DSL-Based Hardware Generation with Scala: Example Fast Fourier Transforms and Sorting Networks

FRANÇOIS SERRE and MARKUS PÜSCHEL, Department of Computer Science, ETH Zurich

We present a hardware generator for computations with regular structure including the fast Fourier transform (FFT), sorting networks, and others. The input of the generator is a high-level description of the algorithm; the output is a token-based, synchronized design in the form of RTL-Verilog. Building on prior work, the generator uses several layers of domain-specific languages (DSLs) to represent and optimize at different levels of abstraction to produce a RAM- and area-efficient hardware implementation. Two of these layers and DSLs are novel. The first one allows the use and domain-specific optimization of state-of-the-art streaming permutations. The second DSL enables the automatic pipelining of a streaming hardware dataflow and the synchronization of its data-independent control signals. The generator including the DSLs are implemented in Scala, leveraging its type system, and uses concepts from lightweight modular staging (LMS) to handle the constraints of streaming hardware. Particularly, these concepts offer genericity over hardware number representation, including seamless switching between fixed-point arithmetic and FloPoCo generated IEEE floating-point operators, while ensuring type-safety. We show benchmarks of generated FFTs, sorting networks, and Walsh-Hadamard transforms that outperform prior generators.

CCS Concepts: • **Hardware → Digital signal processing**; *Application specific integrated circuits*; *High-level and register-transfer level synthesis*;

Additional Key Words and Phrases: Fast Fourier transform, sorting network, Walsh-Hadamard transform, IP core, streaming datapaths, hardware generation, Scala

ACM Reference format:
François Serre and Markus Püschel. 2019. DSL-Based Hardware Generation with Scala: Example Fast Fourier Transforms and Sorting Networks. *ACM Trans. Reconfigurable Technol. Syst.* 13, 1, Article 1 (December 2019), 23 pages.
https://doi.org/10.1145/3359754

1 INTRODUCTION

Many algorithms used in hardware applications in signal processing, communication, and other domains share a common structure consisting of a network of small processing elements. The most prominent example is the so-called butterfly network implementing a fast Fourier transform (FFT). It consists of stages of parallel and almost identical blocks (the butterflies) that operate on two inputs with data permutations in between (see Figure 1(a)). Many other important functions have similar algorithms, with different blocks and different intermittent permutations. Examples

Authors' addresses: F. Serre and M. Püschel, Department of Computer Science, ETH Zurich, Universitätstr. 6, 8092 Zurich, Switzerland; emails: {serref, pueschel}@inf.ethz.ch.

1936-7406/2019/12-ART1 $15.00
https://doi.org/10.1145/3359754

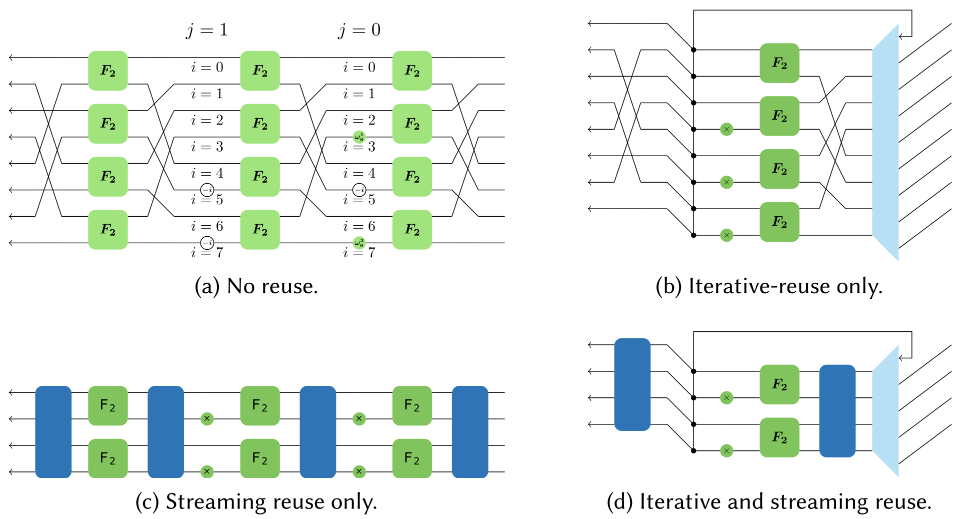


Fig. 1. Radix-2 Pease-FFT datapaths (each from right to left) operating on $2^n = 8$ elements with different types of folding [17]. In panel (a), the design is not folded and consists of three stages (each comprising a perfect-shuffle permutation, an array of butterflies F_2 and an element-wise multiplication by constants), followed by a bit-reversal permutation. Panel (b) is horizontally folded: it implements only one instance of this stage that processes the dataset iteratively. Panel (c) is vertically folded: the dataset is input *streamed* in chunks of $2^k = 4$ elements (the *streaming width*) that enter during $2^t = 2$ consecutive cycles, where $n = t + k$. Panel (d) combines both types of folding.

include the Walsh-Hadamard transform (WHT) [1], fast cosine and sine transforms [2], sorting networks (SNs) [3, 4], and permutation networks [5–10].

The regular structure offers much flexibility in their hardware implementations and thus there has been extensive work, most focusing on the FFT [11–20]. In particular, References [17, 20–22] propose generators for FFTs and sorting networks that are capable of producing an entire design space of implementations with different trade-offs in performance and resource consumption. These generators are built as a back-end of Spiral, a generator of signal processing libraries tuned for a specific platform [23], and operate with different algorithms represented in a domain specific language (DSL) called SPL. They exploit different symmetries (or regularities) of these algorithms to *fold* them temporally (*iterative reuse*, a given dataset is processed several times by the same hardware components, as in Figure 1(b)), or spatially (*streaming reuse*; a given dataset is split into chunks that are processed over several cycles, as in Figure 1(c)), or both (Figure 1(d)) to obtain a space of relevant designs. The desired design is then output as RTL-Verilog.

However, the state of the art of the different components needed in these algorithms continues to improve. As examples, FloPoCo [24] provides an open-source generator for pipelined floating-point arithmetic with arbitrary precision, streaming implementations of linear permutations (represented as dark blue boxes in Figure 1) now achieve proven optimality in terms of latency, routing complexity (number of multiplexers used) and memory used (total memory capacity and number of RAM banks) [25–27], and a new architecture (Figure 2) further reduces RAM usage in some cases by fusing permutations [28]. However, no generator to date combines these features with the flexibility offered by References [17, 22]. One possible cause is the difficulty of programming a generator capable of mapping a high-level design (as in Figures 1 and 2) to a concrete RTL implementation. Some of the challenges are discussed next.

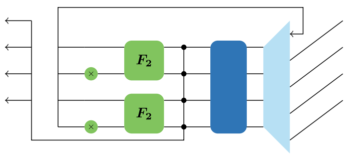


Fig. 2. Radix-2 Pease-FFT dataflow operating with fused permutations.

Mismatch of hardware and software datatypes. A first difficulty, common among high-level synthesis (HLS) tools, is the wide diversity of possible datatypes that hardware designs offer. The precision of (unsigned or signed) integers or fixed point numbers is arbitrary, in contrast to a small set of choices in software. The same applies to floating-point arithmetic, ranging from IEEE754 to the space covered by FloPoCo [24], which offers variable mantissa and exponent width.

Two different evaluation times. A second issue is that a given function may need to be either evaluated during design generation or implemented in the resulting design, or even partially evaluated during generation and partially implemented.

For instance, the FFT involves multiplications with a set of constants, called *twiddle factors*. A twiddle factor $t_{i,j}$ is a complex number that depends on two parameters: the index i of the element and the index of the computation stage j (see Figure 1(a)). In the case of non-iterative designs (Figures 1(a) and 1(c)), the parameter j is known at generation time, while in iterative scenarios (Figures 1(b) and 1(d)), the design would need to implement a *counter* counting the number of datasets that were already processed by the stage. Similarly, the parameter i is known at generation-time for non-streaming designs (Figures 1(a) and 1(b)) for each different multiplier, while in streaming designs (Figures 1(c) and 1(d)), i depends on the multiplier position, and on a *timer* that counts the number of cycles elapsed since the dataset began to enter. As the computation of a twiddle factor would typically involve a ROM containing different possible values, it is essential to exploit during generation as much as possible the structure of i and j to reduce ROM consumption and DSP slices in case of trivial multiplications.

A typical solution for handling this problem consists of writing and maintaining different versions for each different scenario, which is error-prone and time consuming.

Synchronization issues. The design requires pipelining to handle the frequency required by the user. Keeping the example of twiddle factors, an inspection of different FFT algorithms shows that many constants are 1, i ($= \sqrt{-1}$) or $-$i, which results in a trivial multiplication that does not require pipelining. However, it is necessary in this case to add supplementary registers if another non-trivial multiplication exists, to keep the whole dataset synchronized.

Additionally, if the twiddle factor computation is done in hardware, then it may also require pipelining. As this computation is independent of the input to the FFT, it is possible to initiate it in advance to avoid impacting the global latency of the design. However, this requires precise cycle tracking to trigger the counter and the timer at the appropriate time.

