StreamIt: A Language and Compiler for Communication-Exposed Architectures

William Thies, Michael I. Gordon, Michal Karczmarek,
David Maze, and Saman Amarasinghe
{thies, mgordon, karczma, dmaze, saman}@lcs.mit.edu
Laboratory for Computer Science, Massachusetts Institute of Technology

***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 necesary to develop a common machine language to allow programmers to express an algorithm in a way that can be efficiently mapped across these architectures.**

We propose a new common machine language for grid-based software-exposed architectures: StreamIt. StreamIt is a high-level programming language with explicit support for streaming computation. Unlike sequential programs with obscured dependence information and complex communication patterns, a stream program is naturally written as a set of concurrent filters with regular steady-state communication. The language imposes a hierarchical structure on the stream graph that enables novel representations and optimizations within the StreamIt compiler.

We have implemented a fully functional compiler that parallelizes StreamIt applications for Raw, including several load-balancing transformations. Though StreamIt exposes the parallelism and communication patterns of stream programs, analysis is needed to adapt a stream program to a software-exposed processor. We describe a partitioning algorithm that employs fission and fusion transformations to adjust the granularity of a stream graph, a layout algorithm that maps a stream graph to a given network topology, and a scheduling strategy that generates a fine-grained static communication pattern for each computational element. Using the cycle-accurate Raw simulator, we demonstrate that the StreamIt compiler can automatically map a high-level stream abstraction to Raw. We consider this work to be a first step towards a portable programming model for communication-exposed architectures.

I. Introduction

A *common machine language* is an essential abstraction that allows programmers to express an algorithm in a way that can be efficiently executed on a variety of architectures. The key properties of a common machine language (CML) are: 1) it abstracts away the idiosyncratic differences between one architecture and another so that a programmer doesn't have to worry about them, and 2) it encapsulates the common properties of the architectures such that a compiler for any given target can still produce an efficient executable.

For von-Neumann architectures, the canonical CML is C: instructions consist of basic arithmetic operations, executed sequentially, which operate on either local variables or values drawn from a global block of memory. C has been implemented efficiently on a wide range of architectures, and it saves the programmer from having to adapt to each kind of register layout, cache configuration, and instruction set.

However, recent years have seen the emergence of a class of grid-based architectures [1], [2], [3] for which the von-Neumann model no longer holds, and for which C is no longer an adequate CML. The design of these processors is fundamentally different in that they are conscious of wire delays–instead of just arithmetic computations–as the barriers to performance. Accordingly, grid-based architectures support fine-grained, reconfigurable communication between replicated processing units. Rather than a single instruction stream with a monolithic memory, these machines contain multiple instruction streams with distributed memory banks.

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 renders C obsolete as a CML, since it fails to hide the architectural details from the programmer and it imposes abstractions which are a mismatch for the domain.

To bridge this gap, we propose a new common machine language for grid-based processors: StreamIt. The StreamIt language makes explicit the large-scale parallelism and regular communication patterns that these architectures were designed to exploit. A program is represented not as a monolithic memory and instruction stream, but rather as a composition of autonomous filters, each of which contains its own memory and can only communicate with its immediate neighbors via high-bandwidth data channels. In addition, StreamIt provides a low-bandwidth messaging system that filters can use for non-local communication. We believe that StreamIt abstracts away the variations in grid-based processors while encapsulating their common properties, thereby enabling compilers to efficiently map a single source program to a variety of modern processors.

II. Streaming Application Domain

The applications that make use of a stream abstraction are diverse, with targets ranging from embedded devices, to consumer desktops, to high-performance servers. Examples include systems such as the Click modular router [4] and the Spectrumware software radio [5], [6]; specifications such as the Bluetooth communications protocol [7], the GSM Vocoder [8], and the AMPS cellular base station [9]; and almost any application developed with Microsoft's DirectShow library [10], Real Network's RealSDK [11] or Lincoln Lab's Polymorphous Computing Architecture [12].

