

# Formal Datapath Representation and Manipulation for Implementing DSP Transforms

Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel  
Electrical and Computer Engineering Department  
Carnegie Mellon University  
Pittsburgh, PA, U.S.A.  
{pam,franzf,jhoe,pueschel}@ece.cmu.edu

## ABSTRACT

We present a domain-specific approach to representing datapaths for hardware implementations of linear signal transform algorithms. We extend the tensor structure for describing linear transform algorithms, adding the ability to explicitly characterize two important dimensions of datapath architecture. This representation allows both algorithm and datapath to be specified within a single formula and gives the designer the ability to easily consider a wide space of possible datapaths at a high level of abstraction.

We have constructed a formula manipulation system based on this representation and have written a compiler that can translate a formula into a hardware implementation. This enables an automatic “push button” compilation flow that produces a register transfer level hardware description from high-level datapath directives and an algorithm (written as a formula). In our experimental results, we demonstrate that this approach yields efficient designs over a large tradeoff space.

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

**General Terms:** Algorithms, Design

**Keywords:** linear transform, discrete Fourier transform, high-level synthesis, streaming

## 1. INTRODUCTION

Linear signal transforms such as the discrete Fourier transform are ubiquitous in digital signal processing (DSP) and scientific computing. Algorithms for computing these transforms are often highly structured and regular, which makes them well suited for hardware implementation. This regularity allows a wide space of potential datapath structures, each giving a different set of tradeoffs between performance and cost. It is very difficult for a designer to determine the structure that will yield the most efficient datapath for given cost or performance constraints.

**Contribution.** In this paper, we take a domain-specific mathematical representation for describing linear DSP al-

gorithms and extend it to include datapath concepts such as parallelism and explicit datapath reuse. The result is a mathematical language that we compile directly into hardware. Using this language, a designer specifies datapath options at the formula level. This leads to easier exploration of the design space by enabling algorithm restructuring through formula manipulation, which is performed automatically based on high-level directives.

We have constructed a “push button” synthesis system that takes as input an algorithm (written as a formula) and high-level datapath directives (indicating desired qualities of the resulting design); it outputs a design in register-transfer level (RTL) Verilog.

**Organization.** We begin by introducing the tensor (or Kronecker) representation for transform algorithms in Section 2. Then, Section 3 discusses the datapath constructs we consider, how we are able to include them within the existing mathematical representation, and the associated performance and cost metrics. Additionally, we give a high-level view of our synthesis system. In Section 4, we evaluate our generated designs. We present experiments that demonstrate: (a) that the cost/performance tradeoffs obtained are competitive with good hand-designed implementations, (b) that this system produces designs across a wide tradeoff space, and (c) that real benefits are obtained by considering a variety of datapath structures. Lastly, we discuss related work in Section 5 and conclude in Section 6.

## 2. BACKGROUND

**Transforms as matrices.** A linear transform may be viewed as a dense matrix; applying the transform is then a matrix-vector multiplication. For example, an  $n$  point transform characterized by matrix  $A$  is given by

$$y = A_n \cdot x,$$

where  $x$  and  $y$  are the  $n$  point input and output vectors (respectively), and  $A_n$  is an  $n \times n$  matrix. Direct evaluation of the matrix-vector product requires  $O(n^2)$  arithmetic operations.

**Algorithms as formulas.** Fast algorithms exist for many transforms that reduce the arithmetic cost to  $O(n \log n)$ . We view an algorithm as a decomposition of the dense matrix  $A_n$  into a product of *structured sparse matrices*. The tensor (or Kronecker) formulation has been shown to be a compact and efficient way to represent fast transform algorithms [5, 11]. Recently, others have shown that a framework based on this formulation can be used to generate optimized software

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2008, June 8–13, 2008, Anaheim, California, USA

Copyright 2008 ACM 978-1-60558-115-6/08/0006 ...\$5.00.

