A Stream Compiler for Communciation-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. Our compiler targets the Raw processor, and we present results using the cycle-accurate Raw simulator. 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 consider this work to be a first step towards a portable programming model for communication-exposed architectures.

We have a fully functioning compiler that parallelizes StreamIt applications for Raw, with a host of optimizations implemented. We will show that our optimizations are able to increase the performance by up to 350% and the compiled code was able to sustain 2.3 GFLOPS on a 250 MHz Raw processor.

1 Introduction

As the size of the transistor becomes smaller and smaller, the next generation of microprocessors will have to be conscious of wire delays as the fundamental barrier to performance. Recently, a number of architectures have been emerging that address the problem of wire delay by replicating the basic processing unit and exposing the communication between units to a software layer (*e.g.*, Raw [4], SmartMemories [12], TRIPS [18]). 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 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 a very 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, a high level stream language that aims to be portable across communication-exposed machines [21]. 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 [4], 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 technical contributions:

- Filter fusion optimizations, horizontal fusion and vertical fusion, with and without peeking.
- A filter fission optimization.
- A graph reodering 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 implementation of an end-to-end compiler for parallelizing stream 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

```
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(ROUND_ROBIN());
    }});
    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.

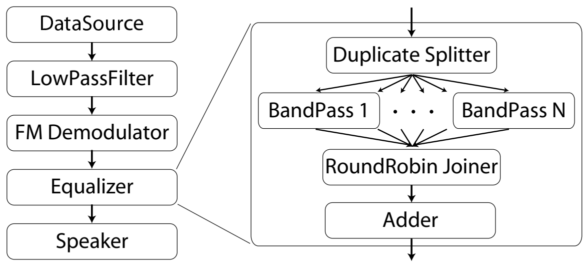
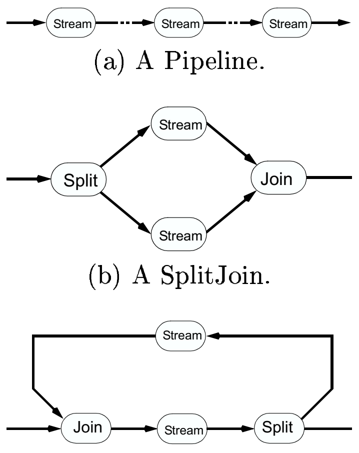


Figure 2: Block diagram of the FM Radio.



(a) A Pipeline.

(b) A SplitJoin.

(c) A FeedbackLoop.

Figure 3: Stream structures supported by StreamIt.

7 describe our algorithms for partitioning, layout, and communication scheduling, respectively. 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 [21]. 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. Designing a cleaner syntax and considering dynamically varying rates will be the subject of future work.

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 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.

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($w_1, \dots, w_n$), 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 round robin 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`.

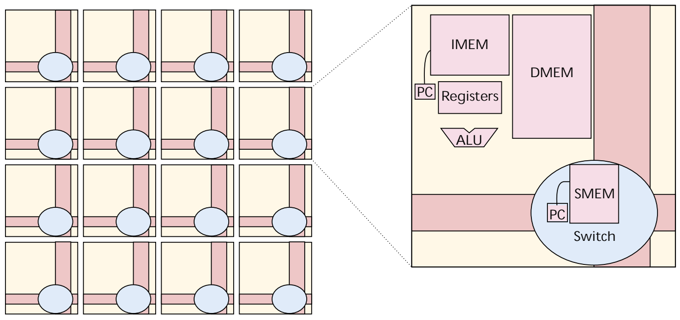


Figure 4: A block diagram of the Raw architecture.

