A Stream Compiler for Communication-Exposed Architectures*

Michael Gordon, William Thies, Michal Karczmarek, Jeremy Wong,
Henry Hoffmann, David Maze and Saman Amarasinghe

MIT Laboratory for Computer Science
Cambridge, MA 02139

{mgordon, thies, karczma, jnwong, hank, dmaze, saman}@lcs.mit.edu

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, iWarp, SmartMemories). 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 paper, 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, much analysis is needed to adapt a stream program to a parallel stream processor. We describe fission and fusion transformations that can be used to adjust the granularity of a stream graph, a layout algorithm for mapping a stream graph to a given network topology, and a scheduling algorithm for generating 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 optimizations. Using the cycle-accurate Raw simulator, we demonstrate that these optimizations can improve performance by up to 145%. We consider this work to be a first step towards a portable programming model for communication-exposed architectures.

n

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 [24], SmartMemories [13], TRIPS [19]). 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, the C programming language served this purpose; it abstracted away the idiosyncratic details between one machine and another, but encapsulated the common properties (such as a single program

*This document is the **second version** of MIT Laboratory for Computer Science Technical Memo LCS-TM-627, May, 2002.

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, 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 [22], 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 granularity of the underlying architecture. Our current backend is for Raw [24], a tiled architecture with fine-grained, programmable communication between processors. However, the compiler consists of 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 static communication pattern for each computational element. We consider this work to be a first step towards a portable programming model for communication-exposed architectures.

This paper 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 optimization.
- A graph reordering optimization.
- 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 rest of this paper is organized as follows. Section 2 provides an introduction to StreamIt, Section 3 contains an overview of Raw, and Section 4 outlines our compiler for StreamIt on Raw. Sections 5, 6, and 7 describe our algorithms for partitioning, layout, and communication scheduling, respectively. Section 8 describes code generation for Raw, and Section 9 presents our results from the cycle-accurate Raw simulator. Section 10 considers related work, and Section 11 contains our conclusions.

2 The StreamIt Language

In this section we provide a very brief overview of the StreamIt language; a more detailed description can be found in [22]. The current version of StreamIt has a syntax that is legal Java in order to simplify our presentation and implementation. However, we have developed a complete compiler that is fully independent from the Java runtime system–our syntax should not be mistaken for a Java library. Also, the current version of StreamIt is designed to support only streams with static input and output rates.

The basic unit of computation in StreamIt is the Filter. An example of a Filter is the `LowPassFilter`, a component of our software radio (see Figure 1). Each Filter contains an `init` function that is called

```
class LowPassFilter extends Filter {,
  float[] weights;

  void init(int sampleRate, float cutOffFreq) {
    setInput(Float.TYPE); setOutput(Float.TYPE);
    setPush(N); setPop(1); setPeek(N);
    weights = calcWeights(sampleRate, cutOffFreq);
  }

  void work() {
    float sum = 0;
    for (int i=0; i<weights.length; i++)
      sum += input.peek(i)*weights[i];
    input.pop();
    output.push(sum);
  }
}

public class Equalizer extends Pipeline {
  void init(float samplingRate, int N) {
    add(new SplitJoin() {
      void init() {
        int bottom = 2500;
        int top = 5000;
        setSplitter(Duplicate());
        for (int i=0; i<N; i++, bottom*=2, top*=2) {
          add(new BandPassFilter(sampleRate, bottom, top));
        }
        setJoiner(RoundRobin());
    }});
    add(new Adder(N));
  }
}

class FMRadio extends Pipeline {
  void init() {
    add(new DataSource());
    add(new LowPassFilter(sampleRate, cutoffFreq));
    add(new FMDemodulator(sampleRate, maxAmplitude));
    add(new Equalizer(samplingRate, 4));
    add(new Speaker());
  }
}
```

Figure 1: Parts of an FM Radio in StreamIt.

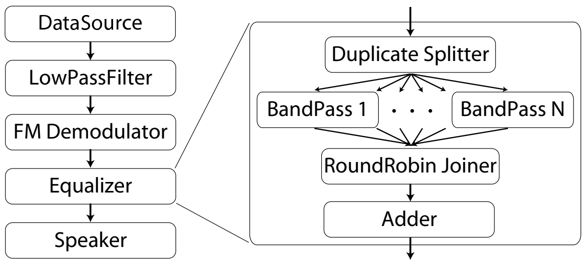
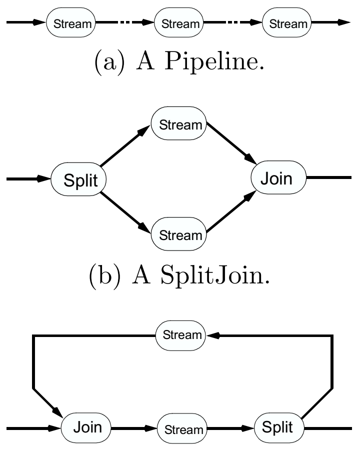


Figure 2: Block diagram of the FM Radio.



(a) A Pipeline.

(b) A SplitJoin.

(c) A FeedbackLoop.

Figure 3: Stream structures supported by StreamIt.

at initialization time; in this case, the `LowPassFilter` calculates `weights`, the coefficients it should use for filtering. 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 using the `input` and `output` channels, which are FIFO queues with types as declared in the `init` function. These high-volume channels support the intuitive operations of `push(value)`, `pop()`, and `peek(index)`, where `peek` returns the value at position `index` without dequeuing the item. The user never calls the `init` and `work` functions–they are called automatically.

The basic construct for composing filters into a communicating network is a Pipeline, such as the FM Radio in Figure 1. Like a Filter, a Pipeline has an `init` function that is called upon its instantiation. However, there is no `work` function, and all input and output channels are implicit; instead, the stream behaves as the sequential composition of filters that are specified with successive calls to `add` from within `init`. That is, the output of `DataSource` is implicitly connected to the input of `LowPassFilter`, who's output is connected to `FMDemodulator`, and so on.

There are two other stream constructors besides Pipeline: SplitJoin and FeedbackLoop (see Figure 3). From now on, we use the word *stream* to refer to any instance of a Filter, Pipeline, SplitJoin, or FeedbackLoop.

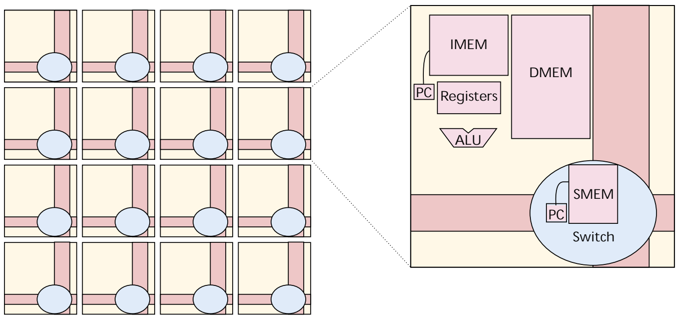


Figure 4: A block diagram of the Raw architecture.

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 calls to `setSplitter` and `setJoiner`, respectively (see Figure 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 last control construct provides a way to create cycles in the stream graph: the FeedbackLoop. Due to space constraints, we omit a detailed discussion of the FeedbackLoop.

2.1 Rationale

StreamIt differs from other stream languages in that it imposes a well-defined structure on the streams; all stream graphs are built out of a hierarchical composition of Filters, Pipelines, SplitJoins, and FeedbackLoops. This is in contrast to other environments, which generally regard a stream as a flat and arbitrary network of filters that are connected by channels. However, arbitrary graphs are very hard for the compiler to analyze, and equally difficult for a programmer to describe. The comparison of StreamIt's structure with arbitrary stream graphs could be likened to the difference between structured control flow and GOTO statements; though the programmer might have to re-design some code to adhere to the structure, the gains in robustness, readability, and compiler analysis are immense.

