Automatic Generation of Streaming Datapaths for Arbitrary Fixed Permutations

Peter A. Milder, James C. Hoe, and Markus Püschel
Carnegie Mellon University
Electrical and Computer Engineering Department
Pittsburgh, PA, U.S.A.
{pam, jhoe, pueschel}@ece.cmu.edu

***Abstract*—This paper presents a technique to perform arbitrary fixed permutations on streaming data. We describe a parameterized architecture that takes as input n data points streamed at a rate of w per cycle, performs a permutation over all n points, and outputs the result in the same streaming format. We describe the system and its requirements mathematically and use this mathematical description to show that the datapaths resulting from our technique can sustain a full throughput of w words per cycle without stalling. Additionally, we provide an algorithm to configure the datapath for a given permutation and streaming width.**

Using this technique, we have constructed a full synthesis system that takes as input a permutation and a streaming width and outputs a register-transfer level Verilog description of the datapath. We present an evaluation of our generated designs over varying problem sizes and streaming widths, synthesized for a Xilinx Virtex-5 FPGA.

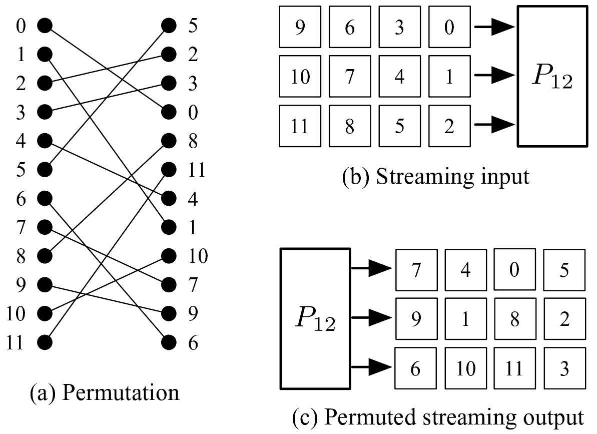

Fig. 1. Examples: permutation and streaming permutation.

I. Introduction

A permutation is a fixed reordering of a given number of data elements. We use P_n to represent a permutation of an n point data vector. Figure 1(a) shows an example of a 12 point permutation P_{12}. Data elements $(0, \dots, 11)$ enter concurrently from the left, are reordered, and exit in permuted order. A hardware implementation of such a permutation is trivial if all n data points are available concurrently: it is simply built as a reordering of wires.

However, this type of dataflow is only practical for small data vectors. Instead, hardware structures often utilize a streaming dataflow, where the n-word data vector is decomposed into w-word subvectors, which flow into the system over n/w consecutive cycles. We call w the *streaming width*, the number of words that flow in or out of the system per cycle. We treat the stream as continuous; once an n-word data vector finishes streaming into the system, a new n-word data vector begins entering on the following cycle.

Figures 1(b) and (c) illustrate a streaming version of the permutation seen in Figure 1(a). Here, $n = 12$ data words, and the streaming width is $w = 3$ data words per cycle. In Figure 1(b) we see that the data labeled $(0, 1, 2)$ enter the system on the first cycle, $(3, 4, 5)$ enter on the second, and so on, for a total of $n/w = 4$ cycles. Figure 1(c) shows the structure's output, where the 12 words flow out of the system in their new permuted order $(5, 2, 3, 0, \dots)$, again with $w = 3$ words per cycle.

Performing a permutation on streaming data is difficult because data must be reordered across time boundaries by storing and retrieving from a memory. For example, the element labeled 1 in Figure 1(b) streams into the system during the first cycle of the input stream, but must be buffered until the third cycle of the output stream (in Figure 1(c)).

A method for implementing streaming permutations must be able to scale as w (the number of data words per cycle) increases. Thus, using one large w-ported memory is infeasible; instead, a solution must use multiple memories and partition the problem across them.

Previous work on RAM-based structures for streaming permutations has been very specialized. For example, [1] gives a method to design streaming implementations of one family of permutations called stride permutations. [2] expands on this significantly, but its technique is still not general; its method only applies to a specific family of permutations, and only when n and w are powers of two.