Handling the latency. As some of the designs produced use a loop (Figures 1(b), 1(d), and 2), special attention must be paid to guarantee that the latency of the inner structure is long enough to avoid collision between the tail and the head of a given dataset. Additionally, this inner latency determines the minimal time separating two datasets, which must be reported to the user.

Contributions. We address the above problems with a novel hardware generator for algorithms with a regular network structure. This self-contained generator employs a systematic design leveraging features provided by Scala, a modern multi-paradigm language:

- We present a hardware generator for a design space of FFTs, WHTs, and variants of Batcher-SNs. This generator is implemented in Scala [29] and leverages Scala's facilities for

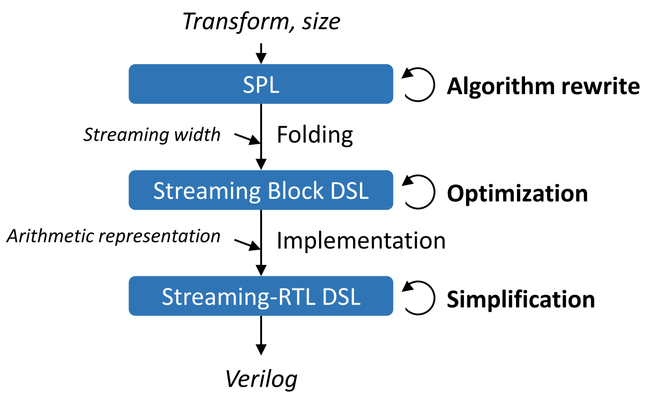


Fig. 3. The different layers of our generator.

embedding DSLs, concepts from lightweight modular staging (LMS) [30] to perform optimization at the DSL levels,and Scala's type system to offer the flexibility and modularity discussed above. While we focus on FFTs, WHTs, and SNs, the generator is extensible to other algorithms with regular structure including the examples mentioned before.

- In the generator, we use two novel DSLs to facilitate streaming optimizations.
- We benchmark against prior generators, and show significant improvements in flexibility, while keeping a comparable performance/resource trade-off.
- We provide access to our generator through a web interface at Reference [31], and provide source code at Reference [32].

This article extends our preliminary work presented in Reference [33], which only supported FFTs. Furthermore, we provide a detailed description of an additional layer of our generator, the Streaming-block DSL (Section 4), and generally a much more detailed description of the actual implementation.

2 GENERATION PIPELINE

In this article, we will refer to all functions that our generator implements as transforms. These include the discrete Fourier transform, the Walsh-Hadamard transform, and sorting networks. Our proposed generator receives as input the desired transform, its size (a power of two), and some parameters that control the design space (e.g., streaming width, iterative reuse applied or not, hardware arithmetic representation). The output is the corresponding design in the form of RTL Verilog. The generation process consists of the three layers pictured in Figure 3. Each of these layers employs a DSL to represent, manipulate, and optimize the algorithm at different levels of abstraction. Each DSL is implemented as embedded DSL inside Scala, and staging is used to allow manipulation. We first give a brief overview and then discuss the last two layers in greater detail in subsequent sections.

2.1 SPL

The first step for generating a hardware implementation consists of choosing a suitable algorithm. Following Reference [17], we represent these algorithms as *breakdown rules* that decompose a large transformation into smaller ones. These rules are represented using SPL, a mathematical language that represents linear algebra operations by matrices and operators on these matrices [23, 34, 35].

Table 1. SPL Operators Used in Our Generator

Operator	Description
First-order operator	
DFT_{2^n}	Discrete Fourier transform for input size 2^n
WHT_{2^n}	Walsh-Hadamard transform for input size 2^n
SN_{2^n}	Sorting network for 2^n inputs
$\pi(P)$	Linear permutation associated with the invertible bit-matrix P [25]
T_i, T'_i	Twiddle factors
X_2^c	Configurable two-input sorter
Higher-order operator	
$A \cdot B$	Composition of operators A and B
$\prod_i A_i$	Enumerated composition of operators A_i
$\bigoplus_i A_i$	Enumerated direct sum (parallel composition) of operators A_i

We introduce next SPL, and describe the rules we use for the different transforms we consider. Our implementation of this DSL in Scala is similar as in Reference [36], and includes the operators in Table 1. Higher-order operators are used to recursively construct algorithms from first-order operators.

DFT. Computing a *discrete Fourier transform* (DFT) of a discrete signal of 2^n elements $x = (x_i)_{0 \le i < 2^n} \in \mathbb{C}^{2^n}$ amounts to multiplying it with a matrix DFT_{2^n}:

$$y = \mathrm{DFT}_{2^n} \cdot x, \text{ where } \mathrm{DFT}_{2^n} = [\omega^{ij}]_{0 \le i,j < 2^n}, \text{ with } \omega = e^{-2i\pi/2^n}.$$

In SPL, a DFT on 2^n points is represented by the corresponding $2^n \times 2^n$ matrix DFT_{2^n}.

An FFT algorithm as the one used in Figure 1, the constant-geometry radix-2^r Pease FFT [37], corresponds to rewriting this matrix as the following product of sparse matrices:

$$\mathrm{DFT}_{2^n} = \pi\left(J_n^r\right) \cdot \prod_{\ell=0}^{n/r-1} \left(T_{n/r-\ell-1} \cdot \left(\bigoplus_{i=1}^{2^{n-r}} \mathrm{DFT}_{2^r} \right) \cdot \pi\left(S_n^r\right) \right). \quad (1)$$

The factors in the iterative product in Equation (1) correspond to the stages in Figure 1(a). In each step there is first a permutation $\pi(S_n^r)$ (the stride-by-2^r permutation), followed by parallel butterflies $\bigoplus \mathrm{DFT}_{2^r}$, followed by twiddle factors T_i. At the end is the radix-2^r-reversal permutation $\pi(J_n^r)$. The product of matrices corresponds to the composition of the corresponding steps.

The permutations are denoted with $\pi(P)$, which indicates that they are *linear*, i.e., that they map linearly the binary representation of their indices (P is the matrix that represents this linear mapping) [25, 38]. $\pi(J_n^r)$ and $\pi(S_n^r)$ are, respectively, the radix-2^r-reversal, and the stride-by-2^r permutation). T_ℓ is a diagonal matrix that performs element-wise complex multiplications with the twiddle-factors,[1]

$\prod$ and $\bigoplus$ are, respectively, the enumerated product and direct sum:

$$\prod_{i=0}^{n-1} M_i = M_0 \cdot M_1 \dots M_{n-1}, \text{ and } \bigoplus_{i=0}^{n-1} M_i = \begin{pmatrix} M_0 & & & \\ & M_1 & & \\ & & \ddots & \\ & & & M_{n-1} \end{pmatrix}.$$

[1] $T_\ell = \bigoplus_{i=0}^{2^{n-r}-1} \bigoplus_{j=0}^{2^r-1} \omega^{j \cdot i_{(r\ell)}}$, where ω is the principal 2^n-th root of unity, and $i_{(r\ell)}$ means that the $r\ell$ least significant bits of i have been set to 0.

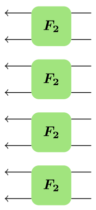


Fig. 4. A column of four parallel butterflies.

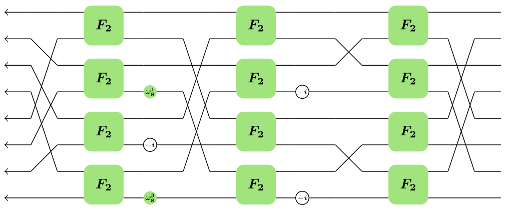


Fig. 5. Radix-2 Cooley-Tukey FFT datapaths operating on $2^n = 8$ elements. This algorithm is used when iterative reuse is not enabled, as the permutations involved require less resources when streamed.

For example, $\bigoplus_{i=1}^{2^{n-r}} \mathrm{DFT}_{2^r}$ represents 2^{n-r} parallel DFTs of size 2^r each. A column of four parallel butterflies, as in Figure 4, is therefore represented by

$$\bigoplus_{i=1}^{4} \mathrm{DFT}_2\,, \text{ where } \mathrm{DFT}_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

As only the twiddle factors depend on ℓ, i.e., change in the iterative product in Equation (1), the Pease FFT algorithm is well suited for iterative reuse. However, for designs with streaming reuse only, or for the generation of the base case 2^r-FFT, the radix-2^r Cooley-Tukey FFT (see Figure 5) is used instead, as the permutations it requires use less resources than for a Pease FFT:

$$\mathrm{DFT}_{2^n} = \pi\left(S_n^{n-r}\right) \cdot \left(\prod_{\ell=0}^{n/r-1} \left(\bigoplus_{i=1}^{2^{n-r}} \mathrm{DFT}_{2^r}\right) \cdot T'_\ell \cdot \pi(Q_\ell)\right) \cdot \pi\left(J_n^r\right), \tag{2}$$

where T'_ℓ is a diagonal matrix, and $\pi(Q_\ell)$ a permutation.