2.2 Messages

StreamIt provides a dynamic messaging system for passing irregular, low-volume control information between filters and streams. Messages are sent from within the body of a filter's `work` function, perhaps to change a parameter in another filter. The central aspect of the messaging system is a sophisticated timing mechanism that allows filters to specify when a message will be received relative to the flow of data items between the sender and the receiver. With the messaging system, StreamIt is equipped to support full application development–not just high-bandwidth data channels, but also events, control, and re-initialization.

3 The Raw Architecture

The Raw Microprocessor [21, 24] addresses the wire delay problem [6] 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 these resources, a compiler such as StreamIt 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 [5] 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 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 4). The compute processor interfaces to the network through a bypassed, register-mapped interface [21] 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 [12]. 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 hops for a inter-tile distance of N hops.

The results of this paper 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 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. All functional units except the floating point and integer dividers are fully pipelined. The mispredict penalty of the static branch predictor is three cycles, as is the load latency. 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.

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: Phases of the StreamIt compiler.

4 Compiling StreamIt to Raw

The phases of the StreamIt compiler are described in Table 1. The front end is built on top of KOPI, an open-source compiler infrastructure for Java [4]. We translate the KOPI syntax tree into 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.

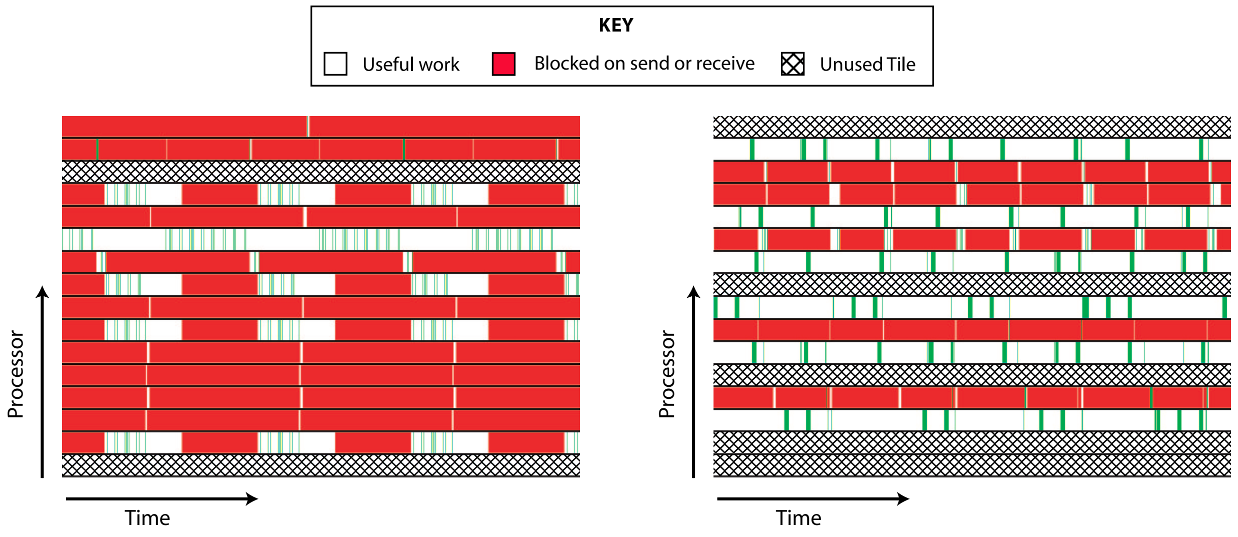
The automatic scheduling of the stream graph is one of the primary benefits that StreamIt offers, and the subtleties of scheduling and buffer management are evident throughout all of the following phases of the compiler. The scheduling is complicated by StreamIt's support for the `peek` operation, which 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 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 [23]. In this paper, we use a simple hierarchical scheduler that constructs a Single Appearance Schedule (SAS) [2] for each filter. An SAS is one where each node appears exactly once in the loop nest denoting the schedule. 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 sections, we refer to the "multiplicity" of a filter as the number of times that it executes in one steady state execution of the entire stream graph.

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

5 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



(a) Original.

(b) Load-balanced.

Figure 5: Execution traces for the (a) original and (b) load-balanced partitionings of the FM Radio. The x axis denotes time, and the y axis denotes the processor. The 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.

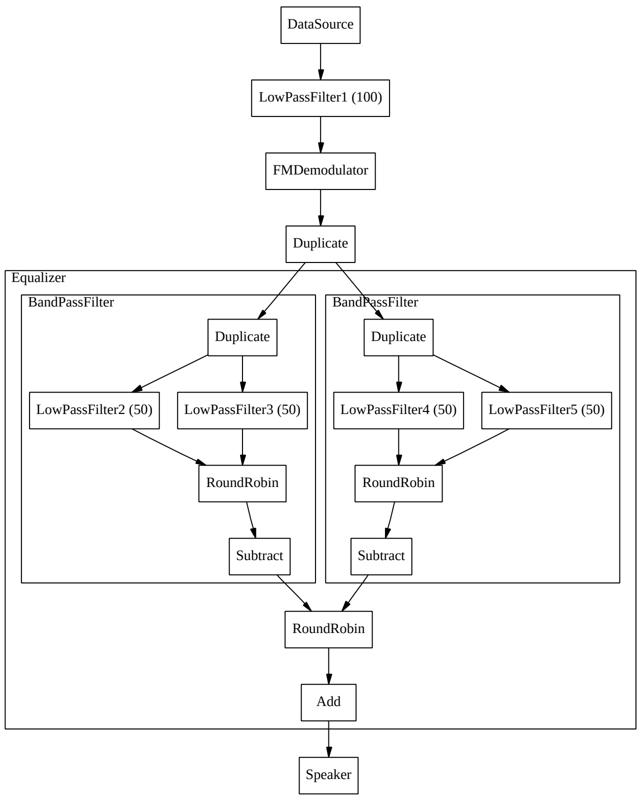


Figure 6: Stream graph of the original FM Radio, in which the Filters are at a granularity that is natural for the algorithm.

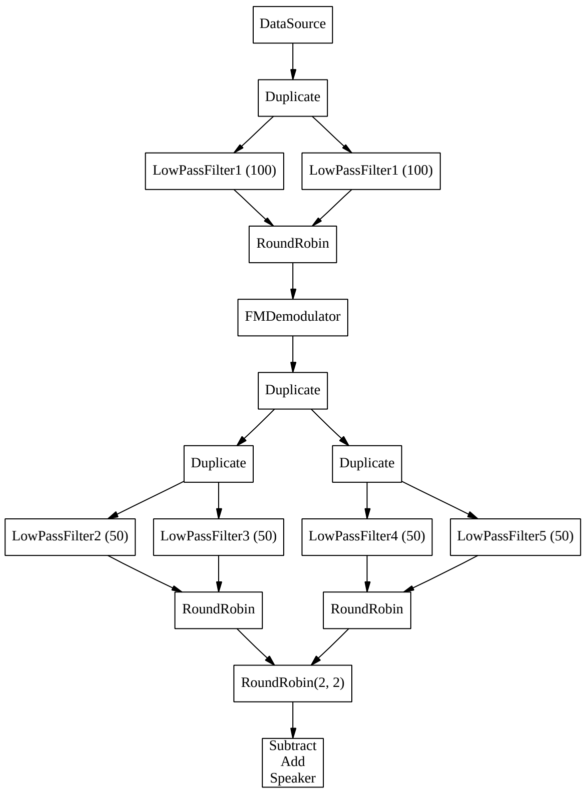


Figure 7: Stream graph of the load-balanced FM Radio. Since the first LowPassFilter was the most demanding node, it was duplicated across a 2-way SplitJoin. Then, to conserve tiles, the Subtract node was hoisted across a downstream Joiner, and the Subtract, Add, and Printer nodes were fused.

