Permuting Streaming Data Using RAMs

MARKUS PÜSCHEL, PETER A. MILDER, and JAMES C. HOE
Carnegie Mellon University

This paper presents a method for constructing hardware structures that perform a fixed permutation on streaming data. The method applies to permutations that can be represented as linear mappings on the bit-level representation of the data locations. This subclass includes many important permutations such as stride permutations (corner turn, perfect shuffle, etc.), the bit reversal, the Hadamard reordering, and the Gray code reordering.

The datapath for performing the streaming permutation consists of several independent banks of memory and two interconnection networks. These structures are built for a given streaming width (i.e., number of inputs and outputs per cycle) and operate at full throughput for this streaming width.

We provide an algorithm that completely specifies the datapath and control logic given the desired permutation and streaming width. Further, we provide lower bounds on the achievable cost of a solution and show that for an important subclass of permutations our solution is optimal.

We apply our algorithm to derive datapaths for several important permutations, including a detailed example that carefully illustrates each aspect of the design process. Lastly, we compare our permutation structures to those of [Järvinen et al. 2004], which are specialized for stride permutations.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design aids—*Automatic Synthesis*

General Terms: Algorithms, Design, Theory, Performance

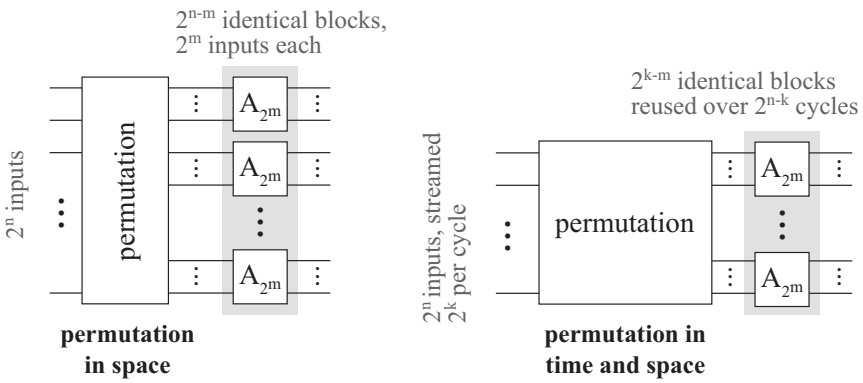
Additional Key Words and Phrases: Permutation, RAM, streaming datapath, stride permutation, matrix transposition, data reordering, linear bit mapping, connection network, switch

1. INTRODUCTION

Streaming architectures are very commonly used in hardware design, particularly when high throughput is desired. A streaming architecture takes as input a vector consisting of a fixed number of data words, divided into subvectors of equal length that enter the system at regular intervals. Similarly, the architecture produces an output vector of the same length and format. Further, a streaming architecture is able to start processing the first subvector of a new input vector immediately after the final portion of the last input vector enters the system. In other words, a streaming architecture takes input and produces output at a fixed rate, with no gap between data vectors.

Often, applications implemented with streaming architectures consist of computation stages separated by data permutations, i.e., re-orderings of data in a predetermined manner. Consider an application that processes a vector of length 2^n and consists of a permutation followed by parallel computation blocks. A non-streaming implementation is shown in Fig. 1(a). All 2^n elements of the input vector enter the system concurrently; hence the permutation is simply a re-ordering in space using wires.

In contrast, in Fig. 1(b), the input vector is broken into subvectors of size 2^k (called the streaming width), $k \leq n$, which enter the datapath in 2^{n-k} consecutive cycles. This yields a saving in the number of computation blocks needed, but also



(a) not streaming (b) streaming

Fig. 1. This figure demonstrates the motivation for this work. On the left is a datapath that receives all 2^n inputs at once, reorders them with a simple hardwired permutation in space, and performs computations A_{2^m} in parallel 2^{n-m} times. On the right, we see a streamed version, which streams in data 2^k per cycle over 2^{n-k} cycles. Now, the data must be permuted in space and time. Designing this space/time permutation is the problem that we address in this work.

produces a problem. Namely, the initial permutation now requires a reordering in space (via wires and switches) and in time (by storing and retrieving data from memory).

When the streaming width (2^k) is non-trivial (i.e., greater than 1 or 2), the difficulty of building a streaming permutation structure greatly increases. Due to the high cost of multi-port memory, the permutation must be performed over several independent banks of memory, and partitioned in such a way to guarantee that the data will be able to be streamed in and out without causing a bank conflict (the need to read or write more than one word in a given memory bank at the same time). In these situations, it is unclear how to build the memory, interconnection networks, and necessary control logic. Except for a few special cases (discussed below), this is an unsolved problem.

In this paper, we propose an efficient method for performing a large class of permutations on data with arbitrary streaming width 2^k ($1 \leq k \leq n$). The method applies to any permutation that is a linear transform on the bit representation of the data addresses. This class includes many important permutations such as stride permutations (also called matrix transpositions or corner turns) and the bit reversal. These permutations occur in a wide range of applications such as linear transform algorithms (e.g., fast Fourier transform, Walsh-Hadamard Transform, discrete sine and cosine transforms), multi-dimensional separable signal processing, sorting networks, and Viterbi coding. In Section 6.2, we provide specific examples (and references) for these applications.

We specify an architectural framework that utilizes dual-port memories (we require one read port and one write port). Memories of this type are abundant on modern field-programmable gate arrays (FPGAs), and are straightforward to implement in an application-specific integrated circuit (ASIC). We view the permutations as linear mappings on the bit representation of the data locations, and

express the hardware constraints (i.e., memory port constraints) as conditions on the structure of the mappings. We provide an algorithm that is able to factor each expressible permutation into terms that fulfill these conditions. In this way, we provide an algorithm to design the complete datapath for the specified permutation. We analyze the implementation costs, derive lower bounds, and show that our algorithm gives optimal solutions for an important subclass of permutations.

Further, for experimental validation, we have written a tool that takes as input a permutation and a streaming width, and uses the algorithm presented in this paper to design the hardware structure. The tool outputs this design in synthesizable Verilog.

Related Work. The stride permutation is a special case of the problem we address in this paper. In [Takala et al. 2003; Järvinen et al. 2004], streaming structures for two-power sizes are developed. However, their solutions are based on different assumptions and are hence different from ours. Additionally, [Gorman and Wills 1995] considers a limited set of streaming stride permutations in the context of hardware implementations of the fast Fourier transform.

In other recent work [Milder et al. 2009], we have developed a RAM-based method that can be used to automatically design a streaming datapath for *any* given permutation. This method is hence more general than the one we present here, but it is much more expensive, because it cannot take advantage of the linearity assumption used in this paper.

In [Parhi 1992], a technique is presented to design data format converters, which are register-based structures that are synthesized to perform a given streaming permutation. This technique is applicable to all permutations and all streaming widths. Designs produced in this way consist of independent registers connected with wires and multiplexers. A benefit of this approach is that it is optimal in the amount of storage used for a given permutation. However, this storage is distributed across many small independent registers, often leading to very complicated control and routing.

Also of note is [Láng 1976], which gives a scheme to permute n data points at a stride of $n/2$ (also known as the perfect shuffle) on an array computer, which consists of several parallel memories connected by a network of switches. Performing a permutation on this type of system has several similarities to the streaming problem we consider.

Lastly, we note that a large amount of work has been done in the area of permutation networks [Benes 1965; Waksman 1968; Lawrie 1975; Pease 1977]. Although there are connections between this work and ours (discussed later), there is a very important distinction: these networks take in and permute 2^n data points concurrently, while our *streaming* problem receives the 2^n data elements at a rate of 2^k per cycle, separated over multiple consecutive cycles. Thus, our structures require memory to reorder across time boundaries, while permutation networks do not.

In Section 6.2, we present a more thorough discussion of this related work and evaluate the result of applying our general method to the stride permutation by comparing with [Järvinen et al. 2004].

Organization. We begin with a description of our notation and present the necessary linear algebra background in Section 2. In Section 3, we state the exact

problem and formulate it mathematically including a suitable cost model. We derive the lower bounds on the implementation costs in Section 4. Section 5 presents the algorithm for constructing our solutions, identifies the cases in which they are optimal, and shows a detailed example. In Section 6, we evaluate our streaming permutations: (a) we provide several examples of important permutations and their streaming solutions and identify those that are optimal, (b) we compare to related work, and (c) we discuss applications of streaming permutations. Lastly, we offer our concluding remarks in Section 7.

2. BACKGROUND AND NOTATION

Permutations and linear bit mappings. In this paper we consider permutations on 2^n points $0, \ldots, 2^n - 1$. For example, the cyclic shift (by 1) is defined by

$$\boldsymbol{C}_{2^n} : \ 0 \mapsto 1 \mapsto 2 \mapsto \ldots \mapsto 2^n - 1 \mapsto 0, \quad \text{or} \quad i \mapsto i + 1 \bmod 2^n.$$

We equally view permutations as matrices and use the same notation for them. There are two choices for associating a matrix with a permutation and they are transposes of each other; we choose the one that makes

$$\boldsymbol{C}_{2^n} = \begin{pmatrix} 0 & & & 1 \\ 1 & & & \\ & \ddots & & \\ & & 1 & 0 \end{pmatrix}. \tag{1}$$

The set of all permutations on 2^n points is denoted with $\mathcal{S}_{2^n}$ and it is a group, called the symmetric group. It has $2^n!$ elements.

We denote the field (also called Galois field) with 2 elements with $\mathbb{F}_2 = \{0, 1\}$ and consider its elements as the two states of a bit. Addition and multiplication in $\mathbb{F}_2$ are equivalent to the "xor" and "and" operations, respectively.[1]

Permutations on $0, \ldots, 2^n - 1$ can equivalently be seen as permutations on the corresponding bit representations $x \in \mathbb{F}_2^n$ of these numbers. We view these x as column vectors and assume the least significant bit is on the bottom. For example, for $n = 2$, the number 1 is represented as $\left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right)$.

In some cases a permutation of $\mathbb{F}_2^n$ is a linear mapping on $\mathbb{F}_2^n$ of the form

$$y = Px, \quad P \in \mathrm{GL}_n(\mathbb{F}_2),$$

where $\mathrm{GL}_n(\mathbb{F}_2)$ is the group of all invertible $n \times n$ bit matrices. Its size is $(2^n - 1)(2^n - 2) \cdots (2^n - 2^{n-1})$. Every bit matrix $P \in \mathrm{GL}_n(\mathbb{F}_2)$ defines a permutation in $\mathcal{S}_{2^n}$. We formally capture this by the mapping

$$\pi : \ \mathrm{GL}_n(\mathbb{F}_2) \to \mathcal{S}_{2^n}, \quad P \mapsto \pi(P). \tag{2}$$

We will identify $\mathrm{GL}_n(\mathbb{F}_2)$ with its image $\pi(\mathrm{GL}_n(\mathbb{F}_2))$ in $\mathcal{S}_{2^n}$.

Consider a small example:

$$P = \begin{pmatrix} 1 & 1 \\ & 1 \end{pmatrix}$$

[1] In this paper we only require the ring structure of $\mathbb{F}_2$. Hence, our method can be generalized by replacing $\mathbb{F}_2$ with any ring of integers modulo a fixed n. This way, for example, a streaming datapath for a radix-3 bit reversal can be derived.

maps $\binom{0}{0} \mapsto \binom{0}{0}, \binom{0}{1} \mapsto \binom{1}{1}, \binom{1}{0} \mapsto \binom{1}{0}, \binom{1}{1} \mapsto \binom{0}{1}$, i.e., $0 \mapsto 0, 1 \mapsto 3 \mapsto 1, 2 \mapsto 2$, which implies

$$\pi(P) = \begin{pmatrix} 1 & & & \\ & & & 1 \\ & & 1 & \\ & 1 & & \end{pmatrix}.$$

To clearly distinguish matrices P that operate on n bits from permutation matrices that operate on 2^n points, we will bold-face the latter as in (1).

We introduce additional notation for matrices. We denote the *direct sum* and the Kronecker or *tensor product* of matrices P, Q respectively with

$$P \oplus Q = \begin{pmatrix} P & \\ & Q \end{pmatrix}, \quad \text{and} \quad P \otimes Q = (p_{i,j}Q)_{i,j} \text{ for } P = (p_{i,j}).$$

Further, I_n is the $n \times n$ identity matrix and J_n is I_n with the columns in reversed order. The stride permutation is defined via

$$\boldsymbol{L}_{2^n,2^s} : \ i2^{n-s} + j \mapsto j2^s + i, \quad 0 \le i < 2^s, 0 \le j < 2^{n-s}.$$

Equivalently, $\boldsymbol{L}_{2^n,2^s}$ transposes a $2^{n-s} \times 2^s$ array stored in row-major order. $\boldsymbol{L}_{2^n,2^{n-1}}$ is also called the perfect shuffle.

The bit-reversal permutation is denoted with $\boldsymbol{R}_{2^n}$.