WHT. The algorithms we are using to compute a WHT are similar to the one used for DFTs, but do not include twiddle factors nor a final bit-reversal permutation. As an example, the Pease-like WHT algorithm is expressed as

$$\mathrm{WHT}_{2^n} = \prod_{j=0}^{n-1} \left(\left(\bigoplus_{i=1}^{2^{n-1}} \mathrm{DFT}_2\right) \cdot \pi(S_n)\right). \tag{3}$$

SN. Sorting networks (SNs) are somewhat similar to FFTs or WHTs but require a different form of butterflies, which are two-input sorters and thus nonlinear. Thus an extension to SPL is required as described in Reference [22] following concepts from Reference [39]. Formally, a two-input sorter is described as X_2^c: if $c = 0$, it sorts the two inputs in ascending order, if $c = 1$ it sorts them in descending order. With this, we can express SNs using the previous formalism. For streaming reuse only, we use a Batcher bitonic SN [3] (see Figure 6):

$$\mathrm{SN}_{2^n} = \prod_{j=0}^{n-2} \left(\left(\bigoplus_{i=0}^{2^{n-1}-1} X_2^0\right) \cdot \prod_{\ell=0}^{n-i-2} \left(\pi(P_\ell) \cdot \left(\bigoplus_{i=0}^{2^{n-1}-1} X_2^0\right)\right) \cdot \pi(Q_j)\right) \cdot \bigoplus_{i=0}^{2^{n-1}-1} X_2^0. \tag{4}$$

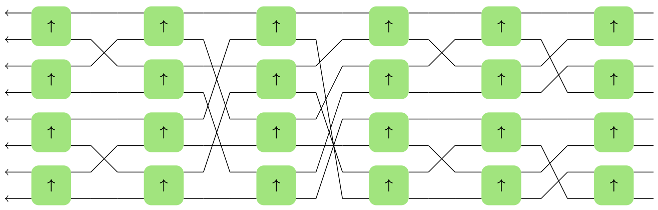


Fig. 6. Batcher bitonic sorting network [3] operating on $2^n = 8$ elements. This design corresponds to the SN1 architecture in Reference [22].

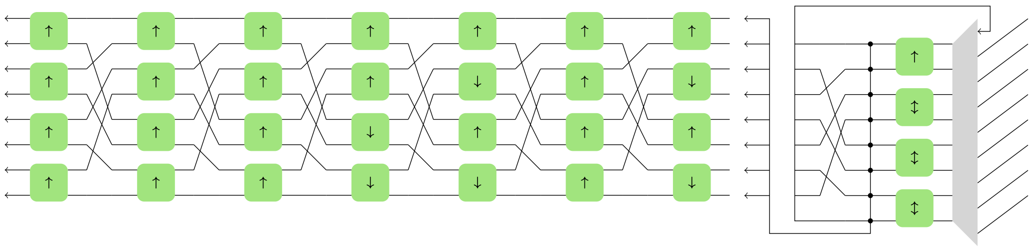


Fig. 7. Constant-geometry sorting network [4] operating on $2^n = 8$ elements. This design loosely corresponds to the SN5 architecture in Reference [22].

This corresponds to the architecture SN1 in Reference [22].

When iterative reuse is desired, a Pease-like network is used [4]:

$$\mathrm{SN}_{2^n} = \left(\bigoplus_{i=1}^{2^{n-1}} X_2^0\right) \cdot \prod_{j=0}^{n^2-n-3} \left(\pi(S_n) \cdot \left(\bigoplus_{i=1}^{2^{n-1}} X_2^{f(i,j)}\right)\right), \text{ where } f \text{ is a binary function.} \quad (5)$$

This second algorithm corresponds to the architecture SN5 of Reference [22], with the following improvements:

- The first $n-1$ stages are removed, as these correspond to a fixed permutation of the inputs, for which the order does not matter. This allows an increase of the throughput of the implementations.
- In Reference [22], X_2^c had an additional pass-through configuration. We change the stages that used this mode such that the sorters perform useless comparisons instead (by copying the configuration of the stage located n places later). Therefore, we only use X_2^c as a sorter or as an inverted sorter, thus reducing the complexity of the implementation.
- When folded for iterative reuse, a loop with an early termination (similar to the structure used in Reference [28] when fusing permutations) allows to simultaneously implement a single stage of sorters while performing only the necessary number of permutations (see Figure 7(b)).

2.2 Streaming-block DSL

In the second step of the generator (Figure 3), the SPL expression is formally folded according to the *streaming width*, i.e., the number of elements of the dataset that the design would be able to handle in each cycle. The DSL used thus expands SPL to include the streaming width (similar to the so-called Hardware-SPL in Reference [17]) but also to include the streaming blocks

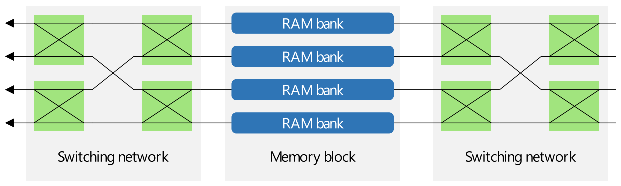


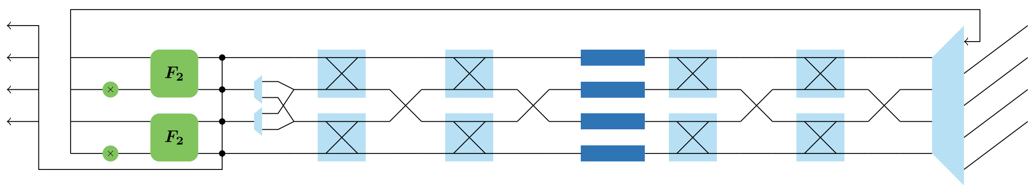
Fig. 8. An implementation of a linear streaming permutation with a streaming width of 4 using the method in Reference [25]. It consists of one stage of RAM banks and two blocks of two stages of switches each.

Table 2. Streaming Blocks Used in the Streaming-block DSL

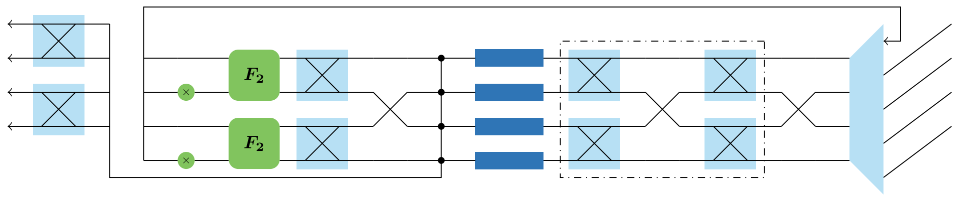
Operator	Description	SPL correspondence
First-order operator		
$\bigoplus \mathrm{DFT}_2$	Butterfly array (add and substract its two inputs)	$\bigoplus \mathrm{DFT}_2$
T_i, T_i'	Twiddle factors	T_i, T_i'
X_2^c	Configurable two-input sorter	X_2^c
$\pi_i(P_0, \dots, P_\ell)$	Array of multiplexers	$\pi \begin{pmatrix} I_t & \\ & P_i \end{pmatrix}$
$\sigma_i(v_0, \dots, v_\ell)$	Single array of switches	$\pi \begin{pmatrix} I_t & & & \\ & 1 & & \\ & & \ddots & \\ v_i^T & & & 1 \end{pmatrix}$
$\sigma_i'((u_0, v_0), \dots, (u_\ell, v_\ell))$	Double array of switches	$\pi \begin{pmatrix} I_t & & & & \\ & & 1 & & \\ & & & \ddots & \\ v_i^T & & & & 1 \\ u_i^T & 1 & & & \end{pmatrix}$
$\tau_i((A_0, B_0), \dots, (A_\ell, B_\ell))$	Array of RAM banks	$\pi \begin{pmatrix} A_i & B_i \\ & I_k \end{pmatrix}$
Higher-order operator		
$A_0 \cdot A_2 \cdots A_\ell$	Composition (without iterative reuse)	$\prod_{i=0}^{\ell} A_i$
$\prod_i A_i$	Composition with iterative reuse (A is implemented only once)	$\prod_{i=0}^{\ell} A_i$
$\prod_i^\ell (\overline{A_i} B_i)$	Composition with iterative reuse and early termination	$B_\ell \prod_{i=0}^{\ell-1} (A_i B_i)$

needed to represent the necessary datapaths for the streaming permutations from References [25, 28] (see Figure 8 and Table 2). These consist of arrays of multiplexers, switches and RAM banks. Additionally, higher order operators reflect the use of iterative reuse, or iterative reuse with early termination.

During this stage, a set of rewriting rules is used to simplify the streaming blocks, particularly in the case of a fused permutation. As an example, a radix-2 Pease DFT on eight elements (Equation (1)), folded with a streaming width of four ports using iterative reuse with fused permutation



(a) Before optimization.



(b) After optimization.

