A Mathematical Approach for Compiling and Optimizing Hardware Implementations of DSP Transforms

Peter A. Milder

August 2010

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy in
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Abstract

Linear signal transforms (such as the discrete Fourier transform) are frequently used in digital signal processing and related fields. Algorithms for computing linear transforms are well-understood and typically have a high degree of regularity and parallelism, making them well-suited for implementation as sequential datapaths on field-programmable gate arrays or application-specific integrated circuits. Nonetheless, transforms are difficult to implement due to the large number of algorithmic options available and ways that algorithms can be mapped to sequential datapaths. Further, the best choices depend heavily on the resource budget and 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.

This thesis proposes the Spiral hardware generation framework, a hardware compilation and optimization tool based on a mathematical formula language. A formula written in this language specifies a particular transform algorithm executed using a particular sequential datapath. This language allows high-level representation of sequential hardware structures that reuse datapath elements multiple times in the computation of a transform. The language accomplishes this by providing a formal connection between structure in the algorithm and sequential reuse in the datapath. This proposed language drives a hardware compilation system that takes as input a problem specification with directives that define characteristics of the desired datapath; the system then automatically generates an algorithm, maps the algorithm to a datapath, and outputs synthesizable register transfer level Verilog.

This thesis evaluates the generated designs when synthesized for field-programmable gate array or application-specific integrated circuit. Its evaluations consider designs across multiple transforms, datatypes, and design goals, and its results show that Spiral is able to automatically provide

a wide tradeoff between cost (e.g., area, power) and performance. This tradeoff space compares well with existing benchmarks, but allows the designer much more flexibility to find the design best suited to his or her needs.

Acknowledgments

First, I would like to thank my academic advisor James C. Hoe for his support and guidance of my graduate studies. His advice and encouragement have made this work possible, and I am very grateful for his time, effort, and expertise.

I also owe a great amount of thanks to Markus Püschel and Franz Franchetti. My work would not be possible without their efforts and ideas, and our collaborations were extremely valuable. I sincerely thank them both for their mentorship and for serving on my thesis committee.

I also thank my committee members Larry Pileggi and David Padua for their valuable feedback and comments, which have helped improve my work and this thesis.

Thank you also to the entire Spiral research team. I owe a special debt of gratitude to Yevgen Voronenko for building so much of the infrastructure that my work depends on and for never tiring of helping me understand it. Thank you also to Jeremy Johnson and José Moura, who have always been encouraging and helpful towards me, and whose efforts to establish Spiral were crucial to all of our work. I have been honored to work on the Spiral team with many graduate students, post-docs, and researchers, especially Volodymyr Arbatov, Christian Berger, Srinivas Chellappa, Bob Koutsoyannis, Marek Telgarsky, Wei Yu, and Qian Yu.

I would also like to thank the other researchers with whom I have collaborated, including Yannis Benlachtar, Rachid Bouziane, Eric Chung, Madeleine Glick, and Robert Killey.

I consider myself lucky to spend my time around so many talented, interesting, and entertaining people: my fellow graduate students. Thank you to those listed above as well as Kristen Dorsey, Brian Gold, Adam Hartman, Yoongu Kim, Peter Klemperer, Ruy Ley-Wild, Brett Meyer, Eriko Nurvitadhi, Michael Papamichael, Aliaksei Sandryhaila, Stephen Somogyi, Evangelos Vlachos, Kai Yu, and many more.

I thank the Department of Electrical and Computer Engineering and its staff, especially Reenie Kirby, Matt Koeske, and Elaine Lawrence.

Lastly, I would like to thank my family for their love and support. Thank you to Lauren Heller, my parents Gary and Frances Milder, and my sister Amy Milder.

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

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

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. Algorithms for computing linear transforms are highly structured and regular, and they exhibit large amounts of parallelism. For these reasons, algorithms in this domain are well-suited for hardware implementation as a sequential datapath on a field-programmable gate array (FPGA) or application-specific integrated circuit (ASIC). This inherent algorithmic structure leads to a large amount of freedom in the way a given algorithm maps to a datapath. Coupled with the fact that multiple algorithmic options are typically available for a given transform, this means that the combined algorithm/datapath space is 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 reason about both sets of options at once. Moreover, the best choices for both algorithm and datapath structure are highly dependent on the context: an application's 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 algorithms to be executed on a specific type of datapath, and (b) exploring different datapaths to execute a specific type of algorithm. 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, designers resort to using solutions from IP vendors that provide common designs across a few cost/performance tradeoff points.

1.2 Thesis Overview

This thesis addresses these problems by introducing a high-level mathematical framework for automatically generating customized hardware implementations of linear DSP transforms. This system covers a wide range of algorithmic and datapath options and frees the designer from the difficult process of manually performing algorithmic and datapath exploration. The designs produced cover a wide cost/performance tradeoff space and are competitive with good hand-designed implementations. The system's automated nature allows easy exploration of a wide space of options without sacrificing the quality of the resulting design.

The key to 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 language in [1] 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 specific sequential datapath on which to execute it.

This mathematical language drives a full compilation system that begins with a problem specification and produces a synthesizable register-transfer level Verilog description. This system begins by taking as input a linear signal transform of a given size as well as high-level hardware directives that describe qualities 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 maps the hardware formula to a corresponding synthesizable register-transfer level Verilog description.

An important challenge in mapping from formula to sequential datapath lies in the hardware implementation of permutations (fixed reorderings of data, commonly occurring in transform algo-

rithms). Permutations are difficult to perform on streaming data because the system must buffer and reorder data elements distributed over a number of cycles. This thesis includes a flexible parameterized architecture for performing any given streaming permutation as well as a discussion and evaluation of the applicability of previous, less flexible methods.

Additionally, this thesis includes an evaluation of designs produced using the proposed generation framework across several transforms (such as the discrete Fourier transform, discrete cosine transform, and others) and two datatypes (single precision floating point and 16 bit fixed point are evaluated here). Results are obtained by synthesizing for FPGA (Xilinx Virtex-6) and ASIC (targeting a commercial 65nm standard cell library). The FPGA results compare performance with resources required, while the ASIC results compare performance with power and area. Results show that Spiral is able to generate designs across a range of cost/performance tradeoffs.

One application for the designs produced using the proposed system is orthogonal frequency-division multiplexing (OFDM) for optical interconnects. Optical OFDM has recently been studied as a way to reduce the complexity and cost of optical components for short distance links (for example, in data centers) [2] or to improve long-distance transmission performance (telecommunications applications) [3]. Optical OFDM depends heavily on the discrete Fourier transform and imposes a high throughput requirement on it. This thesis includes a study of real-time implementations of optical OFDM, using hardware cores automatically generated by the proposed framework.

1.3 Contributions

To summarize, the contributions of this thesis include:

- a formula-based language for specifying transform algorithms and corresponding sequential datapaths,
- an automatic compilation tool that produces synthesizable register-transfer level Verilog,
- a parameterized, scalable architecture capable of performing any fixed streaming permutation,
- an evaluation of the designs produced using this system, spanning multiple transforms, data types, and platforms, and
- an application study of real-time implementations of optical OFDM using generated designs.

1.4 Organization

This thesis is organized as follows. Chapter 2 describes relevant background material on linear transforms, their algorithms, and algorithmic specification in the aforementioned formula language. Then, Chapter 3 explains the proposed mathematical formula language for describing sequentially reused datapaths. Next, Chapter 4 outlines the automatic compilation process and describes each step from transform to formula to hardware implementation. Chapter 5 discusses one important compilation problem: implementing permutations on streaming data. It presents a proposed flexible solution for any fixed streaming permutation. Chapter 6 examines transform algorithms in this context and illustrates the correspondence between algorithmic and datapath structures. Chapter 7 evaluates the designs produced with this methodology on FPGA and ASIC. Chapter 8 presents an application study where generated cores for the discrete Fourier transform are utilized within real-time systems implementing orthogonal frequency-division multiplexing (OFDM) for optical networks. Lastly, Chapter 9 examines related work and Chapter 10 presents concluding remarks.

Chapter 2

Background: Linear Transforms and Algorithms

This chapter presents background material on linear transforms and fast algorithms for their computation. First, Section 2.1 defines linear transforms and gives relevant examples. Then, Section 2.2 shows how a mathematical formula language can specify transform algorithms and how formulas written in it are directly translated to combinational datapaths. This algorithm-specification language explicitly captures regularity or repetition within algorithms. Later, Chapter 3 will show how this regularity leads to hardware structures with explicit datapath reuse. Lastly, Section 2.3 defines the algorithms considered in this thesis.

2.1 Linear Transforms

A linear transform on n points is defined by a 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 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 represent the row and column of a given element (respectively). Thus, computing a four-point DFT (with input vector x and output vector

y) is the matrix-vector product:

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

Direct computation of a linear transform by definition requires $\mathrm{O}\big(n^2\big)$ arithmetic operations.

2.1.1 Discrete Fourier Transform

The discrete Fourier transform and its inverse are defined:

$$\mathrm{DFT}_n = \left[\omega_n^{k\ell}\right]_{0\le k,\ell<n}, \quad \omega_n = e^{-2\pi j/n} \tag{2.1}$$

$$\mathrm{IDFT}_n = (1/n) \cdot \left[\omega_n^{-k\ell}\right]_{0\le k,\ell<n}. \tag{2.2}$$

Also note that

$$\mathrm{DFT}_n = (\mathrm{DFT}_n)^{\mathsf{T}}.$$

That is, the DFT matrix is symmetric.

2.1.2 Two-Dimensional Discrete Fourier Transform

$$\mathrm{DFT}\text{-}2\mathrm{D}_{n\times n} = \mathrm{DFT}_n \otimes \mathrm{DFT}_n, \tag{2.3}$$

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

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

2.1.3 Real Discrete Fourier Transform

When the DFT, which is a complex transform, is performed on real input data (length n), the result is an n point complex output vector with conjugate-even symmetry. Thus, half of its output data is redundant. The real discrete Fourier transform, or RDFT, exploits this property by computing only

the non-redundant outputs, and packing them into a real n point output vector. Depending on the ordering of the output vector, there can be multiple definitions of the RDFT. This thesis uses the following definition for RDFT:

$$\mathrm{RDFT}_n = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & -1 & 1 & -1 & \dots & -1 \\ \begin{bmatrix} \cos \frac{2\pi k\ell}{n} \\ \sin \frac{2\pi k\ell}{n} \end{bmatrix}_{0<k<n/2,\ 0\le\ell<n} & & & & & \end{bmatrix}. \tag{2.4}$$

2.1.4 Discrete Cosine Transform Type 2

There are multiple related *types* of discrete cosine transforms (DCTs) and discrete sine transforms (DSTs). This thesis considers the most common, the DCT of type two, defined:

$$\text{DCT-2} = \left[\cos \frac{k(2\ell+1)\pi}{2n}\right]_{0\le k,\ell<n}. \tag{2.5}$$

The DCT-2 is easily converted to other related transforms:

$$\text{DCT-3}_n = (\text{DCT-2}_n)^{-1} = (\text{DCT-2}_n)^{\mathsf{T}}$$

$$\text{DST-2}_n = \begin{bmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{bmatrix} \cdot \text{DCT-2}_n \cdot \begin{bmatrix} 1 & & & & & \\ & -1 & & & & \\ & & 1 & & & \\ & & & -1 & & \\ & & & & \ddots & \end{bmatrix}$$

$$\text{DST-3}_n = (\text{DST-2}_n)^{-1} = (\text{DST-2}_n)^{\mathsf{T}}$$

The algorithms that are later presented for DCT-2 can easily be converted to DCT-3, DST-2 and DST-3 in this way.

2.2 Formula Representation of Transform Algorithms

Fast transform algorithms enable the computation of an n point transform using $\mathrm{O}\big(n \log n\big)$ arithmetic operations. The term "fast Fourier transform" refers to such an algorithm for computing the discrete Fourier transform. 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,

$$\mathrm{DFT}_4 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.6}$$

Thus, computing $y = \mathrm{DFT}_4 \cdot x$ is equivalent to multiplying the data vector by each matrix from right to left. A fast algorithm decomposes an $n \times n$ transform matrix into a product of sparse matrices ($\mathrm{O}\big(\log n\big)$ many), each which requires $\mathrm{O}\big(n\big)$ operations. Thus, multiplying the input vector by each of these sparse matrices requires $\mathrm{O}\big(n \log n\big)$ operations.

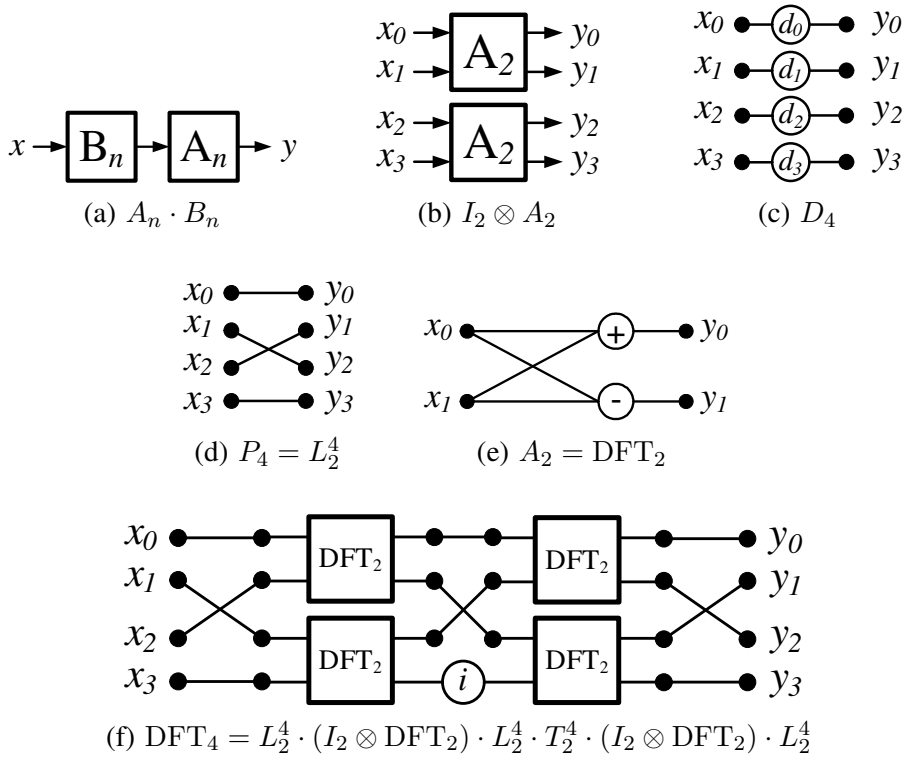
The Kronecker product formalism developed in [1] uses linear algebra concepts to mathematically represent fast transform algorithms by capturing the structure within the sparse matrices of the decomposition. This formalism can serve as the basis for a language that represents transform algorithms as *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:

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

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

This language is a subset of the signal processing language (SPL) used in Spiral, a program generator for software implementations of linear transforms [4, 5].



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

Figure 2.1: Examples of formula elements and their corresponding combinational datapaths.

Next, the following subsections explain each portion of the language, and illustrate the combinational hardware interpretation of each term (in Figure 2.1).

2.2.1 Matrix product

A matrix formula can be decomposed into a product (line 1) or iterative product (line 2) of matrix formulas. Figure 2.1(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. Chapter 3.2 discusses how this repetition is exploited to represent explicit datapath reuse in the proposed hardware compilation and optimization framework.

2.2.2 Tensor product

Line 3 shows that a matrix formula can include the *tensor* (or Kronecker) product of matrices. The $\otimes$ operator is the *tensor product operator*, defined:

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

I_k is the $k \times k$ identity matrix, so

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

a $km \times km$ matrix that is *block-diagonal*. When $I_k \otimes A_m$ is interpreted as dataflow, the A_m matrix is applied k times in parallel to consecutive regions of the km-length input vector. Figure 2.1(b) illustrates this for $k = 2$, $m = 2$.

Line 4 illustrates an indexed version of the tensor product:

$$I_k \otimes_\ell A^m_\ell = \begin{bmatrix} A^m_0 & & & \\ & A^m_1 & & \\ & & \ddots & \\ & & & A^m_{k-1} \end{bmatrix},$$

where the index variable ℓ parameterizes the A matrix.

Although this language only supports the tensor product where the left term is the identity matrix, simple identities rewrite some other structures into this form. For example:

$$A_m \otimes I_k = L^{km}_m (I_k \otimes A_m) L^{km}_k, \tag{2.7}$$

where L is the stride permutation matrix defined in (2.8), below. Further,

$$\begin{aligned} B_k \otimes A_m &= (B_k \otimes I_m)(I_k \otimes A_m) \\ &= L^{km}_k (I_m \otimes B_k) L^{km}_m (I_k \otimes A_m). \end{aligned}$$

The regularity expressed by $I_k \otimes A_m$ is critical to the hardware generation and optimization method proposed in this work. Section 3.1 shows how the regularity represented by this formula is exploited to represent explicit datapath reuse.

2.2.3 Diagonal matrices

This language contains three classes of *base matrices*. First are *diagonal matrices*, written D_n. A diagonal matrix is one that contains non-zero elements only along its main diagonal:

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

where the d_ℓ are constants. Multiplying a vector by D_n results in scaling each element by one constant. Figure 2.1(c) illustrates this for D_4. This thesis uses D_n to represent any generic diagonal matrix of size n. A diagonal matrix can be defined with a parameter, for example D^n_ℓ, where ℓ can be the index variable of an indexed tensor product or iterative product.

2.2.4 Permutations

The second class of base matrix is comprised of *permutations*, or fixed reorderings of data elements. P_n denotes a permutation on n points. As combinational logic, this is implemented as a re-ordering of data by shuffling wires. Figure 2.1(d) illustrates this for a particular permutation on four points. This thesis uses P_n to represent an arbitrary n point permutation; other letters will be used to define other 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^n_m represents the stride-by-m permutation

on n points, which permutes data according to

$$L^n_m : \ i \cdot (n/m) + j \mapsto j \cdot m + i, \quad 0 \le i < m, \ 0 \le j < n/m, \tag{2.8}$$

where $in + j$ represents the given output index, and $jm + i$ is the corresponding input index. For example,

$$L^8_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

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

$$R^{r^t}_r = \prod_{\ell=0}^{t-1} (I_{r^{t-\ell-1}} \otimes L^{r^{\ell+1}}_r). \tag{2.9}$$

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

2.2.5 Computational kernels

The final class of base matrices is comprised 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 formed by directly implementing a matrix-vector multiplication. One example of such a kernel is

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

which is illustrated in Figure 2.1(e). Computational kernels can also be parameterized by index variables: A^n_ℓ.

2.2.6 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) [6] can be written as

$$\mathrm{DFT}_{mn} = L^{mn}_m \cdot (I_n \otimes \mathrm{DFT}_m) \cdot L^{mn}_n \cdot T^{mn}_n \cdot (I_m \otimes \mathrm{DFT}_n) \cdot L^{mn}_m, \tag{2.10}$$

where L^{mn}_m is the stride permutation (2.8), T^{mn}_n is a diagonal matrix of "twiddle factors" (as specified in [7]):

$$T^{mn}_n = \bigoplus_{k=0}^{m-1} \left(\bigoplus_{\ell=0}^{n-1} \omega^{k\ell}_{mn} \right), \quad \omega_n = e^{-2\pi i/n}, \tag{2.11}$$

and $\bigoplus$ is the direct sum operator. So,

$$\begin{aligned} T^4_2 &= \bigoplus_{k=0}^{1} \left(\bigoplus_{\ell=0}^{1} \omega^{k\ell}_4 \right) = \bigoplus_{k=0}^{1} \left(\omega^{k\cdot 0}_4 \oplus \omega^{k\cdot 1}_4 \right) = \omega^{0\cdot 0}_4 \oplus \omega^{0\cdot 1}_4 \oplus \omega^{1\cdot 0}_4 \oplus \omega^{1\cdot 1}_4 \\ &= \mathrm{diag}(1, 1, 1, i). \end{aligned}$$

Figure 2.1(f) shows the corresponding combinational datapath for this algorithm when $n = 2$ and $m = 2$. Again, note that the matrices are applied from right to left.

2.3 Transform Algorithms

This section uses the mathematical language defined in Section 2.2 to specify the transform algorithms considered in this thesis. Later, Chapter 6 will explain how these algorithms fit within the proposed hardware/algorithm specification language, and Chapter 7 will evaluate implementations of the algorithms.

2.3.1 Discrete Fourier Transform

$\mathrm{O}\big(n \log n\big)$ algorithms for the DFT are often called fast Fourier transforms, or FFTs. Note that the DFT is symmetric, so alternate versions of these algorithms may be obtained by transposing the final formula, using property $(AB)^{\mathsf{T}} = B^{\mathsf{T}} A^{\mathsf{T}}$. Additionally, these algorithms can be converted into algorithms for the IDFT by taking the complex conjugate and scaling as defined in (2.2).

Cooley-Tukey FFT. The Cooley-Tukey FFT (fast Fourier transform) algorithm [6] recursively decomposes a DFT of size mn into DFTs of size m and n. It is given by

$$\mathrm{DFT}_{mn} = L^{mn}_m \cdot (I_n \otimes \mathrm{DFT}_m) \cdot L^{mn}_n \cdot T^{mn}_n \cdot (I_m \otimes \mathrm{DFT}_n) \cdot L^{mn}_m, \tag{2.12}$$

where L and T are as defined previously.

Pease FFT. The Pease FFT [8] is an iterative algorithm for the DFT that can be derived from the Cooley-Tukey FFT (as shown in [9]). It is given by

$$\mathrm{DFT}_{r^t} = \left(\prod_{\ell=0}^{t-1} L^{r^t}_r \cdot (I_{r^{t-1}} \otimes \mathrm{DFT}_r) \cdot C^{r^t}_\ell \right) \cdot R^{r^t}_r, \tag{2.13}$$

where C is a diagonal matrix of twiddle factors:

$$C^{r^t}_\ell = L^{r^t}_{r^{t-\ell-1}} \cdot (I_{r^\ell} \otimes T^{r^{t-\ell}}_{r^{t-\ell-1}}) \cdot L^{r^t}_{r^{\ell+1}},$$

where L, R and T are as defined previously.

Here, the DFT of size r^t is computed by first reordering data, then passing through t stages, each which scales data, performs parallel DFT_r computations, and then reorders the data with $L^{r^t}_r$.

Parameter r is called the *radix*. Higher radices give fewer stages, but each stage is more complicated. As the radix increases, the arithmetic cost of the algorithm decreases, because fewer non-trivial multiplications are performed (although the improvement diminishes as r increases).

A key characteristic of the Pease FFT is that the data vector is accessed in the same order in each of the t iterations. In the formula, this can be seen in the fact that only the diagonal matrix C depends on iteration index ℓ. Thus, the only change from stage to stage is in the constants the data vector is scaled by.

Figure 2.2 illustrates the dataflow for the Pease algorithm for DFT_{2^3}:

$$\left(\prod_{\ell=0}^{2} L^8_2 \cdot (I_4 \otimes \mathrm{DFT}_2) \cdot C^8_\ell \right) \cdot R^8_2.$$

The formula is read from right to left, but the dataflow is drawn from left to right. So, Figure 2.2 takes in input vector x on the left, passes it through permutation R^8_2, then through three stages

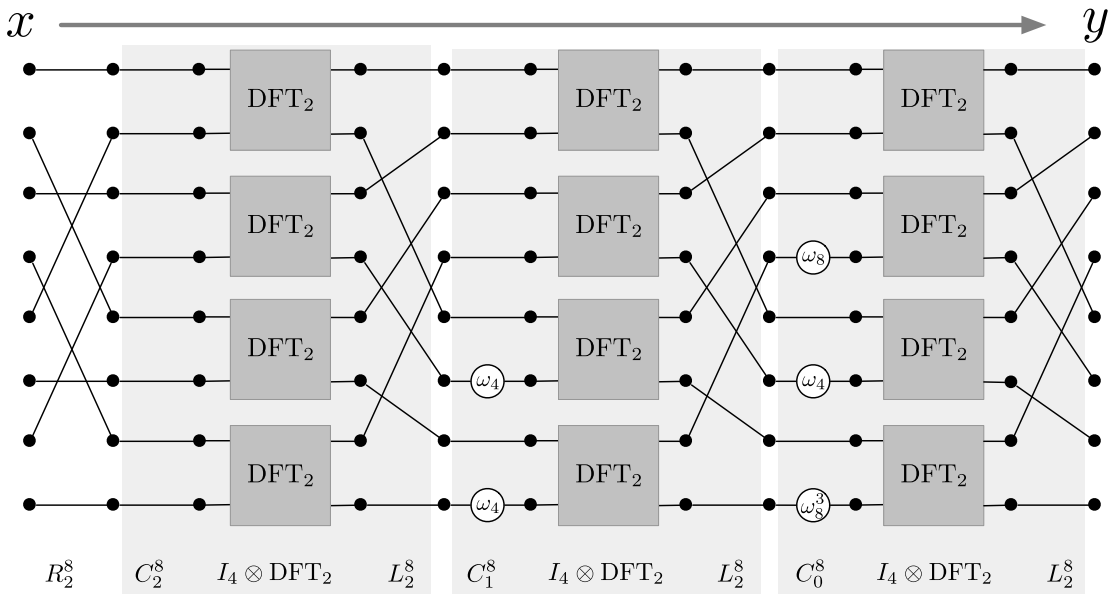


Figure 2.2: Pease FFT dataflow for DFT_{2^3}.

(indicated with shaded background). Each stages performs a scaling by a diagonal matrix C, four parallel DFT_2 computations, and permutation L^8_2. Note that multiplications by 1 are omitted from the diagram, and that $C^8_0 = I_8$.

Iterative Cooley-Tukey FFT. Similar to the Pease FFT, the Iterative Cooley-Tukey algorithm can also be derived from (2.12), but it results in a different structure.

$$\mathrm{DFT}_{r^t} = \left(\prod_{\ell=0}^{t-1} (I_{r^\ell} \otimes \mathrm{DFT}_r \otimes I_{r^{t-\ell-1}}) \cdot (I_{r^\ell} \otimes T^{r^{t-\ell}}_{r^{t-\ell-1}}) \right) \cdot R^{r^t}_r, \tag{2.14}$$

where all matrices are as previously defined. By utilizing the property given in (2.7) and restructuring terms, this algorithm can be expressed

$$\mathrm{DFT}_{r^t} = L^{r^t}_r \left(\prod_{\ell=0}^{t-1} (I_{r^{t-1}} \otimes \mathrm{DFT}_r) \left(I_{r^\ell} \otimes \left(E^{r^{t-\ell}}_\ell \cdot Q^{r^{t-\ell}}_\ell \right) \right) \right) R^{r^t}_r, \tag{2.15}$$

where E is a diagonal matrix of twiddle factors:

$$E^{r^{t-\ell}}_\ell = L^{r^{t-\ell}}_{r^{t-\ell-1}} \cdot T^{r^{t-\ell}}_{r^{t-\ell-1}} \cdot L^{r^{t-\ell}}_r,$$

and Q is a product of two stride permutations:

$$Q_\ell^{r^{t-\ell}} = L_{r^{t-\ell-1}}^{r^{t-\ell}} \cdot (I_r \otimes L_r^{r^{t-\ell-1}}).$$

This thesis will utilize the restructured algorithm (2.15), and refer to it as the Iterative FFT. This algorithm is similar to the Pease FFT: both compute DFT_{r^t} using t stages, each stage consisting of a permutation, a scaling by constants, and parallel DFT_r computations. However, there is one important difference: in each iteration, the Pease algorithm performs a permutation on r^t points and multiplies the vector by an r^t point diagonal matrix. In contrast, the permutation and diagonal matrix in this algorithm grow *smaller* as the stages progress. This means in stage ℓ, the algorithm performs r^ℓ parallel permutations on $r^{t-\ell}$ points. Chapter 6 will show how these differences lead to a different type of hardware implementations than are obtained from the Pease algorithm.

Figure 2.3 illustrates the dataflow for the Iterative FFT algorithm for DFT_{2^3}:

$$\mathrm{DFT}_{2^3} = L_2^8 \left(\prod_{\ell=0}^{2} (I_4 \otimes \mathrm{DFT}_2) \left(I_{2^\ell} \otimes \left(E_\ell^{2^{3-\ell}} \cdot Q_\ell^{2^{3-\ell}} \right) \right) \right) R_2^8.$$

Again, the formula is applied from right to left, while the diagram takes its input x on the left side and produces its output y on the right. First, the data vector is permuted by R_2^8. Then, it goes through three stages, separated with shaded boxes. Note that the first stage consists only of one column of $I_4 \otimes \mathrm{DFT}_2$, because E_2^2 and Q_2^2 are both equal to I_2 and thus do not require hardware.

Mixed-radix FFT. The mixed-radix algorithm (based on (2.12)) breaks down a DFT of size $r^k s^\ell$ into multiple DFTs of sizes r^k and s^ℓ:

$$\mathrm{DFT}_{r^k s^\ell} = L_{r^k}^{r^k s^\ell} \cdot (I_{s^\ell} \otimes \mathrm{DFT}_{r^k}) \cdot L_{s^\ell}^{r^k s^\ell} \cdot T_{s^\ell}^{r^k s^\ell} \cdot (I_{r^k} \otimes \mathrm{DFT}_{s^\ell}) \cdot L_{r^k}^{r^k s^\ell}. \tag{2.16}$$

Then, the DFT_{r^k} and DFT_{s^ℓ} matrices are decomposed using radix r and s algorithms such as (2.13) or (2.15).

Bluestein. The Bluestein FFT [10] is a convolution-based algorithm for any problem size n. It is typically used when the problem size makes it difficult or impossible to perform with the other algorithms. The Bluestein algorithm performs the DFT by scaling the input vector and convolving it with pre-computed coefficients. The convolution of two n length signals can be performed as

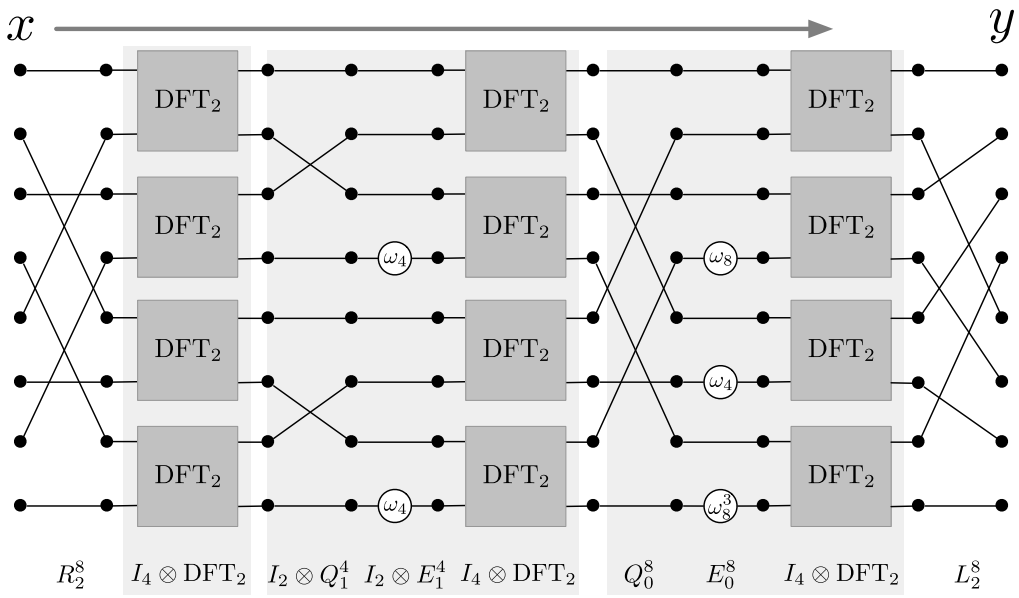


Figure 2.3: Iterative FFT dataflow for DFT_{2^3}.

point-wise multiplication of length $m > 2n-1$ in the frequency domain. This allows a DFT of any given size to be computed using DFTs of two-power size, at the expense of additional operations. This algorithm is given by:

$$\mathrm{DFT}_n = D^{(2)}_{n\times m} \cdot \mathrm{IDFT}_m \cdot D^{(1)}_m \cdot \mathrm{DFT}_m \cdot D^{(0)}_{m\times n}, \tag{2.17}$$

where $m = 2^{\lceil \log_2(n) \rceil + 1}$ is the smallest power of two greater than $2n-1$, and the D matrices are diagonal matrices that scale the vector by constant values. $D^{(0)}_{m\times n}$ and $D^{(2)}_{n\times m}$ are rectangular matrices, so in addition to scaling the data, $D^{(0)}_{m\times n}$ extends the input data vector from n points to m points (by zero padding), and $D^{(2)}_{n\times m}$ shortens the output data vector from m points to n points (by discarding unneeded data).

2.3.2 Two-Dimensional Discrete Fourier Transform

Using the definition of the two-dimensional DFT and a property of the tensor product, the *row-column* algorithm can be derived:

$$\begin{aligned}\mathrm{DFT}\text{-}2\mathrm{D}_{n\times n} &= (I_n \otimes \mathrm{DFT}_n)\cdot L_n^{n^2}\cdot (I_n \otimes \mathrm{DFT}_n)\cdot L_n^{n^2} \\ &= \prod_{\ell=0}^{1}\left((I_n \otimes \mathrm{DFT}_n)\cdot L_n^{n^2}\right)\end{aligned} \tag{2.18}$$

Then, one-dimensional DFT algorithms such as (2.13), (2.15), or (2.16) can be used to compute the DFT_n.

Other algorithms for computing multi-dimensional DFT not considered in this thesis include the vector-radix algorithm [11] and the Dimensionless FFT algorithm [12].

2.3.3 Real DFT

Complex half-size Real DFT. An RDFT of size n can be computed using a complex DFT of size $n/2$ and a post-processing step, as shown in [13]:

$$\mathrm{RDFT}_n = \left((K_2^{\prime n} \otimes I_2)\cdot \left(I_{n/4} \otimes_\ell A_4(\ell)\right)\cdot ((K_2^{\prime n})^{-1} \otimes I_2)\right)\cdot \overline{\mathrm{DFT}_{n/2}}, \tag{2.19}$$

where

$$A_4(0) = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$A_4(\ell \neq 0) = \mathrm{diag}(1,1,1,-1) \cdot (F_2 \otimes I_2) \cdot \begin{bmatrix} 1/2 & & & \\ & 1/2 & & \\ & & c & -s \\ & & s & c \end{bmatrix} \cdot L^4_2 \cdot \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \end{bmatrix},$$

$$c = \cos(2\pi \cdot \ell/n)/2, \qquad s = \sin(2\pi \cdot \ell/n)/2,$$

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

$$K'^{km}_m = (I_{k+1} \oplus J_{k-1} \oplus I_{k+1} \oplus J_{k-1} \ldots) L^{km}_m,$$

J_n is the $n \times n$ identity matrix with columns reversed, and $\overline{\mathrm{DFT}_{n/2}}$ converts an $n/2 \times n/2$ complex matrix into a $n \times n$ real matrix by replacing each complex element $a + bi$ with

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix}.$$

This algorithm works by packing the n real data as $n/2$ complex data, performing a complex $\mathrm{DFT}_{n/2}$ using the DFT algorithms presented above in Chapter 2.3.1. Then it permutes data by $(K^{-1} \otimes I_2)$, performs one column of computational blocks (A), and finally permutes the data again to output data.

One way to reduce the cost of this implementation is to merge the rightmost permutation stage $(K^{-1} \otimes I_2)$ with the output permutation on the DFT algorithm. For example, the iterative Cooley-Tukey algorithm (2.15) contains a stride permutation L on its output side, which can be merged directly. Or, the Pease algorithm (2.13) can be transposed, moving the R permutation to the output side (noting that $R^n_r = (R^n_r)^{\mathsf{T}}$).

