A Computing Origami: Folding Streams in FPGAs

Andrei Hagiescu, Weng-Fai Wong
School of Computing
National University of Singapore
{hagiescu, wongwf}@comp.nus.edu.sg

David F. Bacon, Rodric Rabbah
IBM T.J. Watson
Research Center
{bacon, rabbah}@us.ibm.com

ABSTRACT

Stream processing represents an important class of applications that spans telecommunications, multimedia and the Internet. The implementation of streaming programs in FPGAs has attracted significant attention because of their inherent parallelism and high performance requirements. Languages, tools, and even custom hardware for streaming have been proposed, some of which are commercially available.

There are several significant challenges to realizing streaming applications directly in hardware (FPGAs). Since FPGAs have finite resources, there are often many non-trivial tradeoffs between processing throughput and overall latency. In this paper, we describe an algorithm that computes refinements of stream graphs into designs that optimize processing throughput subject to user-specified area and latency constraints.

Categories and Subject Descriptors

B.6.3 [**Logic design**]: Design aids—*Optimization*

General Terms

Algorithms, Design, Performance

Keywords

FPGA, Streaming, Throughput, Latency

1. INTRODUCTION

There are several existing platforms today that integrate FPGAs with microprocessors, and recent announcements (e.g., [11]) from leading vendors suggest that FPGAs are likely to become widely available as programmable coprocessors. A broad class of applications, including multimedia, networking, graphics, and security codes, provide ample opportunities to exploit FPGA-based acceleration. Sequential parts of a program can be assigned to run on the host processor while the parts of the program with abundant parallelism can be synthesized directly in the FPGA.

DAC 2009, July 26 - 31, 2009, San Francisco, California, USA.

In this paper we focus on a class of applications where ample parallelism is available as a result of stream-oriented processing. Stream processing is a data-centric execution model that is dataflow-driven. Streaming codes are naturally expressed as graphs of *filters* that communicate through FIFO data channels. Dependencies between filters are made explicit by the communication channels. Each filter has its own control flow logic and an independent address space, and it executes repeatedly as long as a sufficient number of tokens are available on its input channels.

Our work addresses the following issue: is there a refinement of an input stream graph that can maximize the processing throughput of the overall graph? Furthermore, because FPGA area is finite, and because latency is typically an important consideration in real-time streaming codes, we are interested in maximizing throughput subject to area *and* latency constraints. As far as we know, we are the first in tackling this combined problem.

The intuition behind our proposed algorithm is the following. We inspect the stream graph to identify filters that cause bottlenecks. We observe that if the filters are stateless – they do not maintain a history of their past execution – then we can use data parallelism to increase their throughput. This is achieved by judiciously replicating the bottleneck filters.

Replicating filters has several advantages. The replicated filters do not require resynthesis as they are all instances of the same filter, and the synthesis results are reusable. This is in contrast to prior work on global optimization of loop nests on an FPGA [12] which requires recompilation and evaluation of the recompiled designs based on heuristics. Such an approach will not scale for large designs.

Our algorithm operates on a stream graph, and a set of synthesized filters corresponding to the nodes in the graph, and determines how to assemble the synthesized filters to achieve the best possible throughput. If a filter is replicated, we use a simple hardware template to route dataflow to and from the replicated filters. This approach also makes the issue of filter synthesis orthogonal to design assembly and generation. Hence this work is complementary to a lot of the ongoing research in the community that address high-level synthesis of the filter code itself.

Our algorithm is briefly described as first aggressively replicating candidate filters (graph unfolding), then refolding the graph to reduce the number of replicas if they are not profitable. The next sections provide a motivating example and discuss related work. Next we describe our stream folding algorithm and present the results of our evaluation.