We have identified a number of properties that are common to such applications–enough so as to characterize them as belonging to a distinct class of programs, which we will refer to as *streaming applications*. We believe that the salient characteristics of a streaming application are as follows:

1) *Large streams of data.* Perhaps the most fundamental aspect of a streaming application is that it operates on a large (or virtually infinite) sequence of data items, hereafter referred to as a *data stream*. Data streams generally enter the program from some external source, and each data item is processed for a limited time before being discarded. This is in contrast to scientific codes, which manipulate a fixed input set with a large degree of data reuse.
2) *Independent stream filters.* Conceptually, a streaming computation represents a sequence of transformations on the data streams in the program. We will refer to the basic unit of this transformation as a *filter*: an operation that–on each execution step–reads one or more items from an input stream, performs some computation, and writes one or more items to an output stream. Filters are generally independent and self-contained, without references to global variables or other filters. A stream program is the composition of filters into a *stream graph*, in which the outputs of some filters are connected to the inputs of others.
3) *A stable computation pattern.* The structure of the stream graph is generally constant during the steady-state operation of a stream program. That is, a certain set of filters are repeatedly applied in a regular, predictable order to produce an output stream that is a given function of the input stream.
4) *Occasional modification of stream structure.* Even though each arrangement of filters is executed for a long time, there are still dynamic modifications to the stream graph that occur on occasion. For instance, if a wireless network interface is experiencing high noise on an input channel, it might react by adding some filters to clean up the signal; a software radio re-initializes a portion of the stream graph when a user switches from AM to FM. Sometimes, these re-initializations are synchronized with some data in the stream–for instance, when a network protocol changes from Bluetooth to 802.11 at a certain point of a transmission. There is typically an enumerable number of configurations that the stream graph can adopt in any one program, such that all of the possible arrangements of filters are known at compile time.
5) *Occasional out-of-stream communication.* In addition to the high-volume data streams passing from one filter to another, filters also communicate small amounts of control information on an infrequent and irregular basis. Examples include changing the volume on a cell phone, printing an error message to a screen, or changing a coefficient in an upstream FIR filter.
6) *High performance expectations.* Often there are real-time constraints that must be satisfied by streaming applications; thus, efficiency (in terms of both latency and throughput) is of primary concern. Additionally, many embedded applications are intended for mobile environments where power consumption, memory requirements, and code size are also important.

III. Language Overview

StreamIt includes stream-specific abstractions and representations that are designed to improve programmer productivity for the domain of programs described above. In this paper, we present StreamIt in legal Java syntax[1]. Using Java has many advantages, including programmer familiarity, availability of compiler frameworks and a robust language specification. However, the resulting syntax can be cumbersome, and in the future we plan to develop a cleaner and more abstract syntax that is designed specifically for stream programs.

A. Filters

The basic unit of computation in StreamIt is the Filter. An example of a Filter is the `FIRFilter`, shown in Figure 1. The central aspect of a filter is the `work` function, which describes the filter's most fine grained execution step in the steady state. Within the `work` function, a filter can communicate with neighboring blocks using the `input` and `output` channels, which are FIFO queues declared as fields in the Filter base class. These high-volume channels support the three intuitive operations: 1) `pop()` removes an item from the end of the channel and returns its value, 2) `peek(`i`)` returns the value of the item i spaces from the end of the channel without removing it, and 3) `push(`x`)` writes x to the front of the channel. The argument x is passed by value; if it is an object, a separate copy is enqueued on the channel.

A major restriction of StreamIt 1.0 is that it requires filters to have static input and output rates. That is, the number of items peeked, popped, and pushed by each filter must be constant from one invocation of the `work` function to the next. In fact, as described below, the input and output rates must be declared in the filter's `init` function. If a filter violates the declared rates, StreamIt throws a runtime error and the subsequent behavior of the program is undefined. We plan to support dynamically changing rates in a future version of StreamIt.