Now we can state the following well-known properties of π defined in (2). In (4) below, C_n^{n-k} is the $(n-k)$th power of the cyclic shift C_n, i.e., a cyclic shift by $n-k$ on n points.

LEMMA 2.1. *Let $P, Q \in \mathrm{GL}_n(\mathbb{F}_2)$. Then the following holds.*

(1) $\pi(PQ) = \pi(P)\pi(Q)$ *and* $\pi(P^{-1}) = \pi(P)^{-1}$ *(i.e., π is a group homomorphism).*
(2) $\pi(P \oplus Q) = \pi(P) \otimes \pi(Q)$.
(3) $\pi(I_n) = \boldsymbol{I}_{2^n}$
(4) $\pi(C_n^{n-k}) = \boldsymbol{L}_{2^n,2^k}$
(5) $\pi(J_n) = \boldsymbol{R}_{2^n}$

We will encounter other relevant pairs of P and $\pi(P)$ in Section 6.

Linear algebra background. We use a number of basic results from linear algebra (e.g., [Bernstein 2005]). If $P \in \mathbb{F}_2^{m \times n}$ is an $m \times n$ bit matrix, then

$$\mathrm{im}(P) = \{Px \mid x \in \mathbb{F}_2^n\} \quad \text{and} \quad \ker(P) = \{x \in \mathbb{F}_2^n \mid Px = 0\}$$

are the *image* and *kernel* (or nullspace) of the linear mapping defined by the matrix P. It holds that $\dim(\mathrm{im}(P)) = \mathrm{rank}(P)$ and $\dim(\ker(P)) + \dim(\mathrm{im}(P)) = n$. Further, $\mathrm{rank}(P + Q) \le \mathrm{rank}(P) + \mathrm{rank}(Q)$ and $\mathrm{rank}(PQ) \le \min(\mathrm{rank}(P), \mathrm{rank}(Q))$.

If $V \le \mathbb{F}_2^n$ is a vector subspace of dimension $\dim(V) = k$, then $|V| = 2^k$. If $x \in \mathbb{F}_2^n$, then any $x + V = \{x + v \mid v \in V\}$ is called a coset of V in $\mathbb{F}_2^n$. Its size is again 2^k and there are precisely 2^{n-k} many different cosets that partition $\mathbb{F}_2^n$.

3. PROBLEM FORMULATION

Given is an invertible linear bit mapping, or bit matrix, $P \in \mathrm{GL}_n(\mathbb{F}_2)$; $\pi(P)$ is the corresponding permutation on 2^n points. We want to design a logic block that

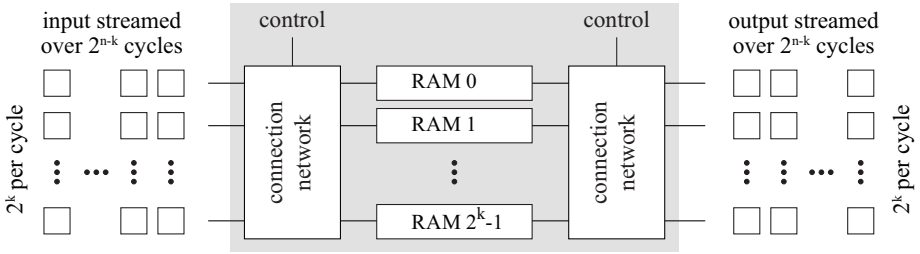


Fig. 2. The problem considered in this paper: permuting an input vector of length 2^n, streamed with width 2^k. The permutation (enclosed by the gray box) is performed in two stages using 2^k available RAMs with write/read port restrictions and suitably designed connection networks.

permutes a vector of 2^n data words with $\pi(P)$. The vector is streamed with a streaming width 2^k, $k \leq n$. This means that in every cycle the logic block takes a segment of length 2^k of the vector as input, and produces a segment of equal length of the permuted vector as output (see Fig. 1(b)). Further, we assume there are 2^k banks of RAM, each of which can hold at least 2^{n-k} data words of the vector. (If each available RAM can only hold 2^c words, $c < n - k$, then 2^{n-k-c} RAMs can be used to simulate a RAM of size 2^{n-k}.) We assume that each of the RAMs is dual-ported. That is, each memory can concurrently read and write one data word in each cycle.

The conditions described above capture the situation on current FPGA platforms, but can also be realized as an ASIC.

The basic idea is to design two connection networks as shown in Fig. 2 and to permute the data in two stages by writing into the RAMs and reading out from the RAMs. We call the first stage (input data to RAM) the *write-stage* and the second stage (RAM to output data) the *read-stage*. Each stage has to obey the port restrictions of the RAMs, i.e., no further buffers are used.

Besides finding a solution, the optimization criterion is to minimize the cost of the connection network.

To mathematically formulate the situation and problem above, we go through several steps:

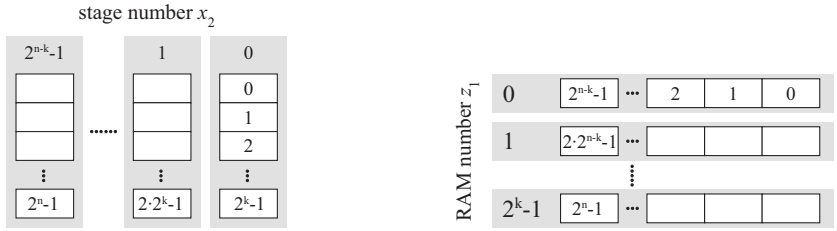
—*Address scheme and stages*: We assign proper addresses to the input vector and the RAMs. The problem is then to find a suitable factorization of the bit matrix P into a matrix product $N^{-1}M$, one matrix for each stage.[2]

—*Constraint: dual-ported*: We formulate this constraint mathematically as a condition on certain submatrices of M and N.

—*Optimization criteria*: We capture the cost of the connection network through two metrics: the connectivity of the network and the cost of the control logic. Both metrics are properties of M and N.

Address scheme and stages. The streamed input vector of length 2^n is indexed with addresses $x \in \mathbb{F}_2^n$ as shown in Fig. 3(a). The stage number is given by

[2]We factor $P = N^{-1}M$ and not $P = NM$ for notational convenience later.



(a) streamed vector (b) RAMs

Fig. 3. Addressing used in this paper. The streamed vector contains 2^n data words streamed in 2^{n-k} stages; 2^k RAMs with space for (at least) 2^{n-k} data words each are available.

the upper $n-k$ bits of x and the location within the stage by the lower k bits, referred to as x_2 and x_1, respectively: $x = \begin{pmatrix} x_2 \\ x_1 \end{pmatrix}$.

The 2^k RAMs are indexed with addresses $z \in \mathbb{F}_2^n$ as shown in Fig. 3(b). The RAM number z_1 is given by the k lower bits of z, and the location z_2 in the RAM by the $n-k$ upper bits: $z = \begin{pmatrix} z_2 \\ z_1 \end{pmatrix}$.

The addressing of the output vector y is analogous to the addressing of the input vector.

We want to perform P, i.e., $y = Px$, in two stages, the write-stage $z = Mx$ and the read-stage $y = N^{-1}z$, which is equivalent to a factorization

$$P = N^{-1}M, \tag{3}$$

where $M, N \in \mathrm{GL}_n(\mathbb{F}_2)$ are again (necessarily) invertible bit matrices. In words, M determines how the streaming data is stored into the RAMs (the write-stage), and N^{-1} determines how it is read out of the RAMs into the resulting data stream (the read-stage). The addressing is chosen such that $M = I_{2^n}$ or $N^{-1} = I_{2^n}$ makes the write or read stage trivial, respectively. This means that the connection network and all address computations vanish.

By partitioning the addresses as explained above, we obtain the following expanded version of the write-stage $z = Mx$:

$$\begin{matrix} \textit{where in RAM} \\ \textit{RAM number} \end{matrix} \quad \begin{pmatrix} z_2 \\ z_1 \end{pmatrix} = \begin{pmatrix} M_4 & M_3 \\ M_2 & M_1 \end{pmatrix} \begin{pmatrix} x_2 \\ x_1 \end{pmatrix} \quad \begin{matrix} \textit{stage number} \\ \textit{where in stage} \end{matrix} \tag{4}$$

The matrix tiling is compatible with the partitioning of the vectors. For example, M_1 is a $k \times k$ matrix.

Analogous to the write-stage, we also expand the read-stage $y = N^{-1}z$. However, there is one crucial difference. Namely, the control logic in Fig. 2 determines from which RAM and where in this RAM to read, given the output stage number and the output stage location. Thus, we have to consider the inverse equation $z = Ny$ and partition N as M above:

$$\begin{matrix} \textit{where in RAM} \\ \textit{RAM number} \end{matrix} \quad \begin{pmatrix} z_2 \\ z_1 \end{pmatrix} = \begin{pmatrix} N_4 & N_3 \\ N_2 & N_1 \end{pmatrix} \begin{pmatrix} y_2 \\ y_1 \end{pmatrix} \quad \begin{matrix} \textit{stage number} \\ \textit{where in stage} \end{matrix} \tag{5}$$

Constraint: dual-ported. Since the RAMs are dual-ported, they allow only

one write (and read) per cycle. Thus, we require that for every stage, each of the 2^k data words is mapped into a different RAM. Mathematically, this means that for any fixed x_2 the mapping

$$z_1 = M_2 x_2 + M_1 x_1$$

is bijective. This is the case if and only if M_1 is invertible, i.e., has full rank: $\mathrm{rank}(M_1) = k$. A similar discussion yields the requirement $\mathrm{rank}(N_1) = k$.

Connectivity. The major cost factors in Fig. 2 are the two connection networks. They are determined by the connections required for writing input data into the RAMs, and for retrieving the output data from the RAMs, collected over all stages. In the best case, the data words are always written to or read from the same RAM, which reduces the connection network to a simple wired connection without control logic. In the worst case, an all-to-all connection has to be supported.

Given M and N, we can measure the required connectivity as follows.

LEMMA 3.1. *Let M be as in (4) (N as in (5)) and assume that M_1 (N_1) is invertible and set* $\mathrm{rank}(M_2) = s$ ($\mathrm{rank}(N_2) = s$). *Then, the 2^k-to-2^k connection network in the write-stage (read-stage) of Fig. 2 decomposes into 2^{k-s} independent 2^s-to-2^s networks, called blocks.*

The input and output index sets of these blocks (subsets of $0, \ldots, 2^k - 1$) are precisely the cosets

$$x_1 + \mathrm{im}(M_1^{-1} M_2) \quad \textit{and} \quad M_1 x_1 + \mathrm{im}(M_2)$$

for the write-stage and are the cosets

$$y_1 + \mathrm{im}({N_1}^{-1} N_2) \quad \textit{and} \quad N_1 x_1 + \mathrm{im}(N_2)$$

for the read-stage.

Further, each block has to support an all-to-all connection and precisely 2^s different configurations.

PROOF. We prove the lemma for the write-stage; the read-stage is handled analogously.

We set $s = \mathrm{rank}(M_2)$. Assume x_1 is the address of a stage location (i.e., a number that indicates one of the 2^k input ports). We accumulate, over all stages, the RAM numbers z_1 that x_1 connects to. This is the set

$$U_{x_1} = \{M_2 x_2 + M_1 x_1 \mid x_2 \in \mathbb{F}_2^{n-k}\} = \mathrm{im}(M_2) + M_1 x_1. \tag{6}$$

The size of this set is 2^s. Now assume x_1' is another address within a stage, and satisfies $U_{x_1} \cap U_{x_1'} \neq \emptyset$. Then

$$M_1(x_1 - x_1') \in \mathrm{im}(M_2) \Leftrightarrow x_1' \in x_1 + M_1^{-1}\,\mathrm{im}(M_2) = x_1 + \mathrm{im}(M_1^{-1} M_2).$$

The size of this set is also 2^s. Conversely, assume x_1' has the above form, i.e., $x_1' = x_1 + M_1^{-1} M_2 x_2$ for some $x_2 \in \mathbb{F}_2^{n-k}$. Then, using (6),

$$U_{x_1'} = \mathrm{im}(M_2) + M_1 x_1 + M_2 x_2 = \mathrm{im}(M_2) + M_1 x_1 = U_{x_1}.$$

In other words, if x_1 and x_1' share one connection target z_1, they share all 2^s targets, which proves the block decomposition. The input and output index sets are also computed as desired.

The above also shows that each block has to support an all-to-all connection. The remaining question is the number of control configurations. Assume two stages x_2, x_2' that connect all x_1 equally, i.e., for all $x_1 \in \mathbb{F}_2^k$,

$$M_2x_2 + M_1x_1 = M_2x_2' + M_1x_1 \Leftrightarrow x_2' \in x_2 + \ker(M_2).$$

The size of this set is 2^{n-k-s} and we get that the 2^{n-k} stages partition into 2^s many groups of size 2^{n-k-s} each, such that within the groups, all connections are equal. These groups are the cosets $x_2 + \ker(M_2)$. Between different groups of stages the connections differ since each x_1 has 2^s many targets ($|U_{x_1}| = 2^s$). This completes the proof. □

Lemma 3.1 implies that it is desirable to minimize $\operatorname{rank}(M_2)$ and $\operatorname{rank}(N_2)$ in (4) and (5) to minimize the area cost of the implementation in Fig. 2. It motivates the following definition.