2. EXAMPLE AND BACKGROUND

A stream graph is shown in Figure 1. It consists of three kinds of nodes. A *splitter* node distributes an input stream to other nodes. A

filter consumes an input stream, performs some computation, and outputs a new stream. A *joiner* aggregates streams, and outputs a single new stream. In the figure, $F1$, $F2$, and $F3$ are filters, the hardware footprint of each filter is correlated to the area of its corresponding rectangle, and the execution latency of the filter is correlated to the length of the rectangle. The splitter and joiner are illustrated using arched double-headed arrows. The edges in the graph describe the dataflow between nodes. Each edge corresponds to a FIFO connecting two nodes together.

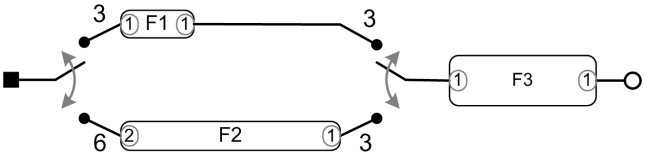

Figure 1: An example stream graph.

Each of the nodes in the stream graph executes when there is a sufficient number of data tokens on its input edge. In the figure, the number of input tokens required to execute each filter is shown in a circle on the left end of each rectangle. The number of data tokens produced in each execution of the filter is shown in a circle on the right side of the rectangle. For example, the filter $F2$ requires 2 data tokens to execute, and when it does, it produces 1 new data token. An execution of a filter is also called a firing.

We use StreamIt [10] to describe stream graphs programmatically and algorithmically. StreamIt is an architecture-independent stream programming language that allows a programmer to focus on describing the algorithm's dataflow (i.e., graph topology) without committing to an implementation or buffering strategy. Each filter in StreamIt declares its data input and output rates per execution. This explicit rate information enables many optimizations that can yield efficient implementations of the stream computation [3, 2, 6]. An example filter declaration is as follows.

```
int->int filter F2(int N) {
  work peek N pop N push N/2 {
    for (int i = 0; i < N/2; i++) {
      int x = pop(); // read/dequeue from input FIFO
      int y = pop();
      push(x+y);     // write/enqueue to output FIFO
    }
  }
}
```

Filters read data from their input FIFO using *pop* statements, and write data to their output FIFO using *push* statements. The filter may have instantiation parameters (`N` in this case), and always encapsulates its computational logic in a *work* function. Filters may also *peek* at their input data, without altering the state of the FIFO. Peeking is useful for sliding-window computations, and provides an opportunity to optimize filters that otherwise require internal buffering (i.e., state) to preserve previous values.

In Figure 1, the dataflow is split between $F1$ and $F2$ in a periodic and round-robin manner, with 3 tokens dispatched to $F1$ and 6 to $F2$. This information is annotated on the edges that fan-out from the splitter. Similarly, the joiner collects data from the input streams in a round-robin manner. The weight annotations on each edge describe how the data is aggregated from the streams: 3 tokens from $F1$ and 3 from $F2$.

A StreamIt program exposes the communication topology to a compiler or synthesis tool that can then decide on the best implementation choices depending on the target platform. The stream graph in the figure can be described as follows in StreamIt.

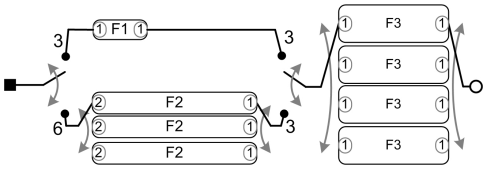

Figure 2: A stream graph with replicated filters that achieves maximum throughput, subject to some area constraint.

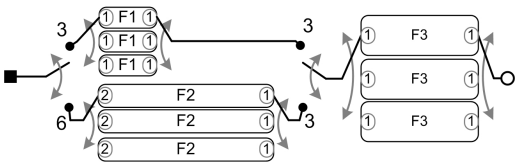

Figure 3: Reducing the latency for the graph in Figure 2 under the same area constraints.

```
int->int pipeline Example() {
  add pipeline {
    add splitjoin {
      split roundrobin(3, 6);
      add F1();
      add F2(2); // instantiating parameterized filter
      join roundrobin(3, 3);
    }
    add F3();
  }
}
```

A *pipeline* in StreamIt is a stream container, connecting a sequence of streams together. Here, the pipeline consists of a *splitjoin* connected to $F3$. The splitjoin is also a stream container, with a splitter at the source, a joiner at the sink, and filters (or other stream containers) between them. The expression *roundrobin(3, 6)* describes the weights of each edge between the splitter and the sibling streams, and similarly for the joiner.