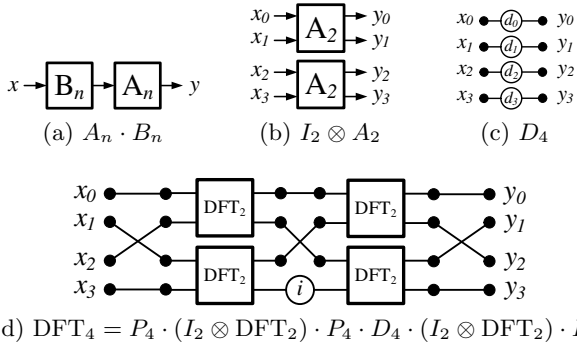


Figure 1: Examples of translating from formula to combinational datapath.

for today’s high-performance computer systems [10].

**Formula language.** This algorithmic representation is captured in a formal language that represents algorithms using *formulas*, with each term in the formula having a corresponding combinational datapath representation. In Backus-Naur form, the language is defined as follows (non-terminals are bold-faced):

$$\mathbf{matrix}_n ::= \mathbf{matrix}_n \cdots \mathbf{matrix}_n$$

$$\left| \prod_i \mathbf{matrix}_n \right.$$

$$\left| I_k \otimes \mathbf{matrix}_m \right. \quad \text{where } n = km$$

$$\left| \mathbf{base}_n \right.$$

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

A matrix formula can be decomposed into a product or iterative product of matrix formulas (lines 1 and 2, illustrated in Figure 1(a)). Matrix  $I_k$  is the  $k \times k$  identity matrix, and  $I_k \otimes \mathbf{matrix}_m$  is a tensor (or Kronecker) product, where  $k$  parallel instances of  $\mathbf{matrix}_m$  are applied to the data vector of size  $n = km$  (Figure 1(b)).

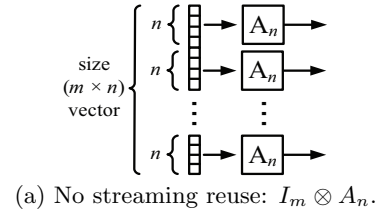
We use  $P_n$  to denote a permutation on  $n$  points and  $D_n$  to represent a diagonal matrix, which has non-zero values along the main diagonal only, causing each value of the input vector to be scaled by a constant (Figure 1(c)). Lastly, we use  $A_n$  to denote a generic dense  $n \times n$  matrix, which corresponds to a computational basic block.

This language is a subset of the signal processing language (SPL) used in Spiral, a program generator for software implementations of linear transforms [10]. An algorithm written in this language can be mapped directly to a combinational datapath (Figure 1(d)), but the resulting datapath is infeasibly large for all but the smallest problem sizes.

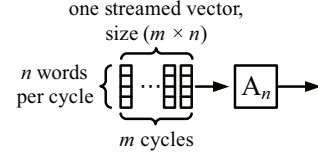
### 3. DATAPATH REPRESENTATION

The tensor language described above can represent a wide range of algorithms, but it does not have the capability of representing *sequential reuse* of datapath components, where one computational block is used many times while solving a single problem. Sequential reuse is necessary for efficient and reasonably sized hardware designs.

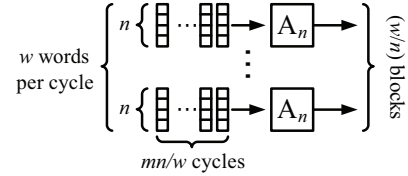
In this section, we describe extensions to our formula language to represent two types of sequential reuse that are relevant for hardware designs. We show how these extensions enable explicit datapath description at the formula level and discuss how formulas are automatically translated into register-transfer level datapath descriptions.



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



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



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

Figure 2: Examples of streaming reuse.

#### 3.1 Streaming Reuse

As we saw in Section 2, the tensor product  $I_m \otimes A_n$  indicates  $m$  data-parallel instantiations of the block  $A_n$  (Figure 2(a)). However, the same computation can be performed by other structures. For example, the tensor product can be interpreted as *reuse in time* (rather than parallelism in space). Then, we build a single instance of block  $A_n$  and reuse it over  $m$  consecutive cycles (Figure 2(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. We call this *streaming reuse* and represent it  $I_m \otimes^{\text{sr}} A_n$ . We define *streaming width* as the number of inputs (or outputs) that enter (or exit) a section of datapath during each cycle. Here, the streaming width is  $n$ .

