] realBuffer;
  float[] imagBuffer;
  // number of items we've seen in
  // relation to inputLength
  int count;
  // our current writing position into the buffers
  int pos;

  public BeamFirFilter(int nt, int inLength,
                       int decRatio) {
    super(nt, inLength, decRatio);
  }

  public void init(int nt, int inLength, int decRatio)
  { // BeamFirFilter::init()
    int i;
    numTaps = nt;
    inputLength = inLength;
    decimationRatio = decRatio;

    input = new Channel(Float.TYPE, 2*decRatio);
    output = new Channel(Float.TYPE, 2);
    real_weight = new float[numTaps];
```

```
    imag_weight = new float[numTaps];
    realBuffer = new float[numTaps];
    imagBuffer = new float[numTaps];
    pos = 0;

    real_weight[0] = 1.0f;
    imag_weight[0] = 0.0f;
    for(i = 1; i < numTaps; i ++) {
      real_weight[i] = 0;
      imag_weight[i] = 0;
      realBuffer[i] = 0;
      imagBuffer[i] = 0;
    }
  }

  public void work()
  { // BeamFirFilter::work()
    float real_curr = 0;
    float imag_curr = 0;
    int i;
    int modPos;

    // pop a new item into the buffer
    realBuffer[pos] = input.popFloat();

    imagBuffer[pos] = input.popFloat();

    // calculate sum
    modPos = pos;
    for (i = 0; i < numTaps; i++) {
      real_curr +=
        realBuffer[modPos]*real_weight[i] +
        imagBuffer[modPos] * imag_weight[i];
      imag_curr +=
        imagBuffer[modPos] * real_weight[i] +
        realBuffer[modPos] * imag_weight[i];
      // increment position in this round of summing
      modPos++;
      if (modPos==numTaps) { modPos = 0; }
    }

    // increment sum
    pos = (pos+1)%numTaps;

    // push output
    output.pushFloat(real_curr);
    output.pushFloat(imag_curr);

    // decimate
    for (i = 2; i < 2*decimationRatio; i++) {
      input.popFloat();
    }

    // increment count
    count+=decimationRatio;

    // when we reach inLength, reset
    if (count==inputLength) {
      count = 0;
      pos = 0;
      for (i=0; i<numTaps; i++) {
        realBuffer[i] = 0;
        imagBuffer[i] = 0;
      }
    } else if (count>inputLength) {

    }
  }
}

class Beamform extends Filter
{ // class Beamform...

  float[] real_weight;
  float[] imag_weight;
  int numChannels;
  int myBeamId;

  public Beamform(int myBeam, int nc) {
    super(myBeam, nc);
  }

  public void init(int myBeam, int nc)
  { // BeamCalc::init()
    int i;
    numChannels = nc;
    myBeamId = myBeam;

    input = new Channel(Float.TYPE, 2*nc);
    output = new Channel(Float.TYPE, 2);
    real_weight = new float[numChannels];
    imag_weight = new float[numChannels];
```

```
    // For now, use identity weights.
    for(i = 0; i < numChannels; i++)
      {
        real_weight[i] = 0;
        imag_weight[i] = 0;
        if( i == myBeamId )
          {
            real_weight[i] = 1;
            imag_weight[i] = 0;
          }
      }
  }

  public void work()
  { // BeamCalc::work()
    float real_curr = 0;
    float imag_curr = 0;
    int i;
    for(i=0; i<numChannels; i++) {
      float real_pop = input.popFloat();
      float imag_pop = input.popFloat();
      // Need to check this boundary cond
      real_curr +=
        real_weight[i] * real_pop -
        imag_weight[i] * imag_pop;
      imag_curr +=
        real_weight[i] * imag_pop +
        imag_weight[i] * real_pop;
    }
    output.pushFloat(real_curr);
    output.pushFloat(imag_curr);
  }
}

class Magnitude extends Filter
{ // class Magnitude...

  public void init()
  {
    input = new Channel(Float.TYPE, 2);
    output = new Channel(Float.TYPE, 1);
  }

  public void work()
  {
    float f1 = input.popFloat();
    float f2 = input.popFloat();
    output.pushFloat(mag(f1, f2));
  }

  /**
   * Return magnitude of (<real>, <imag>)
   */
  private float mag(float real, float imag) {
    return (float)Math.sqrt(real*real + imag*imag);
  }
}

class Detector extends Filter
{ // class Detector...

  int curSample;
  int myBeam;
  int numSamples;
  float thresh;
  int targetSample;
  boolean holdsTarget;

  public Detector(int i,
                  int nSamples,
                  int targetBeam,
                  int tarSample,
                  float cfarThreshold) {
    super(i, nSamples, targetBeam, tarSample,
          cfarThreshold);
  }

  public void init(int i,
                   int nSamples,
                   int targetBeam,
                   int tarSample,
                   float cfarThreshold)
  {
    curSample = 0;
    myBeam = i;
    numSamples = nSamples;
    holdsTarget = (myBeam == targetBeam);
    targetSample = tarSample;

    thresh = 0.1f;
    input = new Channel(Float.TYPE, 1);
```

```
}

public void work()
{
  float val = input.popFloat();

  if(holdsTarget && targetSample == curSample)
    {
      if( !(val >= thresh) ) {
        System.out.println(0);
      } else {
        System.out.println(1);
      }
    }
    else
      {
        if( val >= thresh ) {
          System.out.println(0);
        }
      }

    curSample++;

    if( curSample >= numSamples )
      curSample = 0;
  }
}
```

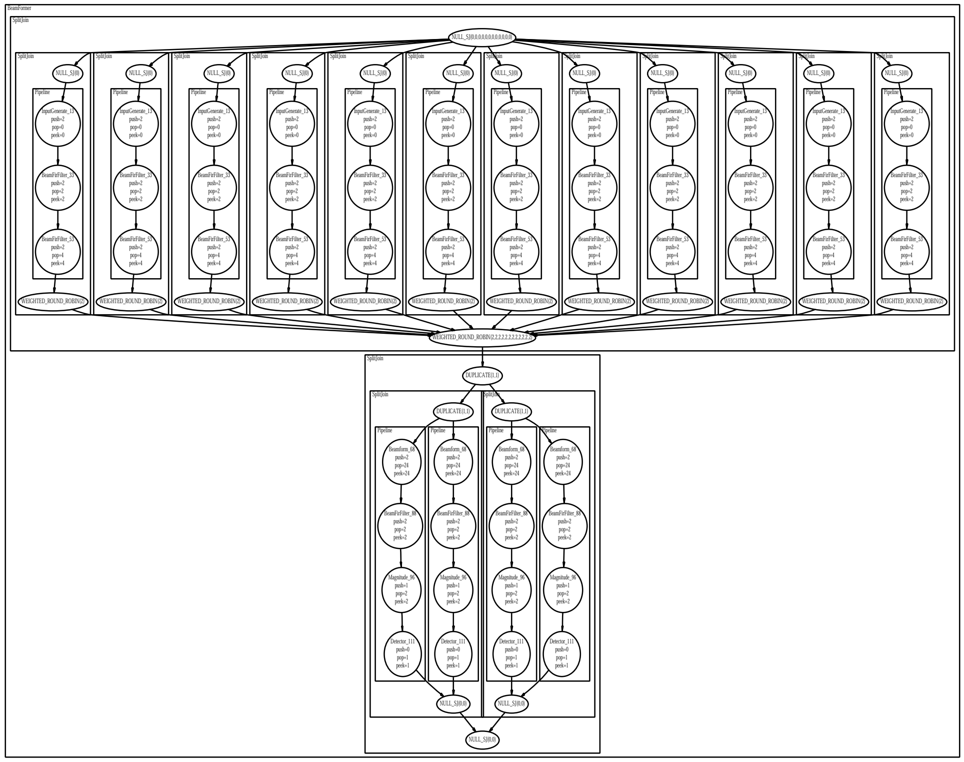


Figure B-1: Radar before partitioning.

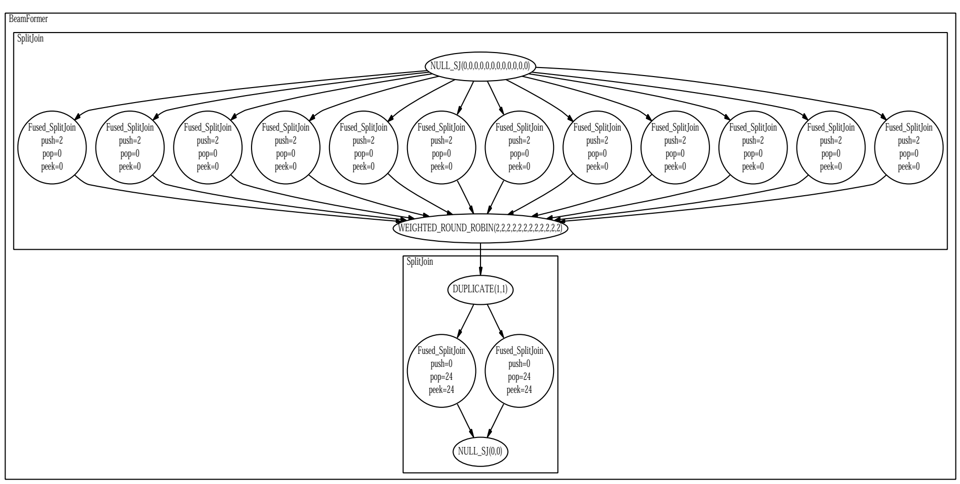


Figure B-2: Radar after partitioning.

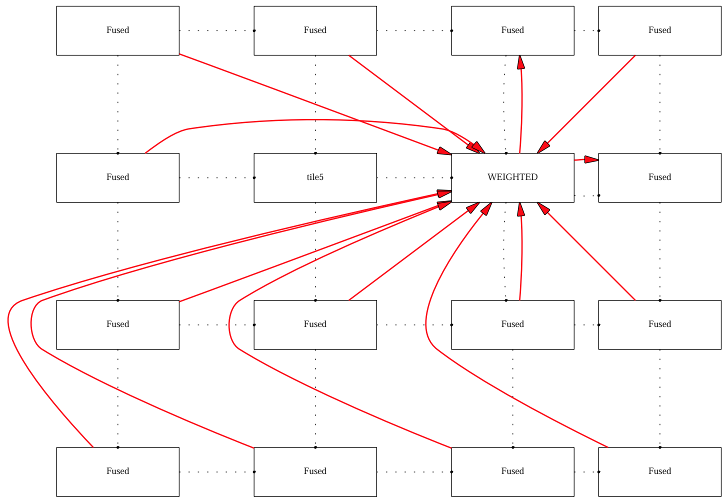


Figure B-3: Radar layout.

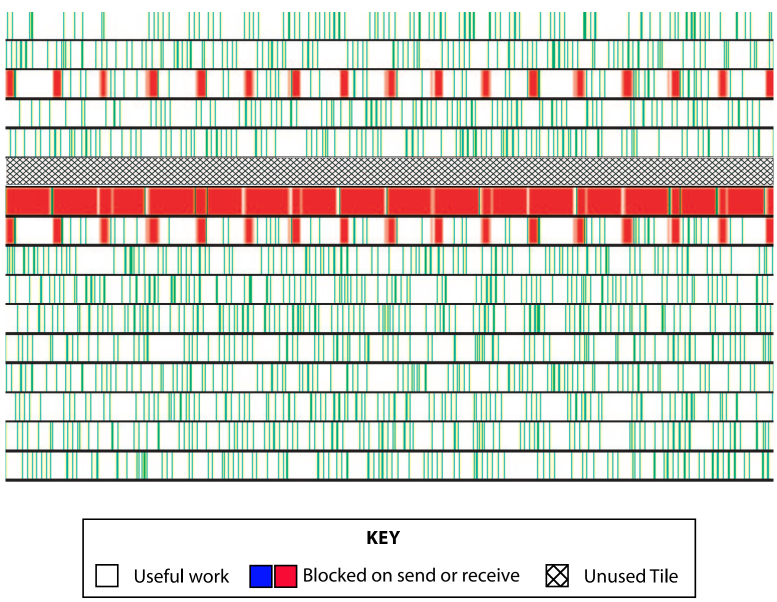


Figure B-4: Radar execution trace.

Appendix C

FM Radio Application

C.1 Description

This benchmark is a software implementation of an FM Radio.

C.2 Code

```
import streamit.*;
import streamit.io.*;
import java.lang.Math;

/*
 * Software equalizer.  This version uses
 * n+1 low-pass filters directly,
 * as opposed to n band-pass filters,
 * each with two low-pass filters.
 * The important observation is that
 * we have bands 1-2, 2-4, 4-8, ...
 * This means that we should run an LPF
 * for each intermediate frequency,
 * rather than two per band.  Calculating
 * this in StreamIt isn't that bad.
 * For a four-band equalizer:
 *
 *              |
 *             DUP
 *    +---------+---------+
 *    |         |         |
 *    |        DUP        |
 *    |    +----+----+    |
 *    |    |    |    |    |
 *   16    8    4    2    1
 *    |    |    |    |    |
 *    |  (dup)(dup)(dup)  |
 *    |    |    |    |    |
 *    |    +----+----+    |
 *    |       RR(2)       |
 *    |         |         |
 *    +---------+---------+
 *       WRR(1,2(n-1),1)
 *              |
 *            (a-b)
 *              |
 *            SUM(n)
 *              |
 *
 * It's straightforward to change
 * the values of 1, 16, and n.  Coming out
 * of the EqualizerSplitJoin is 16 8 8 4 4 2 2 1;
 * we can subtract and scale
 * these as appropriate to equalize.
 */

class FloatNAdder extends Filter
{
    int N;

    public FloatNAdder(int count)
    {
        super(count);
    }
    public void init (final int count)
    {
        N = count;
        input = new Channel (Float.TYPE, count, count);
        output = new Channel (Float.TYPE, 1);
    }

    public void work() {
        float sum = 0.0f;
        int i;
        for (i = 0; i < N; i++)
            sum += input.popFloat();
        output.pushFloat(sum);
    }
}

class FloatDiff extends Filter
{
    public void init()
    {
        input = new Channel(Float.TYPE, 2, 2);
        output = new Channel(Float.TYPE, 1);
    }
    public void work()
    {
        output.pushFloat(input.peekFloat(0) -
                         input.peekFloat(1));
        input.popFloat();
        input.popFloat();
    }
}

class FloatDup extends Filter
{
    public void init()
    {
        input = new Channel(Float.TYPE, 1, 1);
        output = new Channel(Float.TYPE, 2);
    }
    public void work()
    {
        float val = input.popFloat();
        output.pushFloat(val);
        output.pushFloat(val);
    }
}

class EqualizerInnerPipeline extends Pipeline
```

```
{
    public EqualizerInnerPipeline(float rate,
                                  float freq)
    {
        super(rate, freq);
    }
    public void init(final float rate, final float freq)
    {
        add(new LowPassFilter(rate, freq, 64, 0));
        add(new FloatDup());
    }
}

class EqualizerInnerSplitJoin extends SplitJoin
{
    public EqualizerInnerSplitJoin(float rate,
                                   float low,
                                   float high,
                                   int bands)
    {
        super(rate, low, high, bands);
    }
    public void init(final float rate,
                     final float low, final float high,
                     final int bands)
    {
        int i;
        setSplitter(DUPLICATE());
        for (i = 0; i < bands - 1; i++)
            add(new EqualizerInnerPipeline
                (rate,
                 (float)java.lang.Math.exp
                 ((i+1) *
                  (java.lang.Math.log(high) -
                   java.lang.Math.log(low)) /
                  bands + java.lang.Math.log(low))));
        setJoiner(ROUND_ROBIN(2));
    }
}

class EqualizerSplitJoin extends SplitJoin {

    public EqualizerSplitJoin(float rate,
                              float low, float high,
                              int bands)
    {
        super(rate, low, high, bands);
    }

    public void init(final float rate,
                     final float low, final float high,
                     final int bands)
    {
        // To think about: gains.

        setSplitter(DUPLICATE());
        add(new LowPassFilter(rate, high, 64, 0));
        add(new EqualizerInnerSplitJoin(rate, low,
                                        high, bands));
        add(new LowPassFilter(rate, low, 64, 0));
        setJoiner(WEIGHTED_ROUND_ROBIN(1, (bands-1)*2,
                                       1));
    }
}

/**
 * Class Equalizer
 *
 * Implements an Equalizer for an FM Radio
 */

class Equalizer extends Pipeline {

    public Equalizer(float rate)
    {
        super(rate);
    }

    public void init(final float rate)
    {
        final int bands = 10;
        final float low = 55;
        final float high = 1760;
        add(new EqualizerSplitJoin(rate, low,
                                   high, bands));
        add(new FloatDiff());
        add(new FloatNAdder(bands));
    }
}
```

```
class FloatOneSource extends Filter
{
    public void init ()
    {
        output = new Channel(Float.TYPE, 1);
    }
    public void work()
    {
        output.pushFloat(1);
    }
}


class FloatPrinter extends Filter
{
    public void init ()
    {
        input = new Channel(Float.TYPE, 1);
    }
    public void work ()
    {
        System.out.println(input.popFloat ());
    }
}


class FMRadio extends Pipeline
{
    public FMRadio()
    {
        super();
    }

    public void init()
    {
        final float samplingRate = 200000;
        final float cutoffFrequency = 108000000;
        final int numberOfTaps = 64;
        final float maxAmplitude = 27000;
        final float bandwidth = 10000;
        //decimate 4 samples after outputting 1
        add(new LowPassFilter(samplingRate,
                              cutoffFrequency,
                              numberOfTaps, 4));
        add(new FMDemodulator(samplingRate,
                              maxAmplitude, bandwidth));
        add(new Equalizer(samplingRate));
    }
}


/**
 * Class FMDemodulator
 *
 * Implements an FM Demodulator
 *
 */

class FMDemodulator extends Filter {

    float mGain;
    float sampleRate;
    float maxAmplitude;
    float modulationBandwidth;

    public FMDemodulator (float sampRate,
                          float max, float bandwidth)
    {
        super (sampRate, max, bandwidth);
    }

    public void init(float sampRate, float max,
                     float bandwidth)
    {
        input = new Channel (Float.TYPE, 1, 2);
        output = new Channel (Float.TYPE, 1);

        sampleRate = sampRate;
        maxAmplitude = max;
        modulationBandwidth = bandwidth;

        mGain =
            maxAmplitude*(sampleRate
                          /(modulationBandwidth *
                            (float)Math.PI));
    }

    public void work() {
        float temp = 0;
        //may have to switch to complex?
        temp = (float)((input.peekFloat(0)) *
```

```
                    (input.peekFloat(1)));
        //if using complex, use atan2
        temp = (float)(mGain * Math.atan(temp));

        input.popFloat();
        output.pushFloat(temp);
    }
}


/**
 * Class LowPassFilter
 *
 * Implements a Low Pass FIR Filter
 */

class LowPassFilter extends Filter {

    int numberOfTaps;
    float COEFF[];
    float cutoffFreq, samplingRate, tapTotal;
    int mDecimation;

    public LowPassFilter(float sampleRate,
                         float cutFreq,
                         int numTaps, int decimation)
    {
        super(sampleRate, cutFreq, numTaps, decimation);
    }

    public void init(final float sampleRate,
                     final float cutFreq,
                     final int numTaps,
                     final int decimation)
    {
        float pi, m, w;
        //float temptaps[];
        int i;
        samplingRate = sampleRate;
        cutoffFreq = cutFreq;
        numberOfTaps = numTaps;

        pi = (float)java.lang.Math.PI;
        //build the taps, and call super.init(taps[])
        //temptaps = new float[numberOfTaps];

        m = numberOfTaps -1;
        //from Oppenheim and Schafer,
        //m is the order of filter

        mDecimation = decimation;
        input = new Channel (Float.TYPE,
                             1+decimation, numTaps);
        output = new Channel (Float.TYPE, 1);

        //all frequencies are in hz
        COEFF = new float[numTaps];

        if(cutoffFreq == 0.0)
            {
                //Using a Hamming window for filter taps
                tapTotal = 0;

                for(i=0;i<numberOfTaps;i++)
                    {
                        COEFF[i] =
                            (float)(0.54 -
                                    0.46*
                                    java.lang.Math.cos
                                    ((2*pi)*(i/m)));
                        tapTotal = tapTotal + COEFF[i];
                    }

                //normalize all the taps to a sum of 1
                for(i=0;i<numberOfTaps;i++)
                    {
                        COEFF[i] = COEFF[i]/tapTotal;
                    }
            }
        else{
            //ideal lowpass filter ==> Hamming window
            //has IR h[n] = sin(omega*n)/(n*pi)
            //reference: Oppenheim and Schafer

            w = (2*pi) * cutoffFreq/samplingRate;

            for(i=0;i<numberOfTaps;i++)
                {
                    //check for div by zero
                    if(i-m/2 == 0)
                        COEFF[i] = w/pi;
                    else
                        COEFF[i] =
                            (float)
                            (java.lang.Math.sin(w*
                                                (i-m/2))
                             / pi
                             / (i-m/2) *
                             (0.54 - 0.4 *
                              java.lang.Math.cos
                              ((2*pi)*(i/m))));
                }
        }
        //COEFF = temptaps;
        // Is this actually useful?  StreamIt
        //doesn't like .length,
        // and at any rate, COEFF.length
        //will always be numTaps, which
        // will always have the same value as
        //numberOfTaps.  --dzm
        // numberOfTaps = COEFF.length;
    }

    public void work() {
        float sum = 0;
        int i;
        for (i=0; i<numberOfTaps; i++) {
            sum += input.peekFloat(i)*COEFF[i];
        }

        input.popFloat();
        for(i=0;i<mDecimation;i++)
            input.popFloat();
        output.pushFloat(sum);
    }
}

public class LinkedFMTest extends StreamIt
{
    static public void main(String[] t)
    {
        new LinkedFMTest().run(t);
    }

    public void init()
    {
        add(new FloatOneSource());
        add(new FMRadio());
        add(new FileWriter("fm-out", Float.TYPE));
    }
}
```

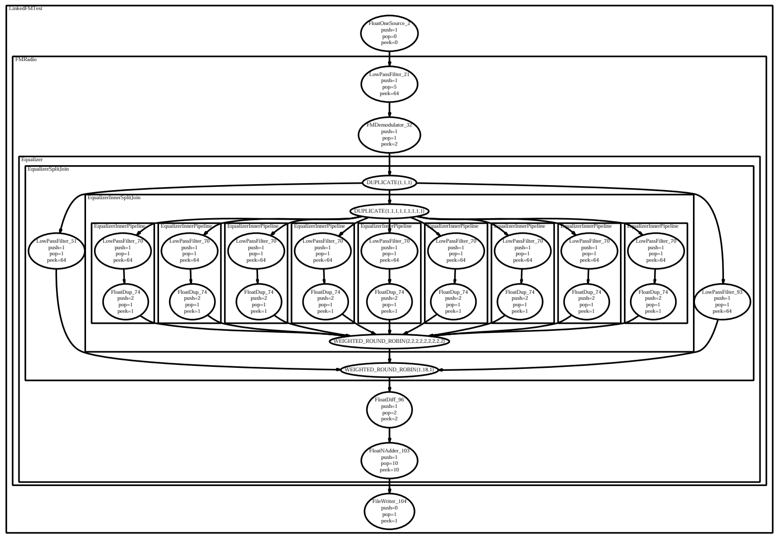


Figure C-1: Radio before partitioning.

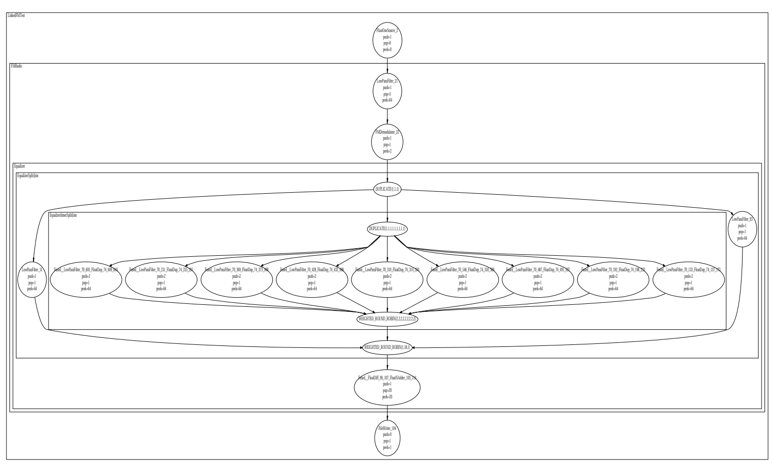


Figure C-2: Radio after partitioning.

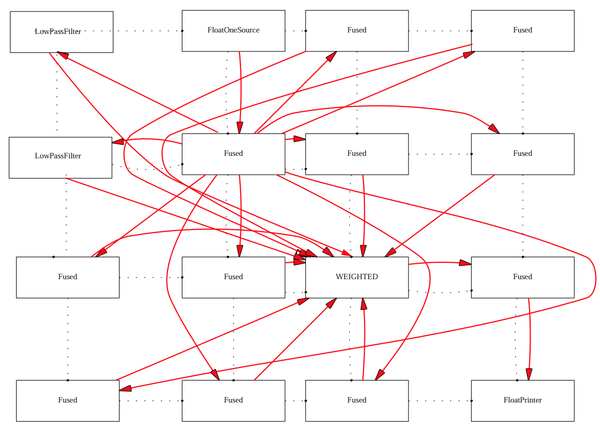


Figure C-3: Radio layout.

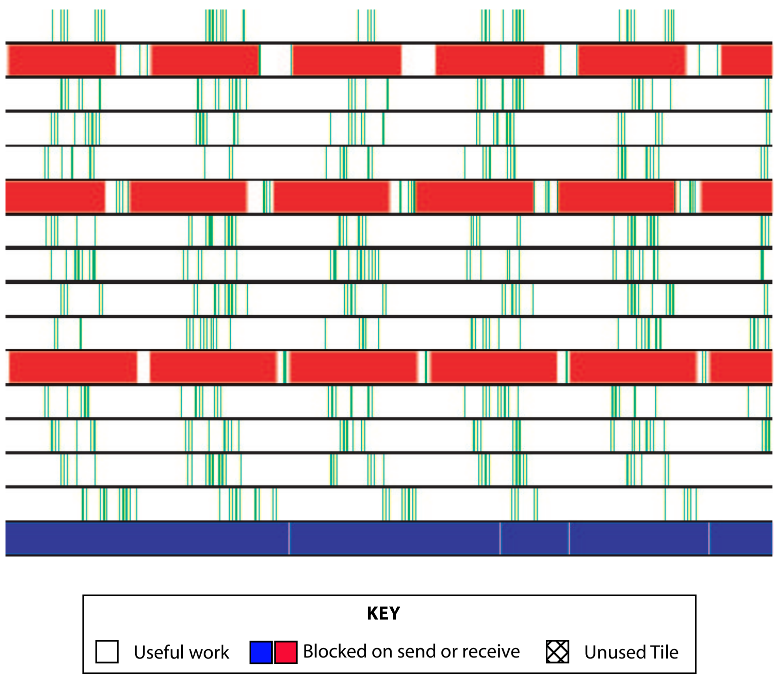


Figure C-4: Radio execution trace.

Appendix D

Bitonic Sort Application

D.1 Description

This benchmark performs a sort on a set of 32 input elements, using the Bitonic Sort algorithm[1]. See [24] Section 5.3.4 - "Networks for Sorting" (particularly the diagram titled "A nonstandard sorting network based on bitonic sorting" in the First Set of Exercises - Fig 56 in second edition).

D.2 Code