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 [23][20] addresses the wire delay [7] problem 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 like 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 the architectural analog to the wiring resources of the processor: four, on-chip 32-bit full-duplex, flow-controlled, point-to-point networks. On the edges of the array, these networks are connected via logical channels [6] 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 [20] that allows instructions to use the networks and the register files interchangeably. In other words, a single instruction can read up to two values out of the networks, compute on them, and send the result out into 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 sequences through a compile-time programmed, 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 a 64-bit VLIW word that includes a basic instruction that operates on a network value or one of the local 4-element register file values (conditional branch with/without decrement, move, and nop). The instruction also has 13 fields that specify the connections between each output of the two crossbars and the network input FIFOs that store values that have arrived from the neighbor tiles or the local compute processor. The input and output possibilities for each crossbar are: North, East, West, South, 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 [11]. 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. This tile employs as compute processor

Stage	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 stream graph.
Code generation	Generates code for the tile and switch processors.

Table 1: Stages of the StreamIt compiler.

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 procesosr's pipelined single-precision FPU operations have a latency of 4 cycles, and the integer multiplier has a latency of 2 cycles.

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 [5]. We translate the KOPI syntax tree into the StreamIt IR (SIR) that encapsulates the hierararchical 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 stages 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 $peek > pop$, since a periodic schedule would return the graph to its initial configuration after every cycle, but it would be impossible to return to the startup configuration if a filter leaves $peek - pop$ items on its channel after every firing.

In the StreamIt compiler, the initialization schedule is constructed via symbolic execution of the stream graph, until each filter has $peek - pop$ 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 [22]. In this paper, we use a simple hierarchical scheduler that constructs a Single Appearance Schedule [2] (SAS) 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 stat 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.

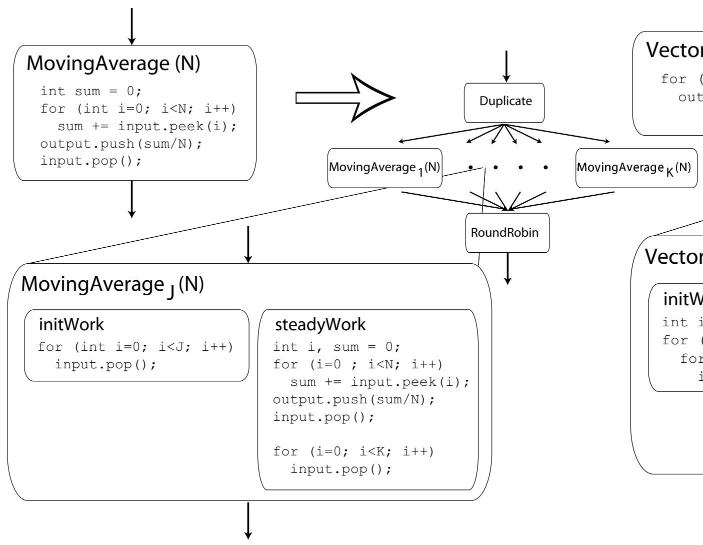
5 Partitioning

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 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, but for streaming programs, the maximum improvement in throughput is given by the following expression:

$$\textit{Maximum speedup}(w, c) = \frac{\sum_1^m c_i \cdot w_i}{MAX_i(c_i \cdot w_i)}$$

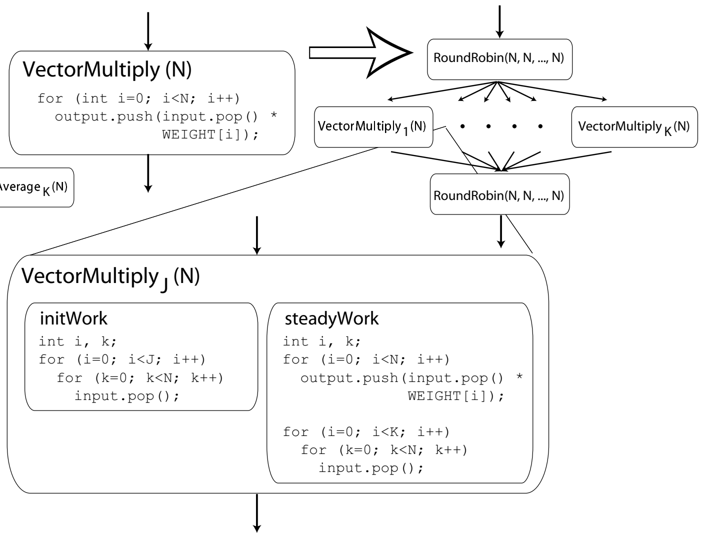


