A Stream Compiler for Communication-Exposed Architectures

Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli,
Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann,
David Maze, and Saman Amarasinghe

MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139

{mgordon, thies, karczma, jasperln, saadat, aalamb, clleger, 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, SmartMemories, TRIPS). However, for their use to be widespread, it will be necessary to develop compiler technology that enables a portable, high-level language to execute efficiently across a range of wire-exposed architectures.

In this 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, some analysis is needed to adapt a stream program to a software-exposed processor. We describe a partitioning algorithm that employs fission and fusion transformations to adjust the granularity of a stream graph, a layout algorithm that maps a stream graph to a given network topology, and a scheduling strategy that generates a fine-grained static communication pattern for each computational element.

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

[*] This version of the paper is dated August 9, 2002 and corrects some typos in the ASPLOS £nal copy. More information on the StreamIt project is available from `http://compiler.lcs.mit.edu/streamit`

ASPLOS X 10/02 San Jose, CA, USA

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 [35], SmartMemories [23], TRIPS [29]). These machines are especially well-suited for streaming applications that have regular communication patterns and widespread parallelism.

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

In this paper, we describe a compiler for StreamIt [33], a high level stream language that aims to be portable across communication-exposed machines. StreamIt contains basic constructs that expose the parallelism and communication of streaming applications without depending on the topology or granularity of the underlying architecture. Our current backend is for Raw [35], a tiled architecture with fine-grained, programmable communication between processors. However, the compiler employs three general techniques that can be applied to compile StreamIt to machines other than Raw: 1) partitioning, which adjusts the granularity of a stream graph to match that of a given target, 2) layout,

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

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

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

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

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

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

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

Figure 1: Parts of an FM Radio in StreamIt.

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.

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. Section 10 considers related work, and Section 11 contains our conclusions.

2. THE STREAMIT LANGUAGE

StreamIt is a portable programming language for high-performance signal processing applications. The current version of StreamIt is tailored for static-rate streams: it requires that the input and output rates of each filter are known at compile time. In this section, we provide a very brief overview of the syntax and semantics of StreamIt, version 1.1. A more detailed description of the design and rationale for StreamIt can be found in [33], which describes version

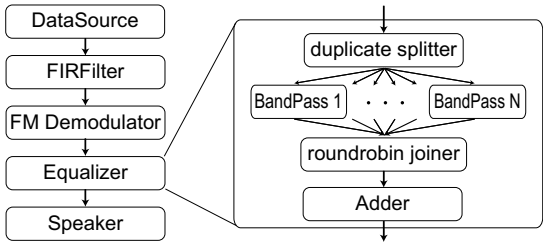
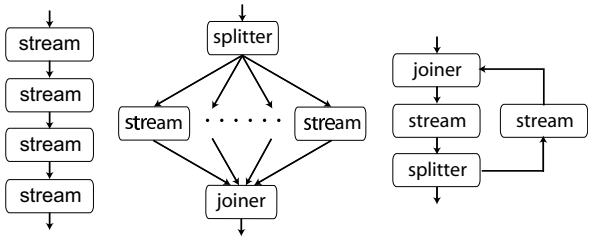


Figure 2: Block diagram of the FM Radio.



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

Figure 3: Stream structures supported by StreamIt.

1.0; the most up-to-date syntax specification can always be found on our website [1].

2.1 Language Constructs

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

In addition to `work`, a filter can contain a `prework` function that is executed exactly once between initialization and the steady-state. Like `work`, `prework` can access the input and output tapes of the filter; however, the I/O rates of `work` and `prework` can differ. In an `FIRFilter`, a `prework` function is essential for correctly filtering the beginning of the input stream. The user never calls the `init`, `prework`, and `work` functions–they are all called automatically.

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

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

There are two other stream constructs 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) $roundrobin(w_1, \ldots, w_n)$, which sends the first w_1 items to the first stream, the next w_2 items to the second stream, and so on. roundrobin is also the only type of joiner that we support; its function is analogous to a roundrobin splitter. If a roundrobin is written without any weights, we assume that all $w_i = 1$. The splitter and joiner type are specified with the keywords `split` and `join`, respectively (see Figure 1); 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.

2.2 Design 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.

3. THE RAW ARCHITECTURE

The Raw Microprocessor [9, 35] addresses the wire delay problem [15] 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 [13] 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 [9] 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.

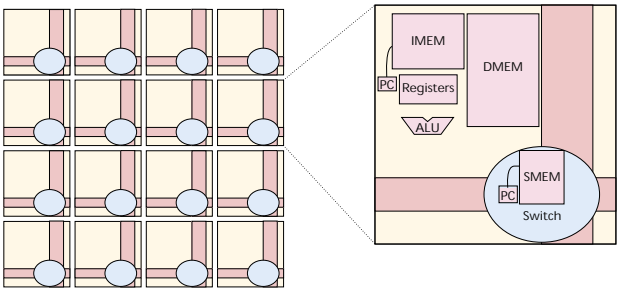


Figure 4: Block diagram of the Raw architecture.

