Computer Generation of Hardware for Linear Digital Signal Processing Transforms

PETER MILDER, FRANZ FRANCHETTI, and JAMES C. HOE, Carnegie Mellon University
MARKUS PÜSCHEL, ETH Zurich

Linear signal transforms such as the discrete Fourier transform (DFT) are very widely used in digital signal processing and other domains. Due to high performance or efficiency requirements, these transforms are often implemented in hardware. This implementation is challenging due to the large number of algorithmic options (e.g., fast Fourier transform algorithms or FFTs), the variety of ways that a fixed algorithm can be mapped to a sequential datapath, and the design of the components of this datapath. The best choices depend heavily on the resource budget and the performance goals of the target application. Thus, it is difficult for a designer to determine which set of options will best meet a given set of requirements.

In this article we introduce the Spiral hardware generation framework and system for linear transforms. The system takes a problem specification as input as well as directives that define characteristics of the desired datapath. Using a mathematical language to represent and explore transform algorithms and datapath characteristics, the system automatically generates an algorithm, maps it to a datapath, and outputs a synthesizable register transfer level Verilog description suitable for FPGA or ASIC implementation. The quality of the generated designs rivals the best available handwritten IP cores.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids—*Automatic synthesis*

General Terms: Algorithms, Design

Additional Key Words and Phrases: digital signal processing transform, discrete Fourier transform, fast Fourier transform, hardware generation, high-level synthesis, linear transform

ACM Reference Format:
Milder, P., Franchetti, F., Hoe, J. C., and Püschel, M. 2011. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. ACM Trans. Des. Autom. Electron. Syst. 17, 2, Article 15 (April 2012), 33 pages.
DOI = 10.1145/2159542.2159547 http://doi.acm.org/10.1145/2159542.2159547

1. INTRODUCTION

Linear signal transforms such as the discrete Fourier transform or discrete cosine transform are ubiquitous in digital signal processing (DSP), scientific computing, and communication applications. Fast algorithms for computing these transforms are

A preliminary version of a portion of this article appeared in the *Proceedings of 45th annual Design Automation Conference*, 2008.
This work was supported by NSF through award 0702386, by ONR through grant N000141110112 and 10219108, and the ARO grant W911NF0710416. The authors would like to thank their sponsors for their support.
Authors' addresses: P. Milder (corresponding author), F. Franchetti, J. C. Hoe, Department of ECE, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213; email: pam@ece.cmu.edu; M. Püschel, Dept. of Computer Science, ETH Zurich, RZ H18, Clausiusstrasse 59, 8092 Zurich, Switzerland.

DOI 10.1145/2159542.2159547 http://doi.acm.org/10.1145/2159542.2159547

highly structured and regular, and they exhibit large amounts of parallelism. For these reasons, they are well suited for hardware implementation as sequential datapaths on field-programmable gate arrays (FPGA) or application-specific integrated circuits (ASIC). The regular structure in transform algorithms yields a large amount of freedom when mapping to a datapath. Given that multiple algorithmic options are typically available for a given transform, the combined algorithm/datapath space quickly becomes far too large for a designer to explore easily.

Further, the algorithmic and datapath options are mutually restricting; the best choice in one domain depends on which choices are made in the other. However, designers typically do not have the tools available to jointly reason about both sets of options. Moreover, the best choices for both algorithm and datapath structures are highly dependent on the context—namely the application-specific performance goals and cost requirements.

Typically, a designer attempting to build a customized hardware implementation of a linear signal transform will alternate between (a) exploring different microarchitectures to execute a given algorithm and (b) exploring algorithms to be executed on a given type of microarchitecture. Either way, the designer reasons about one portion of the problem while keeping an implicit mapping between algorithm and datapath in his or her mind. This human-driven exploration process is difficult and slow, and few designers have the required experience with both algorithm and hardware design domains needed to arrive at the options best suited for their application requirements. Often, they resort to using solutions from IP vendors that provide common designs across a small number of cost/performance trade-off points. Many modern applications in communications, image processing, and scientific computing require increasingly high performance or more specialized implementations, and thus, prebuilt IP solutions are frequently unsuitable.

In this article, we aim to solve these problems through automation. We introduce the Spiral hardware generation framework and system which uses high-level mathematical formalism to automatically generate customized hardware implementations of linear signal transforms. The system covers a wide range of algorithmic and datapath options and frees the designer from the difficult process of manually performing algorithm and datapath exploration. The generated designs cover a wide cost/performance trade-off space, are competitive with good hand-designed implementations, and are scalable to reach high levels of performance. Due to the automation of the design process, it becomes possible to easily explore and select from among a wide space of high quality options.

The basic idea underlying the proposed system is a domain-specific formula-based language for specifying transform algorithms and sequential datapaths on which to execute them. This language extends the matrix formalism in Van Loan [1992] to include datapath concepts, such as parallelism and explicit datapath reuse, which the designer specifies at a high level of abstraction. A single formula in this extended language specifies one particular algorithm and one sequential datapath on which to execute it.

The mathematical language drives a full compilation system that begins with a problem specification and produces a synthesizable register-transfer level Verilog description. This system takes a linear signal transform of a given size as input as well as high-level hardware directives that describe features of the desired datapath. Then, the system utilizes a base of algorithmic knowledge to construct a formula that specifies a transform algorithm. Next, the formula is rewritten (based on user-provided directives) to produce a *hardware formula* that explicitly specifies a datapath with sequential reuse of hardware structures. Lastly, the system compiles the hardware formula to a corresponding synthesizable register-transfer level Verilog description.

We include an evaluation of the generated designs across several transforms (such as the discrete Fourier transform, discrete cosine transform, and others), multiple datatypes, and two platforms: Xilinx FPGAs and a 65nm standard cell library for ASIC implementation. The results show that Spiral is able to generate designs across a wide range of cost/performance trade-offs and to match existing benchmarks where available.

In previous work, Spiral uses similar techniques to automatically produce software implementations of signal transforms, including automatic parallelization and vectorization [Franchetti et al. 2006a; 2006b; Püschel et al. 2005].

The rest of this article is organized as follows. Section 2 describes relevant background material on linear transforms, their algorithms, and algorithmic specification in the previously mentioned formula language. Then, Section 3 explains the proposed mathematical formula language for describing sequentially reused datapaths. Next, Section 4 outlines the automatic compilation process and describes each step from transform to formula to hardware implementation. Section 5 evaluates the designs produced with this methodology on FPGA and ASIC. Lastly, Section 6 examines related work, and Section 7 presents concluding remarks.

2. LINEAR TRANSFORMS AND ALGORITHMS

In this section, we present background material on linear transforms and fast algorithms for their computation. In Section 2.1 we define linear transforms and give relevant examples. Then we show in Section 2.2 how a mathematical formula language can specify transform algorithms and how formulas can be translated to combinational datapaths. Most importantly, the algorithm-specification language explicitly captures regularity and repetition within algorithms.

2.1. Linear Transforms

A linear transform on n points is specified by a (typically) dense $n \times n$ matrix. Applying the transform to an n point input vector is then a matrix-vector multiplication. For example, the discrete Fourier transform on n points is defined as $y = \mathrm{DFT}_n\, x$, where x and y are, respectively, n point complex input and output vectors, and

$$\mathrm{DFT}_n = [\omega_n^{k\ell}]_{0 \le k,\ell < n}, \quad \omega_n = e^{-2\pi i/n}.$$

In this notation, DFT_n is an $n \times n$ matrix, and k and ℓ are the row and column index of a given element (respectively). For example, computing a four-point DFT (with input vector x and output vector y) yields the following matrix-vector product.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

Some important transforms used within this work include

$$\mathrm{DFT}_n = \left[\omega_n^{k\ell}\right]_{0\le k,\ell<n}, \quad \omega_n = e^{-2\pi i/n}, \tag{1}$$

$$\mathrm{DFT}_n^{-1} = \mathrm{IDFT}_n = (1/n)\cdot\left[\omega_n^{-k\ell}\right]_{0\le k,\ell<n}, \tag{2}$$

$$\text{2D-DFT}_{n\times n} = \mathrm{DFT}_n \otimes \mathrm{DFT}_n \tag{3}$$

$$\mathrm{RDFT}_n = \begin{bmatrix} 1 & 1\ 1 & 1 \ \ldots & 1 \\ 1 & -1\ 1 & -1\ \ldots & -1' \\ \begin{bmatrix}\cos\frac{2\pi k\ell}{n}\\ \sin\frac{2\pi k\ell}{n}\end{bmatrix}_{0<k<n/2,\ 0\le\ell<n} \end{bmatrix}, \tag{4}$$

$$\text{DCT-2}_n = \left[\cos\frac{k(2\ell+1)\pi}{2n}\right]_{0\le k,\ell<n}, \tag{5}$$

where $\otimes$ is the *tensor* or *Kronecker product*, defined as

$$B\otimes A = [b_{k,\ell}A], \quad \text{where } B = [b_{k,\ell}]. \tag{6}$$

The tensor product replaces each entry $b_{k,\ell}$ of matrix B with the matrix $b_{k,\ell}A$. Equations (1) and (2) define the discrete Fourier transform and its inverse. The two-dimensional DFT is given by Equation (3). Equation (4) defines a version of the real discrete Fourier transform, and Equation (5) defines the discrete cosine transform of type two.

2.2. Formula Representation of Transform Algorithms

Computing an n point transform by definition requires $\mathrm{O}(n^2)$ arithmetic operations. Fast transform algorithms enable computation using only $\mathrm{O}(n\log n)$ arithmetic operations. The term *fast Fourier transform* (FFT) refers to such an algorithm for computing the discrete Fourier transform, and many different FFTs exist. A fast transform algorithm can be expressed as a decomposition of the $n\times n$ transform matrix into a product of *structured sparse matrices*. For example, one FFT for four points can be expressed as

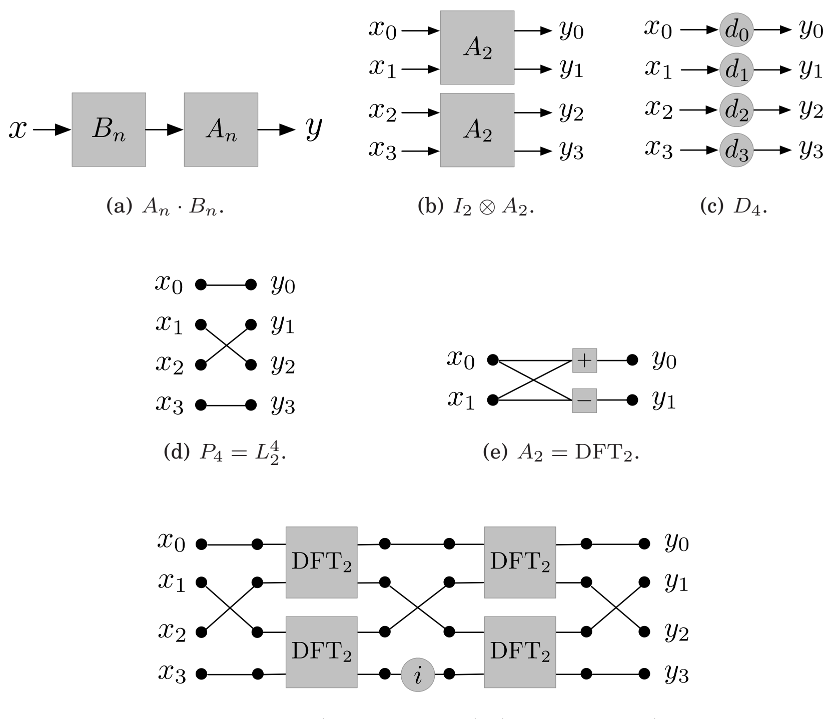
$$\mathrm{DFT}_4 = \begin{bmatrix}1&0&1&0\\0&1&0&1\\1&0&-1&0\\0&1&0&-1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&i\end{bmatrix}\begin{bmatrix}1&1&0&0\\1&-1&0&0\\0&0&1&1\\0&0&1&-1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix}. \tag{7}$$

Thus, computing $y = \mathrm{DFT}_4 \cdot x$ is equivalent to multiplying x by the four matrices from right to left. More generally, the algorithms considered in this article decompose the $n\times n$ transform matrix into a product of sparse matrices ($\mathrm{O}(\log n)$ many), each of which requires $\mathrm{O}(n)$ operations. Thus, the overall operation count is $\mathrm{O}(n\log n)$.

The Kronecker product formalism in Van Loan [1992] uses linear algebra concepts to mathematically represent fast transform algorithms by capturing the structure within the sparse matrices of the decomposition. This formalism serves as the basis for a language that represents transform algorithms as *formulas*, where each term in the formula has a corresponding dataflow interpretation. Thus, a formula in this language can be directly translated into a combinational hardware implementation.

In Backus-Naur form, this language is defined as follows.

$$\begin{array}{lll} \mathbf{matrix}_n ::= & \mathbf{matrix}_n \cdots \mathbf{matrix}_n & \\ & \mid\ \prod_\ell \mathbf{matrix}_n & \\ & \mid\ I_k \otimes \mathbf{matrix}_m & \text{where } n = km \\ & \mid\ I_k \otimes_\ell \mathbf{matrix}_m & \text{where } n = km \end{array}$$



(a) $A_n \cdot B_n$. (b) $I_2 \otimes A_2$. (c) D_4.

(d) $P_4 = L^4_2$. (e) $A_2 = \mathrm{DFT}_2$.

(f) $\mathrm{DFT}_4 = L^4_2(I_2 \otimes \mathrm{DFT}_2)L^4_2 T^4_2(I_2 \otimes \mathrm{DFT}_2)L^4_2$.