Definition 3.2 (Connectivity). *Given a matrix M in (4) or N in (5), we call* $\operatorname{conn}(M,k) = 2^{\operatorname{rank}(M_2)}$ *and* $\operatorname{conn}(N,k) = 2^{\operatorname{rank}(N_2)}$ *the* connectivity *of M and N, respectively, with respect to the streaming width 2^k.*

In the following we will mostly omit the second argument k since it is clear from the context. In words, high connectivity implies high switching cost.

Cost of control. The control blocks in Fig. 2 have to compute z given x and z given y, respectively. This motivates the following cost measure, based upon linear complexity [Bürgisser et al. 1997].

Definition 3.3 (Control cost). *Let M be as in (4) (N as in (5)). Then we call* $\operatorname{cost}(M)$ *(*$\operatorname{cost}(N)$*) the* control cost *of M (N), defined as the linear complexity of M (N), i.e., the minimum number of (binary) additions to compute $z = Mx$ ($z = Nx$).*

Formal problem statement. With the above notation and discussion we can now formally state the problem as follows.

Problem 3.4. *Given are an invertible bit matrix $P \in \mathrm{GL}_n(\mathbb{F}_2)$, partitioned as in (4), and a streaming width 2^k, $k \leq n$.*

(1) Determine a factorization $P = N^{-1}M$, such that $\operatorname{rank}(M_1) = \operatorname{rank}(N_1) = k$*. The goal is to minimize* $\operatorname{rank}(M_2), \operatorname{rank}(N_2)$ *and the complexity of M and N. Necessarily, $M, N \in \mathrm{GL}_n(\mathbb{F}_2)$.*

(2) Design the two connection networks in Fig. 2 from two-port switches. In Section 4.2 we will see that is equivalent to determining decompositions

$$M_2 = T_1 + T_2 + \cdots + T_{\operatorname{rank} M_2} \quad \text{and} \quad N_2 = T_1' + T_2' + \cdots + T'_{\operatorname{rank} N_2},$$

where all T_i and T_i' are matrices of rank 1.

Our goal requires the minimization of four different values, listed above. In Section 4, we will derive lower bounds for these. Later, we will show that our solutions always match the lower bounds for two out of the four values (namely $\operatorname{rank}(N_2)$ and $\operatorname{cost}(N)$), and, for an important subset of permutations, we achieve complete optimality for all four values.

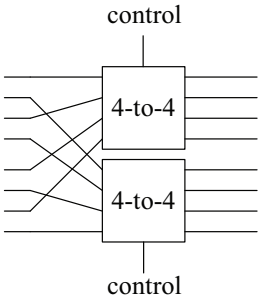


Fig. 4. M in (7) realized as an 8-to-8 connection network decomposed into two 4-to-4 blocks.

Example. Let M be the matrix representing the write stage of a permutation, streamed with 2^3 ports, blocked as in (4):

$$M = \left(\begin{array}{c|c} M_4 & M_3 \\ \hline M_2 & M_1 \end{array}\right) = \left(\begin{array}{ccc|ccc} . & . & 1 & . & . & . \\ . & . & . & 1 & . & . \\ . & . & . & . & 1 & . \\ \hline . & . & . & . & . & 1 \\ 1 & . & . & 1 & . & . \\ . & 1 & . & . & 1 & . \end{array}\right). \tag{7}$$

Since $\mathrm{rank}(M_2) = 2$, Lemma 3.1 asserts that the 8-to-8 network decomposes into two 4-to-4 blocks, i.e.,

$$\mathrm{conn}(M) = 2^{\mathrm{rank}(M_2)} = 2^2 = 4.$$

The input sets for the two blocks are the cosets

$$\begin{aligned} x_1 + \mathrm{im}(M_1^{-1}M_2) &= x_1 + \mathrm{im}\left(\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}\right) \\ &= x_1 + \mathrm{im}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\ &= x_1 + \left\{\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}\right\}, \end{aligned}$$

and the output sets are the cosets

$$M_1 x_1 + \mathrm{im}(M_2) = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} x_1 + \left\{\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}\right\}.$$

Here, the first block has inputs $\{0, 2, 4, 6\}$ and the second block has inputs $\{1, 3, 5, 7\}$. The outputs of the first block are $\{0, 1, 2, 3\}$ and the outputs of the second block are $\{4, 5, 6, 7\}$. This is illustrated in Fig. 4.

Definition 3.3 shows that the control cost, $\mathrm{cost}(M)$, is given by the linear complexity of M. Obviously,

$$\mathrm{cost}(M) \leq 2.$$

Later, we will give a technique to automatically design the 2^s-to-2^s connection networks.

4. LOWER BOUNDS AND THE DESIGN OF CONNECTION NETWORKS

4.1 Lower Bounds

Before we provide an explicit method to compute suitable factorizations $P = N^{-1}M$, we determine lower bounds on the quality, i.e., the connectivity and the control cost, of a possible solution. This will allow us later to identify those cases for which our solutions are optimal.

Similar to M and N in (4) and (5), we also tile P and P^{-1} as

$$P = \begin{pmatrix} P_4 & P_3 \\ P_2 & P_1 \end{pmatrix} \quad \textit{and} \quad P^{-1} = \begin{pmatrix} P'_4 & P'_3 \\ P'_2 & P'_1 \end{pmatrix} \tag{8}$$

for the following discussion.

Connectivity. We derive lower bounds on the connectivity of our interconnection networks. Later, we will show that we reach this lower bound for an important subset of permutations.

THEOREM 4.1. *Assume $P = N^{-1}M$ is a solution of Problem 3.4, then*

$$\operatorname{rank}(M_2) \geq k - \operatorname{rank}(P'_1) \quad \textit{and} \quad \operatorname{rank}(N_2) \geq k - \operatorname{rank}(P_1),$$

which is equivalent to

$$\operatorname{conn}(M) \geq 2^{k-\operatorname{rank}(P'_1)} \quad \textit{and} \quad \operatorname{conn}(N) \geq 2^{k-\operatorname{rank}(P_1)}.$$

PROOF. Assume a solution $P = N^{-1}M$, which implies $NP = M$, i.e., the equation

$$\begin{pmatrix} N_4 & N_3 \\ N_2 & N_1 \end{pmatrix} \begin{pmatrix} P_4 & P_3 \\ P_2 & P_1 \end{pmatrix} = \begin{pmatrix} M_4 & M_3 \\ M_2 & M_1 \end{pmatrix}, \tag{9}$$

and M_1 and N_1 have full rank k. Further, we set $r = \operatorname{rank}(P_1)$, which implies $\operatorname{rank}(P_3) \geq k - r$, since P has full rank. (9) implies $M_1 = N_2P_3 + N_1P_1$. We get

$$\begin{aligned} k &= \operatorname{rank}(N_2P_3 + N_1P_1) \\ &\leq \operatorname{rank}(N_2P_3) + \operatorname{rank}(N_1P_1) \\ &\leq \min(\operatorname{rank}(N_2), \operatorname{rank}(P_3)) + r. \end{aligned}$$

As a consequence $\operatorname{rank}(N_2) \geq k - r$ or

$$\operatorname{conn}(N) \geq 2^{k-r} = 2^{k-\operatorname{rank}(P_1)}$$

as desired.

The bound on $\operatorname{conn}(M)$ is obtained analogously, starting from $MP^{-1} = N$. □

As shown in Lemma 3.1, our solution requires 2^{k-s} separate 2^s-to-2^s switching blocks, which must each be "all-to-all". That is, each of the 2^s input ports must be capable of being connected to each of the 2^s output ports. We design these networks by decomposing them into arrays of 2-to-2 switches, each of which takes two elements and either exchanges them or allows them to pass, based upon a control bit. Later, we will provide a constructive method for assembling these

networks. Furthermore, we show that for an important subset of permutations, our technique decomposes the problem into switching networks in an optimal way (i.e., the resulting networks have optimal connectivity, as defined in Definition 3.2).

Control cost.

THEOREM 4.2. *Assume $P = N^{-1}M$ is a solution, then*

$$\text{cost}(M) \geq k - \text{rank}(P_1') \quad \textit{and} \quad \text{cost}(N) \geq k - \text{rank}(P_1).$$

PROOF. In the proof of Theorem 4.1, we asserted that $\text{rank}(N_2) \geq k - \text{rank}(P_1)$. This implies that N_2 contains at least $k - \text{rank}(P_1)$ non-zero elements. Since $\text{rank}(N_1) = k$, the linear complexity of the matrix $(N_2\ N_1)$, and thus that of N, is also at least $k - \text{rank}(P_1)$.

The bound on $\text{cost}(M)$ is obtained analogously. □

Theorems 4.1 and 4.2 show that the lower bounds for both the connectivity and the control cost are determined by $\text{rank}(P_1)$ and $\text{rank}(P_1')$, respectively. In the worst case, both ranks are zero.

Example. Consider the stride permutation $\boldsymbol{L}_{2^n,4} = \pi(C_n^2)$ streamed with width 2^k, $2 \leq k \leq n-2$. The corresponding tiling of $P = C_n^2$ as in (8) has, independent of k, the form

$$C_n^2 = \left(\begin{array}{ccc|ccc} 0 & 0 & 1 & & & \\ & \iddots & & & & \\ & & 1 & & & \\ & & 0 & 1 & & \\ & & 0 & 0 & 1 & \\ \hline & & & 0 & 0 & 1 \\ & & & & \iddots & \\ & & & & & 1 \\ 1 & & & & & 0 \\ & 1 & & & & 0 \end{array}\right). \tag{10}$$

In this case $\text{rank}(P_1) = \text{rank}(P_1') = k - 2$ (using $P^{-1} = P^T$). This implies that a solution $P = N^{-1}M$ that meets the lower bounds satisfies $\text{conn}(M) = \text{conn}(N) = 4$, i.e., each connection network decomposes into 2^{k-2} individual all-to-all networks, each with $2^2 = 4$ inputs and outputs. Further, $\text{cost}(M) = \text{cost}(N) = 2$.

We will later see that such an optimal solution does indeed exist.

4.2 Connection Networks

In Lemma 3.1, we established that the two connection networks in the write and read stages decompose into blocks with 2^s inputs and outputs, where $s = \text{rank}(M_2)$ or $s = \text{rank}(N_2)$, respectively. In the following we explain how to build these networks from two-port switches.

We consider the write stage given by M partitioned as in (4) and with invertible M_1. The connection network of the write stage, now considered without subsequent writing into the RAMs, performs a permutation for each stage according to $z_1 = M_2 x_2 + M_1 x_1$. If we adopt the addressing scheme in Fig. 3(a) for the input and output data stream of the network, then the network is again represented by a bit

matrix, namely by

$$T = \begin{pmatrix} I_{n-k} & \\ M_2 & M_1 \end{pmatrix}. \tag{11}$$

Conversely, every matrix of this form with invertible M_1 defines a connection network that decomposes into 2^s-to-2^s blocks, where $s = \operatorname{rank}(M_2)$.

To efficiently implement the network we decompose further. First,

$$T = \begin{pmatrix} I_{n-k} & \\ M_2 & I_k \end{pmatrix} \begin{pmatrix} I_{n-k} & \\ & M_1 \end{pmatrix}.$$

Next, we use that

$$\begin{pmatrix} I_{n-k} & \\ A & I_k \end{pmatrix} \begin{pmatrix} I_{n-k} & \\ B & I_k \end{pmatrix} = \begin{pmatrix} I_{n-k} & \\ A+B & I_k \end{pmatrix}. \tag{12}$$

Namely, we write M_2 as a sum of $s = \operatorname{rank}(M_2)$ many matrices T_i of rank one,

$$M_2 = T_1 + \cdots + T_s. \tag{13}$$

This is possible constructively, for example, by performing Gaussian elimination, $M_2 = QM_2'$, where Q is an invertible lower triangular matrix and M_2' is in row echelon form with s nonzero rows. Each row yields a summand T_i':

$$M_2' = T_1' + \cdots + T_s',$$

and setting $T_i = QT_i'$ yields the result.

Using (12) we can now factorize T in (11) and hence decompose the connection network. Analyzing the factors with Lemma 3.1 yields the following theorem.

THEOREM 4.3. *The connection network of the write stage in Fig. 2 is defined by the matrix T in* (11)*. With the previous notation, we have the factorization*

$$T = \begin{pmatrix} I_{n-k} & \\ T_1 & I_k \end{pmatrix} \cdots \begin{pmatrix} I_{n-k} & \\ T_{s-1} & I_k \end{pmatrix} \begin{pmatrix} I_{n-k} & \\ T_s & M_1 \end{pmatrix}.$$

This implies that the connection network can be decomposed into a cascade of s stages. Each stage is a connection network that consists of parallel 2-to-2 connection networks, or two-port switches, simultaneously controlled by one control bit. The input and output sets of these blocks are again cosets as determined by Lemma 3.1. The resulting network has s columns, each consisting of 2^{s-1} switches, giving a total of $s \cdot 2^{s-1}$ switches per network.

