Memory-Efficient Fast Fourier Transform on Streaming Data by Fusing Permutations

François Serre
serref@inf.ethz.ch
Department of Computer Science
ETH Zurich

Markus Püschel
pueschel@inf.ethz.ch
Department of Computer Science
ETH Zurich

ABSTRACT

We propose a novel FFT datapath that reduces the memory requirement compared to state-of-the-art RAM-based implementations by up to a factor of two. The novelty is in a technique to fuse the datapaths for the required perfect shuffle and bit reversal and is applicable to an entire design space of FFT implementations with varying degrees of reuse and number of input ports. We implemented a tool to generate this FFT design space for a given input size and to benchmark against prior work. The results show a reduction of half the RAM banks and/or half the logic complexity used for the permutations. The technique for fusing permutations is more generally applicable beyond the FFT.

CCS CONCEPTS

• **Hardware** → **Digital signal processing**; *Application specific integrated circuits*; *High-level and register-transfer level synthesis*; • **Theory of computation** → *Circuit complexity*;

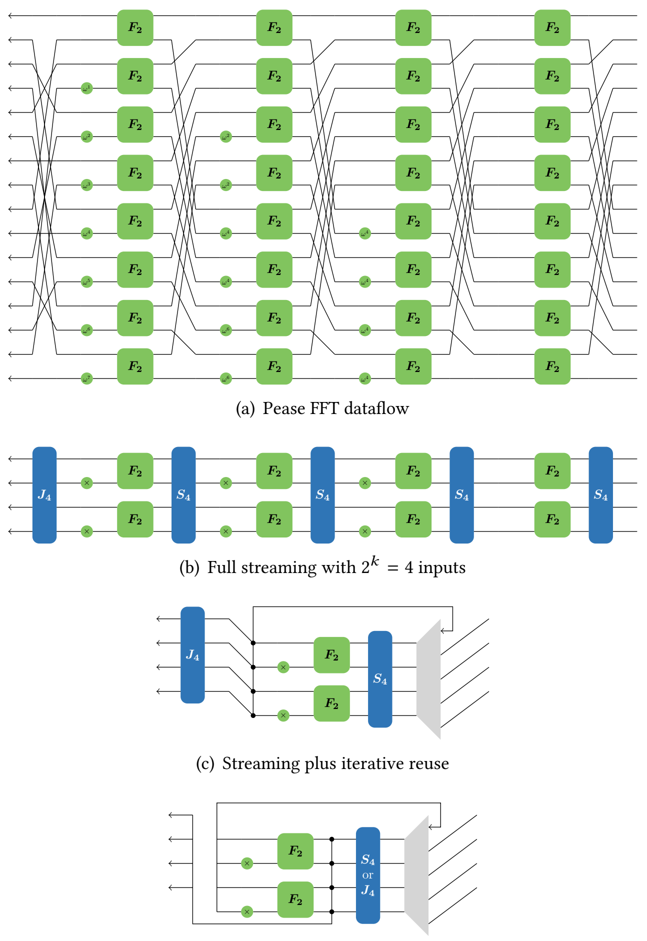
KEYWORDS

Streaming datapath; Fast Fourier Transform; Data reordering; Connection network; Linear permutation; Stride permutation; Bit-reversal

ACM Reference Format:
François Serre and Markus Püschel. 2018. Memory-Efficient Fast Fourier Transform on Streaming Data by Fusing Permutations. In *FPGA '18: 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 25–27, 2018, Monterey, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3174243.3174263

1 INTRODUCTION

The discrete Fourier transform (DFT) is a ubiquitous tool in signal processing and beyond, used in image and speech processing, radar, wireless communication (e.g., in the LTE standard), and many other domains. Thus, fast and efficient implementations of fast Fourier transforms (FFTs), in software and in hardware, and in particular for embedded systems are of high importance. Much work has been devoted to this topic and produced systematic methods to design efficient FFT circuits that cover the entire space of design tradeoffs from large and fast to small and slow [1–5]. In this paper



(a) Pease FFT dataflow

(b) Full streaming with $2^k = 4$ inputs

(c) Streaming plus iterative reuse

(d) Streaming plus iterative reuse with fused permutation

Figure 1: (a) Dataflow (right to left) of the radix-2 Pease FFT for size 2^4; (b) "vertically folded" design for $2^k = 2^2$ input ports; (c) in addition "horizontally folded" design in which the first four stages are iteratively reused; (d) our contribution: design with two permutations fused.

we extend prior work with a novel method that can reduce memory requirement by roughly one half. The method is more generally applicable beyond FFTs: it designs a circuit that can perform a small number of data permutations, which take the input streamed over several cycles. Our contribution is best explained with an example.

Example: FFT on 16 points. Fig. 1(a) shows the dataflow of a radix-2 Pease FFT [6, 7] on $2^n = 16$ points. Note that all dataflows in this paper are from right to left because of the corresponding matrix notation introduced later. The FFT comprises four identical stages (except for the twiddle scaling shown as little circles) of eight parallel butterflies F_2 preceded by a perfect shuffle, followed by the bit reversal permutation. This dataflow can be used for a fully-parallel implementation that has high throughput but also high cost.

The cost can be reduced by exploiting the repetitive structure of this FFT. A first method "folds the dataflow vertically" to obtain a design like Fig. 1(b) [8–10]. Now the circuit operates on streaming data, which means that the dataset arrives on 2^k ports during 2^t cycles, where $n = t + k$. In the figure $k = 2$. However, this design requires streamed permutation circuits (represented with blue boxes) between the butterfly stages. These require memory, as data may now be permuted across cycles, and routing components, as elements arriving on a given input port may need to be directed to different output ports. Efficient, and sometimes proven optimal methods, for implementing these have been developed in the literature. There are two classes of methods. One designs a circuit that can handle any permutation [11, 12], parameterized by the control logic at runtime. However, this flexibility comes at the price of a higher area cost. The second class consists of datapaths that are specialized for the desired permutation [13–17], which thus reduces cost.

Back to the FFT, Figure 1(a) has another symmetry: the first four stages are almost identical. Therefore, it is possible to "fold the dataflow horizontally" to reuse over time a single hardware stage [6]. The two types of folding can be combined [9, 10], resulting, for example, in the design shown in Fig. 1(c). If fully folded in both dimensions, the design is very compact. In the case shown it contains only two butterflies, two complex multipliers, and the hardware to perform the bit reversal (represented in Fig. 1(a) with the blue box labeled with J_4) and the perfect shuffle (labeled with S_4). The work in [9, 10] considers and generates the entire design space given by varying the degree of folding in both dimensions.

The architecture we propose is shown in Fig. 1(d) and fuses the hardware performing the two permutations to reduce cost, and in particular the memory required. Note that the entire discussion in this example can be extended to a Pease FFT of arbitrary radix. The method for fusing the streaming permutation is more generally applicable but the FFT was the motivation for this work.

Contributions. Our main contributions are as follows:

- We present a method to design a specialized datapath that can realize a given (small) set of permutations, taking the input streamed over several cycles. This datapath is cheaper than one capable of performing all permutations. The method is limited to the class of *linear* permutations, which contains bit reversal, perfect shuffle, matrix transpositions, and permutations needed in other FFTs beyond Pease, sorting networks, Viterbi decoders, filter banks, and other algorithms. The datapath we design consists of basic logic and RAMs, which is well-suited for implementation on FPGAs.
- As a major application, we propose a novel variant of a streamed FFT architecture (as shown in Fig. 1(d)) that reduces the RAMs required by prior work by up to one half.
- We implemented a generator [18] that can produce the entire design space sketched in Fig. 1. The input is the FFT size 2^n and the number of input ports for the design on Fig. 1(d). The output is RTL Verilog. The generator also supports different radices larger than 2.
- We show benchmarks to prior work confirming the benefits of the new FFT datapath.