Fig. 1. Examples of formula elements and their corresponding combinational datapaths.

$$| \ \mathbf{base}_n$$

$$\mathbf{base}_n ::= D_n = \mathrm{diag}(d_0, \ldots, d_{n-1}) \mid P_n \mid A_n$$

This is a subset of the signal processing language (SPL) used in Spiral, a program generator for software implementations of linear transforms [Püschel et al. 2005; Xiong et al. 2001].

Next, we explain the language and illustrate the combinational hardware interpretation of its elements in Figure 1.

Matrix product. A matrix formula can be decomposed into a product (line 1) or iterative product (line 2) of matrix formulas. Figure 1(a) illustrates: if $y = (A_n B_n) \cdot x$, then input vector x first is transformed by B_n, then by A_n to produce output vector y. Note that the matrices are applied to the data vector from *right*-to-*left*.

The iterative product $\prod$ is important because it allows the explicit specification of repeated stages that are identical or related. Section 3.2 discusses how we exploit this repetition to represent explicit datapath reuse.

Tensor product. Line 3 shows that a matrix formula can include the *tensor* (or Kronecker) product of matrices, which was defined in Equation (6). Most importantly, if I_k

is the $k \times k$ identity matrix, then

$$I_k \otimes A_m = \begin{bmatrix} A_m & & & \\ & A_m & & \\ & & \ddots & \\ & & & A_m \end{bmatrix}$$

is a $km \times km$ matrix that is *block-diagonal* (omitted entries are zero). If $I_k \otimes A_m$ is interpreted as dataflow, the A_m matrix is applied k times in parallel to consecutive regions of the km-length input vector. Figure 1(b) illustrates this for $k = m = 2$.

Line 4 illustrates an indexed version of the tensor product as

$$I_k \otimes_\ell A_m^{(\ell)} = \begin{bmatrix} A_m^{(0)} & & & \\ & A_m^{(1)} & & \\ & & \ddots & \\ & & & A_m^{(k-1)} \end{bmatrix},$$

where ℓ is a parameter of the matrices $A_m^{(\ell)}$. This construct has the same dataflow as $I_k \otimes A_m$, but it allows for variation among the A blocks.

The regularity captured by $I_k \otimes A_m$ is critical to the hardware generation and optimization method that we propose. In Section 3.1 we explain how this regularity is exploited for explicit datapath reuse.

Diagonal matrices. Our language contains three classes of *base matrices*. First are *diagonal matrices*, written as

$$D_n = \operatorname{diag}(d_0, d_1, \ldots, d_{n-1}) = \begin{bmatrix} d_0 & & & \\ & d_1 & & \\ & & \ddots & \\ & & & d_{n-1} \end{bmatrix},$$

where the d_ℓ are constants and omitted entries are zero. Multiplying a vector by D_n means scaling each element by one constant. Figure 1(c) illustrates this for D_4. A diagonal matrix can be parameterized as in $D_{n,\ell}$, where the values of the n constants depend on ℓ.

Permutations. The second class of base matrices consists of *permutations*, or fixed reorderings of data elements. P_n denotes a permutation on n points. This is implemented as a reordering of data by shuffling wires. Figure 1(d) illustrates this for a particular permutation on four points. We use P_n to represent an arbitrary n point permutation; other letters will be used to define specific permutations in transform algorithms. A permutation can be viewed as a matrix or as a mapping on the indices of data elements. For example, L_m^{mn} represents the stride-m permutation on mn points, which permutes data according to

$$L_m^{mn} :\ in + j \mapsto jm + i, \quad 0 \le i < m,\ 0 \le j < m. \tag{8}$$

Figure 1(d) illustrates L_2^4.

Another important permutation is the base-r digit reversal permutation on n points: R_r^n.

$$R_r^{r^t} = \prod_{\ell=0}^{t-1} (I_{r^{t-\ell-1}} \otimes L_r^{r^{\ell+1}}). \tag{9}$$

If $r = 2$, this permutation is called the *bit reversal* permutation.

$$\mathrm{DFT}_{r^t} = \left(\prod_{\ell=0}^{t-1} L_r^{r^t}\,(I_{r^{t-1}} \otimes \mathrm{DFT}_r) C_{r^t}^{(\ell)}\right) R_r^{r^t}, \tag{11}$$

$$\mathrm{DFT}_{r^t} = L_r^{r^t} \left(\prod_{\ell=0}^{t-1} (I_{r^{t-1}} \otimes \mathrm{DFT}_r) \left(I_{r^\ell} \otimes \left(A_{r^{t-\ell}}^{(\ell)} Q_{r^{t-\ell}}^{(\ell)}\right)\right)\right) R_r^{r^t}, \tag{12}$$

$$\mathrm{DFT}_n = L_{r^k}^n (I_{s^\ell} \otimes \mathrm{DFT}_{r^k}) L_{s^\ell}^n T_{s^\ell}^n (I_{r^k} \otimes \mathrm{DFT}_{s^\ell}) L_{r^k}^n, \quad n = r^k s^\ell, \tag{13}$$

$$\mathrm{DFT}_n = B_{n\times m}\, \mathrm{IDFT}_m\, E_m\, \mathrm{DFT}_m\, F_{m\times n}. \tag{14}$$

$$\text{2D-DFT}_{n\times n} = \prod_{\ell=0}^{1} \left((I_n \otimes \mathrm{DFT}_n) L_n^{n^2}\right). \tag{15}$$

$$\mathrm{RDFT}_n = K_n \left(I_{n/4} \otimes_\ell H_4^{(\ell)}\right) (K_n)^{-1} \overline{\mathrm{DFT}_{n/2}}, \tag{16}$$

$$\mathrm{RDFT}_n = (\mathrm{DFT}_2 \oplus I_{n-2})\, \mathrm{rDFT}_n^{(u)}, \tag{17}$$

$$\mathrm{rDFT}_{2km}^{(u)} = V_{2km}^{(u)} \left(I_k \otimes_\ell \mathrm{rDFT}_{2m}^{(f(u,\ell))}\right) \left(\mathrm{rDFT}_{2k}^{(u)} \otimes I_m\right), \quad u \neq 0, \tag{18}$$

$$\mathrm{rDFT}_{2km}^{(0)} = W_{2km} \left(\prod_{i=0}^{\log_k m} \left(I_m \otimes_\ell \mathrm{rDFT}_{2k}^{(g(i,\ell))}\right) L_{2m}^{2km}\right) L_{km}^{2km}, \tag{19}$$

$$\mathrm{rDFT}_{2km}^{(0)} = W_{2km} \left(\prod_{i=0}^{\log_k m} \left(I_{m/k^i} \otimes X_{2k^{i+1}}^{(i)}\right) \left(I_m \otimes_\ell \mathrm{rDFT}_{2k}^{(h(i,\ell))}\right)\right) L_m^{2km}. \tag{20}$$

$$\text{DCT-2}_{2^k} = \sqrt{\frac{2}{2^k}}\, U_{2^k} \left(\prod_{s=k-1}^{0} G_{2^k}^{(s)} (I_{2^{k-s-1}} \otimes L_{2^s}^{2^{s+1}})\right) K_{2^k}. \tag{21}$$

Fig. 2. Transform algorithms (written in SPL) used in this work.

Computational kernels. The final class of base matrices consists of *computational kernels*. Matrix A_n denotes a generic $n \times n$ computational kernel that takes in n data elements and produces n output elements. Typically, such a kernel is only used when n is small; a combinational datapath is then formed by directly interpreting the matrix as computational elements. One example of such a kernel is

$$\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

which is illustrated in Figure 1(e).

Example. A linear transform algorithm specified in this language can be directly mapped to a combinational datapath. For example, the Cooley-Tukey FFT (fast Fourier transform) [Cooley and Tukey 1965] on four points can be written as

$$\mathrm{DFT}_4 = L_2^4 (I_2 \otimes \mathrm{DFT}_2) L_2^4 T_2^4 (I_2 \otimes \mathrm{DFT}_2) L_2^4, \tag{10}$$

where L is the stride permutation of Equation (8) and T is a diagonal matrix of "twiddle factors" as specified in Johnson et al. [1990]. Note that this formula is equivalent to the matrix representation seen in Equation (7).[1] Figure 1(f) shows the corresponding combinational datapath for this algorithm. Again, note that the formula is read from right-to-left.

2.3. Algorithms

Although SPL is restricted in the type of computations it can represent, it is able to specify a very wide range of transform algorithms. In Figure 2, we show (without complete specification) the algorithms used within this work. In these formulas, K, L, Q, R, S, U, V, W, and X are permutation matrices; A, B, C, E, and F are diagonal matrices; H and G are basic blocks; and $f(), g()$, and $h()$ are indexing functions. Full specifications of these algorithms are available in the following citations or in Milder [2010].

First, Algorithms (11)–(14) compute the discrete Fourier transform. Respectively, they are the Pease FFT [Pease 1968], an iterative variant of the Cooley-Tukey FFT [Cooley and Tukey 1965], the mixed-radix FFT (also derived from Cooley and Tukey [1965]), and the Bluestein FFT [Bluestein 1970].

Algorithm (15) is the row-column algorithm for computing the two-dimensional DFT as several one-dimensional DFTs; it is derived from the definition of the transform.

Algorithms (16)–(20) are used in computing the real discrete Fourier transform, or RDFT. First, as shown in Voronenko and Püschel [2009], Algorithm (16) computes RDFT_n by first computing $\mathrm{DFT}_{n/2}$ followed by a postprocessing step. Algorithm (17), which uses helper Algorithms (18)–(20), is a native RDFT algorithm that we have derived using the framework in Voronenko and Püschel [2009].

Lastly, Algorithm (21) is an algorithm for the discrete cosine transform of type 2 given in Nikara et al. [2006].

3. FORMULA-BASED DATAPATH REPRESENTATION

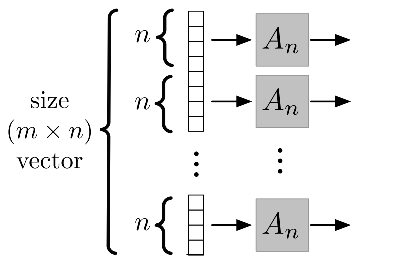
The language described in the previous section (SPL) can represent a wide range of algorithms but not implementation decisions, such as the *sequential reuse* of datapath components, where one computational block is used multiple times during the computation of a single transform. Sequential reuse is necessary since combinational datapaths, such as Figure 1(f), are too large for practical implementation for all but the smallest transform sizes. This section describes our extension to the SPL formula language that allows it to represent two types of sequential reuse that are relevant for hardware designs. The result is a hardware language we call *Hardware SPL* (or HSPL) that enables explicit datapath description at the formula level. Later, we show how this extended language drives our proposed compilation system, allowing the description and generation of a wide trade-off space. The work described in this section was presented in part in Milder et al. [2008].

3.1. Streaming Reuse

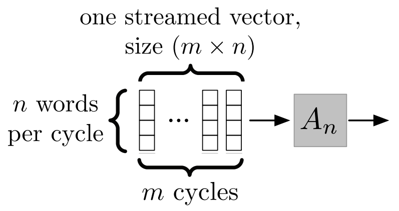
Streaming reuse restructures a datapath with parallelism into a smaller datapath where data elements *stream* through the system over multiple cycles.

As shown in Section 2.2, the tensor product $I_m \otimes A_n$ results in m data-parallel instantiations of block A_n, as shown in Figure 3(a). However, other structures can also perform the same computation. For example, the tensor product can be interpreted as *reuse in time* (rather than parallelism in space). Then, one can build a single instance of block A_n and reuse it over m consecutive cycles (Figure 3(b)). Rather than all mn input points entering the system concurrently, they now stream in and out at a rate of n per cycle. We call this streaming reuse and write $I_m \otimes^{\mathrm{sr}} A_n$. We define *streaming width* to indicate the number of inputs (or outputs) that enter (or exit) a section of datapath during each cycle. Here, the streaming width is n.

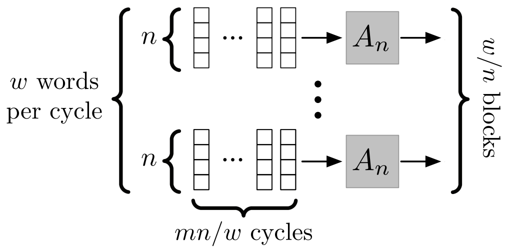
[1]Note that the leftmost matrix in Equation (7) corresponds to $L^4_2(I_2 \otimes \mathrm{DFT}_2)L^4_2$.



(a) No streaming reuse (width = mn): $I_m \otimes A_n$.



(b) Full streaming reuse (width = n): $I_m \otimes^{\text{sr}} A_n$.



(c) Partial streaming reuse (width = w): $I_{mn/w} \otimes^{\text{sr}} \left(I_{w/n} \otimes A_n\right)$.

Fig. 3. Examples of streaming reuse.

The two interpretations of $\otimes$ can be nested in order to build a partially parallel datapath that is reused over multiple cycles (Figure 3(c)). In general, $I_m \otimes A_n$ can be written as $I_{mn/w} \otimes^{\text{sr}} (I_{w/n} \otimes A_n)$, which results in a datapath with a streaming width of w, consisting of w/n parallel instances of A_n, reused over mn/w cycles (w is a multiple of n; $w \leq mn$). Increasing the streaming width increases the datapath's cost and throughput roughly proportionally.