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 a number N, which represents the maximum number of computation units that can be supported, 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:

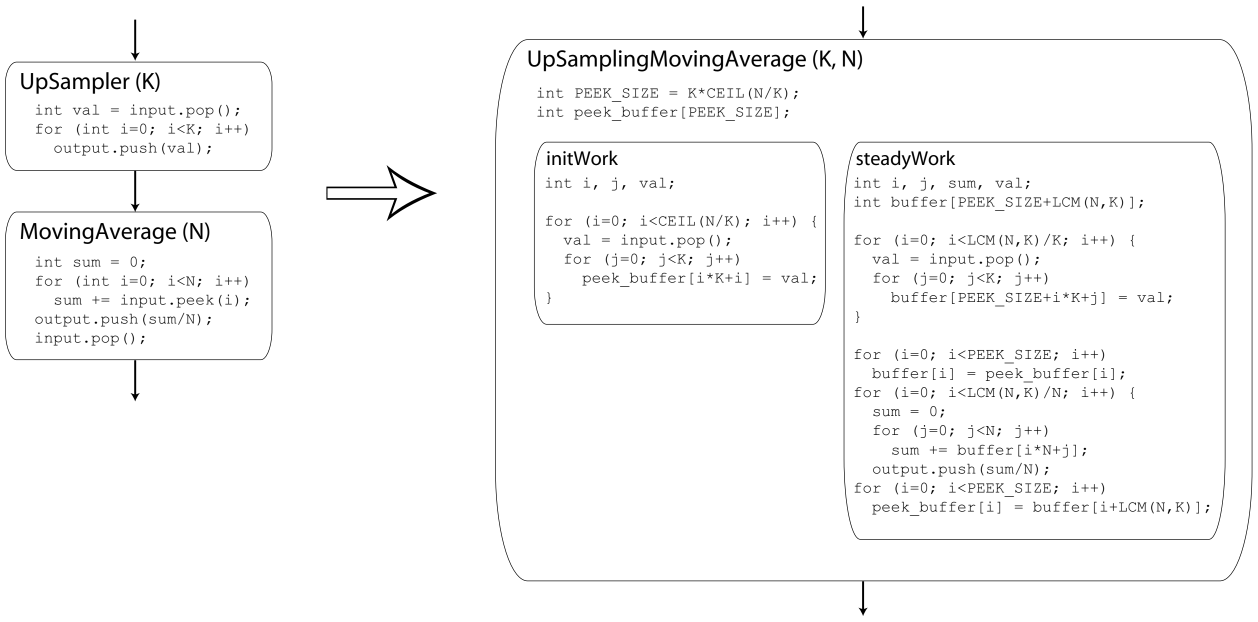
$$\mathit{Maximum\ speedup}(w, c) = \frac{\sum_{i=1}^{N} w_i \cdot c_i}{MAX_i(w_i \cdot c_i)}$$

where $w_1 \ldots w_m$ 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.

5.1 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, the decision of which transformations to apply is done by hand, but the transformations themselves are fully automated.

We use the FM Radio application to illustrate the partitioning process. The original partitioning of the radio (Figure 6) is highly unbalanced, as the first LowPassFilter node constitutes a large fraction of the work in the stream graph. Figure 5(a) provides an execution trace of the original partitioning and demonstrates that this filter is the bottleneck for the computation. The StreamIt compiler performs horizontal fission, filter hoisting, and vertical fusion transformations to obtain the improved stream graph given in Figure 7. As illustrated by the execution trace in Figure 5(b), the load-balanced graph achieves more than twice as much processor utilization as the original, since the computationally intensive nodes have been split and the lighter nodes have been combined. In the following sections, we describe these transformations in more detail.



UpSampler (K)

```
int val = input.pop();
for (int i=0; i<K; i++)
  output.push(val);
```

MovingAverage (N)

```
int sum = 0;
for (int i=0; i<N; i++)
  sum += input.peek(i);
output.push(sum/N);
input.pop();
```

UpSamplingMovingAverage (K, N)

```
int PEEK_SIZE = K*CEIL(N/K);
int peek_buffer[PEEK_SIZE];
```

initWork

```
int i, j, val;

for (i=0; i<CEIL(N/K); i++) {
  val = input.pop();
  for (j=0; j<K; j++)
    peek_buffer[i*K+i] = val;
}
```

steadyWork

```
int i, j, sum, val;
int buffer[PEEK_SIZE+LCM(N,K)];

for (i=0; i<LCM(N,K)/K; i++) {
  val = input.pop();
  for (j=0; j<K; j++)
    buffer[PEEK_SIZE+i*K+j] = val;
}

for (i=0; i<PEEK_SIZE; i++)
  buffer[i] = peek_buffer[i];
for (i=0; i<LCM(N,K)/N; i++) {
  sum = 0;
  for (j=0; j<N; j++)
    sum += buffer[i*N+j];
  output.push(sum/N);
for (i=0; i<PEEK_SIZE; i++)
  peek_buffer[i] = buffer[i+LCM(N,K)];
```

Figure 8: Fusion of a Pipeline into a two-stage filter. In this example, the UpSampler pushes K items on every step, while the MovingAverage filter peeks at N items but only pops 1. To accommodate this peeking behavior, the fused filter maintains a `peek_buffer` for the items that reside on the channel between the two filters. In the initial work 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 temporary `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.

5.2 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.

5.2.1 Vertical Fusion

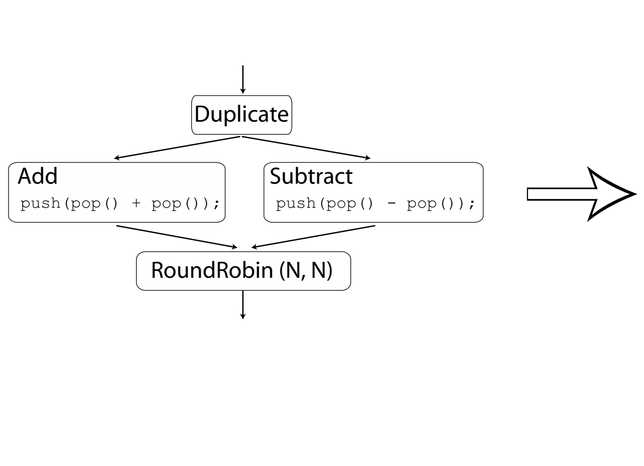
Vertical fusion describes the combination of sequential, pipelined filters into a single unit. We have developed a vertical fusion algorithm for StreamIt filters that we describe below. For the more limited domain of filters that do not contain peek statements, Proebsting and Watterson [17] present a filter fusion algorithm that interleaves the control flow graphs of adjacent nodes. However, they assume that nodes communicate via synchronous `get` and `put` operations, such that StreamIt's asynchronous peek operations and implicit buffer management fall outside the scope of their model.

Our algorithm relies on the static I/O rates of each filter to calculate a legal execution ordering for the filters being fused. Then, the fused filter simulates the execution of this schedule, inlining the code from each of the original filters and using local variables for buffering. In our current implementation, the scheduler computes only the multiplicity of each component filter in relation to the fused filter; then, the fused code is a sequence of loops that each execute a component filter for the appropriate multiplicity, buffering its results in a local array. If the multiplicity is small, then the loop can be unrolled and all array references can be replaced with scalar variables to facilitate optimization.

A subtlety of our algorithm is that the fused filter differs from the originals in that it has two distinct execution phases: one for initialization, and one for steady-state execution (see Figure 8). If any of the component filters peek at elements that they do not consume, then a separate initialization schedule is required to fill all the "peek buffers" in the pipeline (see Section 4). During this initialization, the pipeline as a whole will consume some input, but will not produce any output. Then, during the steady state schedule, the sizes of the buffers are preserved, and the pipeline both produces and consumes items. Thus, when there is peeking in the stream, there will be different I/O rates for the initialization and steady-state phases, and the fused filter will be a *two stage filter*: it executes one work function on its first invocation, and a separate work function on all subsequent invocations. Though these work functions may have different I/O rates, each rate is constant and known at compile time.