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 [22]. Because the routers are pipelined and compile-time scheduled, they can deliver a value from the ALU of one tile to the ALU of a neighboring tile in 3 cycles, or more generally, 2+N cycles for an inter-tile distance of N hops.

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 ASIC chip. With a target clock rate of 250 MHz, the tile employs as compute processor an 8-stage, single issue, in-order MIPS-style pipeline that has a 32 KB data cache, 32 KB of instruction memory, and 64 KB of static router memory. 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.

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 [12]; we use KOPI as our infrastructure because StreamIt has evolved from a Java-based syntax. We translate the StreamIt syntax into the

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.

KOPI syntax tree, and then construct the StreamIt IR (SIR) that encapsulates the hierarchical stream graph. Since the structure of the graph might be parameterized, we propagate constants and expand each stream construct to a static structure of known extent. At this point, we can calculate an execution schedule for the nodes of the stream graph.

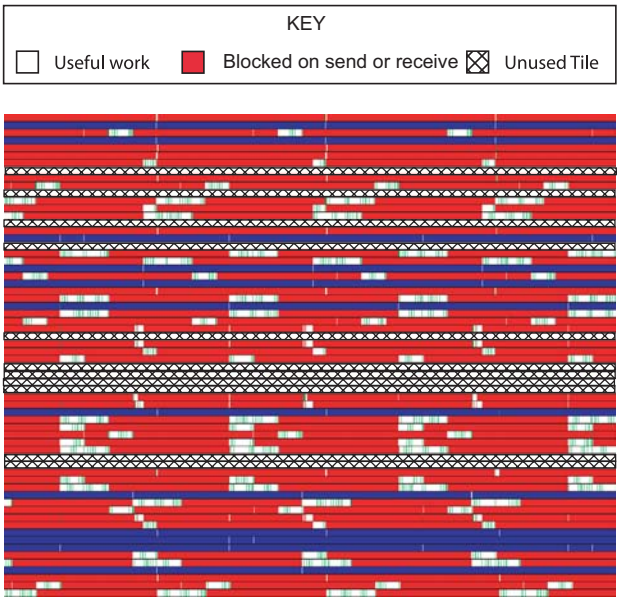
The automatic scheduling of the stream graph is one of the primary benefits that StreamIt offers, and the subtleties of scheduling and buffer management are evident throughout all of the following phases of the compiler. The scheduling is complicated by StreamIt's support for the `peek` operation, which 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 [34]. In this paper, we use a simple hierarchical scheduler that constructs a Single Appearance Schedule (SAS) [5] for each filter. A SAS is a schedule where each node appears exactly once in the loop nest denoting the execution order. We construct one such loop nest for each hierarchical stream construct, such that each component is executed a set number of times for every execution of its parent. In later 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.

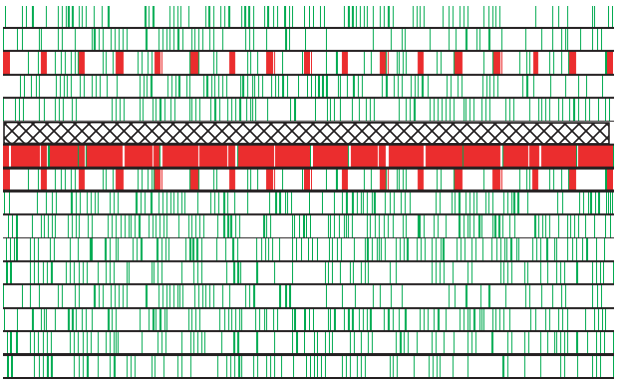
Following the scheduler, the compiler has stages that are specific for communication-exposed architectures: partitioning, layout, and communication scheduling. The next three sections of the paper are devoted to these phases.

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 under consideration. While one could envision each filter running on a separate machine in a parallel system, StreamIt hides the granularity of the



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



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

Figure 5: Execution traces for the (a) original and (b) partitioned versions of the Radar application. The x axis denotes time, and the y axis denotes the processor. Dark bands indicate periods where processors are blocked waiting to receive an input or send an output; light regions indicate periods of useful work. The thin stripes in the light regions represent pipeline stalls. Our partitioning algorithm decreases the granularity of the graph from 53 unbalanced tiles (original) to 15 balanced tiles (partitioned). The throughput of the partitioned graph is 2.3 times higher than the original.