A similar statement holds for the connection network of the read stage in Fig. 2.

In (13), the summands can be permuted into any order, which implies that the stages in Theorem 4.3 can also be permuted.

Each stage in Theorem 4.3 is controlled by one bit. This implies that the entire network has 2^s many possible configurations. This number coincides with the number stated in Lemma 3.1 as desired. The control bit for each stage is calculated by $T_i x_2$ (or $T_i y_2$ when performing N).

The input and output sets of the first (write) stage are determined by

$$x_1 + \operatorname{im}(M_1^{-1} T_s) \quad \text{and} \quad M_1 x_1 + \operatorname{im}(T_s).$$

For all other stages T_i, where $1 \leq i < s$, both the input and output sets are given by

$$x_1 + \mathrm{im}(T_i).$$

For the read stage, the same expressions hold, with N_1 substituted for M_1.

Example. We now continue the example seen in Section 3. As before, let M be as given in (7). By Theorem 4.3, we decompose the rank 2 matrix M_2 into a sum of rank 1 matrices,

$$M_2 = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = T_1 + T_2.$$

This decomposition results in the following factorization of T:

$$T = \begin{pmatrix} I & 0 \\ T_1 & I \end{pmatrix} \begin{pmatrix} I & 0 \\ T_2 & M_1 \end{pmatrix} = \left(\begin{array}{ccc|ccc} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ \hline . & . & . & 1 & . & . \\ 1 & . & . & . & 1 & . \\ . & . & . & . & . & 1 \end{array}\right) \left(\begin{array}{ccc|ccc} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ \hline . & . & . & . & . & 1 \\ . & . & . & 1 & . & . \\ . & 1 & . & . & 1 & . \end{array}\right), \tag{14}$$

which corresponds to breaking the 4-to-4 connection network into two independent 2-to-2 networks.

Recall that T operates on a vector x of length 6. We write x as $(x_5,\ x_4,\ x_3,\ x_2,\ x_1,\ x_0)^T$, where x_5 indicates the most significant bit, and x_0 indicates the least significant bit. Reading T from right to left allows us to determine the resulting connection network, shown in Fig. 5.

First, we use the rightmost term to determine the initial permutation and first switching stage.[3] This stage has input sets

$$\begin{aligned} \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \mathrm{im}(M_1^{-1} T_2) &= \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \mathrm{im} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \right\}, \end{aligned}$$

and output sets

$$M_1 \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \mathrm{im}(T_2) = \begin{pmatrix} x_0 \\ x_2 \\ x_1 \end{pmatrix} + \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right\}.$$

So, the input sets are $\{0, 2\}$, $\{1, 3\}$, $\{4, 6\}$, and $\{5, 7\}$, and the output sets are $\{0, 1\}$, $\{2, 3\}$, $\{4, 5\}$, and $\{6, 7\}$. This, along with the initial blocking examined in the Section 3, gives the initial permutation and switching structure seen here.

Next, the second stage's input and output sets are determined by

$$\begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \mathrm{im}(T_1) = \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \right\}$$

[3]Theorem 4.3 gives the input and output sets in terms x_1, which is the generic description of the bottom k bits of the address. In this example, we explicitly consider a vector of $k = 3$ bits.

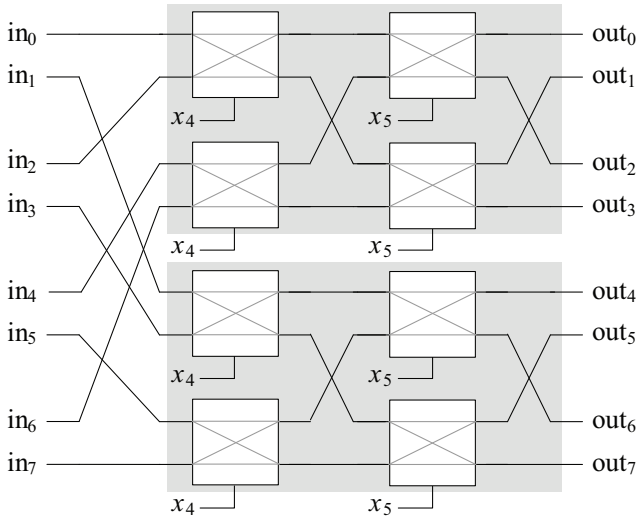


Fig. 5. The 8-to-8 connection network corresponding to (14). Each 4-to-4 block is decomposed into two cascaded stages. The gray boxes highlight the separation between the two independent 4-to-4 blocks. Each column of switches is controlled by a single control value, as shown.

So, both the input and output sets are $\{0,2\},\{1,3\},\{4,6\}$, and $\{5,7\}$. This gives the criss-crossing pattern seen before and after the switching column in Fig. 5.

As expected, this network has a connectivity of 4 and a control cost of 2.

Relationship to permutation networks. A large amount of work has been done on switch-based permutation networks, which could be used in place of the specialized connection networks we construct here. In our related work section (Section 6.2), we compare the costs of our networks to two classes of permutation networks from the literature.

5. THE ALGORITHM

Basic idea. To solve Problem 3.4, we make use of *helper matrices*, denoted as $H \in \mathrm{GL}_n(\mathbb{F}_2)$. A helper matrix has the form

$$H = \begin{pmatrix} I_{n-k} & \\ H_2 & I_k \end{pmatrix} \tag{15}$$

and, due to (12), is always self-inverse:

$$H = H^{-1}.$$

Given P and any H, we now get the factorization

$$P = H \cdot HP. \tag{16}$$

Setting $N^{-1} = H$ and $M = HP$, we observe that N satisfies the rank condition $\mathrm{rank}(N_2) = k$ in Problem 3.4. The remaining question is how to design H such that the rank condition on M is also satisfied and to minimize the connectivity and control costs.

Algorithm. Assume P is given tiled as in (8). If $M = HP$ is tiled as in (4), then

$$M_1 = H_2 P_3 + P_1.$$

In other words, we have to find an H_2 such that $\mathrm{rank}(M_1) = k$. H_2 determines H in (15) and hence the factorization of P in (16). The following theorem explains that this is possible.

THEOREM 5.1. *Let $P \in \mathrm{GL}_n(\mathbb{F}_2)$ be tiled as in* (8) *with* $\mathrm{rank}(P_1) = r \leq k$. *Then there exists H_2 with* $\mathrm{rank}(H_2) = k - r$ *and with exactly $k - r$ non-zero entries such that $M_1 = H_2 P_3 + P_1$ has full rank k.*

PROOF. Before we start, we define $E((i_1, j_1), \ldots, (i_k, j_k))$ as the matrix that has ones at the locations $(i_1, j_1), \ldots, (i_k, j_k)$ and zeros elsewhere. The size of the matrix is clear from the context.

Assume we choose $H_2 = E((i, j))$, then $H_2 P_3 + P_1$ is the matrix P_1 with the jth row of P_3 added to its ith row. This gives the basic idea: we select $k - r$ suitable rows of P_3 and add them to suitable $k - r$ rows of P_1 to correct its rank deficiency. Intuitively, this is possible since P and thus its submatrix $\left(\begin{smallmatrix} P_3 \\ P_1 \end{smallmatrix}\right)$ have full rank.

We first consider the special case where P is a permutation, since it is simpler and important for applications.

P a permutation: P_1 contains r base vectors (as rows); the remaining $k - r$ rows, with row indices $i_1, \ldots, i_{k-r}$ (in any order), are zero. The missing $k - r$ base vectors are in P_3, say at row indices $j_1, \ldots, j_{k-r}$. It follows that $H_2 = E((i_1, j_1), \ldots, (i_{k-r}, j_{k-r}))$ satisfies the requirements.

General P: Here we have to do more work to identify the proper row indices. Since P_1 has rank r, we can permute r linear independent columns of P_1 into the first r locations and perform Gaussian elimination on the columns to zero out the last $k - r$ columns. In other words, there is an invertible upper-triangular matrix $G \in \mathrm{GL}_k(\mathbb{F}_2)$ such that $Q_1 = P_1 G$ has the last $k - r$ columns equal to zero. We set $Q_3 = P_3 G$ and get

$$\begin{pmatrix} P_3 \\ P_1 \end{pmatrix} = \begin{pmatrix} Q_3 \\ Q_1 \end{pmatrix} G^{-1}.$$

Now we identify r linear independent rows of Q_1 and call the other row indices (in any order) $i_1, \ldots, i_{k-r}$. In each of the rows, the rightmost $k - r$ entries are equal to zero. Since $\left(\begin{smallmatrix} Q_3 \\ Q_1 \end{smallmatrix}\right)$ has full rank, there are $k - r$ rows whose subvectors consisting of the $k - r$ rightmost entries are linear independent. We denote their indices by $j_1, \ldots, j_{k-r}$. We set $H_2 = E((i_1, j_1), \ldots, (i_{k-r}, j_{k-r}))$. It is clear that $H_2 Q_3 + Q_1$ has full rank and so does

$$(H_2 Q_3 + Q_1) G^{-1} = M_1.$$

The other requirements are also satisfied by H_2. □

The proof of Theorem 5.1 is constructive and yields the following algorithm for solving Problem 3.4.

ALGORITHM 5.2. *Input: $P \in \mathrm{GL}_n(\mathbb{F}_2)$ and $k \leq n$. Output: $N, M \in \mathrm{GL}_n(\mathbb{F}_2)$ such that $P = N^{-1} M$ and* $\mathrm{rank}(M_1) = \mathrm{rank}(N_1) = k$. *We use the previous notation for the matrix tiles.*

Case 1: P a permutation matrix.

(1) *Locate the $k-r$ all-zero rows of P_1. Denote their indices (in any order) with $i_1, \ldots, i_{k-r}$.*
(2) *Locate the $k-r$ non-zero rows of P_3. Denote their indices with $j_1, \ldots, j_{k-r}$.*
(3) *Set $H_2 = E((i_1, j_1), \ldots, (i_{k-r}, j_{k-r}))$. Output $N^{-1} = H$ and $M = HP$.*

Case 2: Arbitrary P.

(1) *Compute $G \in \mathrm{GL}_k(\mathbb{F}_2)$ such that $Q_1 = P_1 G$ has the last $k-r$ columns equal to zero. Set $Q_3 = P_3 G$.*
(2) *Locate r linear independent rows of Q_1. Denote the remaining row indices (in any order) by $i_1, \ldots, i_{k-r}$.*
(3) *Locate $k-r$ rows of Q_3 such that their subvectors consisting of the last $k-r$ entries are linear independent. Denote the row indices with $j_1, \ldots, j_{k-r}$.*
(4) *Set $H_2 = E((i_1, j_1), \ldots, (i_{k-r}, j_{k-r}))$. Output $N^{-1} = H$ and $M = HP$.*

The correctness of Algorithm 5.2 follows from Theorem 5.1. Termination is obvious.

THEOREM 5.3. *Algorithm 5.2 terminates and is correct.*

Optimality. Algorithm 5.2 always produces a "partially" optimal solution (one stage is optimal). In some important cases the solution is optimal.

THEOREM 5.4. *For given P and k, Algorithm 5.2 produces a solution in which the read-stage is optimal with respect to both connectivity and control cost.*

If P is a permutation matrix, then also the write-stage is optimal with respect to both connectivity and control cost.

PROOF. We have to compare against the lower bounds in Theorems 4.1 and 4.2.

Read-stage: Theorem 5.1 establishes that in Algorithm 5.2 $N_2 = H_2$ has rank $k-r$. Thus $\mathrm{conn}(N) = 2^{k-\mathrm{rank}(P_1)}$, which is minimal (Theorem 4.1). Further, $N = H$ incurs $k-r$ additions, i.e., $\mathrm{cost}(N) = k - \mathrm{rank}(P_1)$, which is minimal (Theorem 4.2).

Write-stage, P a permutation: $M = HP$, which implies $M_2 = H_2 P_4 + P_2$. Since $\mathrm{rank}(P_1) = r$, $\mathrm{rank}(P_2) = k-r$. H_2 is constructed to extract the $k-r$ nonzero rows with indices j_ℓ from P_3. Since P is a permutation, this implies that the j_ℓth row of P_4 is zero. As a consequence $H_2 P_4$ is zero and thus $M_2 = P_2$. Further, $P^{-1} = P^T$ and hence $P_1' = P_1^T$. In summary, $\mathrm{rank}(M_2) = k - r = k - \mathrm{rank}(P_1) = k - \mathrm{rank}(P_1')$, which is minimal (Theorem 4.1). Further, M incurs $k-r$ additions, i.e., $\mathrm{cost}(M) = k - \mathrm{rank}(P_1')$, which is minimal (Theorem 4.2). □

Special properties in the case of bit permutations. For bit permutations P, our algorithm produces optimal solutions. A few other special properties hold in this case and are discussed next.