In this paper, we propose a method that is capable of handling any arbitrary fixed permutation (P_n) streamed at any number of elements per cycle (w). Our technique uses multiple banks of simple memories, allowing it to scale well as w increases. Further, we provide a method to partition the problem such that there are no memory port conflicts, meaning the system sustain its full throughput (w words per cycle) without stalling.

Based upon this technique, we have implemented a fully automatic generation tool. The tool takes as input a permutation P_n and the desired streaming width w; it outputs the design in register-transfer level Verilog. Our designs are appropriate for implementation on field-programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs), but are particularly well-suited for FPGAs, which typically have many simple embedded memories.

Permutations are crucial building blocks in the fast computation of linear signal transforms, such as the discrete Fourier transform, the discrete cosine and sine transforms, and others. The streaming permutation structures we describe in this paper enable the design and generation of a wide variety of hardware implementations of these transforms (e.g., [3]).

Outline. In Section II, we provide background on permutations and streaming datapaths. Sections III, IV, and V explain our parameterized datapath structure, formulate the problem mathematically, and provide an algorithm for computing the solution. Section VI presents an evaluation of the designs built using the proposed technique. Lastly, Section VII discusses related work and Section VIII offers concluding remarks.

II. Background

In this section, we present relevant background on permutations. Then, we explain streaming datapaths and the indexing scheme we use to describe them mathematically.

Permutations. We consider arbitrary permutations on n data points, numbered $(0, \dots, n-1)$. For example, the cyclic shift is defined by

$$C_n: \ 0 \mapsto 1 \mapsto 2 \mapsto \dots \mapsto n-1 \mapsto 0. \qquad (1)$$

If C_4 is applied to data vector (x_0, x_1, x_2, x_3), the result is then (x_3, x_0, x_1, x_2).

An arbitrary permutation P_n is defined by its images $P_n(i)$, $i = 0, \dots, n-1$. For example, $C_n(i) = (i+1) \bmod n$.

Permutations can be equivalently viewed as matrices. We use the matrix representation that makes

$$C_n = \begin{pmatrix} 0 & & & 1 \\ 1 & & & \\ & \ddots & & \\ & & 1 & 0 \end{pmatrix}. \qquad (2)$$

For convenience, we use the same notation for a permutation and its associated matrix.

Streaming datapaths. In this paper, we consider streaming datapaths where an n point data vector is partitioned into w words per cycle over n/w consecutive cycles. We refer to w as the *streaming width* of the system. We assume the elements of such a streamed vector to be indexed by $0, \dots, n-1$. If $i \in \{0, \dots, n-1\}$ is an input index, the corresponding data word is located at port $i \pmod w$ and in cycle $\lfloor i/w \rfloor$. For example, the word labeled 7 in Figure 1(b) is located at port $7 \pmod 3 = 1$ in cycle $\lfloor 7/3 \rfloor = 2$.

In this work, we assume that n is a multiple of w. This can always be achieved by extending the permutation with fixed points: $(i \to i), \quad n \le i < (w \cdot \lceil n/w \rceil)$.

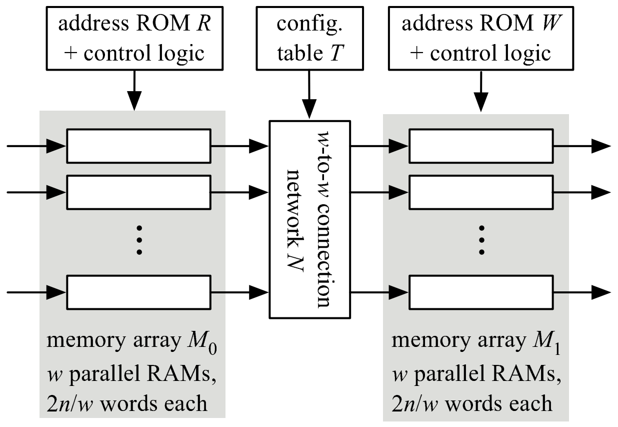
address ROM R + control logic
config. table T
address ROM W + control logic
w-to-w connection network N
memory array M_0 w parallel RAMs, $2n/w$ words each
memory array M_1 w parallel RAMs, $2n/w$ words each

Fig. 2. Datapath proposed in this paper.