```
import streamit.*;

/**
  Compares the two input keys and
  exchanges their order if they are not  sorted.
  sortdir determines if the sort is nondecreasing
  (UP) or nonincreasing (DOWN).
  'true' indicates UP sort and 'false' indicates
  DOWN sort.
 */
class CompareExchange extends Filter
{
    boolean sortdir;
    public CompareExchange(boolean sortdir)
    {
        super(sortdir);
    }
    public void init(final boolean sortdir)
    {
        input = new Channel(Integer.TYPE, 2);
        output = new Channel(Integer.TYPE, 2);
        this.sortdir = sortdir;
    }
    public void work()
    {
        /* the input keys and min,max keys */
        int k1, k2, mink, maxk;

        k1 = input.popInt();
        k2 = input.popInt();
        if (k1 <= k2)
            {
                mink = k1;
                maxk = k2;
            }
        else /* k1 > k2 */
            {
                mink = k2;
                maxk = k1;
            }

        if (sortdir == true)
            {
                /* UP sort */
                output.pushInt(mink);
                output.pushInt(maxk);
            }
        else /* sortdir == false */
            {
                /* DOWN sort */
                output.pushInt(maxk);
                output.pushInt(mink);
            }
    }
}

/**
 Partition the input bitonic sequence of
 length L into two bitonic sequences
 of length L/2, with all numbers in the
 first sequence <= all numbers in the
 second sequence if sortdir is UP
 (similar case for DOWN sortdir)
 Graphically, it is a bunch of CompareExchanges
 with same sortdir, clustered
 together in the sort network
 at a particular step (of some merge stage).
 */
class PartitionBitonicSequence extends SplitJoin
{
    public PartitionBitonicSequence(int L,
                                    boolean sortdir)
    {
        super(L, sortdir);
    }
    public void init(final int L,
                     final boolean sortdir)
    {
        /* Each CompareExchange examines
           keys that are L/2 elements apart */
        this.setSplitter(ROUND_ROBIN());
        for (int i=0; i<(L/2); i++)
            this.add(new CompareExchange(sortdir));
        this.setJoiner(ROUND_ROBIN());
    }
}

/**
    One step of a particular merge stage
    (used by all merge stages except the last)
```

```
    dircnt determines which step we are in the
    current merge stage
    (which in turn is determined by <L, numseqp>)
 */
class StepOfMerge extends SplitJoin
{
    public StepOfMerge(int L, int numseqp, int dircnt)
    {
        super(L, numseqp, dircnt);
    }
    public void init(final int L, final int numseqp,
                     final int dircnt)
    {
        boolean curdir;
        this.setSplitter(ROUND_ROBIN(L));
        for (int j=0; j<numseqp; j++)
            {
                /* finding out the curdir is a
                   bit tricky - the direction depends
                   only on the subsequence num during
                   the FIRST step. So to
                   determine the FIRST step subsequence
                   to which this sequence belongs,
                   divide this sequence's number j by
                   dircnt (bcoz 'dircnt' tells how many
                   subsequences of the current step make
                   up one subseq of the FIRST step).
                   Then, test if that result is even
                   or odd to determine if curdir is UP
                   or DOWN respec.
                */
                curdir = ( (j/dircnt)%2 == 0 );
                /* The last step needs special care to
                   avoid splitjoins with just one
                   branch. */
                if (L > 2)
                    this.add
                        (new PartitionBitonicSequence
                         (L, curdir));
                else /* PartitionBitonicSequence of t
                        he last step (L=2) is simply a
                        CompareExchange */
                    this.add
                        (new CompareExchange(curdir));
            }
        this.setJoiner(ROUND_ROBIN(L));
    }
}

/**
  One step of the last merge stage
  Main difference form StepOfMerge
  is the direction of sort.
  It is always in the same direction - sortdir.
*/
class StepOfLastMerge extends SplitJoin
{
    public StepOfLastMerge(int L, int numseqp,
                           boolean sortdir)
    {
        super(L, numseqp, sortdir);
    }
    public void init(final int L, final int numseqp,
                     final boolean sortdir)
    {
        this.setSplitter(ROUND_ROBIN(L));
        for (int j=0; j<numseqp; j++)
            {
                /* The last step needs special care to
                   avoid splitjoins with just one
                   branch. */
                if (L > 2)
                    this.add
                        (new PartitionBitonicSequence
                         (L, sortdir));
                else /* PartitionBitonicSequence of the
                        last step (L=2) is simply a
                        CompareExchange */
                    this.add
                        (new CompareExchange(sortdir));
            }
        this.setJoiner(ROUND_ROBIN(L));
    }
}

/* Divide the input sequence of length N into
   subsequences of length P and sort each of them
  (either UP or DOWN depending on what subsequence
  number [0 to N/P-1] they get - All even
  subsequences are sorted UP and all odd
  subsequences are sorted DOWN)
  In short, a MergeStage is N/P Bitonic Sorters
  of order P each.
  But, this MergeStage is implemented
  *iteratively* as logP STEPS.
 */
class MergeStage extends Pipeline
{
    public MergeStage(int P, int N)
    {
        super(P, N);
    }
    public void init(final int P, final int N)
    {
        int L, numseqp, dircnt;
        /* for each of the lopP steps (except the
           last step) of this merge stage */
        for (int i=1; i<P; i=i*2)
            {
                /* length of each sequence for the
                   current step - goes like
                   P,P/2,...,2 */
                L = P/i;
                /* numseqp is the number of
                   PartitionBitonicSequence-rs
                   in this step */
                numseqp = (N/P)*i;
                dircnt = i;

                this.add(new StepOfMerge(L, numseqp,
                                         dircnt));
            }
    }
}

/**
 The LastMergeStage is basically one
 Bitonic Sorter of order N i.e., it takes the
 bitonic sequence produced by the h
 previous merge stages and applies a
 bitonic merge on it to produce the final
 sorted sequence.
 This is implemented iteratively as logN steps
*/
class LastMergeStage extends Pipeline
{
    public LastMergeStage(int N, boolean sortdir)
    {
        super(N, sortdir);
    }
    public void init(final int N,
                     final boolean sortdir)
    {
        int L, numseqp;
        /* for each of the logN steps (except the
           last step) of this merge stage */
        for (int i=1; i<N; i=i*2)
            {
                /* length of each sequence for
                   the current step - goes like
                   N,N/2,...,2 */
                L = N/i;
                /* numseqp is the number of
                   PartitionBitonicSequence-rs
                   in this step */
                numseqp = i;

                this.add(new StepOfLastMerge(L, numseqp,
                                             sortdir));
            }
    }
}

/**
  The top-level kernel of bitonic-sort
  (iterative version) -
  It has logN merge stages and all merge
  stages except the last
  progressively builds a bitonic
  sequence out of the input sequence.
  The last merge stage acts on the
  resultant bitonic sequence
  to produce the final sorted sequence
  (sortdir determines if it is
  UP or DOWN).
 */
class BitonicSortKernel extends Pipeline
{
    public BitonicSortKernel(int N, boolean sortdir)
    {
        super(N, sortdir);
    }
    public void init(final int N,
                     final boolean sortdir)
```

```java
    {
        for (int i=2; i<=(N/2); i=2*i)
            this.add(new MergeStage(i, N));
        this.add(new LastMergeStage(N, sortdir));
    }
}

/**
 * Creates N keys and sends it out
 */
class KeySource extends Filter
{
    int N;
    int A[];

    public KeySource(int N)
    {
        super(N);
    }
    public void init(final int N)
    {
        output = new Channel(Integer.TYPE, N);
        this.N = N;

        /* Initialize the input. In future, might
         * want to read from file or generate a random
         * permutation.
         */
        A = new int[N];
        for (int i=0; i<N; i++)
            A[i] = (N-i);
    }
    public void work()
    {
        for (int i=0; i<N; i++)
            output.pushInt(A[i]);
    }
}

/**
 * Prints out the sorted keys and verifies if they
 * are sorted.
 */
class KeyPrinter extends Filter
{
    int N;
    public KeyPrinter(int N)
    {
        super(N);
    }
    public void init(final int N)
    {
        input = new Channel(Integer.TYPE, N);
        this.N = N;
    }
    public void work()
    {
        for (int i=0; i<(N-1); i++)
            {
                System.out.println(input.popInt());
            }
        System.out.println(input.popInt());
    }
}

class DoneTimer extends Filter
{
    int N;
    DoneTimer(int N) {
        super(N);
    }
    public void init(final int N) {
        this.N = N;
        input = new Channel(Integer.TYPE, N);
    }
    public void work() {
        for(int i=0; i < N; i++)
            input.pop();
        System.out.print("Done");
    }
}

/**
 * The driver class
 */
class BitonicSort extends StreamIt
{
    public static void main(String args[])
    {
        (new BitonicSort()).run(args);
    }
    public void init()
    {
        /* Make sure N is a power_of_2 */
        final int N = 32; //16;
        /* true for UP sort and false for DOWN sort */
        final boolean sortdir = true;

        this.add(new KeySource(N));
        this.add(new BitonicSortKernel(N, sortdir));
        this.add(new DoneTimer(N));
    }
}
```

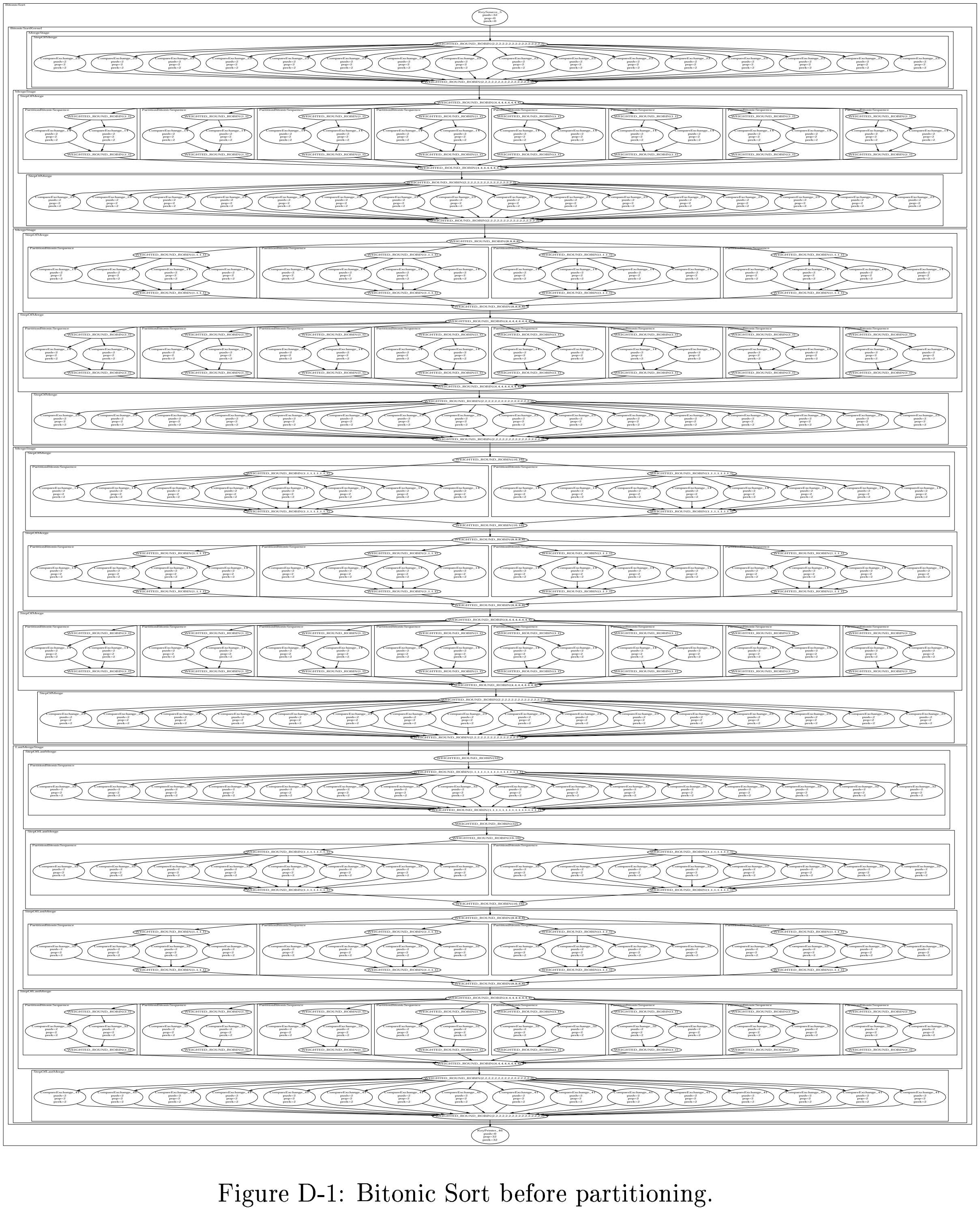


Figure D-1: Bitonic Sort before partitioning.

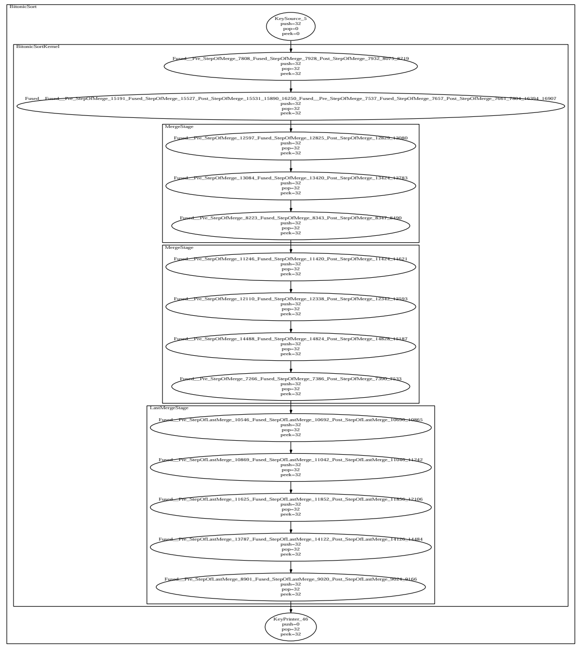


Figure D-2: Bitonic Sort after partitioning.

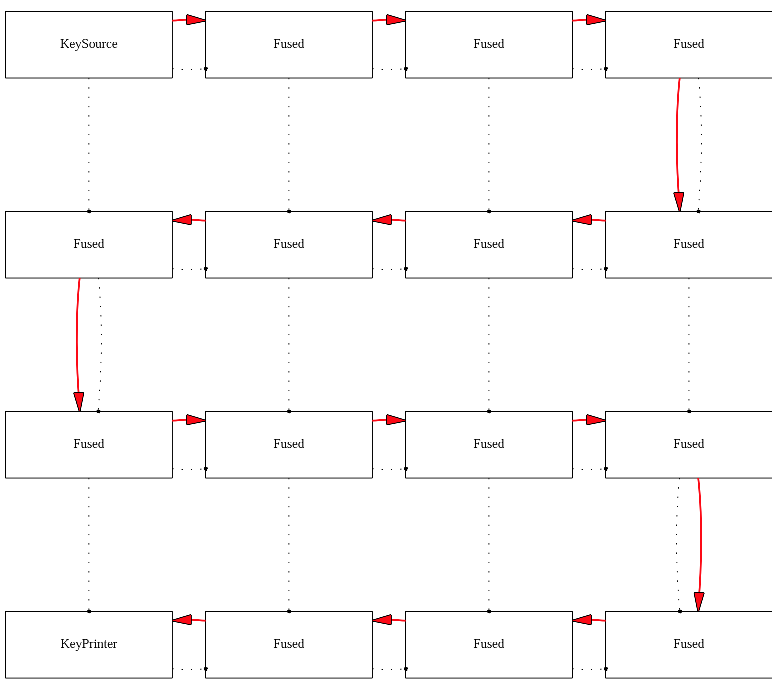


Figure D-3: Bitonic Sort layout.

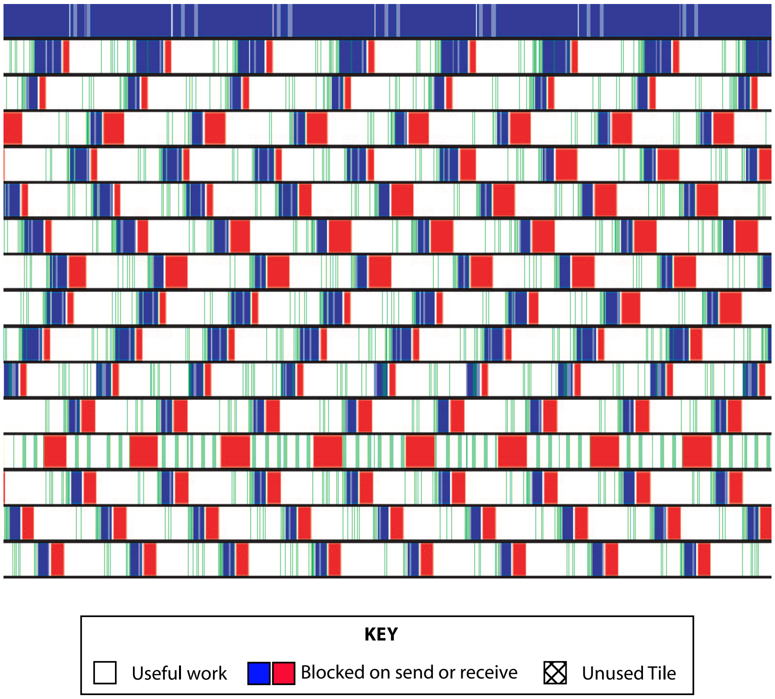


Figure D-4: Bitonic Sort execution trace.

Appendix E

FFT Application

E.1 Description

This benchmark is an FFT on a set of 64 points. A full description of the algorithm can be found at [2]. Details of the Decimation In Time FFT implemented here can be found at [3].

E.2 Code

```
    import streamit.*;

class CombineDFT extends Filter
{
    CombineDFT(int i)
    {
        super(i);
    }
    float wn_r, wn_i;
    int nWay;
    float results[];
    public void init(int n)
    {
        nWay = n;
        input = new Channel(Float.TYPE, 2 * n);
        output = new Channel(Float.TYPE, 2 * n);
        wn_r = (float) Math.cos(2 * 3.141592654 /
                                ((double) n));
        wn_i = (float) Math.sin(2 * 3.141592654 /
                                ((double) n));
        results = new float[2 * n];
    }

    public void work()
    {
        int i;
        float w_r = 1;
        float w_i = 0;
        for (i = 0; i < nWay; i += 2)
            {
                float y0_r = input.peekFloat(i);
                float y0_i = input.peekFloat(i+1);
                float y1_r = input.peekFloat(nWay + i);
                float y1_i =
                    input.peekFloat(nWay + i + 1);

                float y1w_r = y1_r * w_r - y1_i * w_i;
                float y1w_i = y1_r * w_i + y1_i * w_r;

                results[i] = y0_r + y1w_r;
                results[i + 1] = y0_i + y1w_i;

                results[nWay + i] = y0_r - y1w_r;
                results[nWay + i + 1] = y0_i - y1w_i;

                float w_r_next =
                    w_r * wn_r - w_i * wn_i;
                float w_i_next =
                    w_r * wn_i + w_i * wn_r;
                w_r = w_r_next;
                w_i = w_i_next;
            }

        for (i = 0; i < 2 * nWay; i++)
            {
                input.popFloat ();
                output.pushFloat(results[i]);
            }
    }
}

class FFTReorderSimple extends Filter
{
    FFTReorderSimple (int i) { super (i); }

    int nWay;
    int totalData;

    public void init (int n)
    {
        nWay = n;
        totalData = nWay * 2;

        input = new Channel (Float.TYPE, n * 2);
        output = new Channel (Float.TYPE, n * 2);
    }

    public void work ()
    {
        int i;

        for (i = 0; i < totalData; i+=4)
            {
                output.pushFloat (input.peekFloat (i));
                output.pushFloat
                    (input.peekFloat (i+1));
            }

        for (i = 2; i < totalData; i+=4)
            {
                output.pushFloat (input.peekFloat (i));
                output.pushFloat
                    (input.peekFloat (i+1));
            }
```

