StreamIt on Fleet

Amir Kamil
Computer Science Division, University of California, Berkeley
kamil@cs.berkeley.edu

UCB-AK06

July 16, 2008

1 Introduction

StreamIt [1] is a high-level programming language for streaming application. The developers claim [3] that its constructs are designed to "expose the parallelism and communication of streaming applications without depending on the topology or granularity of the underlying architecture." In this memo, we consider StreamIt's application to the Fleet architecture [4].

2 StreamIt Background

StreamIt is a streaming programming language for signal processing applications [1]. It is designed for communication-exposed architectures, such as Raw [5], in which multiple processing cores are arranged on a grid with network connections between adjacent cores. In this section, we provide an overview of the StreamIt language and some of the compiler techniques used to obtain high performance on Raw.

2.1 Language

A StreamIt program is composed of blocks of code called *filters*. In addition to an optional initialization function, a filter consists of a steady-state *work* function that corresponds to a single execution step of the filter. The work function can consist of more or less arbitrary sequential code. A filter also has an input and output queue, which can be accessed via push, pop, and peek functions. *Push* adds an element to the output queue, *pop* removes an element from the input queue, and *peek* returns the value at the given index without dequeuing it. The filter must statically specify its I/O rates for each of these three functions[1].

The composition of one or more filters is called a *stream*. Streams can be built in multiple different ways. In a *pipeline*, the output of one filter is connected to the input of the next, so that the two filters are composed sequentially. A *splitjoin* splits an input queue among mulitple parallel streams and joins their outputs into a single output queue. The input data can be replicated so that each of the parallel streams sees the same input, or it can be divided in a roundrobin fashion, with w_1 items sent to the first stream, w_2, sent to the second stream, and so on until w_n for the last stream, at

[1]The language specification states that the programmer may specify a range for each rate or even that the rate is completely unknown [1]. However, various StreamIt compiler papers require that the rate be static [3, 2].

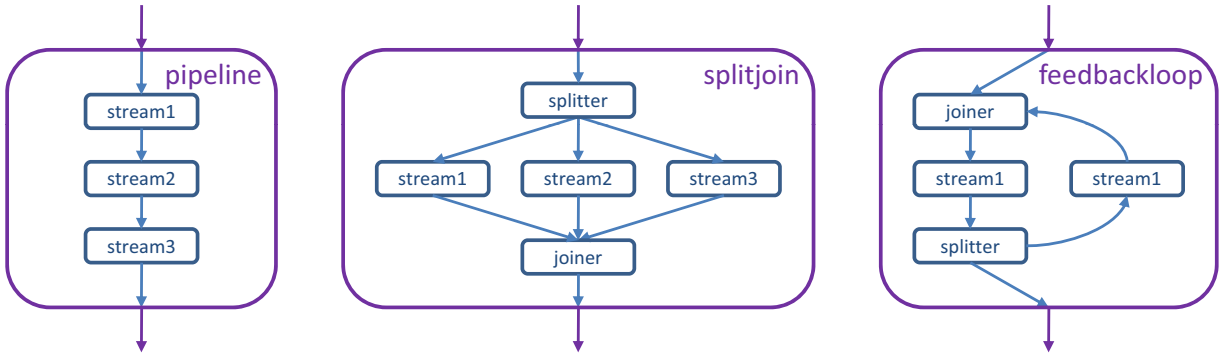

Figure 1: Streams in StreamIt can be composed in pipelines, splitjoins, and feedbackloops.

which point the process restarts with the first stream. The output data can only be joined in this roundrobin manner. A *feedbackloop* consists of an internal loop of two streams. The input to the first stream is a join of the input to the feedbackloop and the output of the second stream, and the output of the first stream is split between the output of the feedbackloop and the input of the second stream. Figure 2.1 shows examples of each type of filter composition.