2 STREAMED LINEAR PERMUTATIONS

As mentioned in the introduction, the bit reversal and the perfect shuffle used in the Pease FFT are *linear permutations*. In this section, we define this class, and review prior work on their implementation as streaming hardware.

Perfect shuffle. The perfect shuffle is the permutation that interleaves the first and second half of a list of 2^n elements. It appears in the first four stages of Fig. 1(a). For instance, if we consider 8 elements indexed from 0 to 7, these get rearranged such that the element i is mapped to the position $2i$ if $i < 4$, or $2i - 7$ otherwise. If we write the binary representation of i as a column vector i_b of 3 bits with the most significant bit on top, this means

$$\begin{pmatrix}0\\0\\0\end{pmatrix} \mapsto \begin{pmatrix}0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\end{pmatrix} \mapsto \begin{pmatrix}0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\end{pmatrix} \mapsto \begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\1\end{pmatrix} \mapsto \begin{pmatrix}1\\1\\0\end{pmatrix},$$

$$\begin{pmatrix}1\\0\\0\end{pmatrix} \mapsto \begin{pmatrix}0\\0\\1\end{pmatrix}, \begin{pmatrix}1\\0\\1\end{pmatrix} \mapsto \begin{pmatrix}0\\1\\1\end{pmatrix}, \begin{pmatrix}1\\1\\0\end{pmatrix} \mapsto \begin{pmatrix}1\\0\\1\end{pmatrix}, \text{ and } \begin{pmatrix}1\\1\\1\end{pmatrix} \mapsto \begin{pmatrix}1\\1\\1\end{pmatrix}.$$

We observe that the perfect shuffle rotates up the binary representation i_b of its indexes.

More generally, for a set of 2^n elements, the perfect shuffle maps an index $0 \leq i < 2^n$ to the index j such that

$$j_b = S_n \cdot i_b,$$

where S_n is the cyclic shift matrix:

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

In summary, the invertible $n \times n$ bit matrix S_n defines the perfect shuffle permutation, which we denote with $\pi(S_n)$, on 2^n elements.

In the Pease FFT for a general radix 2^r, the shuffle between stages is given by $S_{n,r} = S_n^r$.

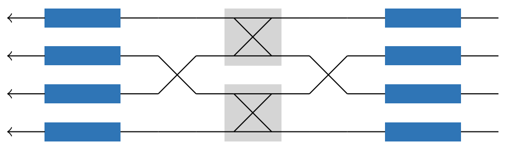
Linear permutations. In general, a linear permutation [19, 20] π on 2^n elements is a permutation such that there exists an $n \times n$ invertible bit matrix[1] P that satisfies, for $0 \leq i < 2^n$,

$$\pi : i \mapsto j \Leftrightarrow j_b = P \cdot i_b. \tag{2}$$

Conversely, for any $n \times n$ invertible bit-matrix P, there is a unique linear permutation that satisfies (2), and we denote it with $\pi(P)$.

For a given n, there are a total of $\prod_{i=0}^{n-1}(2^n - 2^i)$ such P, and thus linear permutations. This means most permutations on 2^n points are not linear (e.g., linear requires that 0 is mapped to 0), but, interestingly, many permutations in signal processing algorithms are linear. Examples include permutations appearing in FFTs, fast

[1]mathematically, $P \in \mathrm{GL}(n, 2)$.


(a) RAM/SNW/RAM

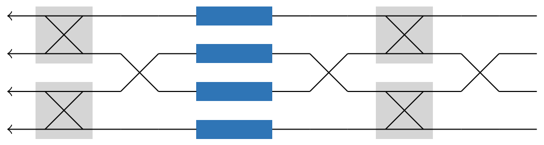

(b) SNW/RAM/SNW

Figure 2: Perfect shuffle on $2^n = 16$ elements, streamed with $2^k = 4$ ports over $2^t = 4$ cycles. RAM banks permute *in time*, i.e., across cycles. Switching networks (SNWs) permute *in space*, i.e. across ports.

cosine transforms, Viterbi decoders, sorting networks, filter banks, and many others.

The linear permutations considered in this paper are even bit-index permutations, a subset of linear permutations for which P is itself a permutation matrix, such as the matrix in (1). However, the method we propose works with any linear permutation.

Bit reversal. Besides the perfect shuffle we consider the bit reversal, which is defined as the permutation that reverses the bits of the indices. Therefore, it is the linear permutation $\pi(J_n)$, where

$$J_n = \begin{pmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{pmatrix}.$$

In the Pease FFT of a general radix 2^r, $r|n$ (r divides n), the bit reversal operates at coarser granularity and is given by $J_{n,r} = J_{n/r} \otimes I_r$. This means that every entry in $J_{n/r}$ is multiplied by the $r \times r$ identity matrix I_r.

Streamed linear permutations. In the streaming reuse structures (Figs. 1(b) and 1(c)), the linear permutations have to permute input data streamed in 2^t chunks of 2^k elements, where $2^n = 2^{t+k}$. Prior work provides optimal RAM-based implementations of such streaming linear permutation (SLP). The first one [17] uses an architecture composed of a *spatial SLP* block consisting of a network of 2×2-switches (SNW) framed by two *temporal SLPs*, each made of an array of 2^k RAM banks (See Fig. 2(a)). The second method [16, 17] uses a spatial SLP, a temporal SLP, and another spatial SLP (Fig. 2(b)). The corresponding generator is available at [18].

We explain these notions more formally next. If 2^n data are streamed through 2^k ports over 2^t cycles, $n = t + k$, then the cycle during which an element arrives corresponds to the t most significant bits of its index, while the port corresponds to the k least significant bits. For instance, for $t = k = 2$, the element indexed with

$$11_b = \left(\begin{array}{c} 1 \\ 0 \\ \hline 1 \\ 1 \end{array}\right) = \left(\begin{array}{c} 2_b \\ \hline 3_b \end{array}\right)$$

arrives during the second cycle on the third port. This suggests blocking the matrix P of a linear permutation $\pi(P)$ to be streamed as

$$P = \begin{pmatrix} P_4 & P_3 \\ P_2 & P_1 \end{pmatrix}, \text{ such that } P_4 \text{ is } t \times t.$$

Namely, an element arriving in cycle c on port p is output at port p' during the cycle c', where

$$p'_b = P_1 p_b + P_2 c_b \text{ and} \tag{3}$$

$$c'_b = P_4 c_b + P_3 p_b. \tag{4}$$

Spatial and temporal SLPs can now be identified using the structure of P [8]:

Spatial SLP. A permutation $\pi(P)$ such that

$$P = \begin{pmatrix} I_t & \\ P_2 & P_1 \end{pmatrix}$$

permutes only across ports, i.e., is spatial, and can be implemented with a switching network (SNW) consisting of rank P_2 stages of 2^{k-1} 2×2-switches [16, 17]. If in addition, $P_2 = 0$, the SNW thus requires no switches and corresponds to a simple rewiring. In this case we call $\pi(P)$ steady.