```
        for (i=0;i<nWay;i++)
            {
                input.popFloat ();
                input.popFloat ();
            }
    }
}

class FFTReorder extends Pipeline
{
    FFTReorder (int i) { super (i); }

    public void init (int nWay)
    {
        int i;
        for (i=1; i<(nWay/2); i*=2) {
            add (new FFTReorderSimple (nWay/i));
        }
    }
}

class FFTKernel1 extends Pipeline
{
    public FFTKernel1 (int i) { super (i); }
    public void init (final int nWay)
    {
        if (nWay > 2) {
            add (new SplitJoin () {
                    public void init () {
                        setSplitter (ROUND_ROBIN (2));
                        add (new FFTKernel1 (nWay / 2));
                        add (new FFTKernel1 (nWay / 2));
                        setJoiner (ROUND_ROBIN (nWay));
                    }
                });
        }
        add (new CombineDFT (nWay));
    }
}

class FFTKernel2 extends SplitJoin
{
    public FFTKernel2(int i)
    {
        super(i);
    }
    public void init(final int nWay)
    {
        setSplitter(ROUND_ROBIN(nWay*2));
        for (int i=0; i<2; i++) {
            add (new Pipeline() {
                    public void init() {
                        add (new FFTReorder (nWay));
                        for (int j=2; j<=nWay; j*=2) {
                            add(new CombineDFT (j));
                        }
                    }
                });
        }
        setJoiner(ROUND_ROBIN(nWay*2));
    }
}

public class FFT2 extends StreamIt
{
    public static void main(String[] args)
    {
        new FFT2().run(args);
    }
    public void init()
    {
        add(new FFTTestSource(64));
        add(new FFTKernel2(64));
        add(new FloatPrinter());
    }
}
```

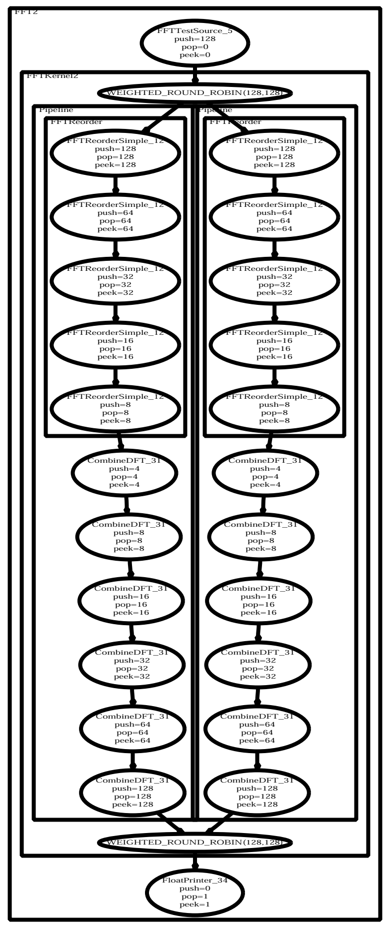


Figure E-1: FFT before partitioning.

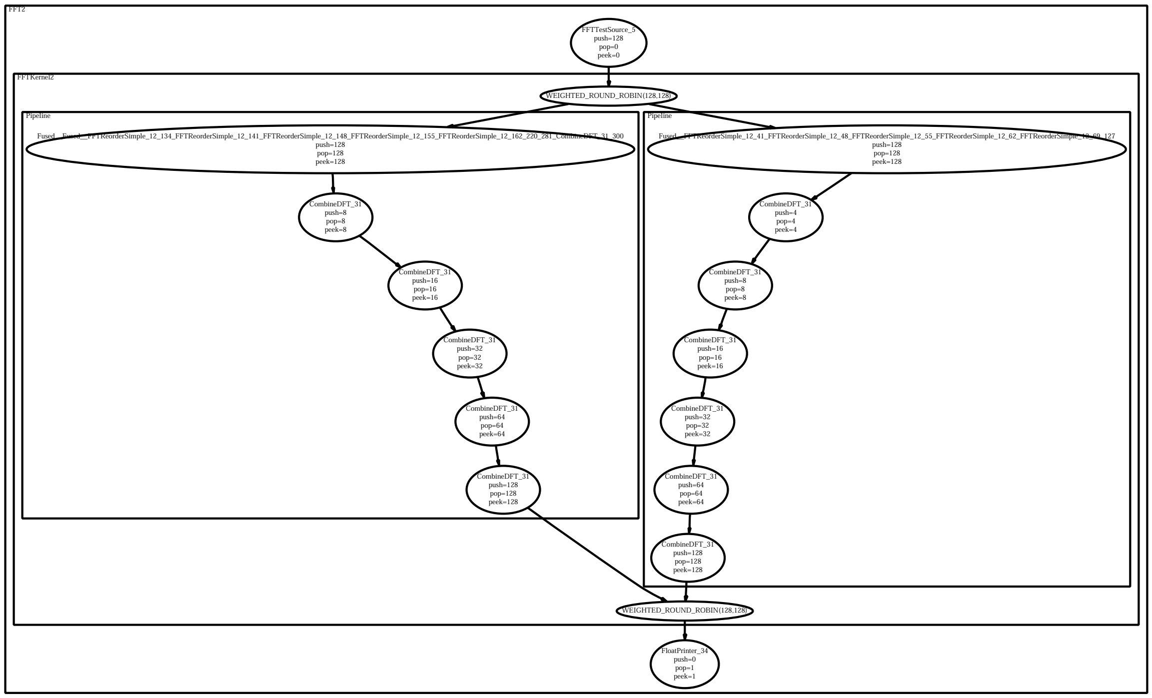


Figure E-2: FFT after partitioning.

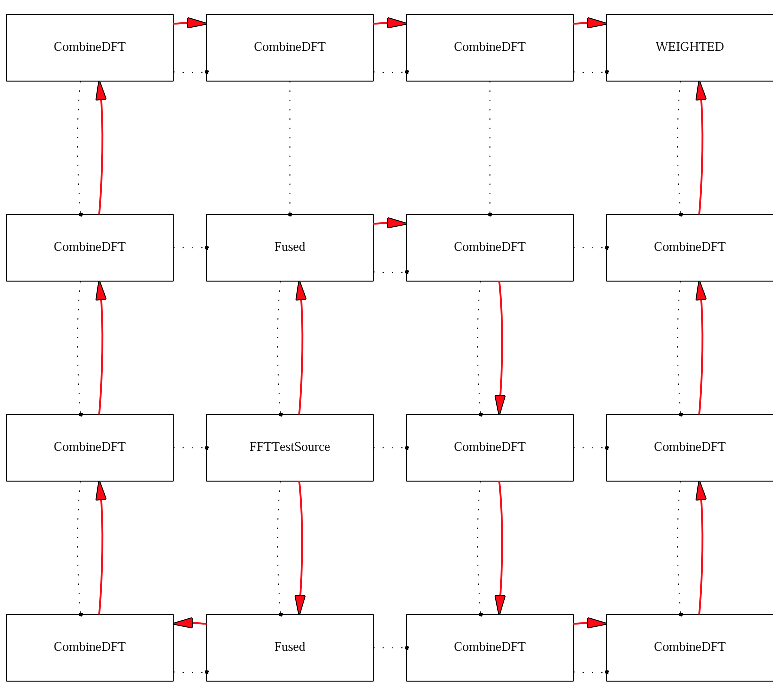


Figure E-3: FFT layout.

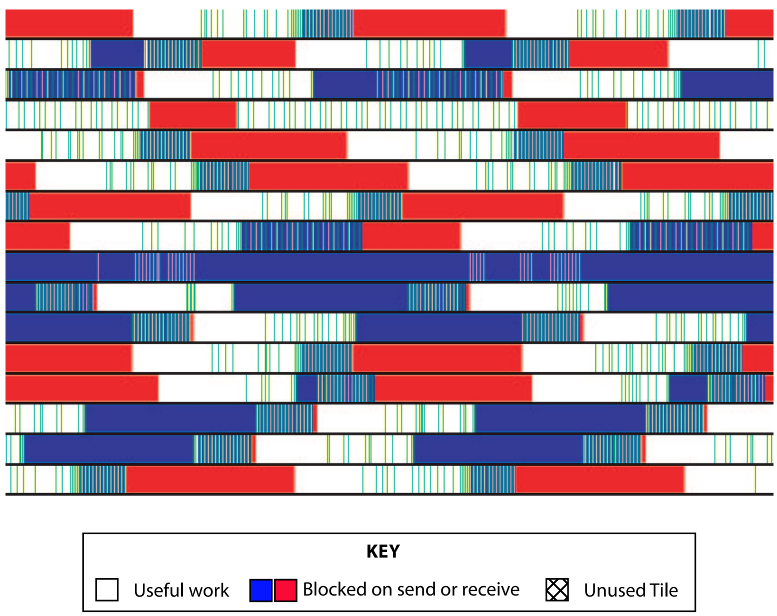


Figure E-4: FFT execution trace.

Appendix F

Filterbank Application

F.1 Description

This benchmark implements an 8-channel bank of filters. The input data is split into 64 different DFT filters, the output of which is then downsampled, upsampled, and recombined to form a processed signal.

F.2 Code

```
    import streamit.*;
import streamit.io.*;

/**
 * Class FirFilter
 *
 * Implements an FIR Filter
 */

class Bank extends Pipeline {


    public Bank (int N,float[] H,float[] F)
    {
        super (N,H,F);
    }

    public void init(  int N,float[] H,float[] F ) {
        add (new Delay_N(H.length-1));
        add (new FirFilter(H));
        add (new DownSamp(N));
        add (new UpSamp(N));
        add (new Delay_N(F.length-1));
        add (new FirFilter(F));
    }
}

// This is the complete Filter Bank Split Join Structure

/**
 * Class Branches
 *
 * Implements Branches  Structure
 */

class Branches extends SplitJoin {


    public Branches (int N_samp,int N_rows,
                     int N_col,float[][] H,float[][] F)

    {
        super (N_samp,N_rows,N_col,H,F);
    }

    public void init( int N_samp,int N_ch,
                      int N_col,float[][] H,
                      float[][] F ) {
        setSplitter(DUPLICATE());
        for (int i=0; i<N_ch ; i++)
            {
                float[] H_ch=new float[N_col];
                float[] F_ch=new float[N_col];
                for (int j=0; j<N_col;j++)
                    {
                        H_ch[j]=H[i][j];
                        F_ch[j]=F[i][j];
                    }
                add (new Bank(N_samp,H_ch,F_ch));
            }
        setJoiner(ROUND_ROBIN());
    }
}

class Combine extends Filter {

    public Combine(int N) {
        super(N);
    }
    int N;

    public void init(int N) {
        input = new Channel(Float.TYPE, N);
        output = new Channel(Float.TYPE, 1);
        this.N=N;
    }

    public void work() {

        float sum=0;
        for (int i=0;i<N;i++)
            sum+=input.popFloat();

        output.pushFloat(sum);
    }
}

class delay extends FeedbackLoop {

    public delay(int N) {
        super(N);
    }

    public void init(int N) {
```

```
        setSplitter(ROUND_ROBIN());
        setDelay(N);
        setBody(new Filter() {
                public void init() {
                    this.input =
                        new Channel(Float.TYPE, 2);
                    this.output =
                        new Channel(Float.TYPE, 2);
                }
                public void work() {
                    this.output.pushFloat
                        (this.input.peekFloat(1));
                    this.output.pushFloat
                        (this.input.peekFloat(0));
                    this.input.popFloat();
                    this.input.popFloat();
                }
            });
        setLoop(new Identity(Float.TYPE));
        setJoiner(ROUND_ROBIN());
    }

    public float initPathFloat(int index) {
        return 0.0f;
    }
}

/** Character Unit delay **/
class Delay_N extends Filter {
    float[] state;
    int N;
    int place_holder;

    public Delay_N(int N) {
        super(N);
    }

    public void init(int N) {
        // initial state of delay is 0
        state=new float[N];
        this.N=N;
        for (int i=0; i<N; i++)
            state[i]=0;
        input = new Channel(Float.TYPE,1);
        output = new Channel(Float.TYPE,1);
        place_holder=0;
    }
    public void work() {
 // push out the state and then update it with the input
 // from the channel
        output.pushFloat(state[place_holder]);
        state[place_holder] = input.popFloat();
        place_holder++;
        if (place_holder==N)
            place_holder=0;
    }
}

class DownSamp extends Filter {

    public DownSamp(int N) {
        super(N);
    }
    int N;

    public void init(int N) {
        input = new Channel(Float.TYPE, N);
        output = new Channel(Float.TYPE, 1);
        this.N=N;
    }

    public void work() {
        output.pushFloat(this.input.popFloat());
        for (int i=0;i<N-1;i++)
            input.popFloat();
    }
}

class source extends Filter {
    int N, pos;
    float[] r;
    public source(float[] r) {super(r);}
    public void init(float[] r){
        output = new Channel(Float.TYPE,1);
        this.r=r;
        N=r.length;
        this.pos = 0;
    }
    public void work(){
        output.pushFloat(r[pos++]);
        if (pos >= N) pos = 0;
    }
}

class sink extends Filter{
    int N;
    public sink(int N) {super(N);}
    public void init(int N){
        input = new Channel(Float.TYPE, 1);
        this.N=N;
    }
    public void work() {
        System.out.println(input.popFloat());
    }

}

class FBtest extends StreamIt {

    static public void main(String[] t)
    {
        FBtest test=new FBtest();
        test.run(t);
    }

    public void init() {
        int N_sim=1024*2;
        int N_samp=/* 32 */ 8;
        int N_ch=N_samp;
        int N_col=32;

        float[] r=new float[N_sim];
        float[][] H=new float[N_ch][N_col];
        float[][] F=new float[N_ch][N_col];

        for (int i=0;i<N_sim;i++)
            r[i]=i+1;

        for (int i=0;i<N_col;i++) {
            //sum+=1;
            //sum=sum/7;

            for (int j=0;j<N_ch;j++){
                //sum+=1;
                H[j][i]=i*N_col+j*N_ch+j+i+j+1;
                //sum++;
                F[j][i]=i*j+j*j+j+i;

            }
        }
        add (new source(r));
        add (new FilterBank(N_samp,N_ch,N_col,H,F));
        add (new sink(r.length));
    }
}

// This is the complete Filter Bank Split Join Structure

/**
 * Class Branches
 *
 * Implements Branches  Structure
 */
class FilterBank extends Pipeline {


    public FilterBank (int N_samp,int N_ch,
                       int N_col ,float[][] H,
                       float[][] F)
    {
        super (N_samp,N_ch,N_col,H,F);
    }

    public void init( int N_samp,int N_ch,
                      int N_col,float[][] H,
                      float[][] F ) {

        add (new Branches(N_samp,N_ch,N_col,H,F));
        add (new Combine(N_samp));

    }


}

// Together with a delay this creats an FIR

/**
 * Class FirFilter
 *
 * Implements an FIR Filter
 */
```

```
class FirFilter extends Filter {

    int N;
    float COEFF[];

    public FirFilter (float[] COEFF)
    {
        super (COEFF);
    }

    public void init(float[] COEFF) {
        this.N=COEFF.length;
        this.COEFF=new float[COEFF.length];

        for (int i=0; i<this.N;i++)
            this.COEFF[i]=COEFF[i];

        input =
            new Channel(Float.TYPE, 1, COEFF.length);
        output = new Channel(Float.TYPE, 1);
    }

    public void work(){
        float sum=0;
        for (int i=0; i<N ; i++)
            sum+=input.peekFloat(i)*COEFF[N-1-i];
        input.pop();
        output.pushFloat(sum);
    }
}

// This is the complete FIR pipeline

/**
 * Class FirFilter
 *
 * Implements an FIR Filter
 */

class FIR extends Pipeline {

    public FIR (float[] COEFF)
    {
        super (COEFF);
    }

    public void init(float[] COEFF) {
        add (new Delay_N(COEFF.length-1));
        add (new FirFilter(COEFF));
    }

}

class UpSamp extends Filter {

    public UpSamp(int N) {
        super(N);
    }
    int N;

    public void init(int N) {
        input = new Channel(Float.TYPE, 1);
        output = new Channel(Float.TYPE, N);
        this.N=N;
    }

    public void work() {
        output.pushFloat(this.input.popFloat());
        for (int i=0;i<N-1;i++)
            output.pushFloat(0);
    }
}
```

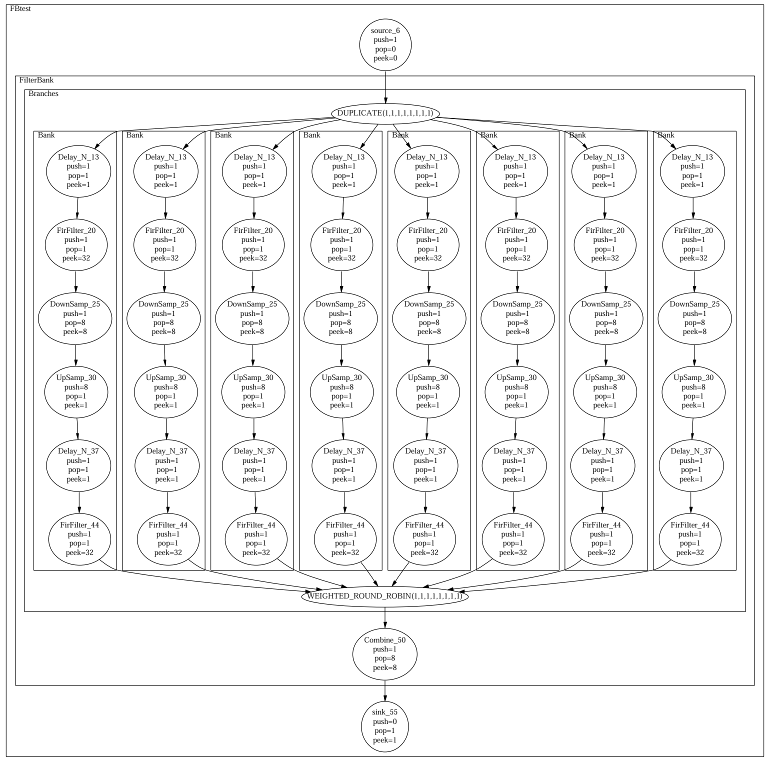


Figure F-1: Filterbank before partitioning.

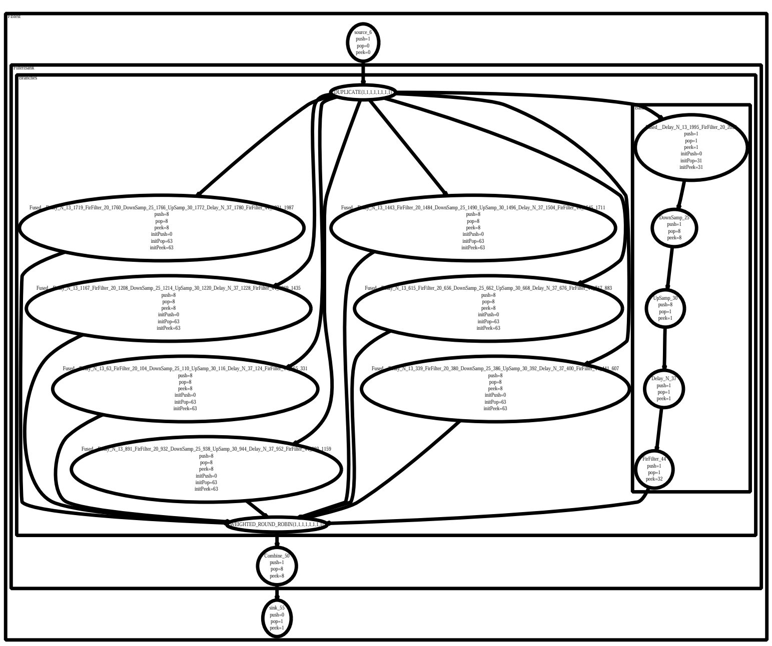


Figure F-2: Filterbank after partitioning.

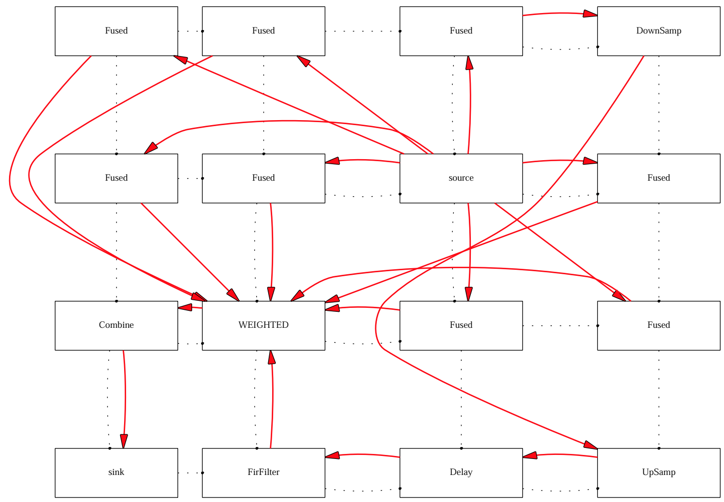


Figure F-3: Filterbank layout.

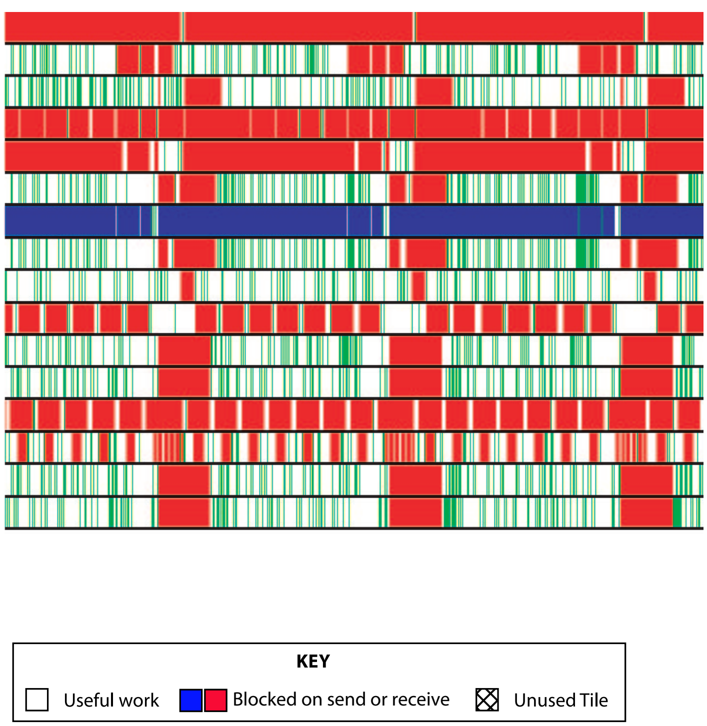


Figure F-4: Filterbank execution trace.

Appendix G

GSM Application

G.1 Description

The decoder portion of the StreamIt GSM Vocoder takes GSM encoded parameters as inputs, and uses these to synthesize audible speech. This is accomplished by processing the parameters through four main filters. The RPE decoder filter produces some "pink noise" that very loosely estimates the speech waveform, using quantized bit sequences and a maximum value parameter from the encoded input. This "pink noise" is fed to the Long Term Prediction portion, which applies long-term characteristics to the sequence through a delay filter within a feedback loop. The resulting signal is then sent to the Short Term Synthesis filter, which decodes high frequency voice characteristics from the encoded parameters and applies these to the signal. Finally, the Post-processing filter identifies peaks in the signal to make it audible. The c reference code for GSM was provided by the Communications and Operating Systems Research Group at the Technische Universitt Berlin.

G.2 Code

```
    import streamit.*;
import streamit.io.*;

class RPEDecodeFilter extends Filter
{
    short[] mXmc;
    short[] FAC;
    short[] xmp;
    short[] ep;

    short shortify(int a)
    {
        if (a >= 32767)
            {
                return 32767;
            }
        else
            {
                if (a <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) a;
                    }
            }
    }
```

```
short gsm_add(short a, short b)

{
    long ltmp = (long) a + (long) b;
    if (ltmp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (ltmp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) ltmp;
                }
        }
}

short gsm_sub(short a, short b)
{
    long ltmp = (long) a - (long) b;
    if (ltmp >= 32767)
        {
            return 32767;
        }
    else
```

```
        {
            if (ltmp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) ltmp;
                }
        }
}

short gsm_mult(short a, short b)
{
    long temp = (long) a * (long) b >> 15;
    if (temp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (temp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) temp;
                }
        }
}

short gsm_mult_r(short a, short b)
{
    long temp = ((long) a * (long) b) + 16384;
    short answer = (short) (temp >> 15);
    return answer;
}

short gsm_abs(short a)
{
    short answer;
    int temp;
    if (a < 0)
        {
            if (a == -32768)
                {
                    answer = 32767;
                }
            else
                {
                    temp = a * -1;
                    if (temp >= 32767)
                        {
                            answer = 32767;
                        }
                    else
                        {
                            if (temp <= -32768)
                                {
                                    answer = -32768;
                                }
                            else
                                {
                                    answer =
                                        (short) temp;
                                }
                        }
                }
        }
    else
        {
            answer = a;
        }
    return answer;
}

public void init()
{
    input = new Channel(Short.TYPE, 15);
    output = new Channel(Short.TYPE, 40);
    mXmc = new short[13];

    xmp = new short[13];
    ep = new short[40 ];

    FAC = new short[8];
    FAC[0] = 29218;
    FAC[1] = 26215;
    FAC[2] = 23832;
    FAC[3] = 21846;
    FAC[4] = 20165;
        FAC[5] = 18725;
        FAC[6] = 17476;
        FAC[7] = 16384;

    }

    public void work()
    {
        short i, k, xmaxc, mc, exp, mant,
            temp, temp1, temp2, temp3;

        for (i = 0; i < 13; i++)
            {
                mXmc[i] = input.popShort();
            }
        xmaxc = input.popShort();
        mc = input.popShort();
        exp = 0;
        if (xmaxc > 15)
            {
                exp = gsm_sub(shortify(xmaxc >> 3),
                              (short) 1);
            }
        mant = gsm_sub(xmaxc, shortify(exp << 3));

        if (mant == 0)
            {
                exp = -4;
                mant = 7;
            }
        else
            {
                while (mant <= 7)
                    {
                        mant = shortify(mant << 1 | 1);
                        exp--;
                    }
                mant = gsm_sub(mant, (short) 8);
            }

        temp1 = FAC[mant];
        temp2 = gsm_sub((short) 6, exp);
        temp3 = shortify(1 << gsm_sub(temp2, (short) 1));

        for (i = 0; i < 13; i++)
            {
                temp = gsm_sub(shortify(mXmc[i] << 1),
                               (short) 7);
                temp <<= 12;
                temp = gsm_mult_r(temp1, temp);
                temp = gsm_add(temp, temp3);
                xmp[i] = shortify(temp >> temp2);
            }

        for(k = 0; k < 40; k++)
            {
                ep[k] = 0;
            }
        for(i = 0; i < 12; i++)
            {

                ep[mc + (3 * i)] = xmp[i];
            }

        for (i = 0; i < 40 ; i++)
            {
                output.pushShort(ep[i]);
            }

    }
}

class LTPFilter extends Filter
{
    short[] QLB;
    short[] drp;
    short nrp;

    short shortify(int a)
    {
        if (a >= 32767)
            {
                return 32767;
            }
        else
            {
                if (a <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
```

```
                return (short) a;
            }
        }
}

short gsm_add(short a, short b)

{
    long ltmp = (long) a + (long) b;
    if (ltmp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (ltmp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) ltmp;
                }
        }
}

short gsm_sub(short a, short b)
{
    long ltmp = (long) a - (long) b;
    if (ltmp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (ltmp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) ltmp;
                }
        }
}

short gsm_mult(short a, short b)
{
    long temp = (long) a * (long) b >> 15;
    if (temp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (temp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) temp;
                }
        }
}

short gsm_mult_r(short a, short b)
{
    long temp = ((long) a * (long) b) + 16384;
    short answer = (short) (temp >> 15);
    return answer;
}

short gsm_abs(short a)
{
    short answer;
    int temp;
    if (a < 0)
        {
            if (a == -32768)
                {
                    answer = 32767;
                }
            else
                {
                    temp = a * -1;
                    if (temp >= 32767)
                        {
                            answer = 32767;
                        }
                    else
                        {
                            if (temp <= -32768)
                                {
                                    answer = -32768;
                                }
                            else
                                {
                                    answer =
                                        (short) temp;
                                }
                        }
                }
        }
    else
        {
            answer = a;
        }
    return answer;
}

public void init()
{
    input = new Channel(Short.TYPE, 162);
    output = new Channel(Short.TYPE, 1);
    drp = new short[160];
    nrp = 40;


    QLB = new short[4];
    QLB[0] = 3277;
    QLB[1] = 11469;
    QLB[2] = 21299;
    QLB[3] = 32767;

}

public void work()
{
    short i, nr, brp, drpp;
    short mBcr = input.popShort();
    short mNcr = input.popShort();

    for (i = 0; i < 160; i++)
        {
            drp[i] = input.popShort();
        }
    nr = mNcr;
    if ((mNcr < 40) || (mNcr > 120))
        {
            nr = nrp;
        }
    nrp = nr;


    brp = QLB[mBcr];
    drpp = 1;
    for (i = 121; i < 161; i++)
        {
            drpp = gsm_mult_r(brp, drp[i - nr]);
        }

    output.pushShort(drpp);