StreamIt also includes a mechanism for sending out-of-band messages to upstream or downstream filters. The receiving filter must specify a handler for each type of message that it can receive. Prior to each execution of the filter's work function, its message queue is checked, and any messages received are passed to the appropriate handlers. The sending filter can time the message to a particular data item passed between the two filters. If the receiving filter is downstream from the sending filter, the message can be synchronized with any past of future data produced by the sender. If the receiving filter is upstream from the sending filter, the message can only be synchronized with data the sender will consume in the future.

The messaging system also allows dynamic reconfiguration of a portion of the stream graph [6]. A message targeted at the initialization function of a stream causes the stream to re-initialize itself, potentially resulting in a new underlying structure for that stream if the initialization parameters are different than they were in the original initialization. The StreamIt compiler recognizes re-initialization and can account for all possible stream configurations.

2.2 Partitioning

The StreamIt compiler attempts to hide the granularity of the underlying architecture by *partitioning* stream programs [3]. In this process, the program is transformed into a set of N load-balanced filters, where N is the number of computational units in the underlying machine. The compiler applies fusion and fission transformations in order to accomplish this.

2.2.1 Fusion Transformations

Fusion transformations combine multiple adjacent filters into a single filter. In *vertical fusion*, pipelined filters are collapsed into a single filter, as showin in Figure 2.2.1. Figure 2.2.1 illustrates *horizontal fusion* combines parallel components of a splitjoin. Since each filter has a static I/O rate, the compiler can simulate the I/O of each filter to determine the amount of buffering required and to translate push, peek, and pop operations into buffer accesses. Vertical fusion may require circular buffers, but the compiler can optimize these accesses to avoid modulo operations [3]. In horizontal fusion, the splitjoin components are executed sequentially within a single combined filter and can

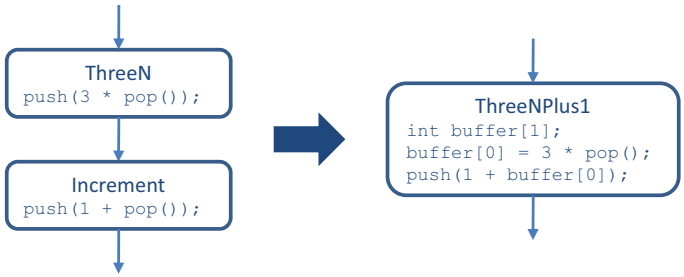

Figure 2: An example of vertical fusion. No circular buffering is required.

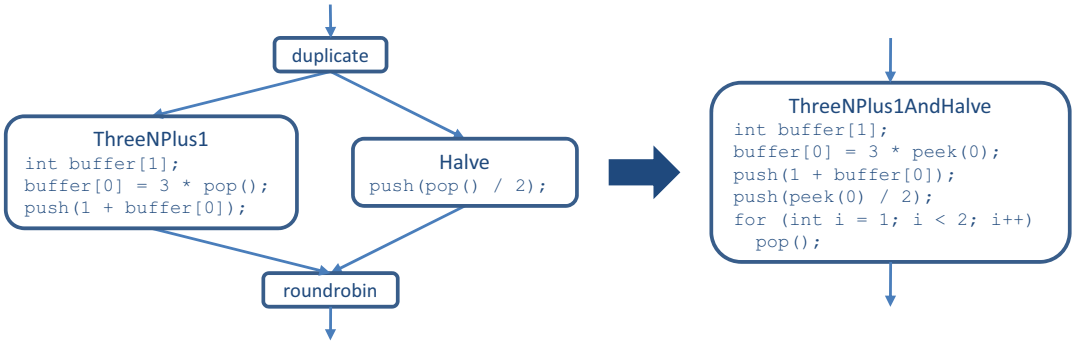

Figure 3: An example of horizontal fusion. No output reordering is required.

share the same input buffer if a duplicate splitter is used. If the output needs to be reordered, a second filter is pipelined with the combined filter to perform the reordering. These two filters may then be vertically fused by the compiler.

2.2.2 Fission Transformations

A stream program can be parallelized by applying *fission* transformations to it. *Vertical fission* produces pipeline parallelism by dividing a single filter into multiple pipelined filters. This does not appear to be implemented in the StreamIt compiler [3]. Data parallism can be exploited in a stateless filter by placing multiple copies of the filter within a splitjoin. This is called *horizontal fission* and is illustrated in Figure 2.2.2.