Temporal SLP. If

$$P = \begin{pmatrix} P_4 & P_3 \\ & I_k \end{pmatrix},$$

then $\pi(P)$ permutes only across cycles, i.e., is temporal, and can be implemented with an array of 2^k RAM banks of 2^t words with a simple control logic [17]. Alternatively, methods based on graph coloring can reduce the size of the RAM banks needed in certain cases [12].

General SLP. A general linear permutation $\pi(P)$ can now be decomposed into three linear permutations using the algorithms of [21]:

$$\pi(P) = \pi(L \cdot C \cdot R) = \pi(L) \cdot \pi(C) \cdot \pi(R), \tag{5}$$

where the factors alternate between spatial and temporal SLPs, yielding the two possibilities in Fig. 2.

Cost and optimality. The first structure in Fig. 2(a) requires 2^{k+1} RAM banks of 2^t elements, and $\text{rank}(P_2) \cdot 2^{k-1}$ 2×2-switches. This design always minimizes the number of 2×2-switches (Theorem 1 of [17]).

The second in Fig. 2(b) uses half the number of RAM banks, 2^k, and $\max(\text{rank}(P_2), n - \text{rank}\,P_1 - \text{rank}\,P_4) \cdot 2^{k-1}$ switches, which may be larger than in the first structure. It has the optimal logic complexity for such a structure, uses the minimal number of RAM banks, and has a minimal latency, if dual-ported memory banks are used (respectively Theorem 2 of [17], Corollary 1 and Lemma 2 of [12]). If in addition [12] is used to implement the temporal SLP, then the RAM size is also minimal.

Streaming the perfect shuffle. As an example, we consider the case of the perfect shuffle permutation in (1) for $n = 4$. For $t = k = 2$ and using the SNW/RAM/SNW structure, the corresponding bit-matrix S_4 would be decomposed as

$$S_4 = \left(\begin{array}{cc|cc} 1 & & & \\ & 1 & & \\ \hline & & 1 & \\ & 1 & & 1 \end{array}\right) \cdot \left(\begin{array}{cc|cc} & 1 & & \\ 1 & & & 1 \\ \hline & & 1 & \\ & & & 1 \end{array}\right) \cdot \left(\begin{array}{cc|cc} 1 & & & \\ & 1 & & \\ \hline & & & 1 \\ 1 & & 1 & \end{array}\right), \tag{6}$$

yielding the design shown in Fig. 2(b), consisting of 4 RAM banks of 2 word and 4 switches. More generally, implementing a streaming perfect shuffle requires 2^k banks of 2^{t-1} words, and 2^k switches [16].

3 STREAMING MULTIPLE LINEAR PERMUTATIONS

The prior techniques from Section 2 are sufficient to implement the streaming permutations, and thus the streaming FFTs shown in Fig. 1(b) and 1(c). However, they cannot be used for the structure in Fig. 1(d), where the same datapath has to handle two different SLPs[2]. An immediate solution would be to use general streaming permutation methods, like [11] or [12]. They propose, respectively, a structure as in Fig. 2(a) and 2(b), but replace the specialized SNWs by complete, and thus more expensive permutation networks. In this section, we propose a method to implement in hardware a datapath capable of rearranging streaming data according to a small number of given linear permutations, thus reducing the implementation cost compared to a general solution.

Problem statement. Formally, we are given a list

$$\pi(P^{(0)}), \pi(P^{(1)}), \dots, \pi(P^{(s-1)})$$

of linear permutations, and a streaming width 2^k. Our goal is to implement an architecture that performs the permutation $\pi(P^{(i)})$ on the i^{th} dataset, streamed over 2^k ports.

The main idea first decomposes each permutation as in (5), i.e., for all $0 \leq i < s$,

$$\pi(P^{(i)}) = \pi(L^{(i)}) \cdot \pi(C^{(i)}) \cdot \pi(R^{(i)}),$$

where each factor is either temporal or spatial. The global architecture can then be implemented by a sequence of blocks that each perform either a sequence of temporal or a sequence of spatial SLPs. We now consider these two cases and describe their implementation.

3.1 Sequence of Temporal SLPs

We assume a given list of bit matrices $P^{(0)}, P^{(1)}, \dots, P^{(s-1)}$, such that all $\pi(P^{(i)})$ are temporal, i.e.,

$$P^{(i)} = \begin{pmatrix} P_4^{(i)} & P_3^{(i)} \\ & I_k \end{pmatrix}, \quad 0 \leq i < s.$$

RAM array. A structure that permutes a dataset i according to $\pi(P^{(i)})$ can be implemented using an array of 2^k dual-ported RAM banks of 2^t words. Each of these banks have a write port connected to one of the inputs of the block, and a read port connected to the corresponding output (see Fig. 3(a)). The write and read addresses ensure that data are correctly permuted (accordingly to (4)), and are respectively controlled using two t-bits timers: c, that starts when a new dataset arrives, and c', that starts when the output begins.

Latency. The output begins as early as possible, to minimize the latency, for each different permutation. Therefore, c' is triggered when c reaches the value corresponding to the maximal lifetime δ_i

[2]The direct sum, i.e. block diagonal composition of two linear permutations is in general not a linear permutation.

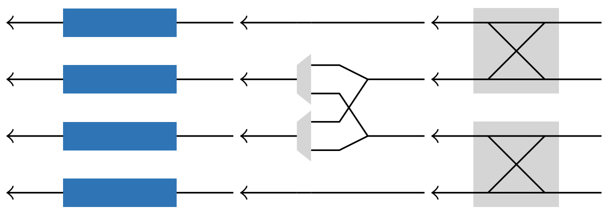


Figure 3: The basic blocks we use, here for a streaming width of $2^k = 4$. (a) can pass any temporal permutation; (b) implements the two spatial steady SLPs $\pi(I_t \oplus J_2)$ and $\pi(I_n)$; (c) implements (9).

of an element in the permutation:

$$\delta_i = \max_{p,c}(c - d(i,p,c)), \qquad \text{with } d(i,p,c)_b = P_4^{(i)} c_b + P_3^{(i)} p_b.$$

Conflict-free addressing. Besides permuting correctly the data, the read and write addresses need to ensure a conflict-free access. This means that an incoming element must not be written to a place where an element of a previous dataset has not been read yet. This issue occurs if $\delta_i < 2^t$. One solution is to use double buffering [11, 16], but this requires doubling the size of each RAM bank.

The solution we propose is to always write an element of a dataset where the same element of the previous dataset was read. Namely, for the p^{th} port, the first dataset received is written consecutively in the bank, i.e., at address c_b. It is then read at the address $(P_4^{(0)})^{-1} c'_b + (P_4^{(0)})^{-1} P_3^{(0)} p_b$, to perform the first permutation $\pi(P^{(0)})$. Then, the second dataset is written where the first dataset was read to avoid conflicts, so at the address $(P^{(0)})_4^{-1} c_b + (P^{(0)})_4^{-1} P_3^{(0)} p_b$. It is then read at the address

$$(P_4^{(0)} P_4^{(1)})^{-1} c'_b + (P_4^{(0)} P_4^{(1)})^{-1} P_3^{(1)} p_b + (P_4^{(0)})^{-1} P_3^{(0)} p_b.$$

More generally, the i^{th} dataset is written (resp. read) at the address $U_i c_b + u_{i,p}$, (resp. $U_{i+1} c'_b + u_{i+1,p}$), where U_i is such that

$$\begin{cases} U_{i+1} = U_i (P_4^{(i \bmod s)})^{-1}, \\ U_0 = I_t, \end{cases}$$

and u_i satisfies

$$\begin{cases} u_{i+1,p} = U_{i+1} P_3^{(i \bmod s)} p_b + u_{i,p}, \\ u_{0,p} = 0. \end{cases}$$

We store the values of (U_i) in a ROM, controlled by a counter. Using AND and XOR gates, the term $U_i c_b$ is computed once for all the banks. Then, this signal is XORed with $u_{i,p}$ for each bank p to obtain the write address. The number of terms of (U_i) stored in the ROM is the least that guarantees conflict-free access (this length is bounded by the period[3] of (U_i)) The read address is obtained similarly.