Although this algorithm has higher arithmetic cost over the native RDFT algorithms presented next, it is useful because all optimizations for the DFT can be immediately applied.

Native RDFT Algorithm: Cooley-Tukey-type. Based on the framework in [13], three algorithms for the RDFT can be defined, similar in structure to the Cooley-Tukey, Pease, and Iterative Cooley-Tukey FFTs (as shown in (2.12), (2.13), and (2.15), respectively). All three algorithms use a helper transform $\mathrm{rDFT}(u)$, where

$$\mathrm{RDFT}_n = (F_2 \oplus I_{n-2}) \cdot \mathrm{rDFT}_n(0).$$

First, a recursive algorithm is defined that will be used as the basis for two iterative algorithms.

$$\mathrm{rDFT}_{2km}(u) = \langle K_m^{2km} \mid K_m'^{2km} \rangle_u \left(\bigoplus_{j=0}^{k-1} \mathrm{rDFT}_{2m}(r_k(u,j)) \right) (\mathrm{rDFT}_{2k}(u) \otimes I_m) \tag{2.20}$$

$$r_k(u,j) = \begin{cases} j/2k, & u = 0, \\ (u + \lfloor j/2 \rfloor)/k, & u \neq 0 \text{ and } j \text{ even}, \\ (1 - u + \lfloor j/2 \rfloor)/k, & u \neq 0 \text{ and } j \text{ odd} \end{cases}$$

$$K_m^{2km} = ((I_k \oplus J_k \oplus I_k \oplus J_k \dots) L_m^{km}) \otimes I_2,$$

$$K_m'^{2km} = ((I_k \oplus Z_k \oplus I_k \oplus Z_k \dots) L_m^{km}) \otimes I_2,$$

$$Z_k = (I_1 \oplus J_{k-1}),$$

$$\langle A \mid B \rangle_i = \begin{cases} A, & i \neq 0 \\ B, & i = 0 \end{cases},$$

where J is I with the column order reversed. The base cases are given by:

$$\mathrm{rDFT}_4(0) = (F_2 \otimes I_2)\,,$$

$$\mathrm{rDFT}_4(u \neq 0) = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & -1 \end{bmatrix} \cdot \mathrm{rDFT}_4(0) \cdot \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & c & -s \\ & & s & c \end{bmatrix} \cdot L^4_2,$$

$$c = \cos(\pi u), \quad s = \sin(\pi u).$$

Native RDFT Algorithm: Constant Geometry. By left-expanding and applying identities to algorithm (2.20), the following constant geometry RDFT algorithm can be derived. Let the problem size be $2km$, where $m = k^p$, $p \geq 1$. Let $q = \log_k m$. The algorithm's radix is $2k$.

$$\mathrm{rDFT}_{2km}(0) = V^{2km}_{k,m} \left(\prod_{i=0}^{\log_k m} \left(I_m \otimes_\ell \mathrm{rDFT}_{2k}(s_{k,q}(0,0,f_{k,m}(i,\ell)))\right) L^{2km}_{2m} \right) L^{2km}_{km}, \qquad (2.21)$$

where

$$f_{k,m}(i,\ell) = \ell \mod m/k^i,$$

$$s_{k,q}(d,u,f) = \begin{cases} u, & d = q \\ r_k(s_{k,q}(d+1,u,f), \lfloor (f \mod k^{d+1})/k^d \rfloor), & d \neq q \end{cases}$$

$$V^{2km}_{k,m} = \left(\prod_{\ell=0}^{\log_k m - 1} \left(K'^{m/k^{(\ell-1)}}_{m/k^\ell} \oplus \left(I_{k^\ell - 1} \otimes K^{m/k^{(\ell-1)}}_{m/k^\ell} \right) \right) \right) \otimes I_2$$

The algorithm begins with a stride permutation, followed by $\log_k m + 1$ stages, each with a permutation and a radix $2k$ computational basic block. This block ($\mathrm{rDFT}_{2k}(...)$) is computed using the base case above when $k = 2$, and using (2.20) otherwise. The structure of this algorithm is called *constant geometry* because the permutation L^{2km}_{2m} is the same in each stage.

Native RDFT Algorithm: Iterative. Also derived from (2.20) is the following iterative algorithm. Let the problem size be $2km$, where $m = k^p$, $p \geq 1$. Let $q = \log_k m$. Again, the algorithm's

radix is $2k$.

$$\mathrm{rDFT}_{2km}(0) = V_{k,m}^{2km} \cdot \left(\prod_{i=0}^{q} \left(I_{m/k^i} \otimes W_i^{2 \cdot k^{i+1}} \right) \cdot \left(I_m \otimes_\ell \mathrm{rDFT}_{2k}(s_{k,q-i}(0,0,\lfloor \ell/k^i \rfloor)) \right) \right) \cdot L_m^{2km}, \tag{2.22}$$

where W is a product of stride permutations:

$$W_{i\neq 0}^{2 \cdot k^{i+1}} = \left(I_k \otimes L_{k^{i-1}}^{2 \cdot k^i} \right) \cdot L_{2k}^{2 \cdot k^{i+1}},$$
$$W_0^{2 \cdot k^{i+1}} = I_{2k},$$

and V and $s()$ are as defined above.

This algorithm is similar to the constant geometry RDFT algorithm (2.21). However, unlike (2.21), this algorithm's permutation W grows more local as the algorithm is performed (that is, as i decreases). So, its different stages have different geometry, but the interconnections do not need to span the entire data vector in all stages. The relationship between this algorithm and the constant geometry RDFT algorithm is similar to the relationship between the Iterative and Pease FFT algorithms (2.15) and (2.13).

2.3.4 Discrete Cosine Transform

An algorithm for the DCT of type 2 that is well-suited for hardware implementation is given in [14].

$$\text{DCT-2}_{2^k} = \sqrt{\frac{2}{2^k}} \cdot U_{2^k} \cdot \left(\prod_{s=k-1}^{0} S_{2^k}^{(s)} (I_{2^{k-s-1}} \otimes L_{2^s}^{2^{s+1}}) \right) P_{2^k}^H. \tag{2.23}$$

Matrices U and P^H represent permutations, and S represents one stage of computation, consisting of a 4 point basic block and a diagonal matrix. The full specification of this algorithm is given below.

The structure of this algorithm is similar to the FFT algorithm given in (2.15). It computes a transform of size n in $\log_2(n)$ stages, each of which reorders data and performs a small computational kernel. The added complexity comes from the M and H matrices, which perform an extra subtraction and data reordering in some basic blocks.

Using the properties of the DCT and DST given in Chapter 2.1.4, this algorithm can easily be used to produce implementations of DCT-3, DST-2, and DST-3. Other similar algorithms can be found in [15] and [16].

Unlike the other algorithms examined in this thesis, this DCT algorithm is defined only for a fixed radix size (two).

Full specification of algorithm (2.23). Matrix S represents one stage of computation, parameterized by stage s:

$$S_{2^k}^{(s)} = (I_{2^{k-1}} \otimes_\ell M_2^{(s,\ell)}) \cdot \mathrm{diag}(g_k(\ell, s)) \cdot (I_{2^k-2} \otimes_\ell H_4^{(s,\ell)}) \cdot (I_{2^{k-1}} \otimes F_2),$$

where

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

$$g_k(\ell, s) = (2^{\mu_s(\lfloor \ell/2 \rfloor)} d(2^{k-s-1} + \lfloor \ell/2^{s+1} \rfloor))^{f_k(\ell,s)}$$

$$d(\ell) = \cos((h_K(\ell - K) + 1/2)\pi/2K), \quad K = 2^{\lfloor \log_2 \ell \rfloor}$$

$$h_1(0) = 0, \quad h_{2N}(2\ell) = h_N(\ell), \quad h_{2N}(2\ell + 1) = 2N - 1 - h_N(\ell)$$

$$f_k(\ell, s) = (\ell \mod 2) + (1 - \tau_0(\ell))(1 - \tau_{k-1}(s))$$

$$\tau_\ell(s) = \begin{cases} 0, & s = \ell \\ 1, & s \neq \ell \end{cases}$$

$$H_4^{(s,\ell)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}^{\mu_{s-1}(\ell)}$$

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Permutations U and P^H are:

$$U_{2^k} = \prod_{\ell=0}^{k-2} (I_{2^{k-\ell}} \oplus (I_{2^{k-2}-2^{k-\ell}} \otimes (I_2 \oplus J_2))) \cdot (I_{2^\ell} \otimes L_{2^{k-\ell-1}}^{2^{k-\ell}}),$$

$$P_{2^k}^H = [n = h_{2^k}(m)]_{m,n=0,1,\ldots,2^k-1},$$

where $h_{2^k}(m)$ is as defined above, and J is the identity matrix with columns reversed.

Chapter 3

Formula-Based Datapath Representation

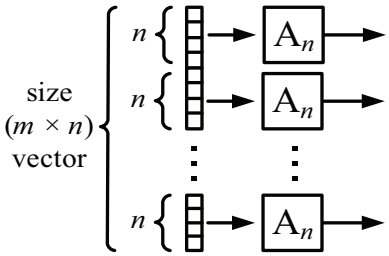
The language described in the previous section can represent a wide range of algorithms, but it does not represent *sequential reuse* of datapath components, where one computational block is used multiple times while computing a single problem. Sequential reuse is necessary since the combinational datapaths realized in Chapter 2 are often too large for practical implementation for all but the smallest transform sizes. This chapter describes extensions to this formula language to represent two types of sequential reuse that are relevant for hardware designs. The result is a *hardware language* that allows explicit datapath description at the formula level. Later, this thesis shows that this extended language drives the proposed compilation system, and that it allows description and generation of a wide tradeoff space.

The work described in this section was presented in part in [17].

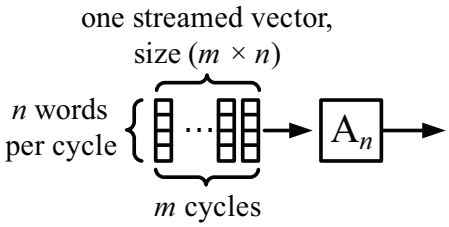
3.1 Streaming Reuse

Streaming reuse restructures a datapath with parallel computation blocks into a smaller datapath where data *stream* through the system over multiple cycles.

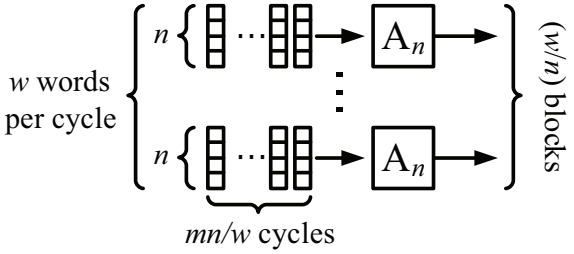
As shown in Section 2.2, the tensor product $I_m \otimes A_n$ results in m data-parallel instantiations of the block A_n (Figure 3.1(a)). However, other structures can also perform the same computation. For example, the tensor product can be interpreted as *reuse in time* (rather than parallelism in



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



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



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

Figure 3.1: Examples of streaming reuse.

space). Then, one can build a single instance of block A_n and reuse it over m consecutive cycles (Figure 3.1(b)). Rather than all mn input points entering the system concurrently, they now stream in and out at a rate of n words per cycle. This is defined as *streaming reuse* and represented as $I_m \otimes^{\text{sr}} A_n$. The *streaming width* indicates the number of inputs (or outputs) that enter (or exit) a section of datapath during each cycle. Here, the streaming width is n.

The two interpretations of $\otimes$ can be nested in order to build a partially parallel datapath that is reused over multiple cycles (Figure 3.1(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 proportionally.

3.1.1 Streaming reuse of other formula constructs

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

$$I_m \otimes_\ell A_\ell^n,$$

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_ℓ, $0 \leq \ell < m$. In the worst case, this can lead to a roughly $m\times$ overhead in the hardware cost, if the m instances cannot share logic between them. However, often this hardware block can be simplified. For example, the DCT-2 algorithm (2.23) presented in Chapter 2.3.4 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, with a comparator controlling the multiplexing logic.

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.

The simplest way to write this as a streaming reuse formula is to reformulate the expression into one that uses $I\otimes_\ell$. So, a diagonal D_n with streaming width w is

$$\mathrm{StreamDiag}(D_n, w) = \left(I_{n/w} \otimes^{\mathrm{sr}}_\ell ((I_w \otimes_k (D_n \circ f(k))) \circ g(\ell))\right), \tag{3.1}$$

where

$$f(k) = \imath_{n/w} \otimes (k)_w,$$
$$g(\ell) = L^n_{n/w} \circ \left((\ell)_{n/w} \otimes \imath_w\right),$$

and $\imath$, $(\ell)_n$, and $\circ$ are indexing functions as defined in [18]:

$$\imath_n \ : \ \mathbb{I}_n \to \mathbb{I}_n;\ i \mapsto i,$$

$$(j)_n \ : \ \mathbb{I}_1 \to \mathbb{I}_n;\ i \mapsto j,$$

$$g \circ f \ : \ \mathbb{I}_m \to \mathbb{I}_N;\ i \mapsto g(f(i)),$$

$$\mathbb{I}_n = \{0, \ldots, n-1\}.$$

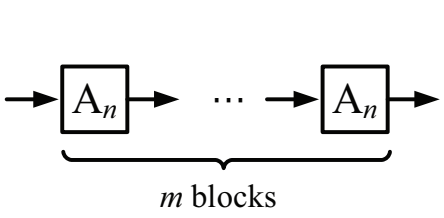
First, the diagonal matrix is split into w partial lookup tables. Each holds the values that will be multiplied by inputs from one of the w input ports. The rightmost term in the formula serves to separate the n point diagonal into n/w segments of size w (each corresponding to one cycle of streaming reuse). The values of these segments can then be computed and the results stored in lookup tables.

This representation also allows easy simplification in the case of certain repeated patterns within the diagonal matrix. For example, every other element of twiddle diagonal T_2^n is equal to 1. So, when this diagonal is streamed (with 2 power streaming width), 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 automatic removal of the unneeded multiplier.

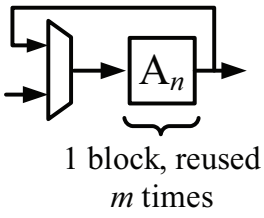
Permutation matrices. Implementing streaming reuse on permutation structures is a difficult problem, but one that is crucially important. A permutation streaming with width w must take in an n point input vector at a rate of w words per cycle. It must buffer the data stream, perform a reordering over the entire n data points, and then stream the data vector out. Chapter 5 presents two methods for constructing this type of system. This thesis uses a streaming permutation wrapper

$$\mathrm{StreamPerm}(P_n, w)$$

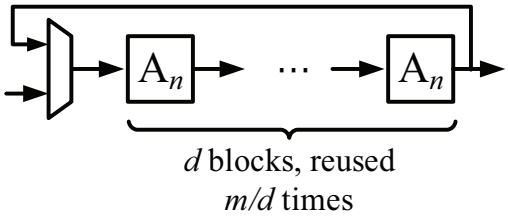
as a way of representing this structure.



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



(b) Full iterative reuse: $\prod_m^{\text{ir}} A_n$.



(c) Partial iterative reuse: $\prod_{m/d}^{\text{ir}} (\prod_d A_n)$.

Figure 3.2: Examples of iterative reuse.

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 blocks cascaded (Figure 3.2(a)).

The same computation can be performed by reusing the A_n block m times (Figure 3.2(b)). Now, the datapath must have a feedback mechanism to allow the data to cycle through the proper number of times. This is called *iterative reuse* and is represented by adding the letters "ir" to the product term: $\prod_m^{\text{ir}} A_n$. By nesting both kinds of product terms, the formula specifies a number of cascaded blocks to be reused a number of times (Figure 3.2(c)). In general, $\prod_m A_n$ can be restructured into $\prod_{m/d}^{\text{ir}}(\prod_d A_n)$, 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 an iterative reuse datapath is built, it is important that the reused portion of the datapath buffer 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.

3.2.1 Iterative Reuse of Other Formula Constructs

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

$$\prod_{\ell=0}^{m-1} A_\ell^n,$$

where the $n \times n$ matrix A is parameterized by iteration 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_ℓ^n (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, the basic block in algorithm (2.21)). This situation is analogous to streaming reuse of the indexed tensor product in Section 3.1.1.

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

$$\prod_{\ell=0}^{m-1} D_\ell^n.$$

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 $\mathrm{StreamDiag}(D_\ell^n, w)$.

An example of this type of formula construct can be found in the Pease FFT (2.13).

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 iteration variable discussed above. When both streaming and iterative reuse are used, the formula becomes

$$\prod_{\ell=0}^{m-1}{}^{\text{ir}}\, \mathrm{StreamPerm}(P_\ell^n, w).$$

Chapter 5 describes how this permutation is implemented.

3.3 Formula-Based Hardware Model

Table 3.1 describes rules for deriving the latency, throughput, and an approximate area cost of four basic formula constructs. Given a matrix formula F, the table provides formulas for latency $L(F)$

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

Table 3.1: Given a matrix formula F, formulas for latency $L(F)$ (cycles), throughput $T(F)$ (transforms per cycle) and approximate area cost $C(F)$ (relative to the area cost of sub-modules).

(cycles), throughput $T(F)$ (transforms per cycle) and approximate area cost $C(F)$ relative to the latency, throughput, and area of F's submodules. The entries of this table can be recursively used to reason about complex formulas. Section 3.4 will give an example for a common formula type that can combine streaming reuse and iterative reuse.

3.4 Combining Streaming and Iterative Reuse

Often, transform algorithms contain the form $\prod_k (I_m \otimes A_n)$. This structure can utilize both iterative reuse (due to the $\prod$) and streaming reuse (due to $I_m \otimes A_n$), allowing a wide range of hybrid implementations that exhibit flexibility across two dimensions. One can restructure this formula to have streaming and iterative reuse of parameterized amounts:

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

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

This parameterized datapath is illustrated in Figure 3.3. 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, one 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 3.1 for the specific parameters of this

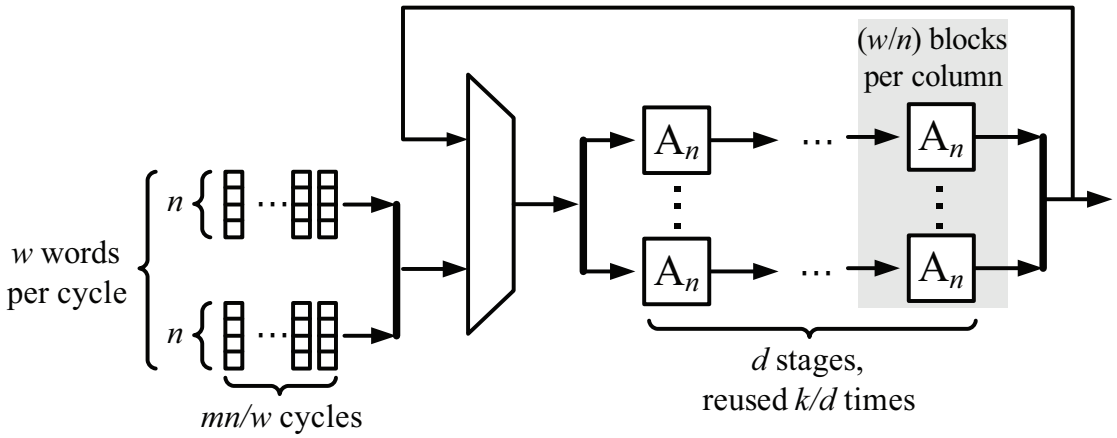


Figure 3.3: Combining iterative and streaming reuse: $\prod_{k/d}^{\text{ir}} \left(\prod_d (I_{nm/w} \otimes^{\text{sr}} (I_{w/n} \otimes A_n)) \right)$.

combined reuse example (Figure 3.3). These calculations assume that B_{mn} (the collective block of A_n blocks) is fully pipelined, i.e., 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 2 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 passes 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 distinction that must be made between 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 $8\times$ more cycles). For this reason, it is preferable to increase d instead of w. However, d must divide k evenly (k is typically the $\log_2$ of the transform size). In many cases, this becomes an "all or nothing" situation, where 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.

Datapath efficiency and vector interleaving. Assume we have an iterative reuse datapath that reuses block B_n. Here, B_n can represent any datapath, including those with further iterative reuse internally. B_n has an inherent latency $L(B_n)$ and throughput $T(B_n)$ (determined by the inverse of the minimum initiation interval of input vectors).

With a single vector recirculating through B_n, the effective throughput of B_n may be further limited to $1/L(B_n)$ if $L(B_n)$ is greater than the minimum initiation interval. In this case the head of the vector is still inside B_n when B_n's input is ready to accept a new iteration.

Let R be a utilization ratio of the effective throughput to the inherent throughput of B_n; this quantifies the portion of B_n's potential throughput that is utilized in the system. For a single vector, $R = (1/L(B_n))/T(B_n)$.

When the utilization by a single vector is sufficiently low, the system can interleave multiple vectors to make use of the full throughput capacity of B_n. Formally, if $R \leq 1/V$ (where V is an integer), the system may interleave V computations through the datapath, increasing the effective throughput and thus increasing the utilization ratio to $R' = (V/L(B_n))/T(B_n)$.

In some cases, a designer may want to increase $L(B_n)$ artificially for better efficiency. For example, if $R = 0.55$, the designer could insert delay buffers in the datapath (increase $L(B_n)$) until R is reduced to 0.5 and then interleave two vectors. This increased utilization yields higher throughput at the expense of added latency, so the designer's particular application requirements will determine the suitability of this approach.

3.5 Summary

This chapter presented a formula-driven hardware paradigm that connects a set of sequential hardware structures with a set of formula constructs. These sequential reuse techniques allow multiple computations from an algorithm to be mapped to a single processing element in the datapath.

First, *streaming reuse* controls the width of the data vector and thus the parallelism of the system. Using these techniques, streaming implementations require that the computation be decomposed into parallel sub-blocks. The formula $I_m \otimes A_n$ mathematically captures this requirement for streaming width w, where w is an integer multiple of n and $w < mn$. This idea is extended when A is allowed to vary with an index variable: $I_m \otimes_\ell A^n_\ell$. Now, since different A matrices are allowed, any $mn \times mn$ matrix that can be decomposed into parallel $n \times n$ subblocks can be streamed with width w. The more similarity that exists among the different the A^n_ℓ blocks, the lower the implementation cost, because in that case the final hardware block can share computational elements between the different iterations.

Similarly, *iterative reuse* (IR) controls the depth of the datapath, which is the number of cascaded stages from input to output. Designs with higher depth have more stages, while lower depths correspond to fewer stages and thus more iterations of the data vector over the structure. This repeated stage structure is written as $\prod_\ell A^n_\ell$. Note that the $n \times n$ matrix A can depend on index ℓ. Thus, any product of matrices can be iteratively reused, but if the A^n_ℓ matrices are not related, there is no benefit from using IR. For example, if A^n_0 and A^n_1 are completely different and cannot share logic, then there would be no benefit to using IR on $\prod_{\ell=0}^{1} A^n_\ell$, because the IR version would have slightly greater area (due to the feedback path), the same or worse latency, and roughly half of the throughput. In this way, the structure of a formula is deeply connected with whether or not iterative reuse provides a benefit.

Thus, the type and amount of sequential reuse that is desired affects the amount of regularity that is beneficial within an algorithm. For example, the Pease FFT algorithm (2.13) and the Iterative FFT algorithm (2.15) are similar except the Pease algorithm has the same permutation in each of its stages, while the Iterative algorithm does not. So, when the user requests a design with IR, the Pease algorithm is more efficient than the Iterative algorithm. In this way, Spiral does not search to select algorithms, but chooses based upon how well an algorithm's structure fits within the paradigm the

user has requested.

Chapter 4

Automatic Compilation from Formula to Datapath

Previous chapters presented a mathematical language for describing linear transform algorithms and how that formula space can be refined to allow explicit specification of sequential reuse. This chapter describes how these concepts are used to drive an automatic compilation framework that maps a transform to an algorithmic formula, transforms the formula to one that includes the desired streaming and iterative reuse characteristics, and maps that hardware formula into a register-transfer level (RTL) Verilog description.

Figure 4.1 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. Then, formula rewriting is used to apply iterative reuse and streaming reuse to the formula; the resulting "hardware formula" now has explicit sequential reuse. Lastly, the hardware formula is then translated into a register-transfer level (RTL) Verilog description. Below, each of these steps is explained.

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 utilizes two directives. First, the

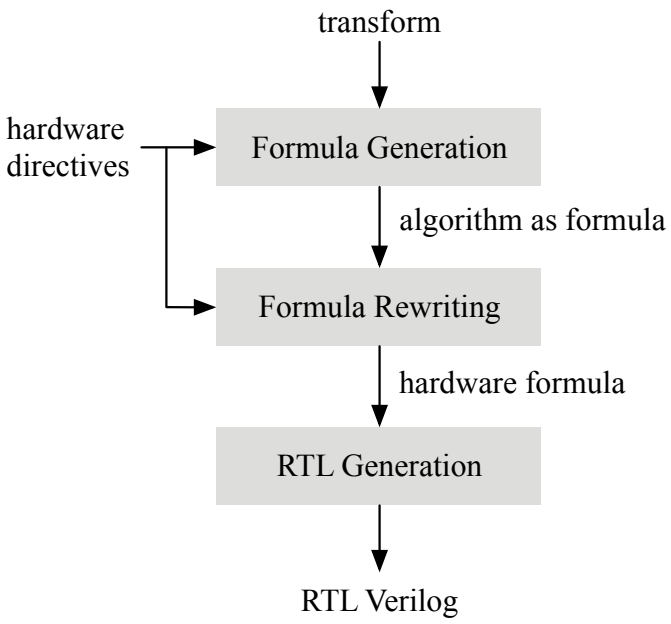


Figure 4.1: Block diagram of hardware compilation system.

streaming tag indicates streaming reuse:

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

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):

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

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.

The tag

$$\underbrace{A_n}_{\text{stream}(w),\ \text{depth}(d_0,\dots)}$$

represents both directives at once.

4.2 Formula Generation: From Transform to Algorithm

The formula generation stage takes in a transform of a fixed size, applies one or more transform algorithms, and outputs a formula that specifies the complete algorithm in the formula language defined in Section 2.2.

This stage contains the system's knowledge of transform algorithms, as detailed in Section 2.3. For each algorithm, it consists of a set of conditions under which it is applicable and the parameterized formula to output. Choices made at this stage are guided by the aforementioned hardware directives, which can dictate which algorithms are appropriate depending on the parameters chosen. Other algorithmic guidance such as desired radix can also be specified by the user at this time.

Chapter 6 discusses transform algorithms in the context of streaming and iterative reuse, and explains which algorithms are used in which reuse scenarios. This portion of the compilation framework is implemented inside of Spiral [4].

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

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

the Pease FFT algorithm (2.13) will be utilized, producing

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

Or, the user could specify depth 7, producing

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

which utilizes the Iterative FFT algorithm (2.15) with radix 2. Lastly, the user could specify radix 4 and depth 3 and the system would first use the Mixed-Radix algorithm (2.16) to decompose the DFT_{128} into DFT_{64} and DFT_2. Then it would use the Iterative algorithm (2.15) to implement

DFT_{64}, producing

$$\underbrace{\mathrm{DFT}_{128}}_{\text{stream(4), depth(3)}} \rightarrow \underbrace{L^{128}_{64}(I_2 \otimes \mathrm{DFT}_{64})L^{128}_{2}T^{128}_{2}(I_{64} \otimes \mathrm{DFT}_2)L^{128}_{64}}_{\text{stream(4), depth(3)}},$$

where DFT_{64} is decomposed:

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

and DFT_4 is further decomposed using (2.12).

4.3 Formula Rewriting: Algorithm to Hardware Formula

Next, the algorithmic formula enters the formula rewriting stage, which takes the formula plus hardware directives and produces a *hardware formula*, which is a restructured and annotated version of the algorithm that includes specification of sequential reuse as discussed in Chapter 3. Thus, the output of this stage corresponds directly to sequential hardware implementation.

The compilation framework accomplishes this restructuring using a *rewriting system* that takes in a tagged formula, and using a set of *rewriting rules*, attempts to restructure the formula or propagate the tag downward. The end result of this stage is a formula with 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

First, Table 4.1 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:

- **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.

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

Table 4.1: Rewriting rules for streaming reuse.

- **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 size of the stream, the rule attempts to unroll the inner tensor product to match the stream size. 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 (3.1) as described in Chapter 3.2.1.
- **stream-perm:** A permutation to be streamed is placed in a streaming permutation wrapper, and will be implemented in hardware as discussed in Chapter 5.

4.3.2 Rewriting for Iterative Reuse

Next, Table 4.2 shows the 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 appropriate 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.

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	$\underbrace{\prod_{\ell=0}^{m-1} A_\ell^n}_{\text{depth}(d_0,d_1,\dots)} \to \prod_{\ell=0}^{(m/d_0)-1}{}_{\text{ir}} \left(\prod_{k=0}^{d_0} \underbrace{A^n_{\ell\cdot d_0+k}}_{\text{depth}(d_1,\dots)} \right)$	m/d_0 integer
product-noIR	$\left(\prod A_n\right) \to A_n \cdot A_n \cdots A_n$	
product-noIR-dep	$\left(\prod_{\ell=0}^{m-1} A_\ell^n\right) \to A_0^n \cdot A_1^n \cdots A_{m-1}^n$	

Table 4.2: Rewriting rules for iterative reuse.

- **product-noIR and product-noIR-dep:** When a product term is encountered without a depth tag, the product is unrolled. Similarly, if the product term has no depth tag but the inner formula depends on ℓ, the ℓ variable is replaced with constants $0, 1, \ldots$ as the product term is unrolled.

Other specialized rules are not needed for the case where streaming permutations or streaming diagonals are iteratively reused and dependent on the iteration index. In the case of permutations, the formula can be directly implemented as described in Chapter 3.2.1 and Chapter 5.

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

After rewriting, the resulting hardware formula is expressed in *Hardware SPL*, an extended version of the language presented in Section 2.2. In addition to the terms allowed in the original language, this hardware language allows tensor products to have explicit streaming reuse, and allows product terms to have iterative reuse. In addition to diagonal matrices D_n and permutation matrices P_n, the language also allows their streaming variants $\mathrm{StreamDiag}(D_n, w)$ and $\mathrm{StreamPerm}(P_n, w)$.

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

4.4.1 Compilation Overview

There are several classes of formula constructs to consider in the RTL generation stage. This section briefly discusses each and indicates which language elements of the hardware SPL language are in each class.

Combinational Formula. Any portion of the formula without iterative reuse ($\prod^{\text{ir}}$), streaming reuse ($I \otimes^{\text{sr}} A$) or explicit sequential meaning (streaming permutations) can automatically be mapped into a combinational datapath, as discussed in Section 2.2. So, this class of formulas contains: computational basic blocks (A_n), unstreamed diagonals and permutations (D_n and P_n), matrix

products without iterative reuse ($A_n \cdot B_n \dots$), and tensor products without streaming reuse ($I_m \otimes A_n$). Additionally, note that the inner term of a streaming diagonal $\mathrm{StreamDiag}(D_n, w)$ is simply a computational basic block (see Chapter 3.1.1), so its inner term fits within this set as well (while its outer term will fall under streaming reuse, below).

When the compiler encounters this type of formula, it constructs a hardware datapath and automatically pipelines the path by inserting staging registers. Additional buffers are added if needed to guarantee that corresponding data words reach the output ports together in the same cycle.

A variation on this class of formula is a basic block with dependence on an index variable, which is set by a $(I \otimes_\ell \dots)$ or $\prod_\ell$ formula. In this case, the compiler determines the possible values for ℓ, and constructs hardware to calculate the result for each. The compiler attempts to simplify the datapaths as much as possible by reducing the dependency on ℓ to the smallest segments possible.

Streaming Reuse. A streaming reuse tensor product $I_m \otimes^{\mathrm{sr}} A_n$ is implemented in hardware as one A_n block, with the input streamed at a rate of n words per cycle over m cycles (as seen in Figure 3.1(b)). When implementing the indexed tensor product $I_m \otimes^{\mathrm{sr}}_\ell A^n_{f(\ell)}$, a hardware counter is constructed, and index variable ℓ is passed to the subblock.

Streaming Diagonal. A diagonal matrix scales each element in the input vector by the corresponding value from the diagonal of the matrix. As shown in Section 3.1.1, a streamed diagonal is represented as an indexed tensor product. The hardware implementation uses w multipliers, where w is the streaming width. Then, the n values from the diagonal are stored in w lookup tables, which feed the multipliers with the appropriate data at each cycle. In some structured diagonals (for example, one where every other diagonal entry is 1), the system can automatically reduce the cost by simplifying away some of the w multipliers.

Streaming Permutation. A permutation with a streaming width w must perform an n point permutation while data streams in and out of the system at a rate of w words per cycle. In Chapter 5, this problem is discussed further and two techniques [19, 20] to construct such systems are explained.

Iterative Reuse. As shown in Fig. 3.2(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 Chapter 3.2). So, the compiler must determine the latency

(in cycles) through A_m and (if necessary) add additional buffering.

In the simplest case, A_m does not depend on the iteration index ℓ, and the only additional hardware required are the feedback path, multiplexer, and multiplexing control logic. However, the compilation framework allows some matrix formulas to be iteratively reused with a dependence on the iteration index. For example, iterative reuse can be used on diagonal matrices, with the diagonal's values changing with the iteration. This is expressed as:

$$\prod_{\ell=0}^{k-1}{}^{\text{ir}} D_\ell^n.$$

In each iteration, the diagonal matrix will scale the n elements of the input vector by n constants, which vary with ℓ. So, when this matrix is iteratively reused, the datapath will still contain the logic needed to scale the data vector, but it must now also contain lookup tables that hold all $\ell \cdot n$ constant values.

Other elements that may be depend on the iteration in an iterative reuse structure in this manner include streaming permutations (described in Chapter 5) and basic blocks.

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 [21], an FPGA area model was created and calibrated for a subset of the design space considered here. Such an approach may be able to be extended to produce an accurate FPGA area model for the full design space. Lastly, the generator is also able to include basic Verilog testbenches for any design it generates. With one command, it is possible to go from transform to formula to RTL, and run a Verilog simulation.

4.4.2 Compilation Stages

The RTL generation stage is implemented as a standalone tool that takes input from Spiral and produces register-transfer level Verilog. The following are the stages undertaken as it translates from a *Hardware SPL* formula to a hardware implementation.

1. **Basic block compilation:** Spiral takes all computational basic blocks (dense matrices) and converts each into a directed acyclic graph that represents the computation of the basic block.

2. **Backend input:** The backend RTL generator reads the formula and basic block code produced by Spiral, and creates an intermediate data structure that contains objects for each class of formula element (e.g. $\prod^{\text{ir}}$, $\mathrm{StreamPerm}$, $I \otimes^{\text{sr}}$, etc.).

3. **Pipelining:** The compiler next adds pipelining stages to all basic blocks and checks that the block's output elements are synchronized correctly. If they are not, the system adds buffers where necessary.

4. **Module sizes:** This stage determines the input/output sizes (number of data points) for each module. Then, it checks these values against the sizes inferred from the original formula for consistency.

5. **Iteration variable:** This compilation stage resolves all iteration variables (e.g., indices from $\prod_\ell A$ or $I \otimes_\ell A$), analyzes them, and propagates them to their appropriate submodules.

6. **Variable ranges:** This stage attempts to determine the minimum and maximum values of each variable in the design's basic blocks. Often, variables that are computed from iteration counters or computed values from lookup tables can be bound. In particular, this information will become useful in simplifying conditional statements and pruning unnecessary computation.