}
}

class AdditionUpdateFilter extends Filter
{
    short[] ep;
    short[] drp;

    short shortify(int a)
    {
        if (a >= 32767)
            {
                return 32767;
            }
        else
            {
                if (a <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) a;
                    }
            }
    }

    short gsm_add(short a, short b)
```

```
{
    long ltmp = (long) a + (long) b;
    if (ltmp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (ltmp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) ltmp;
                }
        }
}

short gsm_sub(short a, short b)
{
    long ltmp = (long) a - (long) b;
    if (ltmp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (ltmp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) ltmp;
                }
        }
}

short gsm_mult(short a, short b)
{
    long temp = (long) a * (long) b >> 15;
    if (temp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (temp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) temp;
                }
        }
}

short gsm_mult_r(short a, short b)
{
    long temp = ((long) a * (long) b) + 16384;
    short answer = (short) (temp >> 15);
    return answer;
}

short gsm_abs(short a)
{
    short answer;
    int temp;
    if (a < 0)
        {
            if (a == -32768)
                {
                    answer = 32767;
                }
            else
                {
                    temp = a * -1;
                    if (temp >= 32767)
                        {
                            answer = 32767;
                        }
                    else
                        {
                            if (temp <= -32768)
                                {
                                    answer = -32768;
                                }
                            else
                                {
                                    answer =
                                                    (short) temp;
                                }
                        }
                }
        }
    else
        {
            answer = a;
        }
    return answer;
}

public void init()
{
    short i;
    input = new Channel(Short.TYPE, 41);
    output = new Channel(Short.TYPE, 160);
    ep = new short[40];
    drp = new short[160 ];
    for (i = 0; i < 160 ; i++)
        {
            drp[i] = 0;
        }
}

public void work()
{
    short i, j, k, drpp;

    for (i = 0; i < 40; i++)
        {
            ep[i] = input.popShort();
        }
    drpp = input.popShort();

    for (j = 121; j < 160; j++)
        {
            drp[j] = gsm_add(ep[j - 121], drpp);
        }
    for (k = 0; k < 120; k++)
        {
            drp[k] = drp[k + 40];
        }

    for (i = 0; i < 160 ; i++)
        {
            output.pushShort(drp[i]);
        }

}
}

class ReflectionCoeffLARppInternal extends Filter
{
    short shortify(int a)
    {
        if (a >= 32767)
            {
                return 32767;
            }
        else
            {
                if (a <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) a;
                    }
            }
    }

    short gsm_add(short a, short b)

    {
        long ltmp = (long) a + (long) b;
        if (ltmp >= 32767)
            {
                return 32767;
            }
        else
            {
                if (ltmp <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) ltmp;
                    }
            }
```

```
    }

    short gsm_sub(short a, short b)
    {
        long ltmp = (long) a - (long) b;
        if (ltmp >= 32767)
            {
                return 32767;
            }
        else
            {
                if (ltmp <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) ltmp;
                    }
            }
    }

    short gsm_mult(short a, short b)
    {
        long temp = (long) a * (long) b >> 15;
        if (temp >= 32767)
            {
                return 32767;
            }
        else
            {
                if (temp <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) temp;
                    }
            }
    }

    short gsm_mult_r(short a, short b)
    {
        long temp = ((long) a * (long) b) + 16384;
        short answer = (short) (temp >> 15);
        return answer;
    }

    short gsm_abs(short a)
    {
        short answer;
        int temp;
        if (a < 0)
            {
                if (a == -32768)
                    {
                        answer = 32767;
                    }
                else
                    {
                        temp = a * -1;
                        if (temp >= 32767)
                            {
                                answer = 32767;
                            }
                        else
                            {
                                if (temp <= -32768)
                                    {
                                        answer = -32768;
                                    }
                                else
                                    {
                                        answer =
                                            (short) temp;
                                    }
                            }
                    }
            }
        else
            {
                answer = a;
            }
        return answer;
    }

    short INVA, MIC, B;

    public ReflectionCoeffLARppInternal(short INVA,
                                        short MIC,
                                        short B)
    {
        super(INVA, MIC, B);
    }

    public void init(final short INVA, final short MIC,
                     final short B)
    {
        input = new Channel(Short.TYPE, 1);
        output = new Channel(Short.TYPE, 1);
        this.INVA = INVA;
        this.MIC = MIC;
        this.B = B;
    }

    public void work()
    {
        short LARc, LARpp, temp1, temp2;

        LARc = input.popShort();
        temp1 = shortify((gsm_add(LARc, MIC)) << 10);
        temp2 = shortify(B << 10);
        temp1 = gsm_sub(temp1, temp2);
        temp1 = gsm_mult_r(INVA, temp1);
        LARpp = gsm_add(temp1, temp1);
        output.pushShort(LARpp);
    }
}

class ReflectionCoeffLARpp extends SplitJoin
{
    public void init()
    {
        setSplitter(ROUND_ROBIN());

        add(new ReflectionCoeffLARppInternal
            ((short)13107, (short)-32, (short)0));
        add(new ReflectionCoeffLARppInternal
            ((short)13107, (short)-32, (short)0));
        add(new ReflectionCoeffLARppInternal
            ((short)13107, (short)-16, (short)2048));
        add(new ReflectionCoeffLARppInternal
            ((short)13107, (short)-16, (short)-2560));
        add(new ReflectionCoeffLARppInternal
            ((short)19223, (short)-8, (short)94));
        add(new ReflectionCoeffLARppInternal
            ((short)17476, (short)-8, (short)-1792));
        add(new ReflectionCoeffLARppInternal
            ((short)31454, (short)-4, (short)-341));
        add(new ReflectionCoeffLARppInternal
            ((short)29708, (short)-4, (short)-1144));
        setJoiner(ROUND_ROBIN());
    }
}

class ReflectionCoeffLARpInternal extends Filter
{
    short shortify(int a)
    {
        if (a >= 32767)
            {
                return 32767;
            }
        else
            {
                if (a <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) a;
                    }
            }
    }

    short gsm_add(short a, short b)

    {
        long ltmp = (long) a + (long) b;
        if (ltmp >= 32767)
            {
                return 32767;
            }
        else
            {
                if (ltmp <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) ltmp;
```

```
                }
        }
}

short gsm_sub(short a, short b)
{
    long ltmp = (long) a - (long) b;
    if (ltmp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (ltmp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) ltmp;
                }
        }
}

short gsm_mult(short a, short b)
{
    long temp = (long) a * (long) b >> 15;
    if (temp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (temp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) temp;
                }
        }
}

short gsm_mult_r(short a, short b)
{
    long temp = ((long) a * (long) b) + 16384;
    short answer = (short) (temp >> 15);
    return answer;
}

short gsm_abs(short a)
{
    short answer;
    int temp;
    if (a < 0)
        {
            if (a == -32768)
                {
                    answer = 32767;
                }
            else
                {
                    temp = a * -1;
                    if (temp >= 32767)
                        {
                            answer = 32767;
                        }
                    else
                        {
                            if (temp <= -32768)
                                {
                                    answer = -32768;
                                }
                            else
                                {
                                    answer =
                                        (short) temp;
                                }
                        }
                }
        }
    else
        {
            answer = a;
        }
    return answer;
}

short mprevLARpp;

public void init()
    {
        input = new Channel(Short.TYPE, 1);
        output = new Channel(Short.TYPE, 1);
        mprevLARpp = 0;
    }

    public void work()
    {
        int i, j, k;
        short mLARp, mLARpp;

        mLARpp = input.popShort();

        mLARp = 0;

        for (k = 0; k < 13; k++)
            {
                mLARp=gsm_add(shortify(mprevLARpp >> 2),
                              shortify(mLARpp >> 2));
                mLARp=gsm_add(mLARp,
                             shortify(mprevLARpp >> 1));
            }

        for (k = 13; k < 27; k++)
            mLARp = gsm_add(shortify(mprevLARpp >> 1),
                            shortify(mLARpp >> 1));

        for (k = 27; k < 39; k++)
            {
                mLARp=gsm_add(shortify(mprevLARpp >> 2),
                            shortify(mLARpp >> 2));
                mLARp = gsm_add(mLARp,
                              shortify(mLARpp >> 1));
            }
        mprevLARpp = mLARpp;
        output.pushShort(mLARp);
    }
}

class ReflectionCoeffLARp extends SplitJoin
{
    public void init()
    {
        setSplitter(ROUND_ROBIN());

        for (int i = 0; i < 8; i++)
            add(new ReflectionCoeffLARpInternal());
        setJoiner(ROUND_ROBIN());
    }
}

class ReflectionCoeffmrrp extends Filter
{
    short shortify(int a)
    {
        if (a >= 32767)
            {
                return 32767;
            }
        else
            {
                if (a <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) a;
                    }
            }
    }

    short gsm_add(short a, short b)

    {
        long ltmp = (long) a + (long) b;
        if (ltmp >= 32767)
            {
                return 32767;
            }
        else
            {
                if (ltmp <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) ltmp;
                    }
            }
    }
```

```
short gsm_sub(short a, short b)
{
    long ltmp = (long) a - (long) b;
    if (ltmp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (ltmp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) ltmp;
                }
        }
}

short gsm_mult(short a, short b)
{
    long temp = (long) a * (long) b >> 15;
    if (temp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (temp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) temp;
                }
        }
}

short gsm_mult_r(short a, short b)
{
    long temp = ((long) a * (long) b) + 16384;
    short answer = (short) (temp >> 15);
    return answer;
}

short gsm_abs(short a)
{
    short answer;
    int temp;
    if (a < 0)
        {
            if (a == -32768)
                {
                    answer = 32767;
                }
            else
                {
                    temp = a * -1;
                    if (temp >= 32767)
                        {
                            answer = 32767;
                        }
                    else
                        {
                            if (temp <= -32768)
                                {
                                    answer = -32768;
                                }
                            else
                                {
                                    answer =
                                        (short) temp;
                                }
                        }
                }
        }
    else
        {
            answer = a;
        }
    return answer;
}

public void init()
{
    input = new Channel(Short.TYPE, 1);
    output = new Channel(Short.TYPE, 1);
}
```

```
    public void work()
    {
        short mLARp, temp, mrrp;
        mLARp = input.popShort();
        temp = gsm_abs(mLARp);
        if (temp < 11059)
            temp = shortify(temp << 1);
        else if (temp < 20070)
            temp = gsm_add(temp, (short) 11059);
        else
            temp = gsm_add((short) (temp >> 2),
                           (short) 26112);
        mrrp = temp;
        if (mLARp < 0)
            mrrp = gsm_sub((short)0, mrrp);
        output.pushShort(mrrp);
    }
}

class ReflectionCoeffCalc extends Pipeline
{
    public void init()
    {
        add(new ReflectionCoeffLARpp());
        add(new ReflectionCoeffLARp());
        add(new ReflectionCoeffmrrp());
    }
}

class ReflectionCoeff extends SplitJoin
{
    public void init()
    {
        setSplitter(WEIGHTED_ROUND_ROBIN(160, 8));

        add(new Filter() {
                public void init()
                {
                    this.input = new Channel(Short.TYPE,
                                             160);
                    this.output = new Channel(Short.TYPE,
                                              40);
                }
                public void work()
                {
                    int i;
                    for (i = 0; i < 120; i++)
                        this.input.popShort();
                    for (i = 0; i < 40; i++)
                        this.output.pushShort
                            (this.input.popShort());
                }
            });

        add(new ReflectionCoeffCalc());
        setJoiner(WEIGHTED_ROUND_ROBIN(40, 8));
    }
}

class ShortTermReorder extends Filter
{
    short mdrp[];
    short mrrp[];

    public void init()
    {
        input = new Channel(Short.TYPE, 8 + 40);
        output = new Channel(Short.TYPE, (8 + 1) * 40);
        mdrp = new short[40];
        mrrp = new short[8];
    }

    public void work()
    {
        short val;
        int i, j;


        for (j = 0; j < 40; j++)
            mdrp[j] = input.popShort();
        for (j = 0; j < 8; j++)
            mrrp[j] = input.popShort();


        for (i = 0; i < 40; i++)
            {
                for (j = 0; j < 8; j++)
                    output.pushShort(mrrp[j]);
                output.pushShort(mdrp[i]);
            }
    }
}
```

```
class ShortTermSynthCalc extends Filter
{
    short[] mrrp;
    short[] v;




    short shortify(int a)
    {
        if (a >= 32767)
            {
                return 32767;
            }
        else
            {
                if (a <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) a;
                    }
            }
    }

    short gsm_add(short a, short b)

    {
        long ltmp = (long) a + (long) b;
        if (ltmp >= 32767)
            {
                return 32767;
            }
        else
            {
                if (ltmp <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) ltmp;
                    }
            }
    }

    short gsm_sub(short a, short b)
    {
        long ltmp = (long) a - (long) b;
        if (ltmp >= 32767)
            {
                return 32767;
            }
        else
            {
                if (ltmp <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) ltmp;
                    }
            }
    }

    short gsm_mult(short a, short b)
    {
        long temp = (long) a * (long) b >> 15;
        if (temp >= 32767)
            {
                return 32767;
            }
        else
            {
                if (temp <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) temp;
                    }
            }
    }

    short gsm_mult_r(short a, short b)
    {
        long temp = ((long) a * (long) b) + 16384;
        short answer = (short) (temp >> 15);
        return answer;
    }

    short gsm_abs(short a)
    {
        short answer;
        int temp;
        if (a < 0)
            {
                if (a == -32768)
                    {
                        answer = 32767;
                    }
                else
                    {
                        temp = a * -1;
                        if (temp >= 32767)
                            {
                                answer = 32767;
                            }
                        else
                            {
                                if (temp <= -32768)
                                    {
                                        answer = -32768;
                                    }
                                else
                                    {
                                        answer =
                                            (short) temp;
                                    }
                            }
                    }
            }
        else
            {
                answer = a;
            }
        return answer;
    }

    public void init()
    {
        input = new Channel(Short.TYPE, 8  + 1);
        output = new Channel(Short.TYPE, 1);
        mrrp = new short[8 ];
        v = new short[9 ];
        for (int i = 0; i < 9 ; i++)
            v[i] = 0;
    }

    public void work()
    {
        int i;
        short sri;

        for (i = 0; i < 8 ; i++)
            mrrp[i] = input.popShort();
        sri = input.popShort();
        for (i = 1; i < 8; i++)
            {
                sri=gsm_sub(sri,
                               gsm_mult(mrrp[8-i], v[8-i]));
                v[9-i]=gsm_add(v[8-i],
                                   gsm_mult_r(mrrp[8-i],sri));
            }
        v[0] = sri;
        output.pushShort(sri);
    }
}

class ShortTermSynth extends Pipeline
{
    public void init()
    {
        add(new ShortTermReorder());
        add(new ShortTermSynthCalc());
    }
}

class LARInputFilter extends Filter
{

    short[] mdata;
    short[] single_frame;
    boolean donepushing;
    public short[] mLarParameters;
    public short[] mLtpOffset;
    public short[] mLtpGain;
    public short[] mRpeGridPosition;
```

```
public short[] mRpeMagnitude;
public short[] mSequence;

public void initInputArrays() {
    mLarParameters = new short[8];
    mLtpOffset = new short[4];
    mLtpGain = new short[4];
    mRpeGridPosition = new short[4];
    mRpeMagnitude = new short[4];
    mSequence = new short[4*13];
}

public void getParameters(short[] input)
{
    int i, j, k, l, m;
    int input_index = 0;
    int num_bits = 0;

    initInputArrays();
    for(i = 0; i < 8; i++)
        {

            switch(i)
                {
                case 0:
                case 1:      num_bits = 6;
                    break;
                case 2:
                case 3:      num_bits = 5;
                    break;
                case 4:
                case 5:      num_bits = 4;
                    break;
                case 6:
                case 7:      num_bits = 3;
                    break;
                }


            mLarParameters[i] = 0;
            for (j = 0; j < num_bits; j++,
                     input_index++)
                {
                    mLarParameters[i] |=
                        input[input_index] <<
                        (num_bits - 1 - i);
                }
        }


    for (k = 0; k < 4; k++)
        {
            mLtpOffset[k] = 0;
            for (l = 0; l < 7; l++)
                {
                    mLtpOffset[k] |=
                        input[input_index] <<
                        (6 - l);
                    input_index++;
                }
            mLtpGain[k] = 0;
            for (l = 0; l < 2; l++)
                {
                    mLtpGain[k] |=
                        input[input_index] << (1 - l);
                    input_index++;
                }
            mRpeGridPosition[k] = 0;
            for (l = 0; l < 2; l++)
                {
                    mRpeGridPosition[k] |=
                        input[input_index] << (1 - l);
                    input_index++;
                }
            mRpeMagnitude[k] = 0;
            for (l = 0; l < 6; l++)
                {
                    mRpeMagnitude[k] |=
                        input[input_index] << (5 - l);
                    input_index++;
                }
            for(l = 0; l < 13; l++)
                {
                    mSequence[k+4*l] = 0;
                    for (m = 0; m < 3; m++)
                        {
                            mSequence[k+4*l] |=
                                input[input_index] <<
                                (2 - m);
                            input_index++;
                        }
```

```
                }
        }
    }
    public void init()
    {
        mdata = new short[260 ];
        single_frame = new short[260 ];
        input = new Channel(Short.TYPE, 260);
        output = new Channel(Short.TYPE, 8);
        donepushing = false;
    }

    public void work()
    {
        int i, j, k;

        for (i = 0; i < 260 ; i++)
            {
                mdata[i] = input.popShort();
            }

        if (donepushing)
            {

            }


        for (k = 0; k < 260 ; k++)
            {
                single_frame[k] = mdata[k];
            }
        getParameters(single_frame);
        for (i = 0; i < 8; i++)
            {
                output.pushShort(mLarParameters[i]);
            }
        donepushing = true;
    }
}
class PostProcessingFilter extends Filter
{
    short msr;
    short shortify(int a)
    {
        if (a >= 32767)
            {
                return 32767;
            }
        else
            {
                if (a <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) a;
                    }
            }
    }

    short gsm_add(short a, short b)

    {
        long ltmp = (long) a + (long) b;
        if (ltmp >= 32767)
            {
                return 32767;
            }
        else
            {
                if (ltmp <= -32768)
                    {
                        return -32768;
                    }
                else
                    {
                        return (short) ltmp;
                    }
            }
    }

    short gsm_sub(short a, short b)
    {
        long ltmp = (long) a - (long) b;
        if (ltmp >= 32767)
            {
                return 32767;
            }
        else
            {
                if (ltmp <= -32768)
```

```
                {
                    return -32768;
                }
            else
                {
                    return (short) ltmp;
                }
        }
}

short gsm_mult(short a, short b)
{
    long temp = (long) a * (long) b >> 15;
    if (temp >= 32767)
        {
            return 32767;
        }
    else
        {
            if (temp <= -32768)
                {
                    return -32768;
                }
            else
                {
                    return (short) temp;
                }
        }
}

short gsm_mult_r(short a, short b)
{
    long temp = ((long) a * (long) b) + 16384;
    short answer = (short) (temp >> 15);
    return answer;
}

short gsm_abs(short a)
{
    short answer;
    int temp;
    if (a < 0)
        {
            if (a == -32768)
                {
                    answer = 32767;
                }
            else
                {
                    temp = a * -1;
                    if (temp >= 32767)
                        {
                            answer = 32767;
                        }
                    else
                        {
                            if (temp <= -32768)
                                {
                                    answer = -32768;
                                }
                            else
                                {
                                    answer =
                                        (short) temp;
                                }
                        }
                }
        }
    else
        {
            answer = a;
        }
    return answer;
}
public void init()
{
    input = new Channel(Short.TYPE, 1);
    output = new Channel(Short.TYPE, 1);
    msr = 0;
}

public void work()
{
    int a;
    short i, k, temp;

    temp = input.popShort();


    temp = gsm_add(temp, gsm_mult_r(msr,
                                (short) 28180));
    msr = temp;
        temp = gsm_add(temp, temp);


        temp = shortify(temp / 8);
        temp = gsm_mult(temp, (short)8);

        output.pushShort(temp);
    }
}

class LTPInputFilter extends Filter
{

    short[] mdata;
    short[] single_frame;
    boolean donepushing;
    public short[] mLarParameters;
    public short[] mLtpOffset;
    public short[] mLtpGain;
    public short[] mRpeGridPosition;
    public short[] mRpeMagnitude;
    public short[] mSequence;

    public void initInputArrays() {
        mLarParameters = new short[8];
        mLtpOffset = new short[4];
        mLtpGain = new short[4];
        mRpeGridPosition = new short[4];
        mRpeMagnitude = new short[4];
        mSequence = new short[4*13];
    }

    public void getParameters(short[] input)
    {
        int i, j, k, l, m;
        int input_index = 0;
        int num_bits = 0;

        initInputArrays();
        for(i = 0; i < 8; i++)
            {

                switch(i)
                    {
                    case 0:
                    case 1:      num_bits = 6;
                        break;
                    case 2:
                    case 3:      num_bits = 5;
                        break;
                    case 4:
                    case 5:      num_bits = 4;
                        break;
                    case 6:
                    case 7:      num_bits = 3;
                        break;
                    }


                mLarParameters[i] = 0;
                for (j = 0; j < num_bits; j++,
                       input_index++)
                    {
                        mLarParameters[i] |=
                            input[input_index] <<
                            (num_bits - 1 - i);
                    }
            }


        for (k = 0; k < 4; k++)
            {
                mLtpOffset[k] = 0;
                for (l = 0; l < 7; l++)
                    {
                        mLtpOffset[k] |=
                            input[input_index] << (6 - l);
                        input_index++;
                    }
                mLtpGain[k] = 0;
                for (l = 0; l < 2; l++)
                    {
                        mLtpGain[k] |=
                            input[input_index] << (1 - l);
                        input_index++;
                    }
                mRpeGridPosition[k] = 0;
                for (l = 0; l < 2; l++)
                    {
                        mRpeGridPosition[k] |=
```

```
                        input[input_index] << (1 - l);
                    input_index++;
                }
            mRpeMagnitude[k] = 0;
            for (l = 0; l < 6; l++)
                {
                    mRpeMagnitude[k] |=
                        input[input_index] << (5 - l);
                    input_index++;
                }
            for(l = 0; l < 13; l++)
                {
                    mSequence[k+4*l] = 0;
                    for (m = 0; m < 3; m++)
                        {
                            mSequence[k+4*l] |=
                                input[input_index] <<
                                (2 - m);
                            input_index++;
                        }

                }
        }
    }

    public void init()
    {
        mdata = new short[260 ];
        single_frame = new short[260 ];
        input = new Channel(Short.TYPE, 260);
        output = new Channel(Short.TYPE, 8);
        donepushing = false;
    }

    public void work()
    {
        int i, j, k;

        for (i = 0; i < 260 ; i++)
            {
                mdata[i] = input.popShort();
            }

        if (donepushing)
            {}
        for (k = 0; k < 260 ; k++)
            {
                single_frame[k] = mdata[k];
            }
        getParameters(single_frame);
        for (i = 0; i < 4; i++)
            {
                output.pushShort(mLtpGain[i]);
                output.pushShort(mLtpOffset[i]);
            }
        donepushing = true;
    }
}
class LTPPipeline extends Pipeline
{
    public void init()
    {
        this.add(new FileReader
                 ("BinarySmallEndianDecoderInput1",
                  Short.TYPE));
        this.add(new LTPInputFilter());

    }
}
class LARPipeline extends Pipeline
{
    public void init()
    {
        this.add(new FileReader
                 ("BinarySmallEndianDecoderInput1",
                  Short.TYPE));
        this.add(new LARInputFilter());

    }
}

class LTPInputSplitJoin extends SplitJoin
{
    public void init()
    {
        this.setSplitter(WEIGHTED_ROUND_ROBIN (0, 1));
        this.add(new LTPPipeline());
        this.add(new Identity(Short.TYPE));

        this.setJoiner(WEIGHTED_ROUND_ROBIN(2, 160));
    }
}
class LTPLoopStream extends Pipeline
{
    public void init()
    {
        this.add(new LTPInputSplitJoin());

        this.add(new LTPFilter());

    }
}
class DecoderFeedback extends FeedbackLoop
{
    public void init()
    {
        this.setDelay(1);
        this.setJoiner(WEIGHTED_ROUND_ROBIN (40, 1));

        this.setBody(new StupidStream());
        this.setSplitter(DUPLICATE ());



        this.setLoop(new LTPLoopStream());
    }

    public short initPathShort(int index)
    {
        return 0;
    }

}


class StupidStream extends Pipeline
{
    public void init()
    {

        this.add(new AdditionUpdateFilter());

    }
}

class LARInputSplitJoin extends SplitJoin
{
    public void init()
    {
        this.setSplitter(WEIGHTED_ROUND_ROBIN (1, 0));
        this.add(new Identity(Short.TYPE));
        this.add(new LARPipeline());
        this.setJoiner(WEIGHTED_ROUND_ROBIN(160, 8));
    }
}

class RPEInputFilter extends Filter
{

    short[] mdata;
    short[] single_frame;
    boolean donepushing;
    public short[] mLarParameters;
    public short[] mLtpOffset;
    public short[] mLtpGain;
    public short[] mRpeGridPosition;
    public short[] mRpeMagnitude;
    public short[] mSequence;

    public void initInputArrays() {
        mLarParameters = new short[8];
        mLtpOffset = new short[4];
        mLtpGain = new short[4];
        mRpeGridPosition = new short[4];
        mRpeMagnitude = new short[4];
        mSequence = new short[4*13];
    }

    public void getParameters(short[] input)
    {
        int i, j, k, l, m;
        int input_index = 0;
        int num_bits = 0;

        initInputArrays();
        for(i = 0; i < 8; i++)
            {

                switch(i)
                    {
                    case 0:
                    case 1:     num_bits = 6;
                        break;
```