Alternative addressing. As remarked in [11], it is also possible to store all the addresses, for every cycle, and for every permutation in a bank. Using this technique with [12] allows to use banks of

[3]The period of (U_i) is sq, where q is the smallest positive integer such that $(P_4^{(s-1)} P_4^{(s-2)} \dots P_4^{(0)})^{-q} = I_t$.

size $\max_i \delta_i$ words. However, in our application, this value is close to 2^t due to the bit reversal.

3.2 Sequence of Spatial SLPs

We assume a given list of bit matrices $P^{(0)}, P^{(1)}, \dots, P^{(s-1)}$, such that all $\pi(P^{(i)})$ are spatial, i.e.,

$$P^{(i)} = \begin{pmatrix} I_t & \\ P_2^{(i)} & P_1^{(i)} \end{pmatrix}, \quad 0 \le i < s. \tag{7}$$

This case is somewhat more complicated. We first design solutions for two special cases from which we then build the solution for the general case.

Multiplexer array. We first consider the case where all the SLPs $\pi(P^{(0)}), \dots, \pi(P^{(s-1)})$ are steady spatial permutations, i.e., for every i,

$$P^{(i)} = I_t \oplus P_1^{(i)} = \begin{pmatrix} I_t & \\ & P_1^{(i)} \end{pmatrix}. \tag{8}$$

Since each such SLP is a different wiring, the list of those can be implemented with an array of 2^k d-input multiplexers, where d is the number of unique matrices in the list (see Fig. 3(b)).

For instance, if the list contains two different matrices $I_t \oplus A$ and $I_t \oplus B$, it is possible to implement both using a structure where each output port p is the output of a multiplexer connected to the inputs $A^{-1}p_b$ and $B^{-1}p_b$, and controlled by a counter. Of course, the multiplexers connected twice to the same input can be simplified to a simple wire, leaving an actual implementation consisting of

$$\left|\{p \mid A^{-1}p_b \neq B^{-1}p_b\}\right| = 2^k - 2^{k-\operatorname{rank}(A^{-1}+B^{-1})}$$

2-input multiplexers. If $A = B$, then $A^{-1} = B^{-1}$ and thus the sum is 0 (since addition is modulo 2), which means the implementation consists only of wires, as expected.

Switching array. We now consider another special case where, for every i, $P_1^{(i)} = I_k$ and all elements of $P_2^{(i)}$ are zero, except for its last row, which we denote with v_i^T. Formally, for every i,

$$P^{(i)} = \begin{pmatrix} I_t & & & \\ & 1 & & \\ & & \ddots & \\ v_i^T & & & 1 \end{pmatrix}. \tag{9}$$

In this case, (3) shows that $\pi(P^{(i)})$ is the permutation that exchanges each pair within a chunk of 2^k elements, every time the corresponding cycle c is such that $c_b \cdot v_{ib} = 1$.

Therefore, P can be implemented using an array of 2^{k-1} 2×2-switches. All these switches are controlled by the output of a single s-input multiplexer that chooses among the results of the scalar products $c_b \cdot v_{ib}$, for $0 \le i < s$. These scalar products are computed using XOR gates on a timer c_b.

Fig. 3(c) shows such a switching array that can implement any spatial $P^{(i)}$ in (9) for $k = 2$.

General Spatial SLP. We return now to the general case (7), which we will decompose into matrices of the form (8) and (9) to implement it with the previous structures. We first consider the matrix M of size $k \times st$ that concatenates the matrices $P_2^{(i)}$:

$$M = \begin{pmatrix} P_2^{(0)} & P_2^{(1)} & \cdots & P_2^{(s-1)} \end{pmatrix}.$$

Using Gaussian elimination, it is possible to find an invertible matrix K of size $k \times k$ such that KM has $m = \operatorname{rank} M$ non-zero rows at the top. This implies that for every i the matrix $KP_2^{(i)}$ has the form

$$KP_2^{(i)} = \begin{pmatrix} v_{1,i}^T \\ v_{2,i}^T \\ \vdots \\ v_{m,i}^T \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

where the m top rows are denoted with $v_{j,i}^T$. Note that some of these may be zero for a given i. Direct computation yields now the decomposition into the prior special cases:

$$\begin{aligned} P^{(i)} = &\begin{pmatrix} I_t & \\ & K^{-1}S_k^{k-m} \end{pmatrix} \cdot \begin{pmatrix} I_t & & & \\ & 1 & & \\ & & \ddots & \\ v_{1,i}^T & & & 1 \end{pmatrix} \begin{pmatrix} I_t & \\ & S_k \end{pmatrix} \cdot \\ &\vdots \\ &\begin{pmatrix} I_t & & & \\ & 1 & & \\ & & \ddots & \\ v_{m,i}^T & & & 1 \end{pmatrix} \begin{pmatrix} I_t & \\ & S_k \end{pmatrix} \cdot \\ &\begin{pmatrix} I_t & \\ & KP_1^{(i)} \end{pmatrix}. \end{aligned}$$

The corresponding architecture can now be read off from right to left:

(1) a multiplexer array that permutes the wires as $\pi(I_t \oplus KP_1^{(i)})$ for the ith dataset,
(2) a sequence of m switching arrays, parameterized, respectively, by $v_m, v_{m-1}, \dots, v_1$, each preceded by a perfect shuffle of the wires, and
(3) a rewiring performing the permutation $\pi(I_t \oplus K^{-1}S_k^{k-m})$.

Cost. The structure that we derived consists of $\operatorname{rank} M$ arrays of 2^{k-1} switches each, and one array of at most 2^k multiplexers.

3.3 General sequence of SLPs

Now we consider the general case of arbitrary invertible bit matrices $P^{(0)}, P^{(1)}, \dots, P^{(s-1)}$. Using [21], we get, for each i, the decomposition

$$P^{(i)} = \begin{pmatrix} I_t & \\ L^{(i)} & I_k \end{pmatrix} \begin{pmatrix} C_4^{(i)} & C_3^{(i)} \\ & C_1^{(i)} \end{pmatrix} \begin{pmatrix} I_t & \\ R^{(i)} & I_k \end{pmatrix},$$

which can be rewritten as

$$P^{(i)} = \begin{pmatrix} I_t & \\ L^{(i)} & I_k \end{pmatrix} \begin{pmatrix} C_4^{(i)} & C_3^{(i)}(C_1^{(i)})^{-1} \\ & I_k \end{pmatrix} \begin{pmatrix} I_t & \\ C_1^{(i)}R^{(i)} & C_1^{(i)} \end{pmatrix}.$$

This decomposition yields two sequences of spatial permutations, and one of temporal permutations. These can be implemented in a straightforward way using the previous structures.

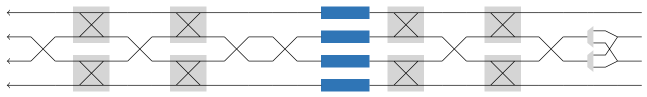


Figure 4: Datapath for the permutation block in Fig. 1(d).