5.2.2 Horizontal Fusion

We refer to the combination of the parallel streams in a SplitJoin construct as "horizontal fusion". 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. However, horizontal fusion requires no buffering of internal items, as the parallel streams do not communicate with each other.



```
push(pop() + pop());
```

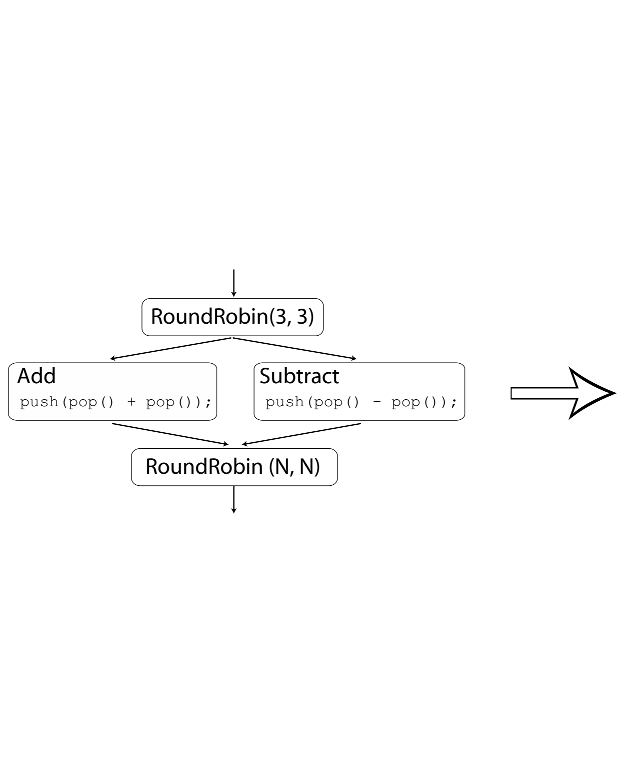
```
push(pop() - pop());
```

```
push(peek(0) + peek(1));
push(peek(0) + peek(1));
for (int i=0; i<2; i++)
  pop();
```

```
int i, j;
for (i=0; i<2; i++)
  for (j=0; j<N; j++)
    push(peek(i+2*j));

for (i=0; i<2; i++)
  for (j=0; j<N; j++)
    pop();
```

Figure 9: Fusion of a Duplicate SplitJoin construct. To fuse a SplitJoin with a Duplicate Splitter, 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.



```
push(pop() + pop());
```

```
push(pop() - pop());
```

```
int i, j, k;
for (i=0; i<2; i++)
  for (j=0; j<2; j++)
    for (k=0; k<3; k++)
      push(peek(3*i+6*j+k));

for (i=0; i<2; i++)
  for (j=0; j<2; j++)
    for (k=0; k<3; k++)
      pop();
```

```
int i;
for (i=0; i<3; i++)
  push(pop() + pop());
for (i=0; i<3; i++)
  push(pop() - pop());
```

```
int i, j, k;
for (i=0; i<3; i++)
  for (j=0; j<2; j++)
    for (k=0; k<N; k++)
      push(peek(3*N*j+N*i+k));

for (i=0; i<3; i++)
  for (j=0; j<2; j++)
    for (k=0; k<N; k++)
      pop();
```

Figure 10: 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 Reorder-RoundRobin filters that recreate the reordering of the RoundRobin nodes. In the above example, the difference the Splitter's weights, the Filter's I/O rates, and the Joiner's weights adds complexity to the reordering. Though this transformation is fully automated in the StreamIt compiler, a general formulation is beyond the scope of this paper.

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 9). 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 10). However, this transformation is invalid if any of the component Filters peeks 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 transformations when all component Filters have $peek = pop$.

A more mathematical description of our horizontal fusion transformation is given in [23].

5.3 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.

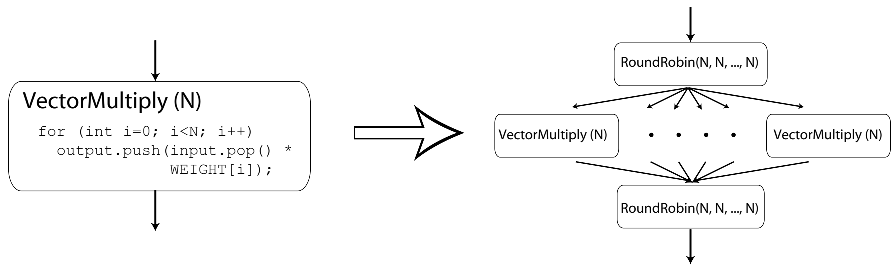
5.3.1 Vertical Fission

Some filters can be split into a pipeline, with each stage performing some 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.

5.3.2 Horizontal Fission

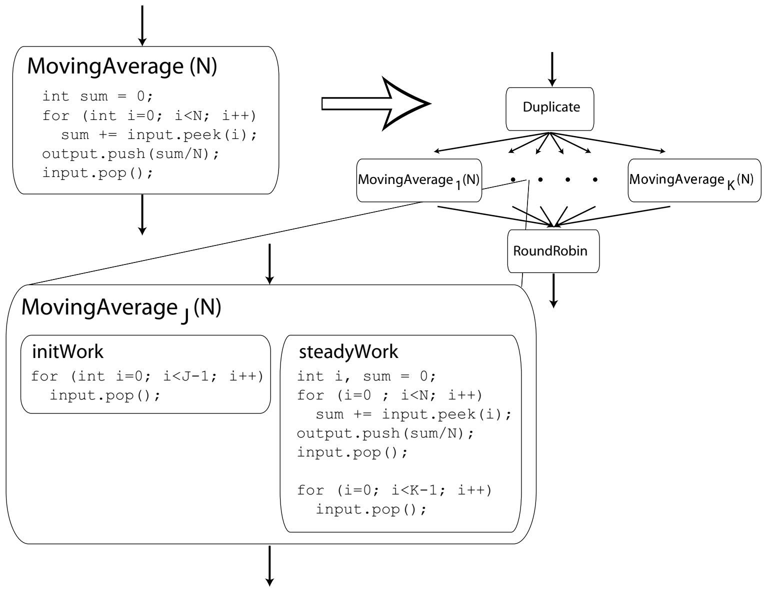
We refer to "horizontal fission" as the process of distributing a single filter across the parallel components of a SplitJoin. We have implemented this transformation for "stateless" filters–that is, filters that contain no fields that are written on one invocation of `work` and read on later invocations. Let us consider such a filter F with 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 11). 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



```
for (int i=0; i<N; i++)
  output.push(input.pop() *
              WEIGHT[i]);
```

Figure 11: 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.



```
int sum = 0;
for (int i=0; i<N; i++)
  sum += input.peek(i);
output.push(sum/N);
input.pop();
```

```
for (int i=0; i<J-1; i++)
  input.pop();
```

```
int i, sum = 0;
for (i=0 ; i<N; i++)
  sum += input.peek(i);
output.push(sum/N);
input.pop();

for (i=0; i<K-1; i++)
  input.pop();
```

Figure 12: 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 initial work 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.

consume, its code does not need to be modified in the component streams–we are just distributing the invocations of F.

2. If *peek* $>$ *pop*, then our transformation produces a SplitJoin that has a set of two-stage filters as its components (see Figure 12). 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 an initialization work 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.

5.4 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, identical stateless filters can be pushed through a Splitter or Joiner node if the weights are adjusted accordingly (Figure 13); a SplitJoin construct can be divided into a hierarchical set of SplitJoins to enable a finer granularity of fusion (Figure 14); and neighboring Splitters and Joiners with matching weights can be eliminated (Figure 15). Many of these transformations were useful in load balancing our test applications.

5.5 Summary

We have described fission and fusion transformations for a large domain of stream constructs. Our vertical fusion transformation applies to any sequence of filters, with or without peeking; our horizontal fusion transformation supports peeking if the Splitter is a Duplicate, but not if the Splitter is a RoundRobin. Our horizontal fission algorithm duplicates any stateless filter into a SplitJoin construct of arbitrary width.

The one domain that we have left unexplored is the fusion and fission of two-stage filters. If a two-stage filter (with a non-empty initial work function) results from one transformation, then no subsequent transformations can be safely performed on them. We are currently extending our transformations to support two-stage filters.

6 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 final layout assigns exactly one node in the stream graph to one computation node in the target. The layout phase assumes that the given stream graph will fit onto the computation fabric of the target and that the Filters are load balanced. Both of these requirements are satisfied by the partitioning phase described above.