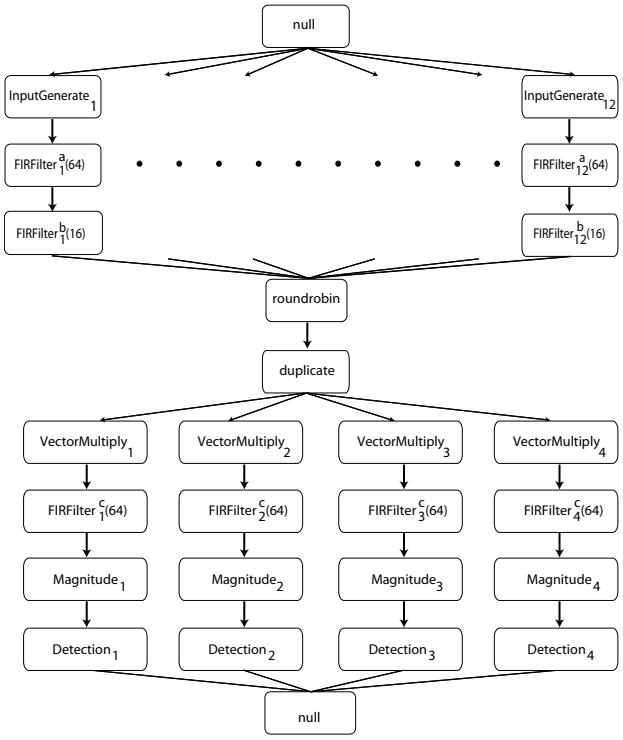


Figure 6: Stream graph of the original 12x4 Radar application. The 12x4 Radarapplication has 12 channels and 4 beams; it is the largest version that fits onto 64 tiles without filter fusion.

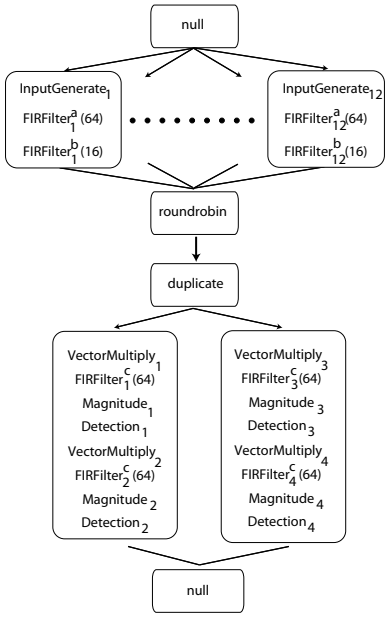


Figure 7: Stream graph of the load-balanced 12x4 Radar application. Vertical fusion is applied to collapse each pipeline into a single filter, and horizontal fusion is used to transform the 4-way splitjoin into a 2-way splitjoin. Figure 5 shows the benefit of these transformations.

target machine from the programmer. Thus, it is the responsibility of the compiler to adapt the granularity of the stream graph for efficient execution on a particular architecture.

We use the word *partitioning* to refer to the process of dividing a stream program into a set of balanced computation units. Given that a maximum of N computation units can be supported, 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.

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, a simple greedy algorithm is used to automatically select the targets of fusion and fission, based on the estimate of the work in each node.

For example, in the case of the Radar application, the original stream graph (Figure 6) contains 52 filters. These filters have unbalanced amounts of computation, as evidenced by the execution trace in Figure 5(a). The partitioner fuses all of the pipelines in the graph, and then fuses the bottom 4-way splitjoin into a 2-way splitjoin, yielding the stream graph in Figure 7. As illustrated by the execution trace in Figure 5(b), the partitioned graph has much better load balancing. In the following sections, we describe in more detail the transformations utilized by the partitioner.

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 Unoptimized Fusion Algorithm

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

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

2. For each pair of neighboring nodes, introduce a circular buffer that is large enough to hold all the items

produced during the initial schedule and one iteration of the steady-state schedule. For each buffer, maintain indices to keep track of the head and tail of the FIFO queue.

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

That is, a naive approach to filter fusion is to simply implement the channel abstraction and to leverage StreamIt's static rates to simulate the execution of the graph. However, the performance of fusion depends critically on the implementation of channels, and there are several high-level optimizations that the compiler employs to improve upon the performance of a general-purpose buffer implementation. We describe a few of these optimizations in detail in the following sections.