The proof of Theorem 5.4 asserts that if P is a bit permutation, then Algorithm 5.2 yields $M_2 = P_2$. In other words, M has the form

$$M = \begin{pmatrix} P_4 & P_3 \\ P_2 & H_2 P_3 + P_1 \end{pmatrix} \tag{17}$$

and hence differs from P only in the bottom right $k \times k$ submatrix. In the general case, both bottom submatrices M_1, M_2 differ.

Further, the matrix $M_1 = H_2 P_3 + P_1$ is a permutation matrix in this case, since it has full rank and at most k and thus precisely k nonzero entries.

Finally, the matrix H_2 obtained by Algorithm 5.2 contains exactly $k - r$ ones, which are located in precisely those rows of P_1 that are zero. The same holds for P_2. Since $\mathrm{rank}(P_2) = \mathrm{rank}(H_2) = k - r$, we get $\mathrm{im}(M_2) = \mathrm{im}(P_2) = \mathrm{im}(H_2) = \mathrm{im}(N_2)$. As a consequence, using Lemma 3.1, both connection networks in Fig. 2 decompose into blocks equivalently.

Connection networks. The read stage of the solutions produced by Algorithm 5.2 always has the form

$$N = N^{-1} = \begin{pmatrix} I_{n-k} \\ E((i_1, j_1), \ldots, (i_{k-r}, j_{k-r}))\ \ I_k \end{pmatrix}.$$

This makes the decomposition of the connection network according to Theorem 4.3 easy by setting

$$T_\ell = E((i_\ell, j_\ell)).$$

Example. As an example, we derive a solution for $P = C_6^2$ streamed with width 2^3, as shown below. We perform Algorithm 5.2 step by step. P is a permutation; thus, Case 1 applies.

$$P = \left(\begin{array}{ccc|ccc} . & . & 1 & . & . & . \\ . & . & . & 1 & . & . \\ . & . & . & . & 1 & . \\ \hline . & . & . & . & . & 1 \\ 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \end{array}\right).$$

Step 1: P_1 has two all-zero rows, namely the last two with indices $i_1 = 2$ and $i_2 = 1$.

Step 2: P_3 has two non-zero rows, namely the last two with indices $j_1 = 2$ and $j_2 = 1$.

Step 3: We get $H_2 = E((2,2),(1,1))$. H_2 is a 3×3 matrix with two 1 values. The desired factorization is readily computed through $N^{-1} = H$, $M = HC_6^2$:

$$C_6^2 = \left(\begin{array}{ccc|ccc} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ \hline . & . & . & 1 & . & . \\ . & 1 & . & . & 1 & . \\ . & . & 1 & . & . & 1 \end{array}\right) \left(\begin{array}{ccc|ccc} . & . & 1 & . & . & . \\ . & . & . & 1 & . & . \\ . & . & . & . & 1 & . \\ \hline . & . & . & . & . & 1 \\ 1 & . & . & 1 & . & . \\ . & 1 & . & . & 1 & . \end{array}\right). \tag{18}$$

M matches (17) and M_1 is a permutation matrix as asserted above. The solution has to be optimal by Theorem 5.4. Indeed, $\mathrm{conn}(M) = \mathrm{conn}(N) = 4$ and $\mathrm{cost}(M) = \mathrm{cost}(N) = 2$, which meets the lower bounds given in Theorems 4.1 and 4.2. Decomposing M_2 and N_2 as described in Theorem 4.3, we construct switching networks that require a total of 16 switches.

It is interesting to note that the solution introduces bit arithmetic, namely two

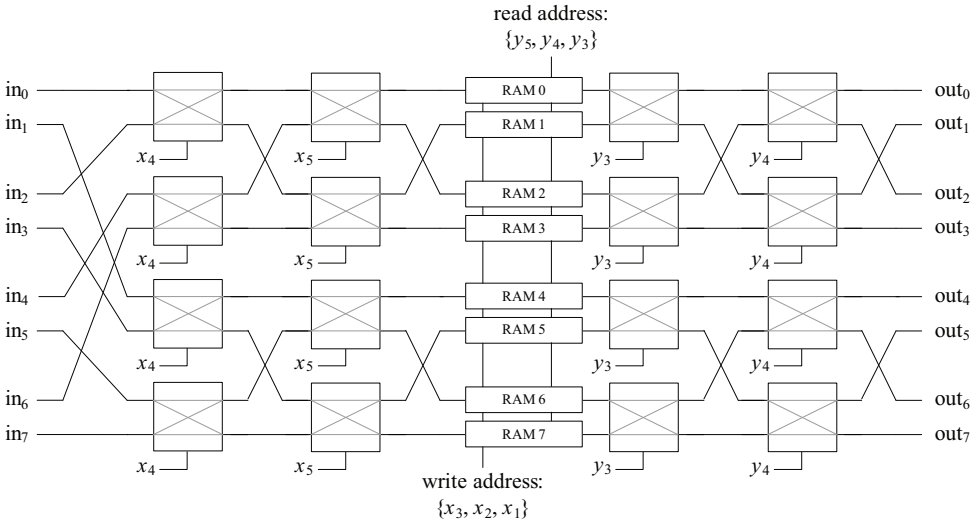


Fig. 6. Datapath for $P = C_6^2$ with streaming width 8, based on the decomposition in (18).

additions or xors for each address computation, even though the original P requires none.

Now, we discuss the complete hardware implementation of this example, seen in Fig. 6. Note that M corresponds to the matrix M seen in (7), so we will implement M as in the previous example, seen in Fig. 5.

We consider the input vector of length 64 to be indexed with addresses $x \in \mathbb{F}_2^6$, with the upper 3 bits corresponding to the stage number, and the lower 3 bits indicating the location within the stage. We write x as $(x_5, x_4, x_3, x_2, x_1, x_0)^T$, with x_5 as the most significant bit and x_0 as the least significant. Likewise, we have an output vector indexed with address $y \in \mathbb{F}_2^6$ of the same form.

Our factorization of M's connection network was previously shown in (14). From this, we determine the following characteristics for the write stage. As explained in the example in Section 4.2, the first stage is controlled by x_4 and the second stage is controlled by x_5.

As discussed in Section 3, the memory write addresses are calculated directly from M. The write addresses are given by $M_4 x_2 + M_3 x_1$. So, this gives memory write addresses

$$M_4 x_2 + M_3 x_1 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_5 \\ x_4 \\ x_3 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} = \begin{pmatrix} x_3 \\ x_2 \\ x_1 \end{pmatrix}.$$

We see that the memory address where each word must be written is the three bit value given by $(x_3, x_2, x_1)^T$. We determine the input and output connections of each block from the cosets, as discussed in Lemma 3.1 and demonstrated in Section 4.2.

We perform the same process for N. Note that $N^{-1} = N$. Following Theorem 4.3,

we decompose N to

$$N = \left(\begin{array}{ccc|ccc} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ \hline . & . & . & 1 & . & . \\ . & 1 & . & . & 1 & . \\ . & . & 1 & . & . & 1 \end{array}\right) = \left(\begin{array}{ccc|ccc} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ \hline . & . & . & 1 & . & . \\ . & 1 & . & . & 1 & . \\ . & . & 1 & . & . & 1 \end{array}\right) \left(\begin{array}{ccc|ccc} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ \hline . & . & . & 1 & . & . \\ . & . & . & . & 1 & . \\ . & . & 1 & . & . & 1 \end{array}\right).$$

The first stage has input sets

$$\begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \mathrm{im}(N_1^{-1} T_2) = \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \mathrm{im} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right\}$$

and output sets given by

$$N_1 \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \mathrm{im}(T_2) = \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \mathrm{im} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right\}.$$

The second stage has input and output sets given by

$$\begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \mathrm{im}(T_1) = \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \mathrm{im} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} x_2 \\ x_1 \\ x_0 \end{pmatrix} + \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \right\}.$$

So, the first stage has input and output sets $\{0,1\}$, $\{2,3\}$, $\{4,5\}$, and $\{6,7\}$. The second stage has input and output sets $\{0,2\}$, $\{1,3\}$, $\{4,6\}$, and $\{5,7\}$. Using the technique described in Section 4.2, we see that the first stage is controlled by y_3, and the second by y_4. We determine the memory read addresses from

$$N_4 y_2 + N_3 y_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_5 \\ y_4 \\ y_3 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} y_2 \\ y_1 \\ y_0 \end{pmatrix} = \begin{pmatrix} y_5 \\ y_4 \\ y_3 \end{pmatrix}.$$

So, the memory read addresses are given by the three bit value $(y_5, y_4, y_3)^T$. The resulting implementation is visualized in Fig. 6.

We consider more examples in the next section.

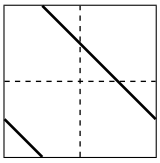
6. EXAMPLES AND APPLICATIONS

In this section, we provide more examples and discuss relevant applications of streaming permutations. First, we examine several important permutations and present the solutions obtained using Algorithm 5.2. For each example, we give the exact specification of the solution as well as the solution's costs. Of particular interest is the stride permutation $\boldsymbol{L}$, which is given in Example 5 below. Later, we discuss related work and compare our solutions against those found in the literature.

6.1 Decomposition examples

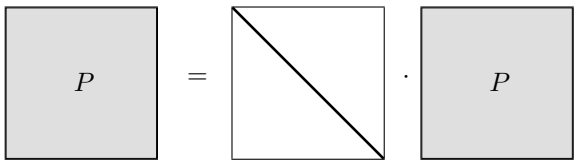
We present several example classes of permutations and derive their implementations for arbitrary streaming width. In addition to an exact specification, we use a visual representation of the solution matrices to show the form of the matrix under

general conditions. In these pictures, each box represents a matrix P, N, or M. The dashed lines show the divisions between the submatrices, and the solid black lines indicate segments with value 1 in the matrix. Gray boxes are used to indicate portions of the matrix that are unknown or not specified in the problem. Lastly, the blank areas of the box indicate portions of the matrix equal to zero. For example, the matrix C_n^2 given in (10) would be represented as



A summary of the results from this section is shown later in Table I.

Example 1: P with full rank(P_1). If the bottom-right matrix P_1 has full rank (i.e., $\text{rank}(P_1) = k$), then P fulfills the restrictions by itself. So, a solution of $P = N^{-1} \cdot M$ is given by $P = I \cdot IP$. This factorization can be visualized as follows:



The cost incurred by N is given by:

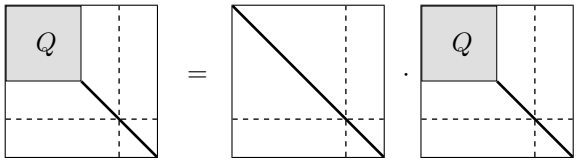
$$\text{conn}(N) = 1 \quad \text{and} \quad \text{cost}(N) = 0.$$

The cost of M, $\text{cost}(M)$ is equal to the linear complexity of P, and the connectivity is given by $\text{conn}(M) = \text{rank}(P_2)$.

Example 2: $\pi(Q) \otimes I$. Given $\pi(P) = \pi(Q) \otimes I_{2^\ell}$, Lemma 2.1 shows that

$$P = Q \oplus I_\ell.$$

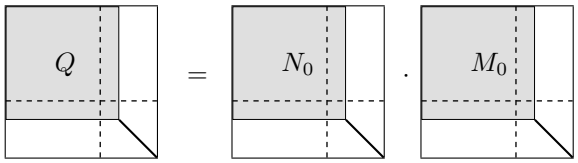
Case 1: $\ell \geq k$. Under this condition, P_1 has full rank and the problem is solved as in Example 1. This produces the following solution:



Case 2: $\ell < k$. If $\ell < k$, we first use Algorithm 5.2 on Q, with streaming width of $k - \ell$. This results in the factorization $Q = N_0^{-1} \cdot M_0$. Once this solution is found, the values for N and M are found according to:

$$N^{-1} = N_0^{-1} \oplus I_\ell, \quad M = M_0 \oplus I_\ell.$$

This solution has the following form:

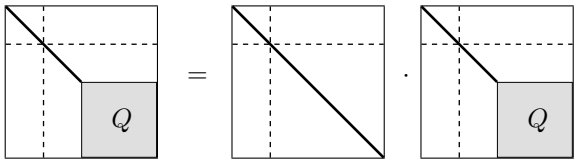


The arithmetic and connection costs of N and M are identical to the costs for N_0 and M_0 (respectively).