```
MovingAverage (N)
  int sum = 0;
  for (int i=0; i<N; i++)
    sum += input.peek(i);
  output.push(sum/N);
  input.pop();
```

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

```
steadyWork
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; i++)
  input.pop();
```

Figure 5: Fission of filter that peeks.

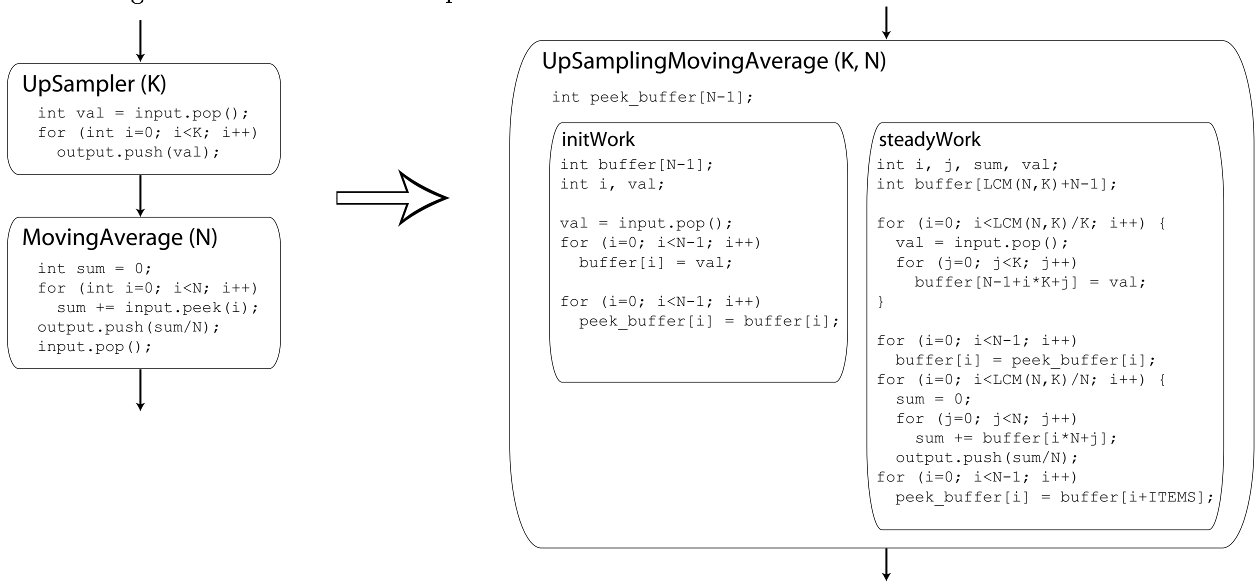


```
VectorMultiply (N)
  for (int i=0; i<N; i++)
    output.push(input.pop() *
                WEIGHT[i]);
```

```
initWork
int i, k;
for (i=0; i<J; i++)
  for (k=0; k<N; k++)
    input.pop();
```

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

for (i=0; i<K; i++)
  for (k=0; k<N; k++)
    input.pop();
```

Figure 6: Fission of filter that does not peek.



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

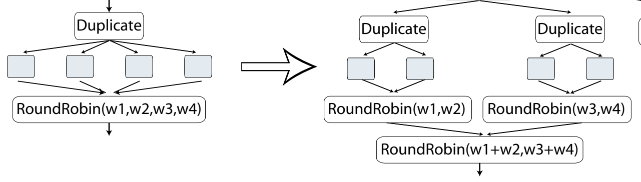
```
MovingAverage (N)
  int sum = 0;
  for (int i=0; i<N; i++)
    sum += input.peek(i);
  output.push(sum/N);
  input.pop();
```