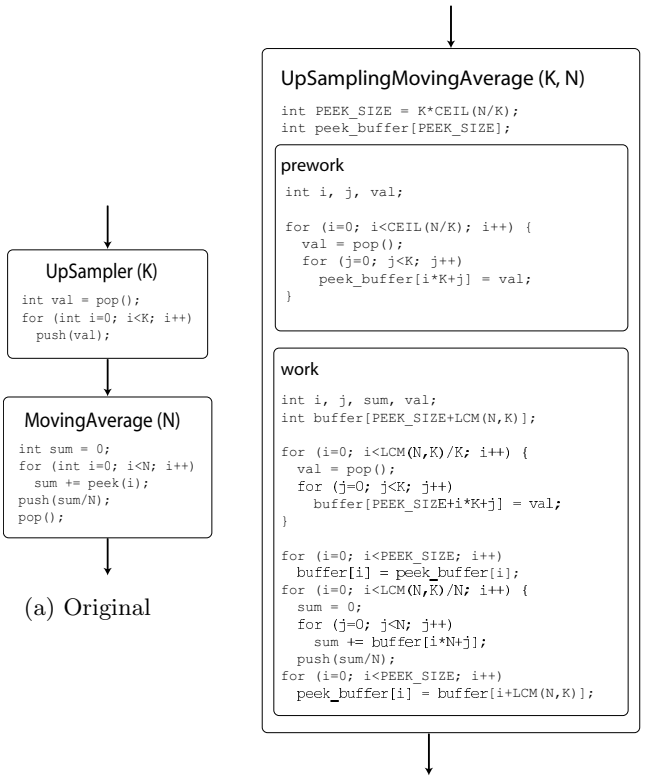
5.2.2 Optimizing Vertical Fusion

Figure 8 illustrates two of our optimizations for vertical fusion: the localization of buffers and the elimination of modulo operations. In this example, the UpSampler pushes K items on every step, while the MovingAverage filter peeks at N items but only pops 1. The effect of the optimizations are two-fold. First, buffer localization splits the channel between the filters into a local `buffer` (holding items that are transfered within `work`) and a persistent `peek_buffer` (holding items that are stored between iterations of `work`). Second, modulo elimination arranges copies between these two buffers so that all index expressions are known at compile time, preventing the need for a modulo operation to wrap around a circular buffer.

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

5.2.3 Optimizing Horizontal Fusion

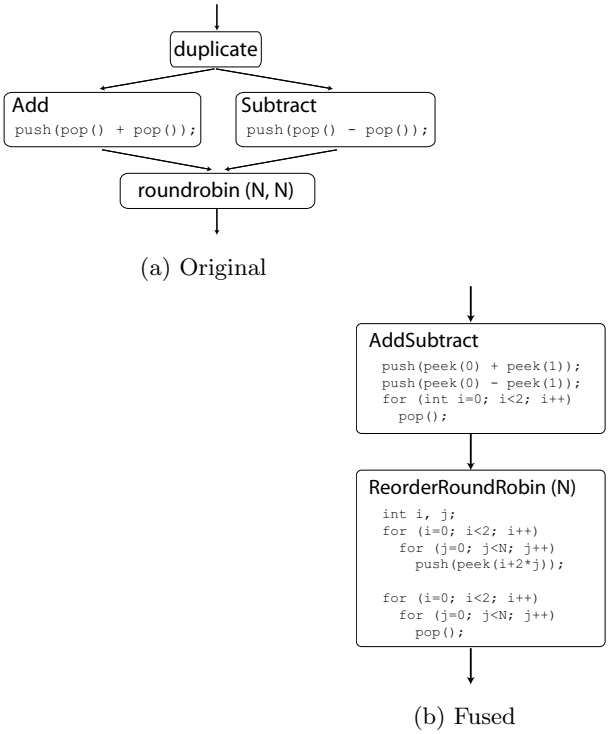
The naive fusion algorithm maintains a separate input buffer for each parallel stream in a splitjoin. However, in the case of a duplicate splitter, the input buffer can be shared between the streams, as illustrated in Figure 9. As in pipeline fusion, the code of the component filters is inlined into 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 reorder-roundrobin filter that re-arranges the output of the fused filter according to the weights of the joiner. Separating this reordering phase from the fused filter also decreases



(a) Original

(b) Fused

Figure 8: Vertical fusion with buffer localization and modulo-division optimizations.



(a) Original

(b) Fused

Figure 9: Horizontal fusion of a duplicate splitjoin construct with buffer sharing optimization.

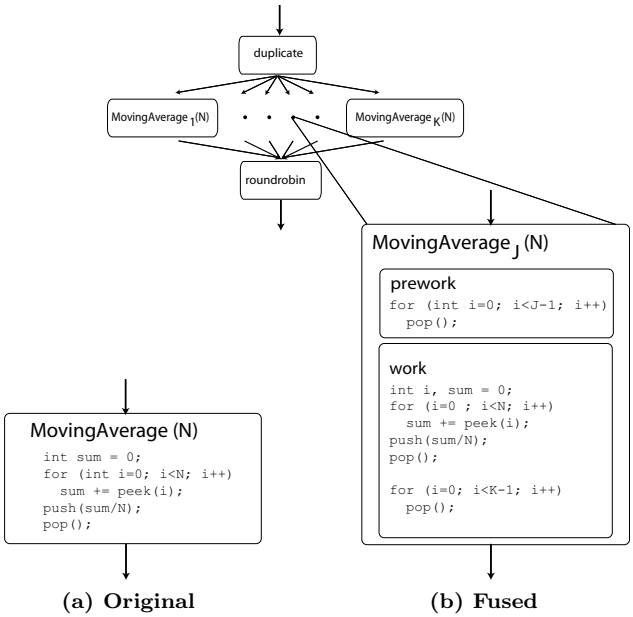


Figure 10: Fission of a filter that peeks.