The layout phase of the StreamIt compiler is implemented using simulated annealing [10]. We choose simulated annealing for its combination of performance and flexibly. 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.

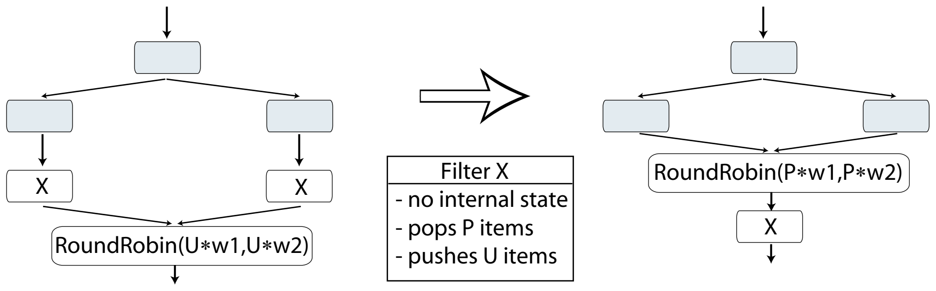


Figure 13: Filter hoisting. This transformation allows a stateless Filter to be moved across a Joiner node if its *push* value evenly divides the weights of the Joiner. This proved to be useful in load balancing the FMRadio application.

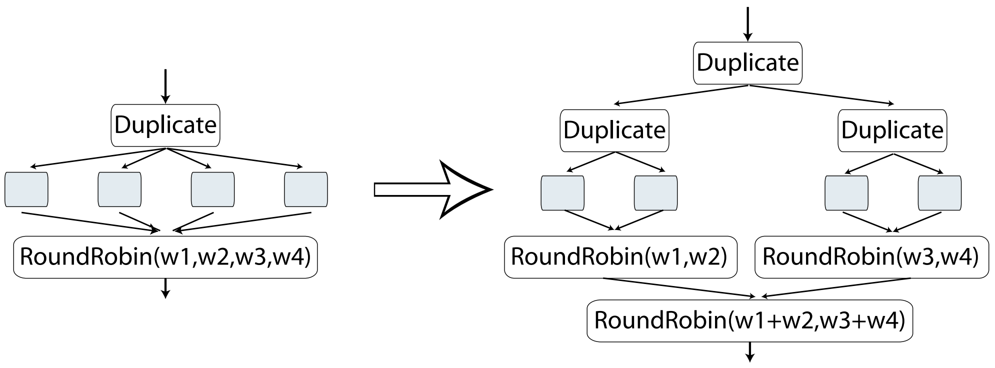


Figure 14: Breaking a SplitJoin into hierarchical units. Though our horizontal fusion algorithms work on the granularity of an entire SplitJoin, it is straightforward to transform a large SplitJoin into a number of smaller pieces, as shown here. Following this transformation, the fusion algorithms can be applied to obtain an intermediate level of granularity. This technique was employed to help load-balance the BeamFormer (see Section 9).

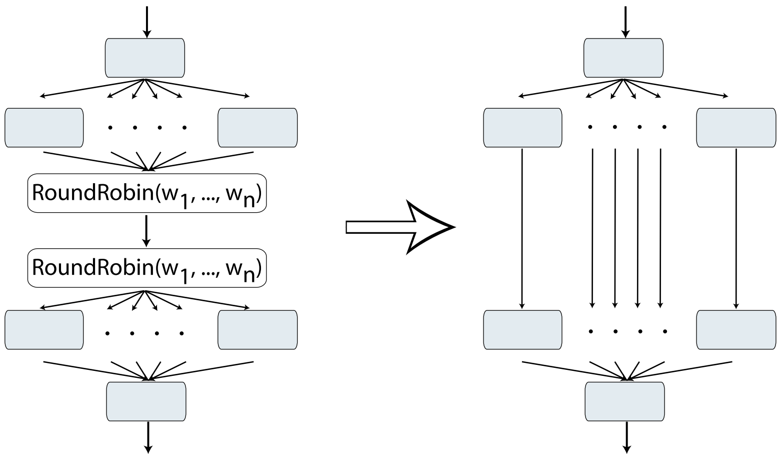


Figure 15: Synchronization removal. If a Splitter is bordering a Joiner with identical weights, the nodes can be removed and the component streams can be connected. This is especially useful in the context of libraries–many distinct components can employ SplitJoins for processing interleaved data streams, and the modules can be composed without having to synchronize all the streams at each boundary.

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 cost function can calculate the number of items that travel over a channel during each execution of the steady state. Furthermore, with knowledge of the routing algorithm, the cost function can determine 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.

6.1 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 7.1). Thus, Joiners occupy their own tile, but Splitters are integrated into the tile of another Filter or Joiner.

Due to the properties of the static network and the communication scheduler (Section 7.1), 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.

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. Our cost function evaluates a given layout of the stream graph, for a fixed value of *channels* and *route*:

$$cost(layout) = \sum_{(src,dst) \in channels} \mathbf{items}(src, dst) \cdot \Big(\mathbf{hops}(routing\text{-}path) + (2 \cdot \mathbf{synch}(routing\text{-}path))^3\Big)$$

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

In this equation, $\mathbf{items}(src, dst)$ gives the number of data items 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 $\mathbf{synch}(p)$ estimates the cost of the synchronization imposed by the path p. We calculate $\mathbf{synch}(p)$ 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.

Figure 16 illustrates how the cost metric varies over time during a run of the simulated annealing algorithm

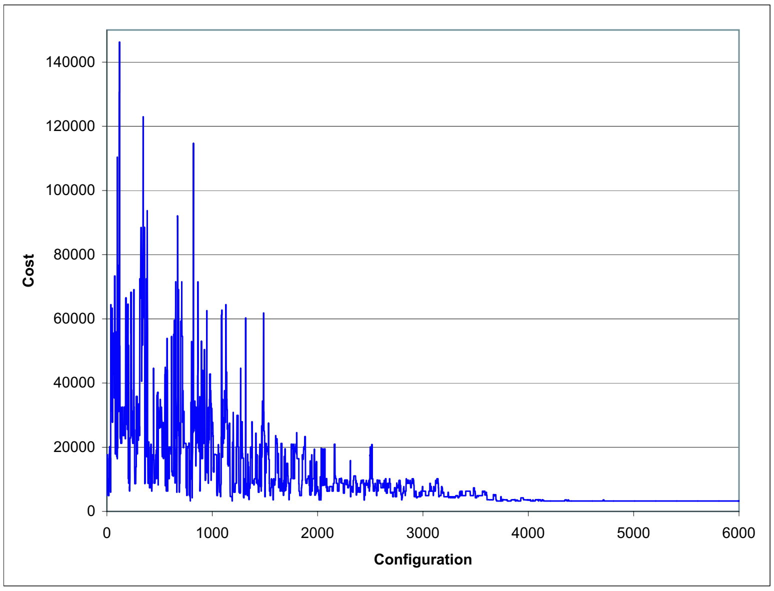


Figure 16: Estimated cost for successive configurations of the load-balanced BeamFormer layout as evaluated by the simulated annealing algorithm.

for the BeamFormer application. The figure illustrates that the cost converges to a value that is significantly lower than a random layout. For the BeamFormer, the layout determined by our algorithm has a throughput that exceeds that of a random layout by a factor of two.