III. Parameterized Datapath

A key aspect of this work is the identification of a scalable datapath structure that can be configured to perform any fixed streaming permutation with any streaming width. In this section, we present this parameterized datapath and discuss its operation.

Figure 2 shows the datapath we consider. It consists of two memory arrays M_0 and M_1, a connection network N, and lookup tables (ROMs) that contain address and control data (R, T, and W).

Each memory array contains w parallel memories, each with capacity $2n/w$. Each has one read port and one write port.[1] Memory arrays M_0 and M_1 are connected to lookup tables R and W (respectively), which hold pre-computed address values.

N represents a connection network that is capable of taking w data points as input and outputting them in any given order. We pre-compute its control data, and store the resulting values in ROM T. When w is a power of two, this network can be built as explained in [4], [5]. The resulting network utilizes $w \log_2(w) - w + 1$ 2-by-2 switches, which is asymptotically optimal. If w is not a power of two, N could be constructed in several ways; in this paper, we simply build the network for the next largest power of two. However, any other appropriate interconnection network or crossbar may be substituted. This decision does not affect the other portions of the datapath (M_0, M_1, R, and W).

As data flows through this structure, it passes through five stages:

1. Write into M_0. Data flows into the system and is written *in order* to the banks of M_0. Element i is written into address $\lfloor i/w \rfloor$ in bank $i \pmod w$. Address values are generated with a $\lceil \log_2(n/w) \rceil$ bit counter.

[1]It is also possible to replace the $2n/w$ word dual ported memory with two single ported memories of size n/w. Then, data is written into one memory while it is read out of the other. This optimization may be beneficial on an ASIC, where the designer can build precisely the necessary memory structures. However, this transformation is not beneficial on most current FPGAs, which typically have embedded dual-ported memories.

2. Read from M_0. On each cycle of this phase, one word is read from each of the w memory banks of M_0. The read addresses ($\lceil \log_2(n/w) \rceil$ bits each) are pre-computed and stored in lookup table R. Each line of R holds the w memory read addresses for a given cycle. Thus, R contains n/w lines, and each line requires $w \cdot \lceil \log_2(n/w) \rceil$ bits.

3. Connection network. The connection network N takes in w elements and outputs them in a permuted order. We pre-compute the values that will control the network, and store them in T, which contains n/w configurations, with $w' \log_2 w' - w' + 1$ bits per configuration (where $w' = 2^{\lceil \log_2 w \rceil}$).

4. Write to M_1. On each cycle of this phase, one word is written into each of the w memory banks of M_1. The write addresses are pre-computed and stored in lookup table W, each line of which holds w write addresses. So, W contains n/w lines, each of width $w \cdot \lceil \log_2(n/w) \rceil$ bits.

5. Read from M_1. Data are read from M_1 in order and flow out of the system. Element i is read from address $\lfloor i/w \rfloor$ of bank $i \pmod w$. Address values are generated with a $\lceil \log_2(n/w) \rceil$ bit counter.

In order to maintain full throughput across multiple problems, M_0 and M_1 are each sized to hold a total of $2n$ words. Then, n words can be written into addresses $(0, \dots, n/w-1)$ of each bank while the previous n words are being read from addresses $(n/w, \dots, 2n/w-1)$. We accomplish this by using a one bit register to determine whether or not to add an offset to the addresses flowing into the RAMs. Every n/w cycles, this bit is complemented.

The effect of this is that up to three n point data vectors can be active in the structure at one time. One set of n data points can be flowing into M_0 (step 1), while a second set flows between M_0 and M_1 (steps 2–4), while a third set of n points flows out of M_1.