Each Filter also contains an `init` function, which is called at initialization time. The `init` function serves two purposes. Firstly, it is for the user to establish the initial state of the filter. For example, the FIRFilter records `weights`, the coefficients that it should use for filtering. A filter can also push, pop, and peek items from within the `init` function if it needs to set up some initial state on its channels, although this usually is not necessary. A user should instantiate a filter by using its constructor, and the `init` function will be called implicitly with the same arguments that were passed to the constructor.

[1]However, for the sake of brevity, the code fragments in this document are sometimes lacking modifiers or methods that would be needed to make them strictly legal Java.

```
class FIRFilter extends Filter {
   float[] weights;
   int N;

   void init(float[] weights) {
      setInput(Float.TYPE); setOutput(Float.TYPE);
      setPush(N); setPop(1); setPeek(N);
      this.weights = weights;
      this.N = weights.length;
   }

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

class Main extends Pipeline {
   void init() {
      add(new DataSource());
      add(new FIRFilter(N));
      add(new Display());
   }
}
```

Fig. 1. An FIR filter in StreamIt.

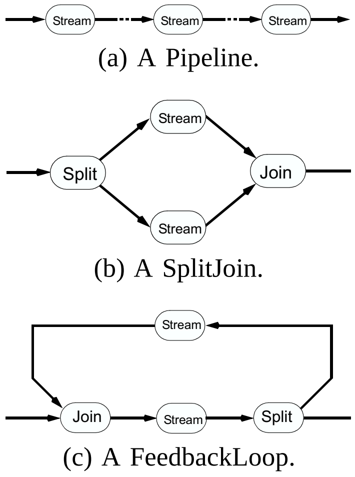


Fig. 2. Stream structures supported by StreamIt.

```
class Delay extends Filter {
  void init(int delay) {
    setInput(Float.TYPE); setOutput(Float.TYPE);
    setPush(1); setPop(1);
    for (int i=0; i<delay; i++)
      output.push(0);
  }
  void work() { output.push(input.pop()); }
}

class EchoEffect extends SplitJoin {
  void init() {
    setSplitter(Duplicate());
    add(new Delay(100));
    add(new Delay(0));
    setJoiner(RoundRobin());
  }
}

class AudioEcho extends Pipeline {
  void init() {
    add(new AudioSource());
    add(new EchoEffect());
    add(new Adder());
    add(new Speaker());
  }
}
```

Fig. 3. An echo effect in StreamIt.

```
class Fibonnacci extends FeedbackLoop {
  void init() {
    setDelay(2);
    setJoiner(RoundRobin(0,1));
    setBody(new Filter() {
       void init() {
         setInput(Integer.TYPE);
         setOutput(Integer.TYPE);
         setPush(1); setPop(1); setPeek(2);
       }
       void work() {
          output.push(input.peek(0)+input.peek(1));
          input.pop();
       }
    });
    setSplitter(Duplicate());
  }

  int initPath(int index) {
    return index;
  }
}
```

Fig. 4. A FeedbackLoop version of Fibonnacci.

The second purpose of the `init` function is to specify the filter's I/O types and data rates to the StreamIt compiler. The types are specified with calls to `setInput` and `setOutput`, while the rates are specified with calls to `setPush`, `setPop`, and `setPeek`. The `setPeek` call can be ommitted if the peek count is the same as the pop count.

B. Connecting Filters

StreamIt provides three constructs for composing filters into a communicating network: Pipeline, SplitJoin, and FeedbackLoop (see Figure 2). Each structure specifies a pre-defined way of connecting filters into a single-input, single-output block, which we will henceforth refer to as a "stream". That is, a stream is any instance of a Filter, Pipeline, SplitJoin, or FeedbackLoop. Every StreamIt program is a hierarchical composition of these stream structures.

The **Pipeline** construct is for building a sequence of streams. Like a Filter, a Pipeline has an `init` function that is called upon its instantiation. Within `init`, component streams are added to the Pipeline via successive calls to `add`. For example, in the `AudioEcho` in Figure 3, the `init` function adds four streams to the Pipeline: an `AudioSource`, an `EchoEffect`, an `Adder`, and a `Speaker`. This sequence of statements automatically connects these four streams in the order specified. Thus, there is no `work` function in a Pipeline, as the component streams fully specify the behavior. The channel types and data rates are also implicit from the connections.