2.3 Layout and Communication Scheduling

Once the stream graph has been partitioned, the StreamIt compiler performs layout and communication scheduling phases to map the graph to hardware [3]. The mapping process is dependent on the hardware architecture.

The layout phase assigns filters from the final stream graph to computation elements in the hardware. For a particular architecture, the layout is computed using the set of legal layouts for that architecture, a cost function that measures the cost of a particular layout based on the communication properties of the stream graph, and a perturbation function. The compiler uses simulated annealing to choose a good layout for the stream graph.

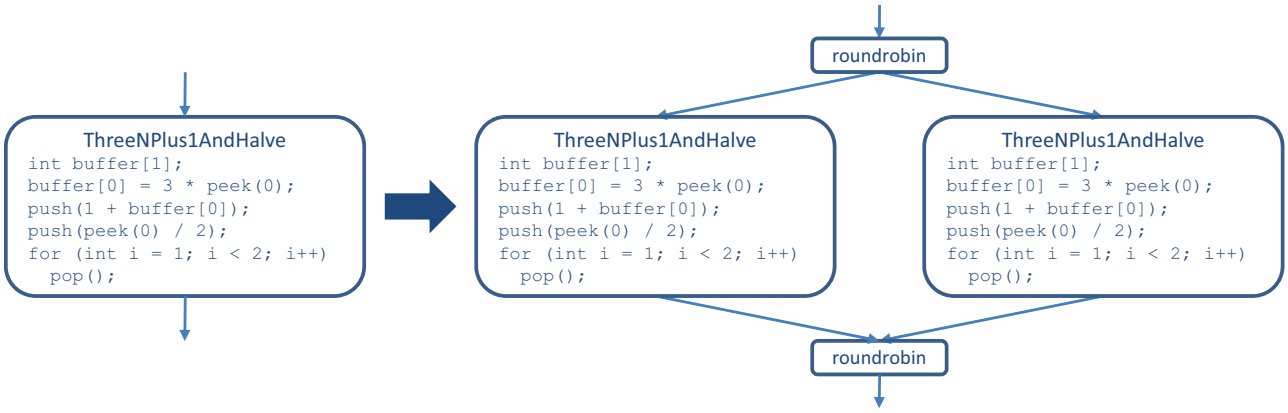

Figure 4: An example of horizontal fission.

Once a layout has been chosen, communication is scheduled between the computational elements in the hardware. Since I/O rates are known at compile-time, this can be done statically. The scheduling algorithm is dependent on the communication model of the underlying hardware.

3 StreamIt Characteristics Suitable for Fleet

The StreamIt language has many features that make it an appealing candidate for compiling to Fleet. This section discusses some of these features.

3.1 Graph Structure

Perhaps the most novel feature of StreamIt is the heirarchical structure it imposes on the stream graph. The difference between these structured graphs and arbitrary stream graphs is likened by the StreamIt authors to the distinction between structured control flow and jump statements [6]. In exchange for restricting expressiveness in some cases, the structure makes the resulting program easier to analyze and compiler as well as easier to read and less error-prone.

Since a Fleet implementation is likely to have a hierarchical layout of ships, StreamIt's hierarchical structure may make it easier to compile to Fleet than a less structured language. Even with a flat Fleet implementation, the structured communication pattern of a StreamIt program would make it easier to schedule communication on Fleet hardware.

3.2 Explicit Communication

The stucture of a StreamIt program cleanly separates communication from computation. All computation is performed inside of filters, and filters can only communicate through the pop, peek, and push operations and the messaging system. Filters can only access local memory, so implicit communication through memory accesses is impossible.

StreamIt's explicit communication model is suitable for Fleet since all communication is explicit in Fleet. In addition, communication between StreamIt filters can be done over the switch fabric instead of through memory as would be done on a multicore implementation of StreamIt.