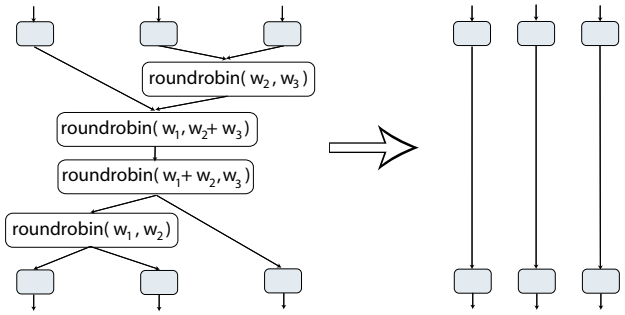


Figure 11: Synchronization removal. If there are neighboring splitters and joiners with matching rates, then the nodes can be removed and the component streams can be connected. The example above is drawn from a subgraph of the 3GPP application; the compiler automatically performs this transformation to expose parallelism and improve the partitioning. Synchronization removal is especially valuable 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 buffer requirements within the fused node. When appropriate, pipeline fusion can be applied to fuse these two nodes into a single filter that represents the entire splitjoin.

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 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 steady-state I/O rates of *peek*, *pop*, and *push*, that is being parallelized into an K-way splitjoin. There are two cases to consider:

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

2. If $peek > pop$, then a different transformation is applied (see Figure 10). In this case, the splitter is a duplicate, since the component filters need to examine overlapping parts of the input stream. The i'th component has a steady-state work function that begins with the work function of F, but appends a series of $(K-1) * pop$ pop statements in order to account for the data that is consumed by the other components. Also, the i'th filter has a prework function that pops $(i-1) * pop$ items from the input stream, to account for the consumption of previous filters on the first iteration of the splitjoin. As before, the joiner is a roundrobin that has a weight of *push* for each stream.

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, neighboring splitters and joiners with matching weights can be eliminated (Figure 11); a splitjoin construct can be divided into a hierarchical set of splitjoins to enable a finer granularity of fusion (Figure 12); and identical stateless filters can be pushed through a splitter or joiner node if the weights are adjusted accordingly.

5.5 Automatic Partitioning

In order to drive the partitioning process, we have implemented a simple greedy algorithm that performs well on most applications. The algorithm analyzes the `work` function of each filter and estimates the number of cycles required to execute it. Because of the static I/O rates in StreamIt, most loops within `work` can be unrolled, allowing a close approximation of the actual cycle count. Multiplying this cycle count by the multiplicity of the filter in the

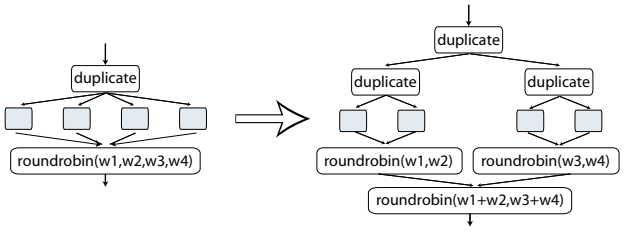


Figure 12: Breaking a splitjoin into hierarchical units. Though our horizontal fusion algorithms work on the granularity of an entire splitjoin, it is straightforward to transform a large splitjoin into a number of smaller pieces, as shown here. Following this transformation, the fusion algorithms can be applied to obtain an intermediate level of granularity. This technique was employed to help load-balance the Radar application (see Section 9).

steady-state schedule, the algorithm obtains an estimate of the steady-state computational requirements for each filter.

In the case where there are fewer filters than tiles, the partitioner considers the filters in decreasing order of their computational requirements and attempts to split them using the filter fission algorithm described above. Fission proceeds until there are enough filters to occupy the available machine resources, or until the heaviest node in the graph is not amenable to a fission transformation. Generally, it is not beneficial to split nodes other than the heaviest one, as this would introduce more synchronization without alleviating the bottleneck in the graph.

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

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

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 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. These requirements are satisfied by the partitioning phase described above.

The layout phase of the StreamIt compiler is implemented using simulated annealing [19]. We choose simulated annealing for its combination of performance and flexibility. To adapt the layout phase for a given architecture, we supply the simulated annealing algorithm with three architecture-specific parameters: a cost function, a perturbation function, and the set of legal layouts. To change the compiler to target one tiled architecture instead of another, these parameters should require only minor modifications.