In a stream graph, nodes fire autonomously and concurrently. Since there is no data dependence between $F1$ and $F2$, they can fire in parallel. The firings of filter $F3$ can be pipelined relative to the other filters. It is obvious from the graph that $F1$ and $F2$ execute the same number of times in order for $F3$ to fire (that is, both $F1$ and $F2$ fire 3 times to produce the requisite amount of data for the joiner, which ultimately provides the data to filter $F3$).

A trivial mapping of the stream graph in Figure 1 into hardware does not produce an efficient implementation: the filters $F1$ and $F2$ are not load-balanced. However, if a filter is stateless – that is, it does not maintain any history of its previous executions – we may *replicate* it to achieve a more load-balanced implementation. Replicating a filter creates a new instance of the filter and adds a splitter to distribute data between the filter and its replicas, and a joiner to collect the new data. The replicas effectively increase the firing rate capability of the filter, but also increase hardware costs: each of the replicas incurs a space overhead in hardware that is equal to the original filter, and in addition, there is an added overhead incurred by the splitter-joiner pair that routes the new dataflow. Since we are interested in realizing stream graphs in hardware, and specifically FPGAs with finite resources, we have to judiciously decide which filters to replicate and to what extent. Our algorithm unfolds and refolds a given stream graph to determine where and to what extent replication will be most profitable. We will generally refer to this process of stream graph refinement as *stream folding*.

The stream graph in Figure 2 illustrates the replication of filters $F2$ and $F3$. The graph achieves the best throughput to area ratio: filters fire continuously, making efficient use of the hardware. At steady state, the throughput of the joiner aggregating the outputs of

$F1$ and $F2$ is three times better than the corresponding joiner in the original graph shown in Figure 1. A hardware design and implementation of a stream graph that uses replication increases throughput, but may also affect the latency of the computation. Figure 3 presents an alternate stream folding strategy that tradeoffs throughput to achieve a lower overall latency.

Our stream folding strategy manages the complexity of the design search space by exploiting the hierarchical nature of the stream containers. Peeking filters may be replicated if they appear at the root of a container (e.g., the first filter in a pipeline). The strategy is simple: duplicate the input stream to the filter and its replicas, and then locally discard the parts of the stream that are not relevant to a particular filter. This approach may create a lot of redundant communication, but others have shown that it is possible to design efficient hardware mechanisms to exploit the structured data reuse [4].

3. RELATED WORK

The work that we present is founded on the idea of judiciously replicating filters to increase the throughput of bottleneck filters. In the context of streaming processing, this idea has been explored in the past. For example, [3] describes a greedy strategy to map filters to a multicore architecture. In that work, if the number of filters is less than the number of cores, the compiler replicates the filters (called fission) with the highest computation requirements. If the number of cores is less than the number of filters (which is the common case), the compiler fuses together the filters or stream containers with the least computation until the number of fused filters equals the number of cores. While this strategy works well when mapping to a platform with a finite number of compute engines, it is not clear how to adapt it for an FPGA where the number of compute engines is undefined. Furthermore, the strategy in [3] is not practical if the stream graph consists of filters that are wildly unbalanced since in that case some filters require fission and others require fusion. Our stream folding algorithm is designed to address this issue in the context of FPGAs. The algorithm essentially performs maximal fission of all candidate filters bounded only by the size of the FPGA, then fuses the filters that do not positively effect throughput.