Example 3: $\boldsymbol{I \otimes \pi(Q)}$. Given $\pi(P) = \boldsymbol{I_{2^\ell}} \otimes \pi(Q_m)$,

$$P = I_\ell \oplus Q_m.$$

Case 1: $m \leq k$. If $m \leq k$, then P_1 has full rank, so the factorization is trivial. Using the "full-rank" solution seen in Example 1, we obtain the following solution:



Case 2: $m > k$. In this case, we first use Algorithm 5.2 on Q, with streaming width k. This results in the factorization $Q = N_0^{-1} \cdot M_0$. Then, N and M are found according to:

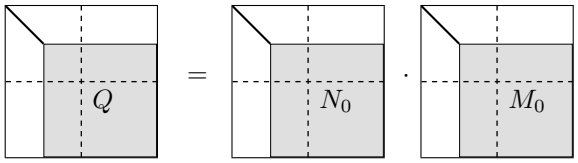
$$N^{-1} = I_{k-m} \oplus N_0^{-1}, \quad M = I_{k-m} \oplus M_0.$$

The resulting solution will have the costs

$$\mathrm{conn}(N) = \mathrm{conn}(N_0), \quad \mathrm{cost}(N) = \mathrm{cost}(N_0),$$

$$\mathrm{conn}(M) = \mathrm{conn}(M_0), \quad \mathrm{cost}(M) = \mathrm{cost}(M_0).$$

The solution has the following form:



Example 4: $\boldsymbol{I \otimes \pi(Q) \otimes I}$. If $\pi(P) = \boldsymbol{I_{2^\ell}} \otimes \pi(Q_m) \otimes \boldsymbol{I_{2^h}}$, then

$$P = I_\ell \oplus Q_m \oplus I_h.$$

If $k \leq h$, then P_1 has full rank, so the solution follows Example 1 above. If $k > h$, P can be regrouped as $P = I_\ell \oplus (Q_m \oplus I_h)$, and the solution can be found as in the $\boldsymbol{I} \otimes \pi(Q)$ case, Example 3.

Example 5: Stride permutation $\boldsymbol{L}$. We derive a solution for stride permutations $\pi(P) = \boldsymbol{L_{2^n,2^s}}$, streamed with width 2^k, $1 \leq k \leq n-1$. Using Lemma 2.1,

$P = C_n^{n-s}$. Let $r = \text{rank}(P_1)$. Then,

$$r = \begin{cases} 0, & k \leq s \text{ and } k \leq n-s \\ k-s, & s < k \leq n-s \\ k-(n-s), & n-s < k \leq s \\ 2k-n, & k > n-s \text{ and } k > s \end{cases}. \tag{19}$$

Applying Algorithm 5.2, we obtain H_2:

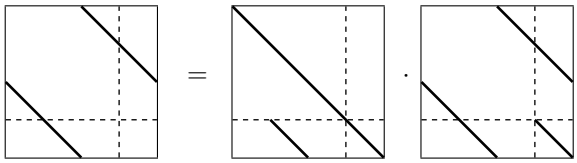
$$H_2 = E((\min(k, n-s) - \ell,\ n - \max(s,k) - \ell) \mid \ell = 1, \ldots, k-r). \tag{20}$$

Thus,

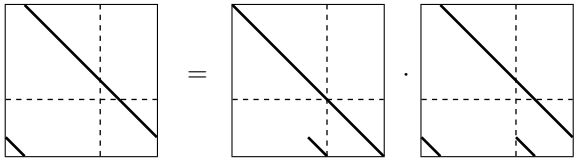
$$H = \begin{pmatrix} I_{n-k} & \\ H_2 & I_k \end{pmatrix} \tag{21}$$

allows us to easily compute the final factorization according to $N^{-1} = H$, $M = HC_n^{n-s}$. This factorization yields $\text{cost}(M) = \text{cost}(N) = k - r$ and $\text{conn}(N) = \text{conn}(M) = 2^{k-r}$.

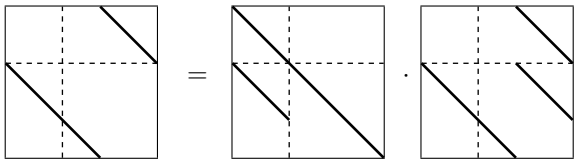
Case 1: $k \leq s$ and $k \leq n-s$. Under these conditions, $P = C_n^{n-s} = N^{-1} \cdot M$ has the form:



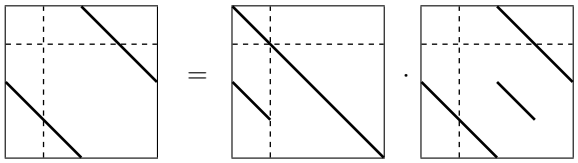
Case 2: $s < k \leq n-s$. In this case, P_1 has one or more non-zero rows, which occur at the top of the matrix. $P = C_n^{n-s} = N^{-1} \cdot M$ has the form:



Case 3: $n-s < k \leq s$. In this case, P_1 has one or more non-zero rows, which occur at the bottom of the matrix. $P = C_n^{n-s} = N^{-1} \cdot M$ has the form:



Case 4: $k > n-s$ and $k > s$. In this case, P_1 has one or more non-zero rows, which occur at the top and bottom of the matrix. $P = C_n^{n-s} = N^{-1} \cdot M$ has the form:



Example 6: Bit reversal $\boldsymbol{R}$. If $\pi(P) = \boldsymbol{R_{2^n}}$, then we see from Lemma 2.1 that

$$P = J_n.$$

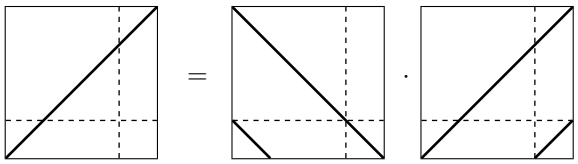
Let $r = \text{rank}(P_1) = 2 \cdot \max(0, k - n/2)$. Applying Algorithm 5.2, we obtain H_2:

$$H_2 = E((r + i, i) \mid i = 0, \ldots, k - r). \tag{22}$$

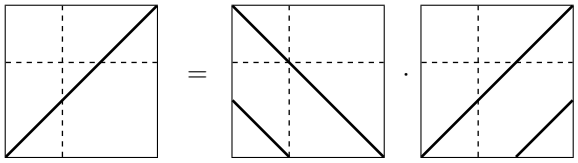
Then, using $P = N^{-1} \cdot M = H \cdot HP$, we produce a solution with the following costs:

$$\text{conn}(M) = \text{conn}(N) = 2^{k-r}, \quad \text{cost}(M) = \text{cost}(N) = k - r.$$

Case 1: $k \leq n/2$. If $k \leq n/2$, then this solution has the following form:



Case 2: $k > n/2$. In this situation, the solution has this form:



Example 7: Hadamard reordering. The Hadamard reordering [Astola and Akopian 1999] permutes the 2^n-element data vector $X = (x_0, x_1, \ldots, x_{2^n-1})^T$ to the vector $Y = (x_{h_{2^n}(0)}, x_{h_{2^n}(1)}, \ldots, x_{h_{2^n}(2^n-1)})^T$, where $h_{2^n}(\cdot)$ is recursively generated according to:

$$\begin{aligned} h_1(0) &= 0 \\ h_{2K}(2i) &= h_K(i) \\ h_{2K}(2i+1) &= 2K - 1 - h_K(i), \quad i = 0, 1, \ldots, K-1. \end{aligned}$$

This permutation is represented as the matrix $\boldsymbol{Z_{2^n}}$. If $\pi(P) = \boldsymbol{Z_{2^n}}$, then $P = S_n$, defined as

$$S_n = \begin{pmatrix} & & & 1 & 1 \\ & & \iddots & \iddots & \\ & \iddots & \iddots & & \\ 1 & \iddots & & & \\ 1 & & & & \end{pmatrix}. \tag{23}$$

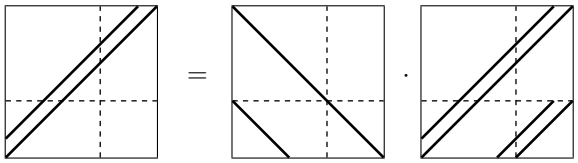
P	$\pi(P)$	solution	conn(M)	conn(N)	cost(M)	cost(N)	optimal
C_n^{n-s}	$\boldsymbol{L_{2^n,2^s}}$	(21)					
Case 1: $k < \min(s, n-s)$,			2^k	2^k	k	k	yes
Case 2: $s \le k \le n-s$,			2^s	2^s	s	s	yes
Case 3: $n-s < k < s$,			2^{n-s}	2^{n-s}	$n-s$	$n-s$	yes
Case 4: $k > \max(n-s, s)$,			2^{n-k}	2^{n-k}	$n-k$	$n-k$	yes
J_n	$\boldsymbol{R_{2^n}}$	(22)					
Case 1: $k \le n/2$,			2^k	2^k	k	k	yes
Case 2: $k > n/2$,			2^{n-k}	2^{n-k}	$n-k$	$n-k$	yes
S_n	$\boldsymbol{Z_{2^n}}$	(24)					
Case 1: $k \le n/2$,			2^k	2^k	k	$n-1+2k$	?
Case 2: $n/2 < k < n/2+1$,			2^{k-1}	2^{k-1}	$k-1$	$n-3+2k$	?
Case 3: $k \ge n/2+1$,			2^{n-k}	2^{n-k}	$n-k$	$3n-2k-1$	?

Table I. Summary of example bit matrices and their solutions.

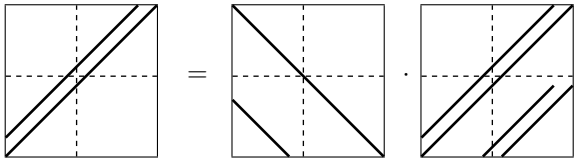
Let $r = \text{rank}(P_1)$. Then, $r = \max(0, 2k-n)$. Applying Algorithm 5.2, we obtain H_2:

$$H_2 = E((k-1-i, k-r-1-i) \mid i = 0, \ldots, k-r-1). \tag{24}$$

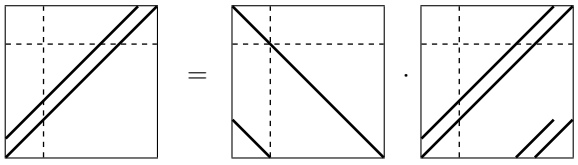
Case 1: $k \le n/2$. The solution $P = N^{-1} \cdot M = H \cdot HP$ has the following form in this case:



Case 2: $n/2 < k < n/2+1$. In this case, the solution has this form:



Case 3: $k \ge n/2+1$. In this case, the solution has the following form:



These solutions have the following costs:

$$\text{conn}(M) = \text{conn}(N) = 2^{k-r}, \quad \text{cost}(M) = n-1+2(k-r), \quad \text{cost}(N) = k-r.$$

Table I presents a summary of results from the examples shown in this section.

6.2 Related work

Stride Permutation Networks for Array Processors. Although there is very little in the literature on RAM-based general streaming permutations, some work has been published on the stride permutation $\boldsymbol{L}$. One solution is presented in [Järvinen et al. 2004]. Like our solution, this method applies to any $\boldsymbol{L}^{2^n}_{2^s}$ with 2^k ports, and a specific network is built for each given (n, s, k).

Instead of using a few large memories, this solution utilizes many smaller FIFO (first-in first-out) memories with interconnection networks between them. This technique requires the minimal amount of total memory usage, which ranges between 2^{n-1} and 2^n memory words (for a permutation on 2^n points). However, there are drawbacks to this approach. In many cases, it is preferable to have larger, centralized memory structures, rather than the large amount of small structures needed here (e.g., FPGA implementation, where the platform contains many built-in RAMs of a particular size).

Now, we quantitatively compare the two implementations in terms of interconnect cost, total memory storage, and memory distribution. We evaluate interconnection cost by counting the number of 2-to-1 multiplexers needed in the final implementation. Each 2-to-2 switch counts as two 2-to-1 muxes. Memory usage is given in the number of data words stored. Memory distribution is analyzed in terms of the total number of memories needed and their sizes.

First we consider our implementations of the permutation $\boldsymbol{L}^{2^n}_{2^s}$, streamed with 2^k ports. We can express the number of multiplexers used in our RAM-based method as

$$M_{\text{RAM}} = \begin{cases} k \cdot 2^k, & k < s \text{ and } k < n - s \\ s \cdot 2^k, & s \le k \le n - s \\ (n-s)2^k, & n - s < j < s \\ (n-k)2^k, & k > n - s \text{ and } k > s \end{cases},$$

and the total number of memory words as

$$D_{\text{RAM}} = 2^{n+1}.$$

This storage requirement is distributed evenly over 2^k RAMs; each must hold 2^{n-k+1} words.