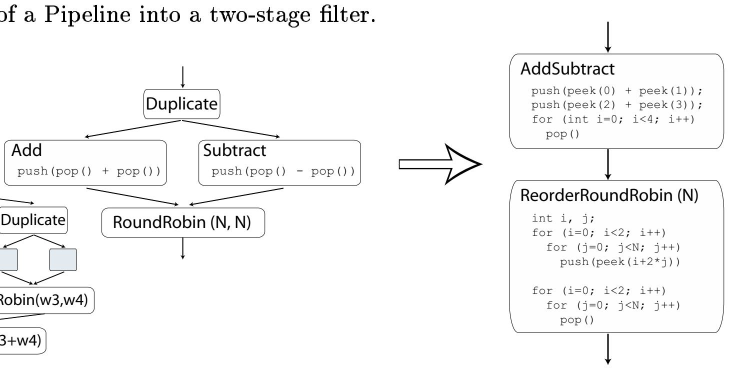
```
UpSamplingMovingAverage (K, N)
 int peek_buffer[N-1];
```

```
initWork
int buffer[N-1];
int i, val;

val = input.pop();
for (i=0; i<N-1; i++)
  buffer[i] = val;

for (i=0; i<N-1; i++)
  peek_buffer[i] = buffer[i];
```

```
steadyWork
int i, j, sum, val;
int buffer[LCM(N,K)+N-1];

for (i=0; i<LCM(N,K)/K; i++) {
  val = input.pop();
  for (j=0; j<K; j++)
    buffer[N-1+i*K+j] = val;
}

for (i=0; i<N-1; 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<N-1; i++)
  peek_buffer[i] = buffer[i+ITEMS];
```