An important aspect to understand about this datapath is that all of the reordering is done in stages 2–4. Stage 1 writes data to M_0 in natural order, and Stage 5 reads data from M_1 assuming it is *already* in permuted order. The problem then becomes: given a permutation P_n, how do we guarantee that, on each cycle, we can read w words from M_0 and write them to the correct locations of M_1 without conflicts (i.e., needing to read/write multiple words from/to the same RAM at the same time)? A solution to this problem implies a datapath configuration that will be capable of performing P_n with full throughput (w words per cycle) without stalling. We formalize this problem and derive a solution in Section IV.

Extension to support multiple permutations. In this work, we assume that each instance of the datapath performs one given permutation. However, the system can be easily extended to support m different given permutations by increasing the size of lookup tables R, T and W by a factor of m. Then, configuration data would be pre-computed for all m permutations, and an additional input would be added to the system to allow for run-time selection between the m sets of configuration data.

Relationship to previous work. The datapath we describe can be viewed as an extension of a structure from a different domain: the input-buffered crossbar switch [6], [7], [8]. This crossbar is used in switching applications, and is able to provide minimum throughput guarantees under certain input arrival assumptions. Our datapath differs from the crossbar in two major ways. First, we have added a second memory array (M_1) located at the connection network's output, which is needed to perform the reorderings we require. Second, we use lookup tables instead of online address computation. This is possible for our application because we can pre-compute all values at design time (while the input buffered crossbar switch must compute these values based upon system inputs).

IV. Problem Formulation

In this section we formulate a mathematical problem that corresponds to mapping a streaming permutation to the datapath presented in Section III. We demonstrate that this problem can be solved for all permutations and streaming widths.

As discussed in the previous section, the ordering of the data inside both memory arrays is fixed: in M_0, it must be in natural order and in M_1, it must be in permuted order. The key problem then is to choose w words each cycle that are read from the w different ports of M_0 that must be written to the w different ports of M_1. Formally, we define the problem as follows:

Problem 1. *Given are a permutation P_n on $I = \{0, \dots, n-1\}$ and a streaming width w. For each of the n/w time steps j, $j = 0, \dots, n/w-1$, find a subset $S_j \subset I$ containing precisely w points such that*

1) *the union of all S_j is I (which implies that the S_j are pairwise disjoint); and*
2) *for every S_j and every $k, \ell \in S_j$ where $k \neq \ell$:*
   $k \neq \ell \pmod w$ *and* $P_n(k) \neq P_n(\ell) \pmod w$.

A solution to Problem 1 will allow us to read (in cycle j) w elements (specified by S_j) from M_0 and write them to M_1. By construction, the w words will be read from w distinct memory banks in M_0 and written to w distinct memory banks in M_1. Since there are no conflicts at the read/write ports, the system can sustain a full throughput of w words per cycle as desired. Thus, a solution to this problem implies values for R, W, and T that will allow the datapath in Figure 2 to perform P_n.

Next, we transform Problem 1 into a form that enables its solution. We start by defining a mapping π_w that collects the set of connections needed between the output of M_0 and the input of M_1 in a matrix.

Definition 2. *The mapping π_w takes as input an $n \times n$ permutation matrix P_n and outputs a $w \times w$ matrix of integers. If $\pi_w(P_n) = [c_{k,\ell} \mid k, \ell = 0, \dots, w-1]$, then*

$$c_{k,\ell} = |\{x \in I \mid x (\text{mod } w) = \ell \text{ and } P_n(x) (\text{mod } w) = k\}|.$$

In words, the (k, ℓ) element of $\pi_w(P_n)$ gives the number of data words to be read from port ℓ of M_0 and written to port k of M_1.

For example, consider the permutation

$$P_{12} = (0, \dots, 11) \rightarrow \\ (3, 7, 1, 2, 6, 0, 11, 9, 4, 10, 8, 5). \tag{3}$$

This permutation corresponds to the example seen in Figure 1. If we assume $w = 3$ ports, we have

$$\pi_3(P_{12}) = \begin{pmatrix} 1 & 2 & 1 \\ 1 & 1 & 2 \\ 2 & 1 & 1 \end{pmatrix}. \tag{4}$$

Lemma 3. *The sum of all elements in a given row or column of $\pi_w(P_n)$ is n/w. A matrix with this property is called a* semi-magic square.