Each of the stream constructs can either be executed on its own, or embedded in an enclosing stream structure. The `AudioEcho` can execute independently, since the first component consumes no items and the last component produces no items. However, the `EchoEffect` must be used as a component, since the first stream inputs items and the last stream outputs items. When a stream is embedded in another construct, the first and last components of the stream are implicitly connected to the stream's neighbors in the parent construct.

The **SplitJoin** construct is used to specify independent parallel streams that diverge from a common *splitter* and merge into a common *joiner*. As in a Pipeline, the components of a SplitJoin are specified with successive calls to `add` from the `init` function. For example, the `EchoEffect` in Figure 3 adds two streams that run in parallel, each of which is a `Delay` filter.

The splitter specifies how items from the input of the SplitJoin are distributed to the parallel components. For simplicity, we allow only compiler-defined splitters, of which there are three types: 1) *Duplicate*, which replicates each data item and sends a copy to each parallel stream, 2) *RoundRobin*(i_1, i_2, ..., i_k), which sends the first i_1 data items to the stream that was added first, the next i_2 data items to the stream that was added second, and so on, and 3) *Null*, which means that none of the parallel components require any input, and there are no input items to split. If the weights are ommitted from a RoundRobin, then they are assumed to be equal to one for each stream. Note that RoundRobin can function as an exclusive selector if one or more of the weights are zero.

Likewise, the joiner is used to indicate how the outputs of the parallel streams should be interleaved on the output channel of the SplitJoin. There are two kinds of joiners: 1) *RoundRobin*, whose function is analogous to a RoundRobin splitter, and 2) *Null*, which means that none of the parallel components produce any output, and there are no output items to join. The splitter and joiner types are specified with calls to `setSplitter` and `setJoiner`, respectively. The `EchoEffect` uses a Duplicate splitter so that each item appears both directly and as an echo; it uses a RoundRobin joiner to interleave the immediate signals with the delayed ones. In `AudioEcho`, an `Adder` is used to combine each pair of interleaved signals.

The **FeedbackLoop** construct provides a way to create cycles in the stream graph. The `Fibonacci` stream in Figure 4 illustrates the use of this construct. Each FeedbackLoop contains: 1) a body stream, which is the block around which a backwards "feedback path" is being created, 2) a loop stream, which can perform some computation along the feedback path, 3) a splitter, which distributes data between the feedback path and the output channel at the bottom of the loop, and 4) a joiner, which merges items between the feedback path and the input channel at the top of the loop. These components are specified from within the `init` function via calls to `setBody`, `setLoop`, `setSplitter`, and `setJoiner`, respectively. The splitters and joiners can be any of those for SplitJoin, except for Null. The call to `setLoop` can be ommitted if no computation is performed along the feedback path.

The FeedbackLoop has a special semantics when the stream is first starting to run. Since there are no items on the feedback path at first, the stream instead inputs items from an `initPath` function defined by the FeedbackLoop; given an index i, `initPath` provides the i'th initial input for the feedback joiner. With a call to `setDelay` from within the `init` function, the user can specify how many items should be calculated with `initPath` before the joiner looks for data items from the feedback channel.

Detailed informaiton of the language can be found in [13]. We have written many programs to understand the scope and limitations of the language. For example, we have written multiple versions of matrix multiply kernels, sorting algorithms, FIR filters and FFT filters in StreamIt. We have also implemented many applications such as a Cyclic Redundancy Checker, an FM Radio with an Equalizer, a Reed Solomon Decoder for HDTV, a Vocoder [14], a Radar Array Front End [12], a GSM Decoder [8], and the physical layer of the 3GPP Radio Assess Protocol [15]. We plan to incorporate the knowlege and insight grained from developing these applications to the next version of StreamIt.