Other relevant stream graph refinements include [2] and [6]. In the former, the cost of the communication introduced by replication is reduced by first fusing filters then replicating the coarsened filters. This reduces the amount of pipeline parallelism that can be effectively exploited in an FPGA. In the latter, an integrated fission and partitioning strategy is offered to replicate filters and assign graph partitions to a finite and predetermined number of cores. The graph partitions are then carefully scheduled using a staging algorithm that attempts to overlap communication and computation. Our algorithm considers and accounts for the latency of the added communication when it determines the extent of replication. Unlike in past work where the communication cost is high and not homogeneous, we can benefit from the FPGA architecture to optimize communication and more accurately account for the added overhead.

This paper does not address the synthesis of the actual computation from StreamIt code (namely the work functions) to a hardware description language. Our emphasis is on the composition of the synthesized modules into an overall space-time efficient design. Recent work [5] specifically addressed the issue of hardware generation from StreamIt, and we believe that work is complementary to our work. Similarly, many of the existing state of the art technologies in this regard can be used to complement our work.

There is also a significant amount of work geared toward improving performance using multiple clocks to drive heterogeneous processors [7], or heterogeneous accelerators synthesized in FPGAs [9]. However, the latter work assumes that the number and type of accelerators in a design is fixed, and the heterogeneous clock assignment finds the optimal set of clock frequencies to assign to each accelerator. The primary contribution of our work compared to these published methodologies is the co-optimization of space and time (latency). Our starting input is a stream graph extracted from a stream program, from which we derive the set of hardware modules to synthesize. In effect, we are simultaneously determining the number and types of "accelerators" to synthesize.

Algorithm 1: Area / throughput design folding

Input: StreamIt program(S), area constraint (AREA)
Output: Replication coefficients

1 **foreach** *Filter* f *in* S **do**
2 | $workFactor[f] = f.latency \cdot S.runs(f)$;
3 | $designPointArea+ = f.area \cdot workFactor[f]$;
4 **end**
5 $scaleLimit = \min_{f.hasState}(\frac{1}{workFactor[f]})$;
6 $scaling = min(AREA/designPointArea, scaleLimit)$;
7 **foreach** *Filter* f *in* S **do**
8 | $replication[f] = \lceil workFactor[f] \cdot scaling \rceil$;
9 **end**
10 **while** $area(replication) > AREA$ **do**
11 | $replication = reduceThroughput(replication)$;
12 **end**

Procedure `reduceThroughput` (*replication*)

min = ∞; $t^S_{out} = throughput(replication)$;
foreach *Filter f in S* **do**
| $\delta t = t^S_{out} - throughput(replication.reduce(f))$;
| **if** $\delta t < min$ **then**
| | $min = \delta t$;
| | $candidate = f$;
| **end**
end
return $replication.reduce(candidate)$;

4. STREAM FOLDING

We propose an algorithm that determines which filters to replicate (and by what factor), in order to maximize processing throughput subject to area and latency constraints. Our philosophy is to describe the desired design topology, and instantiate an implementation that simply stitches together the filters and streams as directed by our algorithm. The algorithm assumes that individual filters are already synthesized and both area and behavior (worst-case execution time estimates) information is retrieved from the implementation. If the filters take less time to execute than the worst-case estimate, the correctness of the solution is not affected.

The input to our algorithm is a stream graph, which we derive from StreamIt code. We refer to our graph refinement strategy as stream folding because we first replicate filters to expose data parallelism (graph unfolding) and then refold the graph judiciously to achieve a desired throughput subject to one or more constraints.

We derive a high-throughput design using the steps shown in Algorithm 1. First, we inspect each filter and compute its work factor by multiplying its latency and its firing rate (line 2). The firing rate (*S.runs(f)*) is calculated by the compiler using a Single Appearance Schedule [1]; it equals the number of firings of a filter so that it is rate-matched to its producer and consumer. The computed work factors determine the total area of an initial design point. Lines 5-6 determine the maximum replication factor that matches the area constraint. Stateful filters are not replicated (although past work has shown it may be profitable to do so [6]), and they impose a scaling limit. If a stateful filter dominates the execution, then repli-