Proof. This follows from Definition 2 and the assumption that one word streams into each input (and out of each output) on each of n/w cycles. □

Lemma 4. *$\pi_w(P_n)$ can be decomposed into a sum of n/w many $w \times w$ permutation matrices, some of which may be repeated.*

Proof. This follows from [9], where it is proven that any semi-magic square can be decomposed into a sum of permutation matrices. The total number must be n/w due to Lemma 3. □

We represent this decomposition as

$$\pi_w(P_n) = \beta_0 Q_0 + \beta_1 Q_1 + \cdots + \beta_{k-1} Q_{k-1}, \tag{5}$$

where the Q_i are permutation matrices and the β_i are integer constants ≥ 1 with $\sum_i \beta_i = n/w$.

Each of the Q_i permutations represents a particular configuration of the switching network N in Figure 2. So, we can choose w corresponding words from M_0 (one from each bank), permute them by Q_i, and write them into the correct locations of M_1 (one word to each bank). Thus, the decomposition in (5) represents a solution to Problem 1.

Continuing our example from (3) and (4), we can decompose $\pi_w(P_{12})$ as

$$\pi_w(P_{12}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + 2 \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

Lemma 5. *$\pi_w(P_n)$ has a decomposition* (5) *that satisfies*

$$k \leq \min(w^2 - 2w + 2, n/w).$$

Proof. [10] gives an upper bound of $w^2 - 2w + 2$ permutations. We can further tighten this bound to $\min(w^2 - 2w + 2, n/w)$, since Lemma 4 shows us that we have at most n/w permutations. □

Lemma 5 shows that even though there are $w!$ many $w \times w$ permutation matrices, a much smaller number is needed in (5). For our datapath (Figure 2) this could lead to a reduced storage requirement for T since some of the k settings may be used multiple times (if $k < n/w$). Taking advantage of this would require storing the values of β_i and adding additional logic to determine when to increment T's address value. This optimization would reduce the number of elements in T from n/w to k, and hence may reduce the overall hardware cost for some problems (if the savings from storing fewer words offsets the added cost of storing the β_i and the added logic). We do not explore this optimization in this paper.

Because the structure proposed in this paper is an extension of the input buffered crossbar switch, our solution for mapping a permutation to the proposed datapath is similar to some approaches used in scheduling the crossbar switch (in particular, [8]).

V. Algorithm

Based upon the problem specified in Section IV, we formulate an algorithm that calculates the parameters needed to perform a given permutation P_n with a streaming width w on the datapath in Figure 2. This algorithm is executed at design time; it determines the control values to be stored into ROMs R, T, and W.

This algorithm computes a decomposition of $\pi_w(P_n)$ of the form (5) and calculates the corresponding values to store in R, T, and W. Recall, we use R and W to denote the collection of addresses for M_0 and M_1, respectively. $L = (Q_0, \dots, Q_{n/w-1})$ represents the list of permutations that the connection network must perform, and T represents the configuration bits associated with each permutation in L (computed using the methods in [4], [5]).

Algorithm 6. Input: P_n and w. Output: R, T, and W.

```
1) $C \leftarrow \pi_w(P_n)$; (as described in Definition 2).
2) while ($C$ contains non-zero entries) do
     • find a permutation $Q$ included in $C$;
     • while $C - Q$ contains no negative entries do
         – $C \leftarrow C - Q$;
         – append permutation $Q$ to $L$;
         – find $(x_0, \dots, x_{w-1})$ s.t. $(x_i (\text{mod } w) = i)$ and
           $(P_n(x_i) (\text{mod } w) = Q(i))$;
         – append $(\lfloor x_i / w \rfloor)$, $0 \leq i < w$ to $R$;
         – append $(\lfloor P_n(x_{Q^{-1}(i)}) / w \rfloor)$, $0 \leq i < w$ to $W$;
3) calculate $T$ based on the permutations stored in $L$, using
   the techniques in [4], [5];
4) output $R, T$, and $W$;
```