Indexed tensor product. Streaming reuse can also be applied to the *indexed tensor product*

$$I_m \otimes_\ell A_n^{(\ell)},$$

where A is an $n \times n$ matrix parameterized by ℓ. When this construct is streamed with width $w = n$, one computational block is built that is capable of performing all $A_n^{(\ell)}$, $0 \leq \ell < m$. In the worst case, this can lead to an overhead of roughly a factor of m, if the m instances cannot share logic between them. However, often this hardware block can be simplified. For example, an algorithm for DCT-2 presented in Nikara et al. [2006] contains the term

$$(I_{2^{k-1}} \otimes_\ell M_2^{(s,\ell)}),$$

where

$$M_2^{(s,\ell)} = \begin{bmatrix} 1 & 0 \\ -\mu_s(\ell) & 1 \end{bmatrix}, \quad \mu_s(\ell) = \begin{cases} 0, & \ell \mod 2^s = 0, \\ 1, & \ell \mod 2^s \neq 0 \end{cases}.$$

When streaming reuse is applied to this formula, only two possible instances of M need to be considered: one that is equivalent to I_2 and one that performs one subtraction. In hardware, this can be realized as one subtractor and one multiplexer, controlled based on ℓ.

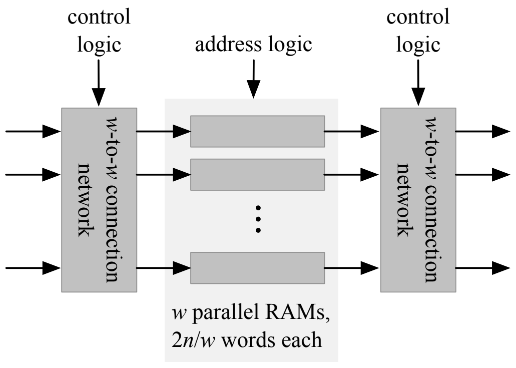

control logic
address logic
control logic
w-to-w connection network
w-to-w connection network
w parallel RAMs, 2n/w words each

Fig. 4. Streaming permutation on n points with streaming width w.

Diagonal matrices. Diagonal matrices scale each data element by a constant. An n point diagonal can easily be streamed with width w by building w multipliers, each holding its own lookup table of n/w constants.

We can write this as a streaming reuse formula by reformulating the expression into one that uses $I \otimes_\ell$. So, a diagonal D_n with streaming width w can be expressed as

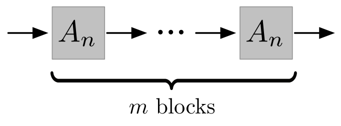
$$I_{n/w} \otimes_\ell^{\text{sr}} D_w^{\prime(\ell)}, \tag{22}$$

where $D_w^{\prime(\ell)} = \operatorname{diag}(d_{\ell w}, \ldots, d_{(\ell+1)w-1})$ is the ℓ-th $w \times w$ subdiagonal of D_n. In hardware, the variable ℓ is provided by a counter to determine which submatrix to use for each cycle.

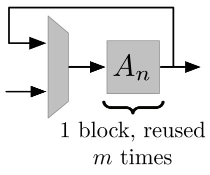
This is implemented in hardware by splitting the D_n into w partial lookup tables. Each holds the values that will be multiplied with data from one of the w input ports. This representation also allows for easy simplification in the case of suitable repeated patterns within the diagonal matrix. For example, every other element of the twiddle diagonal T_2^n is equal to 1. So, when this diagonal is streamed (with a two-power streaming width w), half of the corresponding multipliers will always multiply by 1. Using this representation makes this situation (and similar ones) easy to recognize within the compiler, allowing for automatic removal of unneeded multipliers.

Permutation matrices. Implementing streaming reuse of permutation matrices is a difficult problem, but one that is crucially important for construction of linear transform hardware. A streaming permutation matrix describes how data must be routed between computational elements in space and in time. That is, a permutation streamed at width w must take in an n point input vector at a rate of w words per cycle. It must buffer the data stream, perform a reordering over the entire set of n data points, and then stream the data vector out without stalling. In general, designing these systems is a difficult problem because it requires the designer to partition the incoming data into multiple banks of memory while guaranteeing that there will be no conflicts (when two words must be read from or written to the same memory at the same time).

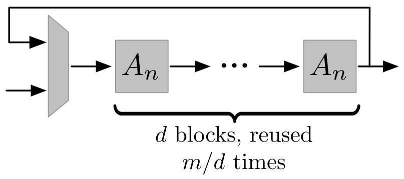
We have developed two methods for automatically implementing streaming permutations that fit our requirements [Püschel et al. 2009; Milder et al. 2009]. The former applies to a subset of streaming permutations but produces designs that are optimal in the complexity of the control logic and in the number of switching elements (for a subset of its supported problems). The latter produces designs that are more expensive, but the technique is applicable to any streaming permutation. Figure 4 shows an example of a permutation on n points with streaming width w, using the former technique.



(a) No iterative reuse (depth = m): $\prod_m A_n$.



(b) Full iterative reuse (depth = 1): $\prod_m^{\mathrm{ir}} A_n$.



(c) Partial iterative reuse (depth = d): $\prod_{m/d}^{\mathrm{ir}} \left(\prod_d A_n\right)$.

Fig. 5. Examples of iterative reuse.

Other approaches, such as the FIFO-based methods for data permutation and routing seen in Järvinen et al. [2004] or Zapata and Argüello [1992] can be used here instead of our approach. We have currently included an implementation of Järvinen et al. [2004] as an alternative in our framework. (We compare our streaming permutations with other approaches [Püschel et al. 2009].)

In this article, we use the notation $\overrightarrow{P_n}^w$ to represent a permutation on n points P_n streamed with width w. If the streaming width can be inferred from the surrounding formula, the parameter w may be omitted.

3.2. Iterative Reuse

The product of m identical blocks A_n can be written as $\prod_m A_n$. A straightforward interpretation of this is a series of m cascaded blocks, as shown in Figure 5(a). However, the same computation can be performed by reusing the A_n block m times (Figure 5(b)). Now the datapath must have a feedback mechanism to allow the data to cycle through m times. We call this *iterative reuse* and represent it by adding the letters "ir" to the product term: $\prod_m^{\mathrm{ir}} A_n$. By nesting both kinds of product terms, the formula specifies a number of cascaded blocks to be reused a number of times (Figure 5(c)). In general, $\prod_m A_n$ can be restructured into $\prod_{m/d}^{\mathrm{ir}}(\prod_d A_n)$, resulting in d cascaded instances of A_n iterated over m/d times (where m/d is an integer). The term *depth* is used to indicate the number of stages built (here, d).

When a datapath with iterative reuse is built, it is important that the reused portion of the datapath buffers the entire vector, so the "head" of the data vector does not feed back too soon and collide with its own "tail." This is equivalent to requiring that the latency (in cycles) be at least $1/(\text{its throughput in transforms per cycle})$. If the datapath does not naturally have this property, it is necessary to add buffers to increase its latency.

Table I. Formulas for Latency $L(F)$, Throughput $T(F)$, and Approximate Area $C(F)$

Formula F	**Latency $L(F)$**	**Throughput $T(F)$**	**Area $C(F)$**
$F_n = A_n^{(0)} \cdots A_n^{(m-1)}$	$\sum_i (L(A_n^{(i)}))$	$\min(T(A_n^{(i)}))$	$\sum_i (C(A_n^{(i)}))$
$F_n = \prod_k^{\text{ir}} A_n$	$\max\left(\frac{k}{T(A_n)}, k \cdot L(A_n)\right)$	$\min\left(\frac{T(A_n)}{k}, \frac{1}{k \cdot L(A_n)}\right)$	$C(A_n) + C(\textbf{mux})$
$F_{mn} = I_m \otimes A_n$	$L(A_n)$	$T(A_n)$	$m \cdot C(A_n)$
$F_{mn} = I_m \otimes^{\text{sr}} A_n$	$L(A_n)$	$T(A_n)/m$	$C(A_n)$

Dependence on the iteration variable. Iterative reuse can also be applied to the iterative product

$$\prod_{\ell=0}^{m-1} A_n^{(\ell)},$$

where the $n \times n$ matrix $A_n^{(\ell)}$ is parameterized by variable ℓ. If this formula is iteratively reused with depth $d = 1$, then one computational structure capable of performing all of the variants of $A_n^{(\ell)}$ (where $0 \leq \ell < m$) is constructed. In the worst case, m different independent blocks must be built, but often the structure of the algorithm allows these blocks to be simplified by sharing logic or arithmetic units (for example, in constant geometry algorithms such as the Pease FFT [Pease 1968]). This situation is analogous to streaming reuse of the indexed tensor product in Section 3.1.

Diagonal matrices. Diagonal matrices are often used in product terms with a dependence on the iteration variable such that

$$\prod_{\ell=0}^{m-1} D_n^{(\ell)}.$$

This formula can be iteratively reused by storing all mn constants, and adding simple logic to choose from the correct n depending on the value of ℓ. This also works in the streaming case, where it can be written as $\prod I_{n/w} \otimes_k^{\text{sr}} D_w^{\prime(k,\ell)}$.

Permutation matrices. Similar to diagonal matrices, iterative reuse can also be applied to permutations. If streaming reuse is not used, then this simply is an instance of the generic dependence on the iteration variable discussed previously. When both streaming and iterative reuse are used, the formula becomes

$$\prod_{\ell=0}^{m-1}{}^{\text{ir}} \left(\overrightarrow{P_n^{(\ell)}}^{w}\right),$$

and the streaming permutation implementation (discussed in Section 3.1) must be extended to support multiple permutations at different times.

3.3. Formula-Based Hardware Model

Given a matrix formula F, Table I provides formulas for computing the latency $L(F)$ (cycles), the throughput $T(F)$ (in transforms per cycle), and an approximate area cost $C(F)$ from the latency, throughput, and relative to the area of F's submodules. The entries of this table can be recursively used to reason about complex formulas containing streaming and iterative reuse.

3.4. Combining Streaming and Iterative Reuse

Often, transform algorithms contain the substructure $\prod_k (I_m \otimes A_n)$. This structure can utilize both iterative reuse (due to the $\prod$) and streaming reuse (due to $I_m \otimes$), thus allowing for a wide range of hybrid implementations that vary in two dimensions. Formally, this is captured by restructuring this formula to have streaming and iterative reuse of parameterized amounts such that

$$\prod_{\ell_0=0}^{k/d-1}{}^{\text{ir}} \left(\prod_{\ell_1=0}^{d-1} \left(I_{nm/w} \otimes^{\text{sr}} (I_{w/n} \otimes A_n) \right) \right),$$

where d is the depth of the cascaded stages (ranging from 1 to k where k/d must be an integer). Parameter w is the streaming width, a multiple of n.

This parameterized datapath is illustrated in Figure 6. Each stage consists of w/n parallel instances of A_n; d stages are built in series. Let B_{mn} represent this array of dw/n many A_n blocks. Data are loaded into the cascaded stages at a rate of w per cycle over mn/w cycles. The vector feeds back and passes through the series of stages a total of k/d times.

Latency and throughput. Given this combined reuse example, we can use the structure of the formula to analyze the effect of parameters d and w on the datapath. The following are calculations that correspond to evaluating the general rules from Table I for the specific parameters of this example (Figure 6). These calculations assume that B_{mn} (the collective group of A_n blocks) is fully pipelined, that is, its throughput is dictated by the problem size and streaming width only: $T(B_{mn}) = w/mn$. The analysis of latency and throughput for this combined reuse example includes the following two cases.

— *Case* 1: *Iterative reuse*. This case occurs when $d < k$, meaning the data will iterate over the internal block at least two times. As discussed in Section 3.2, the internal block's minimum latency is determined by its throughput. So if $d \cdot L(A_n) < mn/w$, buffers are added until they are equal. Thus, internal block B_{mn} has latency $L(B_{mn}) = \max(mn/w, d \cdot L(A_n))$. The latency of the whole system is k/d times this, giving latency $= \max(mnk/dw, k \cdot L(A_n))$. Because this datapath utilizes iterative reuse, a new vector cannot enter until the previous vector begins exiting the datapath, so the throughput (in transforms per cycle) is the inverse of the latency, $\min(dw/mnk, 1/(k \cdot L(A_n)))$.

— *Case* 2: *No iterative reuse*. This case occurs when $d = k$. Now no iterative reuse is performed; the data only pass through the inner block once. The datapath consists of $d = k$ stages, giving latency $= k \cdot L(A_n)$. Because the data never feeds back, the throughput is limited only by the streaming width, giving throughput $= w/mn$ transforms per cycle.

These equations show that increasing w and d will lead to lower latency and higher throughput in equal weights, until either the data flows so quickly that the latency of the computation dominates $(d \cdot L(A_n) > mn/w)$, or d increases until no iterative reuse is performed $(d = k)$.

Flexibility. Additionally, there is one important difference between the parameters d and w: as w grows, the datapath requires greater bandwidth at its ports, and the cost of interconnect and multiplexers increases. In the simple example considered here, a design with $w = 2$, $d = 16$ will have roughly the same cost and performance as a design with $w = 16$, $d = 2$, but the latter will require a bandwidth of 16 words during loading and unloading phases, while the former only requires two words per cycle (over eight times the number of cycles). For this reason, it is usually preferable to increase d instead of w. However, d must divide k evenly (k is typically the $\log_2$ of the transform

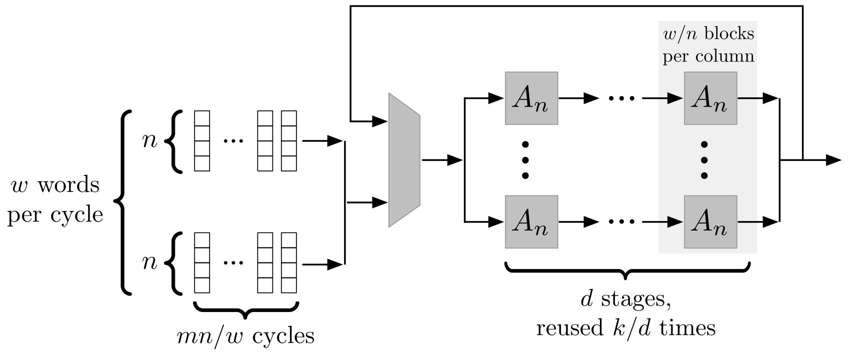


Fig. 6. Combining iterative and streaming reuse: $\prod^{\text{ir}}_{k/d}\left(\prod_d (I_{nm/w} \otimes^{\text{sr}} (I_{w/n} \otimes A_n))\right)$.

size). In many cases, this becomes an all-or-nothing situation in which the only options are $d = 1$ and $d = k$. In those cases, the added flexibility provided by w is important.