IV. Compiling StreamIt to Raw

The StreamIt language aims to be portable across communication-exposed machines. StreamIt exposes the parallelism and communication of streaming applications without depending on the topology or granularity of the underlying architecture. We have implemented a fully-functional prototype of the StreamIt compiler for Raw [2], a tiled architecture with fine-grained, programmable communication between processors. However, the compiler employs three general techniques that can be applied to compile StreamIt to machines other than Raw: 1) partitioning, which adjusts the granularity of a stream graph to match that of a given target, 2) layout, which maps a partitioned stream graph to a given network topology, and 3) scheduling, which generates a fine-grained 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 front end is built on top of KOPI, an open-source compiler infrastructure for Java [16]; we use KOPI as our infrastructure because StreamIt has evolved from a Java-based syntax. We translate the StreamIt syntax into the KOPI syntax tree, and then construct the StreamIt IR (SIR) that encapsulates the hierarchical stream graph. Since the structure of the graph might be parameterized, we propagate constants and expand each stream construct to a static structure of known extent. At this point, we can calculate an execution schedule for the nodes of the stream graph.

The StreamIt compiler is composed of the following stages that are specific for communication-exposed architectures: stream graph scheduling, stream graph partitioning, layout, and communication scheduling. The next four sections provide a brief overview of these phases. For a detailed explanation see [17], [18], [19].

A. Stream Graph Scheduling

The automatic scheduling of the stream graph is one of the primary benefits that StreamIt offers, and the subtleties of scheduling and buffer management are evident throughout the phases of the compiler described below. 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 [20]. Currently, we use a simple hierarchical scheduler that constructs a Single Appearance Schedule (SAS) [21] for each filter. We plan to develop better scheduling heuristics in the future [19]. 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.

B. *Stream Graph Partitioning*

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

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

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 5) contains 52 filters. These filters have unbalanced amounts of computation, as evidenced by the execution trace in Figure 7(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 6. As illustrated by the execution trace in Figure 7(b), the partitioned graph has much better load balancing. In the following sections, we describe in more detail the transformations utilized by the partitioner.

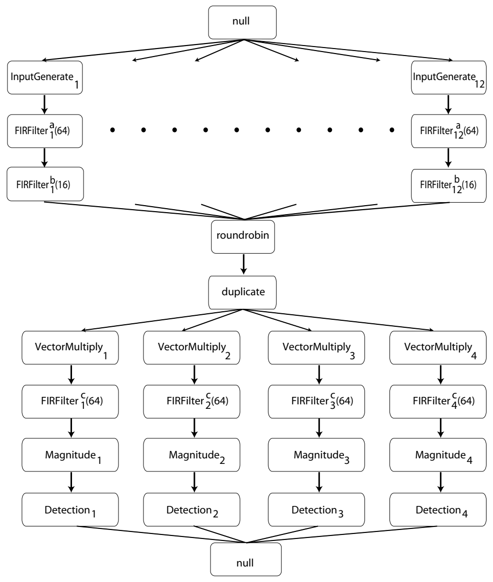


Fig. 5. Stream graph of the original 12x4 Radar application. The 12x4 Radar application has 12 channels and 4 beams; it is the largest version that fits onto 64 tiles without filter fusion.