We can nest the two interpretations of  $\otimes$  in order to build a partially parallel datapath that is reused over multiple cycles (Figure 2(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.

#### 3.2 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 (Figure 3(a)).

We can also perform the same computation by reusing the  $A_n$  block  $m$  times (Figure 3(b)). Now, the datapath must have a feedback mechanism to allow the data to cycle through the proper number of times. We call this *iterative reuse* and represent it by adding the letters “ir” to the product term:  $\prod_m^{\text{ir}} A_n$ . By nesting both kinds of product terms, we specify a number of blocks in sequence to be reused a number of times (Figure 3(c)). In general,  $\prod_m A_n$  can be restructured into  $\prod_m^{\text{ir}} (\prod_d A_n)$ , resulting in  $d$  cascaded instances of  $A_n$ , iterated over  $m/d$  times ( $m/d$  is an integer). We define *depth* as the number of stages built (here,  $d$ ).

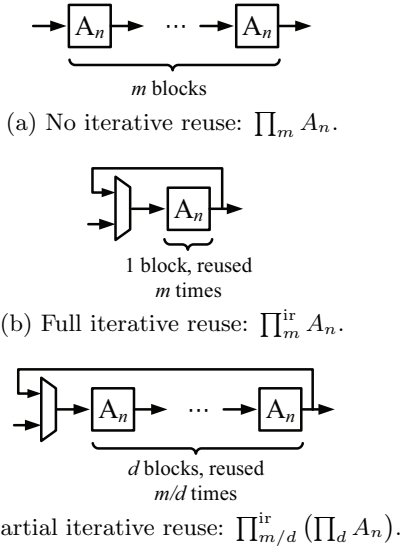


Figure 3: Examples of iterative reuse.

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 maintains sufficient distance from 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. We will see an example of this in the following section.

### 3.3 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. We can restructure this formula to have streaming and iterative reuse of parameterized amounts:  $\prod_{k/d}^{\text{ir}} (\prod_d (I_{nm/w} \otimes^{\text{sr}} (I_{w/n} \otimes A_n)))$ , 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 4. 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  $B_{mn}$  at a rate of  $w$  per cycle over  $mn/w$  cycles. The vector feeds back through  $B_{mn}$  a total of  $k/d$  times.

**Latency and throughput.** Given this combined reuse example, we can analyze the effect of parameters  $d$  and  $w$  on the datapath. In Table 1, we first describe several general rules for deriving the latency, throughput, and an approximate area cost of basic formula constructs.

Below, we present calculations that correspond to evaluating the general rules from Table 1 for the specific parameters of this combined reuse example (Figure 4). In these calculations, we 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:

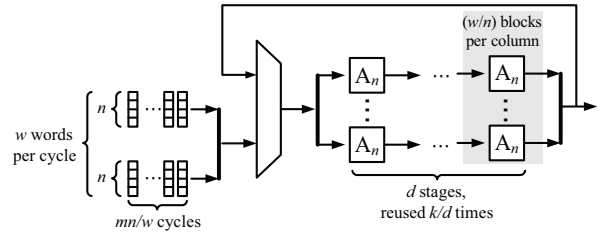


Figure 4: Combining iterative and streaming reuse:  $\prod_{k/d}^{\text{ir}} (\prod_d (I_{nm/w} \otimes^{\text{sr}} (I_{w/n} \otimes A_n)))$ .

- **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 we are utilizing 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.

From these equations, we see 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. For this reason, it is preferable to increase  $d$  instead of  $w$ . However, we also note that  $d$  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, we note that 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 we consider in this paper, 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.

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

**Table 1: Given a matrix formula  $F$ , formulas for latency  $L(F)$  (in cycles), throughput  $T(F)$  (in transforms per cycle) and approximate area cost  $C(F)$  (relative to the area cost of sub-modules).**

We can define a utilization ratio  $R$  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, we 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), we 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. Our compilation framework, discussed next, can utilize either strategy.