Lastly, when the datapath does not employ iterative reuse (i.e., when $d = k$), the designer typically has a wider choice of algorithms, because the internal stages are not required to be uniform. This can lead to substantial cost savings.

3.5. Hardware SPL

Using the concepts explained in this section, we extend SPL to explicitly describe sequential-reuse hardware structures. We call this extended language Hardware SPL (HSPL). HSPL is comprised of SPL, as defined in Section 2.2, with the addition of $I \otimes^{\text{sr}}$, $I \otimes^{\text{sr}}_{\ell}$, $\prod^{\text{ir}}_{\ell}$, and streaming permutations $\overrightarrow{P_n}^{w}$. A formula in SPL describes a transform algorithm, while a formula in HSPL describes a transform algorithm realized as a specific sequential datapath.

In the following section, we show how Spiral automatically translates an SPL formula into HSPL, based on user-specified directives, and how that HSPL formula is then compiled into a register-transfer level Verilog description.

4. AUTOMATIC COMPILATION FROM FORMULA TO DATAPATH

In Section 2, we presented a mathematical language for describing linear transform algorithms (SPL); in Section 3, we extended that language to explicitly specify sequential reuse when mapping to a datapath (HSPL). Now we describe how HSPL is used to drive an automatic compilation framework that maps a transform first to an algorithmic formula, then to a formula that includes the desired streaming and iterative reuse characteristics, and finally to a register-transfer level (RTL) Verilog description.

Figure 7 shows a high-level view of each of the steps in this compilation process. First, a transform enters the system, an algorithm is selected, and a formula representation of that algorithm is produced in SPL. Then, formula rewriting is used to apply iterative reuse and streaming reuse to the formula; the resulting hardware formula (i.e., a formula in HSPL) now has explicit sequential reuse. Lastly, the hardware formula is then translated into a register-transfer level (RTL) Verilog description. We explain each of these steps in the following sections.

4.1. Hardware Directives

This system uses *hardware directives* to include information about the desired features of the hardware implementation to be produced by the compilation framework. Hardware directives are tags placed around a formula or portion of a formula. This work

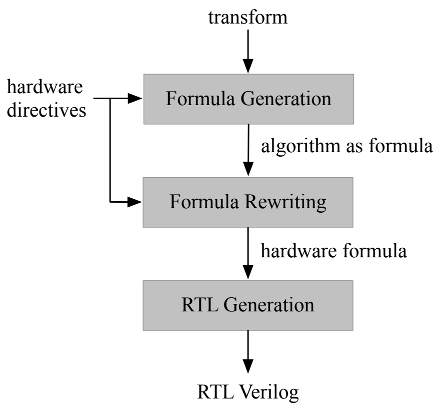


Fig. 7. Block diagram of hardware compilation system.

utilizes two directives. First, the *streaming tag* indicates streaming reuse, as shown by

$$\underbrace{A_n}_{\text{stream}(w)} .$$

This indicates that the contents of A should be restructured such that the resulting hardware formula will be implemented in a block that contains w input and output ports, with data streamed at w elements per cycle, over n/w cycles.

Next, the *depth tag* indicates whether to employ iterative reuse (and of what depth), as shown by

$$\underbrace{A_n}_{\text{depth}(d_0, d_1, \dots)} .$$

This tag indicates that the contents of A_n should be modified so that the top most $\prod$ be restructured to have iterative reuse with depth d_0, its next nested $\prod$ with depth d_1, and so on. Both tags may be used at once.

4.2. Formula Generation: From Transform to Algorithm

The formula generation stage takes a transform of a fixed size as input. It then selects and combines transform algorithms and outputs an SPL formula that specifies the final complete algorithm. This portion of the compilation framework is implemented inside of Spiral [Püschel et al. 2005].

The system represents each algorithm shown in Figure 2 as a parameterized formula. Further, for each algorithm, we include a set of conditions under which it may be applied, as well as guidance rules that choose appropriate algorithms for the requested hardware directives. (We give an example of this below.) Other algorithmic choices, such as the desired radix, can also be specified by the user at this time, which sets parameters seen in the formulas of Figure 2 (e.g., parameter r in Algorithm (11)).

We constructed this stage by expanding Spiral's algorithm specifications to include all algorithms in Figure 2 and added guidance rules where needed.

Example. For example, if the desired transform is a 128-point discrete Fourier transform with radix 2, width 4, and depth 1, that is,

$$\underbrace{\mathrm{DFT}_{128}}_{\text{stream}(4),\ \text{depth}(1)} ,$$

our guidance rules select the Pease FFT Algorithm (11), which is our system's best FFT algorithm when mapping to iterative reuse. This produces

$$\underbrace{\left(\prod_{\ell=0}^{6} L_2^{128}(I_{64}\otimes \mathrm{DFT}_2)C_{128}^{(\ell)}\right) R_2^{128}}_{\text{stream}(4),\ \text{depth}(1)},$$

where C is a diagonal matrix and L and R are permutations, defined in Equations (8) and (9), respectively.

Alternatively, the user could specify depth 7, producing

$$\underbrace{\mathrm{DFT}_{128}}_{\text{stream}(4),\ \text{depth}(7)} \quad\rightarrow\quad \underbrace{L_2^{128}\left(\prod_{\ell=0}^{t-1}(I_{64}\otimes \mathrm{DFT}_2)\left(I_{2^\ell}\otimes\left(E_{2^{7-\ell}}^{(\ell)}\cdot Q_{2^{7-\ell}}^{(\ell)}\right)\right)\right)R_2^{128}}_{\text{stream}(4),\ \text{depth}(7)},$$

which utilizes (12), our system's best algorithm for fully streaming FFTs. (Here E is a diagonal matrix and Q is a permutation. A full definition of this algorithm is available in Milder [2010].)

Lastly, the user could specify radix 4 and depth 3, and the system would first use the Mixed-Radix Algorithm (13) to decompose the DFT_{128} into DFT_{64} and DFT_2. Then it would use the Iterative algorithm to implement DFT_{64}, producing

$$\underbrace{\mathrm{DFT}_{128}}_{\text{stream}(4),\ \text{depth}(3)} \quad\rightarrow \underbrace{L_{64}^{128}(I_2\otimes \mathrm{DFT}_{64})L_2^{128}T_2^{128}(I_{64}\otimes \mathrm{DFT}_2)L_{64}^{128}}_{\text{stream}(4),\ \text{depth}(3)},$$

where DFT_{64} is decomposed as

$$\mathrm{DFT}_{64}\rightarrow L_4^{64}\left(\prod_{\ell=0}^{2}(I_{16}\otimes \mathrm{DFT}_4)\left(I_{4^\ell}\otimes\left(E_{4^{3-\ell}}^{(\ell)}\cdot Q_{4^{3-\ell}}^{(\ell)}\right)\right)\right)R_4^{64}.$$

At this point, all algorithmic choices have been made, and our requested sequential reuse properties are reflected in tags placed around the formula. Next, we will use formula rewriting to automatically convert our tagged formula (SPL) into a hardware formula (HSPL) based on the provided tags.

4.3. Formula Rewriting: Algorithm to Hardware Formula

Next, the SPL formula (representing an algorithm) enters the formula rewriting stage, which takes the formula plus hardware directives and produces a *hardware formula* in HSPL. The output of this stage corresponds directly to a sequential hardware implementation.

The compilation framework accomplishes this restructuring using a *rewriting system* [Dershowitz and Plaisted 2001] that takes in a tagged formula and uses a set of rewriting rules to restructure the formula as dictated by the tag. The hardware formula that is the end result of this stage contains no remaining tags.

In addition to implementing streaming and iterative reuse, rewriting rules within this section of the compiler perform simplifications and optimizations to improve the quality of the generated design.

4.3.1. Rewriting for Streaming Reuse. Table II lists the rewriting rules that the system utilizes for streaming reuse. Each rule takes a formula construct with the stream tag, and either rewrites the formula to implement streaming or pushes the tag inward to be processed at the next level. Each of the rules has a simple explanation.

Table II. Rewriting Rules for Streaming Reuse

name	rule	condition
base-SR	$\underbrace{A_n}_{\text{stream}(n)} \to A_n$	
product-SR	$\underbrace{A_n B_n \cdots Z_n}_{\text{stream}(w)} \to \underbrace{A_n}_{\text{stream}(w)} \cdot \underbrace{B_n}_{\text{stream}(w)} \cdots \underbrace{Z_n}_{\text{stream}(w)}$	
stream-IR	$\underbrace{\prod^{\text{ir}} A_n}_{\text{stream}(w)} \to \prod^{\text{ir}} \underbrace{A_n}_{\text{stream}(w)}$	
stream1	$\underbrace{I_m \otimes A_k}_{\text{stream}(w)} \to I_{mk/w} \otimes^{\text{sr}} \left(I_{w/k} \otimes A_k\right)$	if $mk > w$ and $k \leq w$
stream1-dep	$\underbrace{I_m \otimes_\ell A_k^{(\ell)}}_{\text{stream}(w)} \to I_{mk/w} \otimes^{\text{sr}}_{\ell_0} \left(I_{w/k} \otimes_{\ell_1} A_k^{(\ell_0 \cdot \frac{w}{k} + \ell_1)}\right)$	if $mk > w$ and $k \leq w$
stream2	$\underbrace{I_m \otimes A_k}_{\text{stream}(w)} \to I_m \otimes^{\text{sr}} \underbrace{A_k}_{\text{stream}(w)}$	if $k > w$
stream2-dep	$\underbrace{I_m \otimes_\ell A_k^{(\ell)}}_{\text{stream}(w)} \to I_m \otimes^{\text{sr}}_\ell \underbrace{A_k^{(\ell)}}_{\text{stream}(w)}$	if $k > w$
stream-diag	$\underbrace{D_n}_{\text{stream}(w)} \to I_{n/w} \otimes^{\text{sr}}_\ell D'^{(\ell)}_w$	if $w \mid n$
stream-perm	$\underbrace{P_n}_{\text{stream}(w)} \to \overrightarrow{P_n}^{w}$	if $w \mid n$

Note: Expression $w \mid n$ indicates that w evenly divides n.

—*base-SR*. If the size of a matrix is the same as the desired streaming width, the stream tag is not necessary and can be dropped.
—*product-SR*. If a product of matrices is tagged for streaming, the stream tag is propagated to each individual matrix.
—*stream-IR*. Similarly, a stream tag is propagated into the inner term of an iterative reuse product.
—*stream1 and stream1-dep*. If the size of A is less than or equal to the streaming width, these rules unroll the inner tensor product to match the stream's width. If A depends on index variable ℓ, the same rewriting is performed except the dependence must now change to include two index variables ℓ_0 and ℓ_1.
—*stream2 and stream2-dep*. If the size of A is larger than the size of the stream, the tag is propagated inward, and another rule must restructure A to the right streaming width.
—*stream-diag*. A diagonal to be streamed is rewritten into the streaming diagonal form of Equation (22) as described in Section 3.1.
—*stream-perm*. A permutation to be streamed is placed in a streaming permutation wrapper, and will be implemented in hardware as discussed in Section 3.1.

Table III. Rewriting Rules for Iterative Reuse

name	rule	condition
base-IR	$\underbrace{A_n}_{\text{depth}(d_0,\dots)} \to A_n$	A_k contains no $\prod$
drop-tag-IR	$\underbrace{A_n}_{\text{depth}()} \to A_n$	depth tag empty
product-IR	$\underbrace{\prod_{\ell=0}^{m-1} A_n}_{\text{depth}(d_0,d_1,\dots)} \to \prod_{\ell=0}^{(m/d_0)-1}{}^{\text{ir}} \left(\prod_{k=0}^{d_0} \underbrace{A_n}_{\text{depth}(d_1,\dots)} \right)$	m/d_0 integer
product-IR-dep	$\underbrace{\prod_{\ell=0}^{m-1} A_n^{(\ell)}}_{\text{depth}(d_0,d_1,\dots)} \to \prod_{\ell=0}^{(m/d_0)-1}{}^{\text{ir}} \left(\prod_{k=0}^{d_0} \underbrace{A_n^{(\ell\cdot d_0+k)}}_{\text{depth}(d_1,\dots)} \right)$	m/d_0 integer

4.3.2. Rewriting for Iterative Reuse. Table III lists the rewriting rules used for iterative reuse.

— *base-IR*. If the formula tagged with a depth tag does not contain any product terms, the depth tag is unneeded.
— *drop-tag-IR*. When the depth tag does not have any terms left in its list, it is dropped because it no longer holds any directives.
— *product-IR and product-IR-dep*. When a product term with a depth tag is encountered, iterative reuse is applied with an inner product of the given depth (here, d_0). The first term (d_0) is then removed from the tag, and the tag is propagated inward. Similarly, if the inner formula of a product term has a dependence on the iteration variable, the same process occurs, except the iteration dependence must combine the two product variables ℓ and k.