These metrics are computed for the FIFO-based method described by Järvinen et al. in the following way. Let $r = \min(s, n - s)$. The number of multiplexers required is given by

$$M_{\text{FIFO}} = \begin{cases} 2^{k+2^{n-k}}, & k > n - r \\ 2^{k+2^r} + 2^k \cdot M_{\text{S}}(n-k, r), & r \le k \le n - r \\ 2^k(M_{\text{S}}(r, r-k) + 2^{2^k} + M_{\text{S}}(n-k, r)), & k \le n - r \text{ and } k < r \end{cases},$$

$$M_{\text{S}}(n, s) = \begin{cases} 2^s, & n = 2s \\ 2s(n-s), & n \ne 2s \end{cases}.$$

The total amount of memory needed, given in terms of the total number of words

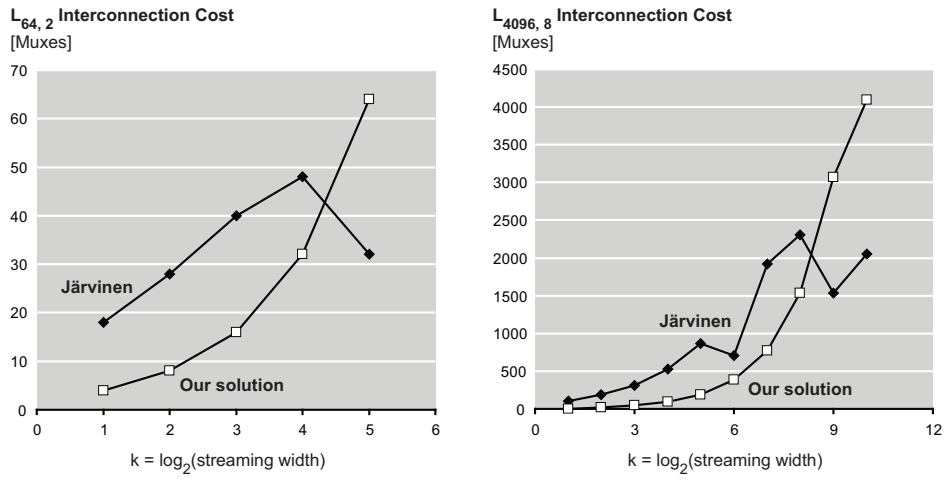


Fig. 7. Comparison of the interconnection cost (measured in 2:1 multiplexers) between our solution and that given by Järvinen et al. Plots demonstrate the trend as the streaming width 2^k increases.

stored, is

$$D_{\text{FIFO}} = \begin{cases} 2^n - 2^k, & k > n - r \\ 2^n - 2^{n-r}, & r \leq k \leq n - r \\ 2^n - 2^r - 2^{n-r} + 2^k, & k \leq n - r \text{ and } k < r \end{cases}.$$

In order to quantify the distribution of memory structures, let F_i represent a FIFO memory that is i elements deep. We can then calculate the total FIFO usage by

$$F_{\text{FIFO}} = \begin{cases} 2^k \cdot \sum_{i=0}^{n-k-1} F_i, & k > n - r \\ 2^k \left(\sum_{i=0}^{r-1} F_i + F_{\text{SPN}}(n-k, r) \right), & r \leq k \leq n - r \\ 2^k \left(F_{\text{SPN}}(r, r-q) + \sum_{i=0}^{k-1} F_i + F_{\text{SPN}}(n-q, r) \right), & k < n - r \text{ and } k < r \end{cases},$$

$$F_{\text{SPN}}(n, s) = \begin{cases} \sum_{i=0}^{s-1} F_{(2^s-1)2^i}, & n = 2s \\ \sum_{i=0}^{n-s-1} \sum_{j=0}^{s-1} F_{i+j}, & n \neq 2s \end{cases}.$$

Using these characterizations, we can compare the two implementations of streaming stride permutations. First, we compare interconnect cost, measured in the number of 2:1 multiplexers. Fig. 7 demonstrates two typical permutations. Note that the irregular pattern seen in the Järvinen line is caused by the different regions in the expression for M_{FIFO}; discontinuities occur at the boundaries.

These graphs demonstrate that for a small number of ports, our proposed permutations have a lower interconnection cost. As the size of the permutation (i.e., n) grows, the crossover point occurs at a larger value of k (i.e., a larger streaming width). For example, we see that for $\boldsymbol{L}_{4096,8}$, our proposed permutation will only have a higher interconnection cost when the number of ports grows to $2^9 = 512$.

Next, we evaluate the total amount of memory used in the two methods. Our proposed implementations use a constant amount of memory: 2^{n+1} data words,

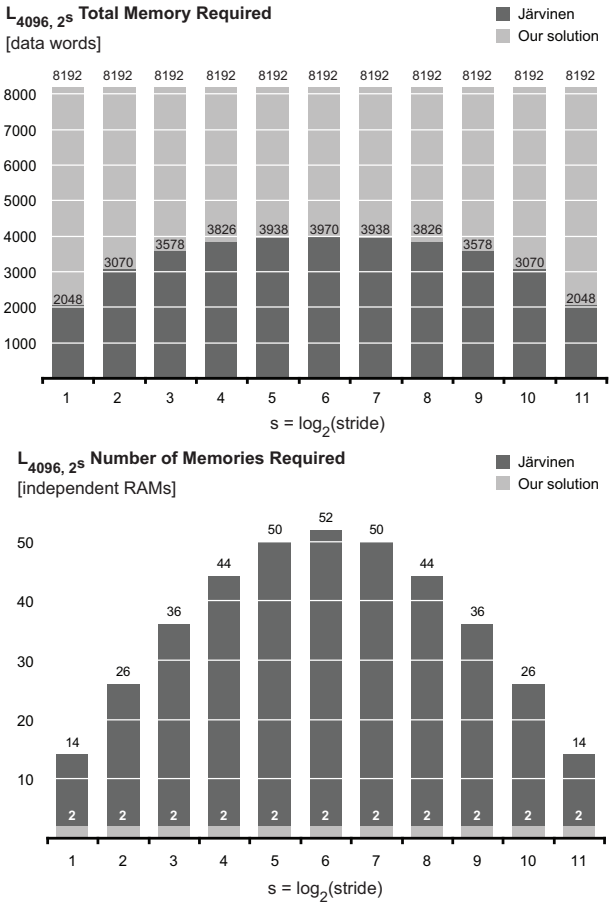


Fig. 8. Comparison of the total amount of memory needed (top) and total number of memories of size ≥ 8 (bottom) for the proposed implementation of $\boldsymbol{L_{4096,s}}$, $2^k = 2$, against the method given by Järvinen et al. Our solutions require between $2\times$ and $4\times$ more memory, but only require 2^k independent memories. Järvinen's implementations require less total storage, but need many more independent memories. We show an example of the breakdown of the memories grouped by size in Fig. 9.

where 2^n is the size of the permutation. In contrast, we see that Järvinen et al. use between 2^{n-1} and 2^n data words. Fig. 8 (top) demonstrates the memory usage of our proposed implementations versus Järvinen's. In this example, we show $\boldsymbol{L_{4096,s}}$, with s from 1 to 11, and with stream width $2^k = 2$. Across all s, our permutations require 8192 words of storage, while Järvinen's implementations range between 2048 and 3970.

Although Järvinen's FIFO-based design requires significantly less storage than the method we propose, the small number of distinct memory structures used in our method make it better suited for many implementations. In Fig. 8 (bottom), we show the total number of memories (of size ≥ 8 data words) for the same set of implementations. Although our designs require more total storage, they only

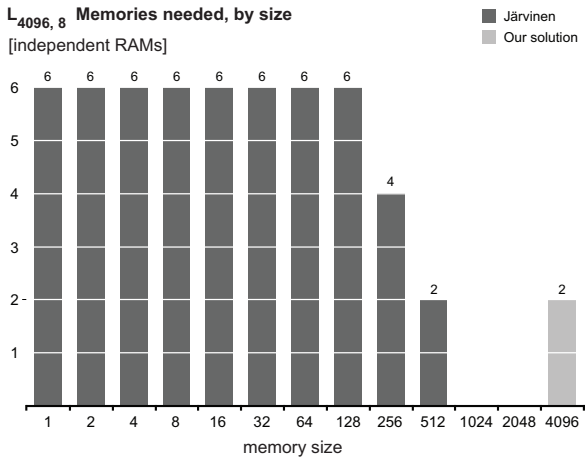


Fig. 9. Illustration of the number of memories needed to implement $\boldsymbol{L}_{\mathbf{4096,2}}$ with $2^k = 2$ ports. The dark bars indicate the memory structures needed for Järvinen's implementation, while the light bars demonstrate the memories needed for ours. We see that the design given by Järvinen et al. requires a total of 54 memory structures, many small. Our proposed design requires 2 large memory structures.

require 2^k distinct memories, while Järvinen's require many more.

In Fig. 9, we show the number of memory structures needed and their capacities for both implementations of $\boldsymbol{L}_{4096,8}$ with two ports ($2^k = 2$). Here, the x-axis shows the size of the memory considered, and the y-axis shows the number of memories of that size needed. We see that our implementation uses two memories, each of size 4096. The implementation by Järvinen uses two memories of size 512, four memories of size 256, and six each of $128, 64, \ldots, 1$. We see that the Järvinen method requires a very large amount of small memory structures, each capable of reading and writing once per clock cycle. In contrast, we see that our design requires only a small number of large memories.

Other work related to stride permutation. Due to the importance of the stride permutation $\boldsymbol{L}$ in the fast Fourier transform, there have been other studies of streaming $\boldsymbol{L}$ implementations.

In [Takala et al. 2003], a memory-based structure is developed that is capable of performing $\boldsymbol{L}_{2^n,2^s}$ on streaming data. This implementation is able to permute data of multiple vector lengths (i.e., values of n) with the same hardware. This solution has a coarse-grained architecture that is similar to our implementations, but there are a few notable distinctions.

In this work, the authors present a memory access scheme that reduces the memory cost from 2^{n+1} data words (as used by our solutions) to 2^n. This optimization is specific to the stride permutation; it relies on the strided reads and writes seen with $\boldsymbol{L}$. Lastly, we note that the generality given (i.e., the ability to perform multiple permutations with the same structure) leads to more complicated address generation and interconnection than is used in our solution.

In [Gorman and Wills 1995], a class of pipelined implementations of the fast Fourier transform is considered. Here, streaming stride permutations $\boldsymbol{L}_{2^n,2^s}$ with

minimum streaming with 2^{n-s} (i.e., $k \geq n-s$) are constructed. Similar to our work, each structure is built to perform one specific permutation. Again, the coarse-grained architecture of this solution space is similar to ours: data passes through an interconnection network, is written into memory, then is read out in a different order and streamed through a last network. Also like our implementations, the storage requirement of this solution is 2^{n+1} data words. If $k \leq s$, then

$$\text{conn}(N) = \text{conn}(M) = 2^{n-s} \quad \text{and} \quad \text{cost}(N) = \text{cost}(M) = n - s$$

for both implementations. However, if $k > s$, then our implementations have

$$\text{conn}(N) = \text{conn}(M) = 2^{n-k} \quad \text{and} \quad \text{cost}(N) = \text{cost}(M) = n - k,$$

while those in [Gorman and Wills 1995] have higher costs

$$\text{conn}(N) = \text{conn}(M) = 2^{n-s} \quad \text{and} \quad \text{cost}(N) = \text{cost}(M) = n - s.$$

In [Láng 1976], implementations of the perfect shuffle ($\boldsymbol{L}_{2^n,2^{n-1}}$) over 2^k memory modules are considered. In this work, Láng assumes that a full 2^k-to-2^k interconnection network connects the memory's outputs back to its inputs. Additionally, this method requires that storage for 2^{n+1} data words be provided (the permutation is performed out of place). Given these assumptions, the author determines a partitioning of the permutation that enables the data to be read and written 2^k words at a time over 2^{n-k} time periods. Compared to our implementation, this solution has equal memory cost and a much higher connectivity cost. In our method, $\text{conn}(M) = \text{conn}(N) = 2$ for all permutations considered by Láng. Our solution decomposes both networks into 2^{k-1} independent 2-to-2 blocks, a significant savings over a full 2^k-to-2^k network.

Other related work. In other recent work [Milder et al. 2009], we have developed a generalized technique for generating a streaming permutation structure for an arbitrary permutation and streaming width. This method also bases the datapath on simple parallel RAMs, but it requires twice as much storage (and twice the number of independent memories) as the method proposed in this paper. Further, [Milder et al. 2009] uses one full w to w switching network (where w is the streaming width) that is not specialized or optimized for the given problem. Lastly, the technique requires a more complicated method to determine memory addresses and switch settings; these values are pre-computed at design time and stored in lookup ROMs.

[Parhi 1992] describes a method for synthesizing register-based data format converter given a permutation and input/output specifications. This technique can produce designs that perform permutations as well as bit-level format conversion of data words. Designs produced by this method consist of independent registers connected with wires and switches or multiplexers. This technique is more general than the one we present in this paper: it applies to any permutation and any streaming width. Furthermore, the designs produced are guaranteed to use the minimal number of registers for the problem.

However, a large drawback of [Parhi 1992] is that its memory consists solely of distributed registers (instead of larger condensed RAMs). This can add considerable cost and complexity to the design, especially on modern FPGAs, where it is more efficient to use the platform's dedicated memory structures.

Permutation networks. In the architecture we describe, we employ specialized permutation networks before and after the RAMs. These networks are spatial; they do not permute across time boundaries. Each network connects 2^k inputs to 2^k outputs (although in most cases we decompose to smaller sub-networks). In the literature, a large amount of work exists on permutation networks of this sort. Here, we discuss the applicability and costs of using these networks in place of our own. Recall, our permutation networks are s stages wide, with each stage containing 2^{k-1} 2-to-2 switches, where $s = \text{rank}(M_2) \leq k$.