```
                case 2:
                case 3:      num_bits = 5;
                    break;
                case 4:
                case 5:      num_bits = 4;
                    break;
                case 6:
                case 7:      num_bits = 3;
                    break;
                }


            mLarParameters[i] = 0;
            for (j = 0; j < num_bits; j++,
                     input_index++)
                {
                    mLarParameters[i] |=
                        input[input_index] <<
                        (num_bits - 1 - i);
                }
        }


    for (k = 0; k < 4; k++)
        {
            mLtpOffset[k] = 0;
            for (l = 0; l < 7; l++)
                {
                    mLtpOffset[k] |=
                        input[input_index] << (6 - l);
                    input_index++;
                }
            mLtpGain[k] = 0;
            for (l = 0; l < 2; l++)
                {
                    mLtpGain[k] |=
                        input[input_index] << (1 - l);
                    input_index++;
                }
            mRpeGridPosition[k] = 0;
            for (l = 0; l < 2; l++)
                {
                    mRpeGridPosition[k] |=
                        input[input_index] << (1 - l);
                    input_index++;
                }
            mRpeMagnitude[k] = 0;
            for (l = 0; l < 6; l++)
                {
                    mRpeMagnitude[k] |=
                        input[input_index] << (5 - l);
                    input_index++;
                }
            for(l = 0; l < 13; l++)
                {
                    mSequence[k+4*l] = 0;
                    for (m = 0; m < 3; m++)
                        {
                            mSequence[k+4*l] |=
                                input[input_index] <<
                                (2 - m);
                            input_index++;
                        }

                }
        }



}

public void init()
{
    mdata = new short[260 ];
    single_frame = new short[260 ];
    input = new Channel(Short.TYPE, 260);
    output = new Channel(Short.TYPE, 60);
    donepushing = false;
}

public void work()
{
    int i, j, k, a;


    for (i = 0; i < 260 ; i++)
        {
            mdata[i] = input.popShort();
        }

    if (donepushing)
        {
```

```
            }


        for (k = 0; k < 260 ; k++)
            {
                single_frame[k] = mdata[k];
            }
        getParameters(single_frame);
        for (i = 0; i < 4; i++)
            {
                for (a = 0; a < 13; a++)
                    {
                        output.pushShort
                            (mSequence[i+4*a]);
                    }
                output.pushShort(mRpeMagnitude[i]);
                output.pushShort(mRpeGridPosition[i]);
            }
        donepushing = true;


    }
}

class HoldFilter extends Filter
{
    short[] mDrp;



    public void init()
    {
        input = new Channel(Short.TYPE, 160);
        output = new Channel(Short.TYPE, 40);
        mDrp = new short[160 ];
    }

    public void work()
    {
        int i, j;
        for (i = 0; i < 160 ; i++)
            {
                mDrp[i] = input.popShort();
            }
        for (j = 0; j < 40; j++)
            {
                output.pushShort(mDrp[j + 120]);
            }

    }

}

class ShortPrinter extends Filter
{
    char c;
    ShortPrinter (char c2)
    {
        super (c2);
    }
    public void init(char c2)
    {
        input = new Channel(Short.TYPE, 1);
        output = new Channel(Short.TYPE, 1);
        this.c = c2;
    }

    public void work()
    {
        short a = input.popShort();
        output.pushShort(a);
        System.out.println(c);
        System.out.println(a);
    }
}

public class Gsm extends StreamIt
{

    public static void main(String args[])
    {
        new Gsm().run(args);

    }

    public void init() {

        this.add(new FileReader
                ("BinarySmallEndianDecoderInput1",
                 Short.TYPE));
```

```
this.add(new RPEInputFilter());

this.add(new RPEDecodeFilter());
this.add(new DecoderFeedback());

this.add(new HoldFilter());
this.add(new LARInputSplitJoin());
this.add(new ReflectionCoeff());
this.add(new ShortTermSynth());
this.add(new PostProcessingFilter());

this.add(new FileWriter
        ("BinarySmallEndianDecoderOutput1",
         Short.TYPE));
    }
}
```

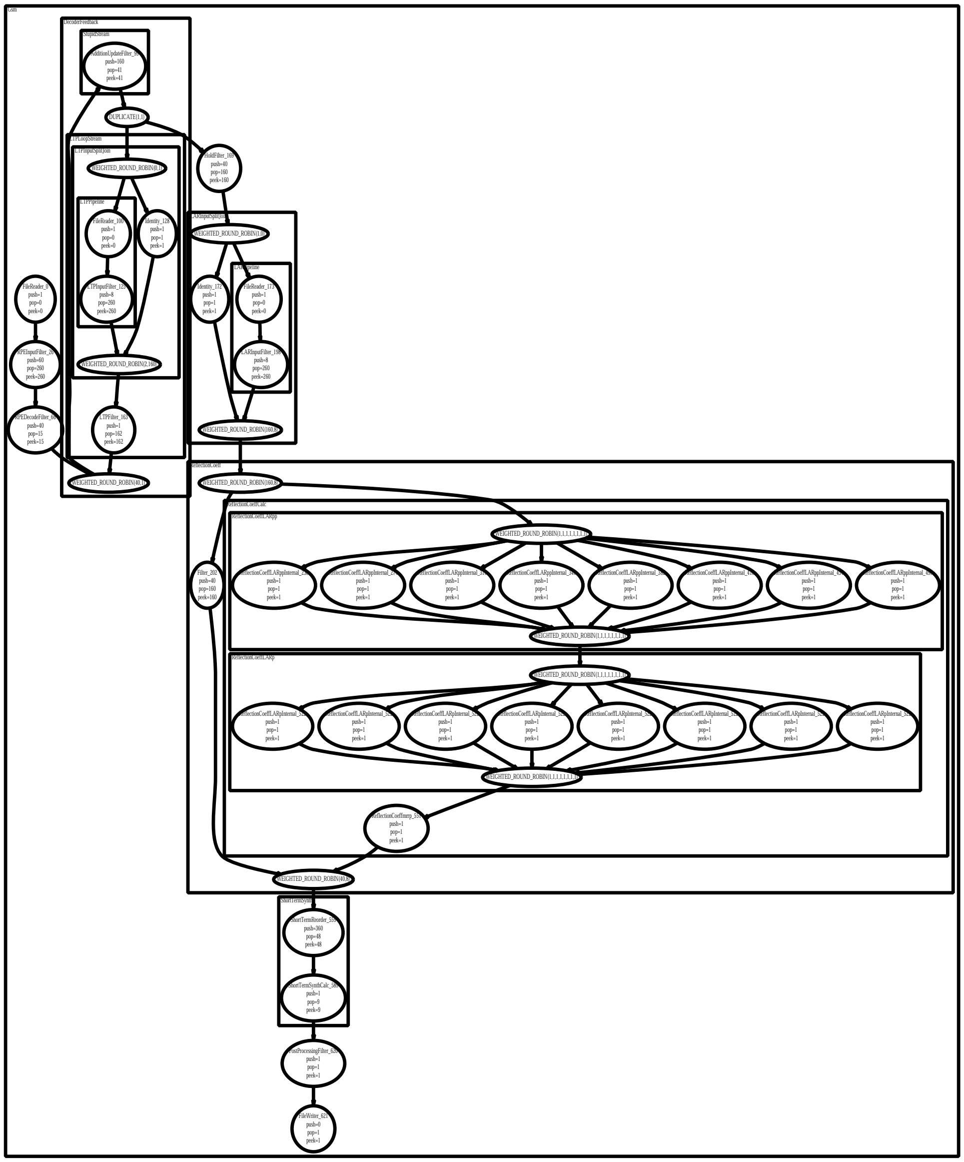


Figure G-1: GSM before partitioning.

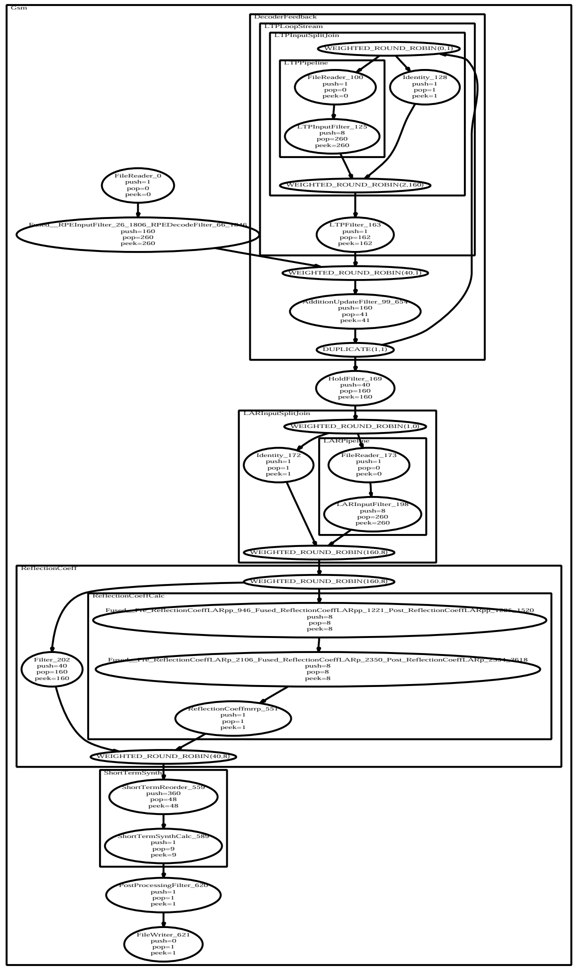


Figure G-2: GSM after partitioning.

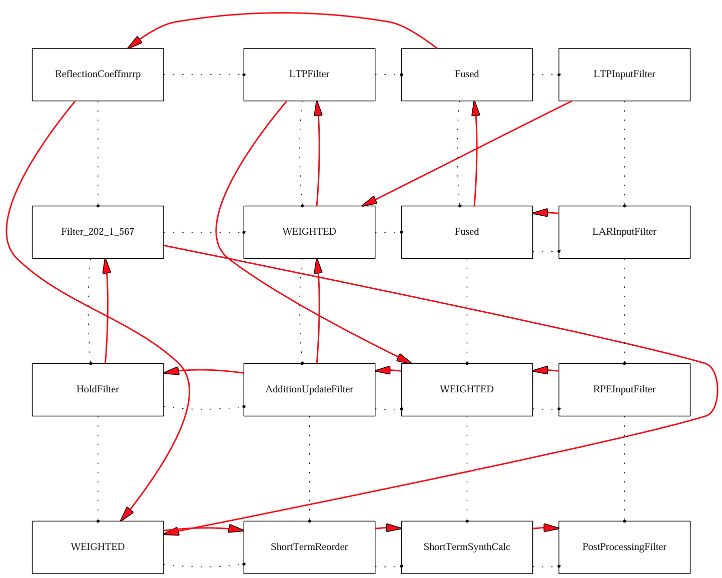


Figure G-3: GSM layout.

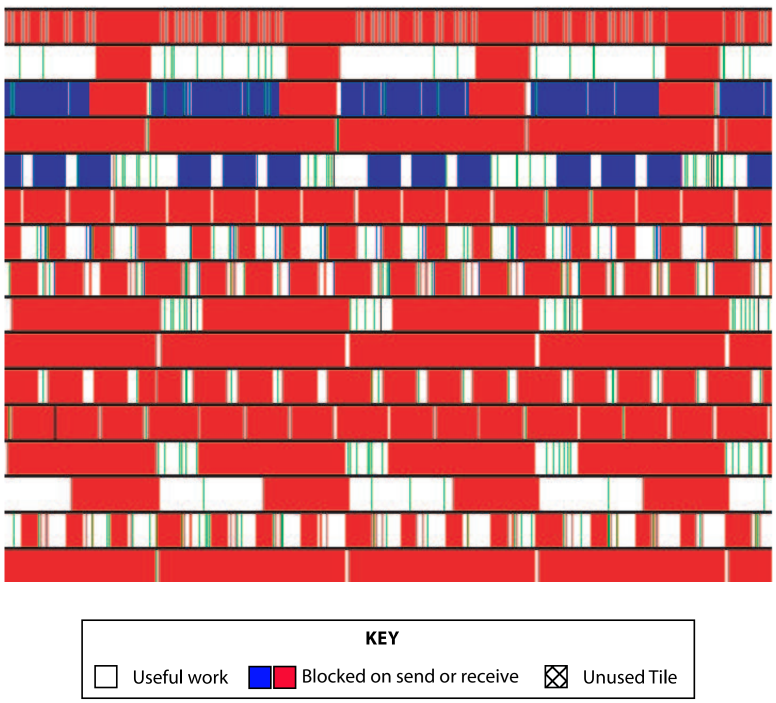


Figure G-4: GSM execution trace.

Appendix H

Vocoder Application

H.1 Description

This benchmark implements a 28-channel Phase Vocoder [37]. A vocoder is a program that implements S. Seneff's Speech Transformation System (Spectrum and/or Excitation) Without Pitch Extraction. The system implemented here is a method for independently modifying any or all of the pitch, formant frequencies, or speed of a wave file. Both the input and output files are wave files. The function of the system may be selected by modifying the constant parameters in the Constants Interface.

H.2 Code

```
import streamit.*;
import streamit.io.*;

class FIRSmoothingFilter extends Filter {
    int cosWinLength;
    int DFTLen;

    public void init(int DFTLen) {
        this.DFTLen = DFTLen;
        cosWinLength = 15;
        input = new Channel(Float.TYPE, DFTLen);
        output = new Channel(Float.TYPE, DFTLen);
    }

    public void work() {
        final int offset = (int) (cosWinLength / 2);
        final float cosWin[] = new float[cosWinLength];
        cosWin[0] = 0.1951f; cosWin[1] = 0.3827f; cosWin[2] =
                                                        0.5556f;
        cosWin[3] = 0.7071f; cosWin[4] = 0.8315f; cosWin[5] =
                                                        0.9239f;
        cosWin[6] = 0.9808f; cosWin[7] = 1.0000f; cosWin[8] =
                                                        0.9808f;
        cosWin[9] = 0.9239f; cosWin[10] = 0.8315f; cosWin[11] =
                                                        0.7071f;
        cosWin[12] = 0.5556f; cosWin[13] = 0.3827f; cosWin[14] =
                                                        0.1951f;

        for(int n=0; n < DFTLen; n++) {
            float y = 0;
            for(int k = 0; k < cosWinLength; k++) {
                //so that when i = 0, k will be at the center
                int i = k - offset;
                if (((n - i) >= 0) && ((n - i) < DFTLen))
                    y += input.peekFloat(n-i) * cosWin[k];
            }
            output.pushFloat(y);
        }

        for(int i=0; i < DFTLen; i++)
            input.popFloat();
    }

    FIRSmoothingFilter(int DFTLen) {
        super(DFTLen);
    }
}

class HanningWindow extends Filter {
    int length;

    public HanningWindow(int DFTLen) {
        super(DFTLen);
    }
    public void init(int DFTLen) {
        this.length = DFTLen;
        input = new Channel(Float.TYPE, 2 * DFTLen);
        output = new Channel(Float.TYPE, 2 * DFTLen);
    }

    public void work() {
        float real = 0;
        float imag = 0;
        //convolution with the series {-1/4, 1/2, -1/4}
        //first and last have to be dealt with specially
        /** Note that every index is doubled (real and imag) **/
        output.pushFloat((input.peekFloat(0) -
                          input.peekFloat(2))/2);
        output.pushFloat((input.peekFloat(1) -
                          input.peekFloat(3))/2);


        for(int i=1; i < length - 1; i++) {
            int n = i << 1;
            real = input.peekFloat(n)/2f;
            real -= (input.peekFloat(n-2)+input.peekFloat(n+2))/
                4f;
            output.pushFloat(real);
            imag = input.peekFloat(n+1)/2f;
            imag -= (input.peekFloat(n-1)+input.peekFloat(n+3))/
                4f;
            output.pushFloat(imag);
```

```
        }

        int n = (length - 1) * 2;
        output.pushFloat((input.peekFloat(n) -
                          input.peekFloat(n-2))/2);
        output.pushFloat((input.peekFloat(n+1) -
                          input.peekFloat(n-1))/2);

        for(int i=0; i < length; i++) {
            input.popFloat(); input.popFloat();
        }
    }
}


class Deconvolve extends Filter {
    public void init() {
        input = new Channel(Float.TYPE, 2);
        output = new Channel(Float.TYPE, 2);
    }

    public void work() {
        float den = input.popFloat();
        float num = input.popFloat();
        output.pushFloat(den);
        if (den == 0)
            output.pushFloat(0f);
        else
            output.pushFloat(num / den);
    }
}

/** DFTFilter expects that the first DFTLen numbers will all be 0.
 * Thus it can skip the initial calculation, and immediately
 * enter the  steady-state.  A more general DFTFilter that
 * can handle non-0 data values within the first DFTLen numbers
 * is below, known as DFTChannel.  The vocoder system assures
 * this requirement by adding a delay of DFTLen 0s to the front
 * of any data.  The system then does an inverse delay to get
 * rid of the (DFTLen/2 - 1) 0s th at precede the actual data.
 *
 **/
class DFTFilter extends Filter
{
    //the rate by which to deteriorate, assuring stability
    float deter;
    //since the previous complex value is multiplied by
    //the deter each
    //time, by the time the last time sample is windowed out it's
    //effect will have been multiplied by deter DFTLen times,
    //hence it needs to be multiplied by deter^DFTLen before
    //being subtracted
    float detern;
    //  float o[];
    int DFTLen;
    float range;
    private boolean first = true;
    private float prevR, prevI;
    private float nextR, nextI;
    private float wR, wI; //represents w^(-k)

    public void work() {
        float nextVal = (float) input.peekFloat(DFTLen);
        float current = (float) input.popFloat();


        prevR = prevR * deter + (nextVal - (detern * current));
        prevI = prevI * deter;
        nextR = prevR * wR - prevI * wI;
        nextI = prevR * wI + prevI * wR;
        prevR = nextR; prevI = nextI;

        output.pushFloat(prevR);
        output.pushFloat(prevI);
    }

    public void init(int DFTLength, float _range) {
        this.DFTLen = DFTLength;
        this.range = _range;
        this.deter = 0.999999f;
        this.detern = 1;
        wR = (float)Math.cos(_range);
        wI = (float)-Math.sin(_range);
        prevR = 0; prevI = 0;

        //need to peek DFTLen ahead of current one
        input = new Channel(Float.TYPE, 1, DFTLength+1);
        output = new Channel(Float.TYPE, 2);
    }

    public DFTFilter(int DFTLen, float range) {
        super(DFTLen, range);
    }
}

class FilterBank extends SplitJoin {
    public void init(final int channels) {
        setSplitter(DUPLICATE());

        for(int k=0; k <= channels/2; k++) {
            //this filter is for the kth range
            final float range = (float)
                ( 2 * 3.1415926535898f * k)/channels;
            add(new DFTFilter(channels,range));
        }

        //send real and imaginary parts together
        setJoiner(ROUND_ROBIN(2));
    }

    FilterBank(int channels) {
        super(channels);
    }

}

class DFTChannel extends Filter
{
    //the rate by which to deteriorate, assuring stability
    float deter;
    //since the previous complex value is multiplied by
    // the deter each time, by the time the last time sample
    // is windowed out it's effect will have been multiplied
    // by deter DFTLen times, hence it needs to be multiplied
    // by deter^DFTLen before being subtracted
    float detern;
    int DFTLen;
    float range;
    private boolean first = true;
    private float prevR, prevI;
    private float nextR, nextI;
    private float wR, wI; //represents w^(-k)

    public void work() {
        if (first) {
            first = false;
            //note: this w = w^k, not w^(-k)
            float wkR, wkI;
            wkR = (float)Math.cos(range);
            wkI = (float)Math.sin(range);
            float wkiR, wkiI; //this represents w^(k*i)
            float nwkiR, nwkiI;
            wkiR = 1f; wkiI = 0f;

            for (int i=0; i < DFTLen; i++) {
                float nextVal = (float) input.peekFloat(i);

                prevR = (prevR + wkiR * nextVal) * deter;
                prevI = (prevI + wkiI * nextVal) * deter;

                nwkiR = wkiR * wkR - wkiI * wkI;
                nwkiI = wkiR * wkI + wkiI * wkR;
                wkiR = nwkiR;
                wkiI = nwkiI;
                detern *= deter;
            }
        }

        float nextVal = (float) input.peekFloat(DFTLen);
        float current = (float) input.popFloat();

        prevR = prevR * deter + (nextVal - (detern * current));
        prevI = prevI * deter;// + (nextVal - (detern * current));

        nextR = prevR * wR - prevI * wI;
        nextI = prevR * wI + prevI * wR;
        prevR = nextR; prevI = nextI;

        output.pushFloat(prevR);
        output.pushFloat(prevI);
        System.out.println("range: " + range + " real: " +
                           prevR + " imag: " + prevI);
    }

    public void init(int DFTLength, float _range) {
        this.DFTLen = DFTLength;
        this.range = _range;
        this.deter = 0.999999f;
        this.detern = 1;
        wR = (float)Math.cos(_range);
        wI = (float)-Math.sin(_range);
        prevR = 0; prevI = 0;

        //need to peek DFTLen ahead of current one
```

```
        input = new Channel(Float.TYPE, 1, DFTLength+1);
        output = new Channel(Float.TYPE, 2);
    }

    public DFTChannel(int DFTLen, float range) {
        super(DFTLen, range);
    }
}
class TransformBank extends SplitJoin {
    public void init(final int channels, final int window) {
        setSplitter(DUPLICATE());

        for(int k=0; k < channels; k++) {
            //this filter is for the kth range
            final float range =
                (float)(2 * 3.1415926535898f * k)/channels;
            add(new DFTChannel(window ,range));
        }

        //send real and imaginary parts together
        setJoiner(ROUND_ROBIN(2));
    }

    TransformBank(int channels, int window) {
        super(channels,window);
    }

}

class SumReals extends SplitJoin {
    public SumReals(int DFT_LENGTH) {
        super(DFT_LENGTH);
    }
    public void init(final int DFT_LENGTH) {
        setSplitter(ROUND_ROBIN());
        add(new SumRealsRealHandler(DFT_LENGTH));
        add(new FloatVoid());
        setJoiner(WEIGHTED_ROUND_ROBIN(1,0));
    }
}

class SumRealsRealHandler extends Pipeline {
    public SumRealsRealHandler(int DFT_LENGTH) {
        super(DFT_LENGTH);
    }
    public void init(final int DFT_LENGTH) {
        add(new SplitJoin() {
                public void init() {
                    setSplitter
                        (WEIGHTED_ROUND_ROBIN
                         (1,DFT_LENGTH - 2, 1));
                    add(new Identity(Float.TYPE));
                    add(new Doubler());
                    //        add(new ConstMultiplier(2.0f));
                    add(new Identity(Float.TYPE));
                    setJoiner
                        (WEIGHTED_ROUND_ROBIN
                         (1, DFT_LENGTH - 2, 1));
                }
            });
        if (DFT_LENGTH % 2 != 0) {
            add(new Padder(DFT_LENGTH,0,1));
        }
        add(new SplitJoin() {
                public void init() {
                    setSplitter(ROUND_ROBIN());
                    add(new Adder((DFT_LENGTH + 1)/2));
                    add(new Adder((DFT_LENGTH + 1)/2));
                    setJoiner(ROUND_ROBIN());
                }
            });
        add(new Subtractor());
        add(new ConstMultiplier
            ((float)(1f / ((DFT_LENGTH - 1) * 2))));
    }
}


/**/
    class SumReals2 extends Filter {
        int length;
        public SumReals2(int length) {
            super(length);
        }

        public void init(int len) {
            this.length = len;
            input = new Channel(Float.TYPE,  2 * len);
            output = new Channel(Float.TYPE, 1);
        }

        public void work() {
            float sum = 0;
            int i=0;
            float first = input.popFloat(); input.popFloat();

            for(i=1; i < length - 1; i++) {
                if (i % 2 == 0)
                    sum += input.popFloat();
                else
                    sum -= input.popFloat();
                input.popFloat();
            }
            sum += sum; //double the internal ones
            sum += first;
            if (i % 2 == 0)
                sum += input.popFloat();
            else
                sum -= input.popFloat();
            input.popFloat();
            sum /= ((length - 1) * 2);
            output.pushFloat(sum);
        }

    }

class MagnitudeStuff extends Pipeline implements Constants {
    public void init(final int DFTLen,
                     final int newLen, final float speed) {
        if (DFTLen != newLen) {
            add(new SplitJoin() {
                    public void init() {
                        setSplitter(DUPLICATE());
                        add(new FIRSmoothingFilter(DFTLen));
                        add(new Identity(Float.TYPE));
                        setJoiner(ROUND_ROBIN());
                    }
                });
            add(new Deconvolve());
            add(new SplitJoin() {
                    public void init() {
                        setSplitter(ROUND_ROBIN());
                        add(new Duplicator
                            (DFT_LENGTH_REDUCED,
                             NEW_LENGTH_REDUCED));
                        add(new Remapper
                            (DFT_LENGTH_REDUCED,
                             NEW_LENGTH_REDUCED));
                        setJoiner(ROUND_ROBIN());
                    }
                });
            add(new Multiplier());
        }
        if (speed != 1.0) {
            add(new SplitJoin() {
                    public void init() {
                        setSplitter(ROUND_ROBIN());
                        for(int i=0; i < DFTLen; i++) {
                            add(new Remapper
                                (n_LENGTH,  m_LENGTH));
                        }
                        setJoiner(ROUND_ROBIN());
                    }
                });
        } else {
            add(new Identity(Float.TYPE));
        }
    }