Cost. The resulting architecture consists of one multiplexer array (as the leftmost sequence of spatial SLPs does not require one) containing a maximum of $2^k - 1$ multiplexers, an array of 2^k RAM banks (except in the special case where all the SLPs are spatial), and $(\operatorname{rank} M_L + \operatorname{rank} M_R) \cdot 2^{k-1}$ 2×2-switches, where

$$\begin{aligned} M_L &= \begin{pmatrix} L_2^{(0)} & L_2^{(1)} & \cdots & L_2^{(s-1)} \end{pmatrix}, \\ M_R &= \begin{pmatrix} R_2^{(0)} & R_2^{(1)} & \cdots & R_2^{(s-1)} \end{pmatrix}. \end{aligned}$$

Optimality. The number of RAM banks and the RAM latency match the bounds given in [12], and are therefore optimal. The number of switches, in the general case, depends on the different degrees of freedom appearing in the decompositions[4] [21], and no optimality can be claimed. Of course, if the sequence of SLPs only contains one unique SLP, the design we obtain only differs from [17] by rewirings, and it therefore inherits the optimality properties (Section 2).

Example: Fusing perfect shuffle and bit reversal. As an example, we design the permutation block in Fig. 1(d) capable of passing a perfect shuffle $\pi(S_4)$, and a bit reversal $\pi(J_4)$. Using the decomposition (6) for S_4, and the following (spatial/temporal/spatial) decomposition for J_4

$$J_4 = \left(\begin{array}{cc|cc} 1 & & & \\ & 1 & & \\ \hline & 1 & 1 & \\ 1 & & & 1 \end{array}\right) \cdot \left(\begin{array}{cc|cc} 1 & & & 1 \\ & 1 & 1 & \\ \hline & & 1 & \\ & & & 1 \end{array}\right) \cdot \left(\begin{array}{cc|cc} 1 & & & \\ & 1 & & \\ \hline & 1 & 1 & \\ 1 & & & 1 \end{array}\right), \tag{10}$$

we derive a datapath that consists of two blocks that performs a sequence of spatial SLPs around a block that performs a sequence of temporal SLPs. For example, this sequence of temporal SLPs contains the two middle permutations in (6) and (10):

$$\pi\left(\left(\begin{array}{cc|cc} & 1 & & \\ 1 & & & 1 \\ \hline & & 1 & \\ & & & 1 \end{array}\right)\right) \text{ and } \pi\left(\left(\begin{array}{cc|cc} 1 & & & 1 \\ & 1 & 1 & \\ \hline & & 1 & \\ & & & 1 \end{array}\right)\right).$$

The resulting implementation consists of an array of two 2-input multiplexers, two stages of two 2×2-switches each, an array of four RAM banks, and two additional stages of two 2×2-switches each (Fig. 4). Compared to an architecture performing only the bit reversal derived using [17] (Fig. 5), it requires only two additional 2-input multiplexers.

More generally, an architecture that can stream both the bit reversal and the perfect shuffle on 2^n points with a streaming width 2^k differs from a bit-reversal-only datapath with the same architecture by only $2^k - 2$ 2-input multiplexers. In other words, the additional support for the perfect shuffle is obtained almost for free.

[4]More precisely, the decomposition in [21] is optimal for each permutation taken individually, but as we compute independently these decompositions for each permutation, there is no guarantee in general that the global sequence yields an optimal rank for M_L and M_R.

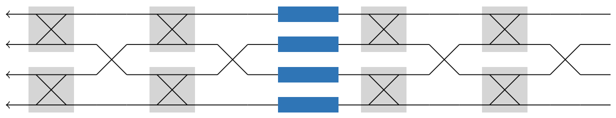


Figure 5: Datapath for a bit reversal on $2^n = 16$ elements streamed on $2^k = 4$ ports [17].

4 APPLICATION: PEASE FFT

To evaluate our fused permutation in a concrete case, we have built a generator [18] capable of producing designs as in Fig. 1(d) for Pease FFTs of arbitrary radix (Fig 1 shows the special case of radix 2). This generator takes as input the size 2^n of the FFT, the number of ports 2^k, the bit-width of the input data, and the desired radix 2^r, with $r|n$ and $r \leq k \leq n$. It outputs the corresponding design in the form of Verilog code. In this section, we briefly explain how this generator works.

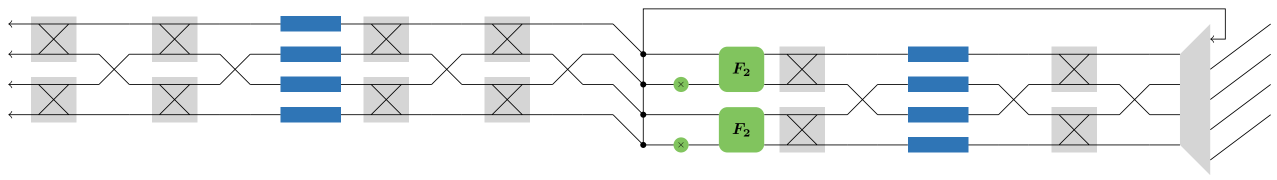
Derivation of the FFT architecture. The generator first considers a Pease FFT algorithm of the corresponding radix, and the sequence of permutations that have to be supported by the permutation block of Fig. 1(d). These are all linear, and the block is designed according to the techniques shown in Section 3. Butterflies and complex multipliers are then added to this permutation block within a loop, as in Fig. 1(d). Some optimizations occur at this time. For instance, with a radix 2, during the implementation of the leftmost spatial permutation, it is possible to choose K such that $v_{j,i} = 0$, for $i < n$ and $j < \min(t,k)$. Therefore, the $\min(t,k) - 1$ leftmost arrays of switches can safely be "unrolled," thus reducing the latency within the loop, and therefore improving the global throughput (see Fig. 6(b)). Only one stage of the leftmost spacial permutation remains in the loop.

Compared to the classic streaming reuse architecture (Fig. 6(a)), the design we obtain has an additional multiplexer stage and an additional switching array stage in the loop, but it does not have a dedicated structure to compute the bit reversal.

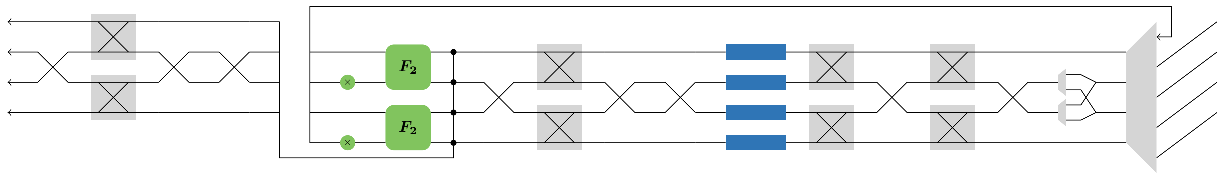
RTL graph. The design is then translated into an RTL graph, where additional optimizations are performed including the following:

- ROMs containing periodic values are simplified.
- ROMs containing a single value are replaced by a constant.
- Trivial arithmetic operations are simplified.
- A multiplexer with inputs coming from two multiplexers sharing the same inputs are fused into a single multiplexer.
- A 2-input multiplexer whose inputs come from two other multiplexers driven by the same control signal is fused to a 4-input multiplexer. This allows the efficient use of 6-input LUTs on current FPGAs.
- ROMs containing the same values are paired.