7. **Data type:** Next, the compiler determines the data type of all variables, including the number of bits of precision for fixed point types. This information can be inferred based upon the input data type and the computations that are performed. Furthermore, the compiler implements scaling at this stage, if the user has requested a scaled fixed point implementation.

8. **Lookup tables:** This stage analyzes the constant lookup tables that are used when implementing streaming diagonal matrices. The compiler examines the tables, removing trivial, unused, or redundant ones, and updating references to them appropriately.

9. **Cleanup:** Next, the compiler performs a simplification and cleanup step. This stage merges identical arithmetic operations, removes trivial ones, and performs other simplifications. For

example, code implementing the following equations:

$$t_0 = a_0 + b_0$$
$$t_1 = a_0 + b_0$$
$$t_2 = a_1 \times 1$$
$$t_3 = a_2 \times 0$$
$$t_4 = t_3 + t_1$$

would be simplified to:

$$t_0 = a_0 + b_0$$
$$t_2 = a_1$$
$$t_4 = t_0.$$

10. **Latency and buffering:** This stage uses formulas like those seen in Section 3.3 to determine the latency of each module in the design. As explained in Section 3.2, modules that are iteratively reused have a minimum latency requirement to keep the head of the vector from colliding with its tail. Once latency is determined, the compiler checks to see if this requirement is met; if it is not, the compiler adds buffers to the datapath. (If a streaming permutation is used within the IR block, some or all of the buffering can be inserted into the streaming permutation at no added cost.)

11. **Throughput:** Next, this stage finds the throughput of each module, in order to determine the overall throughput of the system, which becomes part of the input/output specification for the end user. The compiler calculates throughput using formulas like those seen in Section 3.3.

12. **BRAM allocation (optional):** If the user is targeting an FPGA, he or she may request that block RAMs be used and present the compiler with a budget. In this situation, the compiler will examine all of the memory structures needed within the design and choose the best allocation of the allowed number of BRAMs.

13. **Resource estimation (optional):** If requested by the user, the compiler can perform coarse-grained resource estimation, determining the number, data type, and precision of adders, multipliers, RAMs and ROMs.

14. **Testbench generation (optional):** Based on the design's data type, latency, and throughput, the compiler can generate a Verilog testbench suitable for simulation. The testbench outputs data in a format that can be read by Spiral in order to verify the correctness and quantify the error.

15. **Output:** Lastly, the compiler outputs a synthesizable register transfer level Verilog implementation.

Chapter 5

Permuting Streaming Data

Permutations are crucially important portions of linear transform algorithms, but they are difficult to perform in hardware structures that operate on *streaming* data, such as those described in Section 3.1. Prior work addresses this problem in a limited way for a subset of permutations and a subset of problem sizes. This chapter explores the problem of streaming permutations, explains the prior work, and introduces a new general technique that can implement any fixed permutation of any size.

5.1 Problem

A permutation is a fixed reordering of a given number of data elements. P_n represents a permutation of an n point data vector. Figure 5.1(a) shows an example of a 12 point permutation P_{12}. Data elements $(0, \ldots, 11)$ enter concurrently from the left, are reordered, and exit in permuted order. A hardware implementation of such a permutation is trivial if all n data points are available concurrently: it is simply built as a reordering of wires.

However, when a permutation must be performed on *streaming* data, the implementation is more difficult. Figures 5.1(b) and (c) illustrate a streaming version of the permutation seen in Figure 5.1(a). Here, $n = 12$ data words, and the streaming width is $w = 3$ data words per cycle. In Figure 5.1(b) the data labeled $(0, 1, 2)$ enter the system on the first cycle, $(3, 4, 5)$ enter on the second, and so on, for a total of $n/w = 4$ cycles. Figure 5.1(c) shows the structure's output, where the 12 words flow out of the system in their new permuted order $(5, 2, 3, 0, \ldots)$, again with $w = 3$

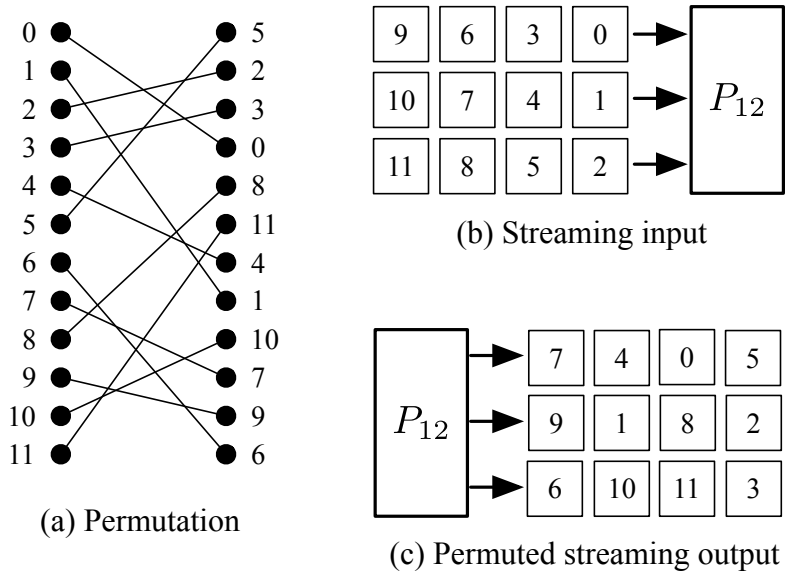


Figure 5.1: Examples: permutation and streaming permutation.

words per cycle.

Performing a permutation on streaming data is difficult because data must be reordered across time boundaries by storing and retrieving from a memory. For example, the element labeled 1 in Figure 5.1(b) streams into the system during the first cycle of the input stream, but must be buffered until the third cycle of the output stream (in Figure 5.1(c)).

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

5.2 Existing Approaches

Stride Permutations. Most of the existing work in the literature on streaming permutations focuses on the stride permutation L. For example, [22] generates hardware implementations of $L^{2^n}_{2^s}$ with streaming width 2^k using a number of small FIFO memories with interconnection networks. Although this technique uses the minimal storages for a given permutation, its usage of multiple small independent memories leads to high overhead and complex control. In another example, [23] is able to perform $L^{2^n}_{2^s}$ with streaming width 2^k where $k \geq n - s$ using memories and interconnection networks. Other similar techniques include [24].

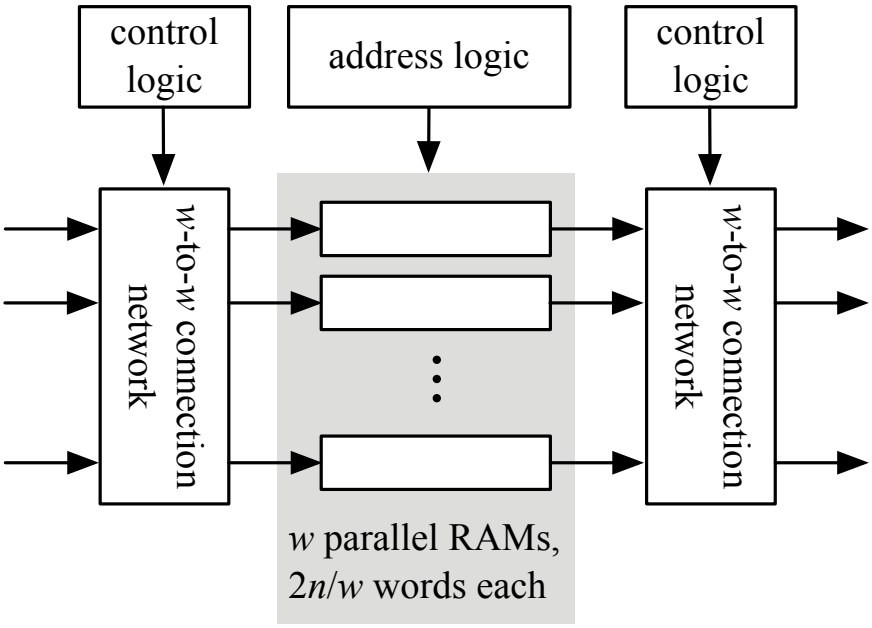


Figure 5.2: Block diagram of streaming permutation structure (n points, w words per cycle) built using the bit matrix method.

Data Format Converters. Parhi [25] introduces a technique to design a register-based data format converter that can be used to implement a streaming permutation. This method uses a register allocation algorithm to design a datapath consisting of individually-controlled registers connected with wires and switches. This technique is general; it can be used on any streaming permutation. However, its need to provide control logic to all of the individual registers and connection networks leads to considerable overhead.

Bit Matrix Method. In [20], Püschel, Milder and Hoe present a method for generating streaming permutation circuits for a subset of all permutations and streaming widths. Figure 5.2 shows a high-level view of the designs produced using this technique, which are comprised of two connection networks and an array of RAMs. All storage is consolidated within w parallel RAMs, each which must hold $2n/w$ data words. This method does not require pre-computed control values—instead, all control is computed online with inexpensive bit operations. Additionally, the connection networks used by this technique are optimized for the given permutation.

The method in [20] works by representing permutations as *bit matrices*—linear mappings on the bit-level representation of data input and output locations. The permutation's bit matrix is then decomposed in a way that fully specifies the datapath and control logic for a given permutation.

Further, the method optimizes the required connection networks and control logic; provable lower bounds are reached for an important subset of problems.

However, this technique does not apply to all permutations: only a subset can be represented as bit matrices, and the technique requires the problem size n and streaming width w to be powers of two.

5.3 General Streaming Permutations

This section presents a parameterized datapath and accompanying mapping technique that is able to be configured to perform any fixed streaming permutation. Similar to [20], this technique uses parallel memories and switching networks; it is scalable with the problem size and the streaming width. Unlike [20], this technique applies to all streaming permutations and does not force the problem size or streaming width to be a power of two. This work was first presented in [19].

5.3.1 Parameterized Datapath

A key aspect of this work is the identification of a scalable datapath structure that can be configured to perform any fixed streaming permutation with any streaming width. This section presents this parameterized datapath and discusses its operation.

Figure 5.3 shows the datapath considered. It consists of two memory arrays M_0 and M_1, a connection network N, and lookup tables (ROMs) that contain address and control data (R, T, and W).

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

N represents a connection network that is capable of taking w data points as input and outputting them in any given order, with ROM T storing its pre-computed control data. When w is a power of two, this network can be built as explained in [26, 27]. The resulting network utilizes $w \log_2(w) - w + 1$ 2-by-2 switches, which is asymptotically optimal. If w is not a power of two, N could

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

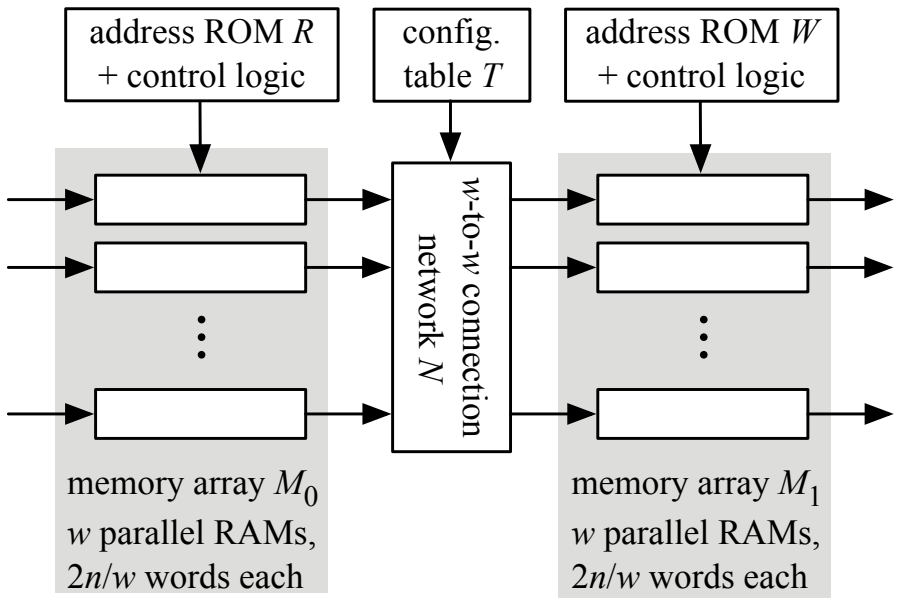


Figure 5.3: Block diagram of streaming permutation structure (n points, w words per cycle) built using the general permutation method.

be constructed in several ways; the simplest is to build the network for the next largest power of two. However, any other appropriate interconnection network or crossbar may be substituted. This decision does not affect the other portions of the datapath (M_0, M_1, R, and W).

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

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

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

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

4. Write to M_1. On each cycle of this phase, one word is written into each of the w memory banks of M_1. The write addresses are pre-computed and stored in lookup table W, each line of

which holds w write addresses. So, W contains n/w lines, each of width $w \cdot \lceil \log_2(n/w) \rceil$ bits.

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

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

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

An important aspect to understand about this datapath is that all of the reordering is done in stages 2–4. Stage 1 writes data to M_0 in natural order, and Stage 5 reads data from M_1 assuming it is *already* in permuted order. The problem then becomes: given a permutation P_n, how does one guarantee that, on each cycle, one can read w words from M_0 and write them to the correct locations of M_1 without conflicts (i.e., needing to read/write multiple words from/to the same RAM at the same time)? A solution to this problem implies a datapath configuration that will be capable of performing P_n with full throughput (w words per cycle) without stalling. Section 5.3.2 formalizes this problem and derives a solution.

Extension to support multiple permutations. This method can also be used if multiple permutations are to be performed by one iterative reuse structure. For example, the formula

$$\prod_{\ell=0}^{k-1}{}^{\text{ir}} \underbrace{P_n^{\ell}}_{\text{stream}(w)},$$

performs a different permutation in each of the k iterations. So, this can be implemented by increasing the size of the pre-computed ROMs by a factor of k, and adding an extra input for the iteration variable ℓ to determine which of the k configuration sets to use.

Tensor product of permutations. Often, algorithms permute n points by performing a smaller k point permutation over n/k consecutive regions of the vector ($k < n$, and k and n/k are integers).

As a formula, this is written $I_{n/k} \otimes P_n$. When streamed with width w where $w < k$, this becomes

$$\underbrace{I_{n/k} \otimes P_k}_{\text{stream}(w)} \rightarrow I_{n/k} \otimes^{\text{sr}} \underbrace{P_k}_{\text{stream}(w)} \rightarrow I_{n/k} \otimes^{\text{sr}} \text{StreamPerm}(P_k, w)$$

Although this is still a permutation on n total points, the implementation cost is equal to a k point streaming permutation.

These structures occur frequently in transform algorithms. For example, each stage (parameterized by ℓ) of the Iterative FFT (2.15) has permutation:

$$I_{r^\ell} \otimes P_{r^{t-\ell}}$$

In iteration ℓ, this performs r^ℓ permutations on $r^{t-\ell}$ points. Thus, when this formula is streamed, the stages corresponding to larger values of ℓ are less expensive to implement.

5.3.2 Problem Formulation

This section formulates a mathematical problem that corresponds to mapping a streaming permutation to the datapath presented in Section 5.3.1. Further, it demonstrates that this problem is solvable for all permutations and streaming widths.

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

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

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

   $k \neq \ell \pmod{w}$ *and* $P_n(k) \neq P_n(\ell) \pmod{w}$.

A solution to Problem 1 will allow us to read (in cycle j) w elements (specified by S_j) from M_0

and write them to M_1. By construction, the w words will be read from w distinct memory banks in M_0 and written to w distinct memory banks in M_1. Since there are no conflicts at the read/write ports, the system can sustain a full throughput of w words per cycle as desired. Thus, a solution to this problem implies values for R, W, and T that will allow the datapath in Figure 5.3 to perform P_n.

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

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

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

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

For example, consider the permutation

$$P_{12} = (0, \ldots, 11) \rightarrow (3, 7, 1, 2, 6, 0, 11, 9, 4, 10, 8, 5). \tag{5.1}$$

This permutation corresponds to the example seen in Figure 5.1. If $w = 3$ ports, then

$$\pi_3(P_{12}) = \begin{pmatrix} 1 & 2 & 1 \\ 1 & 1 & 2 \\ 2 & 1 & 1 \end{pmatrix}. \tag{5.2}$$

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

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

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

of which may be repeated.

Proof. This follows from [28], which proves that any semi-magic square can be decomposed into a sum of permutation matrices. The total number must be n/w due to Lemma 3. □

We represent this decomposition as

$$\pi_w(P_n) = \beta_0 Q_0 + \beta_1 Q_1 + \cdots + \beta_{k-1} Q_{k-1}, \tag{5.3}$$

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

Each of the Q_i permutations represents a particular configuration of the switching network N in Figure 5.3. So, one can choose w corresponding words from M_0 (one from each bank), permute them by Q_i, and write them into the correct locations of M_1 (one word to each bank). Thus, the decomposition in (5.3) represents a solution to Problem 1.

Continuing the example from (5.1) and (5.2), we can decompose $\pi_w(P_{12})$ as

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

Lemma 5. $\pi_w(P_n)$ *has a decomposition* (5.3) *that satisfies*

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

Proof. [29] gives an upper bound of $w^2 - 2w + 2$ permutations. We can further tighten this bound to $\min(w^2 - 2w + 2, n/w)$, since Lemma 4 shows that there are at most n/w permutations. □

Lemma 5 shows that even though there are $w!$ many $w \times w$ permutation matrices, a much smaller number is needed in (5.3). For the proposed datapath (Figure 5.3) this could lead to a reduced storage requirement for T since some of the k settings may be used multiple times (if $k < n/w$). Taking advantage of this would require storing the values of β_i and adding additional logic to determine when to increment T's address value. This optimization would reduce the number of elements in T from n/w to k, and hence may reduce the overall hardware cost for some problems

(if the savings from storing fewer words offsets the added cost of storing the β_i and the added logic). This optimization is not explored here.

5.3.3 Algorithm

Based upon the problem specified in Section 5.3.2, we formulate an algorithm that calculates the parameters needed to perform a given permutation P_n with a streaming width w on the datapath in Figure 5.3. This algorithm is executed at design time; it determines the control values to be stored into ROMs R, T, and W.

This algorithm computes a decomposition of $\pi_w(P_n)$ of the form (5.3) and calculates the corresponding values to store in R, T, and W. Recall, R and W denote the collection of addresses for M_0 and M_1, respectively. $L = (Q_0, \ldots, Q_{n/w-1})$ represents the list of permutations that the connection network must perform, and T represents the configuration bits associated with each permutation in L (computed using the methods in [26, 27]).

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

1. $C \leftarrow \pi_w(P_n)$; (as described in Definition 2).
2. **while** (C contains non-zero entries) **do**
   - **find** a permutation Q included in C;
   - **while** $C - Q$ contains no negative entries **do**
     - $C \leftarrow C - Q$;
     - **append** permutation Q to L;
     - **find** $(x_0, \ldots, x_{w-1})$ s.t. $(x_i (\text{mod } w) = i)$ and $(P_n(x_i) (\text{mod } w) = Q(i))$;
     - **append** $(\lfloor x_i/w \rfloor)$, $0 \leq i < w$ to R;
     - **append** $(\lfloor P_n(x_{Q^{-1}(i)})/w \rfloor)$, $0 \leq i < w$ to W;
3. **calculate** T based on the permutations stored in L, using the techniques in [26, 27];
4. **output** R, T, and W;

The most computationally difficult part of Algorithm 6 is finding a permutation Q_w that may be subtracted from C_w. This can be accomplished in three ways: using a brute force algorithm, mapping the problem to a satisfiability problem, or using an algorithm based on systems of distinct

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

Table 5.1: Summary of memories required for streaming permutation structures.

representatives (e.g., [30, Ch. 5]). This implementation uses the satisfiability approach; it is able to solve practical problem sizes quickly. For the largest problem in this evaluation ($n = 4096, w = 64$), Algorithm 6 completes in approximately 6 minutes.

5.4 Evaluation

This section discusses the implementation of the proposed general streaming permutation method and evaluates them by comparing with the results of [20]. A tool based upon the techniques discussed in this section is implemented as an automated generation tool (built as a module within the main compiler described in Chapter 4). This tool takes as input a permutation P_n and a streaming width w, and outputs a register-transfer level Verilog description of the design. The tool generates an instance of the parameterized datapath, and uses Algorithm 6 to determine the values to store in the lookup tables. Additionally, the bit-permutation-based technique from [20] is also implemented as another module within the compilation tool.

Analysis of Generated Designs. Table 5.1 summarizes the number and size of memories needed by the proposed solution. Additionally, the datapath requires $w' \log_2 w' - w' + 1$ two-input switches, where $w' = 2^{\lceil \log_2 w \rceil}$. In general, the cost of this implementation depends only on n (the number of points in the data vector) and w (the streaming width of the system).[2]

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

Experimental Evaluation. This section evaluates the cost and performance of the proposed generated designs when synthesized for a Xilinx Virtex-5 FPGA. We use Xilinx ISE 9.2 to synthe-

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

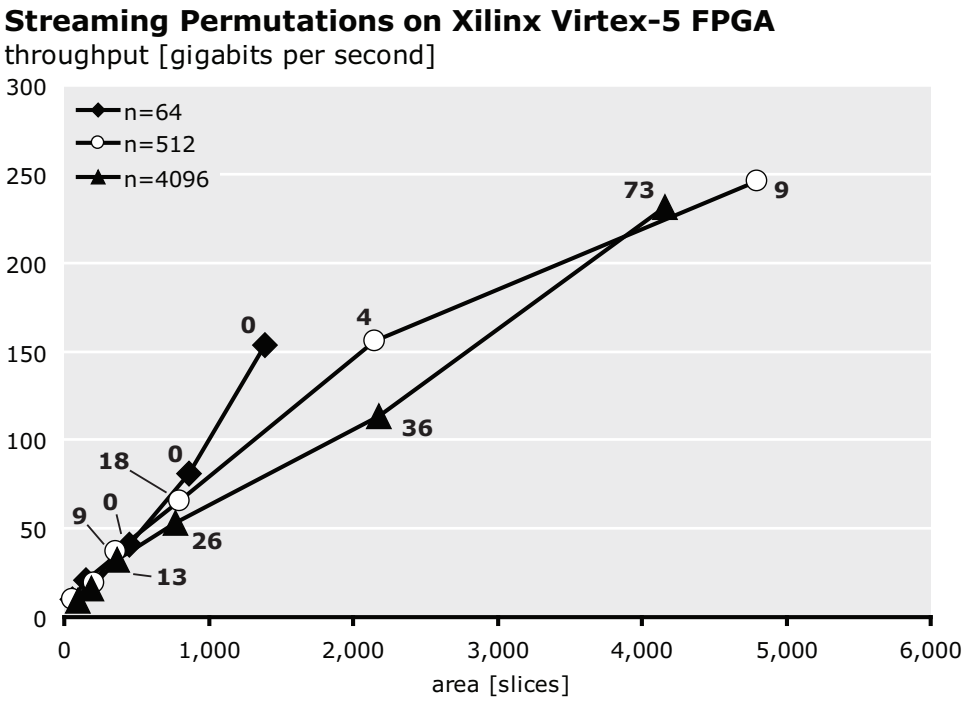


Figure 5.4: Throughput versus slices for $n = 64, 512, 4096$. Labels: number of BRAMs.

size and place/route all designs, and we extract all timing and area measurements after place/route has completed. In this evaluation, the data words are $b = 16$ bits wide.

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

This evaluation quantifies the cost and performance of the following configurations of this datapath: $n = 64$ with $w = 2, 4, 8, 16, 32$, and $n = 512, 4096$ with $w = 2, 4, 8, 16, 32, 64$. Because the cost of the implementation does not depend on the specific permutation being performed, we choose one randomly for each n. Figure 5.4 shows throughput (in gigabits per second) versus FPGA area (in slices, which are the reconfigurable blocks of the FPGA) for all designs. Each line shows a different value of n (the problem size), and the different points within a line correspond to different values of streaming width w: the left-most point corresponds to $w = 2$; w doubles with each successive point. (Recall, the throughput of each design is w words per cycle.) Additionally, we label several of the points with the number of block RAMs (BRAMs) used by that design.

Figure 5.4 shows that as w increases, the resulting designs exhibit higher throughput, but be-

come commensurately more expensive (in area). As n increases, the results do not show a significant change in throughput or area, but a large increase in the number of BRAMs required. This occurs because the size of all memories grows with n (as shown in Table 5.1).

In order to provide a reference point for comparison, these designs are compared to [20], which describes a method for generating streaming permutation circuits for a subset of all permutations and streaming widths.[3] The designs produced by [20] utilize one memory array with interconnection networks at its inputs and outputs; both networks are optimized for the specific permutation considered. Furthermore, all memory and switch configurations are calculated online; no lookup tables are used. For an important class of problems, [20] is able to produce designs with the optimum address logic and switching network (given the assumed architecture).

However, the technique in [20] is only applicable to a small subset of streaming permutations. The goal is not to improve on [20]'s cost/performance tradeoff; it instead serves as a way to measure the added costs incurred by moving to the general structure.

The area required by the designs in [20] depends on the permutation being performed (as well as the permutation size and streaming width). So, we compare against designs for two permutations: the stride-by-two permutation, which is in the class of least expensive problems supported by [20], and the bit reversal permutation, which is in the class of most expensive problems. Again, we evaluate designs using a random permutation, since the cost does not depend on the specific permutation being performed.

We synthesize and place/route these benchmark designs using the previously stated assumptions, and present the results in Figure 5.5 for $n = 512$. Again, the plot shows throughput versus area and includes the number of BRAMs for several designs. For small values of w, all three designs have similar costs. However, as w increases, the amount of slices and BRAM required for the general method increases more quickly than those from [20]. If we repeat this experiment for smaller problem sizes (values of n), the difference between the proposed method and the benchmark is reduced; for larger values of n, it is increased.

The Spiral hardware generation framework can implement both the general streaming permutation method [19] and the bit-matrix method [20]. Since the bit-matrix method results in lower

[3] [20] is only able to perform permutations that arise from invertible mappings on the bit representations of the indices, such as the bit reversal or stride permutations. Of the $n!$ possible permutations on n points, [20] is able to perform $(2^m - 1)(2^m - 2) \cdots (2^m - 2^{m-1})$, where $n = 2^m$ and only when n and w are powers of two.

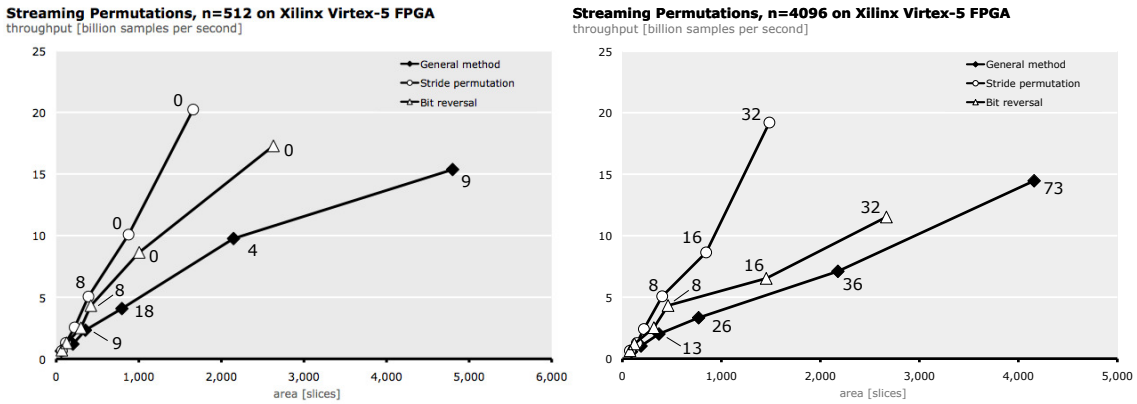


Figure 5.5: Comparison of general streaming permutation method with bit matrix method for $n = 512$ and $n = 4096$. Data labels indicate the number of BRAMs.

cost, it is used when: (1) the transform size and streaming width are powers of two, (2) the permutation can be represented as a bit matrix, and (3) only one permutation is to be computed with the same structure. If these conditions are not met, the general streaming method described in this chapter [19] is used.

Although it is not discussed in this chapter, the compiler also can implement permutations with streaming width $w = 1$ trivially using a single memory (one read port and one write port). Data are written into the memory sequentially, and a ROM is used to encode the read addresses.

Chapter 6

Transform Algorithms

This chapter discusses the transform algorithms presented in Section 2.3 in the context of streaming reuse and iterative reuse, examining the applicability and limitations of each. Chapter 7 will later evaluate implementations of these algorithms on FPGA and ASIC.

First, Section 6.1 discusses hardware structures for algorithms for the discrete Fourier transform, including those that are applicable for two-power and non-two-power problem sizes. Then, Section 6.2 explains algorithms for the two-dimensional DFT and how they can be mapped to hardware with more flexibility than the one-dimensional version. Next, Section 6.3 and Section 6.4 discuss hardware implementation of algorithms for the RDFT and DCT algorithms, respectively.

6.1 Discrete Fourier Transform

Pease FFT. The Pease FFT algorithm with radix r is given by

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

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 $w \geq r$.

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 of Chapter 7 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 Chapter 3.2.1. It can be iteratively reused with depth d where $d \mid t$ (that is, d evenly divides t).

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

$$\underbrace{\mathrm{DFT}_{r^t}}_{\text{stream}(w),\ \text{depth}(d)} \rightarrow \left(\prod_{k=0}^{t/d-1}{}^{\mathrm{ir}} \left(\prod_{\ell=0}^{d-1} \mathrm{StreamPerm}(L_r^{r^t}, w) \cdot (I_{r^t/w} \otimes^{\mathrm{sr}} (I_{w/r} \otimes \mathrm{DFT}_r)) \right.\right.$$
$$\left.\left. \cdot \mathrm{StreamDiag}(C_{kd+\ell}^{r^t}, w) \right) \right) \cdot \mathrm{StreamPerm}(R_r^{r^t}, w).$$

Both permutations used here (stride permutation L and digit-reversal permutation R) are implemented using the bit matrix method [20] when r is a power of two, as explained in Chapter 5.2.

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

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

which is illustrated as hardware with no sequential reuse in Figure 6.1. The 8 point input vector x enters on the left and flows through the datapath, producing output vector y on the right. Below the figure, 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.

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

$$\underbrace{\mathrm{DFT}_{2^3}}_{\text{stream}(2)} \rightarrow$$
$$\left(\prod_{\ell=0}^{2} \mathrm{StreamPerm}(L_2^8, 2) \cdot (I_4 \otimes^{\mathrm{sr}} \mathrm{DFT}_2) \cdot \mathrm{StreamDiag}(C_\ell^8, 2) \right) \cdot \mathrm{StreamPerm}(R_2^8, 2).$$

This is illustrated in Figure 6.2. 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

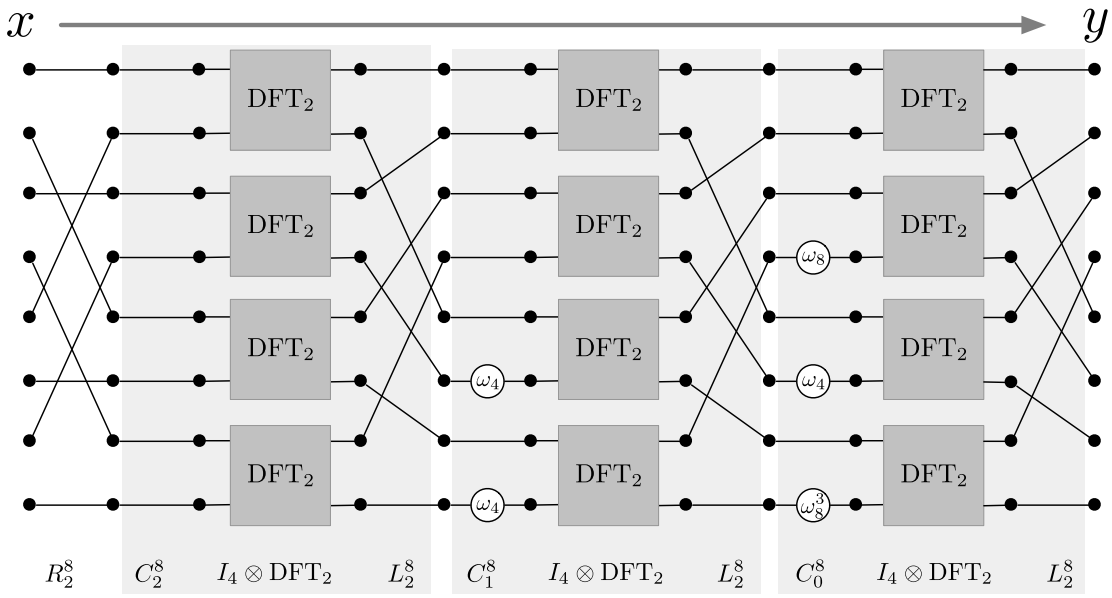


Figure 6.1: Pease FFT: DFT_{2^3}, no sequential reuse.

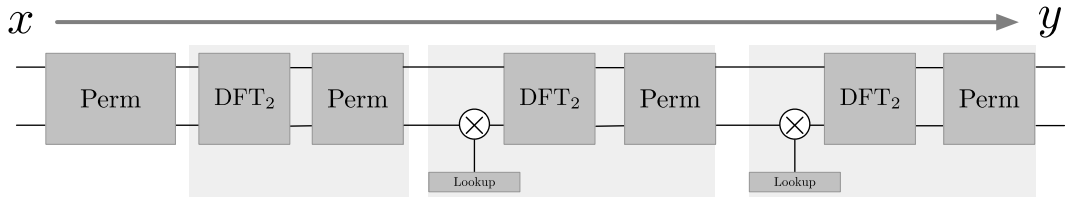


Figure 6.2: Pease FFT: DFT_{2^3}, streaming width $w = 2$.

different constants are needed for different portions of the stream. The input data x flows in at a rate of $w = 2$ words per cycle, then flows through input permutation R_2^8 before flowing through the three stages and exiting as y. A new input vector can begin flowing into the system as soon as the previous one has finished entering.

Next, we can apply iterative reuse with depth $d = 1$.

$$\underbrace{\mathrm{DFT}_{2^3}}_{\text{stream}(2),\,\text{depth}(1)} \rightarrow$$
$$\left(\prod_{\ell=0}^{2}{}^{\text{ir}}\ \mathrm{StreamPerm}(L_2^8, 2) \cdot (I_4 \otimes^{\text{sr}} \mathrm{DFT}_2) \cdot \mathrm{StreamDiag}(C_\ell^8, 2)\right)$$
$$\cdot\ \mathrm{StreamPerm}(R_2^8, 2).$$

Figure 6.3 shows the resulting datapath. Now, the three stages are iteratively reused, collapsed

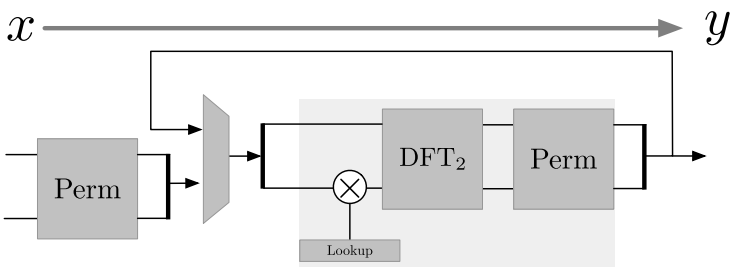


Figure 6.3: Pease FFT: DFT_{2^3}, streaming width $w = 2$, depth $d = 1$.

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.