7 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. The communication scheduler must avoid deadlock and starvation while trying to utilize 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 targets implementing an end-to-end, infinite FIFO abstraction, in which the scheduler needs only to 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 4, 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. We can generate ordering 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 and we will not go into a detailed explanation of deadlock here. In general,

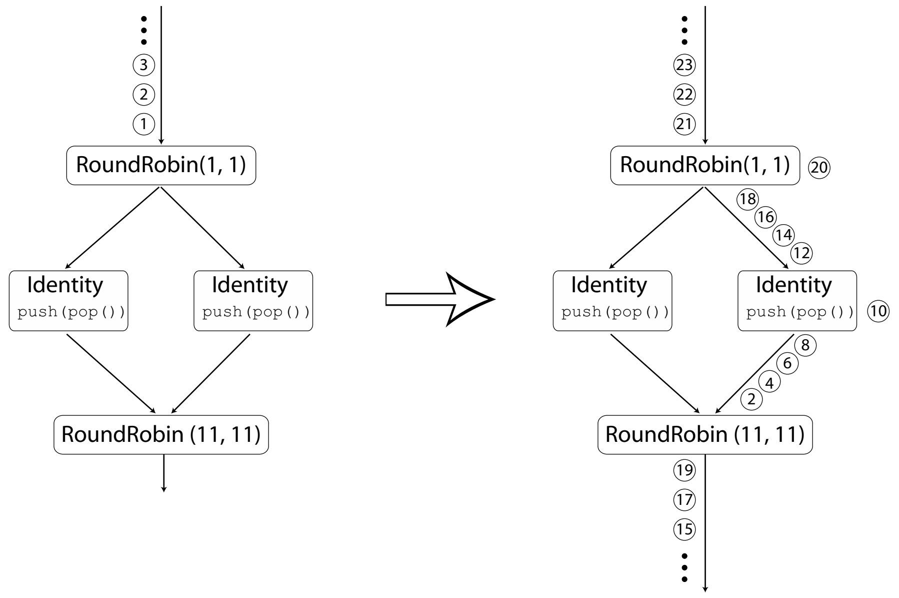
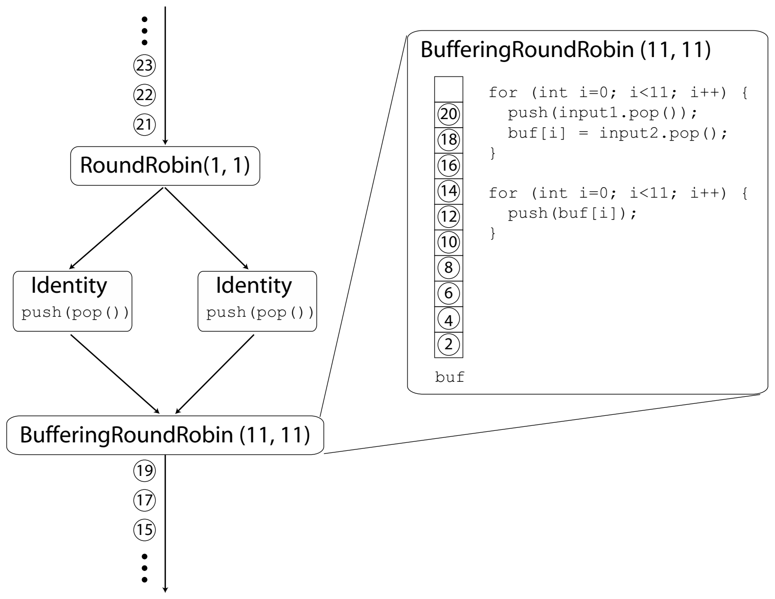


Figure 17: Example of deadlock in a SplitJoin. As the Joiner is reading items from the stream on the left, items accumulate in the channels on the right. On Raw, senders will block once a channel has four items in it. Thus, once 10 items have passed through the Joiner, the system is deadlocked, as the Joiner is trying to read from the left, but the stream on the right is blocked. The Identity filter is blocking on the send of item #10, and the Splitter is blocking on the send of item #20. If the weights on the Joiner were (10, 10), the system would be deadlock-free.



```
for (int i=0; i<11; i++) {
  push(input1.pop());
  buf[i] = input2.pop();
}

for (int i=0; i<11; i++) {
  push(buf[i]);
}
```

Figure 18: Fixing the deadlock with a buffering Joiner. The BufferingRoundRobin 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.

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.

7.1 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 3, 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.

Special care is required in the communication scheduler to avoid deadlock in SplitJoin constructs. Figure 17 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 18). 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 (Section 8), the maximum buffer size of each internal buffer is recorded.

Our current implementation of the communication scheduler is overly cautious in its deadlock avoidance. All FeedbackLoops are serialized by the communication scheduler to prevent deadlock. More precisely, the loop and body streams of each FeedbackLoop cannot execute in parallel. Crossed routes in the layout of the graph are serialized as well, forcing each path to wait its turn at the contention point.

8 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.

Benchmark	Description
FM Radio	A software-based FM Radio with equalizer.
GSM	A GSM decoder.
BeamFormer	A core component of modern radar, sonar, and communications signal processors. [11]
FFT	A 64-element FFT.
CRC	A 32-bit Cyclic Redundancy Check (CRC) Encoder/Decoder.

Table 2: Application Descriptions.

8.1 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. First we will discuss the tile processor code generation. We can directly translate the intermediate representation of most StreamIt expressions into C code. Translations for the `push(value)`, `peek(index)`, and `pop()` expressions of StreamIt require more care.

In the translation, each Filter collects the data necessary to fire in an internal buffer. Before each Filter is allowed to fire, it must receive *pop* items from its switch processor (*peek* items for the initial firing). The buffer is managed circularly and the size of the buffer is equal to the number of items peeked by the Filter. `peek(index)` and `pop()` are translated into accesses of the buffer, with `pop()` adjusting the end of the buffer, and `peek(index)` accessing the $index^{th}$ element from the end 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.

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

As described in Section 7.1, 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 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.

Lastly, to generate the instructions for the switch processor, we directly translate the switch schedules computed by the communication scheduler. The initialization switch schedule is followed by the steady state switch schedule, with the steady state schedule looped infinitely.

9 Results

We have implemented a fully-functional prototype of the StreamIt compiler for the Raw architecture. Our implementation includes all of the phases described in Section 4 as well as the optimizations, but the decision of which optimizations to apply is currently guided by the programmer.

We evaluate the StreamIt compiler for the set of applications shown in Table 2; results appear in Table 3, with MFLOPS calculated using Raw's target clock speed of 250 MHz. We show the performance of the original application, which maps each Filter in the original program to a single Raw tile. In some cases,

Benchmark	Lines of code	Unoptimized			Optimized		
		# of tiles	Throughput (per 10^5 cycles)	MFLOPS	# of tiles	Throughput (per 10^5 cycles)	MFLOPS
BeamFormer (12x4)	596	53	1.1	619	15	2.7	1470
BeamFormer (48x16)	596	209	N/A	N/A	15	0.3	836
FM Radio	511	14	34	107	13	81	258
FFT	174	37	669	12.5	16	657	12.3
GSM Decoder	1925	16	107	N/A	15	107	N/A
CRC	336	47	35.1	N/A	5	52.4	N/A

Table 3: Performance Results.

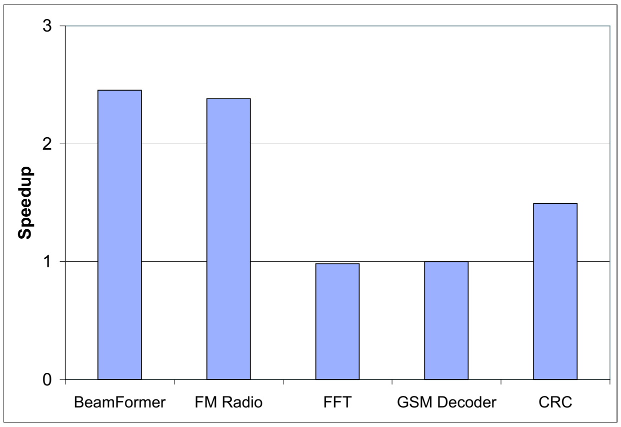


Figure 19: Speedup due to load balancing, normalized to original performance.

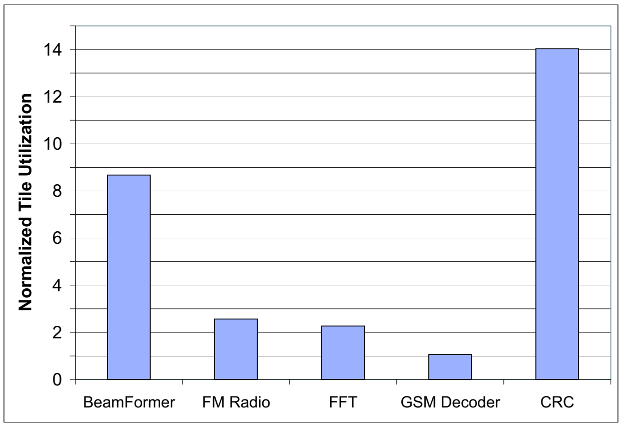


Figure 20: Utilization of a tile after load balancing, normalized to original utilization.