First, we consider a class of $2k-1$ stage networks [Benes 1965; Waksman 1968; Lee 1985]. These networks consist of $2k-1$ stages, each with 2^{k-1} 2-to-2 switches. Networks in this class are capable of performing all $2^k!$ possible permutations on 2^k points. So, with pre-computation of the switch settings, one of these networks could easily be used as a drop-in replacement for our network. However, the generality of the network adds cost; it is more than twice the size of our worst-case network.

Next, we examine a class of k stage networks [Lawrie 1975; Pease 1977; Parker 1980]. These networks are capable of performing many possible permutations in one pass, but some important permutations (e.g., Example 7 above) are not covered. In some cases, networks in this class could be used to replace our networks. However, even in these cases, the network's cost is equal to our worst-case cost.

Potentially, networks in either class could be greatly simplified by specialization, i.e., restricting the set of permutations the network may perform. This would lower the implementation cost by allowing unneeded switches to be removed. However, our constructive bottom-up solution is easily automated, and it builds a network precisely specialized for the set of permutations needed for our problem.

6.3 Applications

The streaming permutations considered in this paper have a wide range of applications. In this section, we list and briefly discuss some of the most important.

Transforms. Permutations are extremely important in fast computation of linear transforms. The Cooley-Tukey fast Fourier transform (FFT) and its variants [Van Loan 1992] use stride permutations $\boldsymbol{L}$ and the bit reversal $\boldsymbol{R}$. A similar Cooley-Tukey-like fast algorithm for the Walsh-Hadamard transform [Beauchamp 1984] also uses the stride permutation $\boldsymbol{L}$.

Streaming permutations are applicable to fast algorithms for the discrete cosine transform (DCT) and discrete sine transform (DST). In [Püschel and Moura 2008], fast Cooley-Tukey type algorithms for these transforms are derived. These algorithms use the stride permutation, as well as the permutation $\boldsymbol{K}_{\boldsymbol{2^n,2^m}}$, defined

$$\boldsymbol{K}_{\boldsymbol{2^n,2^m}} = \left(\boldsymbol{I}_{\boldsymbol{2^{m-1}}} \otimes \left(\boldsymbol{I}_{\boldsymbol{2^{n-m}}} \oplus \boldsymbol{J}_{\boldsymbol{2^{n-m}}}\right)\right) \boldsymbol{L}_{\boldsymbol{2^n,2^m}}.$$

Inspection shows that $\boldsymbol{I}_{\boldsymbol{2^{n-m}}} \oplus \boldsymbol{J}_{\boldsymbol{2^{n-m}}} = \pi(Q)$ for

$$Q = \begin{pmatrix} 1 & & \\ 1 & 1 & \\ \vdots & & \ddots \\ 1 & & & 1 \end{pmatrix}.$$

Hence, $\boldsymbol{K}_{\boldsymbol{2^n},\boldsymbol{2^m}} = \pi(P)$ for

$$P = (I_{m-1} \oplus Q)C_n^m.$$

The $\boldsymbol{K}$ permutation can be represented as a linear transformation on the bit representation, so it can be implemented by Algorithm 5.2.

In addition to Cooley-Tukey type algorithms, fast regularized (or constant geometry) algorithms for computing the DCT and DST exist [Astola and Akopian 1999]. These algorithms require the stride permutation $\boldsymbol{L}$ as well as the Hadamard reordering, $\boldsymbol{Z}_{\boldsymbol{2^n}}$, seen in Example 7 in Section 6.1.

Some versions of the Walsh transform utilize Gray code reordering [Beauchamp 1984]. If we let this permutation be represented by $\pi(Q)$, then Q is a lower-left triangular matrix of ones.

$$Q = \begin{pmatrix} 1 & & & & \\ 1 & 1 & & & \\ 1 & 1 & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & 1 & 1 & \dots & 1 \end{pmatrix}.$$

For any streaming width, the bottom right submatrix Q_1 will have full rank, so Q will be factorized as in Example 1, above.

Additionally, some optimizations in the transform domain can be simply implemented using our method. For example, [Duhamel 1990] shows how the bit reversal and matrix transposition needed in the 2-D FFT can be performed together at a reduced cost. In our framework, this optimization is easily realized as

$$\pi(P) = (\boldsymbol{I}_{\boldsymbol{2^n}} \otimes \boldsymbol{R}_{\boldsymbol{2^n}})\boldsymbol{L}^{\boldsymbol{2^{2n}}}_{\boldsymbol{2^n}} \quad \text{and} \quad P = (I_n \oplus J_n) \cdot C_{2n}^n.$$

Lastly, we note that the initial impetus for this work came from the transform domain. In particular, we were motivated by flexible hardware implementations of the FFT. In [Milder et al. 2008], we use the permutations described in this paper to generate FFT implementations for FPGAs. In Fig. 10 (taken from [Milder et al. 2008]), we demonstrate a large range of tradeoffs between cost (in FPGA area) and throughput (in million samples per second). Additionally, we compare the designs to previous work [Nordin et al. 2005], which uses similar generation techniques, but less efficient permutation structures, and to the commercially-available Xilinx LogiCore library cores, which exhibit limited scalability. The flexible nature of the permutations proposed in this work should enable a similar space of highly flexible hardware designs for other transforms and applications.

Transposition. Permutations can be important applications themselves. For example, the transposition (or corner turn) of an $n \times n$ matrix is simply the stride permutation $\boldsymbol{L}_{n^2,n}$. Transposition is a very important application; it is commonly used in multi-dimensional separable signal processing (e.g., multi-dimensional DFT, image processing, etc.). This permutation is easily implemented with our method.

Other. Other possible applications for this work occur in a variety of domains. For example, sorting networks [Bilardi 1989] and Viterbi coding [Viterbi 1967] both access data in a stride permutation pattern. Our technique can be used in streaming implementations of these applications.

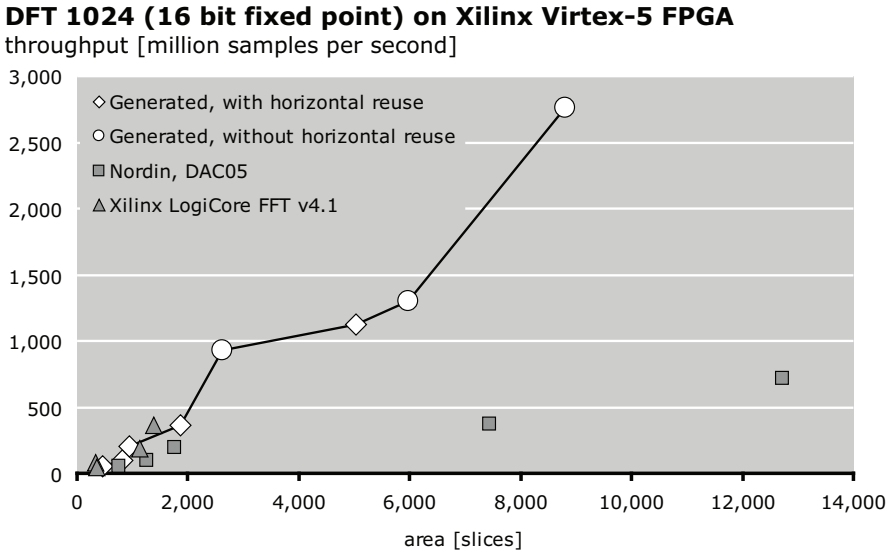


Fig. 10. FPGA cores for the fast Fourier transform, using the streaming permutations proposed in this paper. This figure is taken from [Milder et al. 2008].

7. CONCLUSIONS

Many applications (such as linear signal processing transforms) can benefit from hardware acceleration, but data permutations can be a difficult barrier preventing high performance streaming architectures. The difficulty lies in trying to map the permutation to 2^k independent memory banks without causing conflicts.

In this paper, we described an architecture that uses dual-ported memory and simple interconnection networks. We showed how to mathematically represent the problem and its restrictions. We gave an algorithm that uses matrix manipulations to reach a solution, given a permutation and a streaming width.

If the permutation can be expressed as a permutation on the bits, the solution obtained is provably optimal with regard to network connectivity and control cost. Furthermore, the algorithm and hardware generation are easily automated; we have written a tool that takes a permutation and streaming width, and produces a synthesizable Verilog description.

We provided several example permutations as well as the corresponding solutions produced by the algorithm. We compared our method with relevant related work, and examined prior work on permutation networks and their applicability to our solution. Lastly, we discussed several applications for this method.

Extension. In this paper, we examined the case where a RAM-based datapath is constructed for a specific permutation P. However, some applications require or would benefit from the ability to perform several different permutations $P_{(0)}, P_{(1)}, \ldots, P_{(m-1)}$ with one datapath. This structure would take in streaming data as well as additional control bits to specify which of the m permutations to perform.

Our proposed approach could be adapted to this problem. In some cases, it may only be necessary to extend the control logic (but leave the memories and switching networks intact). In others, the switching networks must be augmented to support different sets of connections. By extending our approach, it may be possible to reduce costs by factoring the permutations in a way to facilitate the sharing of control and switching elements across the different permutations.

ACKNOWLEDGMENT

This work was supported by DARPA under DOI grant NBCH-1050009 and by NSF awards 0325687 and 0702386. A patent application for this work is pending.

Finally, the authors would like to thank Gianfranco Bilardi for helpful discussions and the anonymous reviewers for comments that helped to improve the paper.

REFERENCES

Astola, J. and Akopian, D. 1999. Architecure-oriented regular algorithms for discrete sine and cosine transforms. *IEEE Transactions on Signal Processing 47,* 4, 1109–1124.

Beauchamp, K. G. 1984. *Applications of Walsh and Related Functions*. Academic Press.

Benes, V. E. 1965. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press.

Bernstein, D. S. 2005. *Matrix Mathematics*. Princeton University Press.

Bilardi, G. 1989. Merging and sorting networks with the topology of the omega network. *IEEE Transactions on Computers 38,* 10, 1396–1403.

Bürgisser, P., Clausen, M., and Shokrollahi, M. A. 1997. *Algebraic Complexity Theory*. Springer.

Duhamel, P. 1990. A connection between bit reversal and matrix transposition: Hardware and software consequences. *IEEE Transactions on Acoustics, Speech, and Signal Processing 38,* 11, 1893–1418.

Gorman, S. F. and Wills, J. M. 1995. Partial column FFT pipelines. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing 42,* 6, 414–423.

Järvinen, T. S., Salmela, P., Sorokin, H., and Takala, J. H. 2004. Stride permutation networks for array processors. In *IEEE International Conference on Application-Specific Systems, Architectures, and Processors*.

Láng, T. 1976. Interconnections between processors and memory modules using the shuffle-exchange network. *IEEE Transactions on Computers 25,* 5, 496–503.

Lawrie, D. H. 1975. Access and alignment of data in an array processor. *IEEE Transactions on Computers 24,* 12, 1145–1155.

Lee, K. Y. 1985. On the rearrangeability of $2(\log_2 N) - 1$ stage permutation networks. *IEEE Transactions on Computers 34,* 5, 412–425.

Milder, P. A., Franchetti, F., Hoe, J. C., and Püschel, M. 2008. Formal datapath representation and manipulation for implementing DSP transforms. In *Proceedings of the 45th Annual ACM/IEEE Conference on Design Automation (DAC)*.

Milder, P. A., Hoe, J. C., and Püschel, M. 2009. Automatic generation of streaming datapaths for arbitrary fixed permutations. In *Proceedings of Design, Automation and Test in Europe*.

Nordin, G., Milder, P. A., Hoe, J. C., and Püschel, M. 2005. Automatic generation of customized discrete Fourier transform IPs. In *Proceedings of the 42nd Annual ACM/IEEE Conference on Design Automation (DAC)*.

Parhi, K. K. 1992. 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 39,* 7, 423–440.

Parker, D. S. 1980. Notes on shuffle/exchange-type switching networks. *IEEE Transactions on Computers 29,* 3, 213–222.

Pease, M. C. 1977. The indirect binary N-cube microprocessor array. *IEEE Transactions on Computers 26,* 5, 458–473.

Püschel, M. and Moura, J. M. F. 2008. Algebraic signal processing theory: Cooley-Tukey type algorithms for DCTs and DSTs. *IEEE Transactions on Signal Processing 56,* 4, 1502–1521.

Takala, J. H., Järvinen, T. S., and Sorokin, H. T. 2003. Conflict-free parallel memory access scheme for FFT processors. In *Proceedings of the 2003 International Symposium on Circuits and Systems*.

Van Loan, C. 1992. *Computational Frameworks for the Fast Fourier Transform*. SIAM.

Viterbi, A. J. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory 13,* 2, 260–269.

Waksman, A. 1968. A permutation network. *Journal of the ACM 15,* 1, 159–163.

Received Month 2005; revised Month 2005; accepted Month 2005