Iterative FFT Algorithm. The variant of the Iterative Cooley-Tukey algorithm considered in this work is given by

$$\mathrm{DFT}_{r^t} = L^{r^t}_r \left(\prod_{\ell=0}^{t-1} \left(I_{r^{t-1}} \otimes \mathrm{DFT}_r\right) \left(I_{r^\ell} \otimes \left(E^{r^{t-\ell}}_\ell \cdot Q^{r^{t-\ell}}_\ell\right)\right)\right) R^{r^t}_r,$$

for radix r. It contains permutation matrices L, Q, and R (all of which are implemented using the bit matrix method when r is a power of two), and diagonal matrix E. Similar to the Pease algorithm above, this algorithm can easily be streamed with width $w \geq r$.

However, this algorithm differs from the Pease FFT in an important way. Here, the permutation and diagonal matrices grow smaller and more repetitive as ℓ grows larger. When iterative reuse is not used, this makes the Iterative Cooley-Tukey FFT less expensive to implement than the Pease FFT, because the cost of streaming permutation implementation is related to the number of points being permuted (r^t in the Pease algorithm and $r^{t-\ell}$ in this algorithm).

So, the Pease FFT is less expensive in iterative reuse implementations, and the Iterative Cooley-Tukey FFT algorithm is cheaper in implementations without iterative reuse.

Mixed-Radix Algorithm. The Mixed-Radix FFT algorithm of radices r and s is given by

$$\mathrm{DFT}_{r^k s^\ell} = L^{r^k s^\ell}_{r^k} \cdot \left(I_{s^\ell} \otimes \mathrm{DFT}_{r^k}\right) \cdot L^{r^k s^\ell}_{s^\ell} \cdot T^{r^k s^\ell}_{s^\ell} \cdot \left(I_{r^k} \otimes \mathrm{DFT}_{s^\ell}\right) \cdot L^{r^k s^\ell}_{r^k}.$$

This algorithm is typically used to compute a transform at a non-native radix size (for example, DFT_{512} can be decomposed into DFT_2, which can be implemented directly, and DFT_{256} which can be implemented with radix 4 or 16). Additionally, this algorithm is frequently used in non-two-power-sized problems if the problem size is a product of small primes (for example DFT_{432} can be broken down into DFT_{2^4} and DFT_{3^3}).

Since this algorithm is comprised only of permutations (L), a diagonal matrix (T), and parallel DFTs, streaming reuse can easily be applied with the restriction that the streaming width w must evenly divide $r^k s^\ell$ and it must be a multiple of r and s. For this reason, implementations of this algorithm quickly lose flexibility as the radix sizes increase.

The individual DFT_{r^k} and DFT_{s^ℓ} blocks can be computed with either the Pease or Iterative FFT algorithms, depending on whether or not iterative reuse is desired (as described above).

In principle it is possible to use iterative reuse to collapse the two DFT stages into one combined stage. However, the two blocks (DFT_{r^k} and DFT_{2^ℓ}) are so different that the overhead of doing so is high.

Although the algorithm still holds when $r = s$, there is no benefit of using this algorithm over the radix r algorithms considered above (Pease and Iterative FFTs).

Bluestein Algorithm. The Bluestein algorithm performs a DFT of any given problem size n by performing scaling and a forward and inverse DFT of size m where $m > 2n - 1$.

$$\mathrm{DFT}_n = D^{(2)}_{n\times m} \cdot \mathrm{IDFT}_m \cdot D^{(1)}_m \cdot \mathrm{DFT}_m \cdot D^{(0)}_{m\times n},$$

Typically, this algorithm is used to perform non-power-of-two size DFTs, where m is chosen to be a power of two. This allows the bulk of the computation to be performed using the Pease and Iterative Cooley-Tukey algorithms.

Further, the forward and inverse DFT_n are closely related in structure, so the two stages of the algorithm can be iteratively reused in the following way. First, the formula is regrouped:

$$\mathrm{DFT}_n = I_{n\times m} \left(\prod_{\ell=0}^{1} D^{g(\ell)}_m \cdot \mathrm{DFT}^{f(\ell)}_m \right) \cdot D^{(0)}_{m\times n},$$

where

$$f(\ell) = \begin{cases} -1, & \ell = 0 \\ 1, & \ell = 1 \end{cases} \quad \text{and} \quad g(\ell) = \begin{cases} 2, & \ell = 0 \\ 1, & \ell = 1 \end{cases}.$$

So, this product term can now be iteratively reused with depth 1 if desired. Then, the inner DFT block can be iteratively reuse (using the Pease algorithm) or not (using the iterative Cooley-Tukey algorithm).

6.2 Two-Dimensional Discrete Fourier Transform

As shown in (2.18), the row-column algorithm for the two-dimensional DFT is given by

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

One can use the Pease FFT (2.13), the Iterative Cooley-Tukey FFT (2.15) or the Mixed-Radix FFT (2.16) to perform the DFT_n. Using Pease (with radix r):

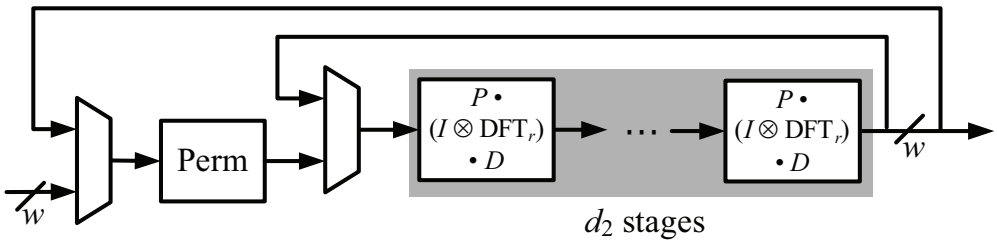
$$\begin{aligned} \text{DFT}_{n\times n} &= \prod_{k=0}^{1} \left(\left(I_n \otimes \left(\left(\prod_{\ell=0}^{t-1} \left(L_r^n \cdot (I_{n/r} \otimes \text{DFT}_r) \cdot C_\ell^n \right) \right) \cdot R_r^n \right) \right) \cdot L_n^{n^2} \right) \\ &= \prod_{k=0}^{1} \left(\left(\prod_{\ell=0}^{t-1} ((I_n \otimes L_r^n) \cdot (I_{n^2/r} \otimes \text{DFT}_r) \cdot (I_n \otimes C_\ell^n)) \right) \cdot (I_n \otimes R_r^n) \cdot L_n^{n^2} \right), \end{aligned}$$

where $t = \log_r(n)$. In order to facilitate discussion of this algorithm, one can write it in a simplified form while still capturing its relevant degrees of freedom:

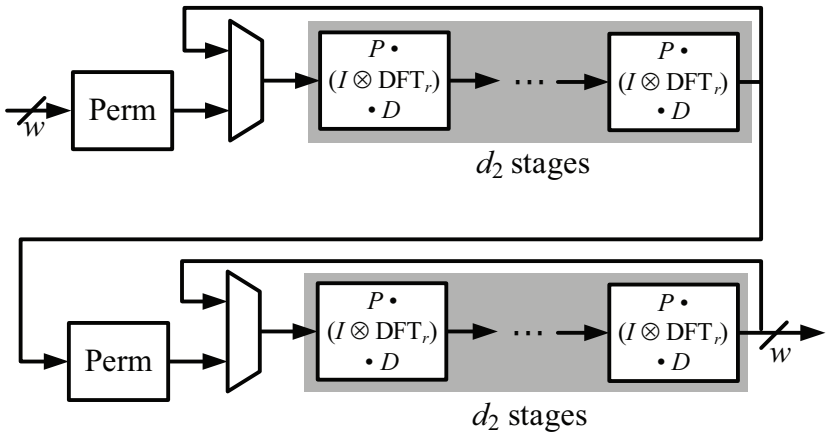
$$\text{DFT}_{n\times n} = \left(\prod_{k=0}^{1} \left(\prod_{\ell=0}^{t-1} P_{n^2} (I_{n^2/r} \otimes \text{DFT}_r) B_{n^2} \right) Q_{n^2} \right),$$

where P and Q are placeholders for the permutation expressions written above, and B is a placeholder for its corresponding diagonal matrix. Using the other algorithms (2.15) and (2.16) similar expressions are obtained.

These algorithms contain two iterative product terms, as seen above. Each may utilize iterative reuse, which is characterized with *depth* parameters d_1 and d_2 (see Chapter 3.2). This is illustrated in Figure 6.4.



(a) $d_1 = 1$. The outer product term is iteratively reused.



(b) $d_1 = 2$. The outer product term is fully unrolled, i.e., not iteratively reused.

Figure 6.4: Illustrations of $\mathrm{DFT}_{n\times n}$ with outer product term parameterized by d_1, inner product term parameterized by d_2, and streaming width w.

Let d_2 be the depth of the inner product; d_2 can be between 1 and t, provided that t/d_2 is an integer. Each internal block (a shaded region in Figure 6.4) consists of d_2 stages of $P{\cdot}(I{\otimes}\mathrm{DFT}_r){\cdot}D$, streamed with w ports. When $d_2 = t$, the Iterative Cooley-Tukey FFT (2.15) is used to decompose DFT_n (or the mixed-radix algorithm if the problem size is not a power of the chosen radix); when $d_2 < t$, the Pease FFT (2.13) is used. Alternatively, the Mixed Radix FFT can be used to decompose DFT_n as well.

Permutation Q can be implemented with the bit matrix method when n is a power of two.

Let d_1 be the depth of the outer product; d_1 can be either 1 or 2. When $d_1 = 1$, the outer product term is iteratively reused (Figure 6.4(a)). When $d_1 = 2$, the outer term is unrolled, giving two cascaded stages as seen in Figure 6.4(b).

6.3 Real Discrete Fourier Transform

RDFT from Complex DFT. An RDFT of size n can be computed from a DFT of size $n/2$ using the following algorithm

$$\mathrm{RDFT}_n = \left((K_2^{\prime n} \otimes I_2) \cdot \left(I_{n/4} \otimes_\ell A_4(\ell)\right) \cdot ((K_2^{\prime n})^{-1} \otimes I_2)\right) \cdot \overline{\mathrm{DFT}_{n/2}},$$

where K' is a permutation (thus $(K' \otimes I)$ is also a permutation), $A_4(\ell)$ is a 4×4 basic block that depends on iteration variable ℓ, and $\overline{\mathrm{DFT}_{n/2}}$ represents an $n \times n$ real matrix that is constructed from the $n/2 \times n/2$ complex matrix $\mathrm{DFT}_{n/2}$ as explained in Chapter 2.3.3. While computing, changing the data vector from a $n/2$ point complex into a n point real vector is simply a reinterpretation: no logic is needed.

Any of the DFT algorithms discussed above may be used to compute $\mathrm{DFT}_{n/2}$. As described previously, the algorithms can be transposed, which moves the bit reversal permutation R to the left side of the formula, where it can be merged with the $(K_2^{\prime n})^{-1} \otimes I_2$ permutation. K' can not be represented as a bit matrix, so the general streaming method is needed.

This algorithm can be streamed with (real) streaming width $w \geq 2r$ where r is the (complex) radix of the implementation of $\mathrm{DFT}_{n/2}$ chosen. Iterative reuse can be applied to $\mathrm{DFT}_{n/2}$ as described above.

This algorithm has higher arithmetic cost than the native RDFT algorithms below, but in some cases, it maps to hardware structures more efficiently. In Chapter 7, both types of algorithms are evaluated over a space of architectures, and the resulting tradeoffs are discussed.

Native RDFT Algorithm: Constant Geometry. Similar to the Pease FFT algorithm, the radix $2k$ RDFT can be computed with a constant geometry algorithm:

$$\mathrm{rDFT}_{2km}(0) = V_{k,m}^{2km} \left(\prod_{i=0}^{\log_k m} \left(I_m \otimes_\ell \mathrm{rDFT}_{2k}(s_{k,q}(0,0,f_{k,m}(i,\ell)))\, L_{2m}^{2km}\right) \right) L_{km}^{2km},$$

where V and L are permutations, and functions s and f are used to compute the parameter to each basic block $\mathrm{rDFT}_{2k}(\cdot)$.

This algorithm is considered to have *constant geometry* because each of its $\log_k m + 1$ stages

performs the same permutation L^{2km}_{2m}. So, this algorithm is ideal for iterative reuse: the only change from stage to stage occurs in the parameter to $\mathrm{rDFT}_{2k}()$. This parameter controls the scaling performed in the basic block; the constants and a portion of the addressing logic are pre-computed and stored in lookup tables; the remaining portion is mapped to logic.

This algorithm can be streamed with streaming width $w \geq 2k$, $w \mid 2km$. Permutation V cannot be represented as a bit matrix and is thus implemented using the general method [19].

Native RDFT Algorithm: Iterative. A similar iterative RDFT algorithm with radix $2k$ is given by

$$\mathrm{rDFT}_{2km}(0) = V^{2km}_{k,m} \cdot \left(\prod_{i=0}^{q} \left(I_{m/k^i} \otimes W_i^{2 \cdot k^{i+1}} \right) \cdot \left(I_m \otimes_\ell \mathrm{rDFT}_{2k}(s_{k,q-i}(0, 0, \lfloor \ell / k^i \rfloor)) \right) \right) \cdot L^{2km}_m,$$

where V, W, and L are permutations. V is implemented with the general method described in Chapter 5.3, while W and L are implemented using the bit matrix method in [20].

This algorithm is similar to the native RDFT algorithm with constant geometry above. However, the permutation (here, W) depends on stage index i: smaller values of i result in smaller permutations and lower cost. So, if iterative reuse is not used, this algorithm will have lower cost than the corresponding instance of the constant geometry algorithm, which requires permutations of size $2km$ in each stage. This tradeoff is analogous with the tradeoff between the Pease and Iterative Cooley-Tukey algorithms for the complex DFT, as explained above in Chapter 6.1.

6.4 Discrete Cosine Transform

This thesis considers an algorithm (given in [14]) for the discrete cosine transform of type two:

$$\text{DCT-2}_{2^k} = \sqrt{\frac{2}{2^k}} \cdot U_{2^k} \cdot \left(\prod_{s=k-1}^{0} S^{(s)}_{2^k} (I_{2^{k-s-1}} \otimes L^{2^{s+1}}_{2^s}) \right) P^H_{2^k},$$

where $S^{(s)}_{2^k}$ represents one computational stage:

$$S^{(s)}_{2^k} = (I_{2^{k-1}} \otimes_\ell M^{(s,\ell)}_2) \cdot \mathrm{diag}(g_k(\ell, s)) \cdot (I_{2^k-2} \otimes_\ell H^{(s)}_4) \cdot (I_{2^{k-1}} \otimes F_2),$$

and U, P^H, and L are permutations. P^H and L can be implemented using the bit matrix method if the problem size is a power of two, but U must always be implemented using the general streaming method.

Each of the terms in this algorithm easily maps to streaming with width $w \geq 4$. If the H_4 matrix is treated as a streaming permutation instead of a basic block, the algorithm may be streamed with $w = 2$, although this gives added overhead.

As the algorithm goes from left to right, the permutation $(I_{w^{k-s-1}} \otimes L^{2^{s+1}}_{2^s})$ grows more local, and thus less expensive to perform on streaming data. The product term can be mapped to iterative reuse, however this dependence of the permutation on the stage number s means that added overhead would be required to support all k permutations within one memory block (see discussion at the end of Chapter 5.3.1). Instead, other constant geometry algorithms could be used to support iterative reuse with less overhead (for example, [15]).

Chapter 7

Evaluation

This chapter presents a set of experiments evaluating the designs generated by Spiral based on the methods described in Chapter 4. The results shown here demonstrate the following aspects of the Spiral-generated hardware implementations:

1. **Cost/performance tradeoff.** Hardware cores generated by Spiral span a tradeoff space between cost (in resources required) and performance (typically throughput). Users can then make a choice along the cost/performance tradeoff space to pick the design that best meets a particular application's needs.

2. **Algorithmic and architectural freedoms.** The freedoms allowed by this system enrich the cost/performance tradeoff space. That is, the flexibility at both the algorithmic level and datapath mapping level contribute to the set Pareto set of best designs produced by the generation tool.

3. **Quality of designs.** The cost and performance of the designs automatically generated by Spiral are comparable to that of existing platform-tuned benchmarks, when available.

4. **Target flexibility.** The generation system allows flexibility in both the implementation target (FPGA, ASIC) and data type (fixed point, floating point).

5. **Problem flexibility.** Spiral is able to generate implementations for multiple transforms, including the DFT (both with two-power and non-two-power problem size), DCT, RDFT, and multi-dimensional transforms.

The remainder of this chapter is organized as follows. First, Section 7.1 describes the evaluation methodology used for this set of experiments, including FPGA and ASIC synthesis. Then, Section 7.2 gives a detailed design space exploration example including evaluation of the effects of different design parameters. Next, Section 7.3 shows results for implementations of the DFT on FPGAs including fixed and floating point data formats and both two-power and non-two-power problem sizes. Then, Section 7.4 shows results of an ASIC evaluation of fixed and floating point DFTs, and looks at power and area versus performance. Further, frequency scaling is discussed in Section 7.4.2 and Section 7.4.3 discusses joint power/area optimization under a fixed performance constraint. Lastly, Section 7.5 shows example results from other transforms, and Section 7.6 summarizes and provides some concluding remarks.

7.1 Methodology

The compilation flow described in Chapter 4 has been implemented in the following way. First, the algorithm generation, formula rewriting, and basic block matrix compilation stages have been built inside of Spiral. The tools to factor streaming permutations as described in Chapter 5 are built partially within Spiral and partially as a separate C program based on the MiniSat SAT solver [31], which is used to perform the factorization (5.3). 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.

All of the experiments here are conducted independent of the data bandwidth that may be available to the implemented core. This is done because the implementations produced by Spiral can be integrated in different types of systems. In some, all data may be produced and consumed on chip, potentially allowing a large amount of data to be processed by the core. Other systems may require input and output data to be transferred on and off chip before and after each computation; the throughput of these systems may be limited by available bandwidth.

The bandwidth required for each design presented in the experiments presented below can easily be calculated. Performance in these experiments is expressed as throughput in billions of samples per second. For a design running at t billion samples per second with a data type requiring b bytes per sample, the total amount of bandwidth (in gigabytes per second) needed is given by $2bt$.

For example, some of the experiments below consider 16 bit fixed point data types. For complex transforms (such as DFT), this means 4 bytes are required per sample. So, a throughput of 1 billion samples per second would require a total of 8 gigabytes per second (4 for input and 4 for output).

7.1.1 FPGA

This section describes the procedures used in the FPGA (field-programmable gate array) evaluations described in this chapter. All FPGA experiments done target the Xilinx Virtex-6 XC6VLX760 or the Xilinx Virtex-5 XC5VLX330 FPGAs (hereafter referred to as Virtex-6 and Virtex-5). 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.

In order to determine the maximum clock frequency obtainable for each design, it is often necessary to iterate over multiple design options and target frequencies. This is done using the Xilinx-provided Xplorer script, configured to use a maximum of four place and route runs to find the best frequency.

Arithmetic units. The experiments detailed here use two data types. For fixed point data types, the numeric computations are simply specified using the + and * operators. Single precision floating point is implemented using IP cores generated using the Xilinx LogiCore IP tool.

The Xilinx FPGAs contain dedicated arithmetic units called DSP slices (DSP48E on the Virtex-5 and DSP48E1 on the Virtex-6). DSP slices contain hard multipliers, accumulators, registers, and interconnect. The multipliers in these slices are used in fixed point and floating point multiplication. Additionally, the multipliers are used as shifters for floating point addition. The single precision floating point adder and multiplier each use two DSP slices, while the 16 bit fixed point multiplier uses one DSP slice.

Memory. Memories can be mapped to two types of structure on Xilinx FPGAs: block RAM (BRAM) or distributed RAM. BRAMs are 36kb dedicated hard memories built into the FPGA. The Virtex-6 contains 720 BRAMs; the Virtex-5 contains 288. Additionally, memory structures can be constructed out of the FPGA's lookup tables (LUTs) which are normally used as logic.

The Spiral hardware compiler can choose between distributed and block RAM by outputting directives for the Xilinx ISE tool. The user can control how Spiral decides whether a memory should be implemented as block or distributed RAM using two tunable methods. First, a budget-

based method allows the user to specify the maximum number of BRAM to use. Then, the tool globally sorts all memory structures needed in the design by size; it chooses which structures to map into the budgeted number of BRAMs in order to maximize utilization.

Secondly, a threshold option allows the user to specify a threshold memory size (in bytes). Any memory structure of that size or larger is then mapped to BRAM. For example, the user could say only to utilize a BRAM if the memory structure implemented would use 8 kilobits of it.

The experiments presented in this chapter use the budget method with a budget of 256 BRAMs. This represents approximately 35% of the BRAM on the Virtex-6 FPGA. For the designs considered here, this budget exceeds the memory requirement of the system; distributed RAM is not used for large memories.

7.1.2 ASIC

This chapter also contains a set of experiments evaluating the Spiral-generated cores when synthesized for an application-specific integrated circuit (ASIC). ASIC synthesis is performed using Synopsys Design Compiler version C-2009.06-SP5, and targets a commercial 65nm standard cell library. Area, timing and power estimates are obtained from Design Compiler post-synthesis.

Arithmetic units. Synopsys DesignWare is used to construct hardware for addition, subtraction, and multiplication (for both fixed point and floating point) in the following way. For each arithmetic unit, a combinational adder/subtractor/multiplier is instantiated, and pipelining is then performed by Design Compiler: a number of registers are placed before/after the arithmetic unit, and tool is run in retiming mode. Then, synthesized results are saved in binary format and are instantiated in the Spiral-generated designs. Different levels of pipelining are needed for different data types and different frequencies. The number of stages needed can be determined for a given frequency by finding the fewest number of registers needed to meet timing. Table 7.1 summarizes these results for 16 and 32 bit fixed point, and single precision floating point.

Memory. No memory compiler was accessible for the particular standard cell library used in these evaluations. So, CACTI version 6.5 [32], a memory estimation tool, is used to estimate power and area for all required memories.

	Latency (cycles)					
frequency (MHz)	16b add	16b mult	32b add	32b mult	fl. point add	fl. point mult
≤ 400	2	2	2	2	2	2
600	2	2	2	2	4	2
800	2	2	2	2	4	4
1000	2	2	2	4	6	6
1200	2	4	2	6	6	6
1400	2	6	2	6	8	8
1600	2	6	2	8	10	10
1800	2	10	4	10	12	14
2000	2	10	6	12	18	18

Table 7.1: Latency (cycles) of arithmetic operators at given frequencies.

7.2 Design Space Exploration Example

In order to examine how different design space freedoms (algorithm, radix, iterative reuse, streaming reuse) contribute to the space of designs, this section provides a detailed evaluation of Spiral-generated implementations of DFT_{256} and DFT_{1024} on a Xilinx Virtex-6 FPGA. Figure 7.1 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 r and the implementation's depth d (for iterative reuse). The diamond-shaped markers indicate designs with full IR (iterative reuse), 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.

As discussed above in Chapter 6.1, Spiral generates hardware cores for the DFT using different algorithms, depending on the situation. In Figure 7.1, the algorithm can be determined based on the parameters displayed. First, the Pease FFT algorithm (2.13) is used when full IR is used (that is, $d = 1$). So, for DFT_{256}, the black, gray, and white diamonds represent the Pease FFT with radix $r = 2, 4$, and 16, respectively. For DFT_{1024} the same data markers represent the same algorithm with radix $r = 2, 4$, and 8.

When IR is not used (represented as circles and squares), two algorithms are used, depending on the radix r: the Iterative FFT (2.15) and the Mixed-Radix FFT (2.16). If the problem size n can

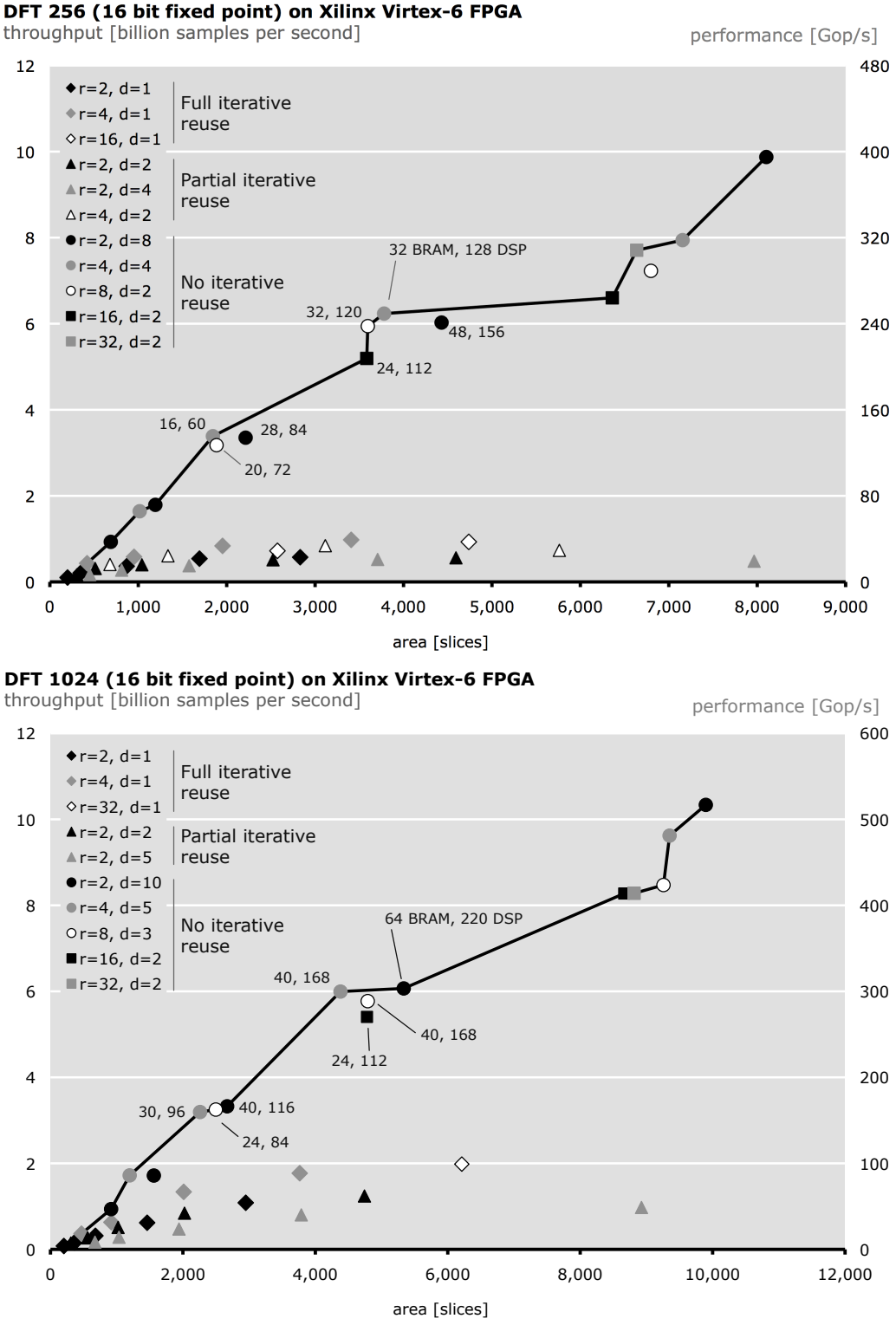


Figure 7.1: Exploring DFT_{256} and DFT_{1024}, fixed point, FPGA, throughput versus slices. Data labels indicate the number of block RAMs and DSP48E1 slices required. Data markers are used to illustrate values of radix r and depth d.

be represented as $n = r^k$ for integer k, then the Iterative FFT algorithm is used directly. If that is not the case, then the Mixed-Radix algorithm decomposes the problem into two stages: one that can be expressed as a power of radix r, and one smaller stage: $n = r^k \cdot n/(r^k)$. For example, DFT_{1024} can be computed with three radix 8 stages and one radix 2 stage: $1024 = 8^3 \cdot 2$.

This evaluation omits designs that use the Mixed-Radix algorithm with IR (iterative reuse) designs (for example, implementing DFT_{1024} with an iteratively-reused implementation of radix 8 DFT_{512} and one stage of DFT_2), although designs of this kind are easily generated using the proposed system.

The black line indicates the Pareto optimal set of designs for each graph. These designs are those that give the best tradeoff between the two metrics being considered (here, throughput and area). So, if a designer only cared about those two measurements, the points along the line are the only ones that he or she must consider. However, other costs are associated with FPGA implementations: required memory (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 tradeoff space. For DFT_{256}, the fastest design is 76× faster than the slowest, but it requires 33× the number of slices, 16× the number of BRAMs, and 59× the number of DSP slices. Similarly, the fastest DFT_{1024} implementation is 132× faster than the slowest, but it uses 49× the slices, 96× the BRAM, and 107× the DSP slices.

In each graph, 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 $w = 4, 8, 16, 32$. In both plots, 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 (while the IR designs previously discussed provide less

extra throughput given the extra area cost). 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, in the DFT_{256} experiment, there are three points near the Pareto front at approximately 3.5 billion samples per second and approximately 2,000 slices. 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.

It may seem counterintuitive that the radix 4 design can provide better results here than the radix 8, since in general higher radices lead to lower arithmetic and permutation costs. However, the difference here lies in algorithm: at radix 8, the overhead of using the Mixed Radix algorithm to decompose DFT_{256} into DFT_4 and DFT_{8^2} offsets the savings of using a radix 8 algorithm for a portion of the computation. In the DFT_{1024} plot, the equivalent radix 4 and 8 designs are much closer: the radix 8 design uses slightly more area but fewer BRAMs and DSP slices. 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).

7.3 Discrete Fourier Transform on FPGA

This section shows results of a detailed study of implementations of the discrete Fourier transform on FPGA. Section 7.3.1 discusses fixed point implementations, Section 7.3.2 discusses floating point implementations, and Section 7.3.3 shows implementations when the problem size is not a power of two.

7.3.1 Fixed Point

Figures 7.2 and 7.3 show throughput (in billion samples per second) versus area (in slices) of the discrete Fourier transform of size 64, 256, 1024, and 4096 with 16 bit fixed point data format on

the Xilinx Virtex-6 FPGA. The secondary axis presents performance in pseudo gigaoperations per second calculated for all algorithms assuming $5n \log_2 n$ operations per n point transform. Here, gray diamonds represent Spiral-generated designs across the degrees of freedom discussed in Chapter 7.2. The solid line connects the Spiral-generated Pareto optimal points. Lastly, the black circles represent the four designs from the Xilinx LogiCore FFT IP library [33]. These designs were implemented using LogiCore and evaluated using the same Xilinx tools as the Spiral-generated designs.

As discussed in Chapter 7.2, a number of design freedoms are used along the Pareto front, including designs with and without iterative reuse, and designs with streaming width w up to 32.

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

Similar trends are seen across the experiments as the problem size increases. However, the absolute area requirements increase with the problem size. This is primarily caused by the added number of stages in designs without iterative reuse, and in the added cost associated with larger permutations and diagonal matrices. Also, it is important to note that although the throughput stays roughly the same as the problem size increases, the performance (operations / second) increases.

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 a sense optimized for latency-based computation because they only employ a small amount of overlapping of multiple problems.

7.3.2 Floating Point

Next, Figures 7.4 and 7.5 repeat the same set of DFT experiments on FPGA using single precision floating point (32 bit). Again, the x-axis shows FPGA area in terms of slices, and y-axis shows

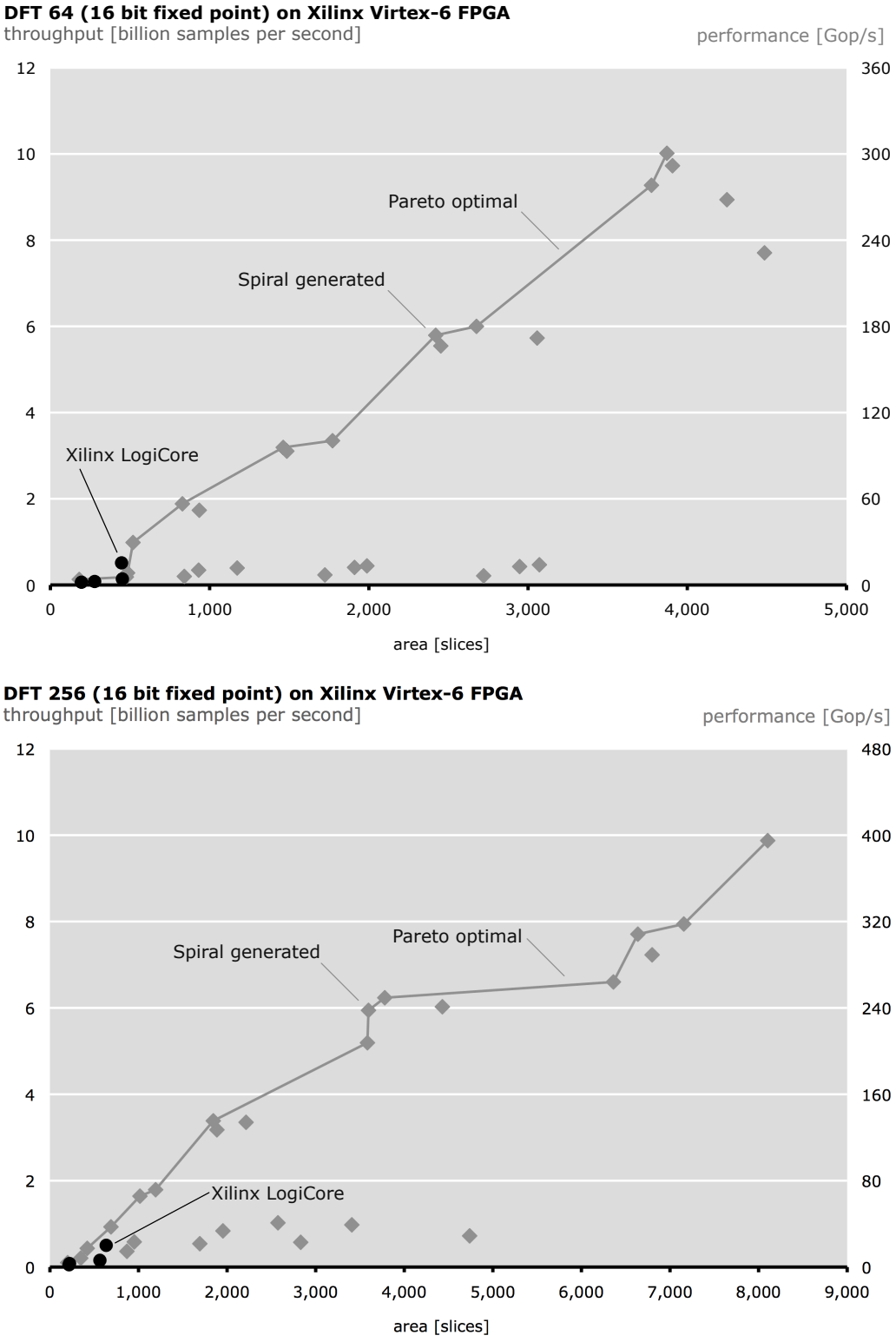


Figure 7.2: DFT_{64} and DFT_{256}, fixed point, FPGA, throughput versus slices compared with Xilinx LogiCore.