Fig. 9. Design of Figure 2 expressed using the streaming block DSL. The necessary streaming permutation is expanded into switches and RAM banks (dark blue rectangles). The optimization here "unrolls" some parts of the permutation to remove an array of multiplexer. Additionally, two arrays of switches were grouped for later mapping to 4-to-1 multiplexers. These optimizations increase the throughput and reduce the area of the final design.

would be represented in the streaming block DSL by

$$\prod_{i=0}^{3}\Bigg(\overline{T_{2-i}\cdot\bigoplus_{\ell=1}^{4}DFT_2}\cdot\pi_i(I_2,S_2,S_2,S_2)\cdot\sigma_i((1),(0),(0),(0))\cdot\pi(S_2)\cdot\sigma_i((0),(1),(1),(1))$$
$$\cdot\,\pi(S_2)\cdot\tau_i(((1),(0\quad 1)),((1),(1\quad 0)),((1),(1\quad 0)),((1),(1\quad 0)))\cdot\pi(S_2)$$
$$\cdot\,\sigma_i((1),(0),(0),(0))\cdot\pi(S_2)\cdot\sigma_i((0),(1),(1),(1))\cdot\pi(S_2)\Bigg),\qquad(6)$$

which corresponds to the dataflow pictured in Figure 9(a).

In this expression, the array of multiplexers $\pi_i(I_2,S_2,S_2,S_2)$ does not perform anything during the last iteration. It is therefore interesting to "push" it after the early termination of the loop, as it can be implemented using only a rewiring $\pi(S_2)$. The array of switches that comes next, $\sigma_i((1),(0),(0),(0))$, does not permute anything during the first three iterations of the loop. It can be therefore "unrolled" into a non-parameterized array of switches $\sigma((1))$, thus saving logic and latency in the loop and therefore increasing throughput. At this point, the expression becomes

$$\sigma((1))\cdot\prod_{i=0}^{3}\Bigg(\overline{T_{2-i}\cdot\bigoplus_{\ell=1}^{4}DFT_2\cdot\pi(S_2)}\cdot\pi(S_2)\cdot\sigma_i((0),(1),(1),(1))\cdot\pi(S_2)$$
$$\cdot\,\tau_i(((1),(0\quad 1)),((1),(1\quad 0)),((1),(1\quad 0)),((1),(1\quad 0)))\cdot\pi(S_2)$$
$$\cdot\,\sigma_i((1),(0),(0),(0))\cdot\pi(S_2)\cdot\sigma_i((0),(1),(1),(1))\cdot\pi(S_2)\Bigg).$$

Continuing these optimizations, and regrouping the two rightmost single arrays of switches finally yields the expression

$$\sigma((1)) \cdot \prod_{i=0}^{3} \left(\overline{T_{2-i} \cdot \bigoplus_{\ell=1}^{4} DFT_2 \cdot \sigma((1)) \cdot \pi(S_2)} \right. \\ \cdot \tau_i(((1), (0 \quad 1)), ((1), (1 \quad 0)), ((1), (1 \quad 0)), ((1), (1 \quad 0))) \cdot \pi(S_2) \\ \left. \cdot \sigma_i'(((1), (0)), ((0), (1)), ((0), (1)), ((0), (1))) \cdot \pi(S_2) \right), \quad (7)$$

pictured in Figure 9(b).

2.3 Streaming-RTL DSL

In the final stage, the streaming blocks are transformed into a dependency graph where each node, called a *signal*, represents a hardware operator that outputs one value per cycle. A signal may have zero (constant signals, inputs, timers and counters), one (flip/flop registers used for pipelining), or more parent signals (see Table 3). In the case of streaming reuse, this graph may contain loops.

The graph is constructed and represented using a *Streaming-RTL* DSL. Hardware datatypes, pipelining decisions and synchronization issues are mostly abstracted from this language. As an example, the implementation of the streaming block for the twiddles T_j can be written within a few lines, and works for every folding scenario and hardware datatype:

```scala
case class Twiddles(n: Int, k: Int, j: Sig[Int])(implicit dt: HW[Complex[Double]])
  extends StreamingBlock[Complex[Double]]](1 << n, 1 << k){
  override def implement(inputs: Vector[Sig[Complex[Double]]]) = {
    // We first declare a timer that ticks for the duration of a dataset
    val timer = Timer(1 << t)

    // we define a (non-staged) Vector containing all 2^n th roots of unity
    val rootsOfUnity = Vector.tabulate(1 << n){i =>
      val angle = -2 * Math.Pi * i / (1 << n)
      Complex(Math.cos(angle), Math.sin(angle))
    }

    // For each input signal,
    inputs.zipWithIndex.map{case (input, p) =>
      // we construct a signal corresponding to the index of a given element
      // (concatenation of the t bits of the timer, and the k bits of the current port p),
      val i = timer ++ Unsigned(k)(p)

      // we compute the corresponding twiddle factor,
      val address = (i & 1) * ((i >>> (j + 1)) << j)
      val twiddle = rootsOfUnity(address)

      // and we return the product of the input signal with this twiddle factor
      input * twiddle
} } }
```

As can be seen, only a few elements in the body of this function (`Timer`, `Unsigned`) may indicate that this code represents a low-level hardware architecture. This improves its readability and therefore its maintainability. However, all signals implicitly carry an underlying hardware type (including the corresponding size in bits), and timing information. All operations are bit- and cycle-accurate, and software and hardware type-safety is ensured. This DSL and its implementation are detailed in the next section.

Once constructed and optimized, the resulting graph is translated to a Verilog file.

Table 3. Example of Nodes (Signals) Used in the Streaming-RTL DSL and Corresponding Syntax

Operator	Description	Example
Constant signal		
`value` is a numerical value.		
`Const(value)`	Constant signal	`Const(5.3)`
Register signal		
`input` is a signal.		
`Register(input)`	Flip/flop register	`input.register()`
Arithmetic signals		
`lhs` and `rhs` are signals of the same type.		
`Plus(lhs, rhs)`	Sum of the operands	`lhs + rhs`
`Minus(lhs, rhs)`	Difference of the operands	`lhs - rhs`
`Times(lhs, rhs)`	Product of the operands	`lhs * rhs`
`And(lhs, rhs)`	Binary AND of the operands	`lhs & rhs`
`Xor(lhs, rhs)`	Binary XOR of the operands	`lhs ∧ rhs`
Memory signals		
`content` is an indexed sequence of numerical values, `address` is an unsigned signal, `input` is a signal, `ram` is a `RAMw` signal.		
`ROM(content, address)`	ROM tabulating `content`	`Vector(1.5, 18.2)(address)`
`RAMw(input, address)`	Write port of a RAM	`val r = RAM(in, address1)`
`RAMr(ram, address)`	Read port of a RAM	`r(address2)`
Multiplexer signal		
`content` is an indexed sequence of signals of the same type, `address` is an unsigned signal.		
`Mux(content, address)`	Multiplexer	`Vector(rhs, lhs)(address)`
Bus manipulation signals		
`lhs` and `rhs` are signals of the same type, `range` is a range of integers.		
`Cons(lhs, rhs)`	Binary concatenation	`lhs ++ rhs`
`Tap(lhs, range)`	Extraction of a selection of bits	`lhs(range)`
Synchronization signals (provided by streaming blocks)		
`size` is an integer.		
`Timer(size)`	Number of cycles since the last dataset entered	`Timer(8)`
`Counter(size)`	Number of datasets that have been processed	`Counter(4)`

3 A DSL FOR "STREAMING-RTL"

Our streaming-RTL DSL (see Figure 3) is used to construct from a streaming-block level representation of an algorithm a dependency graph that represents the final circuit. In this graph, the nodes (signals) represent hardware operators, and the edges the dependencies between these signals. The DSL offers the following features:

- The nodes (*signals*) of the graph are manipulated exactly as the values they would represent in a regular Scala program. Only their type changes.

- The language provides genericity over the actual hardware datatype and precision. However, the datatype can be made explicit, offering bit-accurate control.
- Pipelining and synchronization of data-independent control is performed implicitly, but timing information and manual pipelining remains available.

We discuss next the implementation of these abstractions, using the features offered by the Scala type system.

3.1 Staging and LMS

The implementation of our DSL uses the concept of *staging*, in particular as done in LMS [30], but using our own implementation. Staging allows to distinguish those parts of the computation to be evaluated at generation time and those that will be implemented in hardware via a type annotation. Specifically, staging is done by changing a type `T` to the type `Sig[T]`; the latter means that computations on this type will be delayed, and may become part of the hardware implementation.

For example, in the following code, the first line defines a function `f1` that yields the sum of its parameters (`x` and `y` of type `Double`) augmented by 18. The return type (`Double`) is inferred by the compiler. In the second line however, `f2` returns an expression tree representing the computation on symbolic inputs.

```
def f1(x: Double, y: Double) = x + y + 18
def f2(x: Sig[Double], y: Sig[Double]) = x + y + 18
```

This tree can then be translated (*unparsed*) to RTL-Verilog, yielding an implementation of two adders (adding two signals and an immediate).