The most computationally difficult part of Algorithm 6 is finding a permutation Q_w that may be subtracted from C_w. This can be accomplished in several ways: using a brute force algorithm, mapping the problem to a satisfiability problem, or using an algorithm based on systems of distinct representatives (e.g., [11, Ch. 5]). In our implementation, we choose the satisfiability approach; it is able to solve practical problem sizes very quickly. For the largest problem we consider in this paper ($n = 4096, w = 64$), Algorithm 6 completes in approximately 6 minutes.

VI. Implementation and Evaluation

In this section, we discuss our implementation of the proposed method, and evaluate the designs produced.

name	type	ports	# needed	# words	bits per word
M_0	RAM	2	w	$2n/w$	b
M_1	RAM	2	w	$2n/w$	b
R	ROM	1	1	n	$w \cdot \lceil \log_2(n/w) \rceil$
W	ROM	1	1	n	$w \cdot \lceil \log_2(n/w) \rceil$
T	ROM	1	1	n/w	$w' \log_2 w' - w' + 1$

Note: $w' = 2^{\lceil \log_2 w \rceil}$; b is the number of bits in each word of the datapath's inputs and outputs.

TABLE I
SUMMARY OF MEMORIES REQUIRED.

Synthesis Tool. Based upon the techniques discussed in this paper, we have built a tool that takes as input a permutation P_n and a streaming width w, and outputs a register-transfer level Verilog description of the design. The tool generates an instance of the parameterized datapath, and uses Algorithm 6 to determine the values to store in the lookup tables.

Analysis of Generated Designs. Table I summarizes the number and size of memories needed by the solution we propose. Our solution additionally requires $w' \log_2 w' - w' + 1$ two-input switches, where $w' = 2^{\lceil \log_2 w \rceil}$. In general, the cost of our implementation depends only on n (the number of points in the data vector) and w (the streaming width of the system).[2]

Designs produced using this method have a throughput of w words per cycle, and a latency of $2n/w + \lceil \log_2(w) \rceil + 3$ cycles.

Experimental Evaluation. Here, we evaluate the cost and performance of our generated designs when synthesized for a Xilinx Virtex-5 FPGA. We use Xilinx ISE 9.2 to synthesize and place/route designs, and we extract all timing and area measurements after place/route has completed. In this evaluation, we assume that the data words are $b = 16$ bits wide.

Virtex-5 FPGAs contain on-chip memory structures called block RAM (BRAM). When our design requires a RAM or ROM of size $\geq$ 1024 bits, we map it to a BRAM. Smaller memories are created out of the FPGA's reconfigurable logic elements. This threshold (1024 bits) is a parameter of our generation tool.

We evaluate the cost and performance of several configurations of our datapath: $n = 64$ with $w = 2, 4, 8, 16, 32$, and $n = 512, 4096$ with $w = 2, 4, 8, 16, 32, 64$. Because the cost of our implementation does not depend on the specific permutation being performed, we choose one randomly for each n. Figure 3 shows throughput (in gigabits per second) versus FPGA area (in slices, which are the reconfigurable blocks of the FPGA) for all designs. Each line shows a different value of n (the problem size), and the different points within a line correspond to different values of streaming width w: the left-most point corresponds to $w = 2$; w doubles with each successive point. (Recall, the throughput of each design is w words per cycle.) Additionally, we label several of the points with the number of block RAMs (BRAMs) used by that design.

[2]The cost of implementation can only be further reduced in rare cases where all or part of a lookup table or connection network is redundant.