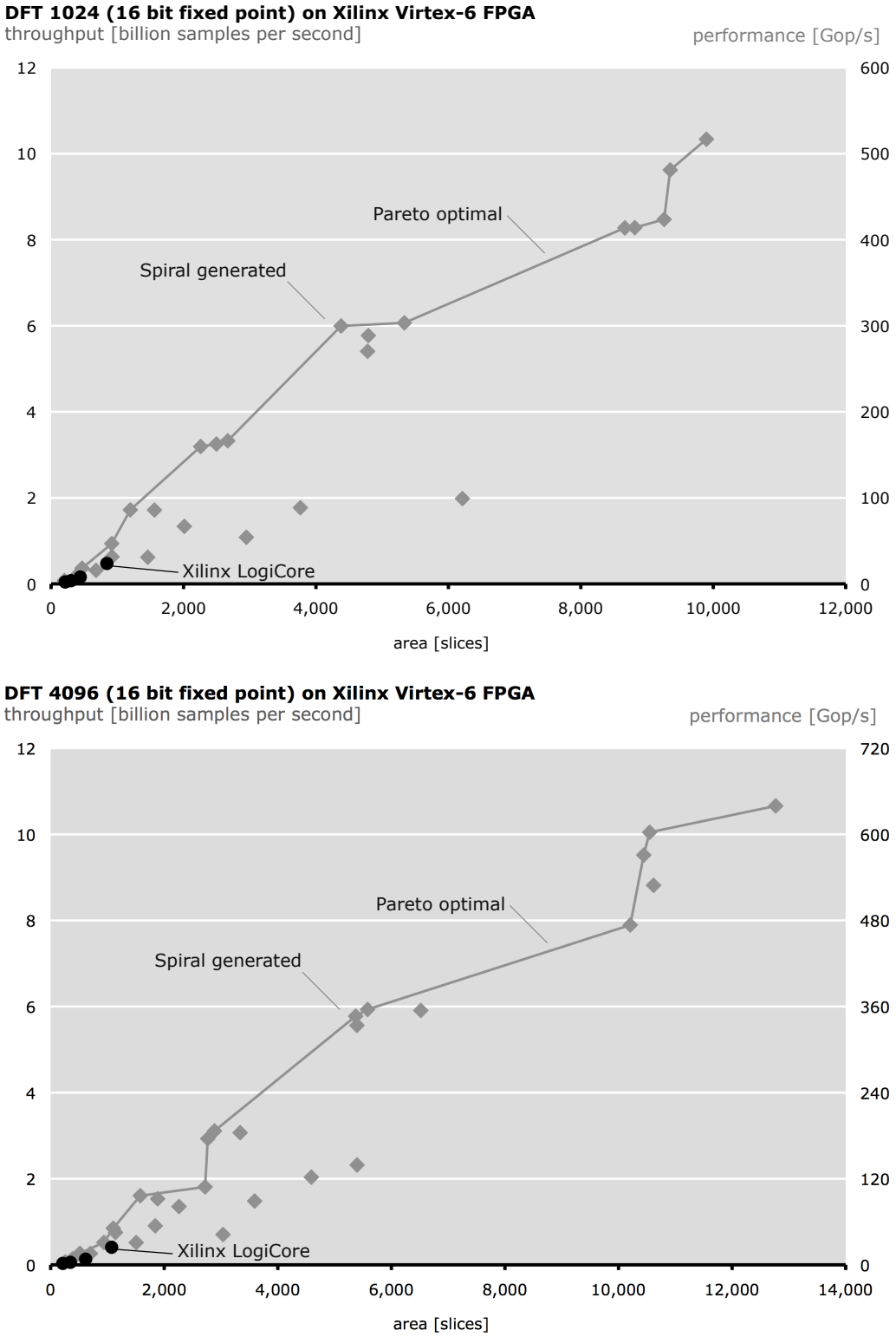


Figure 7.3: DFT_{1024} and DFT_{4096}, fixed point, FPGA, throughput versus slices compared with Xilinx LogiCore.

throughput in billions of samples per second (left) and performance, now in billions of floating point operations per second (Gigaflop/s), on the right.

The Xilinx LogiCore FFT [33] 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 four architectures provided by Xilinx LogiCore (with 25 bit twiddle factors) are shown as black circles in Figures 7.4 and 7.5. Although the Xilinx designs have lower-cost fixed point arithmetic units, they compare closely with the Spiral-generated designs. The biggest difference can be seen on the largest/fastest of the four Xilinx designs.

Additionally, for all but DFT_{64}, a data point corresponding to the 4DSP floating point FFT processor [34] is shown (black triangle). This is a commercially-available IP core that is capable of performing the DFT on a range of input sizes. That is, it is a floating point DFT processor with runtime-configurable length. Area and performance numbers were estimated based on data given in its data sheet [34]. As shown, the 4DSP core has a considerable area overhead relative to the Spiral-generated or Xilinx designs. However, its runtime configurability leads to added overhead, so a direct comparison cannot be made.

Comparing the cost and performance of the Spiral-generated floating point designs (Figures 7.4 and 7.5) to the fixed point implementations (Figures 7.2 and 7.3), the main difference lies in the base cost of computation. This leads to a much higher cost to achieve the same throughput in a floating point implementation as a fixed point implementation. This cost is also affected by added overhead in permutations, diagonal matrices (ROMs to store constants) and registers within the system, which now must store 32 bits instead of 16.

Further, the higher base cost reduces the space of feasible architectural options. In the fixed point experiments, a streaming width up to 32 complex words per cycle was implemented. Here, the maximum streaming width achieved (when iterative reuse is not used) is 16 words per cycle.

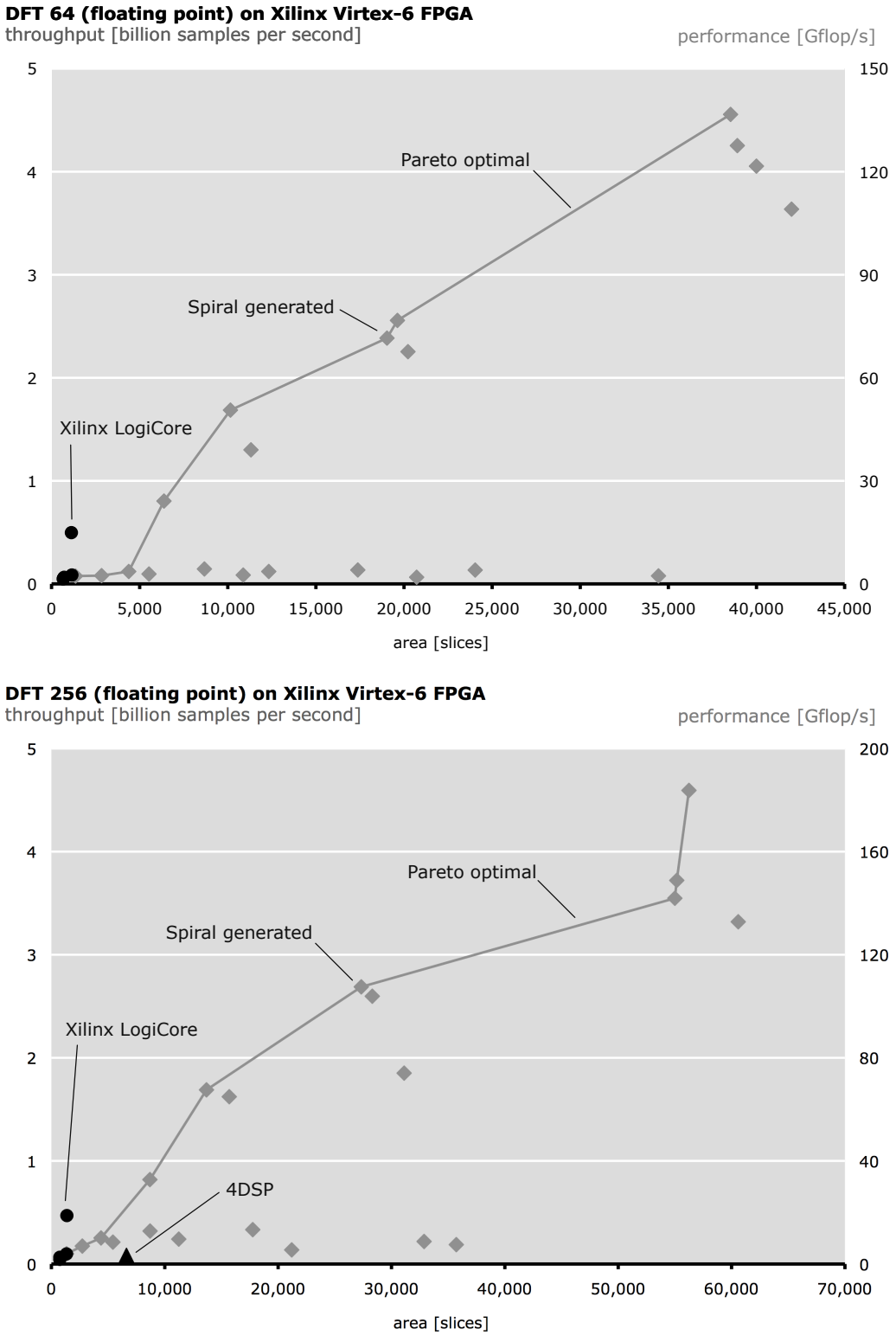


Figure 7.4: DFT_{64} and DFT_{256}, floating point, FPGA, throughput versus slices compared with Xilinx LogiCore and 4DSP FFT cores.

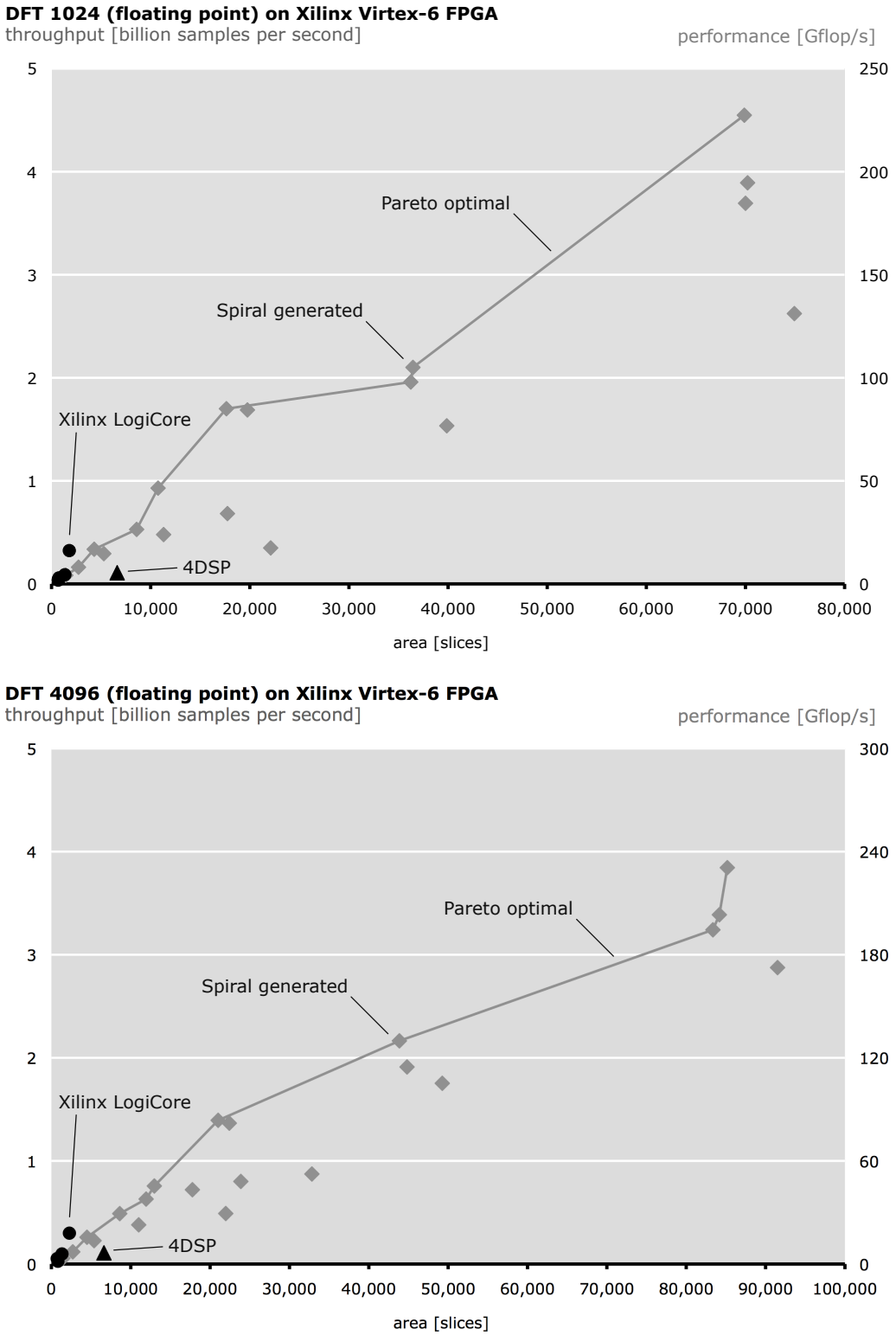


Figure 7.5: DFT_{1024} and DFT_{4096}, floating point, FPGA, throughput versus slices compared with Xilinx LogiCore and 4DSP FFT cores.

7.3.3 DFT with Non-Power-of-Two Problem Size

When the problem size n of DFT_n is not a power of two, a variety of algorithms (discussed in Chapter 6.1) can be utilized. Here, four values of n are considered that illustrate four different classes of problem size. Figures 7.6 and 7.7 examine the throughput versus area for each design. Data labels are used to indicate the number of Virtex-5 DSP slices consumed by each point in the Pareto-optimal set. This evaluation was published in part in [35].

Large prime size. First, Figure 7.6 (top) shows throughput versus area for implementations of DFT_{499}. Because 499 is a large prime number, the only applicable option among those considered is the Bluestein FFT algorithm. Each black diamond represents one Bluestein-based datapath, and the black line illustrates the designs in the Pareto optimal set. The slowest design requires approximately 1,500 slices and has a throughput of 11 million samples per second. The fastest is about $18\times$ larger but $60\times$ faster.

Composite number with larger prime factors. Next, Figure 7.6 (bottom) shows results for DFT_{405}. In addition to the Bluestein algorithm, the mixed radix FFT algorithm can be applied to this problem; it decomposes DFT_{405} into DFT_{3^4} and DFT_5. This enables two additional designs (with streaming width $3 \cdot 5 = 15$), shown as gray circles. Although the larger of these designs provides a much higher throughput than a similarly sized Bluestein-derived design, it is important to note that the cross-over point (where the mixed-radix designs become the better choice) is large (at approximately 15,000 slices). So, if design requirements require a smaller, lower throughput core, the Bluestein algorithm is a better choice.

Composite number with smaller prime factors. Third, Figure 7.7 (top) illustrates throughput and area for DFT_{432}. Similar to the previous example, DFT_{432} can be implemented using the mixed radix algorithm. However, unlike DFT_{405}, here DFT_{432} can be decomposed into smaller radices: 2 and 3 (because $432 = 2^4 \cdot 3^3$). Since the radices are smaller, the mixed radix algorithm can be used with more options for streaming width, yielding a larger set of Pareto-optimal designs than for DFT_{405}. The Bluestein designs are still the best choice for the smallest/slowest designs, but the cross-over point (about 9,000 slices) is lower than in the previous example.

Power of small prime. Lastly, Figure 7.7 (bottom) shows results for $\mathrm{DFT}_{243} = \mathrm{DFT}_{3^5}$. Because the problem size is a power of three, the radix 3 Pease and Iterative FFTs are applied directly

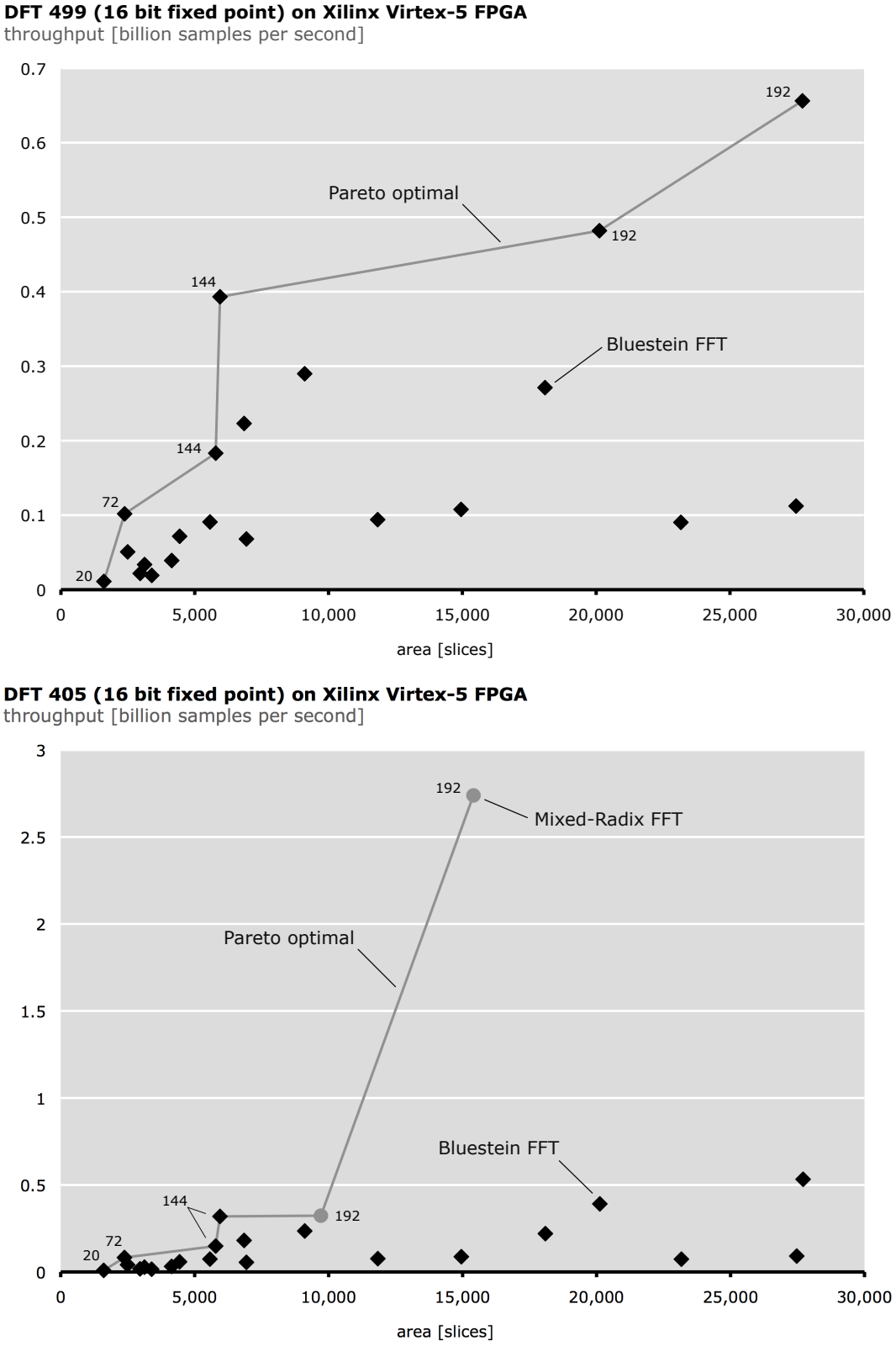


Figure 7.6: DFT_{499} and DFT_{405}, fixed point, FPGA, throughput versus slices.

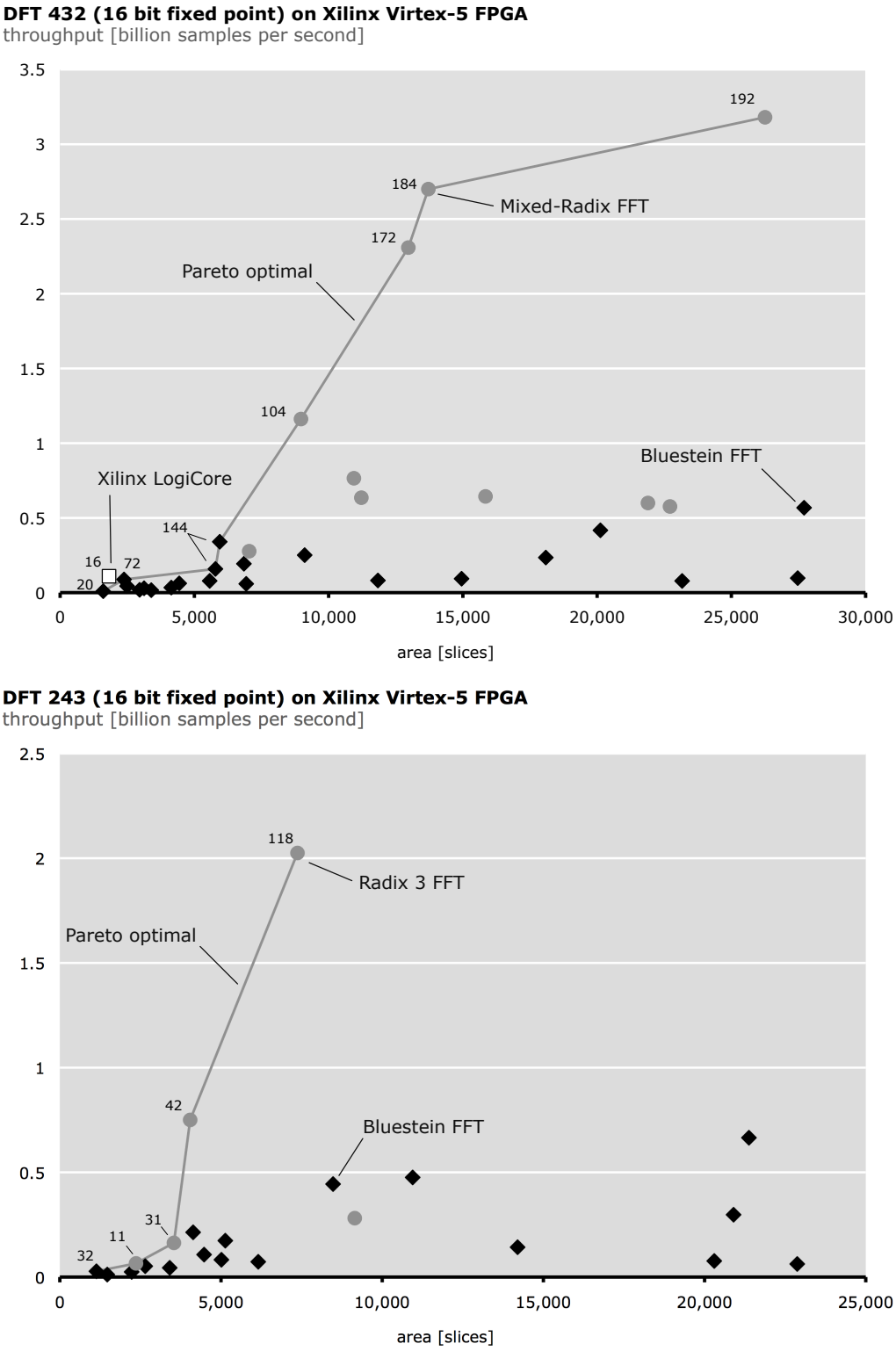


Figure 7.7: DFT_{432} and DFT_{243}, fixed point, FPGA, throughput versus slices, compared with Xilinx LogiCore DFT.

(shown as gray circles). Here, the cores built with the radix 3 algorithms surpass the performance of the Bluestein cores at a much lower area than in the other problems, since the single-radix design can be built with less logic than in the previous multi-radix designs.

Comparison. The Xilinx LogiCore library contains a non-two-power sized DFT core, which is a processor that performs a number of non-two-power-sized DFT computations, and is run-time configurable in transform length. Of the example sizes considered in this evaluation, the LogiCore DFT can only perform DFT_{432}. Its performance and area are given as a white square in Figure 7.7 (top).

7.4 Discrete Fourier Transform on ASIC

As described above in Chapter 7.1.2, Spiral-generated hardware cores can be synthesized for a commercial 65nm standard cell library. First, Section 7.4.1 presents a baseline evaluation of designs with maximum clock frequency. Then, Section 7.4.2 examines frequency scaling and its effect on performance, power, and area. Lastly, Section 7.4.3 uses frequency scaling to implement a set of designs that meet a given throughput target while allowing a tradeoff between power and area.

7.4.1 Baseline: Maximum Frequency

First, this section presents an evaluation of the DFT on ASIC. Here, designs are synthesized with the goal of maximizing the clock frequency. Figures 7.8–7.11 show throughput versus power and area for DFT_{64}, DFT_{256}, DFT_{1024} and DFT_{4096} using 16 bit fixed point data. Figures 7.12–7.15 repeat the same experiments utilizing single precision floating point.

In these plots, the y-axis represents throughput in billions of samples per second (left) and billions of operations (or floating point operations) per second. Additionally, two data markers are used: black diamonds for cores with iterative reuse, and gray circles for fully streaming designs, (that is, those without iterative reuse). 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. Also similar to the FPGA experiments, designs with iterative reuse comprise a larger portion of the

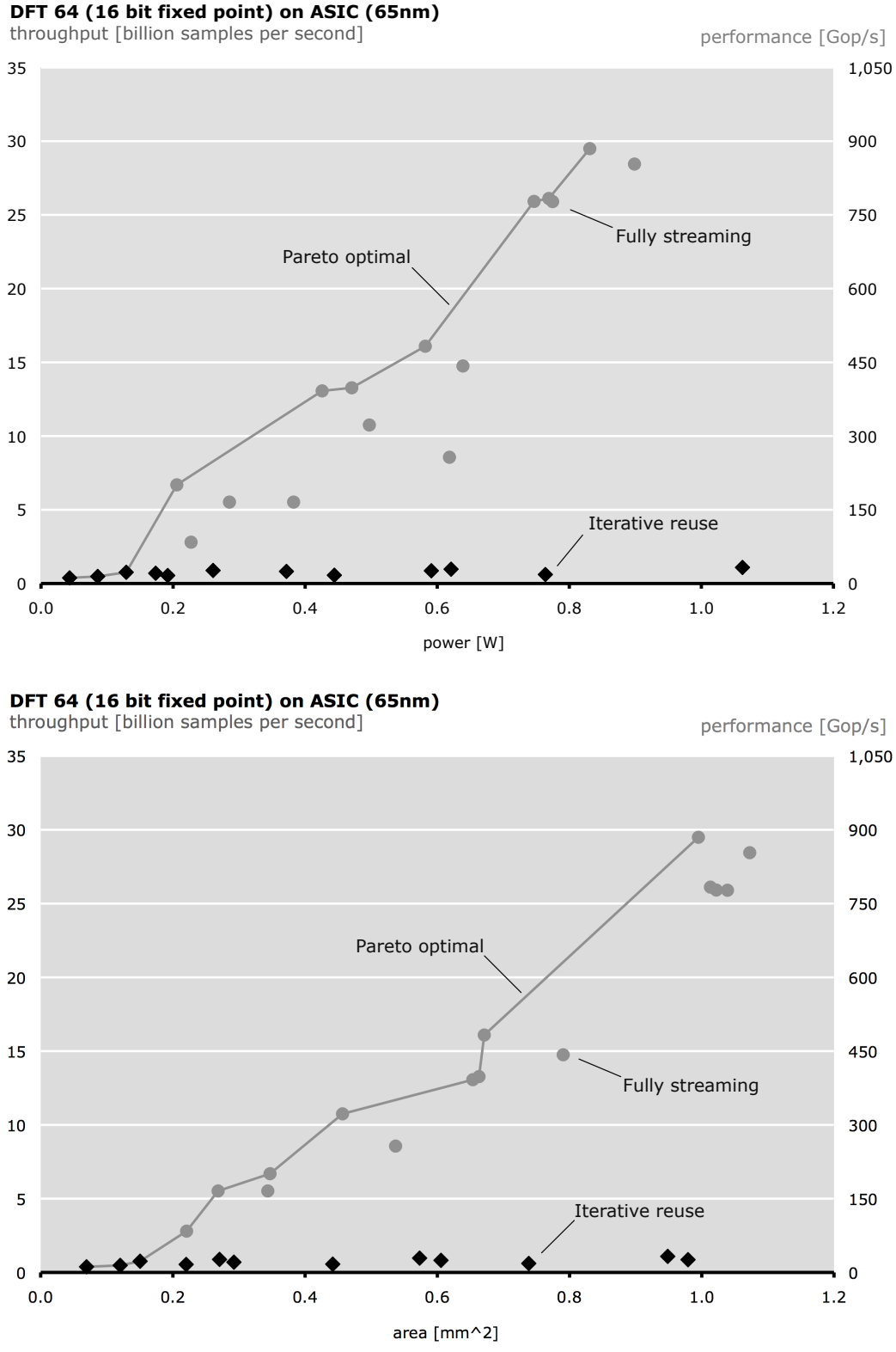


Figure 7.8: DFT_{64} throughput versus power (top) and area (bottom), fixed point on 65nm ASIC.

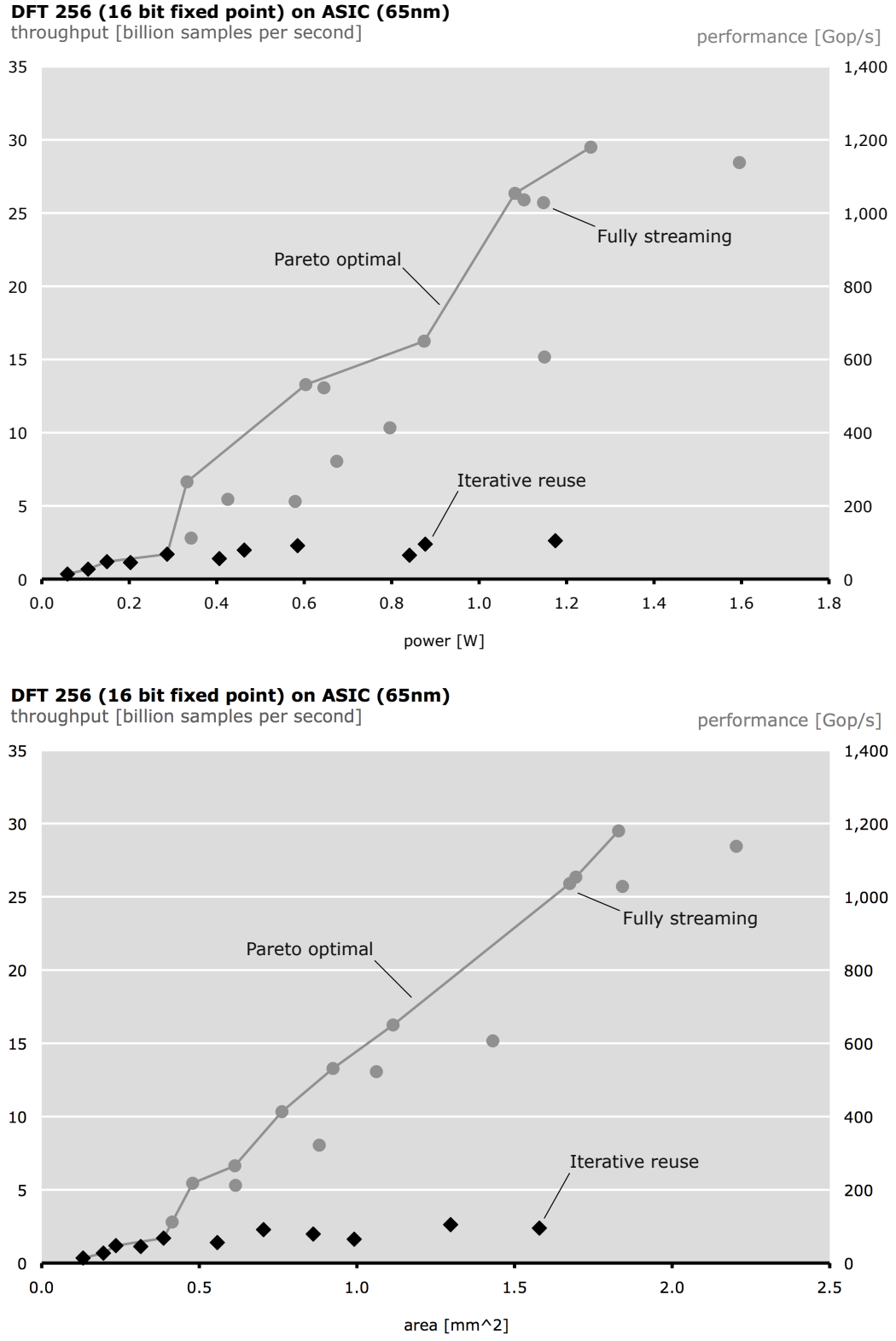


Figure 7.9: DFT_{256} throughput versus power (top) and area (bottom), fixed point on 65nm ASIC.

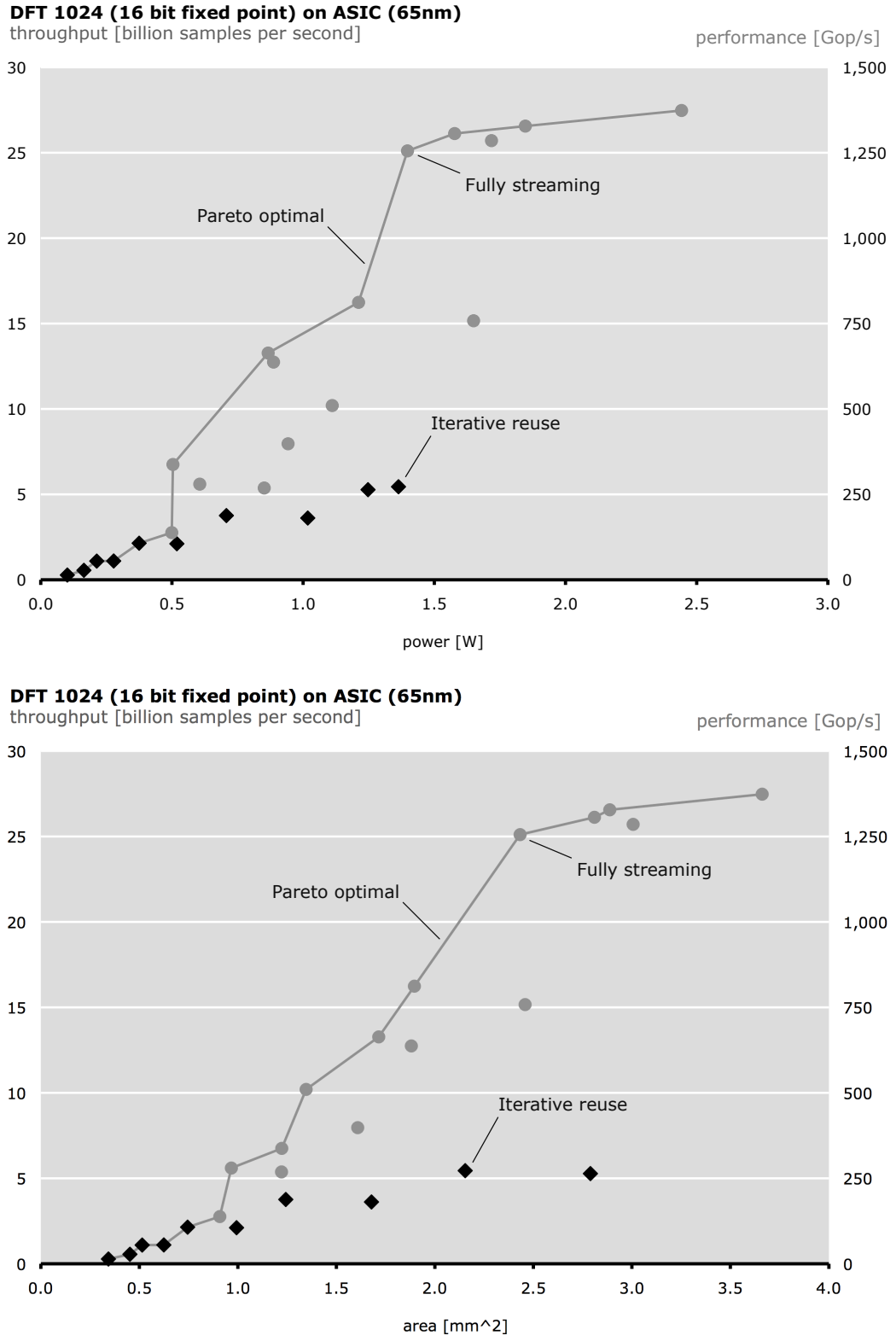


Figure 7.10: DFT_{1024} throughput versus power (top) and area (bottom), fixed point on 65nm ASIC.