Additionally, in this step the design is pipelined and synchronized. In particular, if a control signal needs to be pipelined, the corresponding counters/timers are triggered in advance if possible. Otherwise, a reset value is computed for the registers that were added. As the design contains a loop, it must also be ensured that the



(a) Streaming iterative reuse with [17]



(b) Streaming iterative reuse with fused permutations

Figure 6: Radix-2 Pease FFT, iterative reuse with fused permutation $n = 4, k = 2$.

head of a dataset does not collide with its tail anywhere. Therefore, the design is first generated in a sandbox to measure the latency of its different parts. In a second pass, the latency of the inner temporal permutation is then increased if needed. Conversely, if the latency of the inner part of a loop is higher than the duration of the dataset, it means that the amount of time (the gap) between two datasets must be increased. This information is used in the second pass for the temporal permutation to reduce as much as possible the number of elements of (U_i) (see Section 3) stored in ROM, while ensuring conflict-free addressing in the RAM.

Once these simplifications have been performed, the design is output as Verilog code.

Limitations. Our generator was implemented with a main focus on the high level architecture and on the permutation part. It only supports fixed-point arithmetic, and the pipelining decisions are made with a basic heuristic. Using a more sophisticated approach like FloPoCo [22] for the low level implementation could reduce the area consumption of the produced designs, and add efficient support for floating point arithmetic.

Another limitation of our generator concerns the twiddle factors. In the designs we produce, each complex multiplier has a corresponding ROM that contains all the (real and imaginary) coefficients that it uses. A more distributed approach, along with a simple online computation of these coefficients could reduce further the number of BRAMs used.

5 RESULTS

In this section, we compare the cost and the performance of our generated FFT datapaths with other, state-of-the-art memory-efficient FFT architectures.

Table 1 lists the benchmarks we compare against. We consider two types of designs. The first type (A–D) is the prior iterative reuse structure from [10] exemplified in Fig. 1(c), with different solutions for the streaming permutations. The original [10] uses the permutations from [16], which is A in the table. B and C use different solutions that are not specific to linear permutations. Both, A and B are available online at [23]. D improves the permutations in [16].

The second type (E–G) is the proposed architecture exemplified in Fig. 1(d), again with different solutions for the necessary fused permutation block. E and F is what can be built with prior work that provides a general streamed permutation network. G is our proposed solution specialized to the two permutations that need to be fused. Note that neither E or F has been used within an FFT architecture as proposed here.

Table 1 analyzes the cost and performance for a radix-2 Pease FFT. We discuss these next before we show results after place-and-route.

Cost. For the memory consumption, we list the RAM requirement for the permutation part, excluding the memory used to store the twiddle factors. C theoretically should allow the use of banks of 2^{t-1} words for the perfect shuffle, but when used with the structure in Fig. 1(c), the latency of the inner loop had to be increased to avoid dataset collisions, thus requiring 2^t words for all RAM banks. The gains compared to A–E are a factor of two or four; the only competitive method is F. However, the routing cost is at least a factor of two higher, and even more for t smaller than k.

For the routing requirements, we assume that the methods B, C, E, F using complete permutation networks implement them with [24], i.e., using $(k - 1/2) \cdot 2^k$ $2{\times}2$-switches. We counted 2 multiplexers per switch, and 2^k multiplexers for the loop. Figure 7 plots the formulas in Table 1 for three different numbers of ports and a range of FFT sizes. Our method is better compared to A–D and F. Only E use less multiplexers[5] for large values of t, but requires four times more RAM banks.

In summary, we improve routing cost compared to F (and B and C) and RAM cost compared to A–E, both by at least a factor of two.

Gap. Next we analyze the minimal number of cycles between two datasets, i.e., the gap, which is the inverse of the throughput. In our case, it is constrained by the duration of the input itself (2^t cycles), and by the time the dataset stays in the loop. In Table 1, we assumed that the designs were all targeting $\approx$ 400Mhz on a Virtex 7. This

[5]Using the RAM/SNW/RAM architecture instead would have yield better routing complexity, but for twice the amount of RAM.

Ref	Architecture	Permutation	Memory	Routing (2 : 1 Mux)	Gap (cycles per transform)
A	Fig. 1(c) [10]	Püschel [16]	2^{k+1} banks of 2^{t+1} words	$(\min(t, k) + 3/2) \cdot 2^{k+1}$	$2^t + (n-1) \cdot \max(2^t, 2^{t-1} + 9)$
B	Fig. 1(c) [10]	Milder [11]	2^{k+2} banks of 2^{t+1} words	$(k - 1/4) \cdot 2^{k+2}$	$\geq 2^t + (n-1) \cdot \max(2^t, 2^{t-1} + \lceil k/2 \rceil + 8)$
C	Fig. 1(c) [10]	Koehn [12]	2^{k+1} banks of 2^t words	$(k - 3/8) \cdot 2^{k+3}$	$2^t + (n-1) \cdot \max(2^t, 2^{t-1} + 2\lceil k/2 \rceil + 9)$
D	Fig. 1(c) [10]	Serre [17]	2^{k+1} banks of 2^t words	$(\min(t, k) + 3/2) \cdot 2^{k+1}$	$2^t + (n-1) \cdot \max(2^t, 2^{t-1} + 9)$
E	Fig. 1(d) (novel)	Milder [11]	2^{k+1} banks of 2^{t+1} words	$k \cdot 2^{k+1}$	$\geq 2^t + n \cdot \max(2^t, 2^{t-1} + \lceil k/2 \rceil + 8)$
F	Fig. 1(d) (novel)	Koehn [12]	2^k banks of 2^t words	$(k - 1/4) \cdot 2^{k+2}$	$2^t + n \cdot \max(2^t, 2^{t-1} + 2\lceil k/2 \rceil + 9)$
G	Fig. 1(d) (novel)	Proposed	2^k banks of 2^t words	$(\min(t, k) + 1) \cdot 2^{k+1} - 2$	$2^t + n \cdot \max(2^t, 2^{t-1} + \lceil \min(t, k)/2 \rceil + 8)$

Table 1: Comparison of different architectures using different permutation methods, for a radix-2 Pease FFT, for $k > 1$.

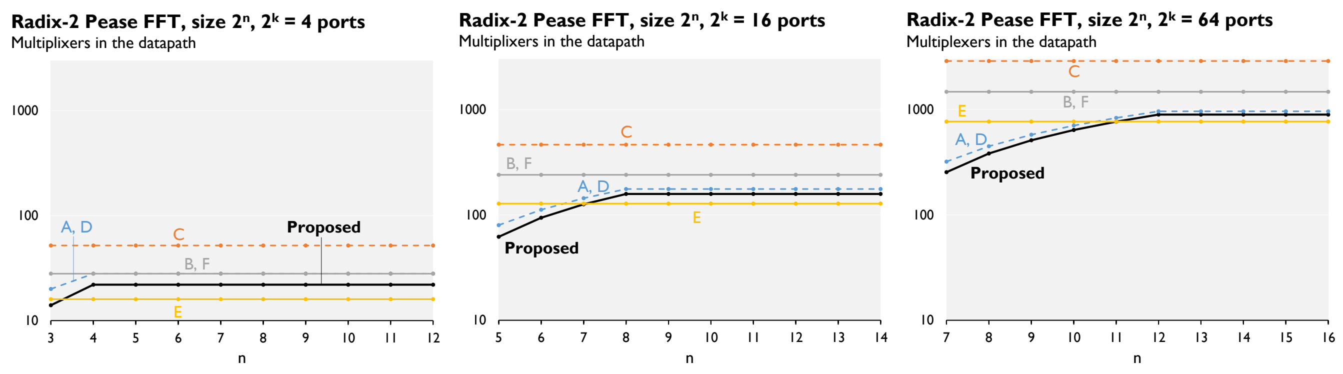


Figure 7: Number of 2-input multiplexers in the datapath of a radix-2 Pease FFT, for different streaming widths. Lower is better.