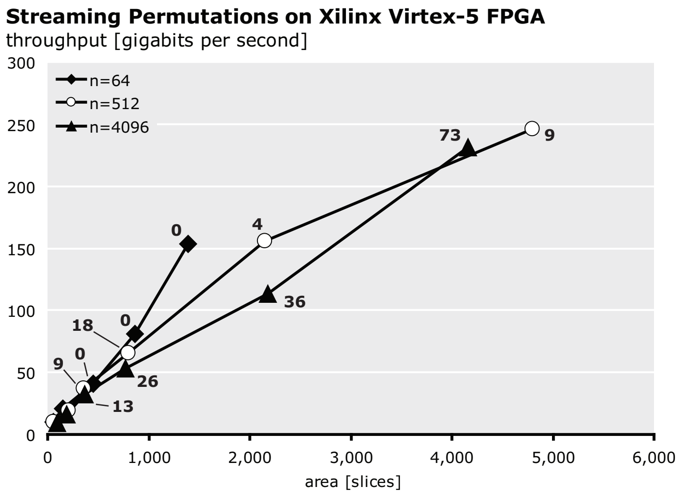

Fig. 3. Throughput versus slices for $n = 64, 512, 4096$. Labels: number of BRAMs.

From Figure 3, we see that as w increases, the resulting designs exhibit higher throughput, but become commensurately more expensive (in area). As n increases, we do not see a significant change in throughput or area, but we see a large increase in the number of BRAMs required. This occurs because the size of all memories grows with n (as shown in Table I).

In order to provide a reference point for comparison, we can compare our designs to [2], which describes a method for generating streaming permutation circuits for a subset of all permutations and streaming widths.[3] The designs produced by [2] utilize one memory array with interconnection networks at its inputs and outputs; both networks are optimized for the specific permutation considered. Furthermore, all memory and switch configurations are calculated online; no lookup tables are used. For many problems, [2] is able to produce designs with the optimum address logic and switching network (given the assumed architecture).

However, the technique in [2] is only applicable to a small subset of streaming permutations. Our goal is not to improve on [2]'s cost/performance tradeoff; we use it as a way to measure the added costs incurred by moving to our general structure.

The area required by the designs in [2] depends on the permutation being performed (as well as the permutation size and streaming width). So, we compare against designs for two permutations: the stride-by-two permutation, which is in the class of least expensive problems supported by [2], and the bit reversal permutation, which is in the class of most expensive problems. Again, we evaluate our designs using a random

[3][2] is only able to perform permutations that arise from invertible mappings on the bit representations of the indices, such as the bit reversal or stride permutations. Of the $n!$ possible permutations on n points, [2] is able to perform $(2^m - 1)(2^m - 2) \cdots (2^m - 2^{m-1})$, where $n = 2^m$ and only when n and w are powers of two.

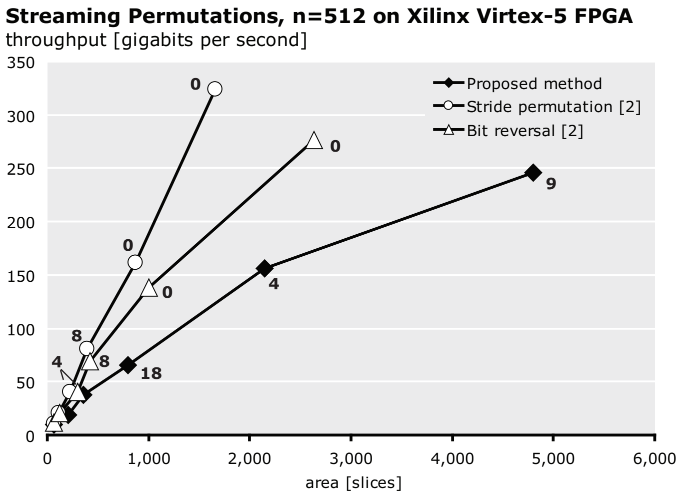

Fig. 4. Comparison of proposed general method with [2]. Labels: number of BRAMs.

permutation, since the cost does not depend on the specific permutation being performed.