This behavior is obtained through the class `Sig[T]`, whose instances represent the nodes in an expression tree:

```
abstract class Sig[T:HW]{
  //timing information
  val delay: Option[Delay]

  //field containing the hardware representation
  val hw = implicitly[HW[T]]
}
```

This class takes as a *type parameter* the type `T` of the expression it represents. This type is expected to come along with a *hardware representation*, provided as a type class `HW` (see Section 3.2). Additionally, each node is expected to provide timing information through the field *delay* (see Section 3.3).

The different types of computation are represented by a class that inherits from `Sig`:

```
//Addition of two nodes
case class Plus[T](lhs: Sig[T], rhs: Sig[T]) extends Sig[T] {...}

//Node containing a constant
case class Const[T:HW](value: T) extends Sig[T] {
  override val delay = None
}

//Pipelining register
case class Register[T](input: Sig[T]) extends Sig[T]{
  override val delay = input.delay.map(_ + 1)
}

...
```

Each instance Sig[T] offers (*lifts*) the same operators as a regular instance of T would (see Section 3.4). These lifted operators return the corresponding node in the form of another instance of Sig.

3.2 Abstraction Over Hardware Datatypes

Type classes [40, 41] are a form of static ad hoc polymorphism, that, contrary to inheritance, allows to retroactively add functionality to existing data types. For instance, the following function f3 is generic in the type T of its parameters, but imposes that this type is numeric:

```
def f3[T:Numeric](x: T, y: T) = x * y
```

In Scala, type classes are implemented using regular classes: f3 expects a third implicit argument of type Numeric[T] containing, among other, the definition of the operator * for two Ts.

Following the concept of *abstraction over data representation* from Reference [36], instances of Sig[T] (*signals* of T) carry their underlying hardware representation in the form of a type class HW[T]:

```
abstract class HW[T](val size: Int){
  //bit representation of a value
  def getBits(value: T): BigInt

  //creates a constant with this hardware representation
  def apply(value: T) = Const(value)(this)
}
```

Not only does this type class provide an additional method getBits that returns the bit representation of a given T, but it carries as meta-information the size in bits of the representation. Concrete hardware representations are instances of classes derived from HW[T]:

```
//Signed integer
case class Signed(_size: Int) extends HW[Int](_size){...}

//Unsigned integer
case class Unsigned(_size: Int) extends HW[Int](_size){...}

//Fixed point number
case class FxP(integral: Int, fractional: Int) extends HW[Double](integral + fractional){...}

//IEE754 floating point
case class IEEE(wE: Int, wF: Int) extends HW[Double](wE + wF + 1){...}

//FloPoCo floating point
case class FloPoCo(wE: Int, wF: Int) extends HW[Double](wF + wE + 3){...}
...
```

A Scala Int could therefore be represented as a signed or unsigned integer of a given size, and a Scala Double can be represented using a fixed-point representation, a FloPoCo number[2] or an IEEE 754 floating-point representation. This information is passed to children nodes, and is used for the implementation of the lifted operators and for the representation of constants in the generated code.

As an example, depending on the underlying hardware type of its parameters, the previous example f2 would seamlessly

- use fixed-point adders and represent 18 as a fixed-point immediate, or
- use FloPoCo generated floating-point adders and represent 18 with the corresponding FloPoCo binary representation, or

[2]The FloPoCo generator is called upon instantiation of the corresponding datatype class to generate the different arithmetic operators. The result of this generation is then parsed to extract the latency of these operators.

- implement a conversion from an IEEE floating-point signal to a FloPoCo representation, implement the FloPoCo adder, and implement the conversion back to an IEEE representation.

3.3 Synchronization

Each signal has a *delay* field that represents the time needed for this signal to output a valid value. It is used to check if two operands are synchronized, and, if it is not the case, to suitably delay one of them using registers.

A delay consists of an integer representing a number of cycles, and a *timeline* indicating to which "reference frame" this delay belongs. This timeline can be

- the *primary* timeline, referring to the number of cycles elapsed since the inputs arrived in the module,
- a *loop* timeline, referring to the number of cycles elapsed since a dataset entered within a loop, or
- a *floating* timeline, used by data-independent signals awaiting to be "synchronized" with another timeline.

As an example, a `Register` would have the same delay as its input with a cycle number incremented by one, while an input signal would have a delay of 0 on the primary timeline.

Loop timelines. In the case of iterative reuse, the *streaming product* (the streaming block that creates the loop and the multiplexer in Figures 1(b) and 1(d)) creates a new loop timeline, and implements its inner expression using this timeline. The corresponding latency is then measured using the maximal delay of the signals that are returned. This information is then used during a second unparsing of the inner expression, where a possible lack of latency is compensated by a FIFO, or an increase of latency of a potential inner temporal permutation. The streaming product then presents its outputs using the same timeline as its inputs, delayed accordingly.

Floating timelines. In our generator, all data-independent control signals rely on counters (that count the number of datasets that have passed) and on timers (that count the number of cycles elapsed since the beginning of the current dataset). To ensure that such control signals become available at the correct instant, each time a new counter or timer is declared, a corresponding floating timeline is created. All data-independent operations performed are then pipelined using this timeline. However, when a signal with a floating timeline and a signal with an external timeline need to be synchronized, a new *floating delay* node is inserted with the expected delay.

As an example, we consider the following function `f4`:

```
def f4(x: Sig[Int]) = {
  val t = Timer(8) + 3
  x ^ t
}
```

This function creates a 3-bit timer, and adds the constant 3 to it. This operation implicitly adds a pipelining register, yielding a signal `t` with a delay of 1 on the floating timeline associated with the timer. The input signal `x` is then xored with `t`. As these two signals are associated with different timelines, a floating delay signal depending on `t` is created with the same `delay` member as `x`, and `f3` finally returns a signal representing a XOR of `x` and the floating delay signal.

After the graph construction, the floating timeline is synchronized with the other timeline such that all floating delays can be implemented using the minimal number of registers. In particular, this ensures that data-dependent signals never have to be uselessly delayed. In our example, the floating timeline is synchronized such that the floating delay is implemented with a direct assignment. Thus, a delay of one cycle on the floating timeline corresponds to the delay of `x`.

To prevent nodes of a floating timeline from being synchronized with different, incompatible timelines, and to avoid circular dependencies between floating timelines, the first time a node of a floating timeline is synchronized with a node from another timeline, the floating timeline is marked as “being in translation” with this other timeline, and an error is thrown if a node is later synchronized with a third timeline. With this relation, when the graph is built, timelines form a set of trees, rooted by the primary and loop timelines. Floating timelines are then synchronized starting from the roots.

Synchronization tokens. When the graph is unparsed, token synchronization signals are generated to trigger the different counters and timers. Tokens for loop timelines are generated by “ORing” tokens of the primary timeline. As the maximal throughput of the design is known at this time, tokens of the primary timeline can be generated using consecutive resettable timers instead of a resettable shift-register.

In our previous example, the timer declared within `f3` receives its token one cycle before `x` becomes available, ensuring that `t` is computed at the right time.

3.4 Smart Constructors

Lifted operators are provided using *implicit classes*, which make it possible to add a posteriori methods and operators to existing objects.

For instance, the following class provides a + operator to any `Sig[T]`, when `T` is a numeric type:

```scala
implicit class NumericSig[T:Numeric](lhs: Sig[T]){
  //the default hardware representation when creating Const is the one of the left-hand side
  implicit val hw = lhs.hw

  //lhs + rhs in the case where lhs is a Sig[T] and rhs a T
  def +(rhs: T): Sig[T] = lhs + Const(rhs)

  //lhs + rhs in the case where both lhs and rhs are Sig[T]
  def +(rhs: Sig[T]): Sig[T] = {
    ensure(rhs.hw == hw) //check if lhs and rhs have the same representation
    (lhs, rhs).synch match {
      //both lhs and rhs are Const
      case (Const(x), Const(y)) => Const(x + y)

      //one of the operands is null
      case (Zero(), _) => rhs
      case (_, Zero()) => lhs

      //otherwise, we create a new node specialized for the hardware representation
      case (lhs, rhs) => hw match {
        case _: FloPoCo => PlusFPC(lhs, rhs)
        case _: IEEE => (lhs.toFPC + rhs.toFPC).toIEEE
        case _: Signed | _: Unsigned | _: FxP => Plus(lhs, rhs).register
        case _ => throw new Exception("No hardware implementation for +")
} } } }
```

Here, the operator first checks that the two operands have the same hardware type (ensuring type-safety). It then synchronizes them, and handles particular cases (if the two operands are constants, or if one of them is the constant zero). Finally, it creates a new Plus signal, according to the hardware datatype, and adds pipelining registers (in the case of a FloPoCo operator, the signal `PlusFPC` already takes into account the latency of the operator. A final register is added. For signed, unsigned integers and fixed-point representations, the pipeline has always a depth of one cycle. It would however be possible to adapt it to the size of the operands, the target architecture or the target frequency. In case of an IEEE representation, the pipelining is handled by the underlying call to the FloPoCo operator.).