Figure 7: Fusion of a Pipeline into a two-stage filter.





```
Add
 push(pop() + pop())
```

```
Subtract
 push(pop() - pop())
```

```
AddSubtract
 push(peek(0) + peek(1));
 push(peek(2) + peek(3));
 for (int i=0; i<4; i++)
   pop()
```

```
ReorderRoundRobin (N)
 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 8: Breaking a SplitJoin into hierarchical units.

Figure 9: Fusion of a SplitJoin construct.

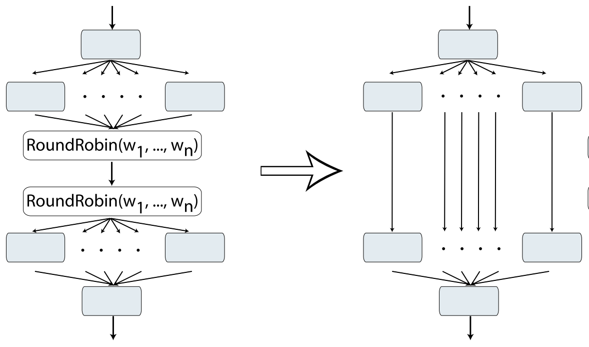


Figure 10: Synchronization removal.

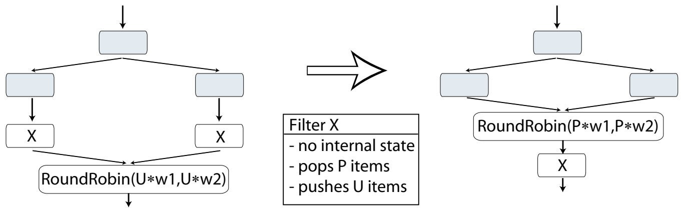


Figure 11: Filter hoisting.

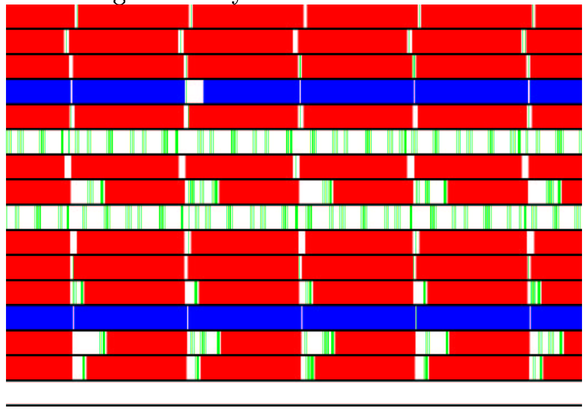


Figure 12: The execution trace for BeamFormer before load balancing. The x axis is time, and the y axis is processor. The dark bands indicate periods where processors are blocked waiting to receive an input or send an output. Light regions indicate periods of useful work.

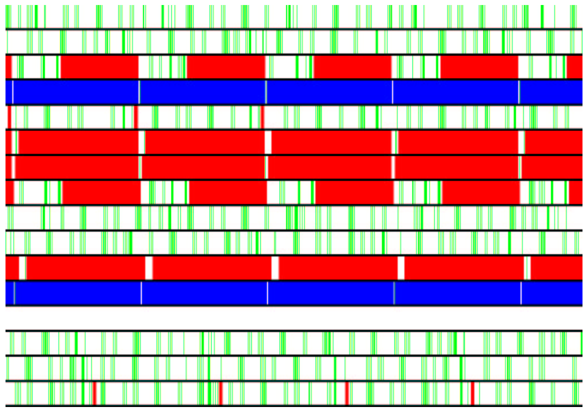


Figure 13: The execution trace for the load-balanced partitioning of BeamFormer. See Figure 12 for an explanation.

where $w_1 \ldots w_m$ denote the amount of work in each of the m 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, then the performance will suffer by 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 BeamFormer application [10] to illustrate the partitioning process (see Section 9 for more information about the BeamFormer). Figure 12 illustrates a blocking diagram for the original partitioning of the application. The dark bands indicate where a processor is waiting to send or receive an item, while the light areas indicate periods of useful work. With the filters partitioned in the original configuration, most of the processors spend their time waiting on two filters in the stream (they each block a path of a SplitJoin). The StreamIt compiler performs four vertical fusion transformations and two four-way horizontal fission transformations to split up the computationally-intensive nodes while combining the less demanding nodes. The resulting blocking diagram appears in Figure 13, illustrating approximately four times as much processor utilization as the original. In the following sections, we describe these transformations in more detail.

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.

5.3 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 [16] 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 7). 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. 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.

5.4 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. 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. Also, for Duplicate splitters, the `pop` expressions in component filters need to be converted to `peek` expressions so that items are not consumed before subsequent filters can read them.

5.5 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.

There are many types of fission transformations. We have implemented a data-parallel transformation for stateless filters that places a duplicate of the filter on each path of an n-way SplitJoin (see Figures 5-6). By "stateless" we mean that the filter contains 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*. Our transformation produces a SplitJoin that has a set of two-stage filters as its components. The i'th component has a steady-state work function that is exactly the same as in F, and an initialization work function that simply pops $(i-1) * pop$ items from the input stream (to account for the consumption of previous filters). If $peek = pop$, the splitter is a RoundRobin that routes *pop* elements to each component stream. However, if $peek > pop$, then the splitter is a Duplicate, and each stream contains a decimator to consume items that are unused by the component filter. In either case, the joiner is a RoundRobin that has a weight of *push* for each input.

Instead of duplicating the entire contents of a filter, 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.

5.6 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 11); a SplitJoin construct can be divided into a hierarchical set of SplitJoins to enable a finer granularity of fusion (Figure 8); and neighboring splitters and joiners with matching weights can be eliminated (Figure 10).

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 each Filter is 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 [9]. We chose to use simulated annealing for its combination of performance and flexibly. To retarget the layout phase of the compiler, we simply supply the annealing algorithm with three elements: the cost function, a perturbation function, and the set of legal layouts. Furthermore, for most tiled targets these three functions could be reused.

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. These properties can be used to tailor the cost function to target architecture.

Depending on the communication model of the target architecture, these metrics can be extremely important in generating a good layout. For architectures with non-uniform communication, the layout phase can use this knowledge to take advantage of the specific properties of the architecture.

6.1 Layout on Raw