    MagnitudeStuff(final int DFTLen, final int newLen,
                   final float speed) {
        super(DFTLen, newLen, speed);
    }
}

class Multiplier extends Filter {
    public void init() {
        input = new Channel(Float.TYPE, 2);
        output = new Channel(Float.TYPE, 1);
    }

    public void work() {
        output.pushFloat(input.popFloat() * input.popFloat());
    }
}

class ConstMultiplier extends Filter {
    float c;
    boolean first = true;

    public void init(float mult) {
        this.c = mult;
        input = new Channel(Float.TYPE, 1);
        output = new Channel(Float.TYPE, 1);
```

```
    }

    public void work() {
        output.pushFloat(input.popFloat() * c);
    }

    ConstMultiplier(float c) {
        super(c);
    }
}

class Accumulator extends Filter {
    float val = 0;
    public Accumulator() {}
    public void init() {
        input = new Channel(Float.TYPE, 1);
        output = new Channel(Float.TYPE, 1);
    }

    public void work() {
        val += input.popFloat();
        output.pushFloat(val);
    }
}

class Doubler extends Filter {
    public Doubler() {}
    public void init() {
        input = new Channel(Float.TYPE, 1);
        output = new Channel(Float.TYPE, 1);
    }

    public void work() {
        //you are within the work function of doubler
        output.pushFloat(input.peekFloat(0) + input.peekFloat(0));
        input.popFloat();
    }
}

class Summation extends Pipeline {
    public Summation(int length) {super(length);}
    public void init(final int length) {
        if (length == 1) {
            add(new Identity(Float.TYPE));
        } else {
            add(new SplitJoin() {
                    public void init() {
                        setSplitter(ROUND_ROBIN());
                        add(new Summation((length+1)/2));
                        add(new Summation(length/2));
                        setJoiner(ROUND_ROBIN());
                    }
                });
            add(new Adder(2));
        }
    }

}

class Adder extends Filter {
    int N;
    public Adder(int length) {
        super(length);
    }
    public void init(final int length) {
        N = length;
        input = new Channel(Float.TYPE, length);
        output = new Channel(Float.TYPE, 1);
    }
    public void work() {
        float val = 0;
        for(int i=0; i < N; i++)
            val += input.popFloat();
        output.pushFloat(val);
    }
}

class Subtractor extends Filter {
    public void init() {
        input = new Channel(Float.TYPE, 2);
        output = new Channel(Float.TYPE, 1);
    }

    public void work() {
        output.pushFloat(input.peekFloat(0) - input.peekFloat(1));
        input.popFloat();input.popFloat();
    }
}

class PhaseUnwrapper extends Filter {
    float estimate, previous;

    public void init() {
        input = new Channel(Float.TYPE, 1);
        output = new Channel(Float.TYPE, 1);
        previous = 0f;
        estimate = 0f;
    }

    public void work() {
        float unwrapped = input.popFloat();
        unwrapped += estimate;
        float delta = unwrapped - previous;
        while (delta > 2 * Math.PI * (11f / 16f)) {
            unwrapped -= 2 * Math.PI;
            delta -= 2 * Math.PI;
            estimate -= 2 * Math.PI;
        }
        while (delta < -2 * Math.PI * (11f / 16f)) {
            unwrapped += 2 * Math.PI;
            delta += 2 * Math.PI;
            estimate += 2 * Math.PI;
        }
        previous = unwrapped;
        output.pushFloat(unwrapped);
    }

    PhaseUnwrapper() {
        super();
    }
}

/**/
class FirstDifference extends Filter {
    private float prev;

    public FirstDifference() {
        super();
    }

    public void init() {
        input = new Channel(Float.TYPE, 1, 1);
        output = new Channel(Float.TYPE, 1);
        prev = 0f;
    }

    public void work() {
        output.pushFloat(input.peekFloat(0) - prev);
        prev = input.popFloat();
    }
}

class InnerPhaseStuff extends Pipeline implements Constants {

    public void init(float c, float speed) {
        add(new PhaseUnwrapper());
        add(new FirstDifference());
        if (c != 1.0) {
            add(new ConstMultiplier(c));
        }
        if (speed != 1.0) {
            add(new Remapper(n_LENGTH, m_LENGTH));
        }
        add(new Accumulator());
    }
    public InnerPhaseStuff(float c, float speed) {
        super(c, speed);
    }
}

class PhaseStuff extends Pipeline implements Constants {

    public void init(final int DFTLen,
                     final int newLen, final float c,
                     final float speed) {

        if (speed != 1.0 || c != 1.0) {
            add(new SplitJoin() {
                    public void init() {
                        setSplitter(ROUND_ROBIN());
                        for(int i=0; i < DFTLen; i++) {
                            add(new InnerPhaseStuff(c, speed));
                        }
                        setJoiner(ROUND_ROBIN());
                    }
                });
            if (newLen != DFTLen) {
                add(new Duplicator
                    (DFT_LENGTH_REDUCED, NEW_LENGTH_REDUCED));
            }
        } else {
            add(new Identity(Float.TYPE));
        }
    }
```

```java
    PhaseStuff(int DFTLen, int newLen, float c, float speed) {
        super(DFTLen, newLen, c, speed);
    }
}

/** Linear Interpolater just takes two
    neighbouring points and creates
 * <interp - 1> points linearly between the two **/
class LinearInterpolator extends Filter {
    int interp;

    public void init(int interpFactor) {
        this.interp = interpFactor;
        input = new Channel(Float.TYPE, 1,2);
        output = new Channel(Float.TYPE, interpFactor);
    }

    public void work() {
        float base = input.popFloat();
        float diff = input.peekFloat(0) - base;
        final int goal = interp;

        output.pushFloat(base);
        //already pushed 1, so just push another (interp - 1)
        //floats
        for(int i = 1; i < goal; i++)
            output.pushFloat(base + ((float) i / interp) * diff);
    }

    LinearInterpolator(int interp) {
        super(interp);
    }
}

/** Linear Decimator just passes on one point and pops the next
 * decimFactor - 1 **/
class Decimator extends Filter {
    int decim;

    public void init(int decimFactor) {
        this.decim = decimFactor;
        input = new Channel(Float.TYPE, decimFactor);
        output = new Channel(Float.TYPE, 1);
    }

    public void work() {
        output.pushFloat(input.popFloat());
        //already popped 1, so just pop another (interp - 1)
        //floats
        for(int goal = decim - 1; goal > 0; goal--)
            input.popFloat();
    }

    Decimator(int decim) {
        super(decim);
    }
}

/** Remapper is a combination interpolator/decimator.
 * It's goal is to
 * map one stream from size n (oldLen) to size m (newLen).
 * To do this, it calculates [c = gcd(m,n)], interpolates
 * linearly by m/c, and then decimates by n/c.
 **/
class Remapper extends Pipeline {

    public void init(int oldLen, int newLen) {

        if (newLen == oldLen) {
            add(new Identity(Float.TYPE));
        } else {
            if (newLen != 1)
                add(new LinearInterpolator(newLen));
            if (oldLen != 1)
                add(new Decimator(oldLen));
        }
    }

    int gcd(int a, int b) {
        return (b == 0) ? a : gcd(b, a % b);
    }

    Remapper(int oldLen, int newLen) {
        super(oldLen, newLen);
    }
}

class Duplicator extends Filter {
    int oldLen, newLen;

    Duplicator(int oldLen, int newLen) {
        super(oldLen, newLen);
    }

    public void init(int oldLength, int newLength) {
        this.oldLen = oldLength;
        this.newLen = newLength;
        output = new Channel(Float.TYPE, newLength);
        input = new Channel(Float.TYPE, oldLength);
    }

    public void work() {
        if (newLen <= oldLen) {
            int i;
            for(i=0; i < newLen; i++)
                output.pushFloat(input.popFloat());
            for(i = newLen; i < oldLen; i++) {
                input.popFloat();
            }
        } else {
            float orig[] = new float[oldLen];
            for(int i=0; i < oldLen; i++)
                orig[i] = input.popFloat();
            for(int i=0; i < newLen; i++)
                output.pushFloat(orig[i%oldLen]);
        }
    }
}

class RandSource extends Filter
{
    final int length = 6;
    int i = 0;
    int x = 0;

    public void work() {
        output.pushInt(x);
        if (i < length)
            x = 2 * x + 1;
        else
            x = (x - 1) / 2;

        i++;
        if (i == (length * 2)) {
            x = 1; i = 0;
        }
    }
    public void init() {
        output = new Channel(Integer.TYPE, 1);
    }
    public RandSource() {
        super();
    }
}

class PlateauSource extends Filter
{
    int length = 6;
    int i = 0;
    int x = 0;
    int up = 1;

    public void work() {
        output.pushInt(x);
        if (i == length) {
            i = 0;
            if (up == 1) {
                x++;
            } else {
                x--;
            }
            if (x == length) {
                up = -1; i = 0;
            }
            if (x == 0) {
                up = 1;
            }
        } else {
            i++;
        }

    }
    public void init(int length) {
        this.length = length;
        output = new Channel(Integer.TYPE, 1);
    }
    public PlateauSource(int length) {
        super(length);
    }
}

class StepSource extends Filter
{
```

```
    int x, length;
    int up;

    public void work() {
        output.pushInt(x);
        if (x == length) { up = 0;} else if (x == 0) { up = 1; }
        if (up == 1) {
            x += 1;
        } else {
            x -= 1;
        }
    }
    public void init(int len) {
        this.length = len;
        this.up = 1;
        this.x = 0;
        output = new Channel(Integer.TYPE, 1);
    }
    public StepSource(int length) {
        super(length);
    }
}

class AddSource extends Filter
{
    float x, length;

    public void work() {
        output.pushFloat(x);
        x += length;
    }
    public void init(float len) {
        this.length = len;
        output = new Channel(Float.TYPE, 1);
    }
    public AddSource(float len) {
        super(len);
    }
}

class ModularFilter extends Filter {
    float mod;
    public void work() {
        output.pushFloat(input.popFloat() % mod);
    }
    public void init(float mod) {
        this.mod = mod;
        output = new Channel(Float.TYPE, 1);
        input = new Channel(Float.TYPE, 1);
    }
    public ModularFilter(float mod) {
        super(mod);
    }
}

class FunkySource extends Pipeline {
    public void init(final float c) {
        add(new SplitJoin() {
                public void init() {
                    add(new SineSource(c * 4f));
                    add(new AddSource(c));
                    setJoiner(ROUND_ROBIN());
                }
            });
        add(new Filter() {
                public void init() {
                    output = new Channel(Float.TYPE, 1);
                    input = new Channel(Float.TYPE, 2);
                }
                public void work() {
                    output.pushFloat(input.popFloat() +
                                     input.popFloat());
                }
            });
    }
    public FunkySource(float c) {
        super(c);
    }
}

class SineSource extends Filter
{
    float theta, prev;

    public void work() {
        prev = prev + theta;
        output.pushFloat((float)Math.sin(prev));
    }
    public void init(float theta) {
        this.theta = theta; this.prev = theta * 3.2f;
        output = new Channel(Float.TYPE, 1);
    }
    public SineSource(float theta) {
        super(theta);
    }
}

class SineFilter extends Filter
{
    float theta;

    public void work() {
        output.pushFloat((float)Math.sin(theta*input.popFloat()));
    }
    public void init(float theta) {
        this.theta = theta;
        input = new Channel(Float.TYPE, 1);
        output = new Channel(Float.TYPE, 1);
    }
    public SineFilter(float theta) {
        super(theta);
    }
}

class WaveReader extends Filter
{
    boolean first = true;
    short channels, bitsPerSample;
    int size;
    public WaveReader() {
    }

    public void init() {
        input = new Channel(Short.TYPE, 22);
        output = new Channel(Short.TYPE, 22);
    }

    short next() {
        short current = input.popShort();
        output.pushShort(current);
        return current;
    }

    int   nextInt() {
        return (int) (next() &0xffff) | (next() << 16);
    }

    char[] nextId(char[] id) {
        for(int i=0; i < id.length / 2; i++) {
            short current = next();
            id[(i << 1)] = (char) (current & 0x7F);
            id[(i << 1) + 1] = (char) (current >> 8);
        }
        return id;
    }

    public void work() {
        if (first) {
            first = false;
            char GROUP_ID[] = new char[4];
            char RIFF_TYPE[] = new char[4];
            char CHUNK_ID[] = new char[4];
            int  chunkSize;

            nextId(GROUP_ID);
            chunkSize = nextInt();
            System.out.println(String.valueOf(GROUP_ID));
            if (!String.valueOf(GROUP_ID).
                equalsIgnoreCase("RIFF"))
                System.exit(-1);
            System.out.println("Size: " +
                               Integer.toHexString(chunkSize));

            nextId(RIFF_TYPE);
            System.out.println(String.valueOf(RIFF_TYPE));
            if (!String.valueOf(RIFF_TYPE).
                equalsIgnoreCase("WAVE"))
                System.exit(-1);

            nextId(CHUNK_ID);

            while (!String.valueOf(CHUNK_ID).
                   equalsIgnoreCase("fmt ")) {
                chunkSize = nextInt();
                for(int i=0; i < chunkSize / 2; i++)
                    next();
                nextId(CHUNK_ID);
            }

            chunkSize = nextInt();
            if (next() != 1 || chunkSize != 16) {
                System.err.println
                    ("Error: Cannot handle compressed WAVE");
                System.exit(-1);
```

```
        }

        channels = next();
        System.out.println("Channels: " + channels);
        int dataRate = nextInt();
        System.out.println("Data Rate: " + dataRate + "hz");
        int avgBytesPerSec = nextInt();
        short blockAlign = next();
        short bitsPerSample = next();
        System.out.println(bitsPerSample + " * " +
                           channels + " = " +
                           blockAlign * 8);
        if (bitsPerSample != 16) {
            System.err.println
                ("Error: Can only handle 16 bit samples (" +
                      bitsPerSample + " bits)");
            System.exit(-1);
        }

        nextId(CHUNK_ID);

        while (!String.valueOf(CHUNK_ID).
               equalsIgnoreCase("data")) {
            chunkSize = nextInt();
            for(int i=0; i < chunkSize / 2; i++)
                next();
            nextId(CHUNK_ID);
        }
        chunkSize = nextInt();
        System.out.println("Size: " +
                           Integer.toHexString(chunkSize));
    } else {
        for (int i=0; i < 22; i++)
            next();
    }
  }
}

class WaveHeader extends Filter {
    float speed, freq;

    public WaveHeader(float speed, float freq) {
        super(speed,freq);}
    public void  init(float speed, float freq) {
        this.speed = speed; this.freq = freq;
        input = new Channel(Short.TYPE, 22);
        output = new Channel(Short.TYPE, 22);
    }
    short next() {
        return input.popShort();
    }
    void pass() {
        send(next());
    }
    void send(short s) {
        output.pushShort(s);
    }
    void sendInt(int i) {
        send((short) (i & 0xffff));
        send((short) (i >> 16));
    }
    int nextInt() {
        return (int) (next() &0xffff) | (next() << 16);
    }


    public void work() {
        /** Structure is: "RIFF" file_length "WAVE" "fmt "
         * chunk_length compression channels sample_rate data_rate
         *       bytes_per_frame bits_per_sample "data" length
         **/
        pass(); pass(); //"RIFF"
        int file_length = nextInt();
        //file_length is data chunk + 36 bytes of header info
        file_length = (int) Math.round((file_length - 36)* speed)
            + 36;
        sendInt(file_length);
        pass(); pass(); //"WAVE"
        pass(); pass(); //"fmt "
        pass(); pass(); //fmt chunk_length; must be 16
        pass(); //compression; must be 1
        pass(); //channels; for now, assuming 2 channels
        pass(); pass(); //sample_rate; don't care about it
        pass(); pass(); //data_rate; should be same
        //same bytes_per_frame and bits_per_sample (16)
        pass(); pass();
        pass(); pass(); //"data"
        int samples = nextInt();
        samples = (int) Math.round(samples * speed);
        sendInt(samples);
    }
}
```

```
class WaveSplitter extends Filter
{
    boolean first = true;
    short channels, bitsPerSample;
    int size;
    int channel, current;
    public WaveSplitter(int c) {
        super(c);
    }

    public void init(int c) {
        this.channel = c; this.current = 0;
        input = new Channel(Short.TYPE, 1);
        output = new Channel(Short.TYPE, 1);
    }

    short next() {
        short temp = input.popShort();
        output.pushShort(temp);
        return temp;
    }

    int   nextInt() {
        return (int) (next() &0xffff) | (next() << 16);
    }

    char[] nextId(char[] id) {
        for(int i=0; i < id.length / 2; i++) {
            short current = next();
            id[(i << 1)] = (char) (current & 0x7F);
            id[(i << 1) + 1] = (char) (current >> 8);
        }
        return id;
    }

    public void work() {
        if (first) {
            first = false;
            char GROUP_ID[] = new char[4];
            char RIFF_TYPE[] = new char[4];
            char CHUNK_ID[] = new char[4];
            int  chunkSize;

            nextId(GROUP_ID);
            chunkSize = nextInt();
            System.out.println(String.valueOf(GROUP_ID));
            if (!String.valueOf(GROUP_ID).
                equalsIgnoreCase("RIFF"))
                System.exit(-1);
            System.out.println("Size: " +
                               Integer.toHexString(chunkSize));

            nextId(RIFF_TYPE);
            System.out.println(String.valueOf(RIFF_TYPE));
            if (!String.valueOf(RIFF_TYPE).
                equalsIgnoreCase("WAVE"))
                System.exit(-1);

            nextId(CHUNK_ID);

            while (!String.valueOf(CHUNK_ID).
                   equalsIgnoreCase("fmt ")) {
                chunkSize = nextInt();
                for(int i=0; i < chunkSize / 2; i++)
                    next();
                nextId(CHUNK_ID);
            }

            chunkSize = nextInt();
            if (next() != 1 || chunkSize != 16) {
                System.err.println
                    ("Error: Cannot handle compressed");
                System.exit(-1);
            }

            channels = input.popShort();
            output.pushShort((short)1);
            System.out.println("Channels: " + channels);
            int dataRate = nextInt();
            System.out.println("Data Rate: " + dataRate + "hz");
            int avgBytesPerSec = nextInt();
            short blockAlign = input.popShort();
            output.pushShort((short)(blockAlign / channels));
            short bitsPerSample = next();
            System.out.println(bitsPerSample + " * " +
                               channels + " = " +
                               blockAlign * 8);
            if (bitsPerSample != 16) {
                System.err.println
                    ("Error: Can only handle 16 bit samples (" +
                          bitsPerSample + " bits)");
```

```java
                System.exit(-1);
            }

            nextId(CHUNK_ID);

            while (!String.valueOf(CHUNK_ID).
                   equalsIgnoreCase("data")) {
                chunkSize = nextInt();
                for(int i=0; i < chunkSize / 2; i++)
                    next();
                nextId(CHUNK_ID);
            }
            chunkSize = nextInt();
            System.out.println("Size: " +
                               Integer.toHexString(chunkSize));
        }
        if (current % channels == channel) {
            next();
            current++;
        }
        else {
            current++;
            input.popShort();
        }

    }

}

class Delay extends Filter {
    float delay[];
    int length;

    public Delay(int N) {
        super(N);
    }

    public void init(int N) {
        delay = new float[N];
        for(int i=0; i < N; i++)
            delay[i] = 0;
        length = N;
        input = new Channel(Float.TYPE, 1);
        output = new Channel(Float.TYPE, 1);
    }

    public void work() {
        output.pushFloat(delay[0]);
        for(int i=0; i < length - 1; i++)
            delay[i] = delay[i+1];
        delay[length - 1] = input.popFloat();
    }
}

class FrontPadder extends Filter {
    int length, padding;

    public FrontPadder(int N, int i) {
        super(N, i);
    }

    public void init(int N, int i) {
        length = N;
        padding = i;
        input = new Channel(Float.TYPE, N);
        output = new Channel(Float.TYPE, N+i);
    }

    public void work() {
        for(int i=0;i < padding; i++)
            output.pushFloat(0f);

        for(int i=0; i < length; i++)
            output.pushFloat(input.popFloat());
    }
}

class Padder extends Filter {
    int length, front, back;

    public Padder(int N, int i, int j) {
        super(N, i, j);
    }

    public void init(int N, int i, int j) {
        length = N;
        front = i;
        back = j;
        input = new Channel(Float.TYPE, N);
        output = new Channel(Float.TYPE, N+i+j);
    }

    public void work() {
        for(int i=0;i < front; i++)
            output.pushFloat(0f);

        for(int i=0; i < length; i++)
            output.pushFloat(input.popFloat());

        for(int i=0; i < back; i++)
            output.pushFloat(0);
    }
}

class InvDelay extends Filter {
    float delay[];
    int length;

    public InvDelay(int N) {
        super(N);
    }

    public void init(int N) {
        delay = new float[N];
        length = N;
        input = new Channel(Float.TYPE, 1,N+1);
        output = new Channel(Float.TYPE, 1);
    }

    public void work() {
        output.pushFloat(input.peekFloat(length));
        input.popFloat();
    }
}
/** RecToPolar **/
class RectangularToPolar extends Filter {
    public void init() {
        input = new Channel(Float.TYPE, 2);
        output = new Channel(Float.TYPE, 2);
    }

    public void work() {
        float x, y;
        float r, theta;
        x = input.popFloat(); y = input.popFloat();

        r = (float)Math.sqrt(x * x + y * y);
        theta = (float)Math.atan2(y, x);

        output.pushFloat(r);
        output.pushFloat(theta);
    }
}

class PolarToRectangular extends Filter {
    public void init() {
        input = new Channel(Float.TYPE, 2);
        output = new Channel(Float.TYPE, 2);
    }

    public void work() {
        float r, theta;
        r = input.popFloat(); theta = input.popFloat();

        output.pushFloat((float)(r * Math.cos(theta)));
        output.pushFloat((float)(r * Math.sin(theta)));
    }
}

class IntToFloat extends Filter {
    public void init() {
        input = new Channel(Integer.TYPE, 1);
        output = new Channel(Float.TYPE, 1);
    }
    public void work() {
        output.pushFloat((float)input.popInt());
    }
}

class IntToDouble extends Filter {
    public void init() {
        input = new Channel(Integer.TYPE, 1);
        output = new Channel(Double.TYPE, 1);
    }
    public void work() {
        output.pushDouble(input.popInt());
    }
}

class ShortToDouble extends Filter {
    public void init() {
        input = new Channel(Short.TYPE, 1);
        output = new Channel(Double.TYPE, 1);
    }
```

```
    public void work() {
        output.pushDouble(input.popShort());
    }
}

class DoubleToShort extends Filter {
    public void init() {
        input = new Channel(Double.TYPE, 1);
        output = new Channel(Short.TYPE, 1);
    }
    public void work() {
        output.pushShort((short) (input.popDouble() + 0.5));
    }
}

class Timer extends Filter {
    private int count, length, num;
    private long lastTime;

    public void init(int N) {
        this.length = N;
        this.count = 0;
        this.num = 0;
        input = new Channel(Short.TYPE, 1);
        output = new Channel(Short.TYPE, 1);
    }

    public void work() {
        output.pushShort(input.popShort());
        count++;
        if (count == length) {
            count = 0;
            System.out.println(num++);
        }
    }

    public Timer(int N) {
        super(N);
    }
}

class CountDown extends Filter {
    private int length, count;

    CountDown(int length) {
        super(length);
    }
    public void init(int len) {
        this.length = len;
        this.count = len;
        input = new Channel(Float.TYPE, 1);
    }
    public void work() {
        count--;
        input.popFloat();
        if (count == 0) {
            count = length;
            System.out.println("done");
        }
    }
}

class IntPrinter extends Filter {
    int x;
    public void work() { int i = input.popInt();
    System.out.print(x+++" ");
    System.out.println(i);
    output.pushInt(i);
    }
    public void init() {
        x = 0;
        input = new Channel(Integer.TYPE, 1);
        output = new Channel(Integer.TYPE, 1);
    }
    IntPrinter() {}
}

class ComplexPrinter extends Filter {
    int real,imag;
    int N;
    public void work() { float f = input.popFloat();
    System.out.print((real++ * 2 * Math.PI /N)+" ");
    System.out.println(f);
    output.pushFloat(f);
    f = input.popFloat();
    System.err.print((imag++ * 2 * Math.PI /N)+" ");
    System.err.println(f);
    output.pushFloat(f);
    if (real == N) {
        real = 0;
        imag = 0;
    }
    }

    public void init(int length) {
        this.N = length;
        real= 0;
        input = new Channel(Float.TYPE, 2);
        imag = 0;
        output = new Channel(Float.TYPE, 2);
    }
    public ComplexPrinter(int length) {
        super(length);
    }
}

class ShortPrinter extends Filter {
    public void work() { short i = input.popShort();
    System.out.println(i); output.pushShort(i);}
    public void init() { input = new Channel(Short.TYPE, 1);
    output = new Channel(Short.TYPE, 1);}
    ShortPrinter() {}
}
class DoublePrinter extends Filter {
    public void work() { double i = input.popDouble();
    System.out.println(i); output.pushDouble(i);}
    public void init() { input = new Channel(Double.TYPE, 1);
    output = new Channel(Double.TYPE, 1);}
    DoublePrinter() {}
}
class FloatPrinter extends Filter {

    public void work() { float i = input.popFloat();
    System.out.println(i);
    output.pushFloat(i);}
    public void init() { input = new Channel(Float.TYPE, 1);
    output = new Channel(Float.TYPE, 1);}
    FloatPrinter() {}
}

class FloatVoid extends Filter {
    public void work() {input.popFloat();}
    public void init() {input = new Channel(Float.TYPE, 1); }
}

class ShortVoid extends Filter {
    public void work() {input.popShort();}
    public void init() {input = new Channel(Short.TYPE, 1); }
}

class IntVoid extends Filter {
    public void work() {input.popInt();}
    public void init() {input = new Channel(Integer.TYPE, 1); }
}

class FloatToShort extends Filter {
    public void work() {
        output.pushShort((short)
                         (input.popFloat() + 0.5f));
    }
    public void init() {input = new Channel(Float.TYPE, 1);
    output = new Channel(Short.TYPE, 1);}
}

class FloatToInt extends Filter {
    public void work() {
        output.pushInt((int) (input.popFloat() + 0.5f)); }
    public void init() {input = new Channel(Float.TYPE, 1);
    output = new Channel(Integer.TYPE, 1);}
}

class ShortToFloat extends Filter {
    public void work() {short i = input.popShort();
    float f = (float) i;
    output.pushFloat(f); }
    public void init() {input = new Channel(Short.TYPE, 1);
    output = new Channel(Float.TYPE, 1);}
}

class VocoderSystem extends SplitJoin
{
    public void init(int DFTLen, int newLen, float c,
                     float speed) {
        setSplitter(ROUND_ROBIN());

        add(new MagnitudeStuff(DFTLen, newLen, speed));
        add(new PhaseStuff(DFTLen, newLen, c, speed));

        setJoiner(ROUND_ROBIN());
    }

    VocoderSystem(int DFTLen, int newLen, float c, float speed) {
        super(DFTLen, newLen, c, speed);
    }
```

```
}

interface Constants {
    //For this system, DFT_LENGTH_NOM is the nominal number of DFT
    //coefficients to use when taking the DFT.
    //Thus the behaviour of
    //the system is that there are DFT_LENGTH_NOM filters between
    //[0, 2 * pi).
    //This code assumes that the DFT_LENGTH_NOM
    //is even, so that the
    //range (pi, 2 * pi) is just a reflection
    //of the range (0, pi).
    //This is because the input signal is real and discrete;
    //discreteness means the fourier
    //transform is periodic with 2 * pi,
    //and since the signal is real the
    // magnitude of the DFT will be even
    //and the phase odd.  Since we only
    //care about the real output of
    //the system, and are only doing the
    //inverse DFT for a single sample
    //at the center of the window, the phase
    //being odd makes no
    //difference.  Thus with filters in the range
    //[0, pi], the entire
    //fourier transform can be represented, thus
    //using approximately
    //half the filters and computation.