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

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. Given fixed values of *channels* and $route$, our cost function evaluates a given layout of the stream graph:

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

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

In this equation, $\mathbf{items}(src, dst)$ gives the number of data

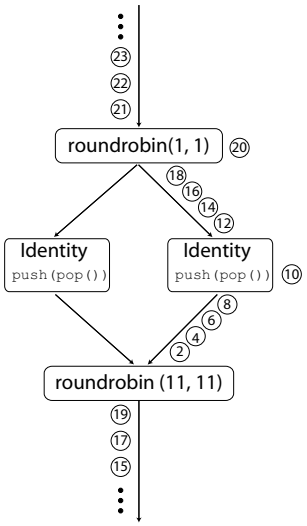


Figure 13: Example of deadlock in a splitjoin. As the joiner is reading items from the stream on the left, items accumulate in the channels on the right. On Raw, senders will block once a channel has four items in it. Thus, once 10 items have passed through the joiner, the system is deadlocked, as the joiner is trying to read from the left, but the stream on the right is blocked.

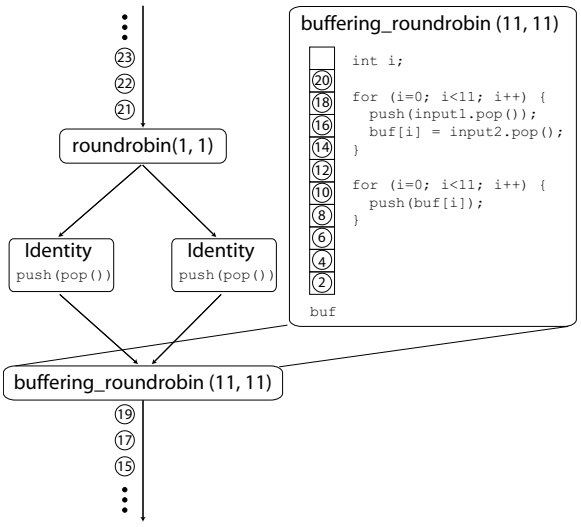


Figure 14: Fixing the deadlock with a buffering joiner. The buffering_roundrobin is an internal StreamIt construct (it is not part of the language) which reads items from its input channels in the order in which they arrive, rather than in the order specified by its weights. The order of arrival is determined by a simulation of the stream graph's execution; thus, the system is guaranteed to be deadlock-free, as the order given by the simulation is feasible for execution on Raw. To preserve the semantics of the joiner, the items are written to the output channel from the internal buffers in the order specified by the joiner's weights. The ordered items are sent to the output as soon as they become available.

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

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

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. Its goal is to avoid deadlock and starvation while utilizing the parallelism explicit in the stream graph.

The exact implementation of the communication scheduler is tied to the communication model of the target. The simplest mapping would occur for 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. One can generate orderings to minimize buffer length, maximize parallelism, or minimize latency.

Deadlock must be carefully avoided in the communication scheduler. Each architecture requires a different deadlock avoidance mechanism 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 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

Benchmark	Description	lines of code	# of constructs in the program				# of filters in the expanded graph
			filters	pipelines	splitjoins	feedbackloops	
FIR	64 tap FIR	125	5	1	0	0	132
Radar	Radar array front-end[20]	549	8	3	6	0	52
Radio	FM Radio with an equalizer	525	14	6	4	0	26
Sort	32 element Bitonic Sort	419	4	5	6	0	242
FFT	64 element FFT	200	3	3	2	0	24
Filterbank	8 channel Filterbank	650	9	3	1	1	51
GSM	GSM Decoder	2261	26	11	7	2	46
Vocoder	28 channel Vocoder [30]	1964	55	8	12	1	101
3GPP	3GPP Radio Access Protocol [3]	1087	16	10	18	0	48

Table 2: Application Characteristics.

Benchmark	250 MHz Raw processor					C on a 2.2 GHz Intel Pentium IV
	StreamIt on 16 tiles				C on a single tile	
	Utilization	# of tiles used	MFLOPS	Throughput (per 10^5 cycles)	Throughput (per 10^5 cycles)	Throughput (per 10^5 cycles)
FIR	84%	14	815	1188.1	293.5	445.6
Radar	79%	16	1,231	0.52	*app. too large*	0.041
Radio	73%	16	421	53.9	8.85	14.1
Sort	64%	16	N/A	2,664.4	225.6	239.4
FFT	42%	16	182	2,141.9	468.9	448.5
Filterbank	41%	16	644	256.4	8.9	7.0
GSM	23%	16	N/A	80.9	*app. too large*	7.76
Vocoder	17%	15	118	8.74	*app. too large*	3.35
3GPP	18%	16	44	119.6	17.3	65.7

Table 3: Performance Results.

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 13 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 14). 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.

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

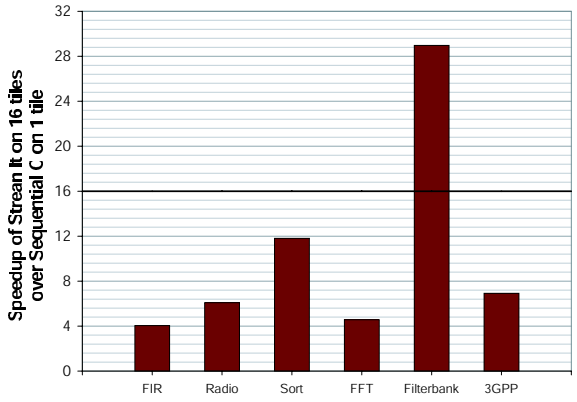


Figure 15: StreamIt throughput on a 16-tile Raw machine, normalized to throughput of hand-written C running on a single Raw tile.