These *smart constructors* are responsible for major optimizations. As an example, the constructor of ROM signals (implemented by adding a new `apply` method on indexed sequences of `T`) checks every bit of the control signal, and returns a smaller ROM in the case where one of them is constant. Particularly, it would return a constant if the control signal is constant, thus guaranteeing an efficient implementation of the twiddle stage T_j, even in non-streaming or non-iterative cases.

4 STREAMING-BLOCK DSL

The Streaming-block DSL is an intermediate language between SPL and the Streaming-RTL DSL (See Figure 3). It supports high-level optimizations relevant for streaming, i.e., optimizations that take place once an algorithm has been "folded" according to a given streaming width.

4.1 Streaming Blocks

Each streaming block represents a hardware module that has the same number (`streamingWidth`) of inputs and outputs of the same hardware type (`HW[T]`), and that performs an operation on a dataset of size `size`. Streaming blocks are comparable to SPL elements augmented with a streaming width information, and are instances of classes derived from `StreamingBlock`:

```
abstract class StreamingBlock[T:HW](size: Int, streamingWidth: Int){
  assert(size % streamingWidth == 0) //Size must be a multiple of the streaming width
  def implement(inputs: Vector[Sig[T]]): Vector[Sig[T]]
  def *(rhs: StreamingBlock[T]) = Product(this, rhs) //composition of blocks
}
```

Streaming blocks are expected to override a virtual method `implement` that constructs the final circuit using the streaming-RTL DSL.[3] An operator `*` allows the composition of blocks as explained in Section 4.2 below.

As an example, an array of butterflies (corresponding to the SPL expression $\bigoplus \mathrm{DFT}_2$) would be implemented as follows:

```
case class ButterflyArray[T:HW:Numeric](_size: Int, _streamingWidth: Int)
  extends StreamingBlock[T](_size, _streamingWidth){
  assert(streamingWidth % 2 == 0) //The streaming width must be even
  override def implement(inputs: Vector[Sig[T]]) = {
    assert(inputs.size == streamingWidth)
    //Compute the sum and difference of each pair of inputs
    inputs.grouped(2).toVector.flatMap{case Vector(a, b) => Vector(a + b, a - b)}
} }
```

4.2 Higher-Order Blocks

Composition of blocks is achieved through the block `Product`:

```
case class Product[T:HW] private (factors: Vector[StreamingBlock[T]])
  extends StreamingBlock[T](factors.head.size, factors.head.streamingWidth){
  //size and streaming width of all factors must be the same
  assert(factors.forall(f => f.size == size && f.streamingWidth == streamingWidth))
  override def implement(inputs: Vector[Sig[T]]) = {
    assert(inputs.size == streamingWidth)
    //implement all factors by connecting the outputs of one to the inputs of the next
    factors.foldRight(inputs)((sb, cur) => sb.implement(cur))
} }
```

[3]The implementation adds an option to add latency and returns the minimal number of cycles (gap) that the circuit can handle between datasets.

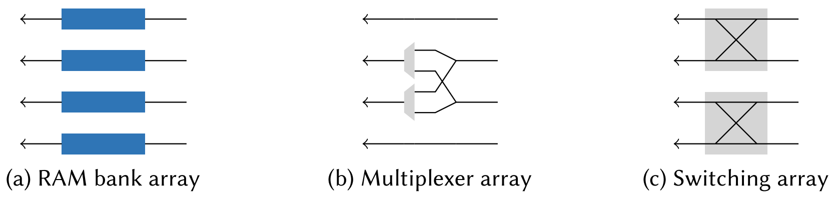


Fig. 10. Permutation streaming blocks, from Reference [28], here for a streaming width of $2^k = 4$. Panel (a) can pass any temporal permutation; panels (b) and (c) are used for spatial permutations.

A companion object of `Product` contains a smart constructor (this is the one called by the operator `*` in `StreamingBlock`), along with a higher-order function that implements the block corresponding to the SPL expression $\prod_{j=0}^{\text{limit}} f(j)$.

```
object Product{
  def apply[T:HW](lhs: StreamingBlock[T], rhs: StreamingBlock[T]): Product[T] =
    (lhs, rhs) match {
    case (Product(f1), Product(f2)) => new Product(f1 ++ f2)
    case (Product(f1), _) => new Product(f1 :+ rhs)
    case (_, Product(f2)) => new Product(lhs +: f2)
    case _ => new Product(Vector(lhs, rhs))
  }
  def apply[T:HW](limit: Int)(f: Sig[Int] => StreamingBlock[T]): StreamingBlock[T] = {
    assert(limit > 0)
    //Unsigned with the minimal size that can contain all j
    val idxType = Unsigned(BigInt(limit - 1).bitSize)
    (0 until limit).map(i => Const(i)(idxType)).map(f).reduce(_ * _)
  }
}
```

Note that the higher-order function expects a function that returns a block, and that takes an integer signal as a parameter (and not directly an integer). This allows us to have another block, `ItProduct` with the same interface, but that produces a loop for iterative reuse (by implementing a multiplexer, creating a new loop timeline, and calling `f` with a `Counter` as a parameter).

The Streaming-block DSL does not directly have any operator corresponding to the direct sum of SPL: $\bigoplus_j f(j)$. Before folding, the SPL expression must therefore have all the direct sums fully distributed. Then, the remaining direct sums must be handled within the streaming blocks themselves, as it is the case with `ButterflyArray`.

4.3 Permutation Blocks

Apart from the direct sum operator, permutations are the only SPL operators that do not have a direct equivalent in the Streaming-block DSL. Only two types of permutations are directly implementable as streaming blocks:

- *Spatial* permutations: these are permutations that permute elements only within the same cycle. They can be implemented using switches (Figure 10(c)) or multiplexers (Figure 10(b)).
- *Temporal* permutations: these permutations permute elements only between cycles, but stay on the same port number. They can be implemented using an array of memory banks as in Figure 10(a).

During the folding operation, general permutations are decomposed into these basic types (as depicted in Figure 8) using the algorithms described in References [25, 28].

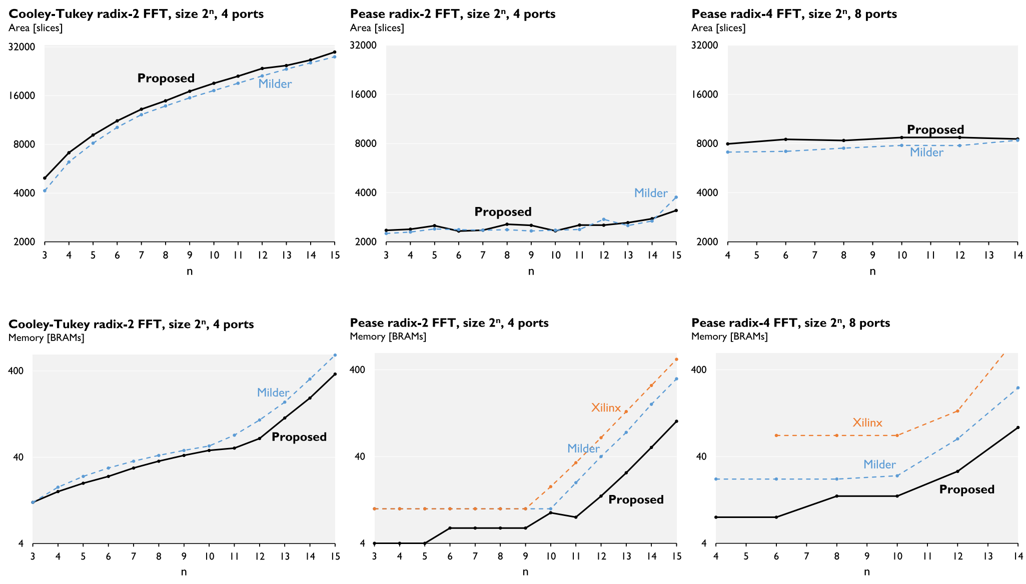


Fig. 11. Resources used by different FFTs (Equation (2)) in different configurations on complex data using 2×32 bits IEEE754 floating-point.

4.4 Optimizations

The optimizations taking place at this step mainly concern streaming permutations and iterative reuse loops that have an early termination. They are described in greater detail in Reference [28]. For instance, a streaming permutation block within an iterative loop may be unrolled under certain conditions, thus reducing the global number of multiplexers used within the whole design, or increasing the throughput of the design. Another optimization consists of fusing two consecutive arrays of 2-input switches into an array of 4-input switches, which can improve the resource used on some FPGA architectures. Figure 9 shows a case where these two optimizations were performed.

5 RESULTS

To validate the designs produced by our generator, we benchmarked them against the equivalent circuits generated with Reference [17]. All designs were synthesized using Vivado 2018.1, targeting a Virtex7 xc7vx1140 FPGA. The floating-point operators used in our designs were generated using FloPoCo 4.1.2, targeting a 700MHz Virtex6 platform.