    /** DFT_LENGTH_NOM numbers:
     *
     *  4: can tell when someone is talking, but not recognize
     *           that it's a voice unless you already know
     *
     *  8: can tell that it's a person talking, if you already
     *           know the script, you can follow the voice
     *
     *  16: can tell that it's a person, can understand the words,
     *    can kind of see that the vocoder is doing something
     *           that may be appropriate
     *
     *  32: better output; less grainy, more believable
     *  64: still better output
     *
     *  128: probably the high-point of good output
     *      vs. computation * and size.  With 128, it'll tradeof
     *           quality in output for * time.
     **/

    public static final int DFT_LENGTH_NOM = 28; //
    public static final int DFT_LENGTH = DFT_LENGTH_NOM/2+1; //
    public static final float FREQUENCY_FACTOR_ARGS[] =
    {1f, 1f, 1f, 1.8f, 0.6f};
    public static final float FREQUENCY_FACTOR = 0.6f;
    public static final float GLOTTAL_EXPANSION_ARGS[] =
    {1f, 1f, 1f, 1.2f, 1/1.2f};
    public static final float GLOTTAL_EXPANSION = 1/1.2f;
    public static final int NEW_LENGTH =
        (int) (DFT_LENGTH * GLOTTAL_EXPANSION / FREQUENCY_FACTOR);


    //DFT_LENGTH_RED and NEW_LENGTH_RED
    //correspond to the reduced ratio
    //of DFT_LENGTH to NEW_LENGTH.  This
    //ratio is needed to avoid
    //interpolating and then decimating by
    //redundant amounts.  Normally
    //these numbers could be calculated by
    //taking the GCD of DFT_L and
    //NEW_L, but the loop unroller is having
    //trouble with it, so they
    //currently need to be set by hand.

    //NOTE: it's very important that NEW_LENGTH_REDUCED *
    // DFT_LENGTH is
    //a multiple of DFT_LENGTH_REDUCED.
    //Otherwise the decimation will
    //not finish completely each window,
    //and the windows will no longer
    //be distinct.
    public static final int DFT_LENGTH_REDUCED_ARGS[] =
    {1,1,1,3,3};
    public static final int DFT_LENGTH_REDUCED = 3;
    public static final int NEW_LENGTH_REDUCED_ARGS[] =
    {1,1,1,2,4};
    public static final int NEW_LENGTH_REDUCED = 4;
    public static final float SPEED_FACTOR_ARGS[] =
    {1f, 2f, 0.5f, 1f, 1f};
    public static final float SPEED_FACTOR = 1f;

    //n_LENGTH and m_LENGTH are similar to DFT_LENGTH_REDUCED and
    //NEW_LENGHT_REDUCED above.
    //The difference is that these should be
    //the reduced ratio of SPEED_FACTOR.  So if SPEED_FACTOR is 2,
    //m_LENGTH should be 2, and n_LENGTH should be 1.
    //If SPEED_FACTOR
    //is 2.5, m_LENGTH should be 5, and n_LENGTH should be 2.
    public static final int n_LENGTH_ARGS[] = {1,1,2,1,1};
    public static final int n_LENGTH = 1;
    public static final int m_LENGTH_ARGS[] = {1,2,1,1,1};
    public static final int m_LENGTH = 1;

}

class Vocoder extends Pipeline implements Constants {


    public void init() {
        add(new FilterBank(DFT_LENGTH_NOM));
        add(new RectangularToPolar());
        add(new VocoderSystem(DFT_LENGTH, NEW_LENGTH,
                              FREQUENCY_FACTOR, SPEED_FACTOR));
        add(new PolarToRectangular());
        add(new SumReals(NEW_LENGTH));
    }
}

class Main extends StreamIt implements Constants {
    public static void main(String args[]) {
        new Main().run(args);
    }

    public void init() {
        add(new StepSource(100));
        add(new IntToFloat());
        add(new Delay(DFT_LENGTH_NOM));
        add(new Vocoder());
        add(new InvDelay((DFT_LENGTH -2) * m_LENGTH / n_LENGTH));
        add(new FloatToShort());
        add(new ShortPrinter());
        add(new ShortVoid());
    }
}
```

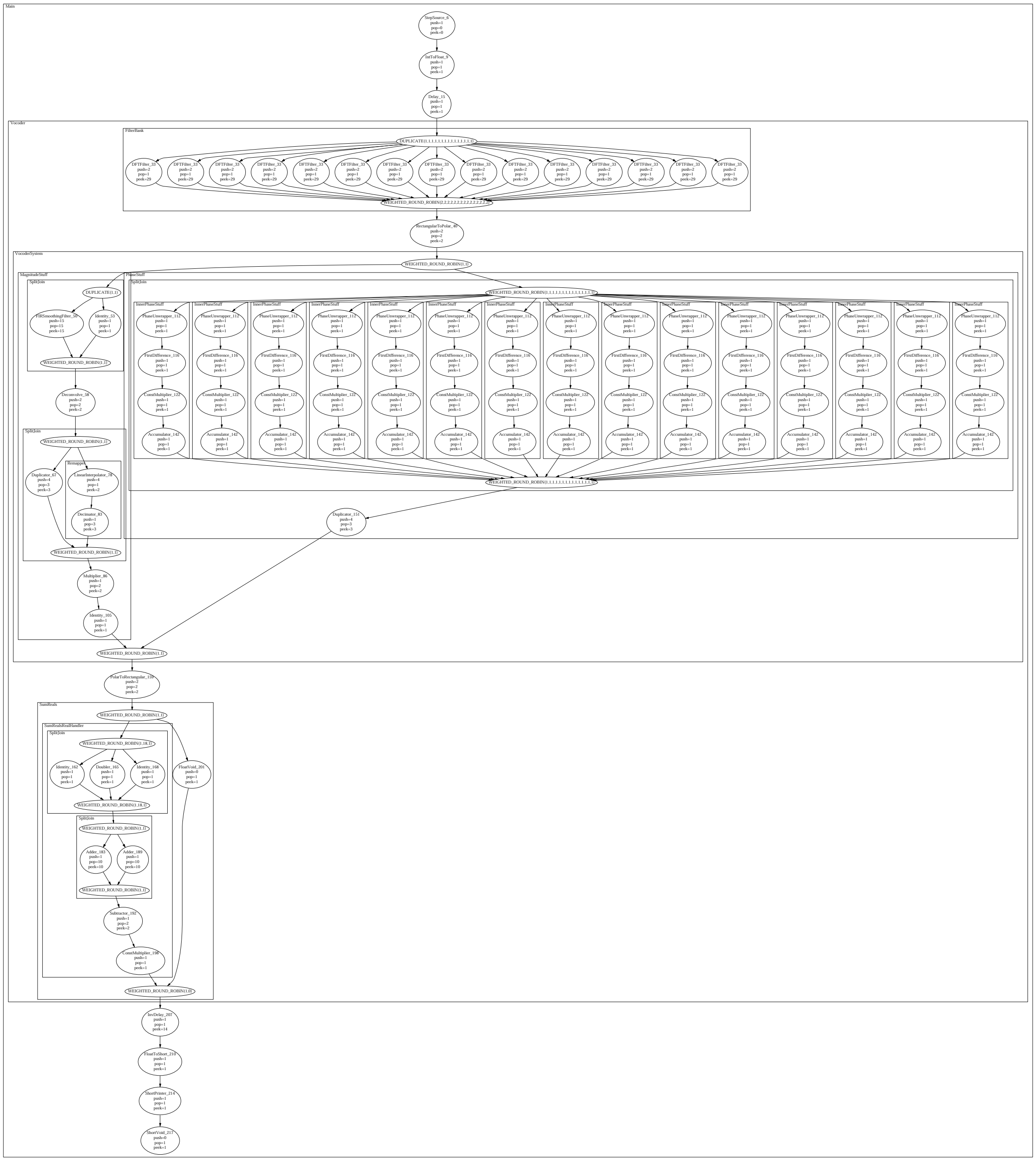


Figure H-1: Vocoder before partitioning.

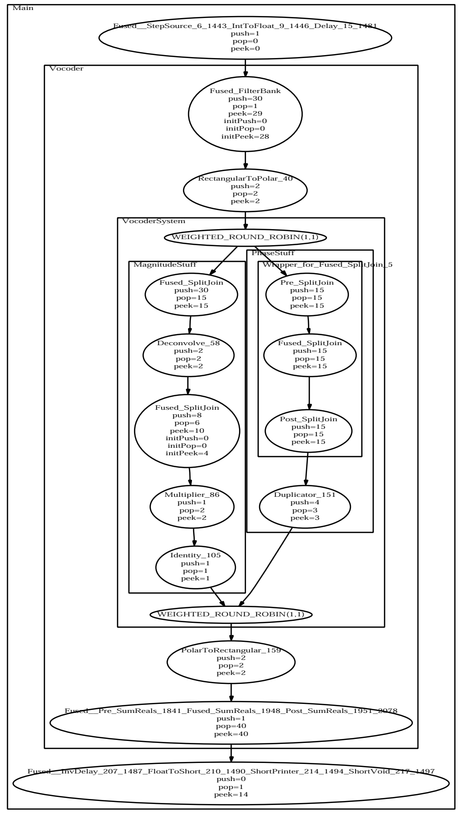


Figure H-2: Vocoder after partitioning.

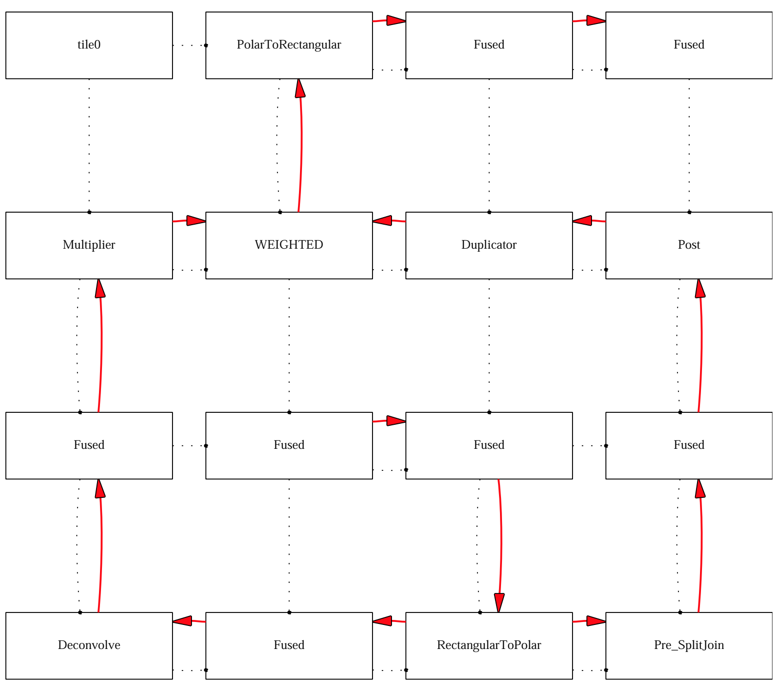


Figure H-3: Vocoder layout.

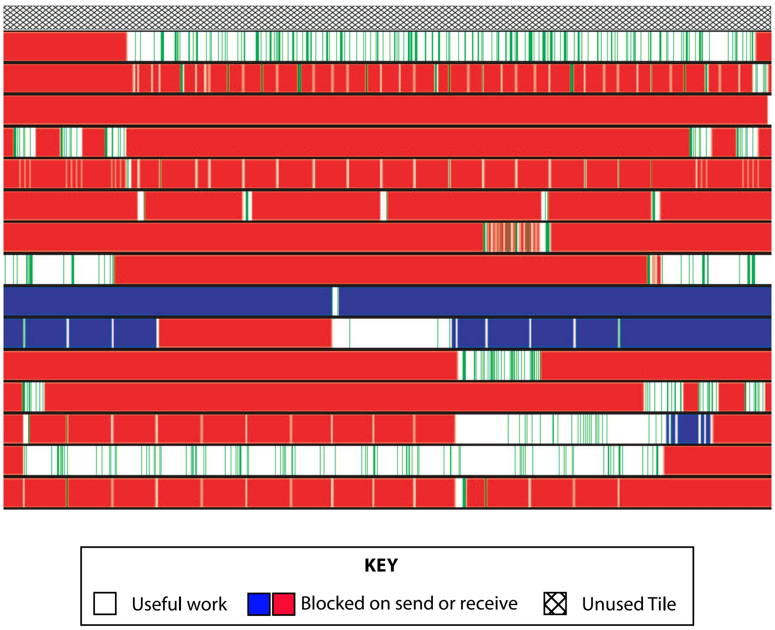


Figure H-4: Vocoder execution trace.

Appendix I

3GPP Application

I.1 Description

This application implements algorithms for data and channel estimation that are deployed in 3GPP TDD handhelds and basestations. This application first calculates the system matrix, A, using the channel impulse response, h and the user signature c. Then uses the matrix A to perform pre-withening and matched filtering operations. These operations require solving a system of linear equations based on (A_h.A) and are, therefore, very computationally intensive. As a result an efficient matrix factorization scheme, called Cholsky decomposition is used.

I.2 Code

```
import streamit.*;
import streamit.io.*;

class SourceAHL extends Pipeline{

    public SourceAHL(int W, int Q, int N, int K, float[][] h,
         float[][] C) {super (W,Q,N,K,h,C);}

    public void init(int W, int Q,int N,int K,float [][] h,
        float [][]C){
        add (new Sourceh(W,K,h));
        add (new GenA(W,Q,N,K,C));
        add (new AandL(Q*N+W-1,K*N));
    }
}

class AandL extends SplitJoin{
// the input to this filter is the matrix A(row oriented),
//the out put is matrix A (row oriented) plus its cholskey
//decomposition factor L}

    public AandL ( int M, int N){super(M,N);}

    public void init(int M, int N){
        setSplitter(DUPLICATE());
        add (new Identity(Float.TYPE));
        add (new GenL(M,N));
        setJoiner(WEIGHTED_ROUND_ROBIN(M*N,N*(N+1)));
    }
}

class GenL extends Pipeline{

    public GenL (int M, int N) {super (M,N);}

    public void init(int M,int N) {
        add (new RowCol(M,N));
        add (new SelfProd(M,N));
        add (new choldAhA(N));
        //          add (new chold(N));
    }
}

class choldAhA extends SplitJoin{// the input is AhA, the
\\output is cholskey decomposition, N is the dim of Aha

    public choldAhA(int N) {super(N);}

    public void init(int N){
        setSplitter(DUPLICATE());
        add (new chold(N));
        add (new Identity(Float.TYPE));
        setJoiner(WEIGHTED_ROUND_ROBIN(N*(N+1)/2,N*(N+1)/2));
    }
}


class backs extends Filter // this Filter performs back
//substition LTd=y.
{
    int    N; //  the dimension of the matrix
    float[][]  LT; // L is the input matrix
    float[]  d; // d is the output result
    float[]  y; //

    float    sum ; //this will be used as a buffer variable

    public backs(int N){ super (N);}
    public void init (int N) {
        input = new Channel(Float.TYPE, N+N*(N+1)/2);
        output = new Channel(Float.TYPE, N);

        y=new float[N];
        d=new float[N];
        LT=new float[N][N];
        //this.LT=LT;
        this.N=N;
```

```
    }

    public void work() {

        for (int i=0; i<N ; i++)
            y[i]=input.popFloat();


        for (int i=0; i<N;i++)
            for (int j=i; j<N;j++){
                LT[i][j]=input.popFloat();
            }

        for (int i=N-1; i>=0;i--)
            {
                sum=y[i];
                for (int j=i+1; j<N ; j++)
                    sum -= LT[i][j]*y[j];
                y[i]=sum/LT[i][i];
                output.pushFloat(y[i]);
            }
    }


}

class chold extends Filter
// this Filter performs the cholesky decomposition through
{
    int    N; //  the dimension of AhA
    float[][]  A; // A is the input matrix

    float[]  p; // p is the out put elements on the diagonal
    float    sum; // sum will be used as a buffer
    public chold(int N){ super (N);}
    public void init (int N) {
        input = new Channel(Float.TYPE, N*(N+1)/2);
        output = new Channel(Float.TYPE, N*(N+1)/2);
        A= new float[N][N];
        p=new float[N];
        this.N=N;

    }


    public void work() {
        float sum; // sum serves as a buffer
        for (int i=0; i<N;i++)
            {
                for (int j=0; j<=i ; j++)
                    A[i][j]=input.popFloat();

            }

        for (int i=0; i <N ; i++) {
            for (int j=i; j<N ; j++) {
                sum=A[j][i];
                for (int k=i-1 ; k>=0; k--)
                    sum-=A[k][i]*A[k][j];
                if ( i==j)
                    {
                        p[i]=(float)Math.sqrt(sum);
                        output.pushFloat(p[i]);
                    }
                else
                    {
                        A[i][j]=sum/p[i];
                        output.pushFloat(A[i][j]);
                    }
            } }

    }


}


class LrL extends SplitJoin{// performes the forward
    //substitution
    public LrL(int N) {super (N);}
    public void init(int N) {
        setSplitter
            (WEIGHTED_ROUND_ROBIN(N+N*(N+1)/2,N*(N+1)/2));
        add (new forw(N));
        add (new Identity(Float.TYPE));
        setJoiner(WEIGHTED_ROUND_ROBIN(N,N*(N+1)/2));
    }

}

class AddAHLAhA extends Pipeline{
// calculates the matrix AH (row oriented?) and L and
// adds them to the tape, plus a copy of AhA
    public AddAHLAhA(int W,int Q,int N, int K, float[][] h,
                     float[][] C   ) {super (W,Q,N,K,h,C);}
    public void init(int W,int Q,int N, int K, float[][] h,
                     float [][] C) {
        add (new SourceAHL(W,Q,N,K,h,C));
    }

}

class vectdouble extends SplitJoin{// duplicates a vector
    public vectdouble( int M) {super (M);}
    public void init(int M) {
        setSplitter(DUPLICATE());
        add (new Identity(Float.TYPE));
        add (new Identity(Float.TYPE));
        setJoiner(ROUND_ROBIN(M));
    }

}

class sourcerSplit extends SplitJoin {
    public sourcerSplit(int M,int end,int left)
    {/*super(M,end,r);*/}

    public void init(int M,int end,int left) {
        setSplitter(WEIGHTED_ROUND_ROBIN(0,left-M));
        add(new Sourcer(M));
        add(new Identity(Float.TYPE));
        setJoiner(WEIGHTED_ROUND_ROBIN(M,left-M));
    }
}

class multvectdoub extends Pipeline{// duplicates a vector
    //and makes a copy
    public multvectdoub( int M,int N,int end,int left)
    {/*super (M,N,end,r);*/}
    public void init(int M, int N,int end,int left) {
        add (new sourcerSplit(M,end,left));
        add (new multvect(M,N));
        add (new vectdouble(N));
    }

}


class AhrL1 extends SplitJoin{// calculates Ahr and
    //duplicates L and passes  Ahr,L (2 of them) to
    //the next level
    public AhrL1( int M,int N,int end)
    {/*super (M,N,end,r);*/}
    public void init(int M,int N,int end) {
        setSplitter
            (WEIGHTED_ROUND_ROBIN(M*(N+1)-M,N*(N+1)/2));
        add (new multvectdoub(M,N,end,M*(N+1)));
        add (new vectdouble(N*(N+1)/2));
        setJoiner(WEIGHTED_ROUND_ROBIN(2*N,N*(N+1)));
    }

}

class dsolve extends Pipeline { //input to this pipeline
    //is Ahr(N),L(N*N) and the output is d
    public dsolve(int N) {super(N);}
    public void init(int N){
        add (new LrL(N));
        add (new backs(N));
    }
}

class split_ahrd extends SplitJoin{
    //In:2* Ahr(N)+ 2 * L(N*(N+1)/2)
    public split_ahrd( int N) {super (N);}
    public void init(int N) {
        setSplitter(WEIGHTED_ROUND_ROBIN(N,N*(N+1)+N));
        add (new vectdouble(N));
        add (new dsolve(N));
        setJoiner(WEIGHTED_ROUND_ROBIN(2*N,N));
    }

}


class Ahrd extends Pipeline{// the input is Ar, L ,
    //the output is Ahr,d,AhA
    public Ahrd( int M,int N,int end) {
        /*super (M,N,end,r);*/}
    public void init(int M,int N,int end) {
        add (new AhrL1(M,N,end));
        add (new split_ahrd(N));
```

```
    }

}

class AhrdAhA extends SplitJoin{// the input is r,
    //L,AhA, the output is Ahr,d,AhA
    public AhrdAhA(int M,int N,int end)
    {/*super (M,N,end,r);*/}
    public void init(int M,int N,int end) {
        setSplitter
            (WEIGHTED_ROUND_ROBIN(M*(N+1)+N*(N+1)/
                                  2-M,N*(N+1)/2));
        add (new Ahrd(M,N,end));
        add (new Identity(Float.TYPE));
        setJoiner(WEIGHTED_ROUND_ROBIN(3*N,N*(N+1)/2));
    }

}

class AhrL2 extends SplitJoin{// calculates Ahr
    //and duplicates L, suitable for use in the second stage
    public AhrL2( int M,int N) {super (M,N);}
    public void init(int M,int N) {
        setSplitter(WEIGHTED_ROUND_ROBIN(M*(N+1),N*(N+1)/2));
        add (new multvect(M,N));
        add (new vectdouble(N*(N+1)/2));
        setJoiner(WEIGHTED_ROUND_ROBIN(N,N*(N+1)));
    }

}

class Sourcer extends Filter {
    int N;
    float[] r;
    public Sourcer(int N) {super(N, r);}
    public void init(int N){
        r = new float[6];
        r[0]=1;
        r[1]=2;
        r[2]=3;
        r[3]=4;
        r[4]=5;
        r[5]=6;
        output = new Channel(Float.TYPE, N);
        this.N=N;
    }
    public void work(){
        for(int i=0;i<N;i++)
            output.pushFloat(r[i]);
    }
}

class SinkD extends Filter{
    int N;
    public SinkD(int N) {super(N);}
    public void init(int N){
        input = new Channel(Float.TYPE, N);
        this.N=N;
    }
    public void work() {
        System.out.println("Starting");

        for (int i=0; i< N;i++)
            {
                System.out.println(input.popFloat());
            }

    }
}

class error_est extends Filter{
// this class estimates the error in signal detection

    int N;
    float[] Ahr,d;

    public error_est(int N) {super(N);}
    public void init(int N){
        this.N=N;
        input = new Channel(Float.TYPE, 2*N);
        output = new Channel(Float.TYPE, 1);
        Ahr=new float[N];
        d= new float[N];

    }
    public void work() {
        float sigma=0;
        for (int i=0; i< N;i++){
            Ahr[i]=input.popFloat();
                }
        for (int i=N-1; i >=0; i--){
            d[i]=input.popFloat();
        }
        for (int i=0; i <N ; i++)
            sigma+=(d[i]-Ahr[i])*(d[i]-Ahr[i]);
        output.pushFloat(sigma);