### 3.4 Compilation: From Math to RTL

We have built a compilation framework that takes an algorithm written as a formula, automatically manipulates it to describe a datapath, and translates the resulting design into register-transfer level (RTL) Verilog. A full explanation of this compilation framework is outside of the scope of this paper, but we present a high-level description here.

First, the algorithm is expanded into a formula in the language defined in Section 2. This formula corresponds to the computation that will be performed but does not specify the structure of the design that will perform it. Next, *datapath directives* are added that give desired characteristics of the final implementation (e.g., streaming width). A formula rewriting system then propagates these directives into the formula and restructures each term to match the desired characteristics. A *hardware formula* is produced that explicitly specifies the datapath architecture.

Lastly, the hardware formula is translated to an RTL netlist that has the desired reuse characteristics. Computational blocks are implemented according to the base matrices and streaming permutations are built with memory and interconnection networks.

## 4. EVALUATION

In this section, we evaluate designs produced using the proposed method. First, we explain our methodology. Then, we give several examples of transform algorithms that utilize  $I \otimes$  and  $\prod$  and demonstrate that streaming and iterative reuse lead to a wide tradeoff space in the resulting designs.

We compare our generated designs with existing benchmarks in order to demonstrate the quality of our cores. We also evaluate a transform with a wider tradeoff space and

examine how high-level design decisions affect the resulting design. Lastly, we discuss the generality of this approach and show other transform algorithms that utilize  $I \otimes$  and  $\prod$ , i.e., algorithms that can be implemented with streaming and iterative reuse.

### 4.1 Methodology

We have implemented the compilation framework that is described in Section 3.4 as a new backend to the Spiral formula generation framework [10]. Spiral is used to generate the starting formula for a given transform, and we have modified the tool to perform the formula manipulation associated with our extensions to the tensor formula language. Lastly, we have written a standalone compiler that translates a hardware formula into a register-transfer level (RTL) description. The tools are integrated, resulting in a completely automated flow from problem description to RTL Verilog.

In this section, we evaluate various designs produced with our framework. Here, we target the Xilinx Virtex-5 LX 330 FPGA and generate designs that use a 16 bit fixed point data type.<sup>1</sup> We use Xilinx ISE 9.1i to synthesize and place and route the designs. When memory is required, we use an on-chip block RAM (BRAM) if we can utilize 50% of its storage capacity. Otherwise, we use distributed RAM, i.e., memory distributed across the FPGA's logic cells. Although an FPGA platform provides a convenient target for evaluation, the designs we generate are not limited to FPGAs.

### 4.2 Quality of Generated Designs

In this section, we demonstrate that the designs produced by our framework are competitive with cores that are commercially available or found in recent literature. We choose the discrete Fourier transform (DFT) of size 1024; we evaluate our designs relative to cores from the commercially available Xilinx LogiCore FFT version 4.1 and the designs from our previous work [8]. The algorithm and architectures we considered in [8] are subsets of the space we consider here.

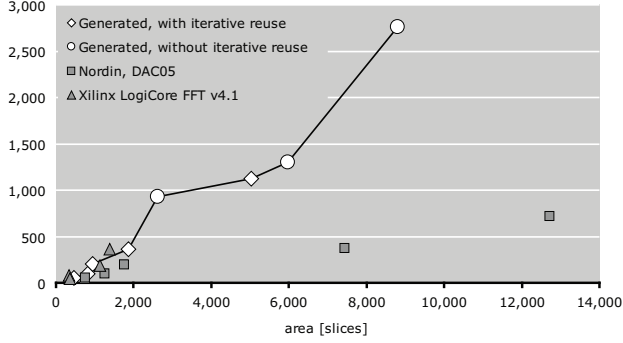
We generate cores based upon two algorithms: the Pease FFT [9] and an iterative version of the Cooley-Tukey FFT [3]. Although they are different algorithms, at the high level, both are of the form:

$$\text{DFT}_n = \left( \prod_{i=0}^{\log_r(n)-1} P_n(I_{n/r} \otimes \text{DFT}_r) D_n \right) P_n,$$

where  $r$  is a power of two. (See Section 2 for an explanation of the terms in this formula.) We generate a variety of designs using these algorithms with various depths, streaming

<sup>1</sup>Data type and bit width are parameters of our generation framework. Currently, our tool supports fixed point data types of any bitwidth and single precision floating point.