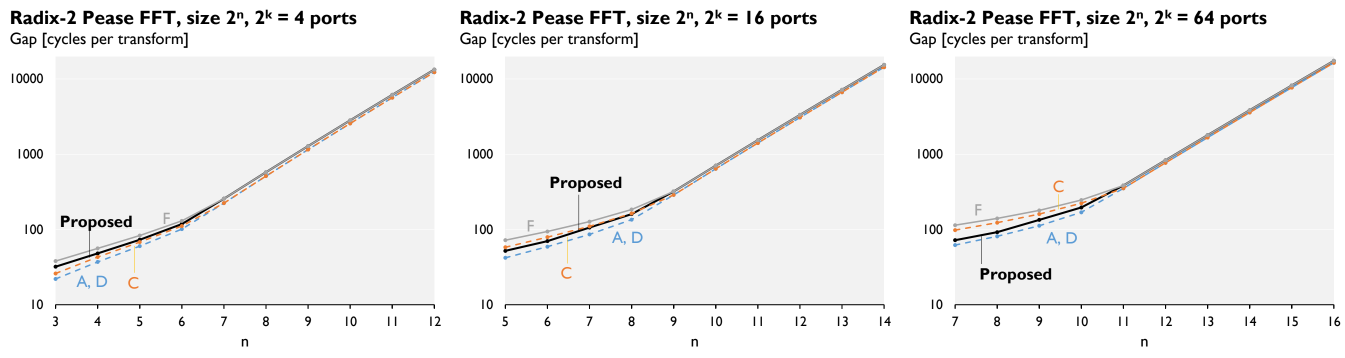


Figure 8: Gap of a radix-2 Pease FFT in number of cycles between two transforms, for different streaming widths. Lower is better.

requires a 4 cycles pipelining for the arithmetic part (butterfly and multiplications), and one register every 2 multiplexers (a complete permutation network has therefore a latency of $\lceil k/2 \rceil + 1$ cycles). Additionally, we assumed that all temporal permutations (even when fused) were done using the minimal possible latency; a feature that can easily be obtained using dual-ported RAM. However, with [11] (B and E), the total "temporal latency" depends on the chosen decomposition, and we can therefore only provide a lower bound. The corresponding formulas in Table 1 are plotted in Fig 8. It appears that, for $n - k = t \geq 5$, the latency required to avoid two datasets overlapping in the loop dominates the intrinsic inner latency of the loop. Thus, the term 2^t becomes the dominant term in the max, and the gap becomes $n \cdot 2^t$ for all streaming iterative reuse architectures (A–D), and $(n + 1) \cdot 2^t$ for the fused permutation structure (E–G). Thus, as n, and hence t, increases the gaps of the different solutions converge. The same is then also true for the throughputs.

Results after place and route. Among the prior FFT solutions only A ([10]) is available online at [23]. We compare these designs

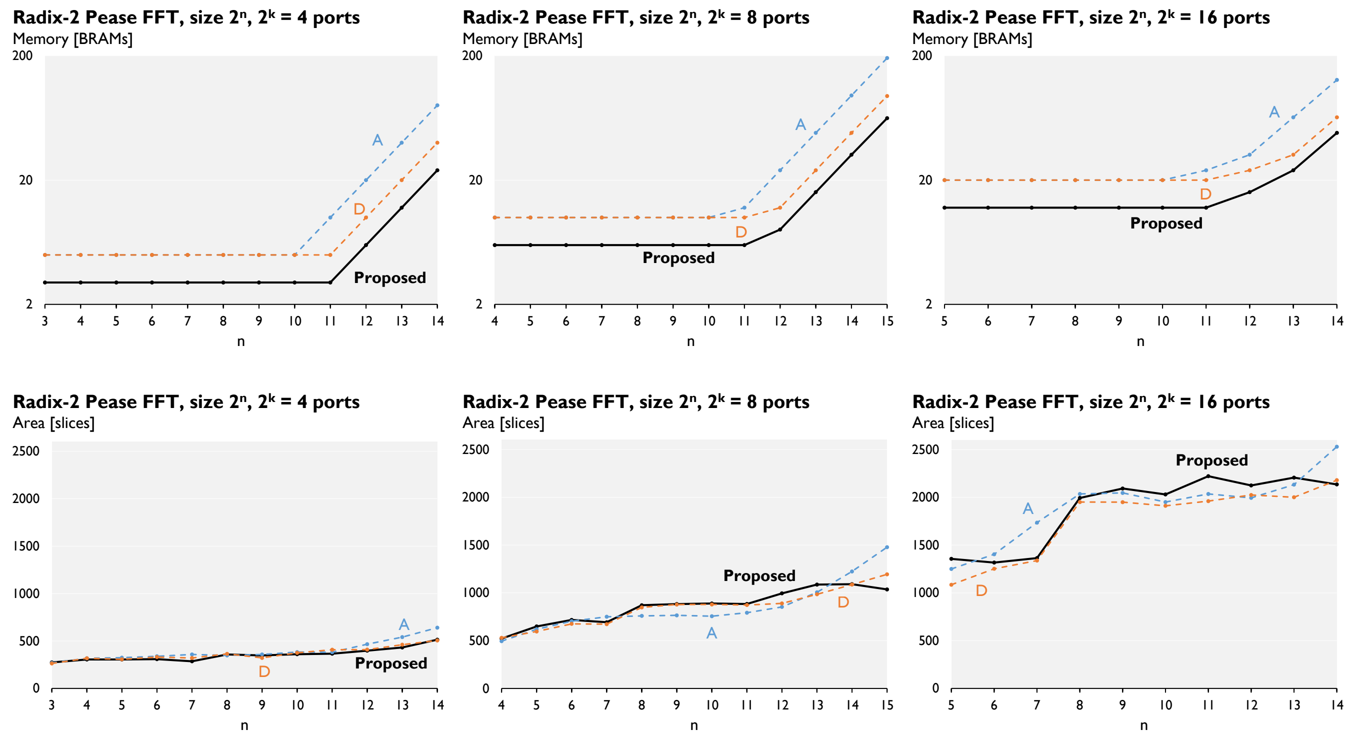


Figure 9: Resources used by a radix-2 Pease FFT. Lower is better.

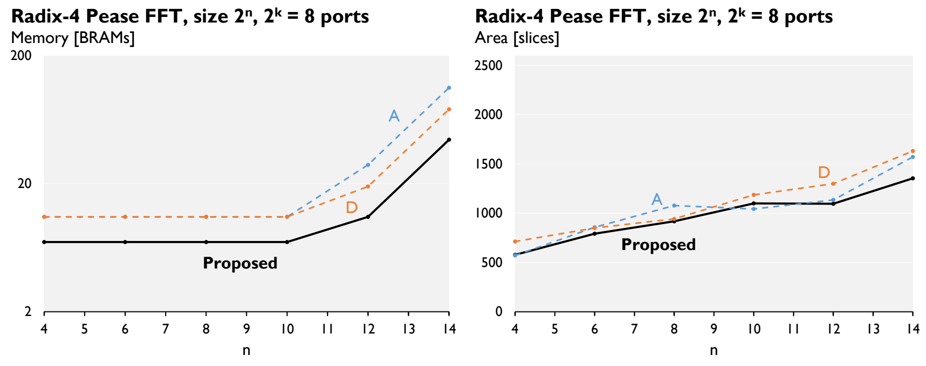


Figure 10: Resources used by a radix-4 Pease FFT. Lower is better.

with our solution G after place-and-route. For completeness we also implemented a variant of our generator [18] that produces the FFTs in D, which reduces the RAM cost of A.