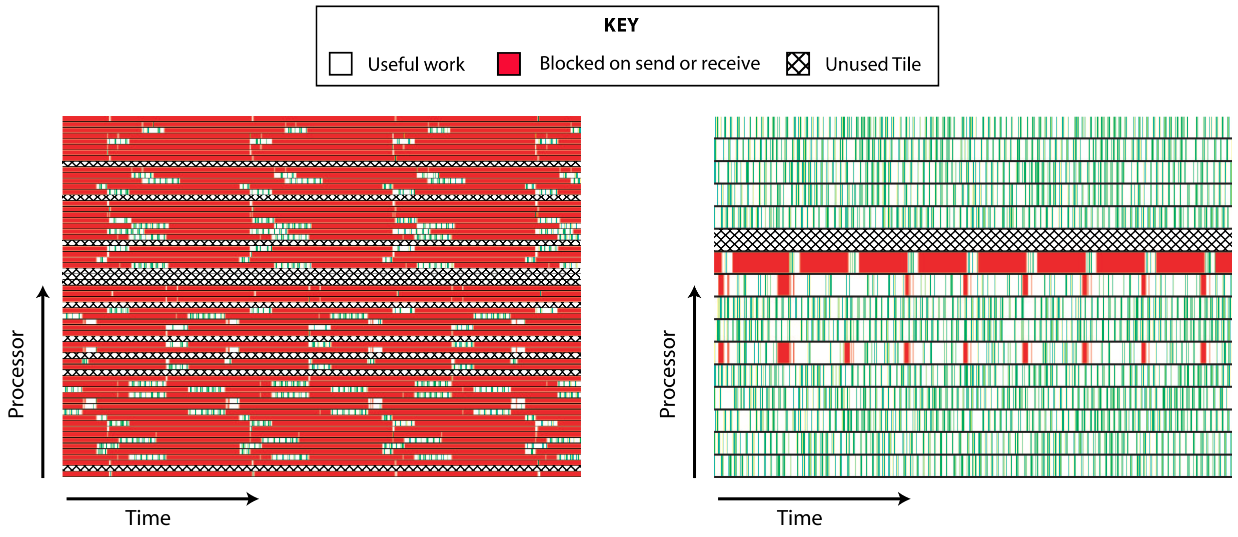
we need to use an 8x8 Raw processor in order to have enough tiles for the Filters; otherwise, we target a 4x4 configuration of Raw. To obtain the “optimized” numbers, we perform a series of fusion, fission and reordering transformations to create a load-balanced set of filters that can be mapped onto a 4x4 Raw processor. Figures 19 and 20 depict the impact of our load balancing transformations on throughput and tile utilization, respectively.

The results show that for programs with substantial computation requirements, the StreamIt compiler is able to extract good performance out of the Raw processor. For example, the BeamFormer application (see Figure 22 for source code) shows a sustained 1.47 GFLOPS rate, after a 145% improvement due to our optimizations. Figures 23 and 24 give the original and optimized configurations of the BeamFormer's stream graph, and Figure 21 illustrates the impact of our load balancing optimizations .

10 Related Work

The Transputer architecture [1] shares many similarities with Raw. A Transputer system is either an array or a grid of tiles, where neighbors are interconnected with unbuffered point-to-point links. The programming language used for the Transputer is Occam [8]: a streaming language similar to CSP [7]. 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 [14].

DSPL is a language with simple filters interconnected in a flat acyclic graph using unbuffered channels [15]. Unlike the Occam compiler for the Transputer, the DSPL compiler automatically maps the graph into the



(a) Original.

(b) Load-balanced.

Figure 21: Execution traces for the (a) original (on 64 tiles) and (b) load-balanced (on 16 tiles) partitionings of the BeamFormer. Graphs are shaded as in Figure 5.

```
class BeamFormer extends Pipeline {

   void init(numChannels, numBeams) {
      add(new SplitJoin() {
         void init() {
            setSplitter(Null());
            for (int i=0; i<numChannels; i++) {
               add(new InputGenerate(i));
               add(new FIRFilter1(64));
               add(new FIRFilter2(16));
            }
            setJoiner(RoundRobin());
          }});

      add(new SplitJoin() {
         void init() {
            setSplitter(Duplicate());
            for (int i=0; i<numBeams; i++) {
               add(new VectorMult(i));
               add(new FIRFilter3(64));
               add(new Magnitude());
               add(new Detect(i));
            }
            setJoiner(Null());
         }});
   }
}
```

Figure 22: Top-level StreamIt code for the BeamFormer.

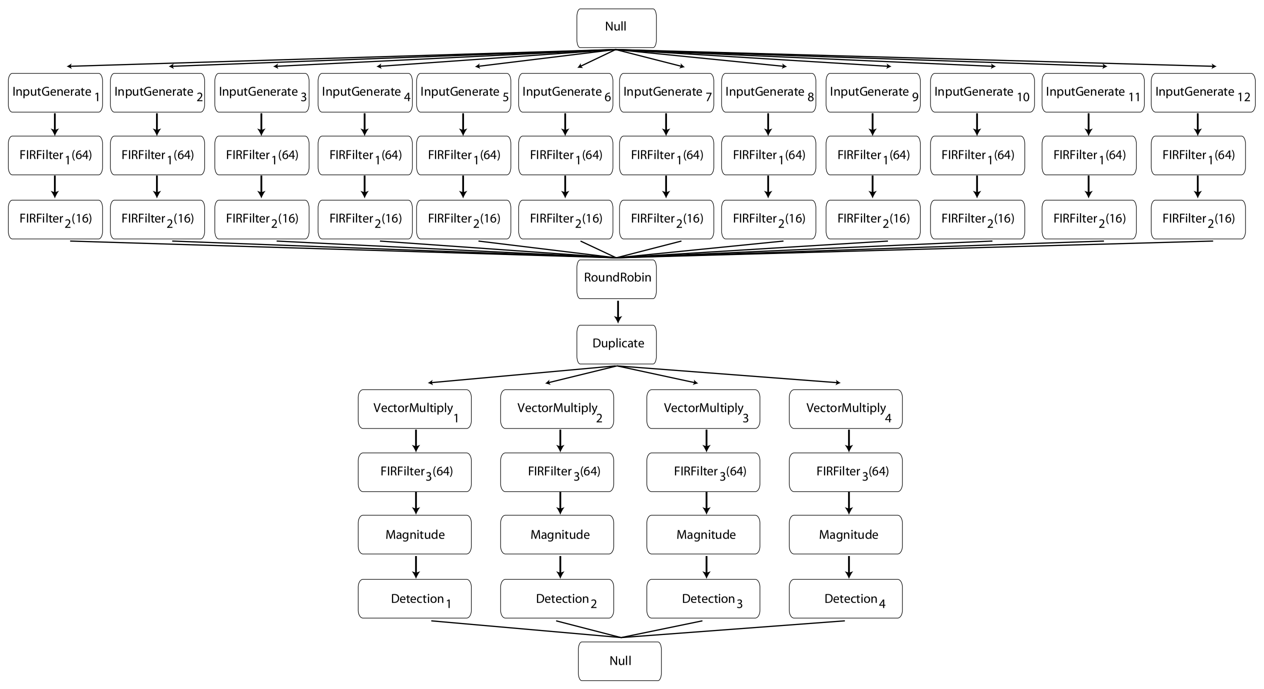


Figure 23: Stream graph of the original 12x4 BeamFormer. The 12x4 BeamFormer has 12 channels and 4 beams; it is the largest version that fits onto 64 tiles without filter fusion.