For Raw, the layout phase maps nodes in the stream graph to the tile processors. The layout phase does not assign each node in the stream graph to a processor. Splitter nodes are folded into their neighboring upstream Filter and neighboring joiners are collapsed into a single tile (see Section 7.1). Again, all nodes needing assignment are mapped to exactly one tile processor.

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.

The cost function is the most important parameter of the simulated annealing algorithm. After some experimentation, we have arrived at the following cost function:

$$cost(channels) = \sum_{c \in channels} items(c) \cdot \left(hops(c) + (2 \cdot synch(c))^3\right)$$

Where $items(c)$ gives the number of data items that traverse the channel c for each steady state execution. $hops(c)$ gives the number of intermediate tiles traversed on the route of the channel. Finally, $synch(c)$ attempts to give a cost to the added synchronization imposed by this channel assignment. $synch(c)$ calculates the number of tiles that are assigned to Filters plus the number of tiles involved in routing *other* channels on the route.

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(). 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.

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, the scheduler needs only to decide to whom to send an item and from whom to receive an item. Information 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. We can 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, 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 Raw's Communication Scheduler

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 of each Filter, instead it assumes that each Filter fires instantaneously. The flow-control of the static network allows this relaxation. Instead the simulator operates on the granularity of a data item. Modeling the path each data item follows.

We simulate the graph for both an initialization schedule and a steady state schedule, both are described in Section 4. A push model schedule is used for both phases. We define a push model schedule as a schedule that advances computation on the furthest downstream node in the stream graph at each step. The push model schedule allows the implementationto disregard the FIFO buffers that connect each neighboring node in the stream graph. This is because data items will not be accumulating at the source, dest, or intermediate nodes. Each destination will consume the data item as it is produced (modulo the latency of routing).

To assure deadlock free execution of Joiner nodes, we must allow the Joiner nodes to receive data in the firing order of its upstream Filters. So, the Joiner must buffer data until it can fire in the order specified by its incoming weights. First, remember that we collapse all neighboring Joiners into a single Joiner node. We create an internal buffer for each incoming channel of the collapsed Joiner. During execution of the simulation, we record the order of receives to and send from these internal buffers.

To create this Joiner buffer schedule, the simulator must keep track which internal buffer can send data. The order can be calculated from the weights of the joiner (as each buffer corresponds to an upstream Filter). When the current buffer in this order has more than one item in it, the Joiner can fire. The simulator can then determines the next internal buffer to send data. To faciliate code generation (Section 8), the maximum buffer size of each internal buffer is recorded.

Our current implemenation 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 the 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.

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 and Statements into C code. Translations for the `push(value)`, `peek(i)`, 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 this Filter. `peek(i)` and `pop()` are translated into accesses of the buffer, with `pop()` adjusting the end of the buffer, and `peek(i)` accessing the i^{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.

Next, we generate code for the collapsed Joiner nodes. As described in Section 7.1, the communication scheduler computes a internal buffer schedule for each collapsed Joiner. This schedule exactly describes the order in which to send and receive data items from within the Joiner. The schedule is annotated

Benchmark	Description
FM Radio	A software-based FM Radio with equalizer
GSM	A GSM Decoder
Beamformer	code of modern radar, sonar, and communications signal processors [10]
FFT	A 64-element FFT filter
CRC	A 32-bit Cyclic Redundancy Check (CRC) Encoder/Decoder

Table 2: Application Characteristics

Benchmark	Lines of code	Unoptimized			Optimized		
		# of filters	Throughput cycles / output	MFLOPS	# of filters	Throughput cycles / output	MFLOPS
BeamFormer	596	15	125000	525	16	35714	2300
FM Radio	511	14	4466	141.5	14	1742	363.5
FFT	174	31	156	35.75	15	3601	4.25
GSM Decoder	1925	14	1706	N/A	16	1656	N/A
CRC	336	47	2847	N/A	5	1909	N/A

Table 3: Performance Results

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 in structions 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; the compiler phases are described in Table 1. We have implemented all the phases of the compiler as well as the optimizations except for the optimization selection, which is currently done by the programmer.

