Defying the Speed of Light: Wire-Exposed Architectures and Spatially-Aware Compilers

Saman Amarasinghe
Laboratory for Computer Science, Massachusetts Institute of Technology,
200 Technology Square, Cambridge, MA 02139
1-617-253-8879 / saman@lcs.mit.edu / http://compiler.lcs.mit.edu/commit

Abstract

As wire-delay is becoming the most important design constraint of microprocessors, wire-expose architectures are emerging as a strong solution to maintaining the performance growth of modern processors. However, a von Neumann programming model cannot harness the power of these machines.

We have developed StreamIt, a novel language as a viable common machine language for wire-exposed architectures. It abstracts away the target's granularity, memory layout, and network interconnect, while capturing the notion of replicated processors that communicate in regular patterns. With this representation, we believe that the StreamIt compiler will match the performance of C code that was hand-tailored for a given grid-based machine.

1. INTRODUCTION

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[2], SmartMemories[1], TRIPS[3]). However, for them to be effective, it is necessary to use languages that circumvent the von Neumann bottleneck, inherent in current programming languages such as C, C++ and Java.

StreamIt is a streaming language that is ideally suited for wire-exposed architectures. StreamIt provides novel high-level representations to improve programmer productivity and program robustness within the streaming domain. At the same time, StreamIt exposes the inherent communication pattern of the program, allowing the compiler to aggressively optimize and effectively utilize wire-exposed architectures.

Though StreamIt exposes the parallelism and communication patterns of stream programs, much analysis is needed to adapt a stream program to a wire-exposed processor. These include fission and fusion transformations that is used to adjust the granularity of a stream graph, a layout algorithm for mapping a stream graph to a given network topology, and a scheduling algorithm for generating a fine-grained static communication pattern for each computational element.

We have developed a StreamIt compiler for the Raw architecture. The Raw processor, developed at MIT, is a simple tiled architecture that fully exposes the interconnectivity to the compiler. We demonstrate that the StreamIt compiler can automatically map a high-level stream abstraction to Raw without loosing performance. We consider this work to be a first step towards a portable programming language for wire exposed architectures.

2. The StreamIt Language

In this section we provide a brief overview of the StreamIt language; please see [6] for a more detailed description. The current version of StreamIt is legal Java syntax to simplify our presentation and implementation, and it is designed to support only streams with static input and output rates.

2.1 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.

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

   void int(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();
      putput.push(sum);
   }
}

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

Figure 1. An FIR Filter in 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.

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.

2.2 Connecting Filters

StreamIt provides three constructs for composing filters into a communicating network: Pipeline, SplitJoin, and FeedbackLoop. 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.

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. 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.

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

Each of the stream constructs can either be executed on its own, or embedded in an enclosing stream structure. 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.

3. The Raw Architecture

The Raw Microprocessor[1] addresses the wire delay problem by providing direct instruction set architecture (ISA) analogs to three underlying physical resources of the processor: gates, wires and pins. Because ISA primitives exist for these resources, a compiler 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 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.

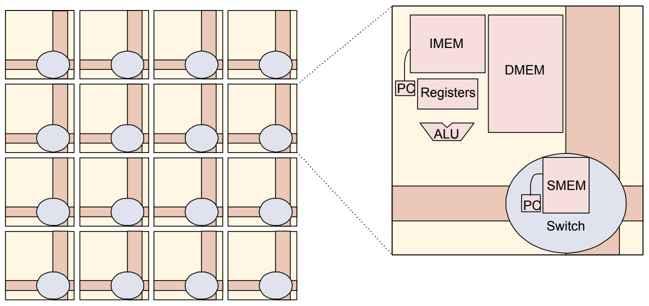

Figure 2. A block diagram of the Raw architecture.

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 2). The compute processor interfaces to the network through a bypassed, register-mapped interface[1] that allows instructions to use the networks and the register files interchangeably. In other words, a single instruction can read up to two values from the networks, compute on them, and send the result out onto the networks, with no penalty. Reads and writes in this fashion are blocking and flow-controlled, which allows for the computation to remain unperturbed by unpredictable timing variations such as cache misses and interrupts.