Algorithm 2: Latency constrained design folding

Input: Best throughput configuration (baseRepl)
Output: Latency constrained configuration (latRepl)

```
1  latRepl = null ; T = ∞;
2  while throughput(baseRepl) ≤ T do
3      if feasibleImprovement(baseRepl) then
4          candidates = simAnnealing(baseRepl, T);
5          foreach candidate in candidates do
6              if throughput(candidate) < T then
7                  latRepl = candidate;
8                  T = throughput(latRepl);
9              end
10         end
11     end
12     baseRepl = reduceThroughput(baseRepl);
13 end
14 return latRepl ;
```

cation of other filters is not likely to be profitable. Line 5 determines which of the stateful filters constrains the replication; it is the stateful filter with the greatest work factor. The first term in the *min* equation on line 6 determines how much of the FPGA resources are available for replication. The resulting scaling factor is used to determine the replication counts for all filters (line 8). Due to rounding, a design that realizes the calculated replicas may have an area slightly larger than the specified area constraint although the design will be on the pareto-optimal frontier with respect to throughput and area. Finally, we refine the design, reducing its throughput while maintaining it on the pareto-optimal front of the design space. Each iteration in lines 10-12 reduces the area of the new design by eliminating filter replicas one at time, starting with filter replicas that only marginally improve throughput. Finally, a maximum throughput design that fits the available area is obtained, and we next determine if it satisfies the latency constraint.

Algorithm 2 generates base designs sorted by throughput, starting with the one achieving the highest throughput as constructed in the previous step. We use simulated annealing to search through the potential candidates while pruning away as much of the infeasible design space as possible. We do this using a custom neighbor visit function that avoids illegal configurations defined by the area constraint, throughput bounds and the maximum synthesizable replication.

The latency of each evaluated design depends on the actual input arrival rates, and the latency constraint may be satisfied only for arrival rates lower than the maximum sustainable by the base design. Finding such a solution adds a lower bound to the throughput of subsequent base designs (line 5-9). Only base designs above this bound are tried (line 2) as they can offer additional replication possibilities (more spare area is available to selectively increase replication) thereby yielding solutions with better throughput than those previously identified.

As long as a candidate is found in one of the steps of the exploration, the search converges easily, being limited to a few tightly constrained simulated annealing steps. However, if no candidate is found, a larger number of possible base designs is explored. We further prune them by checking if an area unconstrained design having the same throughput as the base design can offer the required latency (line 3). Such a design is obtained by replicating all filters except the bottleneck by as much as possible.

4.1 Calculating Throughput

We use the hierarchical nature of the stream graphs derived from StreamIt to efficiently compute the overall throughput of a streaming program. The maximum input throughput t_{in} and output throughput t_{out} of a filter F_i is defined as follows

$$t_{in}(i) = \frac{pop(F_i)}{latency(F_i)}, t_{out}(i) = \frac{push(F_i)}{latency(F_i)}$$

where $pop(F_i)$ and $push(F_i)$ respectively equal the number of data elements dequeued from and enqueued to the input and output FIFOs of the filter F_i, and $latency(F_i)$ equals the number of cycles spanned by a single firing of F_i. The pop and push values are readily available from StreamIt programs. The throughput of the stream containers follows.

Pipeline and SplitJoin throughput.

The throughput of a pipeline is equal to the lowest throughout of its filters. Furthermore, since individual filters may push and pop at different rates, the rates observed at different points in the pipeline will vary, although filters have to sustain correlated rates. We define the throughput limitation imposed by a filter F_i on the output of a pipeline consisting of the filters $P = \{F_1, \ldots, F_n\}$ as

$$t^P_{out}(i) = t_{out}(i) \cdot \prod_{i<j\leq n} \frac{push(F_j)}{pop(F_j)}$$

and therefore, the actual output throughput of the pipeline is

$$t^P_{out} = \min_{1\leq i\leq n} t^P_{out}(i).$$

For a splitjoin $SJ = \{F_1, \ldots, F_n\}$ where the joiner weights are $(w_1, \ldots, w_n)$, the output throughput is

$$t^{SJ}_{out} = \min_{1\leq n} \left(t_{out}(i) \cdot \frac{\sum_{1\leq j\leq n} w_j}{w_i} \right).$$

Overall throughput.