Figures 11, 12, and 13 show results after place-and-route for a variety of transforms, algorithms, hardware datatypes and foldings. Each of these presents, for a given transform size, the resources used in terms of logic slices and memory obtained for our design and the corresponding design from [17] or [22]. Cooley-Tukey FFTs and WHTs (Figures 11(a), 11(d), and 12) and Stone SNs (Figures 13(a) and 13(c)) are implemented using only streaming reuse. Batcher SNs (Figures 13(b) and 13(d)) use both streaming and iterative reuse. Pease FFTs (Figures 11(b), 11(c), 11(e), and 11(f)) use streaming and iterative reuse with fused permutations, as described in Reference [28].

Figures 11(e) and 11(f) also show a comparison with the designs obtained using Xilinx IP generator Fast Fourier Transform 9.1. The parameters were set to resemble as much as possible our architecture. A radix-2 Burst IO Lite architecture is used with four channels (in Figure 11(e)), and a radix-4 Burst IO is used with eight channels (in Figure 11(f)), in each case with a natural output ordering, and using 32 bit fixed point representation for both inputs and phase factors. However,

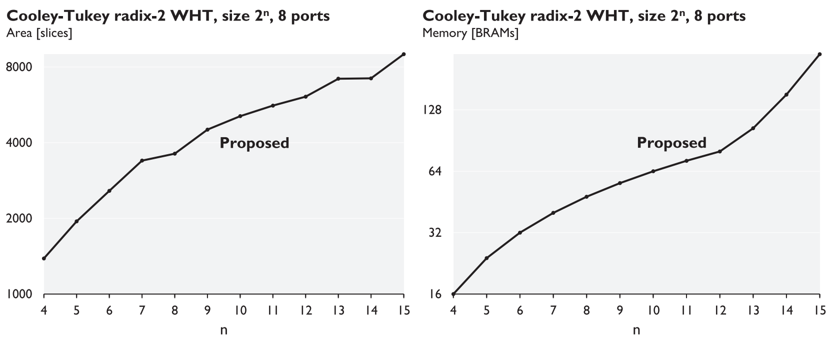


Fig. 12. Resources used by a radix-2 Cooley-Tukey WHTs (Equation (3)), with a streaming width of 8 on complex data using 2×32bits fixed-point.

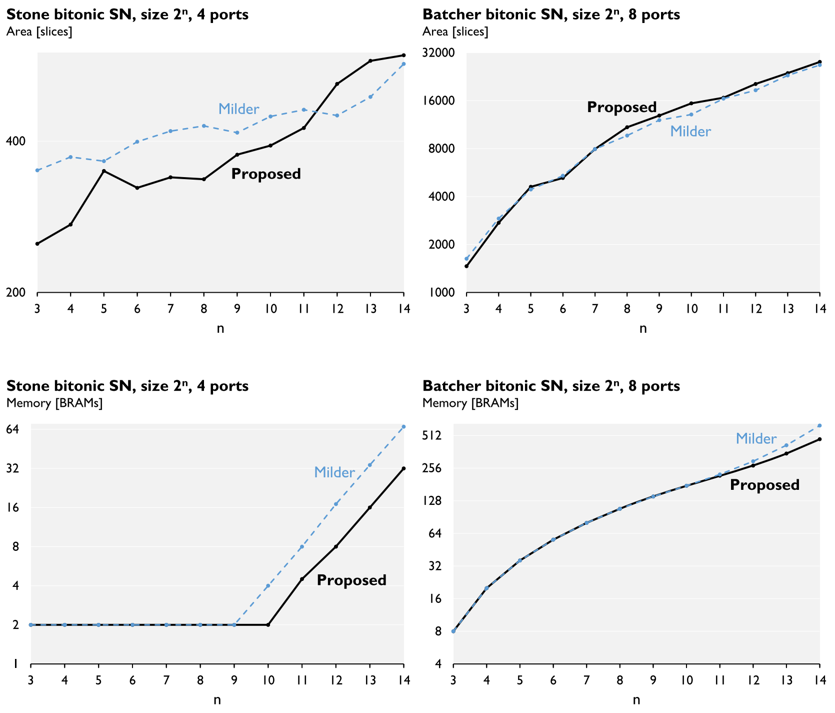


Fig. 13. Resources used by different bitonic sorting networks (Equation (4)), with different streaming configuration on complex data using 2×32bits fixed-point.

as neither floating point computation nor the streaming IO architecture are available when using multiple channels, no comparison can be made for logic consumption, nor for the Cooley-Tukey architecture.

All our designs were generated with sufficient pipelining to reach the same frequencies as Reference [17] or Reference [22] (around 400MHz). The throughput of these designs are therefore the same. Additionally, designs requiring complex multiplications (Figure 11) use an algorithm that yield the same number of DSP slices as Reference [17].

We observe that the logic area consumption slightly increases. The gain obtained using FloPoCo for the arithmetic part and the use of 4-input multiplexers is counter-balanced by the additional logic needed to implement memory conflict avoidance as described in Reference [25].

On the other side, the number of BRAM tiles required is lower in average with our generator. This is a direct consequence of the streaming permutations being implemented with Reference [25]. As it does not use double buffering, the capacity required to implement streaming permutations is halved. However, this reflects on the number of BRAMs only when the double buffer does not fit into a single BRAM, that is, for large sizes of n ($n > 10$ in Figure 11 and $n > 9$ in Figure 13(c)). In addition, in the case of FFTs with iterative reuse (Figures 11(e) and 11(f)), the stride permutation and the bit-reversal are fused, allowing to halve the number of BRAMs used for streaming permutations. However, these techniques do not affect the number of RAM slices used as ROMs to store the twiddle factors. Finally, both the designs we propose and those generated using [17] require significantly fewer RAMs than Xilinx Fast Fourier transform IP cores.

In summary, our generator produces designs that use an equal amount or less memory than Reference [17] or Reference [22], particularly for large sizes, for the same number of DSP blocks, and for a comparable area consumption. Our generator is thus able to improve the state of the art for important parts of the design space of the considered transforms, and yields new Pareto optima. Benchmarks for other designs are available in Reference [28], which uses the same generator.

6 LIMITATIONS AND RELATED WORK

We compare to related work and discuss limitations.

6.1 Hardware DSLs Implemented in Scala

The DSL we propose is specifically crafted for the generation of streaming Fourier transforms and sorting networks on FPGAs, and provides only the primitives and the amount of abstraction needed for this purpose. This differentiates it from lower level hardware description languages written in Scala. For instance, Chisel [42] can represent a much wider variety of hardware designs, but requires the pipelining registers to be manually added. Targeting dataflow hardware, DFiant [43] proposes a dependency-driven automatic pipelining similar to ours, but does not seem to support automatic synchronization of data-independent controls. It uses literal types to expose the hardware datatype and precision to the user, thus enforcing type safety at compile-time. In our case, the hardware datatype is abstracted (provided via a type class), and hardware type safety is only ensured at generation time. However, high-level synthesis tools [44, 45] would offer even higher abstractions, up to the dataset level, but would not allow the user to program at the port-level, thus making the implementation of our permutation streaming blocks difficult.

LMS [30] itself not only provides staging, but offers a tool chain to implement and compile DSLs. Particularly, it grants automatic common subexpression elimination during the construction of the dependency graph. However, in our case, floating timelines reduce the efficiency of such an optimization during the graph construction, and our tests have shown that synthesis software such as Vivado already provide it, thus limiting the use of implementing it. LMS provides as well a facility to manipulate the generated graph, but as ours already includes timing information, these manipulations are limited to timing invariant ones (fusing ROMs that contain identical values for instance), for which a direct implementation is possible. The main optimizations in our graph are made during its generation, using smart constructors.

The pipeline proposed in Reference [46] to generate matrix operations illustrates the capability of LMS to target hardware. It shares many similarities with ours, particularly its use of LMS and FloPoCo. However, a significant part of the final RTL design is outsourced to the external back-end LegUp [47].

6.2 Hardware Generator for FFTs

Our generator only handles the generation of power-of-two-sized FFTs, whereas Reference [17] covers a larger set of sizes that can be factored into small primes and additional transforms closely related to FFTs. An according extension of our generator should be relatively straightforward. Note that Reference [17] works as a back-end of Spiral [23], a generator written on a modified version of the GAP computer algebra system, thus requiring high skills for its development.

SPL and Spiral have as well been implemented and enhanced in Haskell [48] and in Scala [49] to produce efficient FFT implementations in C. A VHDL back-end for this compiler is being developed [48].

6.3 Hardware Generator for Sorting Networks

The work in References [21, 22] presents a generator for streaming sorting networks, and we have shown how our generator outperform its RAM consumption for the algorithms we support. However, References [21, 22] cover a larger space of algorithms (called SN2–4), and was originally targeting (and thus optimized for) a platform (Xilinx Virtex 5) older than the one we used for our benchmarks (Xilinx Virtex 7).

Sorting is a classic topic in computer science [3, 50], and many high-performance sorting networks have been manually implemented on FPGAs, using two-input sorters [51, 52] or using other basis elements like linear sorters [53].