The area and the RAM consumption of different designs for a radix-2 FFT are shown in Fig. 9, after place and route on a Xilinx Virtex 7 xc7vx1140 using Vivado 2014.4, using an element size of 16 bits. We observe that, as the number of RAM bank does not depend on n in the considered designs, the memory consumption stays constant until the capacity of the BRAM is reached. As expected from Table 1 our design requires fewer BRAMs; since the twiddle factors are stored in BRAMs as well, the RAM usage is not exactly halved. The logic area is roughly comparable and includes the control, which was not included in Table 1. While our control logic is arguably more efficient than storing all the switch configuration and addresses for all cycles, it is more complex than the one used in A or D, which explains why we do not require fewer slices.

Fig. 10 shows some results for a radix-4 FFT, which slightly reduces the number of multiplications needed but can only be folded at the granularity of DFT_4 blocks, which themselves are implemented using four butterflies. The overall behavior and comparison is analogous to the radix-2 case.

Discussion. Because our main target is FPGA, which contains BRAM modules, we compared our work with other RAM-based permutation techniques. However, other approaches based on registers [13] or distributed buffers [15] could be beneficial on platforms where grouping several memory elements does not improve the cost (ASICs).

Hardware architectures to compute DFTs are a classic topic in the literature, and other approaches that also use a RAM capacity equal to the size of the dataset (2^n) exist. However, these works are based on in-place algorithms [25], or consist of parameterized architectures [26] that do not provide the same flexibility as a generated streamed architecture (for instance, the number of ports is constrained by the radix used in the algorithm).

ACKNOWLEDGMENTS

The authors would like to thank Peter A. Milder for helpful discussions, and the anonymous reviewers for their suggestions.

6 CONCLUSIONS

We proposed a novel method to design a datapath capable of realizing a number of fixed streamed linear permutations. As main application we proposed a new variant of a folded Pease FFT that requires only one permutation block for both, the internal shuffles and the final bit reversal. While in some FFT applications, the bit reversal can be omitted, in many others it cannot, e.g., if frequency components need to be processed in order from low to high. For those, our new architecture offers novel Pareto-optimal tradeoffs between performance and logic/memory cost across an entire design space of FFTs given by the chosen radix and number of input ports. These should directly translate to increased energy efficiency for a wide range of resource-constrained embedded applications.

REFERENCES

[1] D. Cohen, "Simplified control of FFT hardware," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 24, no. 6, pp. 577–579, 1976.

[2] P. Kumhom, J. R. Johnson, and P. Nagvajara, "Design, optimization, and implementation of a universal FFT processor," in *Proc. International ASIC/SOC Conference (ASIC)*, pp. 182–186, 2000.

[3] A. Cortés, I. Vélez, and J. F. Sevillano, "Radix r^k FFTs: Matricial representation and SDC/SDF pipeline implementation," *IEEE Transactions on Signal Processing*, vol. 57, no. 7, pp. 2824–2839, 2009.

[4] S. He and M. Torkelson, "A new approach to pipeline FFT processor," in *Proc. Parallel Processing Symposium (IPPS)*, pp. 766–770, 1996.

[5] B. Akin, F. Franchetti, and J. C. Hoe, "FFTs with near-optimal memory access through block data layouts: Algorithm, architecture and design automation," *Journal of Signal Processing Systems*, vol. 85, no. 1, pp. 67–82, 2015.

[6] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *Journal of the ACM*, vol. 15, no. 2, pp. 252–264, 1968.

[7] J. H. Takala, T. S. Jarvinen, and H. T. Sorokin, "Conflict-free parallel memory access scheme for FFT processors," in *Proc. International Symposium on Circuits and Systems (ISCAS)*, vol. 4, pp. 524–527, 2003.

[8] K. J. Page, J. F. Arrigo, and P. M. Chau, "Reconfigurable-hardware-based digital signal processing for wireless communications," in *Advanced Signal Processing Algorithms, Architectures and Implementations* (P. SPIE, ed.), vol. 3162, pp. 529–540, 1997.

[9] G. Nordin, P. A. Milder, J. C. Hoe, and M. Püschel, "Automatic generation of customized discrete Fourier transform IPs," in *Proc. Design Automation Conference (DAC)*, pp. 471–474, 2005.

[10] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 2, pp. 15:1–15:33, 2012.

[11] P. A. Milder, J. C. Hoe, and M. Püschel, "Automatic generation of streaming datapaths for arbitrary fixed permutations," in *Proc. Design, Automation and Test in Europe (DATE)*, pp. 1118–1123, 2009.

[12] T. Koehn and P. Athanas, "Arbitrary streaming permutations with minimum memory and latency," in *Proc. International Conference on Computer-Aided Design (ICCAD)*, pp. 1–6, 2016.

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

[14] K. J. Page and P. M. Chau, "Folding large regular computational graphs onto smaller processor arrays," in *Advanced Signal Processing Algorithms, Architectures and Implementations* (P. SPIE, ed.), vol. 2846, pp. 383–394, 1996.

[15] T. Järvinen, P. Salmela, H. Sorokin, and J. Takala, "Stride permutation networks for array processors," in *Proc. International Conference on Application-Specific Systems, Architectures and Processors Proceedings (ASAP)*, pp. 376–386, 2004.

[16] M. Püschel, P. A. Milder, and J. C. Hoe, "Permuting streaming data using RAMs," *Journal of the ACM*, vol. 56, no. 2, pp. 10:1–10:34, 2009.

[17] F. Serre, T. Holenstein, and M. Püschel, "Optimal circuits for streamed linear permutations using RAM," in *Proc. International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 215–223, 2016.

[18] F. Serre, "DFT and streamed linear permutation generator for hardware." https://acl.inf.ethz.ch/research/hardware/, 2018.

[19] M. C. Pease, "The indirect binary n-cube microprocessor array," *IEEE Transactions on Computers*, vol. 26, no. 5, pp. 458–473, 1977.

[20] J. Lenfant and S. Tahé, "Permuting data with the Omega network," *Acta Informatica*, vol. 21, no. 6, pp. 629–641, 1985.

[21] F. Serre and M. Püschel, "Generalizing block LU factorization: A lower-upper-lower block triangular decomposition with minimal off-diagonal ranks," *Linear Algebra and its Applications*, vol. 509, pp. 114–142, 2016.

[22] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[23] P. A. Milder, "Spiral DFT/FFT IP core generator." http://www.spiral.net/hardware/dftgen.html, 2008.

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

[25] B. G. Jo and M. H. Sunwoo, "New continuous-flow mixed-radix (CFMR) FFT processor using novel in-place strategy," *IEEE Transactions on Circuits and Systems I (TCAS-I)*, vol. 52, no. 5, pp. 911–919, 2005.

[26] M. Garrido, M. Ángel Sánchez, M. L. López-Vallejo, and J. Grajal, "A 4096-point radix-4 memory-based FFT using DSP slices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 1, pp. 375–379, 2017.