    }
}


class choldsigma extends Filter
 // this Filter performs the cholesky decomposition through
{
    int    N; //  the dimension of AhA
    float[][]  A; // A is the input matrix

    float[]  p; // p is the out put elements on the diagonal
    float    sum; // sum will be used as a buffer
    float    sigma;
    public choldsigma(int N){ super (N);}
    public void init (int N) {
        input = new Channel(Float.TYPE, N*(N+1)/2+1);
        output = new Channel(Float.TYPE, N*(N+1)/2);
        A= new float[N][N];
        p=new float[N];
        this.N=N;
    }


    public void work() {
        float sum; // sum serves as a buffer
        sigma=input.popFloat();
        for (int i=0; i<N;i++)
            {
                for (int j=0; j<=i ; j++)
                    A[i][j]=input.popFloat();

            }

        for (int i=0; i <N ; i++) {
            for (int j=i; j<N ; j++) {
                sum=A[j][i];
                for (int k=i-1 ; k>=0; k--)
                    sum-=A[k][i]*A[k][j];
                if ( i==j)
                    {
                        p[i]=(float)Math.sqrt(sum+sigma/N);
                        output.pushFloat(p[i]);
                    }
                else
                    {
                        A[i][j]=sum/p[i];
                        output.pushFloat(A[i][j]);
                    }
            }
        }
    }

}

class error_split extends SplitJoin{
    // performs error estimation for the
    //first 2*N elements and copies the AhA
    public error_split( int N) {super (N);}
    public void init(int N) {
        setSplitter(WEIGHTED_ROUND_ROBIN(2*N,N*(N+1)/2));
        add (new error_est(N));
        add (new Identity(Float.TYPE));
        setJoiner(WEIGHTED_ROUND_ROBIN(1,N*(N+1)/2));
    }
}

class Lest extends Pipeline{
//  this pipeline estimates the error and
// then performes the cholskey decomp
    public Lest( int N) {super (N);}
    public void init(int N) {
        add (new error_split(N));
        add (new choldsigma(N));
        add (new vectdouble(N*(N+1)/2));
    }

}


class  Ahrchold extends SplitJoin{
// copies Ahr to its out put and performes the compensated
// cholesky decomp with Ahr,d,AHA
    public Ahrchold( int N) {super (N);}
    public void init(int N) {
        setSplitter(WEIGHTED_ROUND_ROBIN(N,2*N+N*(N+1)/2));
```

```
        add (new Identity(Float.TYPE));
        add (new Lest(N));
        setJoiner(WEIGHTED_ROUND_ROBIN(N,N*(N+1)));
    }
}

class dcalc extends StreamIt {

    static public void main(String[] t)
    {
        StreamIt test=new dcalc();
        test.run(t);
    }


    public void init() {
        int K;
        int N;
        int Q;
        int W;
        float[][] h;
        float[][] C;
        float[] r;
        K=2;
        N=2;
        Q=2;
        W=2;
        h=new float[2][2];
        C=new float[2][2];
        r=new float[6];
        h[0][0]=1;
        h[0][1]=3;
        h[1][0]=2;
        h[1][1]=5;
        C[0][0]=1;
        C[0][1]=0;
        C[1][0]=1;
        C[1][1]=2;
        r[0]=1;
        r[1]=2;
        r[2]=3;
        r[3]=4;
        r[4]=5;
        r[5]=6;

        add(new SourceAHL(W,Q,N,K,h,C));
        add(new AhrdAhA(Q*N+W-1,K*N,K*N*(Q*N+W-1)+
                        (K*N)*(K*N+1)));
        add(new Ahrchold(K*N));
        add(new LrL(K*N));
        add(new backs(K*N));
        add(new FileWriter("out",Float.TYPE));
    }
}

/**
 * Simple parameterized delay filter.
 **/

public class DelayPipeline extends Pipeline {
    public DelayPipeline(int delay) {
        super(delay);
    }
    public void init(int delay) {
        // basically, just add a bunch of unit delay filters
        for (int i=0; i<delay; i++) {
            this.add(new Delay());
        }
    }
}

/** Character Unit delay **/
class Delay extends Filter {
    float state;
    public void init() {
        // initial state of delay is 0
        this.state = 0.0f;
        input = new Channel(Float.TYPE,1);
        output = new Channel(Float.TYPE,1);
    }
    public void work() {
        // push out the state and then update
        // it with the input
        // from the channel
        output.pushFloat(this.state);
        this.state = input.popFloat();
    }
}

/**
 * Class FirFilter
 *
 * Implements an FIR Filter
 */

public class FirFilter extends Filter {

    int N;
    float COEFF[];

    public FirFilter (float[] COEFF)
    {
        super (COEFF);
    }

    public void init(float[] COEFF) {
        this.N=COEFF.length;
        //this.COEFF=COEFF;
        this.COEFF=new float[2];
        this.COEFF[0]=COEFF[0];
        this.COEFF[1]=COEFF[1];
        input = new Channel(Float.TYPE, 1, COEFF.length);
        output = new Channel(Float.TYPE, 1);
    }

    public void work(){
        float sum=0;
        for (int i=0; i<N ; i++)
            sum+=input.peekFloat(i)*COEFF[N-1-i];
        input.pop();
        output.pushFloat(sum);
    }
}

class forw extends Filter // this Filter performs
                   //forward substition LY=b.
{
    public forw(int N) { super (N);}
    int    N; //  the dimension of the matrix
    float[][]  L; // L is the input matrix
    float[]  y; // y is the output result
    // we do not need to store the vector b
    float    sum ; //this will be used as a buffer variable

    public void init(int N) {
        input = new Channel(Float.TYPE, N+N*(N+1)/2);
        output = new Channel(Float.TYPE, N);
        y=new float[N];
        L=new float[N][N];
        this.N=N;
    }

    public void work() {
        for (int i=0; i <N; i++) {
            y[i]=input.popFloat();
        }

        for( int i=0; i <N; i++)
            for (int j=i; j<N; j++){
                L[j][i]=input.popFloat();
            }

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

                sum= y[i];
                for (int j=0; j<i ; j++)
                    sum -= L[i][j]*y[j];
                y[i]=sum/L[i][i];
                output.pushFloat(y[i]);
            }
    }
}

class DelMat extends SplitJoin {
// genrates the proper delays for the convolution of C and h

    public DelMat(int Q, int  N) {super (Q,N);}

    public void init(int Q,int N) {
        setSplitter(DUPLICATE());
        add(new Identity(Float.TYPE));
        for(int i=1;i<N;i++){
            add(new DelayPipeline(i*Q));
        }
        setJoiner(ROUND_ROBIN());
    }
}

class ConvMat extends SplitJoin{
// generates the matrix consisting of the convolution
//of h and c. reads h column wise as in [1]

    public ConvMat(int K, int W, int Q,int N,float[][] C)
```

```
    {super (K,W,Q,N,C);}
    public void init(int K,int W, int Q,int N,float[][] C){
        float[] Crow;
        setSplitter(ROUND_ROBIN(W));
        for (int i=0;i<K;i++){
            Crow = new float[Q];
            add(new extFilt(W,W+N*Q-1,Crow));
        }
        setJoiner(ROUND_ROBIN(W+N*Q-1));
    }
}

class SplitMat extends SplitJoin {
    // connects the ConvMat to
    // DelMat

    public SplitMat(int W,int Q,int K, int N)
    {super (W,Q,K,N);}

    public void init(int W,int Q,int K, int N){
        setSplitter(ROUND_ROBIN(N*Q+W-1));
        for (int i=0;i<K;i++){
            add(new DelMat(Q,N));
        }
        setJoiner(ROUND_ROBIN(N));
    }
}

class AddZeroEnd extends SplitJoin{
// adds (M-L)zeros to a sequence of length L to make
// it have the right size
    public AddZeroEnd(int L, int M) {super (L,M);}
    public void init(int L,int M) {
        setSplitter(WEIGHTED_ROUND_ROBIN(L,0));
        add (new Identity(Float.TYPE));
        add (new ZeroGen());
        setJoiner(WEIGHTED_ROUND_ROBIN(L,M-L));
    }

}

class AddZeroBeg extends SplitJoin{
// adds M zeros to the begining of a sequence of
//length L to make it have the right size
    public AddZeroBeg( int M,int L) {super (M,L);}
    public void init(int M,int L) {
        setSplitter(WEIGHTED_ROUND_ROBIN(0,L));
        add (new ZeroGen());
        add (new Identity(Float.TYPE));
        setJoiner(WEIGHTED_ROUND_ROBIN(M,L));
    }

}

class ZeroGen extends Filter{
// this filter just generates a sequence of zeros
    public void init() {
        output = new Channel(Float.TYPE, 1);
    }
    public void work(){
        output.pushFloat(0);
    }
}

class extFilt extends Pipeline{
// this filter performs the convolution of L
// and then extends the sequenc
    public extFilt(int W,int M,float[] impulse)
    {super (W,M,impulse);}
    public void init(int W, int M,float[] impulse){
        add (new AddZeroBeg(impulse.length-1,W));
        add (new FirFilter(impulse));
        add (new AddZeroEnd(W+impulse.length-1,M));
    }
}


class GenA extends Pipeline{
// the whole matrix A generator, the input is column wise
// and the out put is row wise
    public GenA(int W,int Q, int N, int K, float[][] C)
    {super (W,Q,N,K,C);}
    public void init(int W, int Q, int N, int K,
                     float[][] C)
    {
        add(new ConvMat(K,W,Q,N,C));
        add(new SplitMat(W,Q,K,N));
    }
}

class multvect extends Filter // this Filter performs b=AHr
{
    int    N; //  the dimension of the matrix
    float[][]  AH; // AH is the input matrix
    //  it is not neccessary to save b. b is
    // generated in the order b[0],b[1],b[2]....
    float[]  r;//
    float    sum; //sum will be used as a buffer
    int    M;

    public multvect(int M,int N) { super (M,N);}
    public void init (int M,int N) {
        input = new Channel(Float.TYPE, M+N*M);
        output = new Channel(Float.TYPE, N);
        r=new float[M];
        AH=new float[N][M];
        this.N=N;
        this.M=M;
    }


    public void work() {
        for (int i=0; i<M ; i++)
            r[i]=input.popFloat();
        for (int i=0; i<M;i++)
            for (int j=0; j<N;j++)
                AH[j][i]=input.popFloat();
        for (int i=0; i<N;i++)
            {
                sum=0;
                for (int j=0; j<M ; j++)
                    sum += AH[i][j]*r[j];
                output.pushFloat(sum);
            }
    }
}

class RowCol extends SplitJoin
// this Filter converts the elements of an m by n
// matrix from row by row format to column by column format
{
    int    M;// the number of rows
    int    N;// the number of columns


    public RowCol(int M, int N){ super (M,N);}
    public void init ( int M, int N) {
        setSplitter(ROUND_ROBIN());
        for (int i=0; i< N;i++)
            add(new Identity(Float.TYPE));
        setJoiner(ROUND_ROBIN(M));

    }
}

class SelfProd extends Filter
// this Filter mutiplies a matrix by its conjugate
//M is the number of rows, N is the number columns,
//elements of the A are read column by column
{
    int    M;// the number of rows
    int    N;// the number of columns

    public SelfProd(int M, int N){ super (M,N);}

    public void init ( int M, int N) {
        input = new Channel(Float.TYPE, N*M);
        output = new Channel(Float.TYPE, N*(N+1)/2);

        this.M=M;
        this.N=N;

    }


    public void work() {
        float[][] A=new float[M][N];
        for (int i=0; i<N;i++)
            {
                for (int j=0; j<M;j++)
                    A[j][i]=input.popFloat();

                for (int k=0; k<=i ; k++)
                    {
                        float prod=0;
                        for(int j=0; j<M; j++)
                            {
                                prod=prod+ A[j][i]*A[j][k] ;
                            }
                        output.pushFloat(prod);
                    }
            }
    }
}
```

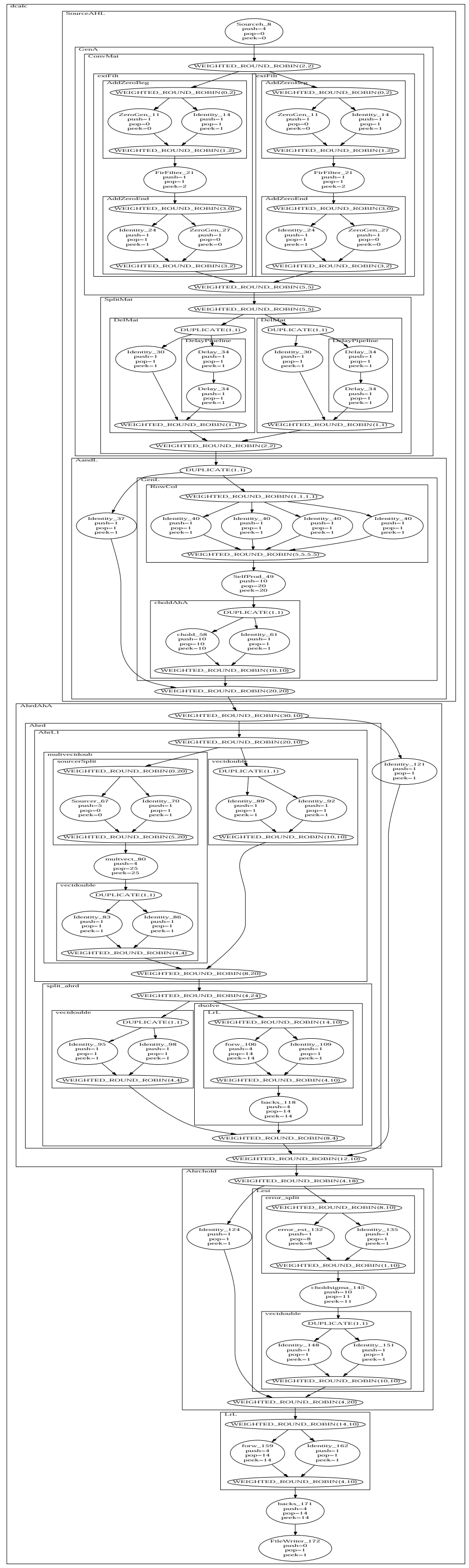


Figure I-1: 3GPP before partitioning.

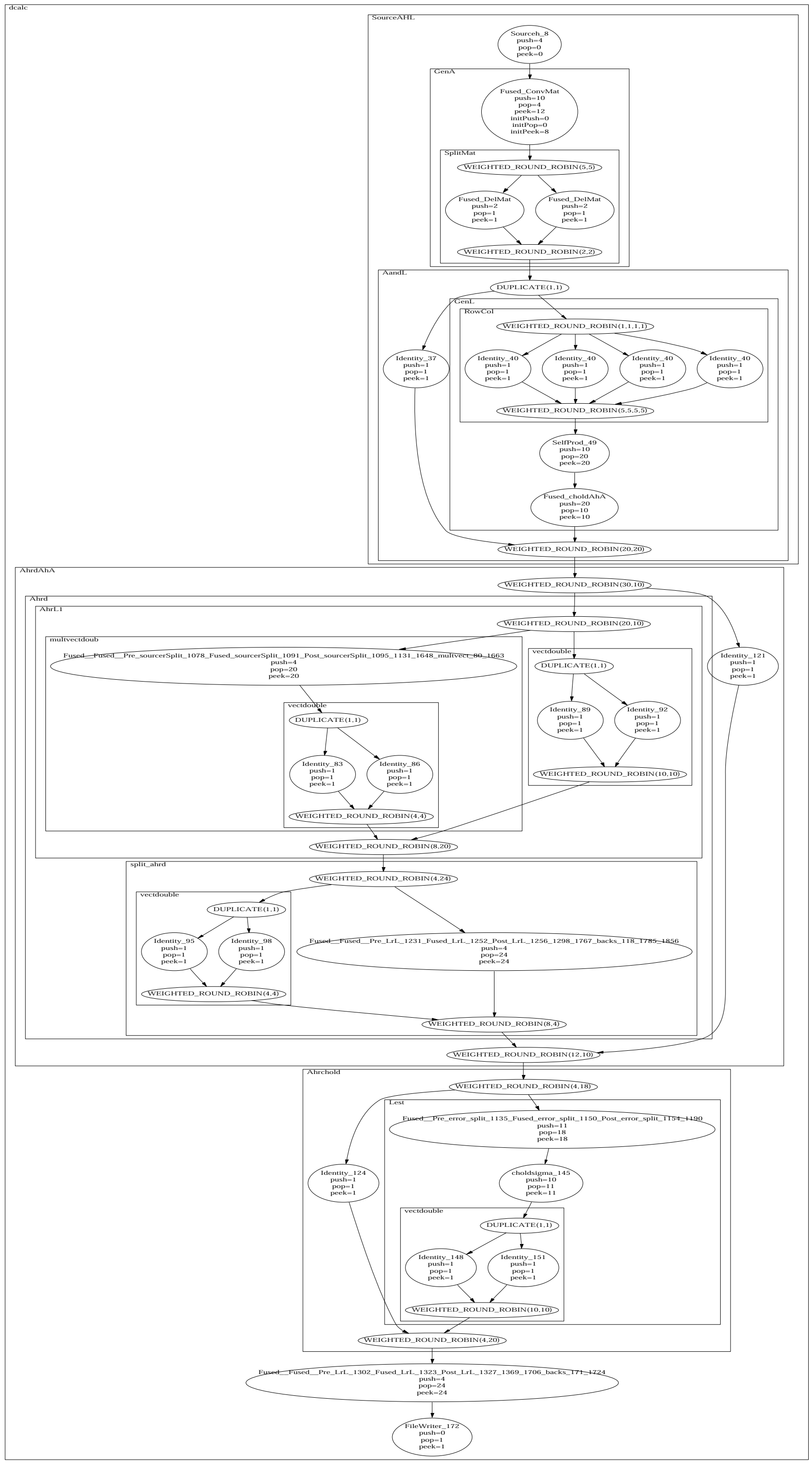


Figure I-2: 3GPP after partitioning.

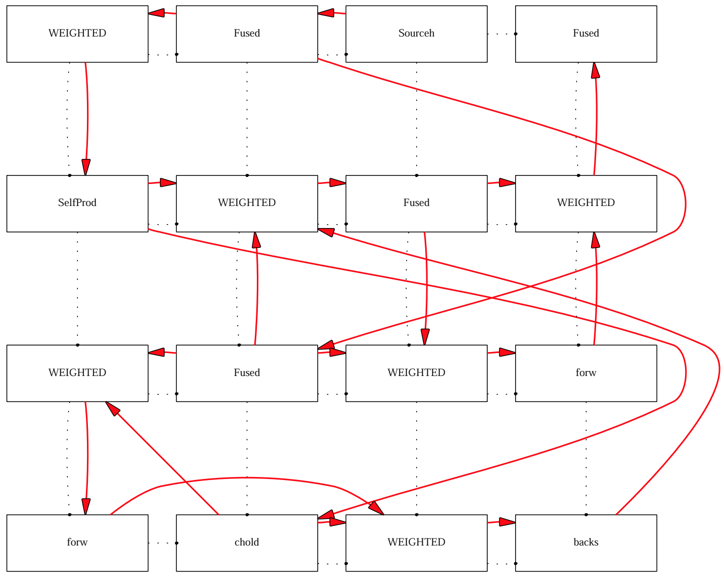


Figure I-3: 3GPP layout.

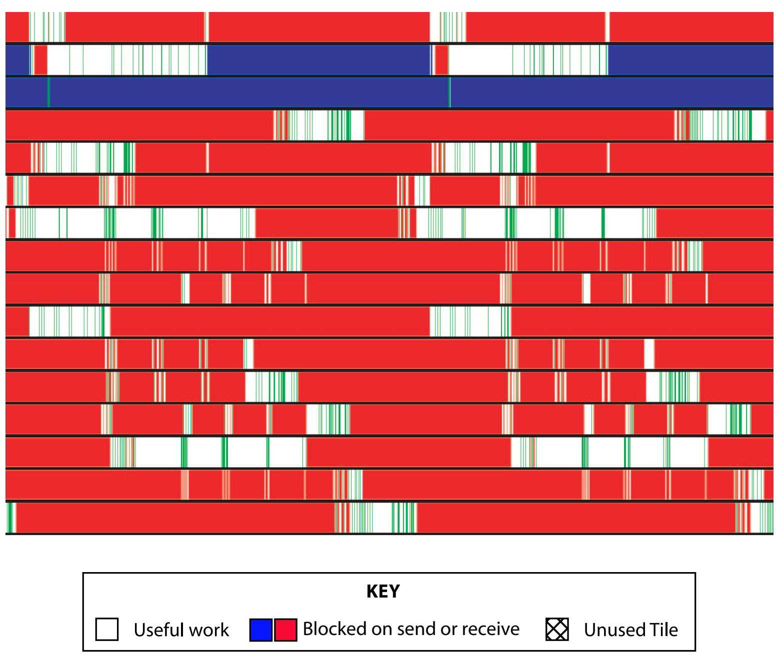


Figure I-4: 3GPP execution trace.

Bibliography

[1] http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm.

[2] http://www.cs.berkeley.edu/~randit/papers/csd-00-1106.pdf.

[3] Lecture notes for University of California, Berkeley's class CS267. http://www.cs.berkeley.edu/~demmel/cs267/lecture24/lecture24.html.

[4] Streamit homepage. http://compiler.lcs.mit.edu/streamit.

[5] *The Transputer Databook.* Inmos Corporation, 1988.

[6] 3rd Generation Partnership Project. *3GPP TS 25.201, V3.3.0, Technical Specification*, March 2002.

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

[8] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs.* Kluwer Academic Publishers, 1996.

[9] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. *IEEE Trans. on Signal Processing*, 1996.

[10] Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H. T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin, P. S. Tseng, Jim Sutton, John Urbanski, and Jon Webb. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing*, 1988.

[11] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE): Extended Abstract. In *Proceedings of the Conference on Field Programmable Logic and Applications*, 2000.

[12] Michael Bedford Taylor et. al. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. IEEE Micro vol 22, Issue 2, 2002.

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

[14] Thierry Gautier, Paul Le Guernic, and Loic Besnard. Signal: A declarative language for synchronous programming of real-time systems. *Springer Verlag Lecture Notes in Computer Science*, 274, 1987.

[15] Vincent Gay-Para, Thomas Graf, Andre-Guillaume Lemonnier, and Erhard Wais. Kopi Reference manual. http://www.dms.at/kopi/docs/kopi.html, 2001.

[16] Thomas Gross and David R. O'Halloron. *iWarp, Anatomy of a Parallel Computing System.* MIT Press, 1998.

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

[18] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. In *Proc. of the IEEE*, 2001.

[19] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

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

[21] Ujval J. Kapasi, Peter Mattson, William J. Dally, John D. Owens, and Brian Towles. Stream scheduling. In *Proc. of the 3rd Workshop on Media and Streaming Processors*, 2001.

[22] Michal Karczmarek. Constrained and Phased Scheduling of Synchronous Data Flow Graphs for the StreamIt Language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, August 2002.

[23] S. Kirkpatrick, Jr. C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598), May 1983.

[24] Donald Ervin Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, 1998.

[25] Andrea S. LaPaugh. Layout Algorithms for VLSI Design. *ACM Computing Surveys*, 28(1), March 1996.

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

[27] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, January 1987.

[28] Edward A. Lee. Overview of the Ptolemy Project. UCB/ERL Technical Memorandum UCB/ERL M01/11, Dept. EECS, University of California, Berkeley, CA, March 2001.

[29] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman P. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. MIT-LCS Technical Memo LCS-TM-572, Cambridge, MA, December 1997.

[30] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular recongurable architecture. In *ISCA 2000*, Vancouver, BC, Canada.

[31] David May, Roger Shepherd, and Catherine Keane. Communicating Process Architecture: Transputers and Occam. *Future Parallel Computers: An Advanced Course, Pisa, Lecture Notes in Computer Science*, 272, June 1987.

[32] A. Mitschele-Thiel. Automatic Configuration and Optimization of Parallel Transputer Applications. *Transputer Applications and Systems '93*, 1993.

[33] David R. O'Hallaron. The ASSIGN Parallel Program Generator. Carnegie Mellon Technical Report CMU-CS-91-141, 1991.

[34] Todd A. Proebsting and Scott A. Watterson. Filter Fusion. In *POPL*, 1996.

[35] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *International Symposium on Microarchitecture*, 1998.

[36] K. Sankaralingam, R. Nagarajan, S.W. Keckler, and D.C. Burger. A Technology-Scalable Architecture for Fast Clocks and High ILP. University of Texas at Austin, Dept. of Computer Sciences Technical Report TR-01-02, 2001.

[37] Stephanie Seneff. Speech transformation system (spectrum and/or excitation) without pitch extraction. Master's thesis, Massachussetts Institute of Technology, 1980.

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

[39] Michael Taylor. *The Raw Prototype Design Document, V3.03.* Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, March 2002.

[40] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. Technical Report MIT-LCS-TR-859, Mass. Inst. of Technology, July 2002.

[41] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction*, Grenoble, France, 2002.

[42] William Thies, Michal Karczmarek, Michael Gordon, David Maze, Jeremy Wong, Henry Hoffmann, Matthew Brown, and Saman Amarasinghe. StreamIt: A Compiler for Streaming Applications. MIT-LCS Technical Memo LCS-TM-622, Cambridge, MA, 2001.

[43] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9), 1997.

[44] Elliot L. Waingold. SIFt: A Compiler for Streaming Applications. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2000.