7 CONCLUSION

The overall theme in our work is the principled design of domain-specific hardware generators using state-of-the-art languages and language features. This article followed this theme with the design and implementation of a generator for streaming FFTs and sorting networks inside Scala, using embedded DSLs and the concept of staging. Specifically, our generator employs a pipeline of three abstraction levels, corresponding to three levels of DSLs. Two of them, the streaming-block DSL and the streaming-RTL DSL are novel and were specifically designed to include state-of-the-art components and enable the transformations and optimizations needed in these transforms. The produced designs improve over prior work on memory usage. The generator should be easily extendable to other DSP components related to FFTs. A web version of our generator is available at Reference [31] and its source code is available at Reference [32].

REFERENCES

[1] Jacques Hadamard. 1893. Résolution d'une question relative aux déterminants. *Bulletin des sciences mathématiques* 17 (1893), 240–246.
[2] J. Astola and D. Akopian. 1999. Architecture-oriented regular algorithms for discrete sine and cosine transforms. *IEEE Trans. Signal Process.* 47, 4 (1999), 1109–1124.
[3] Ken Edward Batcher. 1968. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference*(AFIPS'68), Vol. 32. 307–314.
[4] Harold S. Stone. 1971. Parallel processing with the perfect shuffle. *IEEE Trans. Comput.* 20, 2 (1971), 153–161.
[5] Váaclav Edvard Beneš. 1965. *Mathematical Theory of Connecting Networks and Telephone Traffic.* Academic Press.
[6] Abraham Waksman. 1968. A permutation network. *J. ACM* 15, 1 (1968), 159–163.
[7] Marshall C. Pease. 1977. The indirect binary n-cube microprocessor array. *IEEE Trans. Comput.* 26, 5 (1977), 458–473.
[8] Jacques Lenfant and Serge Tahé. 1985. Permuting data with the Omega network. *ACTA Informatica* 21, 6 (1985), 629–641.
[9] David Steinberg. 1983. Invariant properties of the shuffle-exchange and a simplified cost-effective version of the Omega network. *IEEE Trans. Comput.* 32, 5 (1983), 444–450.
[10] David Nassimi and Sartaj Sahni. 1981. A self-routing Benes network and parallel permutation algorithms. *IEEE Trans. Comput.* 30, 5 (1981), 332–340.

[11] Danny Cohen. 1976. Simplified control of FFT hardware. *IEEE Trans. Acoust. Speech Signal Process.* 24, 6 (1976), 577–579.

[12] Pinit Kumhom, Jeremy R. Johnson, and Prawat Nagvajara. 2000. Design, optimization, and implementation of a universal FFT processor. In *Proceedings of the International ASIC/SOC Conference (ASIC'00)*. 182–186.

[13] Ainhoa Cortés, Igone Vélez, and Juan F. Sevillano. 2009. Radix r^k FFTs: Matricial representation and SDC/SDF pipeline implementation. *IEEE Trans. Signal Process.* 57, 7 (2009), 2824–2839.

[14] Shousheng He and Mats Torkelson. 1996. A new approach to pipeline FFT processor. In *Proceedings of the Parallel Processing Symposium (IPPS'96)*. 766–770.

[15] Berkin Akin, Franz Franchetti, and James C. Hoe. 2015. FFTs with near-optimal memory access through block data layouts: Algorithm, architecture and design automation. *J. Signal Process. Syst.* 85, 1 (2015), 67–82.

[16] Byung G. Jo and Myung H. Sunwoo. 2005. New continuous-flow mixed-radix (CFMR) FFT Processor using novel in-place strategy. *IEEE Trans. Circ. Syst. I* 52, 5 (2005), 911–919.

[17] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. 2012. Computer generation of hardware for linear digital signal processing transforms. *ACM Trans. Design Autom. Electron. Syst.* 17, 2 (2012), 15:1–15:33.

[18] Mario Garrido, Miguel Ángel Sánchez, María Luisa López-Vallejo, and Jesùs Grajal. 2017. A 4096-point Radix-4 memory-based FFT using DSP slices. *IEEE Trans. Very Large Scale Integr. Syst.* 25, 1 (2017), 375–379.

[19] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. 2008. Linear transforms: From math to efficient hardware. In *Proceedings of the Workshop on High-Level Synthesis Colocated with DAC*.

[20] Grace Nordin, Peter A. Milder, James C. Hoe, and Markus Püschel. 2005. Automatic generation of customized discrete Fourier transform IPs. In *Proceedings of the Design Automation Conference (DAC'05)*. 471–474.

[21] Marcela Zuluaga, Peter A. Milder, and Markus Püschel. 2012. Computer generation of streaming sorting networks. In *Proceedings of the Design Automation Conference (DAC'12)*. 1245–1253.

[22] Marcela Zuluaga, Peter A. Milder, and Markus Püschel. 2016. Streaming sorting networks. *ACM Trans. Design Autom. Electron. Syst.* 21, 4 (2016).

[23] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE Spec. Issue* 93, 2 (2005), 232–275.

[24] Florent de Dinechin and Bogdan Pasca. 2011. Designing custom arithmetic data paths with FloPoCo. *IEEE Design Test Comput.* 28, 4 (2011), 18–27.

[25] François Serre, Thomas Holenstein, and Markus Püschel. 2016. Optimal circuits for streamed linear permutations using RAM. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'16)*. 215–223.

[26] Thaddeus Koehn and Peter Athanas. 2016. Arbitrary streaming permutations with minimum memory and latency. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'16)*. 1–6.

[27] François Serre. 2019. *Optimal Streaming Permutations and Transforms: Theory and Implementation*. Ph.D. Dissertation. ETH Zurich.

[28] François Serre and Markus Püschel. 2018. Memory-efficient fast Fourier transform on streaming data by fusing permutations. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. 219–228.

[29] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala*. Artima Inc.

[30] Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM* 55, 6 (June 2012), 121–130.

[31] François Serre. 2018. SGen—A streaming hardware generator. Retrieved from https://acl.inf.ethz.ch/research/hardware/.

[32] François Serre. 2018. DFT and streamed linear permutation generator for hardware. Retrieved from https://github.com/fserre/sgen.

[33] François Serre and Markus Püschel. 2018. A DSL-based FFT hardware generator in Scala. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'18)*. 315–322.

[34] Jianxin Xiong, Jeremy Johnson, Robert W. Johnson, and David Padua. 2001. SPL: A language and compiler for DSP algorithms. In *Proceedings of the Conference on Programming Languages Design and Implementation (PLDI'01)*. 298–308.

[35] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. 1990. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circ. Syst. Signal Process.* 9, 4 (1990), 449–500.

[36] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. 2013. Spiral in Scala: Towards the systematic construction of generators for performance libraries. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE'13)*. 125–134.

[37] Marshall C. Pease. 1968. An adaptation of the fast Fourier transform for parallel processing. *J. ACM* 15, 2 (1968), 252–264.

[38] Markus Püschel, Peter A. Milder, and James C. Hoe. 2009. Permuting streaming data using RAMs. *J. ACM* 56, 2 (2009), 10:1–10:34.

[39] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. 2009. Operator language: A program generation framework for fast kernels. In *Proceedings of the IFIP Working Conference on Domain Specific Languages (DSL WC'09) (Lecture Notes in Computer Science)*, Vol. 5658. Springer, 385–410.

[40] Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 60–76.

[41] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2017. Staging for generic programming in space and time. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE'17)*. 15–28.

[42] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the Design Automation Conference (DAC'12)*. 1216–1225.

[43] O. Port and Y. Etsion. 2017. DFiant: A dataflow hardware description language. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'17)*. 1–4.

[44] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the International Conference on Machine Learning (ICML'11)*. 609–616.

[45] Nithin George, HyoukJoong Lee, David Novo, Muhsen Owaida, David Andrews, Kunle Olukotun, and Paolo Ienne. 2015. Automatic support for multi-module parallelism from computational patterns. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'15)*. 1–8.

[46] Nithin George, David Novo, Tiark Rompf, Martin Odersky, and Paolo Ienne. 2013. Making domain-specific hardware synthesis tools cost-efficient. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'13)*. 120–127.

[47] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'11)*. 33–36.

[48] Geoffrey Mainland and Jeremy Johnson. 2017. A Haskell compiler for signal transforms. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences (GPCE'17)*. 219–232.

[49] Georg Ofenbeck. 2017. *Generic Programming in Space and Time*. Ph.D. Dissertation. ETH Zurich.

[50] Donald Ervin Knuth. 1978. *The Art of Computer Programming (Addison-Wesley Series in Computer Science and Information*, 2nd ed. Addison-Wesley Longman Publishing, Boston, MA.

[51] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2012. Sorting networks on FPGAs. *VLDB J.* 21, 1 (2012), 1–23.

[52] Ren Chen, Sruja Siriyal, and Viktor Prasanna. 2015. Energy and memory efficient mapping of bitonic sorting on FPGA. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. 240–249.

[53] J. Ortiz and D. Andrews. 2010. A configurable high-throughput linear sorter system. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing, Workshops, and Ph.D. Forum (IPDPSW'10)*. 1–8.

Received December 2018; revised July 2019; accepted August 2019