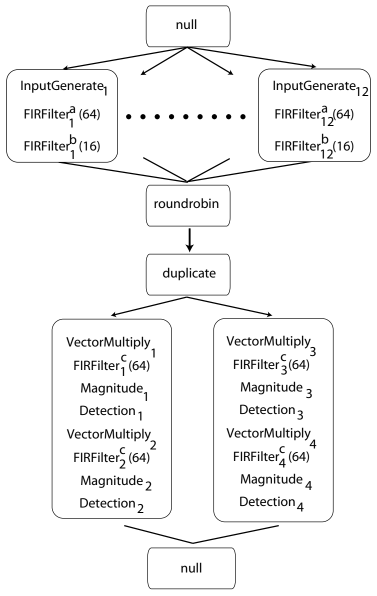
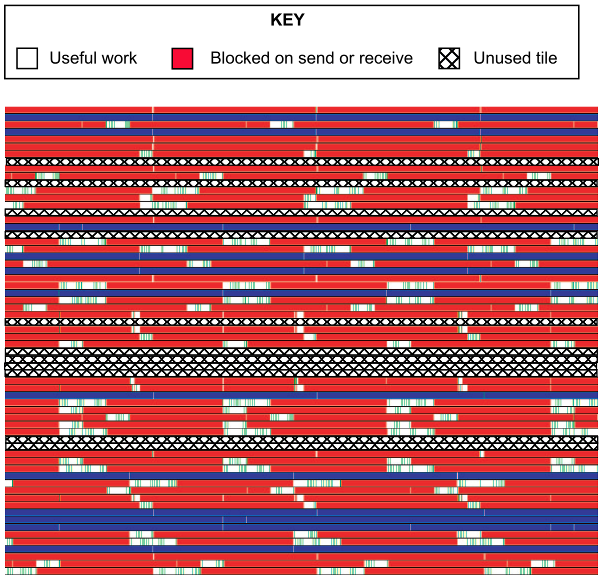
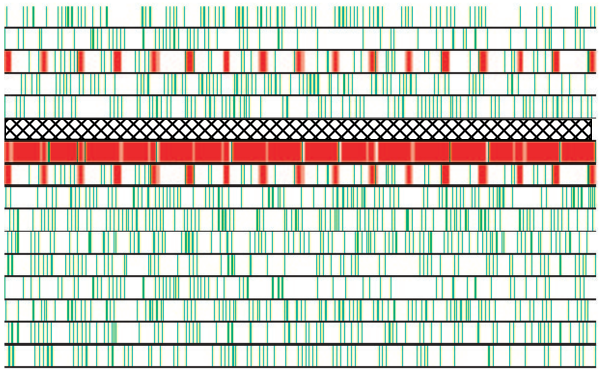


Fig. 6. Stream graph of the load-balanced 12x4 Radar application. Vertical fusion is applied to collapse each pipeline into a single filter, and horizontal fusion is used to transform the 4-way splitjoin into a 2-way splitjoin. Figure 7 shows the benefit of these transformations.

1) 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.



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



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

Fig. 7. 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.

2) 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.

3) 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 a splitjoin construct can be divided into a hierarchical set of splitjoins to enable a finer granularity of fusion and identical stateless filters can be pushed through a splitter or joiner node if the weights are adjusted accordingly.

4) 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. 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. 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.

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.

C. 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 [22]. 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.

Benchmark	Description	lines of code	# of constructs in the program: filters	pipelines	splitjoins	feedbackloops	# of filters in the expanded graph
FIR	64 tap FIR	125	5	1	0	0	132
Radar	Radar array front-end [12]	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 [14]	1964	55	8	12	1	101
3GPP	3GPP Radio Access Protocol [15]	1087	16	10	18	0	48

TABLE I
APPLICATION CHARACTERISTICS.

Benchmark	250 MHz Raw processor: StreamIt on 16 tiles: Utilization	# of tiles used	MFLOPS	Throughput (per 10^5 cycles)	C on a single tile: Throughput (per 10^5 cycles)	C on a 2.2 GHz Intel Pentium IV: 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 II
PERFORMANCE RESULTS.

Phase ordering between stream graph parttioning and layout can lead to suboptimal results. We plan to develop a unified approach for partitioning and layout in the future.

D. 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 IV, 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.

E. Preliminary Results

We evaluate the StreamIt compiler for the set of applications shown in Table I; our results appear in Table II.

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 8 and 9 illustrate the speedups obtained by StreamIt compared to the C implementations[2].

[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.