We synthesize and place/route these benchmark designs using the previously stated assumptions, and present the results in Figure 4 for $n = 512$. Again, we plot throughput versus area and include the number of BRAMs for several designs. For small values of w, all three designs have similar costs. However, we see that as w increases, the amount of slices and BRAM required for our general method increases more quickly than those from [2]. If we repeat this experiment for smaller problem sizes (values of n), the difference between our method and the benchmark is reduced; for larger values of n, it is increased.

VII. Related Work

We do not know of any prior work on the general class of RAM-based streaming permutation structures considered in this paper. As discussed in Section VI, [2] provides a generation technique for a subset of streaming permutations. Other approaches (e.g., [1]) consider streaming implementations of a specific family (stride permutations). [12] builds streaming permutation structures using a register allocation method, resulting in a large number of individual registers connected with switches or multiplexers.

As discussed previously, the structure we consider in this paper and our mathematical approach are related to the input-buffered crossbar switch [6], [7], [8], which is able to perform network switching with minimum throughput guarantees. We have modified certain aspects of this switch (replacing online scheduling logic with pre-computed lookup tables), and have added components (M_1) that are necessary to adapt the design for streaming permutations.

Furthermore, the task of calculating the schedule for the input-buffered crossbar switch is related to the problem we solve with Algorithm 6. Some approaches use a technique similar to ours, where the mapping from input to output ports is represented as a matrix that is then decomposed [7], [8]. Others approach the problem in different ways, e.g., as a bipartite matching problem [6].

VIII. Conclusions

In this work, we described a parameterized architecture for performing an arbitrary fixed permutation on a streamed data vector. This design consists of double-ported RAMs, a switching network, and lookup ROMs that store pre-computed address and control data. We presented a technique that can map any streaming permutation onto this datapath, and a scheduling algorithm that guarantees the datapath can operate at full throughput without stalling. Using these techniques, we have developed a fully-automated tool that outputs a given design in register-transfer level Verilog. Lastly, we evaluated designs produced by our tool, and compared them with designs from an existing, less general technique.

Acknowledgments

This work was supported by DARPA under the DOI grant NBCH1050009 and the ARO grant W911NF0710416 and by NSF awards 0325687 and 0702386.

References

[1] T. Järvinen, P. Salmela, H. Sorokin, and J. Takala, "Stride permutation networks for array processors," in *Proc. IEEE Intl. Conf. on Application-Specific Systems, Architectures and Processors*, 2004.

[2] M. Püschel, P. A. Milder, and J. C. Hoe, "Permuting streaming data using RAMs," *Journal of the ACM*, in press.

[3] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Formal datapath representation and manipulation for implementing DSP transforms," in *Proc. Design Automation Conference*, 2008.

[4] K. Y. Lee, "On the rearrangeability of $2(\log_2 n) - 1$ stage permutation networks," *IEEE Transactions on Computers*, vol. 34, no. 5, pp. 412–425, May 1985.

[5] A. Waksman, "A permutation network." *Journal of the ACM*, vol. 15, no. 1, pp. 159–163, 1968.

[6] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand, "Achieving 100% throughput in an input-queued switch," *IEEE Transactions on Communications*, vol. 47, no. 8, August 1999.

[7] S. Li and N. Ansari, "Input-queued switching with QoS guarantees," in *Proc. INFOCOM (Joint Conference of the IEEE Computer and Communications Societies)*, 1999.

[8] C.-S. Chang, W.-J. Chen, and H.-Y. Huang, "Birkhoff-von Neumann input-buffered crossbar switches for guaranteed-rate services," *IEEE Transactions on Communications*, vol. 49, no. 7, January 2001.

[9] D. Kőnig, "Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre," *Mathematische Annalen*, vol. 77, pp. 453–465, 1915–1916.

[10] D. B. Leep and G. Myerson, "Marriage, magic, and solitaire," *American Mathematical Monthly*, vol. 106, no. 5, pp. 419–429, 1999.

[11] M. Hall, Jr., *Combinatorial Theory*. Wiley-Interscience, 1986.

[12] K. K. Parhi, "Systematic synthesis of DSP data format converters using life-time analysis and forward-backward register allocation," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 7, pp. 423–440, 1992.