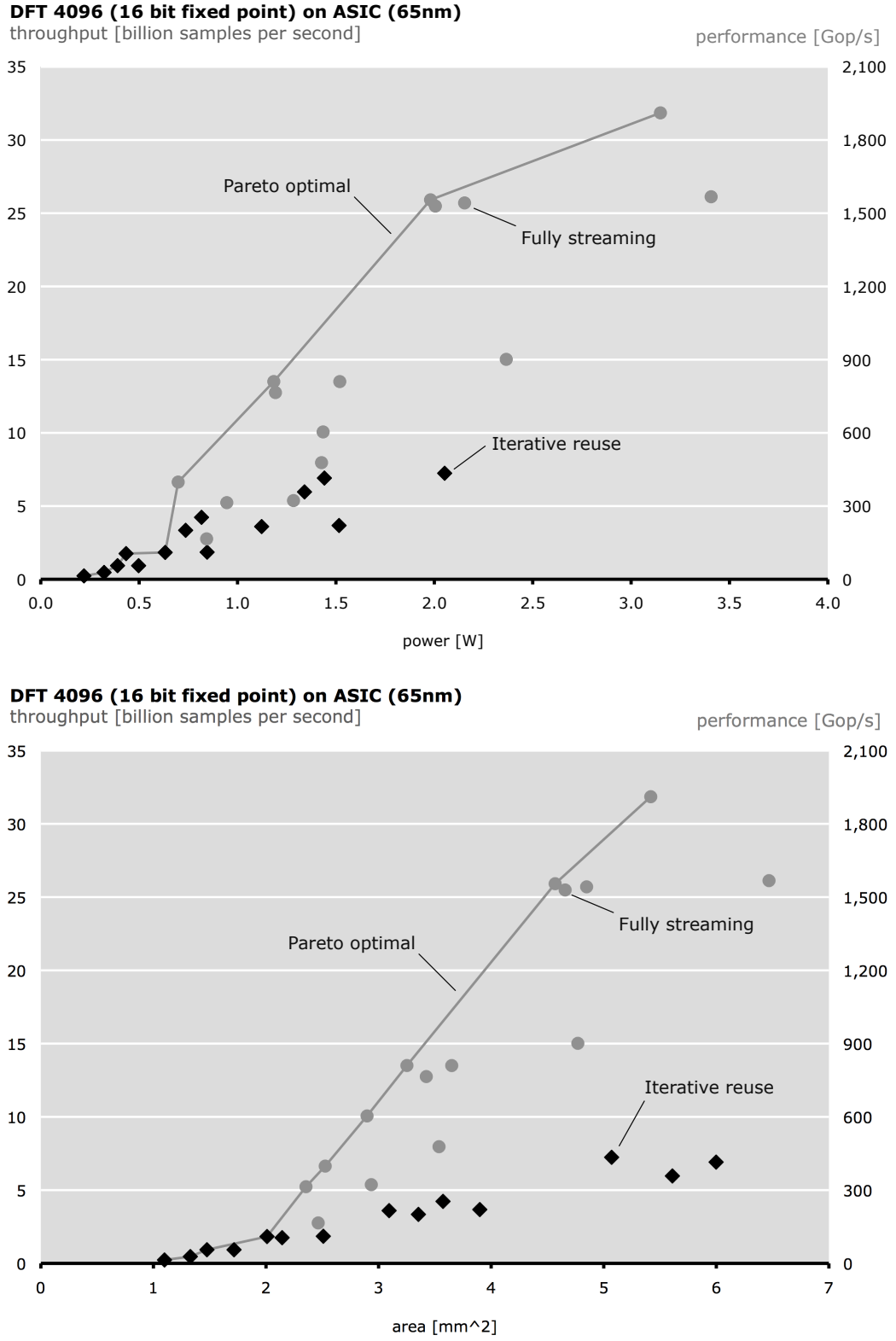


Figure 7.11: DFT_{4096} throughput versus power (top) and area (bottom), fixed point on 65nm ASIC.

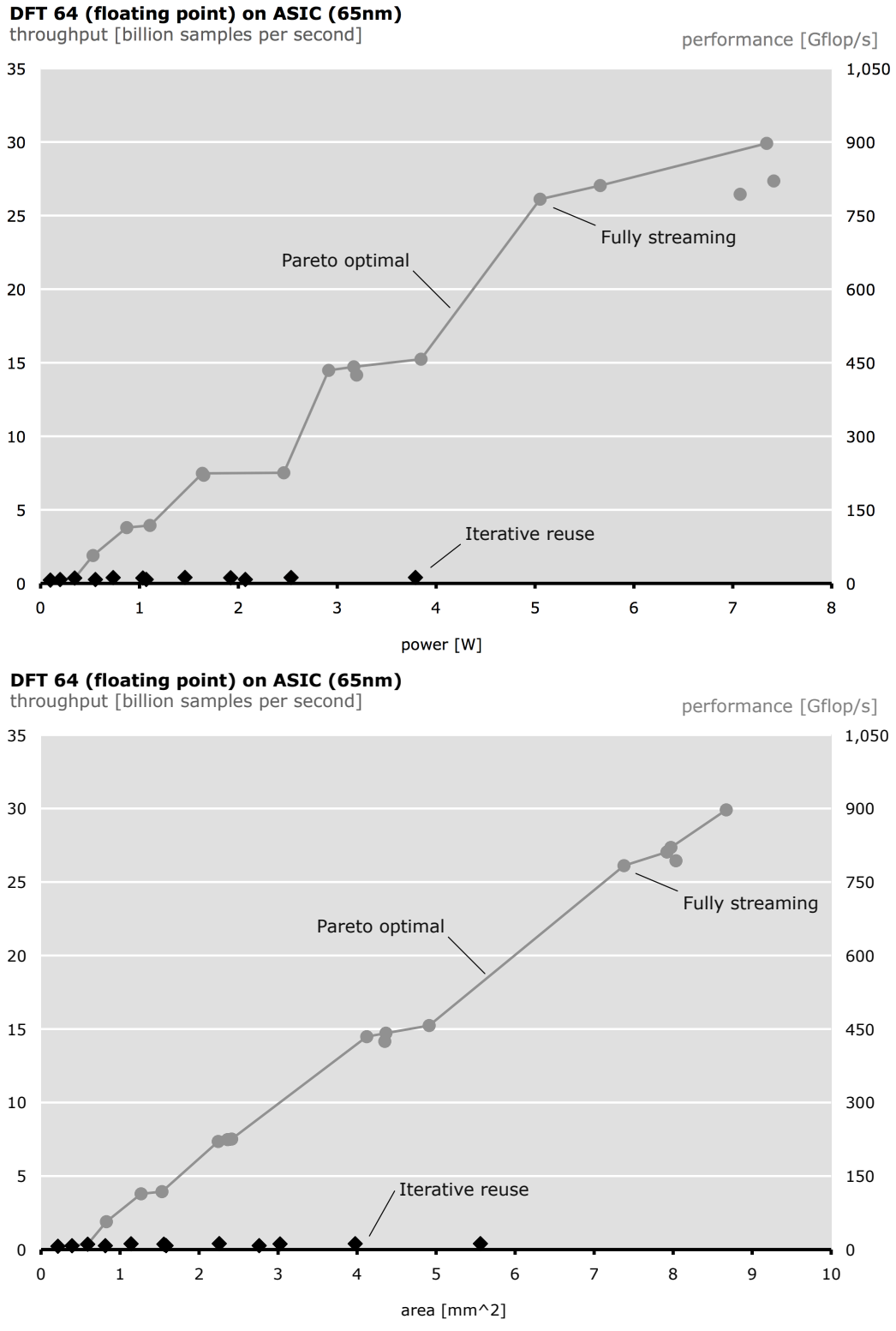


Figure 7.12: DFT_{64} throughput versus power (top) and area (bottom), floating point on 65nm ASIC.

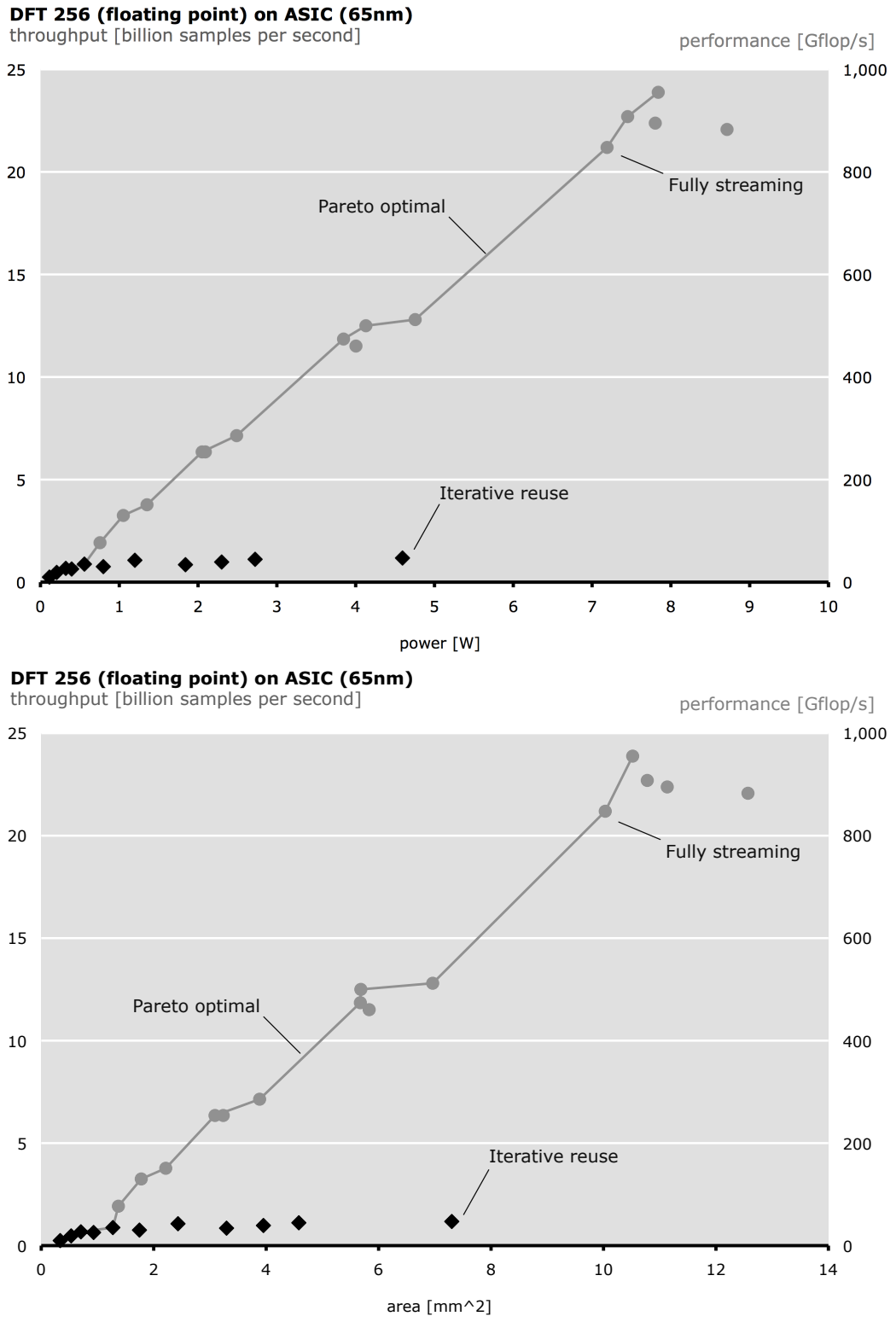


Figure 7.13: DFT_{256} throughput versus power (top) and area (bottom), floating point on 65nm ASIC.

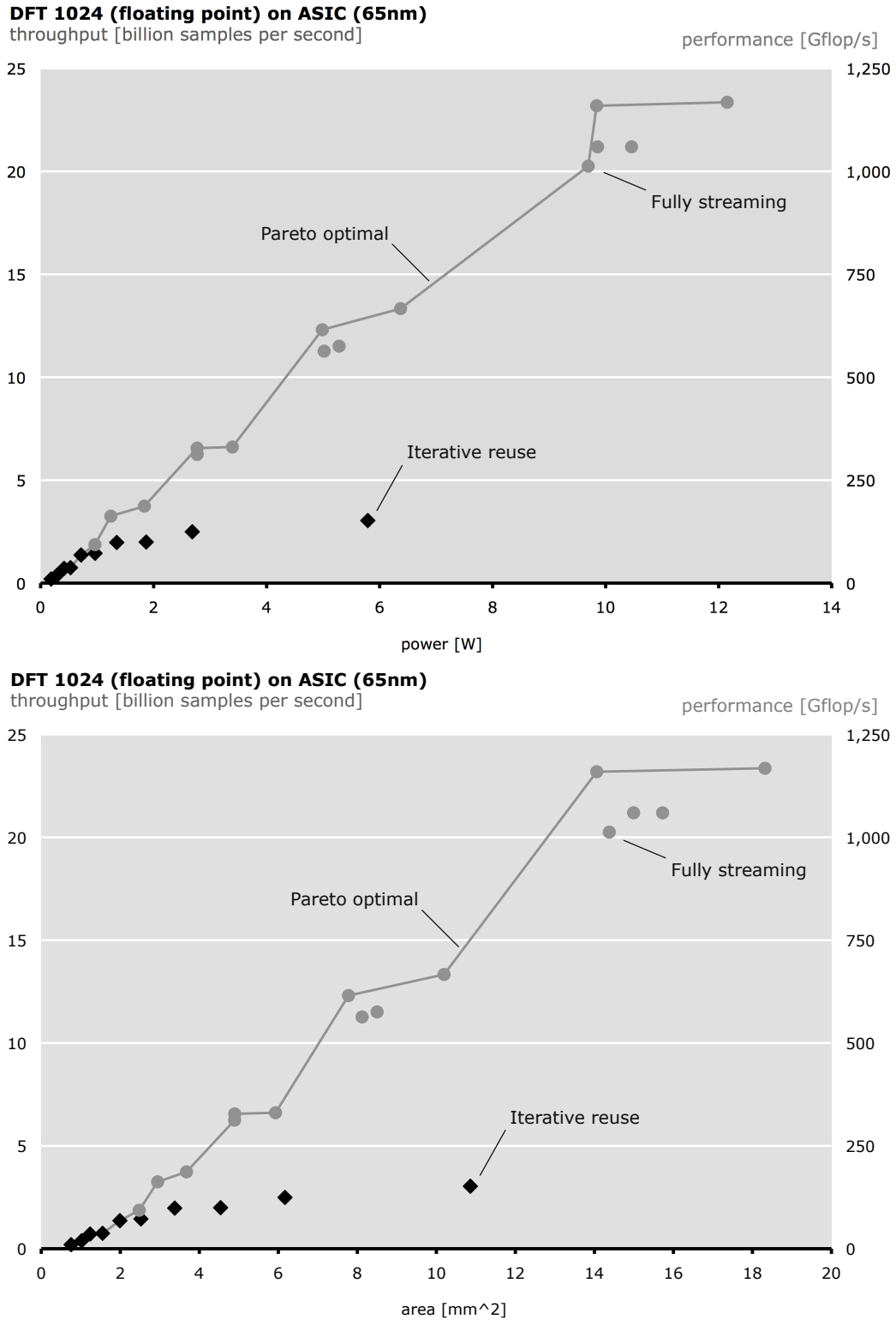


Figure 7.14: DFT_{1024} throughput versus power (top) and area (bottom), floating point on 65nm ASIC.

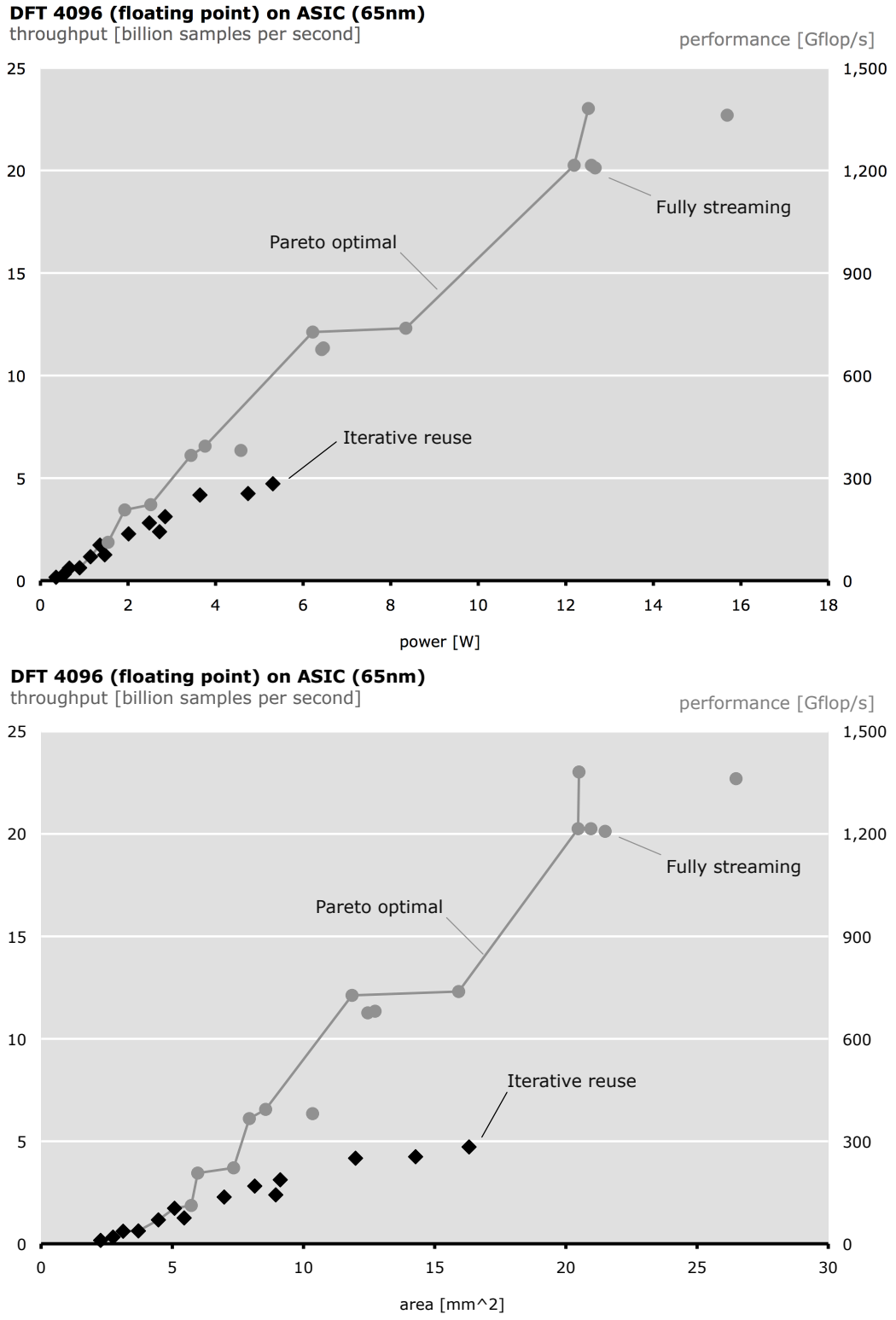


Figure 7.15: DFT_{4096} throughput versus power (top) and area (bottom), floating point on 65nm ASIC.

Pareto optimal set of designs when the problem size is large.

Floating point designs are considerably more expensive than their 16 bit fixed point counterparts. Further, the increased area and complexity lead to lower synthesized frequencies. For example, the fastest point considered in the fixed point DFT_{1024} experiment can be synthesized to run at 858 MHz, producing a throughput of 27.5 billion samples per second. The corresponding floating point design only reaches 730 MHz, giving a slower throughput of 23.4 billion samples per second. Further, the area and power requirements increase dramatically: from 3.7 mm^2 and 2.4W to 18.3 mm^2 and 12.2 W.

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

7.4.2 Frequency Scaling

The previous results showed throughput versus power and area when all designs were synthesized targeting the maximum frequency. However, when power efficiency is a concern, it is also necessary to consider designs running at lower clock frequencies. This section demonstrates the trends in performance, power, and area when designs are re-synthesized 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 shown in Table 7.1).

Figure 7.16 revisits the 16 bit fixed point DFT_{1024} experiment detailed above. Now, the Pareto suboptimal points are removed, leaving only the Pareto front, shown as black diamonds. Then gray diamonds are used to show how throughput, power, and area of five of the Pareto optimal designs are effected as the designs are regenerated and re-synthesized targeting 200, 400, and 600 MHz.

As the frequency is decreased, throughput and power are reduced commensurately. As seen in the bottom graph of Figure 7.16, area is reduced only slightly as frequency decreases.

This shows that it can be more power-efficient to generate a larger, more parallel design and run it at a lower frequency than to build a smaller design running at a higher clock frequency. These larger designs have more algorithmic freedom, so they can be built with more efficient algorithms, using higher radices.

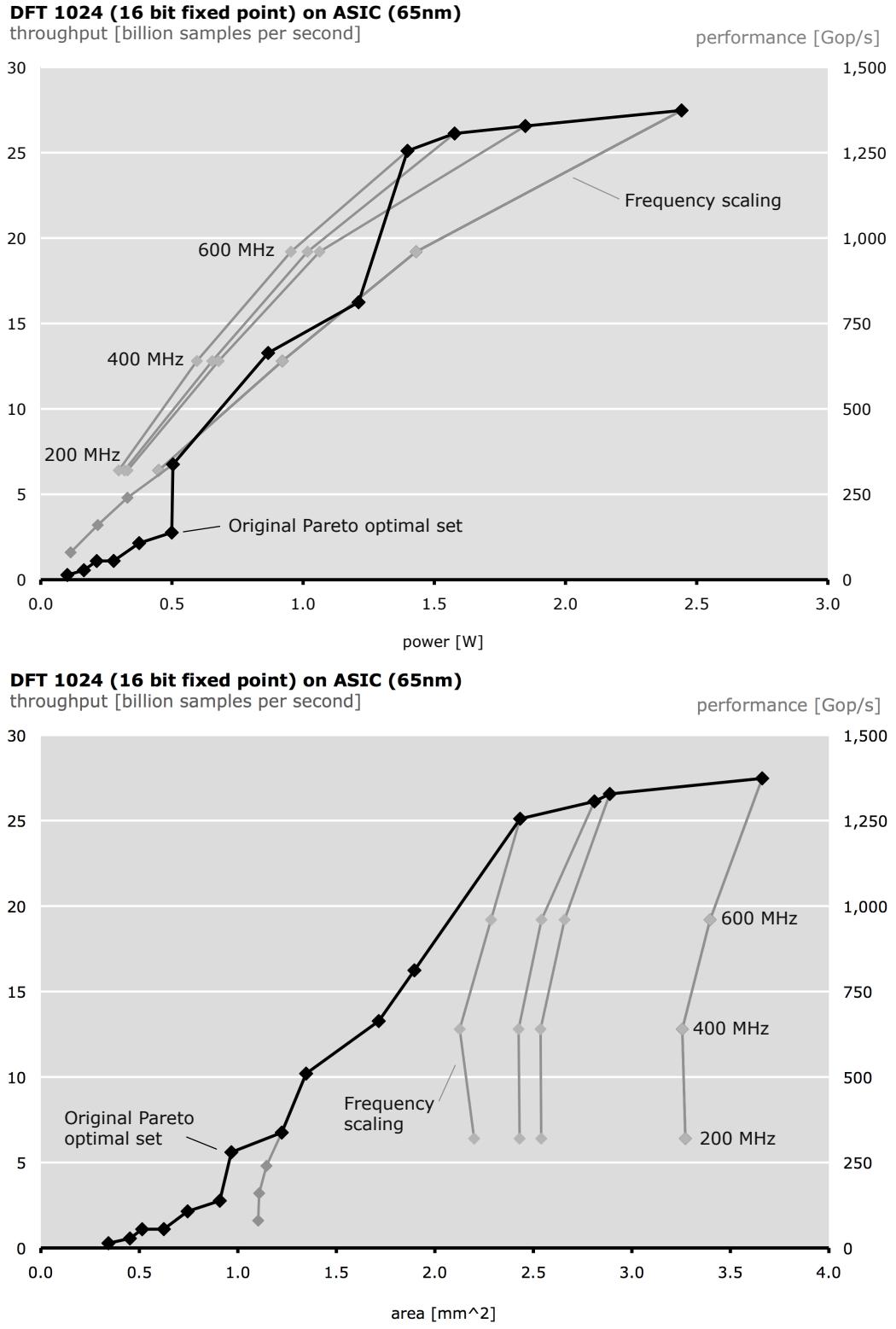


Figure 7.16: DFT_{1024}, fixed point on ASIC: throughput versus power (top) and area (bottom) with frequency scaling.

As an example, Figure 7.16 shows a point from the original Pareto optimal set at approximately 7 billion samples per second, with area of 0.9 mm^2 consuming approximately 0.5 Watts (while clocked at 1.4 GHz). However, similar throughput can be obtained using a larger design clocked at 200 MHz, which requires only 0.3 W. However, this leads to an in increase in area; the larger design requires 2.2 mm^2. This type of tradeoff 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 points from one design, more evaluations (not shown here) confirm these trends.

7.4.3 Power/Area Optimization Under Throughput Requirement

Often, a design requires that a transform be implemented to reach a given throughput requirement. The frequency scaling experiments discussed above can be modified to reflect the type of exploration required for such a situation.

Figure 7.17 again studies fixed point implementations of DFT_{1024} on 65nm ASIC in the presence of frequency scaling. However, now the designs are not synthesized at arbitrary frequencies. Instead, each is synthesized at precisely the frequency needed to yield a performance of a fixed throughput target: two billion samples per second, illustrated with the dashed line. As shown in the top plot, reducing the frequency of the more parallel designs yields much lower power at the target throughput. However, as shown in the bottom plot of Figure 7.17, this requires a much larger area than the less parallel designs.

This process results in nine different designs, each with equal throughput. Some require less power but more area; others the opposite. Figure 7.18 collects the set of possible tradeoffs and displays them graphically. The y-axis shows the power of each design, while the x-axis shows the area. The black line highlights the five designs that 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.

Although the results here and in Section 7.4.2 examine frequency scaling only for ASICs, this technique should extend to FPGA as well.

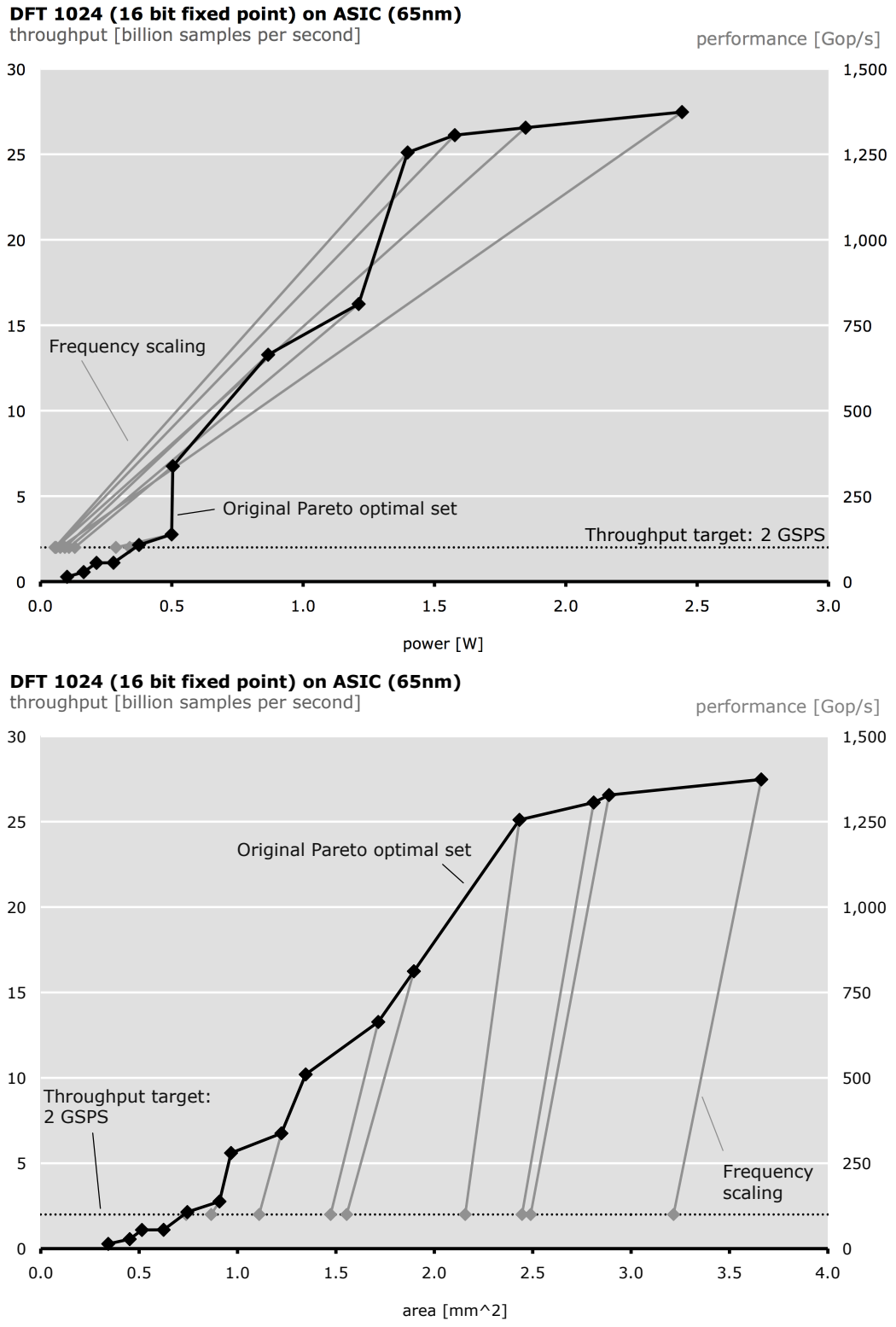


Figure 7.17: DFT_{1024}, fixed point, throughput versus power/area on ASIC, frequency scaled to reach 2 billion samples per second.

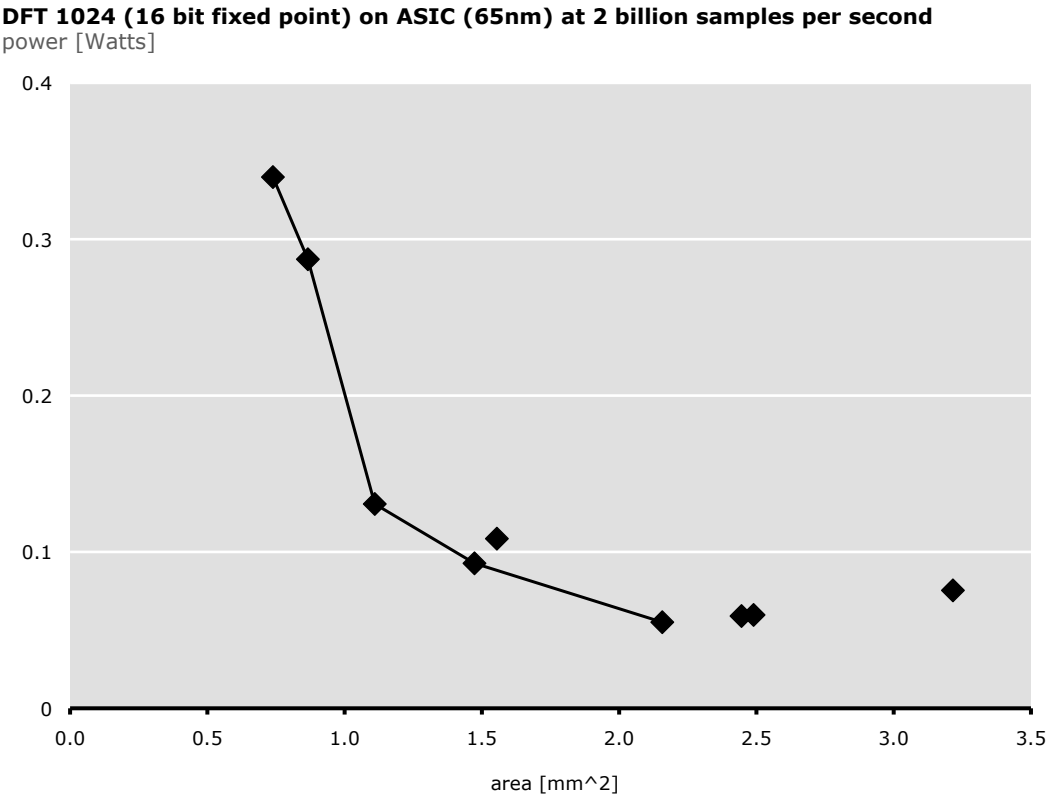


Figure 7.18: DFT_{1024}, fixed point, power versus area, fixed throughput of 2 billion samples per second.

7.5 Other Transforms

Although most of this evaluation has focused on the discrete Fourier transform, Spiral generates designs for other transforms as well. This section demonstrates results for the two-dimensional discrete Fourier transform DFT-2D, the real discrete Fourier transform RDFT, and DCT-2, the discrete cosine transform of type 2.

7.5.1 Two-dimensional DFT

As discussed in Chapter 6.2, the row-column algorithm for computing the two-dimensional DFT consists of two iterative product terms, giving two opportunities for iterative reuse: the outer term implemented with depth d_1, and an inner term with depth d_2, as illustrated in Figure 6.4. These two degrees of freedom in iterative reuse are combined with freedom in the streaming width as well as the algorithmic freedoms (including radix), yielding a wide space of designs to explore.

Figure 7.19 illustrates throughput versus area for DFT-2D$_{16\times16}$ and DFT-2D$_{64\times64}$, with 16 bit fixed point data type on Xilinx Virtex-6 FPGA. Four different data markers are used to illustrate whether each design employs iterative reuse on the inner and outer product.

The gray line highlights the Pareto optimal set of designs, which includes designs where: both products are iteratively reused (diamonds), only the outer product is iteratively reused (circles), and neither product is iteratively reused (squares). Although not visually illustrated, the Pareto optimal set also depends on the freedoms in streaming width, algorithm, and radix to span the full tradeoff space. If any of those freedoms were restricted, the set of Pareto optimal points would be reduced.

7.5.2 Real Discrete Fourier Transform

Four algorithms for the real discrete Fourier transform (RDFT) are defined in Section 2.3.3. First is the "Complex half-size" algorithm (2.19), which casts an RDFT_n into a $\mathrm{DFT}_{n/2}$ with a post-processing stage. When this algorithm is used, all of the DFT freedoms discussed above are relevant.

Then, three native RDFT algorithms are defined: a constant geometry algorithm (2.21), an iterative algorithm (2.22), and a recursive algorithm (2.20). As explained in Section 6.3, algorithm (2.21) is used when iterative reuse is requested, algorithm (2.22) is used when no iterative reuse is utilized, and the recursive algorithm (2.20) is used to perform the basic blocks.