Like the formula generation stage, this portion of the compilation framework is also implemented inside of Spiral.

4.4. RTL Generation: Hardware Formula to RTL Verilog

The RTL generation stage takes in a hardware formula (HSPL) and produces a synthesizable register-transfer level (RTL) Verilog description of the corresponding datapath. This portion of the compilation framework is partially implemented inside Spiral and partially implemented as a standalone backend that runs alongside Spiral.

The majority of the formula constructs in HSPL are mapped to hardware in a straightforward fashion, as described in Section 3. However, a few constructs present additional challenges, which we describe here.

Basic Blocks. Any portion of the formula without iterative reuse, streaming reuse, or explicit sequential meaning (e.g., streaming permutations) is considered a computational basic block, written A_n or $A_n^{(\ell)}$. These blocks can automatically be mapped into a combinational datapath, as discussed in Section 2.2. When the compiler encounters a basic block, it constructs a hardware datapath and automatically pipelines it by inserting staging registers. (This pipelining can easily be tuned within the compiler.)

Lastly, additional buffers are added if needed to guarantee that corresponding data words reach the output ports together in the same cycle.

When the block includes dependence on an index variable $A_n^{(\ell)}$, the compiler determines the possible values for ℓ, and constructs hardware for each, as well as a counter to provide the values of ℓ as computation progresses. The compiler attempts to simplify the datapaths as much as possible by reducing the dependency on ℓ to the smallest segments possible.

Iterative Reuse. As shown in Figure 5(c), an iterative reuse structure $\prod_{\ell=0}^{k-1}{}^{\mathrm{ir}} A_m$ is built with an input multiplexer and feedback loop. In addition, it is necessary to ensure that the latency through the iteratively reused block A_m is sufficient to prevent the vector's head from colliding with its own tail at the input multiplexer (see Section 3.2). So, the compiler must determine the latency (in cycles) through A_m and (if necessary) add additional buffering.

Other Functionality of the RTL Generation Stage. In addition to translating the hardware formula into an RTL description, the RTL generation stage includes added functionality. For example, latency and throughput (relative to the cycle time) are computed for all blocks and can be reported at the top level or any level below. Other options include basic cost reporting (e.g., counting the numbers and size of RAMs, ROMs, and arithmetic units). In previous work [Milder et al. 2006], we created and calibrated an FPGA area model for a subset of the design space considered here. Such an approach could be extended to produce an accurate area model for the full design space. Lastly, the generator is also able to include basic Verilog testbenches for any design it generates.

4.5. Example: Fast Fourier Transform

This section provides an example of the formula rewriting and compilation used when generating implementations of the discrete Fourier transform.

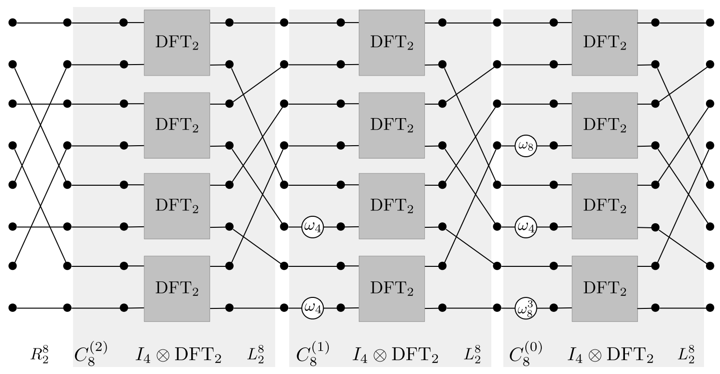
As shown in Algorithm (11), the Pease FFT algorithm with radix r is given by

$$\mathrm{DFT}_{r^t} = \left(\prod_{\ell=0}^{t-1} L_r^{r^t} (I_{r^{t-1}} \otimes \mathrm{DFT}_r) C_{r^t}^{(\ell)} \right) R_r^{r^t},$$

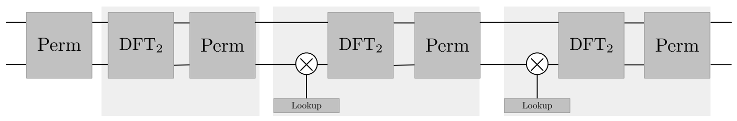
where C is a diagonal matrix, and L and R are permutation matrices. All of its terms can be used with streaming reuse: the permutations and diagonals as described previously, and the tensor product $I_{r^{t-1}} \otimes \mathrm{DFT}_r$ can be restructured to any width w such that w is a multiple of r, and w evenly divides r^{t-1}.

In general, larger values of radix r reduce the computational cost (by reducing the number of multiplications performed). However, this comes at the cost of reduced flexibility, since the streaming width w must be $\geq r$. The evaluations in Section 5 illustrate the effects of varying the radix of this and other algorithms.

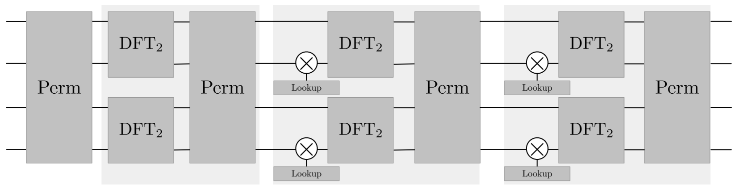
This algorithm is a good candidate for iterative reuse, since it only depends on the product term's index ℓ in the parameter to the diagonal matrix C, which can be implemented as described in Section 3.2. It can be iteratively reused with depth d, where $d \mid t$ (that is, d evenly divides t).



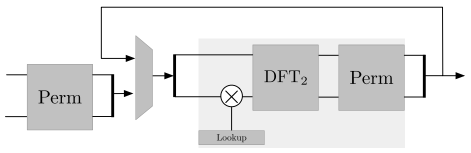
(a) No sequential reuse.



(b) Streaming width $w = 2$.



(c) Streaming width $w = 4$.



(d) Streaming width $w = 2$, depth $d = 1$.

Fig. 8. Pease FFT: DFT_{2^3}.

So this algorithm can be tagged and restructured automatically to have streaming width w and depth d in the following manner.

$$\underbrace{\mathrm{DFT}_{r^t}}_{\text{stream}(w),\ \text{depth}(d)} \rightarrow \left(\prod_{k=0}^{t/d-1}{}^{\text{ir}} \left(\prod_{\ell=0}^{d-1} \overrightarrow{L_r^{r^t}} \left(I_{r^t/w} \otimes^{\text{sr}} \left(I_{w/r} \otimes \mathrm{DFT}_r \right) \right) \cdot \left(I_{n/w} \otimes_m^{\text{sr}} C_w'^{(m,kd+\ell)} \right) \right) \right) \overrightarrow{R_r^{r^t}}.$$

Both permutations used here (stride permutation L and digit-reversal permutation R) are implemented using Püschel et al. [2009] when r is a power of two and Milder et al. [2009] when it is not.

For example, consider DFT_{2^3}. First, the radix 2 Pease FFT (11) gives the formula

$$\left(\prod_{\ell=0}^{2} L_2^8 (I_4 \otimes \mathrm{DFT}_2) C_8^{(\ell)}\right) R_2^8,$$

which is illustrated in Figure 8(a) as hardware with no sequential reuse. The 8 point input vector x enters on the left and flows through the datapath, producing output vector y on the right. (Recall, the formula is read and implemented from right to left.) At the bottom of the diagram, each portion of the datapath is annotated with the matrix that describes its computation. The shaded boxes illustrate the three stages of the product term. Note that $C_8^{(2)} = I_8$, so its section of the datapath performs no computation.

Next, we can apply streaming reuse to this formula. For example, at width $w = 2$:

$$\underbrace{\mathrm{DFT}_{2^3}}_{\text{stream}(2)} \rightarrow \left(\prod_{\ell=0}^{2} \overrightarrow{L_2^8} \left(I_4 \otimes^{\mathrm{sr}} \mathrm{DFT}_2\right) \left(I_4 \otimes_m^{\mathrm{sr}} C_2'^{(m,\ell)}\right)\right) \overrightarrow{R_2^8}.$$

This is illustrated in Figure 8(b). As before, the three stages are enclosed in shaded blocks. Now the permutations are drawn as blocks with two inputs and two outputs, reflecting their streaming width of $w = 2$. The multipliers used for the diagonal matrices now have lookup tables attached, because different constants are needed for different portions of the stream. The input x flows in at a rate of $w = 2$ words per cycle, then flows through input permutation R_2^8, before passing through the three stages and exiting as y. A new input vector can begin entering the system as soon as the previous one has finished loading.

Alternatively, if the user had requested streaming width $w = 4$, we would have

$$\underbrace{\mathrm{DFT}_{2^3}}_{\text{stream}(4)} \rightarrow \left(\prod_{\ell=0}^{2} \overrightarrow{L_2^8} \left(I_2 \otimes^{\mathrm{sr}} (I_2 \otimes \mathrm{DFT}_2)\right) \left(I_2 \otimes_m^{\mathrm{sr}} C_4'^{(m,\ell)}\right)\right) \overrightarrow{R_2^8},$$

which is illustrated in Figure 8(c).

Next, we can apply iterative reuse with depth $d = 1$ and with streaming reuse $w = 2$ such that

$$\underbrace{\mathrm{DFT}_{2^3}}_{\text{stream}(2),\ \text{depth}(1)} \rightarrow \left(\prod_{\ell=0}^{2}{}^{\mathrm{ir}} \overrightarrow{L_2^8} \left(I_4 \otimes^{\mathrm{sr}} \mathrm{DFT}_2\right) \left(I_4 \otimes_m^{\mathrm{sr}} C_2'^{(m,\ell)}\right)\right) \overrightarrow{R_2^8}.$$

Figure 8(d) shows the resulting datapath. Now the three stages are iteratively reused, collapsed into one stage with a feedback path. The multiplier's lookup table is increased to hold the constants needed for all three iterations. The streaming permutations and basic blocks are identical in all three stages, so they can be iteratively reused with no added cost. As before, input vector x flows into the system at a rate of two words per cycle. However, now a new input vector cannot begin streaming in immediately after the previous one; instead it must wait until the first vector has iterated through.

5. EVALUATION

This section presents a set of experiments evaluating transform hardware cores generated by Spiral. The results show that the algorithmic and architectural freedoms described by our formalism yield generated designs that span a wide cost/performance trade-off space. A subset of this space constitutes the Pareto optimal solutions that a designer would consider for applications. We target FPGAs and ASICs using fixed- and floating-point data types. Further, we generate implementations for several transforms, including the DFT, DCT, RDFT, and multidimensional transforms.

Benchmarks. In addition to evaluating trade-offs, we compare our FPGA FFT results with implementations from the Xilinx LogiCore IP library [Xilinx, Inc. 2010] to establish that our baseline designs are comparable with platform-optimized commercial designs. Later, Section 6 discusses how common transform architectures in the literature fit within our design space. For further comparisons, please see Milder et al. [2008], where we compare our designs against our previous work (which uses different hardware generation techniques), and Milder [2010], which contains a comparison of the relative cost and performance of our designs with various architectures in the literature, as well as a commercially available floating point FFT FPGA processor core [4DSP, LLC 2007].

5.1. Methodology

We have implemented the compilation flow described in Section 4 in the following way. The algorithm generation, formula rewriting, and basic-block matrix compilation stages are built inside of Spiral. The tools to implement streaming permutations are implemented as described in Milder et al. [2009] and Püschel et al. [2009]. Lastly, a Verilog backend written in Java takes as input a hardware formula and outputs synthesizable register-transfer level Verilog. The entire compilation flow is initiated and run through Spiral. Then, designs are synthesized targeting FPGA or ASIC as follows.

FPGA. The FPGA evaluations described in this article target the Xilinx Virtex-6 XC6VLX760 or the Xilinx Virtex-5 XC5VLX330 FPGAs. All FPGA synthesis is performed using Xilinx ISE version 12.0, and the area and timing data shown in the results are extracted after the final place and route are complete. Multiple runs were performed using the Xilinx Xplorer script in order to determine the maximum frequency of each design.

We used Xilinx LogiCore single-precision floating-point IP cores for all floating-point experiments, and we used the FPGA's dedicated arithmetic units called DSP slices when possible (fixed-point multiplication and floating-point addition and multiplication).

Memories can be mapped to two types of structures on Xilinx FPGAs: block RAM (BRAM), which are 36kb dedicated hard memories built into the FPGA, or distributed RAM, which is constructed out of the FPGA's programmable logic elements. The Spiral hardware compiler can choose between distributed and block RAM by outputting directives for the Xilinx ISE tool. It allows the user to direct it by providing a threshold (that is, use a BRAM if we will utilize $x\%$ of it), or a budget (i.e., use up to x BRAMs as efficiently as possible). In our experiments, we allowed a budget of 256 BRAMs ($\sim 35\%$ of the Virtex-6). For the designs considered here, this budget is sufficient for the memory requirement of the designs, i.e., no distributed RAM is used for large memories.