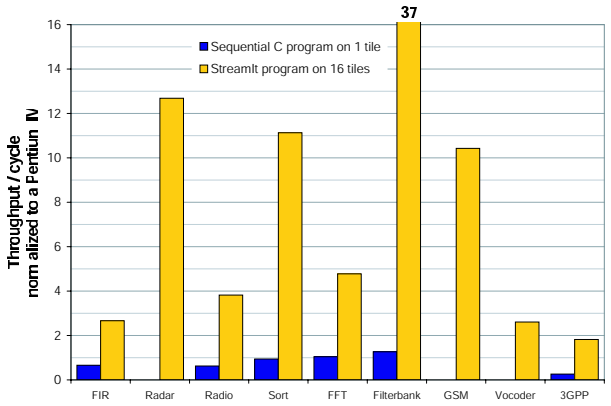


Figure 16: Throughput of StreamIt code running on 16 tiles and C code running on a single tile, normalized to throughput of C code on a Pentium IV.

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 the send instruction. Also, the communication scheduler calculates the maximum size of each buffer. With this information the code generation phase can produce the code necessary to realize the internal buffer schedule on the tile processor.

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 looping infinitely.

9. RESULTS

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

We evaluate the StreamIt compiler for the set of applications shown in Table 2; our results appear in Table 3. For each application, we compare the throughput of StreamIt with a hand-written C program, running the latter on either a single tile of Raw or on a Pentium IV. For Radio, GSM, and Vocoder, the C source code was obtained from a third party; in other cases, we wrote a C implementation following a reference algorithm. For each benchmark, we show MFLOPS (which is N/A for integer applications), processor utilization (the percentage of time that an *occupied tile* is not blocked on a send or receive), and throughput. We also show the performance of the C code, which is not available for C programs that did not fit onto a single Raw tile (Radar, GSM, and Vocoder). Figures 15 and 16 illustrate the speedups obtained by StreamIt compared to the C implementations[2].

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

StreamIt performs respectably compared to the C implementations, although there is room for improvement. The aim of StreamIt is to provide a higher level of abstraction than C without sacrificing performance. Our current implementation has taken a large step towards this goal. For instance, the synchronization removal optimization improves the throughput of 3GPP by a factor of 1.8 on 16 tiles (and by a factor of 2.5 on 64 tiles.) Also, our partitioner can be very effective–as illustrated in Figure 5, partitioning the Radar application improves performance by a factor of 2.3 even though it executes on less than one third of the tiles.

The StreamIt optimization framework is far from complete, and the numbers presented here represent a first step rather than an upper bound on our performance. We are actively implementing aggressive inter-node optimizations and more sophisticated partitioning strategies that will bring us closer to achieving linear speedups for programs with abundant parallelism.

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

10. RELATED WORK

The Transputer architecture [2] is an array of processors, where neighboring processors are connected with unbuffered point-to-point channels. The Transputer does not include a separate communication switch, and the processor must get involved to route messages. The programming language used for the Transputer is Occam [17]: a streaming language similar to CSP [16]. 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 [24].

DSPL is a language with simple filters interconnected in a flat acyclic graph using unbuffered channels [25]. Unlike the Occam compiler for the Transputer, the DSPL compiler automatically maps the graph into the available resources of the Transputer. The DSPL language does not expose a cyclic schedule, thus the compiler models the possible executions of each filter to determine the possible cost of execution and the volume of communication. It uses a search technique to map multiple filters onto a single processor for load balancing and communication reduction.

The Imagine architecture is specifically designed for the streaming application domain [28]. 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 [18]. 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 [7] is a scalable multiprocessor with configurable communication between nodes. In iWarp, one can set up a few FIFO channels for communicating between non-neighboring nodes. However, reconfiguring the communication channels is more coarse-grained and has a higher cost than on Raw, where the network routing patterns can be reconfigured on a cycle-by-cycle basis [32]. ASSIGN [26] is a tool for building large-scale applications on multiprocessors, especially iWarp. ASSIGN starts with a coarse-grained flow graph that is written as fragments of C code. Like StreamIt, it performs partitioning, placement, and routing of the nodes in the graph. However, ASSIGN is implemented as a runtime system instead of a full language and compiler such as StreamIt. Consequently, it has fewer opportunities for global transformations such as fission and reordering.