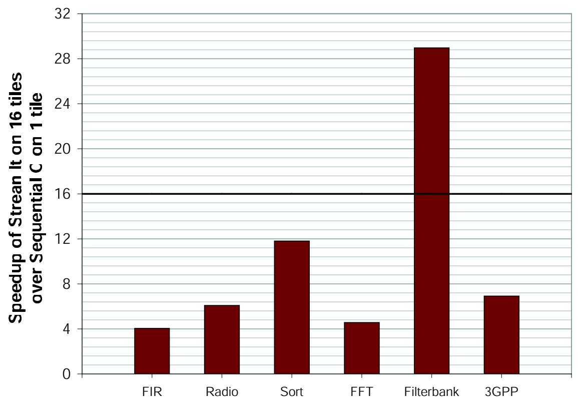


Fig. 8. StreamIt throughput on a 16-tile Raw machine, normalized to throughput of hand-written C running on a single Raw tile.

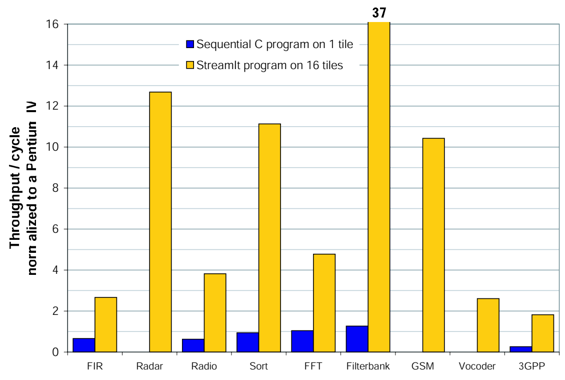


Fig. 9. Throughput of StreamIt code running on 16 tiles and C code running on a single tile, normalized to throughput of C code on a Pentium IV.

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 7, 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.

References

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

[2] 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: The Raw Machine,” MIT-LCS Technical Report TR-709, 1997.

[3] 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.

[4] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The click modular router,” *ACM Trans. on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000. [Online]. Available: http://www.acm.org/pubs/citations/journals/tocs/2000-18-3/p263-kohler/

[5] D. Tennenhouse and V. Bose, “The SpectrumWare Approach to Wireless Signal Processing,” Wireless Networks, 1999.

[6] V. Bose, M. Ismert, M. Welborn, and J. Guttag, “Virtual radios,” IEEE/JSAC, Special Issue on Software Radios, April 1999.

[7] B. Volume and B. July, “Bluetooth Spec. Vol. 1,” Bluetooth Consortium, 1999.

[8] M. Mouly and M. Pautet, *The GSM System for Mobile Communications*. Palaiseau, France: Cell&Sys, 1992.

[9] EIA/TIA, “Mobile station-land station compatibility spec.” Tech. Rep. 553, 1989.

[10] Microsoft Corporation, “Microsoft directshow,” Online Documentation, 2001.

[11] RealNetworks, “Software Developer’s Kit,” Online Documentation, 2001.

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

[13] W. Thies, M. Karczmarek, and S. Amarasinghe, “StreamIt: A Language for Streaming Applications,” in *Proceedings of the International Conference on Compiler Construction*, Grenoble, France, 2002.

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

[15] *3GPP TS 25.201, V3.3.0, Technical Specification*, 3rd Generation Partnership Project, March 2002.

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

[17] M. Gordon, W. Thies, M. Karczmarek, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, “A stream compiler for communication-exposed architectures,” in *In ASPLOS X*, October 2002.

[18] M. Gordon, “A Stream-Aware Compiler for Communication Exposed Architectures,” Master’s thesis, M. I. T., Department of Electrical Engineering and Computer Science, August 2002.

[19] M. Karczmarek, “Constrained and Phased Scheduling of Synchronous Data Flow Graphs for StreamIt Language,” Master’s thesis, M. I. T., Department of Electrical Engineering and Computer Science, September 2002.

[20] 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.

[21] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996, 189 pages. [Online]. Available: http://www.wkap.nl/book.htm/0-7923-9722-3

[22] S. Kirkpatrick, J. C.D. Gelatt, and M. Vecchi, “Optimization by Simulated Annealing,” *Science*, vol. 220, no. 4598, May 1983.