ASIC. We target ASIC designs by synthesizing the generated cores using the Synopsys Design Compiler version C-2009.06-SP5, targeting a commercial 65nm standard cell library. Area, timing, and power estimates are obtained from Design Compiler

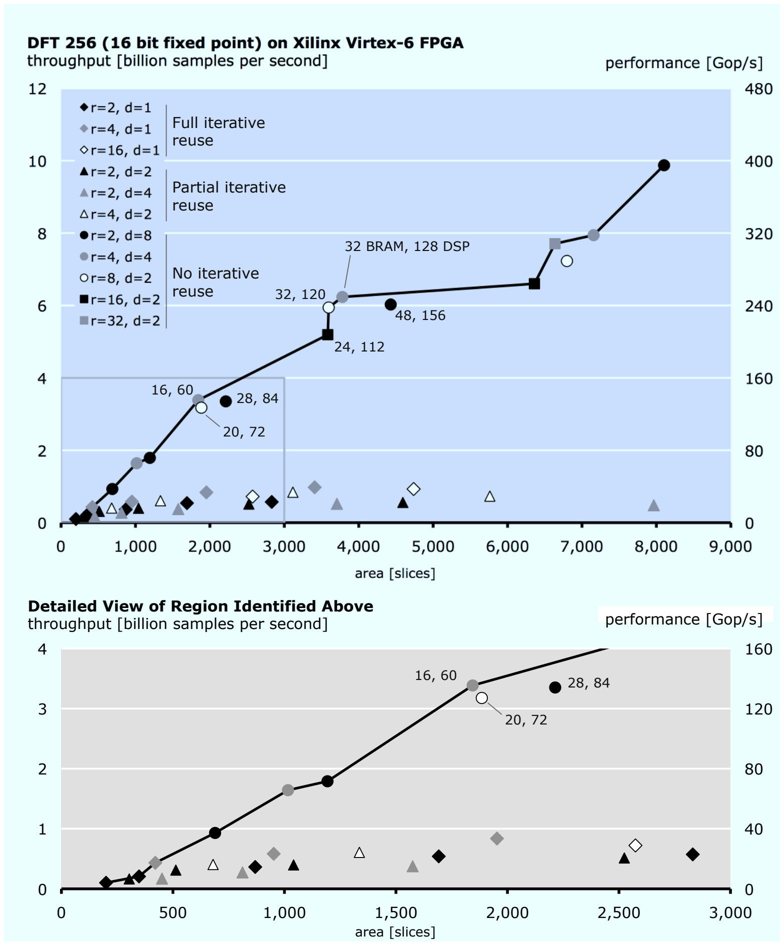


Fig. 9. Generated designs for DFT_{256}, fixed-point, FPGA, throughput versus slices. The Pareto-optimal designs are connected with a line. Data labels in the plot indicate the number of block RAMs and DSP48E1 slices required. The top plot shows results of the full exploration, while the bottom plot is cropped and zoomed to clearly show the designs in the region indicated.

post-synthesis. We use Synopsys DesignWare for arithmetic units, allowing the tool to place pipeline stages within the units automatically.

No memory compiler was accessible for the particular standard cell library used in these evaluations, so we used CACTI version 6.5 [Muralimanohar et al. 2009], a memory estimation tool, to estimate power and area for all required memories.

5.2. Design Space Exploration Example

In order to examine how different design space freedoms (algorithm, radix, iterative reuse, streaming reuse) contribute to our space of designs, this section provides a detailed evaluation of Spiral-generated implementations of DFT_{256} on a Xilinx Virtex-6 FPGA. Figure 9 shows throughput (in billion samples per second) versus area (given in slices) on the Xilinx Virtex-6 FPGA. Given a fixed problem size n, performance (in pseudo-gigaoperations per second, assuming $5n \log_2 n$ operations per DFT_n) is proportional to throughput; the right-hand y-axis shows performance for each design.

Different data markers are used to illustrate the different parameters: an algorithm's radix (basic block size) r and the implementation's iterative reuse (IR) depth d. The diamond-shaped markers indicate designs with full IR, and the triangular markers indicate partial IR. Lastly, designs marked with circles and squares contain no IR. For each series, the streaming width w starts from $w = r$ and goes up to 32. The top plot shows the results of the full exploration, while the bottom plot is cropped to clearly show the designs with lower throughput and area.

As previously discussed, Spiral generates hardware cores for the DFT using different algorithms, depending on the situation. In Figure 9, the algorithm can be determined based on the parameters displayed. First, the Pease FFT algorithm (11) is used when IR is used. So, the black, gray, and white diamonds represent the Pease FFT with radix $r = 2, 4$, and 16, respectively. When IR is not used (represented as circles and squares), Algorithms (12) and (13) are used.

The black line indicates the Pareto optimal set of designs for each graph. These designs are those that give the best trade-off between the two metrics being considered (here, throughput and area). So, if a designer only cares about those two metrics, the points along the line are the only ones to consider. However, other costs are associated with FPGA implementations: memory required (number of block RAMs or BRAMs), and dedicated arithmetic units called DSP48E1 slices. Several points in each graph are annotated with the number of BRAMs (first number) and DSP slices (second number) required.

The set of designs considered covers a wide trade-off space. Here, the fastest design is 76 times faster than the slowest but requires 33 times the number of slices, 16 times the number of BRAMs, and 59 times the number of DSP slices.

The smallest and slowest design is an IR core with depth $d = 1$ generated from the Pease FFT with radix $r = 2$ and streaming width $w = 2$. From there, the black diamonds illustrate the same algorithm, radix, and depth, with increasing streaming width: $w = 2, 4, 8, 16, 32$. As the streaming width increases (following the line of black diamonds), the designs increase in throughput and in cost. Quickly, the radix 2 IR designs become Pareto-suboptimal. The next Pareto optimal point is $r = 2$, $w = 2$, and $d = 2$; now the IR product term has twice the depth. Of the designs with partial IR, only this point contributes to the Pareto front.

Next, the gray diamonds show the $d = 1$ designs from the Pease radix 4 algorithm, with width $w = 4, 8, 16, 32$. The IR radix 4 points contribute just one design to the Pareto front. Following that, the circles and squares represent fully streaming designs (those without IR), which provide the rest of the Pareto front. So, the IR designs provide the small/slow end of the Pareto optimal front, but as they scale larger, their performance per area is quickly eclipsed by the streaming designs.

Further interesting comparisons can be made by examining the behavior of different radices. For example, there are three points near the Pareto front at approximately 3.3 billion samples per second and approximately 2,000 slices. (These are the three highest throughput points visible in the bottom plot.) Each of these designs is fully unrolled and streamed with width $w = 8$; their only difference is in radix r (2, 4, or 8). All three points are similar in throughput and area, however the data labels indicate that they require differing amounts of BRAM and DSP slices. The least expensive of the three is the radix 4 design, which requires only 16 BRAM and 60 DSP slices.

In general, higher radices lead to lower arithmetic and permutation costs, so it may seem counterintuitive that the radix 4 design can provide better results here than radix 8. However, the difference lies in algorithm: at radix 8, the system first must use the Mixed-Radix FFT algorithm (13), which decomposes DFT_{256} into DFT_4 and DFT_{8^2}, but introduces additional overhead (seen in (13) as diagonal matrix T and permutation matrices L). This additional overhead costs more than is saved by using radix 8 for a portion of the computation. The Mixed-Radix algorithm is most useful when the problem size is such that $r = 2$ is the only natural radix supported (for example, Mixed-Radix FFT is necessary to compute DFT_{128} with any radix other than 2).

In this way, we see that the available choices (in algorithm, radix, streaming width, and IR depth) all contribute to the space of Pareto-optimal designs at different cost/performance trade-off points.

5.3. Fast Fourier Transform on FPGA

Next, we show the results of benchmarking implementations of the fast Fourier transform on FPGA with fixed-point and floating-point data types. Figure 10 shows throughput (in billion samples per second) versus area (in slices) of the discrete Fourier transform of size 1024 on the Xilinx Virtex-6 FPGA with 16-bit fixed-point data format (left) and single-precision floating-point (right) data formats. Here, gray diamonds represent Spiral-generated designs across the degrees of freedom discussed in Section 5.2. The solid line connects the Spiral-generated Pareto-optimal points, which include designs with and without iterative reuse and streaming width w up to 32. The black circles represent the four designs from the Xilinx LogiCore FFT IP library [Xilinx, Inc. 2010]. These designs were implemented using LogiCore and evaluated using the same Xilinx tools as the Spiral-generated designs. The plots in the top row of this figure show the full range of points considered, while the bottom row is cropped and zoomed to clearly show the designs in the region near the origin.

The Xilinx LogiCore FFT does not provide a full floating-point implementation. Rather, it provides an implementation where the input and output are given in single-precision floating-point, but internally, computations are performed using a fixed-point implementation with twiddle factors stored as 24- or 25-bit fixed-point values. This results in arithmetic units that are much less expensive to implement than the full floating-point implementations generated by Spiral.

The Xilinx IP cores closely match the cost/performance of the smallest Spiral-generated designs. However, as more resources are allowed to be consumed, the Spiral-generated designs are able to obtain a commensurate increase in throughput. For example, for the fixed-point DFT_{1024} study, the largest/fastest design requires 49 times the slices, 96 times the BRAM, and 107 times the DSP48E1 slices of the smallest/slowest design point but has 132 times the throughput. Compared with the largest/fastest design in the Xilinx LogiCore library, the fastest Spiral-generated design uses 11.8 times the slices, 19.2 times the BRAM, and 18.8 times the DSP slices while providing 16.5 times throughput.

If latency were used as the performance metric instead of throughput, similar trends would also be seen, except the designs without iterative reuse (the higher performance/cost designs) would exhibit a performance penalty because they are optimized for throughput: different portions of the datapath process different data vectors at once. The iterative reuse designs are in this sense optimized for latency-based computation because they only employ a small amount of overlapping of multiple problems. That is, they are able to match the latency of non-iterative reuse designs at lower cost.

Although not presented here, we have repeated these evaluations over multiple transform sizes with similar results. Additionally, in Section 5.5 we evaluate DFT implementations where the problem size is not a power of two.

5.4. Fast Fourier Transform on ASIC

Next, we synthesize Spiral-generated hardware cores targeting a commercial 65nm standard cell library. First, we perform a baseline evaluation of designs with maximum clock frequency. Then, we repeat at lower frequencies and examine the effect on

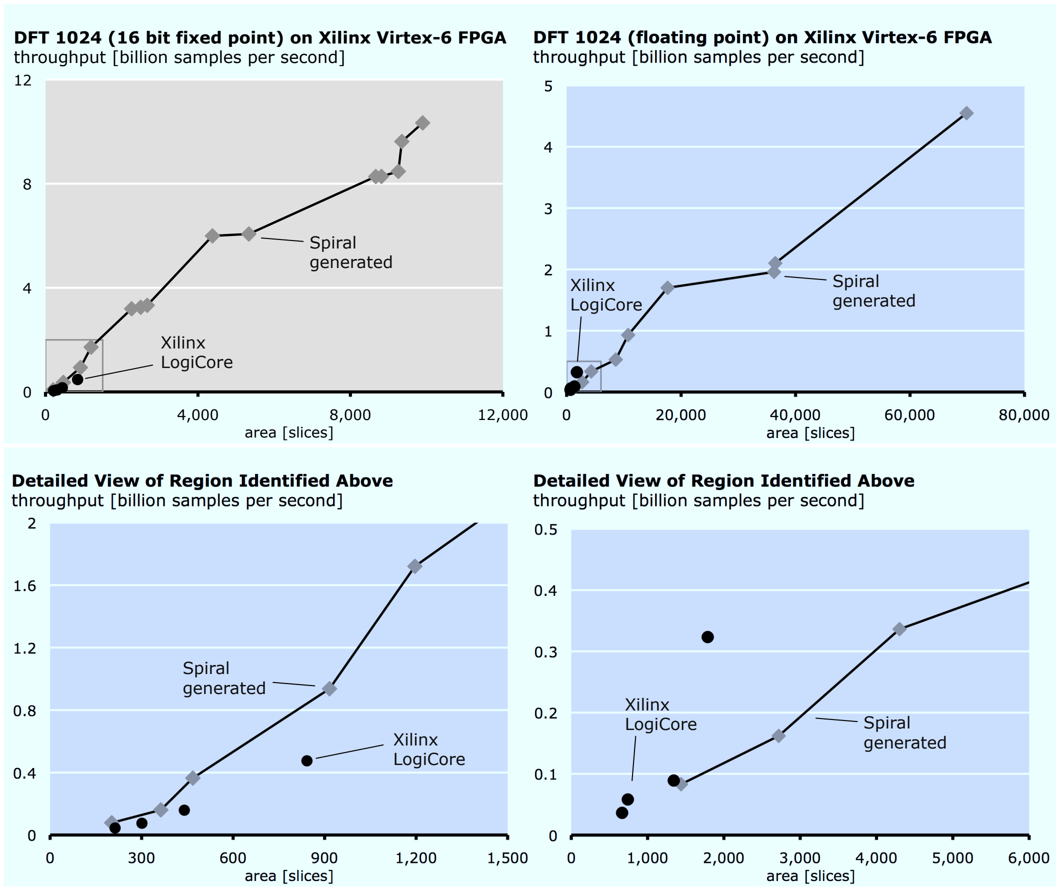


Fig. 10. Throughput versus area, compared with Xilinx LogiCore: DFT_{1024} fixed-point (left) and floating-point (right). The top row shows the full comparison, while the plots in the bottom row are cropped and zoomed to clearly show the designs with low throughput and area.

performance, power, and area. Lastly, we use this flexibility in frequency to implement a set of designs that meet a given throughput target while allowing a trade-off between power and area.

Baseline: Maximum Frequency. First, we synthesize designs with the goal of maximizing the clock frequency. Figure 11 shows throughput versus power (top row) and throughput versus area (bottom row) for DFT_{1024} (left) and DFT_{4096} (right), utilizing 16-bit fixed-point data. Figure 12 repeats this experiment using floating-point data. Two data markers are used in each plot: black diamonds for cores with iterative reuse and gray circles for designs without IR. A gray line is used to highlight the Pareto-optimal set.