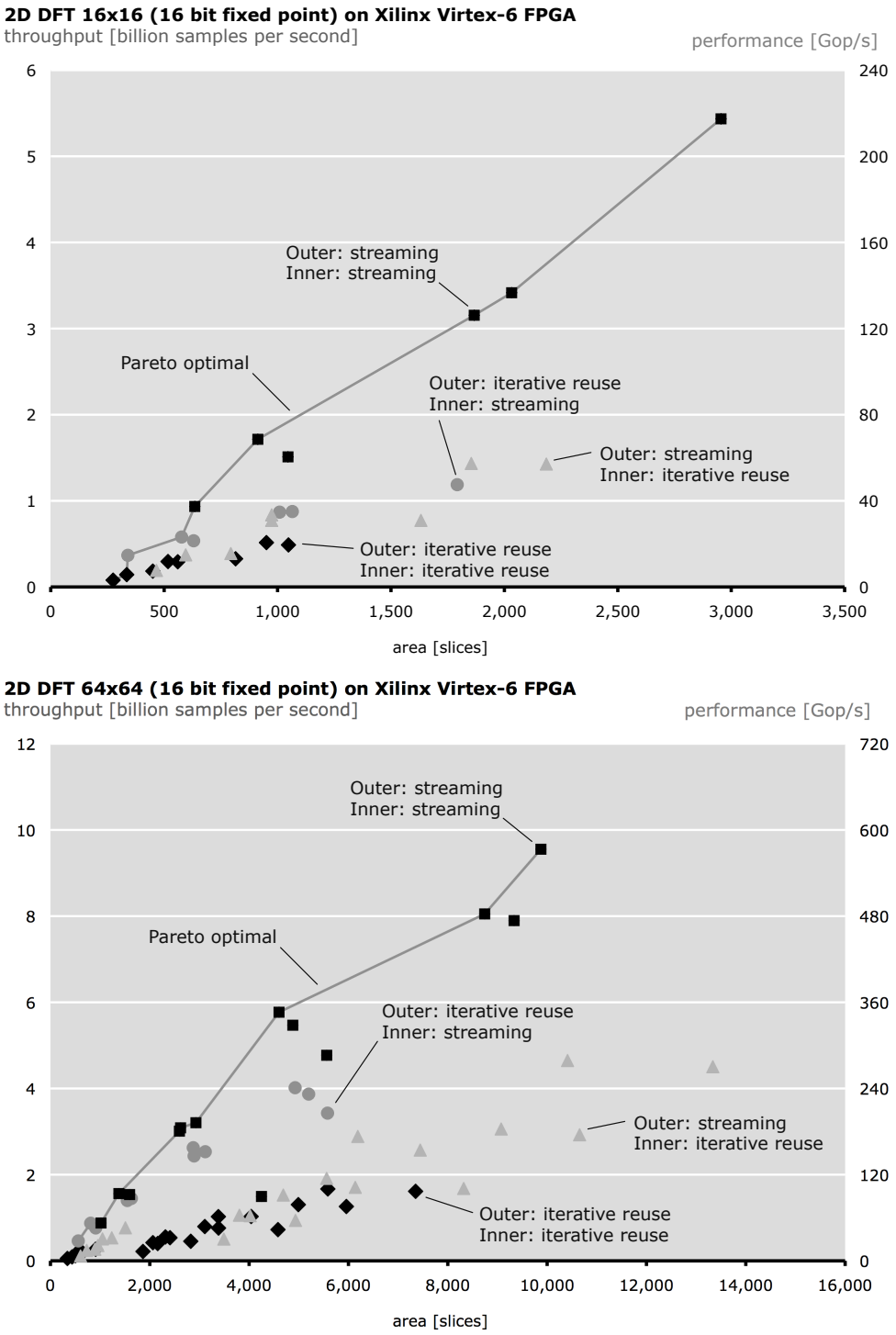


Figure 7.19: DFT -2D$_{16\times16}$ and DFT -2D$_{64\times64}$, fixed point, throughput versus area on FPGA.

Lastly, it is possible to compute an RDFT_n using an algorithm for DFT_n by simply utilizing the real inputs only. This technique has higher algorithmic cost than the others, but can serve as a useful baseline for comparison.

Figure 7.20 shows throughput versus area for Virtex-5 FPGA implementations of RDFT_{128} and RDFT_{2048}. In both plots, the different data markers represent the different algorithms listed above: circles for the complex half-size algorithm, diamonds for the native RDFT algorithms, and triangles for the complex DFT algorithm. Streaming widths are considered from $w = 4$ to $w = 32$. In this experiment, Spiral's memory options are set to allow no block RAMs to be utilized so that any differences in memory requirements between the algorithms are included within the final area count.

For both plots, the smallest point on the Pareto optimal front is the point corresponding to the complex DFT algorithm. From there, however, the complex algorithm becomes suboptimal, due to its increased arithmetic cost. Then, as the amount of area increases, the native RDFT algorithms appear on the Pareto front (diamonds), followed by the complex half-size RDFT algorithms (circles). The exact trend and ordering of these points depends on which radices are available based upon the problem's size.

7.5.3 Discrete Cosine Transform

This thesis also includes an evaluation of an algorithm (2.23) for the DCT of type 2. Other similar algorithms suitable for hardware generation with Spiral are given in [14, 16, 15].

Figure 7.21 shows throughput versus area for $\mathrm{DCT\text{-}2}_n$ for $n = 64, 256, 1024, 4096$ on the Xilinx Virtex-6 FPGA. Each line represents one value of n with streaming width starting from $w = 4$ (bottom-left) and increasing as the area/throughput increases. All points shown here are fully streaming (that is, no iterative reuse is utilized). Several designs are annotated with the number of BRAMs and DSP slices consumed by each design.

7.6 Evaluation Summary

This chapter provides an evaluation of FPGA and ASIC implementations of transforms generated by Spiral using the techniques presented in this thesis. In addition to the two platforms, the experiments

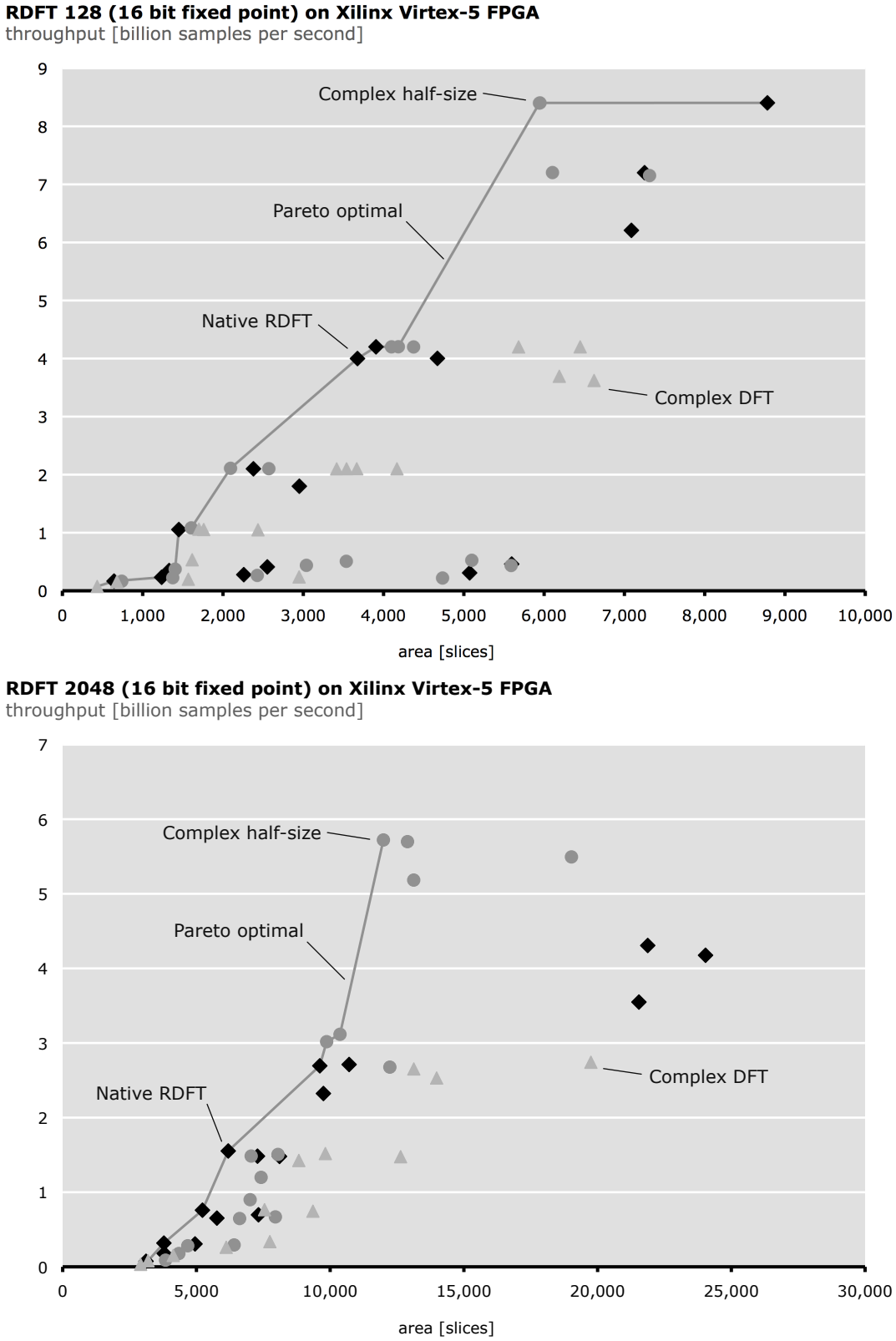


Figure 7.20: RDFT_{128} and RDFT_{2048}, fixed point, throughput versus area on FPGA.

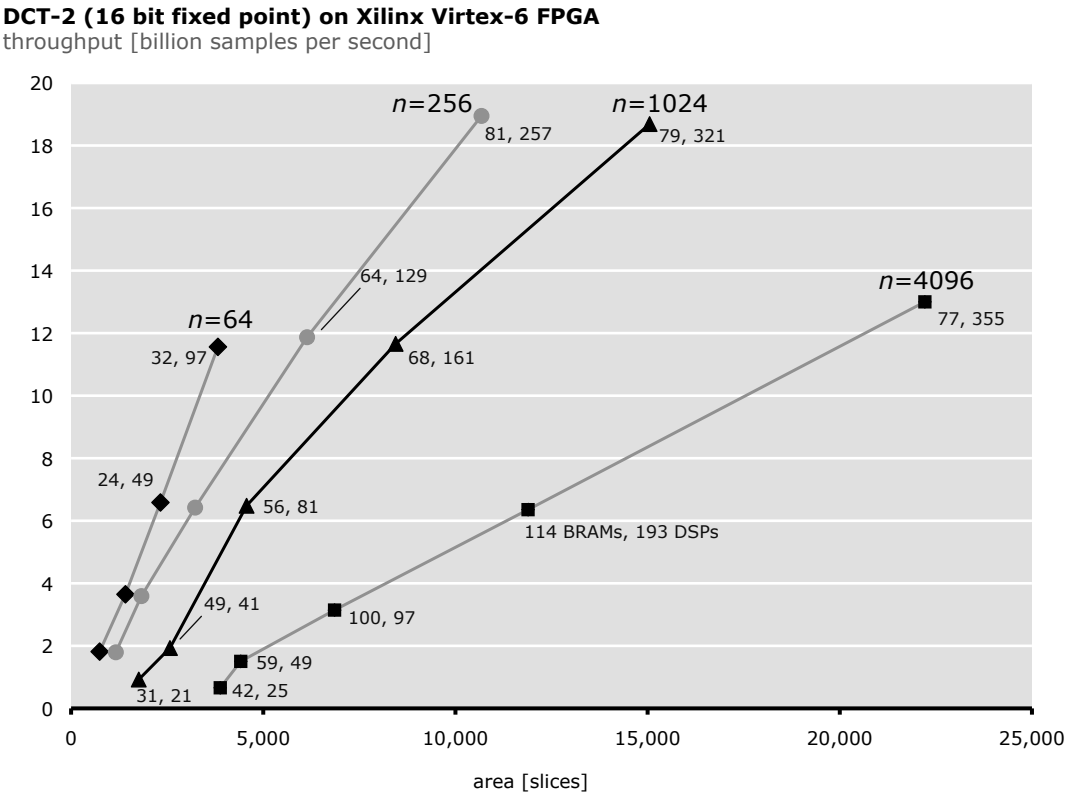


Figure 7.21: DCT-2_n, fixed point, throughput versus area on FPGA. Data labels indicate the number of BRAMs and DSP slices.

presented here include different data types (fixed point and floating point) and different transforms (DFT, DFT -2D, DCT-2, RDFT).

Given a problem, a datatype, and a platform, Spiral is able to provide a space of designs with varying performance and costs. This set of designs spans multiple algorithmic options and multiple ways of mapping from algorithm to datapath. The flexibility that Spiral allows in these design dimensions allow the user to easily explore a wide space of designs in order to determine the one that best meets his or her goals.

Next, Chapter 8 uses the proposed Spiral hardware generation framework to explore and implement designs used in real-time transceivers for orthogonal frequency-division multiplexing (OFDM) for optical interconnects. Then Chapter 9 discusses related work, and in particular compares the designs considered here with approaches found in the literature.

Chapter 8

Orthogonal Frequency-Division Multiplexing for Optical Networks

Orthogonal frequency-division multiplexing (OFDM) is a multi-carrier modulation technique that is widely used in communication applications. The most computationally expensive portion of OFDM is computing the DFT and its inverse. Recently, the optical communications community has considered OFDM as a way to reduce the cost of short distance optical interconnect (for example, in data centers) or improve the performance of long distance links.

This chapter describes an application study where the Spiral hardware generation system is used to provide implementations of the DFT and IDFT used in optical OFDM transmitters and receivers. This work was published in [36, 37, 38, 39] in collaboration with the listed coauthors.

8.1 FPGA Prototype

This section describes the construction of an FPGA-based prototype of an optical transmitter (detailed in [36, 37, 38]). Figure 8.1 shows a high-level view of this system, implemented in part on a Xilinx Virtex-4 FX 100 FPGA. First, input bits enter the system, and are mapped to complex symbols using quadrature phase-shift keying (QPSK), which maps two bits into one of four complex numbers. Then, a collection of symbols are fed as an input vector into an IDFT_{128} core. Then, the IDFT is computed, and the output data travel out of the FPGA into a 4-bit digital-to-analog converter, (running at 21.4 billion samples per second), and into a Mach-Zehnder optical modulator.

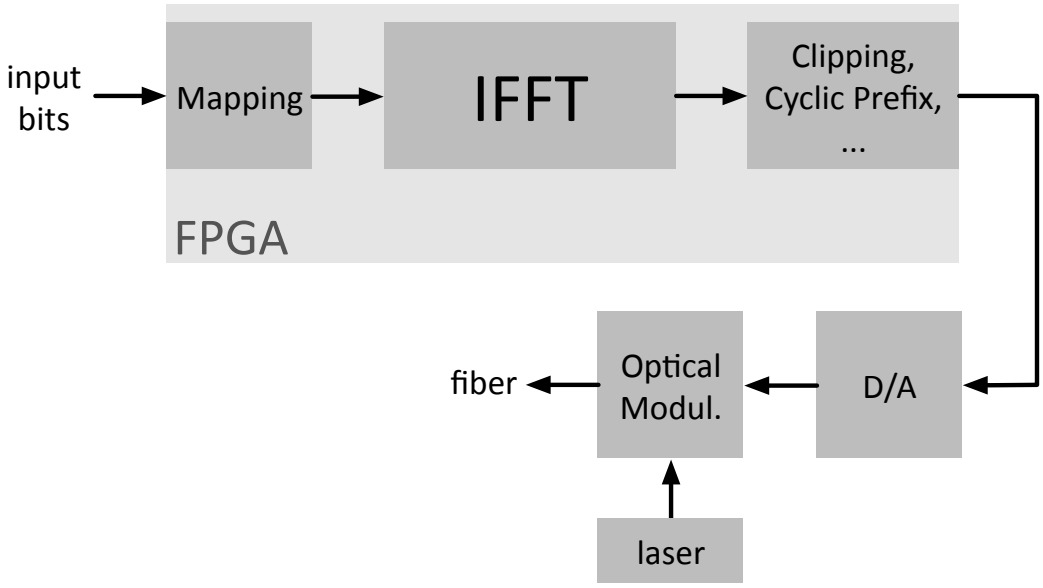


Figure 8.1: OFDM FPGA prototype transmitter.

Each output frame (128 samples) encodes 25 QPSK symbols, totaling 50 bits. Thus, the system transmits at a rate of $21.4 \times 50/128 = 8.35$ gigabits per second. The transmitter's bit error rate is less than 10^{-3}.

The remainder of this section describes how the Spiral hardware generation tool was used in the exploration and implementation of the IDFT_{128} needed within this transmitter. The IDFT core operates on fixed-point complex data and must appropriately scale the data to guarantee that overflow will not occur. The implementation must support a throughput of one transform per cycle (128 complex samples per cycle) at a frequency of 167.25 MHz. This gives an overall throughput requirement of 167.25 million transforms per second or 21.4 billion samples per second. Typically, hardware implementations of the IDFT utilize sequential reuse, as described in Chapter 3. However, in the proposed application, all 128 data elements must be processed in parallel (as opposed to a few at a time) in order to meet the throughput requirement. Therefore, a fully unrolled datapath (one that does not include any sequential reuse) is feasible. Such a datapath requires a large number of computational elements, but allows additional freedom in the algorithm selection. This design study includes evaluation of automatically generated IDFT cores derived from two families of IFFT algorithms: the Pease algorithm (2.13), which is regular and simple, and algorithms based on the Cooley-Tukey algorithm (2.12), which has a less regular structure, but when higher radices are used, lower computational cost.

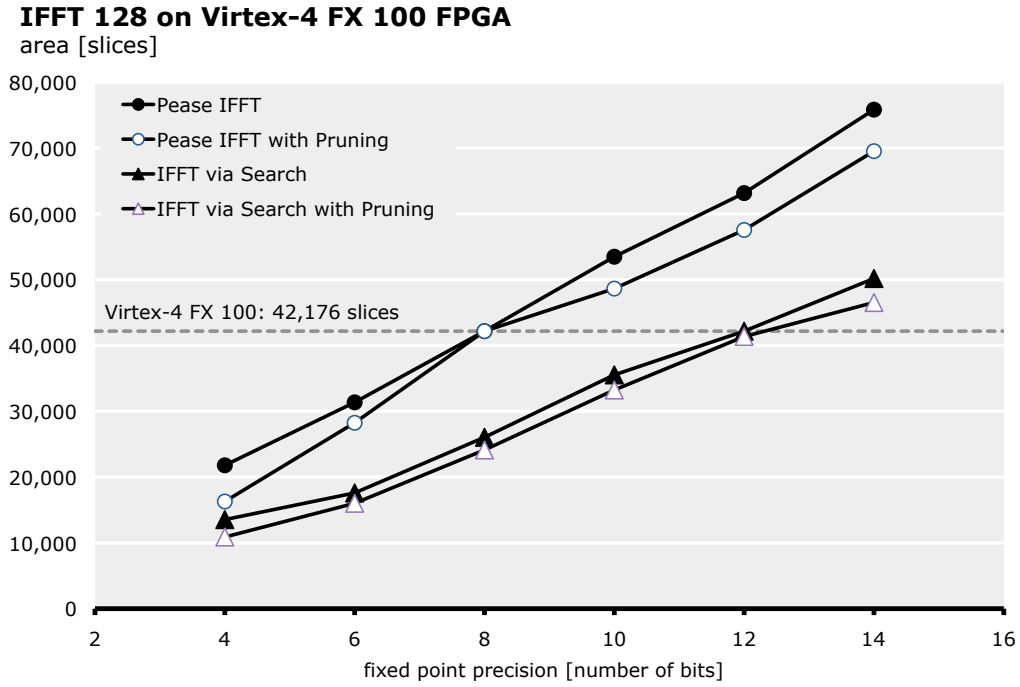


Figure 8.2: Area versus fixed-point precision of IDFT cores for OFDM on FPGA.

Pease IFFT Algorithm. The radix 2 Pease IFFT algorithm has a highly regular structure and is frequently used in hardware implementations of the IFFT. The algorithm (for n data points) consists of $\log_2(n)$ stages, each performing $n/2$ "butterflies" or basic blocks that perform one addition, one subtraction, and one complex multiplication. A generated fully unfolded implementation for $n = 128$ requires 2308 adders/subtractors and 908 multipliers (each operating on real values). The tool's automatic pipelining results in 36 stages. Figure 8.2 (black circles) plots the number of FPGA slices (reconfigurable logic elements) required against the fixed-point precision (in bits). Each design is synthesized and plate/routed using the Xilinx Integrated Software Environment (ISE) suite of FPGA tools. Using this algorithm, only the 4-, 6-, and 8-bit designs fit on the target FPGA.

Searching across a family of algorithms. Next, a wider set of IFFT algorithms are considered, based on (2.12), which is recursive. Using Spiral, many different breakdowns of these algorithms are generated and flattened into an unrolled structure. Then, search is employed to identify the algorithm with the lowest algorithmic cost. For this system, the best algorithm found is a mix of radix 8 and radix 16 IFFTs, resulting in 2192 adders/subtractors and 664 multipliers (a reduction of more than 25% of the multipliers used in the Pease implementation). Again, designs based on

	Fixed point precision (number of bits)					
	4	6	8	10	12	14
Mean RMS Error	6.6×10^{-2}	1.9×10^{-2}	4.9×10^{-3}	1.2×10^{-3}	3.0×10^{-4}	7.5×10^{-5}
Std. Dev. RMS Error	2.2×10^{-3}	5.4×10^{-4}	1.3×10^{-4}	3.6×10^{-5}	8.6×10^{-6}	2.2×10^{-6}
Mean SNR	8.9×10^{-2}	1.3	2.0×10^{1}	3.3×10^{2}	5.3×10^{3}	8.8×10^{4}
Std. Dev. SNR	6.2×10^{-3}	7.8×10^{-2}	1.1	1.9×10^{1}	3.0×10^{2}	5.1×10^{3}

Table 8.1: RMS error and SNR for IDFT_{128} designs.

this algorithm are generated for various values of fixed-point precision. Figure 8.2 plots the number of slices needed for each design (as black triangles). Designs based on this algorithm require on average 37% fewer slices than their Pease IFFT counterparts. Designs of up to 12 bits of precision are able to fit on the target FPGA, an improvement of four bits.

Numerical Accuracy. Table 8.1 presents an analysis of the root-mean-squared error (RMSE) and SNR for the IDFT generated designs. For each, 512 IDFT_{128} computations are performed on OFDM input symbols, and RMSE and SNR are calculated for each vector. For each design, Table 8.1 reports the mean and standard deviation of the 512 values for RMSE and SNR, calculated as follows, where S is the measured output vector and T is the expected output vector:

$$\mathrm{RMSE} = \sqrt{\sum_{n=0}^{N-1} \frac{|S_n - T_n|^2}{N}}$$

$$\mathrm{SNR} = \frac{\sum_{n=0}^{N-1} |T_n|^2}{\sum_{n=0}^{N-1} |S_n - T_n|^2}$$

A two bit increase in fixed-point precision causes the maximum magnitude of each data word to increase by a factor of 4. Table 8.1 illustrates that such an increase yields a decrease in RMSE of approximately a factor of 4 and an increase in SNR of approximately a factor of $4^2 = 16$.

Pruning. Depending on which subcarriers are utilized, a number of IFFT inputs will always be zero. Because this study considers fully unrolled designs, this makes it possible to prune out a number of unnecessary operations from the early stages of the IFFT, resulting in a modest decrease in required area. This is done by adding a wrapper around the IFFT core and allowing the synthesis tool to perform simplification. In Figure 8.2 (white data markers), this technique is evaluated on both IFFT algorithms discussed earlier, assuming that only a quarter of the IFFT inputs are nonzero. On average, this results in an 8.3% decrease in the number of slices.

8.2 ASIC Design Study

Further work in [39] presents a design study of a higher-throughput OFDM transceiver synthesized as an ASIC. This transceiver assumes signal converters (digital-to-analog and analog-to-digital converters) that process 28 billion samples per second, with 6 bit resolution on the D/A and 8 bit resolution on the A/D. This higher precision allows more channels of the transmitted signal to carry data, allowing a higher overall data rate. The system encodes 100 bits per 128 transmitted samples, giving an overall data rate of $28 \times 100/128 = 21.875$ gigabits per second. The paper [39] then performs simulation to determine the fixed point precision needed: 10 bits for the transmitter (which includes IDFT_{128}) and 14 bits for the receiver (which includes DFT_{128}.

Next, Spiral generates twenty different IDFT and DFT implementations (with 10 bit and 14 bit fixed point data types, respectively). All designs are based on the Iterative Cooley-Tukey FFT algorithm with varying radices (2.15). The designs considered process either 32, 64, or 128 samples per cycle, and thus must be clocked at 875, 437.5, or 218.75 MHz (respectively) in order to meet the throughput requirement of 28 Gsamples/s. To meet the OFDM performance target, each design must perform approximately 10^{12} fixed point operations per second. The designs that process more samples per cycle have a higher area but are clocked at a lower frequency and thus consume less power. This allows a tradeoff between power consumption and area.

Each design is synthesized using Synopsys Design Compiler Ultra targeting a commercial 65nm standard cell library. Transceivers are constructed from each generated IDFT and DFT implementation by integrating appropriate modules for QPSK mapping, de-mapping, and clipping/scaling. Then, each is synthesized and its frequency, area, and power are estimated. Figure 8.3 shows Design Compiler's reported area and power consumption for each design. Each point on the graph indicates the power (y-axis) and area (x-axis) of a single implementation of the transmitter or receiver. The solid lines indicate the Pareto-optimal set of designs (the set of designs that give the best tradeoff between power and area). The transmitter's Pareto-optimal set contains five designs, ranging from 0.72 mm^2 and 0.44 W to 0.95 mm^2 and 0.17 W. The receiver's Pareto-optimal set contains three designs ranging from 1.14 mm^2 and 0.73 W to 1.35 mm^2 and 0.24 W.

Typically, power is the most critical constraint within the system, so one would choose the lowest/rightmost design from the Pareto-optimal set. However, if the system area is constrained,

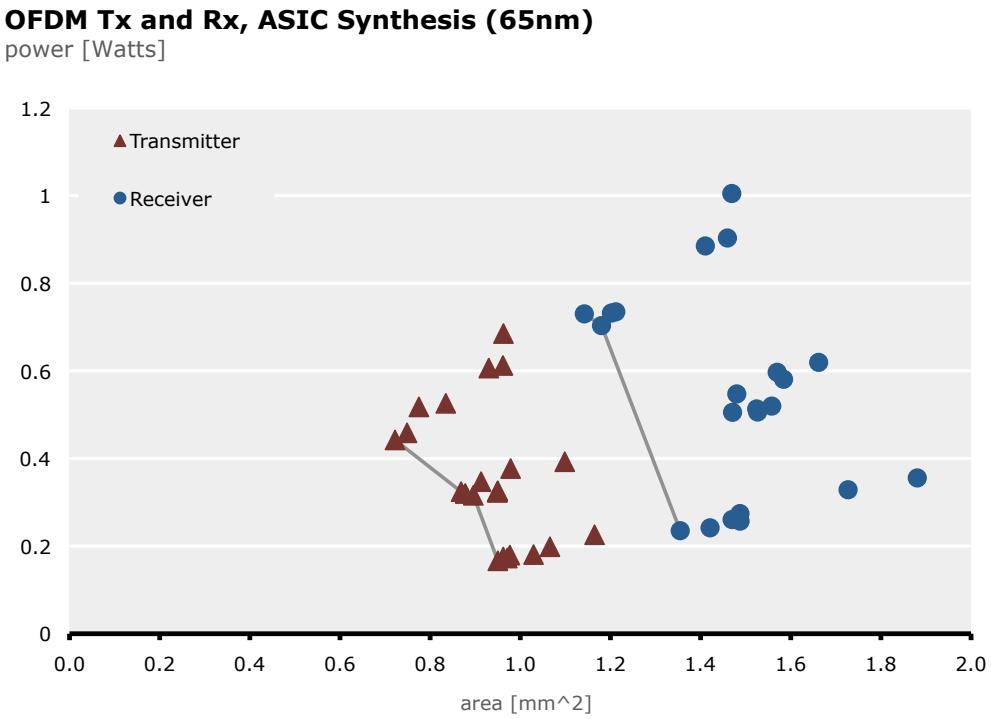


Figure 8.3: Power versus area for OFDM transmitters and receivers.

a smaller (yet higher power) design could be selected. The most power-efficient transmitter consumes 7.6 mW/Gb/s, while the most power-efficient receiver consumes 10.7 mW/Gb/s. In both the transmitter and receiver, the most power-efficient designs process 128 samples per cycle at 218.75 MHz, while the most area-efficient designs process 32 samples per cycle at 875 MHz. For these architectures, the mixed radix Cooley-Tukey FFT algorithm has lowest cost when radices 8 and 16 are used. Simpler algorithms, such as the commonly-seen radix 2 FFT have higher computational cost and correspond to the least efficient points seen on the graph.

Chapter 9

Related Work

9.1 Formula-based Hardware Representation

The tensor formula language used in this thesis has been previously used as a way to describe transform algorithms. For example [1, 7, 4] use this type of representation to discuss software implementations of the DFT.

Although the work presented in this thesis (published in part in [17]) was the first to extend the tensor formula language to support a general class of hardware implementations, variants of this mathematical language have been used in the process of constructing special purpose hardware for the DFT. For example, [40] uses the tensor product-based mathematical language in the process of designing a universal DFT processor that is scalable in the number of processing elements. More recently, [41] studies 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, [14, 15, 16] use the tensor formula representation as a way to algorithms suitable for pipelined hardware implementation.

9.2 Hardware Implementation of Transforms

This section gives an overview of work in the literature on hardware architectures for transforms. First, Sections 9.2.1, 9.2.2, and 9.2.3 discuss fully-folded, pipelined, and other flexible implementations of the DFT. Section 9.2.4 examines work that eliminates or reduces the number of multipliers

needed in a DFT implementation, and Section 9.2.5 discusses systolic array architectures for the DFT. Then, Sections 9.2.6 and 9.2.7 show examples of recent work on non-power-of-two sized DFTs and other transforms.

9.2.1 Fully-Folded DFT Processors

The simplest hardware implementations of the discrete Fourier transform are fully-folded architectures that use a small processing element sequentially over the course of the transform computation. The cost and performance of these fully-folded processors is closest to the designs in this thesis with full streaming and iterative reuse.

An early example can be found in [42], which consists of a complex butterfly and multiplier, two memories, and control logic. This idea is extended in [43], which implements a radix r FFT algorithm on a system with one DFT_r basic block and $2r$ memories.

The basic block in this type of system is improved in [44], which partitions the data into two memory banks. Then, two data can be read, pass through a butterfly unit, and be written on every cycle. The address and bank locations are computed using inexpensive bit arithmetic on the vector element number. In this way, this work can be viewed as a predecessor of the streaming permutation implementations in [20].

More recently, [45] partitions the memory in a system of this type into a small cache and a larger main memory. Then, the FFT factorization is chosen to make best use of the data in the local cache.

Architectures of this type have also been implemented on FPGA. For example, [46] uses two radix 4 processing elements and three banks of memory on a Xilinx Virtex-II FPGA.

9.2.2 Pipelined DFT Implementations

A common structure for implementing the DFT is the pipelined (or streaming) architecture. These designs are popular because they are relatively easy to implement and typically obtain much higher performance than the fully-folded processors discussed above. Additionally, designs of this type are easy to integrate into surrounding systems; typically they take in and emit one complex word per cycle. This type of architecture corresponds to designs with no iterative reuse and full streaming reuse (width $w = 1$). Although this is narrower than the smallest width considered in the designs

architecture	throughput (words / cyc.)	multipliers	mult. util.	adders	adder util.	apx. mem. size
R2SDF	1	$\log_2 n - 2$	50%	$2\log_2 n$	50%	n
R2MDC	1	$\log_2 n - 2$	50%	$2\log_2 n$	50%	$1.5n$
R2MDC-mod	2	$\log_2 n - 2$	100%	$2\log_2 n$	100%	$9.5n$
Spiral radix 2	w	$(w/2)(\log_2 n - 2)$	100%	$w\log_2 n$	100%	$6n$
R4SDF	1	$\log_4 n - 1$	75%	$8\log_4 n$	25%	n
R4MDC	1	$3(\log_4 n - 1)$	25%	$8\log_4 n$	25%	$2.5n$
R4MDC-mod	4	$3(\log_4 n - 1)$	100%	$8\log_4 n$	100%	$18.5n$
R4SDC	1	$\log_4 n - 1$	75%	$3\log_4 n$	75%	$2n$
R2^2SDF	1	$\log_4 n - 1$	75%	$4\log_4 n$	50%	n
Spiral radix 4	w	$(3w/4)(\log_4 n - 1)$	100%	$2w\log_4 n$	100%	$4.67n$

All multipliers and adders operate on complex data, and all implementations have bit/digit-reversed output.

Table 9.1: Summary of pipelined DFT architectures.

of this thesis, typically these designs sacrifice performance/cost scalability. That is, most can be thought of as architectures for the special case of $w = 1$.

There are two common types of pipelined DFT architectures: *single-path delay-feedback* (abbreviated SDF) and *single-path or multi-path delay-commutator* (abbreviated SDC and MDC), using the terminology from [47]. In each case, delay elements and switches are used to buffer and reorder the data streams as required by the algorithm. Table 9.1 summarizes the costs and throughput of different versions of these designs and compares them with the closest designs from the framework described in this thesis. In all cases, the designs in Table 9.1 take data in natural input order and produce it in radix 2 or 4 digit reversed order (that is, reordered by R^n_r).

Early examples of SDF architectures can be found in [48], which implements a simple radix-2 pipeline, and in [49] which utilizes a similar structure, but reduces the amount of memory required (see line labeled "R2SDF" in Table 9.1).

The MDC architecture is introduced in [50] has a higher memory cost with the same throughput for one DFT_n (line "R2MDC" in Table 9.1). However, this technique is able to compute two interleaved DFTs concurrently with double the throughput. In other words, it can compute $\mathrm{DFT}_n \otimes I_2$ with a throughput of two words per cycle and no additional overhead. In order to convert this into a traditional throughput-driven streaming design like those considered in this thesis, the expression can be rewritten

$$\mathrm{DFT}_n \otimes I_2 \rightarrow L^{2n}_n \cdot (I_2 \otimes \mathrm{DFT}_n) \cdot L^{2n}_2.$$

The streaming stride permutations L can then be computed using the bit-matrix-based streaming permutation method described in Chapter 5.2. This design, with added overhead for the L permutations, is given in line "R2MDC-mod" in Table 9.1). Although this allows speedup with no arithmetic cost, the memory cost is substantial. In the literature, this technique is not considered for this architecture; existing papers only consider this design for the interleaved case.

These techniques can also be extended to higher radices, as shown in the table's "R4SDF" and "R4MDC" lines. As shown, the radix 4 SDF architecture reduces the number of multipliers while increasing the number of adders. The radix 4 MDC architecture [51] has higher costs, but again can be used with interleaved data (computing $\mathrm{DFT}_n \otimes I_4$) which can be rewritten as in the radix 2 case above (indicated "R4MDC-mod" here).

Further work then expands on these designs. In [52], the commutator designs of [50, 51] are simplified, producing the radix 4 SDC design "R4SDC". This design reduces both arithmetic and memory requirements, but it does not have the property of the radix r MDC architecture that allows it to compute $\mathrm{DFT}_n \otimes I_r$ with no additional hardware cost.

He and Torkelson [47] summarize various types of FFT pipeline processor architectures and propose a "radix 2^2" single-path delay feedback (SDF) architecture that combines the processing element structure of earlier radix 2 designs with the higher-level implementation of a radix 4 algorithm. This algorithm and datapath are used by Xilinx in the LogiCore FFT IP that this thesis uses as a benchmark in Section 7.3. Essentially, [47] maps a radix 4 algorithm onto an SDF pipeline with basic blocks that are more similar to radix 2 basic blocks than previous techniques had used. In a straightforward implementation of an SDF radix 4 algorithm, there would be added complexity because the algorithm is not mapped well to a hardware structure that matches the overall pipeline. This distinction between radix 2 and radix 4 basic blocks (and their utilization) is only relevant in this manner type of limited-utilization pipeline. The drawbacks of going to higher radices described in [47] are not encountered when using the techniques proposed in this thesis.