It is possible to apply these relations to the whole stream graph in a composable manner,

$$t^S_{out} = \min_{i\in S} t^S_{out}(i) = \min_{i\in S} (C_i \cdot t_{out}(i))$$

where C_i is a constant that can be determined by analyzing the unmodified stream graph. To prove this relation, assume a stream can sustain a throughput $t' > t^S_{out}$. Propagating this downwards through the stream hierarchy, for all stream containers $\hat{S}, t^S_{out}(\hat{S}) \geq t'$. Continuing the stream decomposition and applying this relation down to individual filters, results in $\forall i, t^S_{out}(i) \geq t'$, which is a contradiction.

The replication of a filter has the effect of multiplying its throughput by its *replication factor*. If r_i is the replication factor, we can modify the above formula to

$$t^S_{out} = \min_{i\in S} (r_i \cdot C_i \cdot t_{out}(i)).$$

4.2 Calculating Latency

We envision that stream graphs will run on FPGAs that are coupled to host processors. Data transport between the host and FPGA is achieved through a bulk transfer mechanism (e.g., DMA). We call the number of clock cycles between such transfers the *initiation interval* (II). The minimum initiation interval can be computed based on the reciprocal of the highest sustainable throughput in the stream graph. Results are expected to be ready after a time interval called the *latency*.

Data-token reordering and local congestion at a filter's input due to non-periodic data arrival are the major factors for latency variation. While replication improves throughput, it often increases the latency. We believe it is important to obtain exact latency bounds

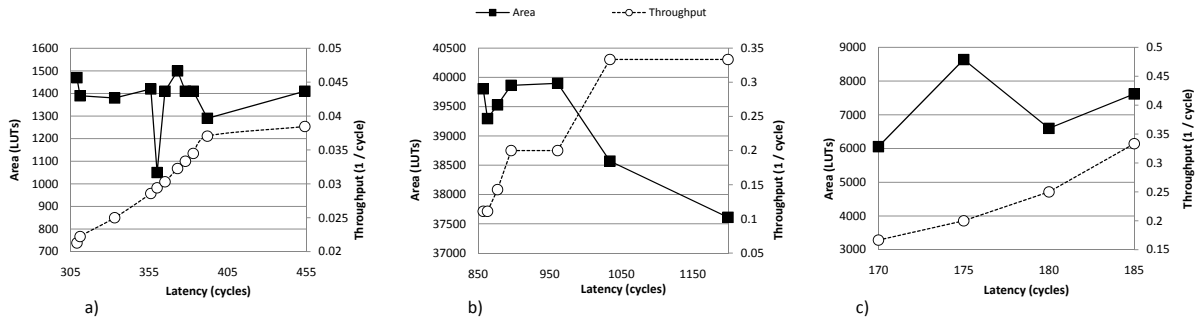

Figure 4: (a) Synthetic example, (b)FFT2, (c) Matrix multiply.

Figure 5: Replication factors for instances of filter *CombineDFT* in FFT. The dotted line separates the replication required to achieve the maximum throughput for a specific design point from the additional replication introduced to decrease latency.

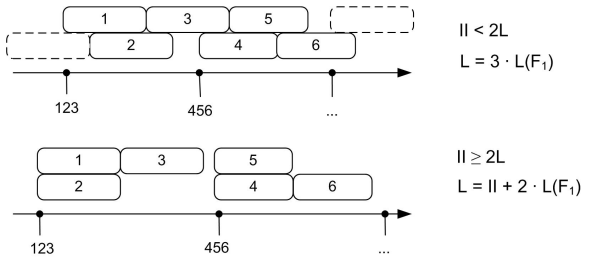

Figure 6: Schedule used to determine latency. Three data tokens arrive every II. With two replicas, computation occurs in parallel.

Table 1: Example latency calculation.