Similar to the FPGA results, here the iterative reuse designs provide the least expensive (smallest and lowest power) but slowest designs. For example, for DFT_{1024} with fixed-point data, the fastest design is approximately 100 times faster than the slowest but requires 11 times the area and 24 times the power. In both the ASIC and FPGA experiments, designs with iterative reuse comprise a larger portion of the Pareto-optimal set when the FFT problem size is large.

For both the fixed-point and the floating-point experiments, designs up to streaming width $w = 32$ are considered. Wider designs can easily be generated but become

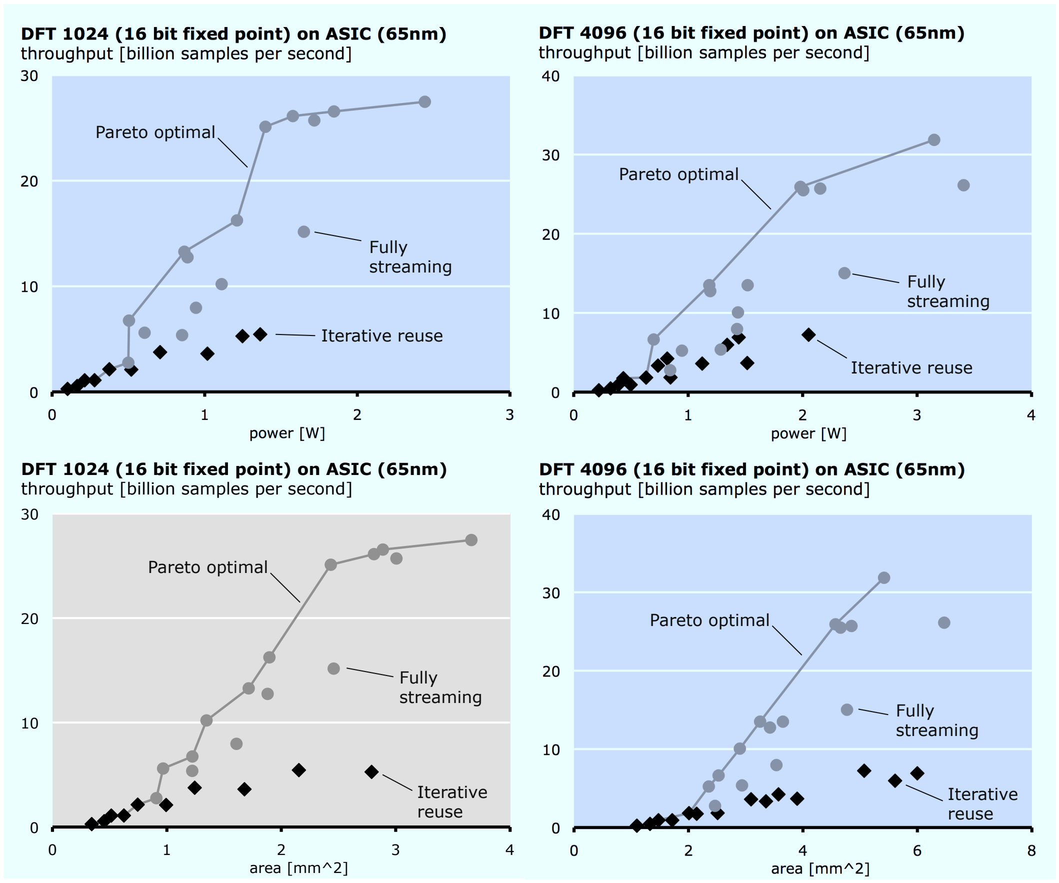


Fig. 11. DFT_{1024} (left) and DFT_{4096} (right), 16-bit fixed-point on 65nm ASIC. Throughput versus power (top row) and throughput versus area (bottom row).

increasingly difficult for Design Compiler to synthesize, increasing the synthesis time and memory requirement, so they are not considered in these experiments.

Reduced Frequency. The previous results showed throughput versus power and area when all designs were synthesized targeting the maximum frequency. However, when energy efficiency is a concern, it is also necessary to consider designs running at lower clock frequencies. Now we explore the trends in performance, power, and area when designs are resynthesized at lower frequencies. In addition to savings that may be provided by the synthesis tool, area and power are further saved by reducing the amount of pipelining necessary in the arithmetic units (as previously discussed in Section 5.1).

Figure 13 (top row) revisits the 16-bit fixed-point DFT_{1024} experiment previously shown in Figure 11. Now the Pareto-suboptimal points are removed, leaving only the Pareto front, shown as black diamonds. Then gray diamonds are used to show how the throughput, power, and area of five designs are effected as the systems are regenerated and resynthesized, targeting 200, 400, and 600 MHz. As the frequency is decreased, throughput and power are reduced commensurately. As seen in the top-right graph of Figure 13, area is reduced only slightly as frequency decreases.

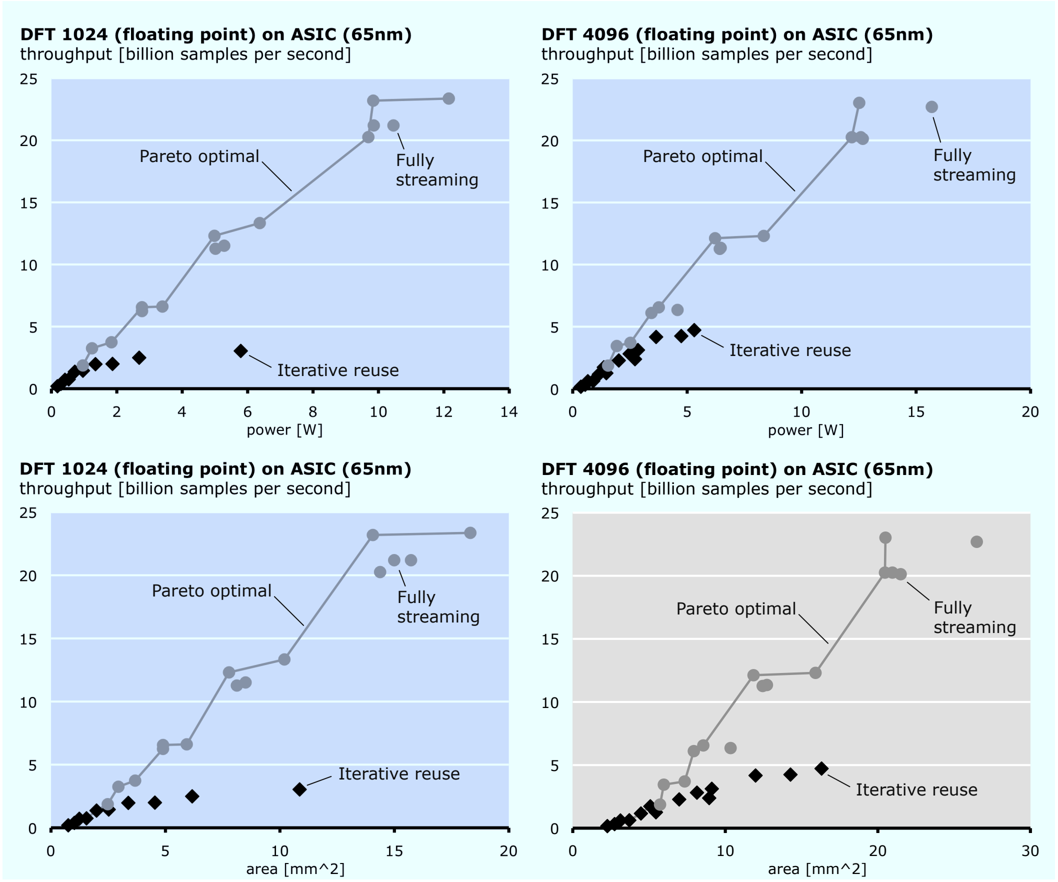


Fig. 12. DFT_{1024} (left) and DFT_{4096} (right), floating-point on 65nm ASIC. Throughput versus power (top row) and throughput versus area (bottom row).

This shows that a large parallel design clocked at a low frequency can be more power-efficient than a smaller design at a high frequency. Further, Spiral can exploit more of its algorithmic freedoms for larger designs, which can use higher radices.

As an example, Figure 13 (top) contains a point from the original Pareto-optimal set at approximately 7 billion samples per second with an area of 0.9 mm^2 consuming approximately 0.5 Watts (while clocked at 1.4 GHz). However, a similar throughput can be obtained using a 2.2mm^2 design that only consumes 0.3 Watts (while clocked at 200 MHz). This type of trade-off between area and power given a fixed performance constraint is explored in the following section.

Although the graphs in this section only demonstrate these effects on five designs for one problem, more evaluations (not shown here) confirm these trends.

Power/Area Optimization Under Throughput Requirement. Often a system requires that a transform be implemented to meet a given throughput requirement. The preceding frequency reduction experiments can be modified to reflect the type of exploration required for such a situation.

Figure 13 (bottom row) again shows fixed-point implementations of DFT_{1024} on 65nm ASIC at several frequencies. However, now the designs are not synthesized at arbitrary frequencies. Instead, each is synthesized at precisely the frequency needed

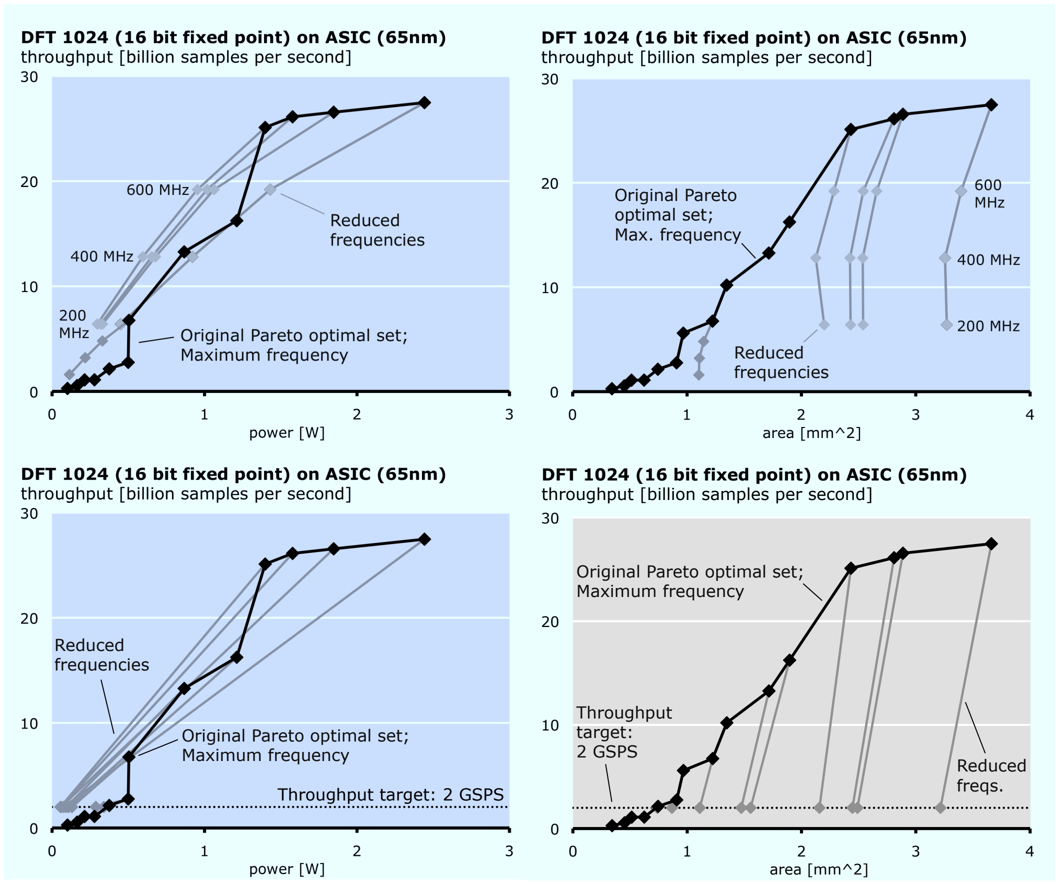


Fig. 13. Top row: DFT_{1024} throughput versus power (left) and area (right) with frequencies reduced to 200, 400, 600 MHz. Bottom row: DFT_{1024} throughput versus power (left) and area (right) with frequencies needed to reach 2 billion samples per second.

to yield a performance of a fixed throughput target: two billion samples per second, illustrated with the dashed line. As shown in the left plot, reducing the frequency of the large designs allows them to precisely reach the target throughput with low power consumption. However, as shown in the right plot of Figure 13, this requires a much larger area.

Repeating this technique for all designs capable of reaching the performance target gives us nine points with equal throughput. Some require less power but more area; others the opposite. This allows a set of power-area trade-offs, with each design having equal throughput. Of the nine designs considered here, five are Pareto-optimal with respect to power and area. Since performance is equivalent for each design, the designer can then choose whichever point best matches the required power and area.