3.3 Static I/O Rates

Assuming StreamIt filters can only have static I/O rates, as multiple StreamIt papers have suggested (see §2.1), communication can be completely scheduled statically for a particular StreamIt program. Static scheduling of the Fleet switch fabric would ensure that it was being used as efficiently as possible while providing a guarantee against deadlock. In addition, the instruction count can be minimized by using repeating instructions.

3.4 Automatic Parallelization

The fission transformations described in §2.2.2 allow a StreamIt program to be parallelized by the compiler. A StreamIt program could thus scale to the size of a particular Fleet implementation without requiring the programmer to be aware of the size.

4 StreamIt Characteristics Unsuitable for Fleet

The StreamIt language also contains numerous features that are difficult to implement on Fleet.

4.1 Filter Size

As discussed in §3.2, the filter abstraction cleanly separates communication from computation. However, this assumes that each filter can run entirely on a single computational unit of the target architecture. Unfortunately, this is not the case for Fleet, as the computational units are simple ships. Since StreamIt filters can contain more or less arbitrary sequential code, compiling a filter to Fleet appears to be nearly as difficult as compiling any other sequential language to Fleet.

4.2 Peeking

Filters that peek beyond the front of their input queues require their input to be buffered and indexable. Since neither the switch fabric nor even FIFO ships are indexable, this implies that each such filter must have an associated scratchpad to store its input. Sharing a scratchpad among multiple filters would make it a bottleneck, increasing the input latency to those filters.

4.3 Single Input and Output

StreamIt filters can only have a singal input and output, and only basic splitters and joiners are allowed. This requires programmers to perform various contortions in order to write certian simple codes. For example, if the programmer wishes to add two vectors, the streams representing these two vectors must first be interleaved before each pair of elements can be added, as illustrated in Figure 4.3. Other cases are downright impossible, such as performing the merge step of a merge sort without knowing how many items there are, since the two substreams must be joined using an irregular pattern.

Unlike StreamIt filters, Fleet ships can have multiple inputs or outputs. Since the single input and output requirement introduces serialization where it isn't necessary in the hardware, StreamIt programs may not run at full efficiency on a Fleet implementation.

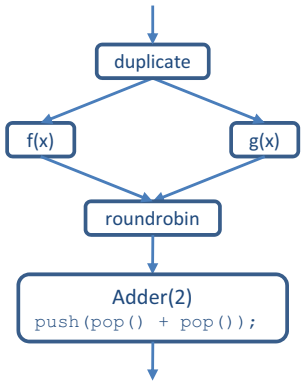

Figure 5: A stream that computes $f(x) + g(x)$. The partial results must be interleaved before being added.

4.4 Out-of-Band Messages

StreamIt's messaging system allows messages to be timed relative to data movement between the sending and receiving filters. When the message sender is upstream of the receiver, the message can be synchronized to data produced in the past. This appears to require all data to be held for as long as a future message may be synchronized to it. This is inefficient on even a synchronous network, but Fleet's asynchronous switch fabric may complicate things even further.

5 Conclusion

The StreamIt language introduced many features and restrictions that made it appropriate for communication-exposed architectures such as Raw. Its hierarchical structure and static communication model allow programmers to write code that takes maximal advantage of such architectures while avoiding many of the pitfalls of parallel programming.

However, StreamIt was not designed with Fleet in mind, so some of its features may prove to be very difficult to implement on Fleet. Instead, it may serve as an inspiration for a new language designed for Fleet that includes the positive features of StreamIt but removes those features that cannot be mapped efficiently to Fleet.

References

[1] Streamit language specification, version 2.1, September 2006. http://cag.csail.mit.edu/streamit/papers/streamit-lang-spec.pdf.

[2] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[3] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, New York, NY, USA, 2002. ACM.

[4] I. E. Sutherland. FLEET - A One-Instruction Computer, August 2005. http://research.cs.berkeley.edu/class/fleet/docs/people/ivan.e.sutherland/ies02-FLEET-A.Once.Instruction.Computer.pdf.

[5] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002.

[6] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.