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

4. Compiling StreamIt for 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]. 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 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, see [8] for more details.

4.1 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.

4.2 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.

4.3 Layout

The goal of the layout phase is to assign nodes in the stream graph to computation nodes in the target architecture while minimizing the communication and synchronization present in the final layout. The layout 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[9]. 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.

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.

4.4 Communication Scheduler

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

The exact implementation of the communication scheduler is tied to the communication model of the target. The simplest mapping would occur for 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. 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.

5. Results

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[4], and Vocoder[7], the C source code was obtained from a third party; in other cases, we wrote a C implementation following a reference algorithm. We show the performance of the C code, which is not available for C programs that did not fit onto a single Raw tile (Radar[5], GSM, and Vocoder). Figures 3 and 4 illustrate the speedups obtained by StreamIt compared to the C implementations.

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.

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.

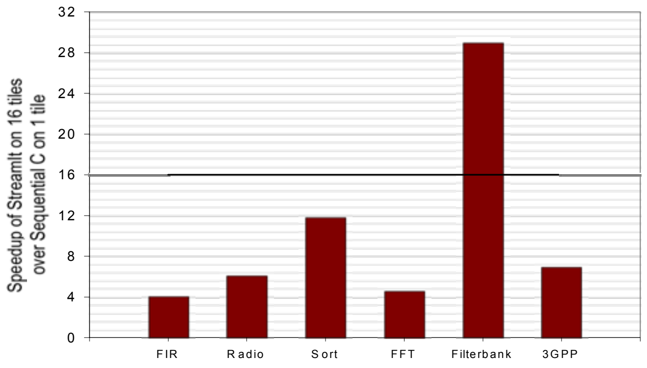

Figure 3. StreamIt throughput on a 16-tile Raw machine, normalized to throughput of hand-written C running on a single Raw tile.

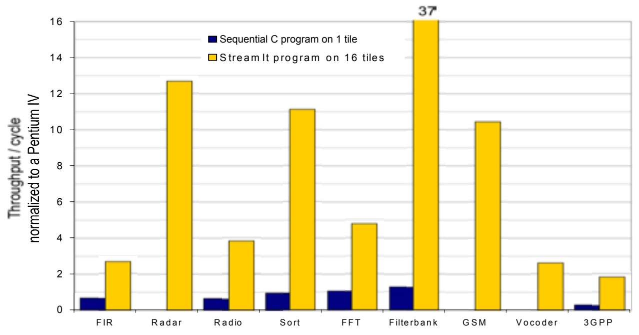

Figure 4. Throughput of StreamIt code running on 16 tiles and C code running on a single tile, normalized to throughput of C code on a Pentium IV.

6. ACKNOWLEDGEMENTS

This paper presents contributions of many members of the MIT Commit and Raw projects. 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.

7. References

[1] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, M. Horowitz. Smart memories: A modular recongurable architecture. In *ISCA 2000*, Vancouver, BC, Canada.

[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, A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9), 1997.

[3] K. Sankaralingam, R. Nagarajan, S. Keckler, 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] M. Mouly, M. Pautet, The GSM System for Mobile Communications *Cell & Sys*, 1992.

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

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

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

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

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

8. BIBLIOGRAPHY

Saman P. Amarasinghe is an Associate Professor in the Department of Electrical Engineering and Computer Science at Massachusetts Institute of Technology and a member of the MIT Laboratory for Computer Science. He received his BS in Electrical Engineering and Computer Science from Cornell University in 1988, and his MSEE and Ph.D from Stanford University in 1990 and 1997, respectively. Currently he leads the Commit compiler group and is the co-leader of the MIT Raw project. His research interests are in discovering novel approaches to improve the performance of modern computer systems without unduly increasing the complexity faced by either application developers, compiler writers, or computer architects.