5.5. Other Transforms

Although most of this evaluation has focused on the fast Fourier transform with two-power problem size, we generate designs for other transforms as well. Figure 14 shows throughput versus area on FPGA for four different transforms. First, Figure 14 (top-

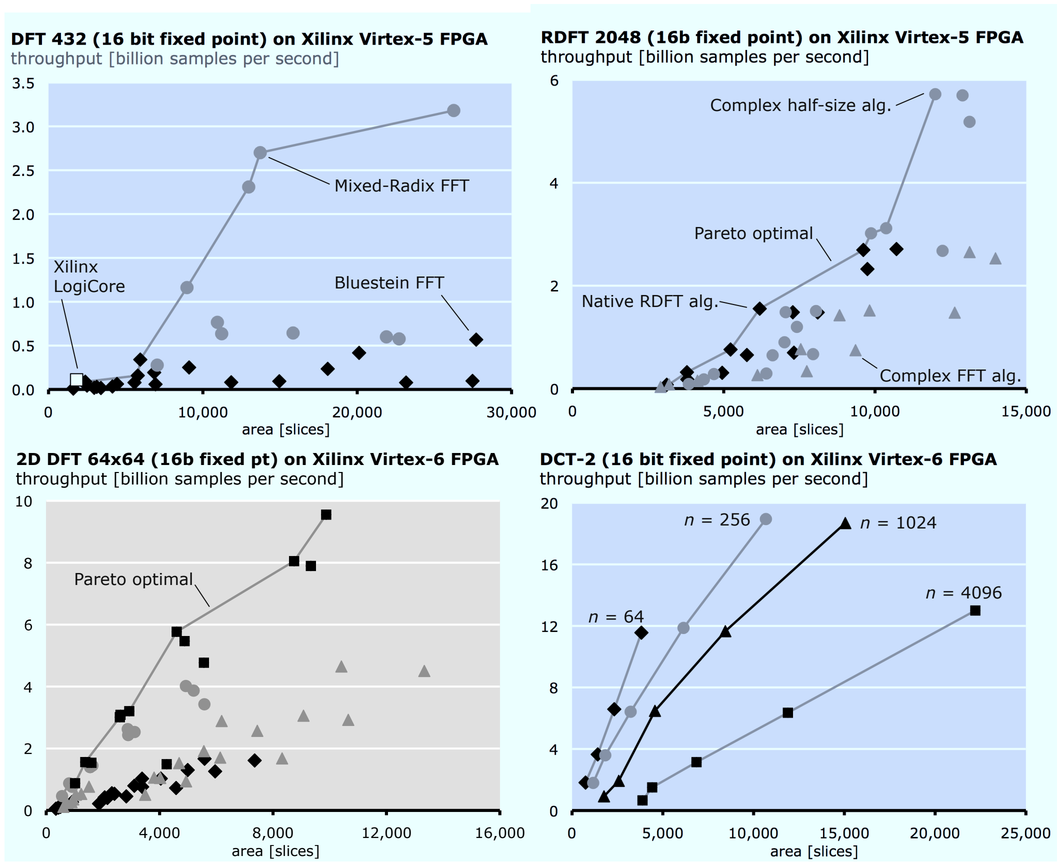


Fig. 14. DFT_{432} (top-left), RDFT_{2048} (top-right), $\text{2D-DFT}_{64\times 64}$ (bottom-left), and DCT-2_n (bottom-right).

left) shows DFT_{432}, which uses the mixed-radix FFT (13) and the Bluestein FFT (14) to compute the DFT with problem size not equal to a power of two.

Next, Figure 14 (top-right) shows results for RDFT_{2048} using our Native RDFT Algorithms ((17)–(20)), the Complex half-size Algorithm (16), and the complex FFT algorithms described previously.

Then, Figure 14 (bottom-left) shows results for the two-dimensional discrete Fourier transform $\text{2D-DFT}_{64\times 64}$, which is computed using the row-column algorithm (15) as well as the previously discussed one-dimensional FFT algorithms. Here, the different data markers indicate different iterative reuse choices.

Lastly, Figure 14 (bottom-right) shows throughput versus area for DCT-2_n for $n = 64, 256, 1024, 4096$ on the Xilinx Virtex-6 FPGA using Algorithm (21) from Nikara et al. [2006]. All points shown here are fully streaming (i.e., no iterative reuse is utilized).

6. RELATED WORK

The formula language (SPL) used in this article has been previously used as a way to describe transform algorithms. For example Van Loan [1992], Püschel et al. [2005], and Johnson et al. [1990] use this type of representation to specify and generate software implementations of the FFT. The formula rewriting system that we use to produce our designs (part of Spiral) is used in automatic parallelization and vectorization of software in Franchetti et al. [2006a, 2006b], respectively.

Although our work (first presented in part in Milder et al. [2008]) was the first to extend SPL into HSPL to support a general class of hardware implementations, variants of this mathematical language have been used in the process of designing special-purpose hardware for the FFT. For example, Kumhom et al. [2000] uses the tensor product-based mathematical language to derive algorithms used in a universal FFT processor that is scalable in the number of processing elements. More recently, Cortés and Vélez [2009] study pipelined architectures, using the tensor language to describe a family of FFT algorithms and providing a hardware interpretation of certain formula constructs. For the discrete cosine transform, Nikara et al. [2006], Takala et al. [2000], and Astola and Akopian [1999] use the tensor formula representation as a way to represent algorithms suitable for pipelined hardware implementation.

Our compilation toolflow has some features in common with general-purpose high-level synthesis (HLS) systems. Some recent papers have explored the benefits of using HLS to implement FFTs in FPGAs or as ASICs. For example, Sukhsawas and Benkrid [2004] implement a pipelined FFT processor in Handel-C, but they do not consider architectural trade-offs (i.e., cost/performance). An ASIC-based 802.11a transmitter (which primarily consists of IFFT) is studied in Dave et al. [2006], which uses Bluespec SystemVerilog to explore different values for streaming width and iterative reuse depth from a few source implementations. (Dave et al. [2006] also examine the trade-off between area and power at a fixed performance target, as we do in Section 5.4). Similarly, Kee et al. [2008] target the FFT on an FPGA, expressing a radix 2 FFT algorithm as a double loop using National Instruments LabVIEW, where each loop can then be unrolled by the tool. These papers make good use of HLS systems to easily exploit some architectural-level trade-offs, but they are unable explore algorithmic options (or the joint algorithm/architecture space).

Many different hardware implementations of linear transforms have been studied (primarily, the FFT). In general, these implementations use a few classes of architectures that are analogous to portions of our hardware parameter space. For example, our designs with full IR and SR are analogous to fully folded special-purpose processors, such as those of Cohen [1976] (FFT) and Takala et al. [2000] (DCT). Three of the Xilinx LogiCore designs that we compare with in Figure 10 are of this class of design. Others (e.g., Zapata and Argüello [1992]) use full IR with parameterizable parallelism (various levels of SR).

Pipelined designs such as those of He and Torkelson [1996] (FFT) and Nikara et al. [2006] (DCT) are very frequently used. These designs are analogous to the architectures we generate that do not employ iterative reuse. Typically, those found in the literature are limited to a streaming width of one word per cycle (while we are interested in varying w to allow cost/performance trade-off). However, some (e.g., Gorman and Wills [1995]) allow scaling of the streaming width. In Figure 10, we compare our designs against those of He and Torkelson [1996], which are used in the Xilinx LogiCore FFT (the fastest/largest of the four LogiCore designs). In Milder [2010], we give a quantitative comparison of the relative cost and performance of several different types of pipelined implementations of the FFT.

7. CONCLUSIONS

This article presented an automated system for generating hardware cores for computing linear signal processing transforms. This system automatically implements hardware designs across a wide space of cost/performance trade-offs, allowing the user to choose the implementation that best fits his or her application. The key to this system lies in the mathematical expression of relevant degrees of freedom, both in the space of transform algorithms and in the space of sequential datapaths.

In this article, we explained the tensor formula language for specifying algorithms, showed our extensions to enable explicit hardware specification, and showed how this mathematical language enables automatic compilation into RTL Verilog. We evaluated our generated designs for several transforms on FPGAs and ASICs, evaluating area, power, and throughput. For each platform and transform, our tool is able to give a large cost/performance trade-off space.

REFERENCES

4DSP, LLC 2007. *4DSP floating point fast Fourier transform V2.6*. 4DSP, LLC.

ASTOLA, J. AND AKOPIAN, D. 1999. Architecture-oriented regular algorithms for discrete sine and cosine transforms. *IEEE Trans. on Signal Process. 47,* 4, 1109–1124.

BLUESTEIN, L. I. 1970. A linear filtering approach to computation of discrete Fourier transform. *IEEE Trans. Audio Electroacoust. 18,* 4, 451–455.

COHEN, D. 1976. Simplified control of FFT hardware. *IEEE Trans. on Acoust. Speech, Signal Process. 24,* 6, 577–579.

COOLEY, J. W. AND TUKEY, J. W. 1965. An algorithm for the machine calculation of complex Fourier series. *Math. Computat. 19,* 90, 297–301.

CORTÉS, A. AND VÉLEZ, I. 2009. Radix r^k FFTs: Matricial representation and SDC/SDF pipeline implementation. *IEEE Trans. on Signal Process. 57,* 7, 2824–2839.

DAVE, N., PELLAUER, M., GERDING, S., AND ARVIND. 2006. 802.11a transmitter: a case study in microarchitectural exploration. In *Proceedings of the ACM/IEEE International Conference on Formal Methods and Models for Codesign*. 59–68.

DERSHOWITZ, N. AND PLAISTED, D. A. 2001. Rewriting. In *Handbook of Automated Reasoning, Vol. 1*, A. Robinson and A. Voronkov, Eds. Elsevier, 535–610.

FRANCHETTI, F., VORONENKO, Y., AND PÜSCHEL, M. 2006a. FFT program generation for shared memory: SMP and multicore. In *Proceedings of ACM/IEEE Conference on Supercomputing*.

FRANCHETTI, F., VORONENKO, Y., AND PÜSCHEL, M. 2006b. A rewriting system for the vectorization of signal transforms. In *Proceedings of High Performance Computing for Computational Science (VECPAR)*. Vol. 4395. Springer, 363–377.

GORMAN, S. F. AND WILLS, J. M. 1995. Partial column FFT pipelines. *IEEE Trans. Circuits Syst. II: Analog Digital Signal Process. 42,* 6, 414–423.

HE, S. AND TORKELSON, M. 1996. A new approach to pipeline FFT processor. In *Proceedings of the International Parallel Processing Symposium*. 766–770.

JÄRVINEN, T. S., SALMELA, P., SOROKIN, H., AND TAKALA, J. H. 2004. Stride permutation networks for array processors. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*. 51–71.

JOHNSON, J. R., JOHNSON, R. W., RODRIGUEZ, D., AND TOLIMIERI, R. 1990. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circuits, Syst. and Signal Process. 9*, 449–500.

KEE, H., PETERSEN, N., KORNERUP, J., AND BHATTACHARYYA, S. S. 2008. Systematic generation of FPGA-based FFT implementations. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*. 1413–1416.

KUMHOM, P., JOHNSON, J., AND NAGVAJARA, P. 2000. Design, optimization, and implementation of a universal FFT processor. In *Proceedings of the 13th IEEE ASIC/SOC Conference*. 182–186.

MILDER, P. A. 2010. A mathematical approach for compiling and optimizing hardware implementations of DSP transforms. Ph.D. dissertation, Electrical and Computer Engineering, Carnegie Mellon University.

MILDER, P. A., AHMAD, M., HOE, J. C., AND PÜSCHEL, M. 2006. Fast and accurate resource estimation of automatically generated custom DFT IP cores. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 211–220.

MILDER, P. A., FRANCHETTI, F., HOE, J. C., AND PÜSCHEL, M. 2008. Formal datapath representation and manipulation for implementing DSP transforms. In *Proceedings of the 45th Annual ACM/IEEE Conference on Design Automation (DAC)*. 385–390.

MILDER, P. A., HOE, J. C., AND PÜSCHEL, M. 2009. Automatic generation of streaming datapaths for arbitrary fixed permutations. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe*. 1118–1123.

MURALIMANOHAR, N., BALASUBRAMONIAN, R., AND JOUPPI, N. P. 2009. CACTI 6.0: A tool to model large caches. Tech. rep. HPL-2009-85, Hewlett-Packard Laboratories.

NIKARA, J., TAKALA, J. H., AND ASTOLA, J. 2006. Discrete cosine and sine transforms—regular algorithms and pipeline architectures. *Signal Process.*, 230–249.

PEASE, M. C. 1968. An adaptation of the fast Fourier transform for parallel processing. *J. ACM 15,* 2, 252–264.

PÜSCHEL, M., MILDER, P. A., AND HOE, J. C. 2009. Permuting streaming data using RAMs. *J. ACM 56,* 2, 10:1–10:34.

PÜSCHEL, M., MOURA, J. M. F., JOHNSON, J., PADUA, D., VELOSO, M., SINGER, B. W., XIONG, J., FRANCHETTI, F., GAČIĆ, A., VORONENKO, Y., CHEN, K., JOHNSON, R. W., AND RIZZOLO, N. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE 93,* 2, 232–275.

SUKHSAWAS, S. AND BENKRID, K. 2004. A high-level implementation of a high performance pipeline FFT on Virtex-E FPGAs. In *Proceedings of the IEEE Symposium on VLSI*. 229–232.

TAKALA, J. H., AKOPIAN, D. A., ASTOLA, J., AND SAARINEN, J. P. P. 2000. Constant geometry algorithm for discrete cosine transform. *IEEE Transactions on Signal Processing 48,* 6, 1840–1843.

VAN LOAN, C. 1992. *Computational Frameworks for the Fast Fourier Transform*. SIAM.

VORONENKO, Y. AND PÜSCHEL, M. 2009. Algebraic signal processing theory: Cooley-Tukey type algorithms for real DFTs. *IEEE Trans. Signal Process. 57,* 1, 205–222.

Xilinx, Inc. 2010. *Xilinx LogiCore IP fast Fourier transform v7.1*. Xilinx, Inc.

XIONG, J., JOHNSON, J., JOHNSON, R., AND PADUA, D. 2001. SPL: A language and compiler for DSP algorithms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 298–308.

ZAPATA, E. L. AND ARGÜELLO, F. 1992. A VLSI constant geometry architecture for the fast Hartley and Fourier transforms. *IEEE Trans. Parallel and Distrib. Syst. 3,* 1, 58–70.

Received July 2011; revised November 2011; accepted November 2011