Input	replica 1	replica 2	Constraint	replica 1	replica 2	Constraint
0	$[0, L)$		$II \geq \frac{3L}{2}$	$[0, L)$		$II \geq 2L$
0		$[0, L)$			$[0, L)$	
0	$[L, 2L)$			$[L, 2L)$		
II		$[L, 2L)$			$[II, L)$	
II	$[2L, 3L)$		$II < 2L$	$[II, II + L)$		
II		$[2L, 3L)$			$[II + L, II + 2L)$	
	Interval: $[\frac{3L}{2}, 2L)$			Interval: $[2L, \infty)$		

that can offer guarantees especially for real-time stream performance.

Given a stream graph, we determine a valid II where the same set of delays hold. There is a finite set of such intervals and they can be computed starting from the minimum sustainable II. We capture the event arrival time at each filter input as a linear expression $\alpha II + \beta$ and we derive the output time of the result as a linear expression, generating an additional constraint on the upper bound of the II where necessary. We process all filters, until we obtain the overall latency of the stream and a constrained interval where the linear expression holds. We then analyze the adjacent interval, generating a new constraint on the II that will define new intervals recursively.

Table 1 shows the computations necessary in case of a single filter, replicated two times, with 3 input tokens appearing each initiation interval. The corresponding schedule is presented in Figure 6.

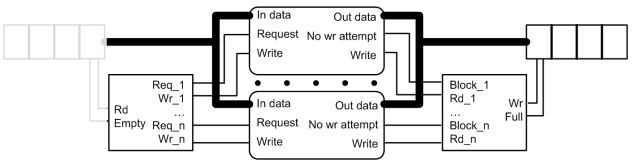

Figure 7: Replication mechanism in hardware.

Our implementation hierarchically iterates over the stream structure, deriving output times based on the input times. In case of replicated filters, it maintains a set of ready times for each replica as linear dependencies on II. The input tokens are already in order and ordering constraints are generated to ensure that the current replica is ready to fire when its data arrive.

4.3 Supporting Hardware

The distribution and gathering of data to and from replicas require hardware resources akin to programmed multiplexers. Our design for such a mechanism is illustrated in Figure 7. The design permits a single data token to be routed per clock cycle. We have automatically synthesized the replication logic for a wide range of replication factors to obtain its associated resource requirements.

5. RESULTS

We evaluated our stream folding algorithm using the streaming benchmarks provided with the StreamIt compiler. These provide us with realistic stream graphs and cover filters with a wide range of area and latency requirements. We also included a synthetic benchmark to check the performance of our implementation beyond the available benchmarks. We implemented our algorithm in Java as a backend for the StreamIt compiler, using Opt4J [8] to perform simulated annealing. For the purpose of achieving low latency designs, we impose a set of constraints on the HDL generation of the filters. These constraints are easily accommodated by current hardware compilers. Namely, we require that a filter reads all of its inputs in consecutive clock cycles. Similarly, we require that

Table 2: Design space solutions under various constraints.

	Minimum area			Best throughput			Constrained design				
Stream	LUTs	Latency	II	LUTs	Latency	II	LUTs	Latency	II	Constraint	Run time
MatrixMult	1498	480	19	7618	185	3	4558	175	7	$Latency \leq 175$	1.14s
Serpent	3028	1027	4	3878	773	2	3053	901	4	$Latency \leq 910$	0.73s
FFT2	37610	1199	3	43370	764	2	39530	868	7	$AREA \leq 40000$ $Latency \leq 880$	34.7s
FMRadio	37458	371	39	87564	371	13	62511	371	20	$Area \leq 65000$	1.01s
DCT	45752	349	3	137256	349	1	91504	349	2	$AREA \leq 120000$	0.73s
BitonicSort	43920	1042	3	131760	1042	1	47400	1282	2	$AREA \leq 50000$	18.3s
Synthetic example	350	309	135	15990	504	2	1490	309	47	$Latency \leq 309$ $Area \leq 1500$	0.43s

a filter writes its output in consecutive cycles. This ensures that arriving data does not block during distribution unless all the filters are currently busy. We perform a one-time hardware synthesis and platform mapping for each filter to empirically determine its resource usage (e.g., area requirements) for the Virtex 4 FPGA architecture, considering LUTs, DSP blocks and Block RAM as part of the area metric.