Each of these architectures has throughput of 1, 2 or 4 complex words per cycle. In contrast, the fully-streaming designs from this thesis have a throughput of w per cycle ($w = 2^k$, $w \mid n$, and $w \geq r$).

To better illustrate the comparison between the different pipelined architectures, Table 9.2 normalizes the throughput, multipliers, adders, and approximate memory for the considered designs.

architecture	throughput	mults	adders	apx. memory
R2SDF	1	2	1.33	1
R2MDC	1	2	1.33	1.5
R2MDC-mod	2	2	1.33	9.5
Spiral radix 2	w	w	$2w/3$	6
R4SDF	1	1	2.67	1
R4MDC	1	3	2.67	2.5
R4MDC-mod	4	3	2.67	18.5
R4SDC	1	1	1	2
R2^2SDF	1	1	1.33	1
Spiral radix 4	w	$3w/4$	$2w/3$	4.67

Table 9.2: Summary of pipelined DFT architectures, normalized.

The designs built generated by Spiral compare well with the existing designs. Relative to the R2SDF architecture, the proposed radix 2 designs have $w\times$ higher throughput with $w/2\times$ the number of adders, $w/2\times$ the number of multipliers, and $6\times$ the number of memories. So, for $w > 4$, the proposed method yields superlinear speedup in all metrics. For $w \leq 4$, it yields superlinear speedup except with respect to memory required. Relative to R2MDC-mod, the proposed radix 2 design has $w/2\times$ the speedup with $w/2\times$ the multiplier and adder cost at 63% of the memory requirement.

For radix 4, Table 9.2 shows similar comparisons. Compared to the R2^2SDF architecture, the proposed design has a speedup of w with $3w/4\times$ multipliers, $w/2\times$ adders, and 4.67 times the memory. So, for $w > 4$, the proposed design yields superlinear speedup in all metrics. For $w = 4$, it yields superlinear speedup except with respect to memories.

Lastly, [41] presents a technique (related to the tensor formula language used in this thesis) to represent arbitrary radix SDC and SDF datapaths. These techniques could be incorporated into the generation system considered in this thesis.

9.2.3 Increasing Flexibility

In addition to the sequential processors and narrow pipelines discussed above, some work in the literature has allowed more freedom in the width and depth of a datapath.

For example, [53] includes discussion of a *parallel iterative* architecture that consists of one fully parallel stage of the FFT with full feedback. That is, full iterative reuse (depth $d = 1$) and no streaming reuse.

Another type of design is shown in [54], which maps the Pease FFT algorithm onto a rectangular

array processors consisting of one column of processing elements connected with an interconnection network that performs $L^n_{n/2}$. Other work [55, 56] are similar ([56] considers arbitrary radix r).

Wider versions of pipeline architectures are considered in [23], which presents a method to take a standard pipelined FFT implementation and increase its streaming width resulting in what they call a "partial column" implementation. With this approach, the difficult problem becomes performing streaming stride permutations with non-trivial streaming width. In this work, a technique with memories and interconnection networks is provided to perform $L^{2^n}_{2^s}$ with streaming width 2^k where $k \geq n - s$. As shown in [20], the connection networks used in [23]'s solution are more expensive than those used in the bit-matrix streaming permutation method (see Chapter 5.2).

Reference [40] implements the Dimensionless FFT algorithm, which allows multiple dimensions of DFT to be computed only by changing the constants stored in diagonal matrices. For example, one implementation can compute DFT_{16}, $(\mathrm{DFT}_2 \otimes \mathrm{DFT}_8)$, $(\mathrm{DFT}_4 \otimes \mathrm{DFT}_4)$, and so on. The architecture considered here is similar to [54], with a set of processors in parallel connected via interconnection network.

Reference [57] studies the energy-efficiency of FFT implementations on a Xilinx Virtex-II Pro FPGA, considering pipelined and iterative reuse designs. In [58], floating point implementations on FPGA are considered; all designs examined use streaming reuse of parameterizable width, and iterative reuse with depth 1, 2, or $\log_2 n$ (that is, no iterative reuse).

Other recent works [59] and [60] use high-level synthesis techniques to get architectural flexibility from one high-level description. These are discussed below in Chapter 9.3.

9.2.4 Multiplierless Implementations

Some prior work has focused on avoiding multiplication within hardware implementations of the DFT. For example, [61], [62], and [63] replace the complex multipliers in the SDF pipelined architectures and parallel pipeline architectures (as discussed above) with CORDIC complex rotators. This eliminates the need for complex multipliers and lookup tables. More recently, [64] revisited these techniques and reduced the cost of the rotator unit. Replacing multipliers with CORDIC rotators is orthogonal to the parameters studied in this thesis; such techniques could be applied as an alternate method for implementing the diagonal matrices that contain twiddle factors.

Other work has focused on reducing the number of multipliers in other ways. In [65], a nearly

multiplierless implementation of an FFT is proposed that reformulates the transform as a cyclic convolution. The result is a width $w = 1$ pipeline that requires only two multipliers but $3n/4 + 13$ adders to compute DFT_n. In practice, this technique is only feasible for small problem sizes. At $n = 1024$, [65] requires 2 multipliers and 781 adders (versus 8 multipliers and 20 adders for the simple radix 2 SDF datapath [49] discussed above).

A benefit of the technique in [65] is increased transform accuracy for the same fixed point input/output precision. For example, the authors show that a standard multiplier-based pipeline needs 28 bits to match the precision of their proposed 22 bit design. They further show that for small sizes ($n = 16, 32, 64$), their technique requires less resources than the R2SDF pipeline.

9.2.5 Systolic Arrays

The DFT has also been implemented using systolic arrays. Often, these implementations use $\mathrm{O}(n^2)$ algorithms that compute the DFT as a matrix-vector multiplication or other similar technique [66, 67, 68]. The advantage of a systolic array is that it is parallel and regular; the algorithms used in these papers map well to this type of system. However, this comes at the cost of higher arithmetic cost (number of operations per transform). This disparity grows greatly as the problem size increases.

Another approach [69] combines an $O(n \log n)$ FFT algorithm at the highest level, with a lower-level systolic array to compute the basic blocks. In the end, techniques like this end up structurally similar to other DFT architectures such as those discussed above.

9.2.6 Non-two-power sizes

Most previous work on hardware implementations of non-power-of-two sized DFTs has focused on producing a solution for a specific situation (a given problem size and performance requirement). For example, [70] and [71] consider the specific problem sizes and performance requirements of the Digital Radio Mondiale (DRM) radio broadcasting standard, while [72] presents an FPGA-based pipeline design of a 3,780 point inverse DFT used in a digital television standard.

A more flexible solution is given in [73], which uses a convolution-based approach to provide an algorithm and architecture for computing the DFT when the problem size is a prime number or a

product of primes.

9.2.7 Other Transforms

Similar implementations of other transforms also appear in the literature. For example, [74] maps an RDFT algorithm to a datapath similar to the SDF pipelines commonly seen for the DFT.

Another RDFT implementation appears in [75], where an algorithm similar to the complex half-size RDFT algorithm (2.19) is used to compute RDFT_n using $\mathrm{DFT}_{n/2}$ and post-processing. This paper then computes $\mathrm{DFT}_{n/2}$ using a fully-folded DFT processor similar to the ones discussed in Section 9.2.1.

Other papers explore these types of implementations of discrete cosine and sine transforms (DCTs and DSTs). For example, [15] derives constant geometry algorithms for DCTs and DSTs that are similar in structure to the Pease FFT (2.13). In [16], this algorithm is simplified for the special case of $\mathrm{DCT\text{-}2}$. Later, [14] extends these ideas further to enable pipelined architectures for DCTs and DSTs of type 2, 3, and 4. This is the source of the $\mathrm{DCT\text{-}2}$ algorithm presented in this thesis as (2.23) and evaluated in Section 7.5.3.

9.3 High-Level Synthesis

High-level synthesis (HLS) is an automated process that takes as input a behavioral description of a digital system and produces a register-transfer level implementation. By separating algorithm specification from implementation issues, HLS aims to ease the burden on designers by allowing them to ignore lower-level implementation issues. Further, by allowing multiple datapaths to be built from the same specification, HLS aims to lower the cost of design space exploration. A recent overview of HLS can be found in [76].

Many commercial products for high-level synthesis are available, based on different input languages. Many are based on subsets of C, C++, or SystemC, including Catapult C Synthesis (Mentor Graphics), Handel-C (Mentor Graphics), Synphony (Synopsys), Cadence C-to-Silicon Compiler, and Cynthesize (Forte Design Systems). Bluespec Compiler is based on Bluespec SystemVerilog, which extends a synthesizable subset of SystemVerilog to include atomic transactions. Lastly, other HLS tools are based on graphical environments, such as NI LabVIEW (National Instruments) and

Simulink HDL Coder (MathWorks).

Some recent papers have explored the benefits of using HLS to implement the DFT in FPGA or ASIC. First, [77] implements the radix 2^2 SDF structure from [47] on an FPGA using a C-based high-level synthesis language called Handel-C. This design is parameterizable in the number of bits to use for data and twiddle factors, the transform size (number of points), and scaling factors. However, it does not consider any architectural tradeoffs.

Reference [59] presents an ASIC transmitter for the 802.11a wireless standard written in Bluespec SystemVerilog. The most computationally expensive portion of this transmitter is the IDFT_{64}. This implementation is of the radix-2 Cooley-Tukey FFT, and it uses the capabilities of the HLS language to allow the design to be parameterizable in streaming width and iterative reuse depth. This paper also examines the tradeoff between area and power at a fixed performance target. Similar to the evaluation this thesis performs in Section 7.4.3, [59] obtains a tradeoff between area and power at a fixed performance target.

Similarly, [60] targets the DFT on an FPGA, expressing a radix 2 FFT algorithm as a double loop using National Instruments LabVIEW. Each loop can then be unrolled by the tool. The outer loop corresponds to the FFT's stages; unrolling corresponds to increasing the depth of the implementation. Similarly, the inner loop corresponds to the individual processing elements within the stage; unrolling this loop corresponds to increasing the streaming width.

These papers present a glimpse of the strength of high-level synthesis: by expressing an algorithm at a higher level of abstraction, the tool is able to aid in the low-level design of multiple architectures. However, the HLS tools are not well-equipped to handle the algorithmic-level exploration and optimizations made possible by Spiral's ability to manipulate algorithms and architectures within the same mathematical language. For example, although LabVIEW can manipulate [60]'s radix 2 FFT algorithm into designs with different width and depth, it is not able to convert to a radix 4 algorithm or a mixed radix implementation.

9.4 Streaming Permutations

The work on streaming permutations presented in Chapter 5 is unique in that the methods presented are the only ones that design RAM-based datapaths for a generalized family of streaming

permutations ([19] is applicable to any streaming permutation, and [20] can apply to an important subset). The only other fully-general streaming permutation method is [25], which builds streaming permutation structures using a register allocation method, resulting in a large number of individual registers connected with switches or multiplexers. Others consider just a small subset of streaming permutations: in [78, 79, 22], a method based on FIFOs and switches is used to build streaming stride permutations (L in this thesis) with two-power size. In [23], [80], and [81], subsets of streaming stride permutations are considered in the context of the DFT.

Chapter 10

Conclusions and Future Work

10.1 Overview

This thesis presented the Spiral hardware generation framework, an automated system based on a mathematical language that produces hardware cores for computing linear signal processing transforms. This system automatically implements hardware designs across a wide space of cost/performance tradeoffs, 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.

Chapter 2 presents relevant background on linear transforms and their algorithms. Transforms are represented as matrix-vector products, and algorithms are decompositions of transform matrices into products of structured sparse matrices. These decompositions can be written using a mathematical language based on linear algebra.

Chapter 3 shows how structure within transform algorithms can be mapped to two types of sequential datapaths. First, the tensor product $I_m \otimes A_n$ can be implemented as m parallel instances of A_n or as a system that employs *streaming reuse*, which would result in $< m$ parallel instances of A_n with the data vector streamed through the system over multiple cycles. This can be generalized to allow dependence on an iteration variable: $I_m \otimes_\ell A_\ell^n$. The cost of this implementation depends on the degree of similarity between the A_ℓ^n matrices.

The iterative matrix product $\prod_{\ell=0}^{m-1} A_\ell^n$ can be implemented as a cascade of A blocks (m deep) or as a system with *iterative reuse*, where the data vector feeds back and iterates over a shallower

system ($< m$ deep). Like streaming reuse, iterative reuse allows dependence on variable ℓ but this dependence can increase the system cost. Section 3.4 presents an example system that uses both streaming and iterative reuse, and explores the effect they have on cost and performance.

Chapter 3 defines variants of $\otimes$ and $\prod$ that allow a formula to explicitly specify the degree of streaming reuse and iterative reuse in a formula. The result is a new extended formula language that allows specification of an algorithm and a sequential reuse datapath on which to compute it. By formally capturing the desired freedoms at both algorithm and datapath levels, this language enables automatic compilation and exploration.

Chapter 4 describes the automated compilation and optimization process. First, the system takes as input a problem (for example, DFT_{1024}) and hardware directives that specify the amount and type of sequential reuse requested (as *tags*). Then, the system decomposes the base transform matrix using one or a combination of several transform algorithms, making choices based on the hardware directives supplied. Then, the algorithm is rewritten (based upon the supplied tags) into a formula in a hardware specification language that describes both algorithm and sequential datapath. Lastly, the formula is passed to a hardware generation stage that builds the corresponding datapath in synthesizable register transfer level Verilog.

Chapter 5 addresses an important challenge that arises when mapping transform algorithms to sequential datapaths: streaming permutations. A permutation is a fixed reordering of data elements, and a *streaming permutation* is one where the n data elements stream in and out of the system at a fixed rate of w words per cycle. In order to perform a permutation on streaming data it is necessary to use memory to reorder the position of data within the stream (reorder "in time"). In the simplest case where $w = 1$, this is easy: a single memory with one write port and one read port is sufficient. However, as w grows, the use of multiported memory quickly become infeasible. So, it is necessary to use multiple independent memories. However, now a new challenge arises: how to orchestrate the data's writes and reads in and out of memory while guaranteeing that no port conflicts will occur. Chapter 5 discusses solutions to this problem, including the "bit matrix" method [20] and the "general streaming permutation" method [19], which are used by Spiral in different situations.

Next, Chapter 6 revisits transform algorithms in the context of the sequential reuse hardware paradigm considered in this thesis. It discusses algorithms for the DFT (of two-power and non-two-power size), RDFT, two-dimensional DFT, and DCT. For each algorithm, it explains how sequential

reuse can be applied, and with what restrictions. When multiple algorithms can be applied, it discusses how the compiler's formula generation stage chooses between them.

Chapter 7 presents results from FPGA and ASIC synthesis of a space of designs generated using Spiral. For each, it illustrates the space of cost/performance tradeoffs obtainable, and compares with benchmarks where available. This chapter demonstrates designs across several transforms (DFT, 2DDFT, RDFT, DCT2), two platforms (65nm standard cell ASIC and FPGA), and two data types (fixed point and floating point). The FPGA results show performance versus area (in units of FPGA slices, BRAMs, and DSP slices). The ASIC results compare performance against area and power. They also explore the effect of frequency scaling on power, area, and performance. When targeting a fixed performance goal, Spiral is able to generate multiple designs that meet the goal while exhibiting a tradeoff between power and area: larger designs can be clocked at lower frequencies, reducing power.

Chapter 8 evaluates Spiral-generated designs used within orthogonal frequency-division multiplexing (OFDM) transceivers for optical networks. The freedoms in hardware and algorithm allowed by Spiral are exploited to determine the implementations best suited for this challenging application.

Lastly, Chapter 9 discusses related work in the literature, including work on hardware implementation of transforms, high-level synthesis, and permutations. This chapter includes comparisons between the most commonly-seen DFT architectures and those that are produced using the methods described in this thesis.

10.2 Directions for Future Work

This thesis concludes with a brief discussion of some possible future extensions of this work.

Interfacing with off-chip memory. The architectures considered in this thesis require that the entire data vector reside on-chip at once. That is, the memory elements in buffers and streaming permutations must store the entire data vector using on-chip memories. This means that the minimum area of a design produced by Spiral depends on the transform size, regardless of the performance requirement.

A promising direction for future work is to expand the generator to include designs that interact

with off-chip memory. This would require changes to both the hardware paradigm and the types of algorithms considered, but would use the existing infrastructure to generate the lower-level implementations. This type of *memory streaming* is considered in [82, 83], which extends Spiral to generate software for distributed memory architectures.

Register- and FSM-based basic blocks. The sequential hardware paradigm explored in this work uses dense matrices to specify computational basic blocks. That is, each processing element has an associated matrix and contains an arithmetic unit for each operation implied by the base matrix. For example, consider the matrix

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

In this thesis, this block would always be implemented with one adder and one subtractor; it would have a minimum streaming width of $w = 2$.

However, if lower throughput such as $w = 1$ is desired, another implementation could be produced that uses: one adder/subtractor, three data registers, a two-bit counter, and two control multiplexers. Data would be buffered in registers until it is consumed, and a small finite state machine (FSM) would provide control. Reducing the throughput requirement further (for example one word every two cycles) could lead to further simplifications.

In this way, a more flexible set of register- and FSM-based datapaths could serve as computational elements. Additionally, this type of technique can lead to less dependence on algorithmic regularity to obtain cost/performance tradeoffs. Further, this extension may be particularly useful if this work were extended outside the domain of linear transforms.

Extension to other kernels. The mathematical language used in this thesis represents the problem to be computed as a matrix-vector product. So, the problems that can be considered are limited to the domain of linear transforms. However, the types of algorithmic structure and dataflow that this thesis exploits appear in algorithms for other types of problems. Recent work [84] introduces Operator Language (OL), a language related to the one used in this thesis, which can represent non-linear problems while still mathematically capturing structure within the algorithm. [84] uses OL to generate software for Viterbi decoders, synthetic aperture radar image reconstruction, sorting

networks, matrix-matrix multiplication, and circular convolution. The system in this thesis may be able to be extended to support a subset of OL, allowing hardware generation for a wider space of problems.

Improvements to streaming permutation structures. This thesis uses two methods for designing datapaths to perform permutations on streaming data: the bit matrix method [20] and the general method [19]. As explained in Chapter 5, the bit matrix method generates switching networks and control/addressing logic specifically for the given permutation. The general streaming permutation method instead uses a generic all-to-all network, and uses lookup tables to store all of the necessary control and address values. So, it is trivial to use the general method to construct one structure to perform multiple permutations (in the event that a streaming permutation is iteratively reused) by extending the lookup tables to store precomputed constants for each of the permutations desired. However, it may be possible to extend the bit matrix method for the same purpose at lower cost.

Another possible extension is an improvement to the general streaming permutation method. When generating designs using this technique, a matrix representing data locations is decomposed into a sum of permutation matrices, as shown in (5.3):

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

where $\pi_w(P_n)$ is a $w \times w$ semi-magic square (the sum of each of its rows and columns is n/w), each of the Q_i are permutation matrices, and the β_i are integers > 0. Given a $\pi_w(P_n)$, the values of Q_i and β_i are not unique: the matrix can be decomposed in multiple ways.

Currently, Spiral performs this factorization by finding any permutation contained in $\pi_w(P_n)$ and subtracting it as many times as possible (that is, finding a β_i and Q_i). However, the overall cost of the implementation's connection network (T in Figure 5.3) is related to the set of permutations found here. In the worst case, a full generic connection network must be implemented to support all of the Q_i, consisting of $w' \log_w w' - w' + 1$ switches, where $w' = 2^{\lceil \log_2 w \rceil}$ and w is the streaming width. However, if $\pi_w(P_n)$ can be factorized using a subset of permutations, a simplified connection network with approximately half the cost could be used (such as those in [85, 86, 87]).

Lastly, both streaming permutation methods described here require double-buffering. This dou-

bles the memory requirement of each implementation so the system can begin buffering one input vector while the previous vector streams out. This overhead may be eliminated for some permutations if two schedules can be produced which overlap such that schedule 1 reads data from the exact locations that schedule 2 is about to write to. Both permutation methods considered in this thesis may be able to be extended to support this for a subset of permutations.

Hardware/software co-design. In addition to the hardware generation work described in this thesis, Spiral also generates software implementations of transforms for many platforms [4]. Thus, a natural direction for future work is in the co-design of hardware and software. Preliminary work on this topic has been presented in [88], which uses a library of FPGA implementations of the DFT generated by Spiral (using the techniques described in this thesis) as accelerators for an embedded processor. Given a set of accelerators, Spiral searches across a family of recursive algorithms in order to generate a library of implementations that use both hardware and software. Extensions to this work could include a larger search space of possible hardware implementations and a more sophisticated family of parameterized processors, all of which could be tuned based on desired system characteristics, such as off-chip memory bandwidth or power and area budgets.

Resource estimation. Spiral is currently able to provide resource counts from a hardware formula, giving the number and type of arithmetic and memory units. However, it could be extended to estimate area in much more detail. This could significantly speed up the exploration process by allowing designers to find the points that best fit their implementation goals more easily. Previous work [21] performed detailed area estimation for designs generated using the Pease FFT with radix 2, streaming reuse of a parameterizable width, and iterative reuse of depth 1 on an FPGA.

Bibliography

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

[2] Y. Benlachtar, R. Bouziane, R. I. Killey, C. R. Berger, P. Milder, R. Koutsoyannis, J. C. Hoe, M. Püschel, and M. Glick, "Optical OFDM for the data center," in *International Conference on Transparent Optical Networks*, 2010.

[3] B. J. C. Schmidt, A. J. Lowery, and J. Armstrong, "Experimental demonstrations of electronic dispersion compensation for long-haul transmission using direct-detection optical OFDM," *Journal of Lightwave Technology*, vol. 26, no. 1, pp. 196–203, 2008.

[4] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proc. of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[5] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 298–308, 2001.

[6] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[7] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures," *Circuits, Systems, and Signal Processing*, vol. 9, pp. 449–500, 1990.

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

[9] J. R. Johnson, "Pease FFT algorithm," Tech. Rep. DU-MCS-98-01, Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA, November 1998.

[10] L. I. Bluestein, "A linear filtering approach to computation of discrete Fourier transform," *IEEE Tranactions on Audio and Electroacoustics*, vol. 18, no. 4, pp. 451–455, 1970.

[11] G. E. Rivard, "Direct fast fourier transform of bivariate functions," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-25, no. 3, pp. 250–252, 1977.

[12] L. Auslander, J. R. Johnson, and R. W. Johnson, "Dimensionless fast Fourier transforms," Tech. Rep. DU-MCS-97-01, Department of Mathematics and Computer Science, Drexel University, 1997.

[13] Y. Voronenko and M. Püschel, "Algebraic signal processing theory: Cooley-Tukey type algorithms for real DFTs," *IEEE Transactions on Signal Processing*, vol. 57, no. 1, pp. 205–222, 2009.

[14] J. Nikara, J. H. Takala, and J. Astola, "Discrete cosine and sine transforms—regular algorithms and pipeline architectures," *Signal Processing*, pp. 230–249, January 2006.

[15] J. Astola and D. Akopian, "Architecture-oriented regular algorithms for discrete sine and cosine transforms," *IEEE Transactions on Signal Processing*, vol. 47, no. 4, pp. 1109–1124, 1999.

[16] J. H. Takala, D. A. Akopian, J. Astola, and J. P. P. Saarinen, "Constant geometry algorithm for discrete cosine transform," *IEEE Transactions on Signal Processing*, vol. 48, no. 6, pp. 1840–1843, 2000.

[17] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Formal datapath representation and manipulation for implementing DSP transforms," in *Proceedings of the 45th Annual ACM/IEEE Conference on Design Automation (DAC)*, pp. 385–390, 2008.

[18] F. Franchetti, Y. Voronenko, and M. Püschel, "Formal loop merging for signal transforms," in *Programming Languages Design and Implementation (PLDI)*, pp. 315–326, 2005.

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

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

[21] P. A. Milder, M. Ahmad, J. C. Hoe, and M. Püschel, "Fast and accurate resource estimation of automatically generated custom DFT IP cores," in *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 211–220, 2006.

[22] T. S. Järvinen, P. Salmela, H. Sorokin, and J. H. Takala, "Stride permutation networks for array processors," in *IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 51–71, 2004.

[23] S. F. Gorman and J. M. Wills, "Partial column FFT pipelines," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 42, no. 6, pp. 414–423, 1995.

[24] T. Láng, "Interconnections between processors and memory modules using the shuffle-exchange network," *IEEE Transactions on Computers*, vol. 25, no. 5, pp. 496–503, 1976.

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

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

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

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

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

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

[31] N. Eén and N. Sörensson, "Minisat." http://minisat.se.

[32] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," Tech. Rep. HPL-2009-85, Hewlett-Packard Laboratories, 2009.

[33] Xilinx, Inc., *Xilinx LogiCore IP Fast Fourier Transform v7.1*, April 2010.

[34] 4DSP, LLC, *4DSP Floating Point Fast Fourier Transform V2.6*, September 2007.

[35] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Hardware implementation of the discrete fourier transform with non-power-of-two problem size," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2010.

[36] Y. Benlachtar, P. M. Watts, R. Bouziane, P. A. Milder, R. Koutsoyannis, J. C. Hoe, M. Püschel, M. Glick, and R. I. Killey, "21.4 GS/s real-time DSP-based optical OFDM signal generation and transmission over 1600 km of uncompensated fibre," in *European Conference on Optical Communication (ECOC)*, 2009.

[37] Y. Benlachtar, P. M. Watts, R. Bouziane, P. A. Milder, D. Rangaraj, A. Cartolano, R. Koutsoyannis, J. C. Hoe, M. Püschel, M. Glick, and R. I. Killey, "Generation of optical OFDM signals using 21.4 GS/s real time digital signal processing," *Optics Express*, vol. 17, no. 20, pp. 17658–17668, 2009.

[38] Y. Benlachtar, P. M. Watts, R. Bouziane, P. A. Milder, R. Koutsoyannis, J. C. Hoe, M. Püschel, M. Glick, and R. I. Killey, "Real-time digital signal processing for the generation of optical orthogonal frequency division multiplexed signals," *IEEE Journal of Selected Topics in Quantum Electronics (to appear)*, 2010.

[39] R. Bouziane, P. A. Milder, R. Koutsoyannis, Y. Benlachtar, C. Berger, J. C. Hoe, M. Püschel, M. Glick, and R. I. Killey, "Design studies for an ASIC implementation of an optical OFDM transceiver," in *European Conference on Optical Communication (to appear)*, 2010.

[40] P. Kumhom, J. Johnson, and P. Nagvajara, "Design, optimization, and implementation of a universal FFT processor," in *Proc. 13th IEEE ASIC/SOC Conference*, pp. 182–186, 2000.

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

[42] R. R. Shively, "A digital processor to generate spectra in real time," *IEEE Transactions on Computers*, vol. C-17, no. 5, pp. 485–491, 1968.

[43] M. J. Corinthios, "The design of a class of fast Fourier transform computers," *IEEE Transactions on Computers*, vol. C-20, no. 6, pp. 617–623, 1971.

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

[45] B. Baas, "A low power, high-performance, 1024-point FFT processor," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 3, pp. 380–387, 1999.

[46] G. Szedo, V. Yang, and C. Dick, "High-performance FFT processing using reconfigurable logic," in *Proc. Asilomar Conference on Signals, Systems and Computers*, pp. 1353–1356, 2001.

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

[48] G. D. Bergland and H. W. Hale, "Digital real-time spectral analysis," *IEEE Transactions on Electronic Computers*, vol. EC-16, no. 2, pp. 180–185, 1967.

[49] H. L. Groginsky and G. A. Works, "A pipeline fast Fourier transform," *IEEE Transactions on Computers*, vol. C-19, no. 11, pp. 1015–1019, 1970.

[50] G. C. O'Leary, "Nonrecursive digital filtering using cascade fast Fourier transformers," *IEEE Transactions on Audio and Electroacoustics*, vol. 18, no. 2, pp. 177–183, 1970.

[51] B. Gold and T. Bially, "Paralellism in fast Fourier transform hardware," *IEEE Transactions on Audio and Electroacoustics*, vol. AU-21, no. 1, pp. 5–16, 1973.

[52] G. Bi and E. V. Jones, "A pipelined FFT processor for word-sequential data," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 12, pp. 1982–1985, 1989.

[53] G. D. Bergland, "Fast Fourier transform hardware implementations—an overview," *IEEE Transactions on Audio and Electroacoustics*, vol. AU-17, no. 2, 1969.

[54] G. Miel, "Constant geometry fast Fourier transforms on array processors," *IEEE Transactions on Computers*, vol. 42, pp. 371–375, 1993.

[55] E. L. Zapata and F. Argüello, "Application-specific architecture for fast trnasofmr based on the successive doubling method," *IEEE Transactions on Signal Processing*, vol. 41, no. 3, pp. 1476–1481, 1993.

[56] J. A. Hidalgo, J. López, F. Argüello, and E. L. Zapata, "Area-efficient architecture for fast Fourier transform," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 46, no. 2, pp. 187–193, 1999.

[57] S. Choi, R. Scrofano, V. K. Prasanna, and J.-W. Jang, "Energy-efficient signal processing using FPGAs," in *Proc. International Symposium on Field Programmable Gate Arrays*, pp. 225–234, 2003.

[58] K. S. Hemmert and K. D. Underwood, "An analysis of the double-precision floating-point FFT on FPGAs," in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 171–180, 2005.

[59] N. Dave, M. Pellauer, S. Gerding, and Arvind, "802.11a transmitter: a case study in microarchitectural exploration," in *MEMOCODE*, pp. 59–68, 2006.

[60] H. Kee, N. Petersen, J. Kornerup, and S. S. Bhattacharyya, "Systematic generation of FPGA-based FFT implementations," in *Proc. International Conference on Acoustics, Speech, and Signal Processing*, pp. 1413–1416, September 2008.

[61] A. M. Despain, "Fourier transform computers using CORDIC iterations," *IEEE Transactions on Computers*, vol. C-23, no. 10, pp. 993–1001, 1974.

[62] A. M. Despain, "Very fast Fourier transform algorithms hardware for implementation," *IEEE Transactions on Computers*, vol. C-28, no. 5, pp. 333–341, 1979.

[63] E. H. Wold and A. M. Despain, "Pipeline and parallel-pipeline FFT processors for VLSI implementations," *IEEE Transactions on Computers*, vol. C-33, no. 5, pp. 414–426, 1984.

[64] M. D. Macleod, "Multiplierless implementation of rotators and FFTs," *EURASIP Journal on Applied Signal Processing*, vol. 2005, no. 17, pp. 2903–2910, 2005.

[65] C.-D. Chien, C.-C. Lin, C.-H. Yang, and J.-I. Guo, "Design and realisation of a new hardware efficient IP core for the 1-d discrete Fourier transform," *IEE Proc. Circuits, Devices and Systems*, vol. 152, no. 3, pp. 247–258, 2005.

[66] G. E. Bridges, W. Pries, R. D. McLeod, M. Yunik, P. G. Gulak, and H. C. Card, "Dual systolic architectures for VLSI digital signal processing systems," *IEEE Transactions on Computers*, vol. C-35, no. 10, pp. 916–923, 1986.

[67] J. A. Beraldin, T. Aboulnasr, and W. Steenaart, "Efficient one-dimensional systolcic array realization of the discrete Fourier transform," *IEEE Transactions on Circuits and Systems*, vol. 36, no. 1, pp. 95–100, 1989.

[68] C.-M. Liu and C.-W. Jen, "On the design of VLSI arrays for discrete Fourier transform," in *IEE Circuits, Devices and Systems*, pp. 541–552, 1992.

[69] V. Boriakoff, "FFT computation with systolic arrays, a new architecture," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 41, no. 4, pp. 278–284, 1994.

[70] M. D. van de Burgwal, P. T. Wolkotte, and G. J. M. Smit, "Non-power-of-two FFTs: Exploring the flexibility of the Montium TP," *International Journal of Reconfigurable Computing*, vol. 2009, 2009.

[71] D.-S. Kim, S.-S. Lee, J.-Y. Song, K.-Y. Wang, and D.-J. Chung, "Design of a mixed prime factor FFT for portable digital radio mondiale receiver," *IEEE Trans. on Consumer Electronics*, vol. 54, no. 4, pp. 1590–1594, 2008.

[72] Z. Yang, Y. Hu, C. Pan, and L. Yang, "Design of a 3780-point IFFT processor for TDS-OFDM," *IEEE Transactions on Broadcasting*, vol. 48, no. 1, pp. 57–61, 2002.

[73] C. Cheng and K. K. Parhi, "Low-cost fast VLSI algorithm for discrete Fourier transform," *IEEE Transactions on Circuits and Systems I*, pp. 791–806, December 2007.

[74] Y. Wu, "New FFT structures based on the Bruun algorithm," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 38, no. 1, pp. 188–191, 1990.

[75] A. Wang and A. P. Chandrakasan, "Energy-aware architectures for a real-valued FFT implementation," in *Proc. International Symposium on Low Power Electronics and Design*, pp. 360–365, 2003.

[76] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Science + Business Media B. V., 2008.

[77] S. Sukhsawas and K. Benkrid, "A high-level implementation of a high performance pipeline FFT on Virtex-E FPGAs," in *Proc. IEEE Computer Society Symposium on VLSI*, pp. 229–232, 2004.

[78] J. Takala, T. Järvinen, P. Salmela, and D. Akopian, "Multi-port interconnection networks for radix-r algorithms," in *Proceedings of IEEE International Conference on Acoustics, Speech, Signal Processing*, pp. 1177–1180, 2001.

[79] J. H. Takala, T. S. Järvinen, and H. T. Sorokin, "Conflict-free parallel memory access scheme for FFT processors," in *Proceedings of the 2003 International Symposium on Circuits and Systems*, pp. IV–524–IV–527, 2003.

[80] T. Láng, "Interconnections between processors and memory modules using the shuffle-exchange network," *IEEE Transactions on Computers*, vol. 25, no. 5, pp. 496–503, 1976.

[81] Y. Ma, "An effective memory addressing scheme for FFT processors," *IEEE Transactions on Signal Processing*, vol. 47, no. 3, pp. 907–911, 1999.

[82] S. Chellappa, F. Franchetti, and M. Püschel, "Computer generation of fast Fourier transforms for the Cell Broadband Engine," in *International Conference on Supercomputing (ICS)*, pp. 26–35, 2009.

[83] S. Chellappa, *Computer Generation of Fourier Transform Libraries for Distributed Memory Architectures*. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University, 2010.

[84] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel, "Operator language: A program generation framework for fast kernels," in *IFIP Working Conference on Domain Specific Languages (DSL WC)*, vol. 5658 of *Lecture Notes in Computer Science*, pp. 385–410, Springer, 2009.

[85] D. H. Lawrie, "Access and alignment of data in an array processor.," *IEEE Transactions on Computers*, vol. 24, no. 12, pp. 1145–1155, 1975.

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

[87] D. S. Parker, "Notes on shuffle/exchange-type switching networks.," *IEEE Transactions on Computers*, vol. 29, no. 3, pp. 213–222, 1980.

[88] P. D'Alberto, P. A. Milder, A. Sandryhaila, F. Franchetti, J. C. Hoe, J. M. F. Moura, M. Püschel, and J. Johnson, "Generating FPGA accelerated DFT libraries," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 173–184, 2007.