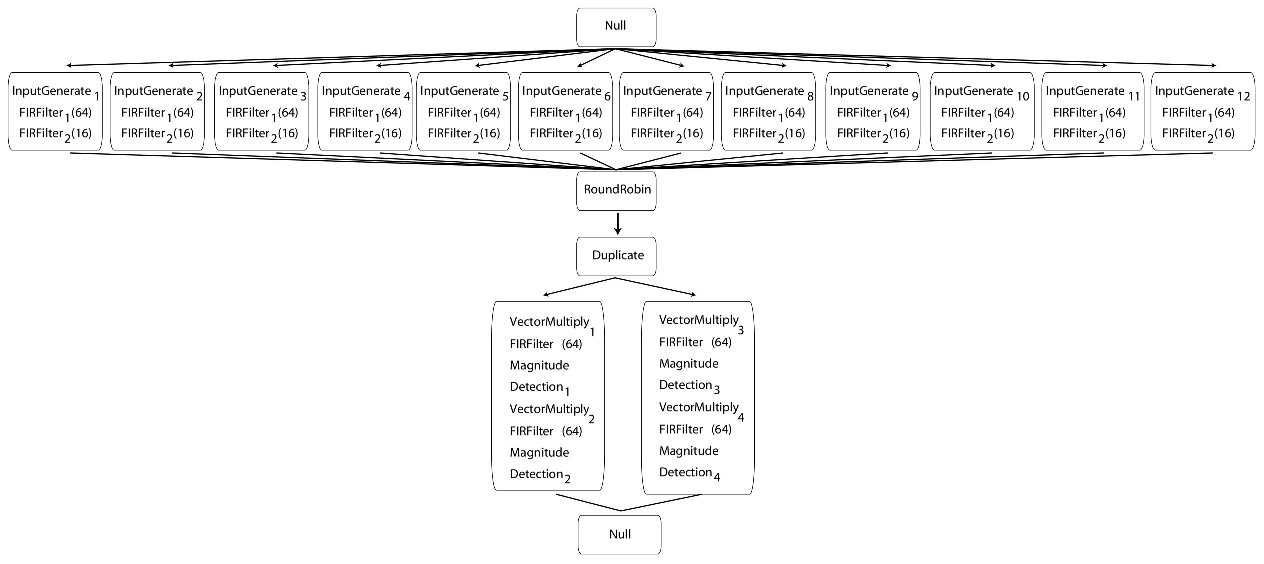

Null
InputGenerate 1 FIRFilter 1 (64) FIRFilter 2 (16)
InputGenerate 2 FIRFilter 1 (64) FIRFilter 2 (16)
InputGenerate 3 FIRFilter 1 (64) FIRFilter 2 (16)
InputGenerate 4 FIRFilter 1 (64) FIRFilter 2 (16)
InputGenerate 5 FIRFilter 1 (64) FIRFilter 2 (16)
InputGenerate 6 FIRFilter 1 (64) FIRFilter 2 (16)
InputGenerate 7 FIRFilter 1 (64) FIRFilter 2 (16)
InputGenerate 8 FIRFilter 1 (64) FIRFilter 2 (16)
InputGenerate 9 FIRFilter 1 (64) FIRFilter 2 (16)
InputGenerate 10 FIRFilter 1 (64) FIRFilter 2 (16)
InputGenerate 11 FIRFilter 1 (64) FIRFilter 2 (16)
InputGenerate 12 FIRFilter 1 (64) FIRFilter 2 (16)
RoundRobin
Duplicate
VectorMultiply 1 FIRFilter (64) Magnitude Detection 1 VectorMultiply 2 FIRFilter (64) Magnitude Detection 2
VectorMultiply 3 FIRFilter (64) Magnitude Detection 3 VectorMultiply 4 FIRFilter (64) Magnitude Detection 4
Null

Figure 24: Stream graph of the load-balanced 12x4 BeamFormer. 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 21 shows the benefit of these transformations.

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 [18]. 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 [9]. 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.

The iWarp system [3] is a scalable multiprocessor with configurable communication between nodes. In iWarp, one can set up FIFO channels for communicating between non-neighboring tiles. However, reconfiguring the communication channels is more coarse-grained and has a higher cost than on Raw, where each cycle can be route items to a different location. ASSIGN [16] 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.

A large number of programming languages have included a concept of a stream; see [20] for a survey. However, the compilers for these languages have focused only on efficient sequential execution of the program.

11 Conclusion

In this paper, we describe the StreamIt compiler and a backend for the Raw architecture. The stream graph of a StreamIt program exposes the data communication pattern to the compiler while the lack of global synchronization frees the compiler to radically reorganize the program for efficient execution on the underlying architecture. The StreamIt compiler demonstrates the power of this flexibility by totally reorganizing large programs for better load balancing. We were able to map many programs onto the Raw processor and to obtain good performance.

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 we can map a high-level stream program, written to reflect the composition of the application, onto Raw and achieve good processor utilization and load balance, leading to a factor of two speedup on two applications.

Unlike all previous streaming languages, the structured streams of StreamIt makes it possible for us to approach the optimization and parallelization problems very systematically. It enables us to define multiple optimizations–targeting different constructs and requirements–and to compose them in a hierarchical manner.

The ability to do global transformations across multiple filters, that may have originated from very different parts of the application, makes it possible for the compiler to find optimization opportunities that may elude even an experienced programmer. Such capabilities enable programmers to write portable streaming applications and map them efficiently onto any given architecture. This has the potential to create a programming standard for the emerging class of communication-exposed architectures. The StreamIt compiler takes a fist step towards this goal.

References

[1] *The Transputer Databook*. Inmos Corporation., 1988.

[2] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996. 189 pages.

[3] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing*, pages 330–339, 1988.

[4] V. Gay-Para, T. Graf, A.-G. Lemonnier, and E. Wais. Kopi Reference manual. http://www.dms.at/kopi/docs/kopi.html, 2001.

[5] T. Gross and D. R. O'Halloron. *iWarp, Anatomy of a Parallel Computing System*. The MIT Press, Cambridge, MA, 1998.

[6] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. In *Proceedings of the IEEE*, pages 490–504, April 2001.

[7] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[8] Inmos Corporation. *Occam 2 Reference Manual*. Prentice Hall, 1988.

[9] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, and B. Towles. Stream scheduling. In *Proceedings of the 3rd Workshop on Media and Streaming Processors*, pages 101–106, 2001.

[10] S. Kirkpatrick, J. C.D. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[11] J. Lebak. Polymorphous Computing Architecture (PCA) Example Applications and Description. External Report, Lincoln Laboratory, Massachusetts Institute of Technology, August 2001.

[12] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. P. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Architectural Support for Programming Languages and Operating Systems*, pages 46–57, 1998.

[13] 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.

[14] D. May, R. Shepherd, and C. Keane. Communicating Process Architecture: Transputers and Occam. *Future Parallel Computers: An Advanced Course, Pisa, Lecture Notes in Computer Science*, 272:35–81, June 1987.

[15] A. Mitschele-Thiel. Automatic Configuration and Optimization of Parallel Transputer Applications. *Transputer Applications and Systems '93*, pages 1052–1067, 1993.

[16] D. R. O'Hallaron. The assing parallel program generator. Carnegie Mellon Technical Report CMU-CS-91-141, 1991.

[17] T. A. Proebsting and S. A. Watterson. Filter Fusion. In *Symposium on Principles of Programming Languages*, pages 119–130, 1996.

[18] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *International Symposium on Microarchitecture*, pages 3–13, 1998.

[19] K. Sankaralingam, R. Nagarajan, S. Keckler, and D. Burger. A Technology-Scalable Architecture for Fast Clocks and High ILP. The University of Texas at Austin, Department of Computer Sciences Technical Report TR-01-02, 2001.

[20] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7):491–541, 1997.

[21] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, J.-W. Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. S. M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. IEEE Micro vol 22, Issue 2, 2002.

[22] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction, to appear*, Grenoble, France, 2002.

[23] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe. StreamIt: A Compiler for Streaming Applications. MIT-LCS Technical Memo LCS-TM-622, Cambridge, MA, 2001.

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