We explored the throughput achievable under arbitrary area and latency constraints. Our design-space exploration identifies lower latency implementations if it is acceptable to degrade the throughput. As shown in Figure 4, the area of a design is not trivially correlated with either latency or throughput.

We closely inspected FFT2 (a fine grained implementation of FFT) and modified the benchmark so that each floating-point operation is encapsulated in a filter. This allows us to explore the possibility of using stream folding to determine an optimized number of floating-point units to use in a design. We used Xilinx LogiCORE IPs as building blocks for these filters. The total number of filters in this FFT implementation is 118 filters (compared to 22 in the original benchmark). The results are shown in Figure 4(b). The graph shows that there is a significant opportunity in stream folding.

A benchmark that has a tighter range of latency variation is matrix multiply (Figure 4(c)). We found that no solutions were possible if the latency was constrained any tighter than shown. Note that the benchmarks exhibit non-monotonic relationships between througput and area.

In Figure 5 we represent replication factors of different design solutions of the original FFT. Each group of bars represent the replication factors for instances of the CombineDFT filter (the main computational filter in FFT2) in a particular design point. The dotted line for each group of bars represents the overall replication factor that yields the same maximum throughput for that design point. Designs that are subject to lower latency constraints require greater replication of more filters.

Table 2 shows several design points obtained using our algorithm. For each benchmark, we show the design solution that has (a) the minimum area, (b) the best throughput (area limited only by the device size we considered, a VFX140), and (c) a constrained design between the two. The results are meant to demonstrate the versatility of our algorithm.

Most of our benchmarks are explored in a few seconds on a Core2 Duo 2.33GHz, as long as individual filter replicas monotonically contribute toward lower latencies. In cases with tight latency constraints, joiners might introduce notable adverse latency increases that degrade the convergence of the our algorithm. During our extensive testing, which included additional synthetic benchmarks, the longest runtimes were on the order of minutes.

6. CONCLUDING REMARKS

We have studied the problem of mapping high level descriptions of streaming applications in the form of stream graphs into FPGAs. We proposed replication as a mechanism to increase processing throughput. Our algorithm yields solutions that satisfy area *and* latency constraints. We therefore have a means to automatically and efficiently realize stream applications directly in hardware. Although we have found the scaling of the exploration algorithm to be satisfactory (up to 118 filters), we would like to further investigate and fine-tune the algorithm by introducing more heuristics to guide the design space exploration. Finally, we would like to generalize the technique to decouple filters from each other so as to implement them in different clock domains.

7. REFERENCES

[1] S. Bhattacharyya, P. Murthy, and E. Lee. Kluwer Academic Press, 1996.

[2] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS'06*.

[3] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. Meli, A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS'02*.

[4] Z. Guo, B. Buyukkurt, and W. Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. *SIGPLAN Not.*, 39(7):249–256, 2004.

[5] A. Hormati, M. Kudlur, D. Bacon, S. Mahle, and R. Rabbah. Optimus: efficient realization of streaming applications on fpgas. In *CASES'08*.

[6] M. Kudlur, , and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI '08*.

[7] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO '03*.

[8] Optimization framework for java. http://www.opt4j.org/.

[9] S. Sirowy, Y. Wu, S. Lonardi, and F. Vahid. Clock-frequency assignment for multiple clock domain systems-on-a-chip. In *DATE'07*.

[10] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *CC '02*.

[11] AMD Unveils Torrenza Innovation Socket. http://www.hpcwire.com/hpc/917955.html, 2007.

[12] H. Ziegler and M. Hall. Evaluating heuristics in automatically mapping multi-loop applications to fpgas. In *FPGA'05*.