SCORE (Stream Computations Organized for Reconfigurable Execution) is a stream-oriented computational model for virtualizing the resources of a reconfigurable architecture [8]. Like StreamIt, SCORE aims to improve portability across reconfigurable machines, but it takes a dynamic approach of time-multiplexing computations (divided into "compute pages") from within the operating system, rather than statically scheduling a program within the compiler.

Ptolemy [21] is a simulation environment for heterogeneous embedded systems, including the domain of Synchronous Dataflow (SDF) that is similar to the static-rate stream graphs of StreamIt. While there are many well-established scheduling techniques for SDF [5], the round-robin nodes in our stream graph require the more general model of Cyclo-Static Dataflow (CSDF) [6] for which there are fewer results. Even CSDF does not have a notion of an initialization phase, filters that peek, or a dynamic messaging system as supported in StreamIt. In all, the StreamIt compiler differs from Ptolemy in its focus on optimized code generation for the nodes in the graph, rather than high-level modeling and design.

Proebsting and Watterson [27] present a filter fusion algorithm that interleaves the control flow graphs of adjacent nodes. However, they assume that nodes communicate via synchronous `get` and `put` operations; StreamIt's asynchronous peek operations and implicit buffer management fall outside the scope of their model.

A large number of programming languages have included a concept of a stream; see [31] for a survey. Synchronous languages such as LUSTRE [14], Esterel [4], and Signal [11] also target the embedded domain, but they are more control-oriented than StreamIt and are not aggressively optimized for performance. Sisal (Stream and Iteration in a Single Assignment Language) is a high-performance, implicitly parallel functional language [10]. The Distributed Optimizing Sisal Compiler [10] considers compiling Sisal to distributed memory machines, although it is implemented as a coarse-grained master/slave runtime system instead of a fine-grained static schedule.

11. CONCLUSION

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

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

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

12. ACKNOWLEDGEMENTS

We are very grateful to Anant Agarwal, Michael Taylor, David Wentzlaff, Walter Lee, Matt Frank, and the rest of the Raw group for developing the Raw infrastructure and for investing a lot of time supporting us. We also thank Andy Ong, John Chapin, Vanu Bose, and Stephanie Seneff for valuable conversations regarding applications for StreamIt. Our thanks to Mani Narayanan as well for implementing the BitonicSort application and helping to debug the compiler. This work is supported in part by a grant from DARPA (PCA F29601-04-2-0166), an award from NSF (CISE EIA-0071841), and fellowships from the Singapore-MIT Alliance and the MIT-Oxygen Project.

13. REFERENCES

[1] Streamit homepage. `http://compiler.lcs.mit.edu/streamit`.

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

[3] 3rd Generation Partnership Project. *3GPP TS 25.201, V3.3.0, Technical Specification*, March 2002.

[4] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2), 1992.

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

[6] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Trans. on Signal Processing*, 1996.

[7] 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*, 1988.

[8] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE): Extended Abstract. In *Proceedings of the Conference on Field Programmable Logic and Applications*, 2000.

[9] M. B. T. et. al. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. IEEE Micro vol 22, Issue 2, 2002.

[10] J. Gaudiot, W. Bohm, T. DeBoni, J. Feo, and P. Mille". The Sisal Model of Functional Programming and its Implementation. In *Proceedings of the Second Aizu International Symposium on Parallel Algorithms/Architectures Synthesis*, 1997.

[11] T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. *Springer Verlag Lecture Notes in Computer Science*, 274, 1987.

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

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

[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proc. of the IEEE*, 79(9), 1991.

[15] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. In *Proc. of the IEEE*, 2001.

[16] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

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

[18] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, and B. Towles. Stream scheduling. In *Proc. of the 3rd Workshop on Media and Streaming Processors*, 2001.

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

[20] J. Lebak. Polymorphous Computing Architecture (PCA) Example Applications and Description. External Report, Lincoln Laboratory, Mass. Inst. of Technology, 2001.

[21] E. A. Lee. Overview of the Ptolemy Project. UCB/ERL Technical Memorandum UCB/ERL M01/11, Dept. EECS, University of California, Berkeley, CA, March 2001.

[22] 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*, 1998.

[23] 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.

[24] 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, June 1987.

[25] A. Mitschele-Thiel. Automatic Configuration and Optimization of Parallel Transputer Applications. *Transputer Applications and Systems '93*, 1993.

[26] D. R. O'Hallaron. The ASSIGN Parallel Program Generator. Carnegie Mellon Technical Report CMU-CS-91-141, 1991.

[27] T. A. Proebsting and S. A. Watterson. Filter Fusion. In *POPL*, 1996.

[28] 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*, 1998.

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

[30] S. Seneff. Speech transformation system (spectrum and/or excitation) without pitch extraction. Master's thesis, Massachussetts Institute of Technology, 1980.

[31] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7), 1997.

[32] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. Technical Report MIT-LCS-TR-859, Mass. Inst. of Technology, July 2002.

[33] 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.

[34] 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.

[35] 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), 1997.