In the Table 3, we evaluate the effectiveness of StreamIt compiler by compiling the above applications on to the Raw processor. The executables are simulated on either a 4x4 or an 8x8 Raw processor using Raw's cycle accurate simulator. We show the performance of the original application, which is run on a processor configuration with sufficent tiles so that each filter can be mapped to an individual tile. Next we perform a series of fusion, fisson and reindexing transformations to create a load balanced set of filters that can be mapped onto a 4x4 Raw processor.

The results show that for program with substantial computation requirements such as the BeamFormer, the StreamIt compiler is able to extract exteremly good performance out of the Raw processor. A sustained 2.3 GFLOPS rate after 350% improvement due to our optimizations. However programs such as FFT, which was created to experiment with the languatge features of StreamIt, has very little work per filter. We were

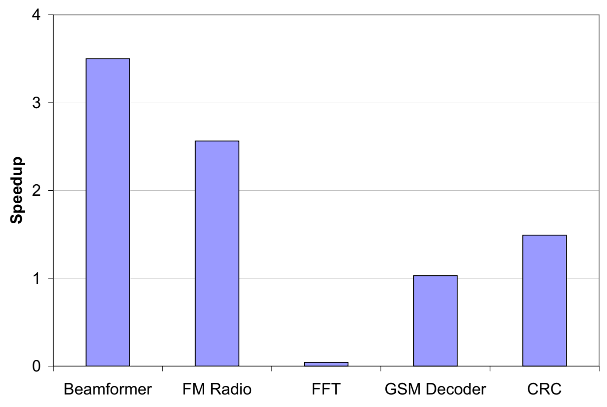


Figure 14: The speedup due to optimizations normalized to original performance.

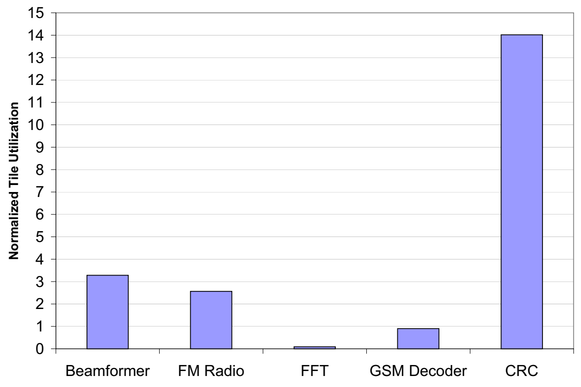


Figure 15: Utilization of a tile ofter optimization normalized to the original utilization.

unable to amortize the communication overhead for FFT, leading to a slowdown. Better tile code generation and traditional optimizations may help in this.

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, Occam, is a streaming language similar to CSP. 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 [13].

DSPL is a language with simple filters interconnected in a flat acyclic graph using unbuffered channels [14]. Unlike the Occam compiler for the Transputer, the DSPL compiler automatically maps the graph into the available resouces 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 [17]. 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 [8]. 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, the reconfiguration is more coarse-grained and has a higher cost than on Raw, where each cycle can be route items to a different location. ASSIGN [15] 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 more limited opportunities for global transformations such as fission and reordering as in StreamIt.

A large number of programming languages have included a concept of a stream; see [19] for a survey. However, the compilers for these languages have focused on efficient sequential execution of the program.

11 Conclusion

In this paper, we describe the StreamIt compiler 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 reoganize the program for efficient execution on the underline architecture. The StreamIt compiler demonstrates the power of this flexibility by totally reoganizing large programs for better load balance. We were able to map many of programs on to the Raw processor and obtain very good performance, which can go as high as 2.3 GFLOPS.

We introduce a collection of optimizations, vertical and horizontal filter fusion, vertical and horizontal filter fission and filter reordering transformations, 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 three 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 in a very systermatic manner. It enables us to define multiple optimizations – targetting different constructs and requirements – and to compose them them in a hirearchical 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 ellude even an experience programmer. Such capabilities enables the programmers to write protable streaming applications and map them efficiently onto any given architecture. This has the potential of creating a programming standard for emerging 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 '88*, pages 330–339, 1988.

[4] E. Waingold *et al.* Baring it all to Software: The Raw Machine. MIT-LCS Technical Memo 709, Cambridge, MA, 1997.

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

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

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

[8] 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.

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

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

[11] 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.

[12] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular recongurable architecture, 2000.

[13] 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.

[14] A. Mitschele-Thiel. Automatic Configuration and Optimization of Parallel Transputer Applications. pages 1052–1067, 1993.

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

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

[17] 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.

[18] 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.

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

[20] 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.

[21] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications, 2002.

[22] 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.

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