**DFT 1024 (16 bit fixed point) on Xilinx Virtex-5 FPGA**  
throughput [million samples per second]



**Figure 5: Throughput for varying implementations of DFT<sub>1024</sub>.**

widths, and radices (values of  $r$  in the formula above).

Here we consider steady state throughput (given in million samples per second) as our performance metric and area (in terms of FPGA slices) as our cost metric. Figure 5 shows throughput for varying implementations of DFT<sub>1024</sub>. From our data we plot only the *Pareto optimal* points, i.e., those that are not eclipsed by another design that is both smaller and faster.

From these results, we see that the cost and performance values of the LogiCore designs are similar to those of our smallest cores. Furthermore, we see that our larger cores provide a commensurate increase in performance for the extra resources they consume.

Our previous work [8] covers a small subset of the datapath and algorithmic options we consider in this paper. In Figure 5, we see that the added flexibility of our current method leads to significant improvements over [8]; the designs in our Pareto optimal set all provide higher performance at lower cost.

Similar trends are obtained if we choose a different value for  $n$  and/or measure latency instead of throughput.

### 4.3 Automatic Design Space Exploration

In this section, we consider an algorithm for the two-dimensional discrete Fourier transform (2D DFT). This algorithm utilizes  $I \otimes$  and two  $\prod$  terms, giving a very wide space of possible datapaths. This algorithm operates on  $n^2$  points and has the following form:

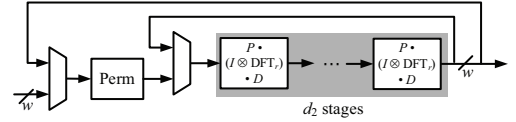
$$\text{DFT}_{n \times n} = \left( \prod_{k=0}^1 \left( \prod_{\ell=0}^{t-1} P_{n^2} (I_{n^2/r} \otimes \text{DFT}_r) D_{n^2} \right) P_{n^2} \right),$$

where  $t = \log_r(n)$ . This gives two iterative product terms, as seen above. Each may utilize iterative reuse, which we characterize with *depth* parameters  $d_1$  and  $d_2$  (see Section 3.2). This is illustrated in Figure 6.

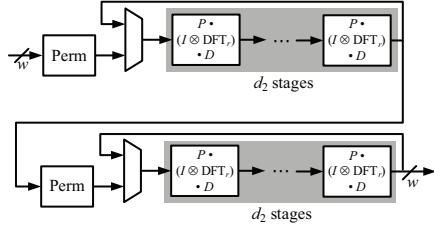
We define  $d_2$  to be the depth of the inner product;  $d_2$  can be between 1 and  $t$ , provided that  $t/d_2$  is an integer. Each internal block (a shaded region in Figure 6) consists of  $d_2$  stages of  $P \cdot (I \otimes \text{DFT}_r) \cdot D$ , streamed with  $w$  ports.

We define  $d_1$  to 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(a)). When  $d_1 = 2$ , the outer term is unrolled, giving two cascaded stages as seen in Figure 6(b).

**Exploration.** Now, we present results of a datapath exploration for 2D DFT<sub>64×64</sub>. We generate cores across all



(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: Illustrations of DFT<sub>n×n</sub> with outer product term parameterized by  $d_1$ , inner product term parameterized by  $d_2$ , and streaming width  $w$ .**

possible values of  $d_1$  and  $d_2$  with the streaming width  $w$  ranging between  $r$  and 16. Parameter  $r$ , the radix, is 2, 4, or 8. Summing these possibilities for  $d_1, d_2, w$ , and  $r$ , we have a total of 52 different architectures in this design space.

We generate each hardware core, synthesize it, and place and route it. In Figure 7, we show the throughput (in million samples per second) versus area (in FPGA slices) for all 52 data points, with different markers for each value of  $w$ . The black line passes through the Pareto optimal points.

In this data set, the smallest Pareto optimal point is the maximally folded design:  $w = 2, d_1 = 1, d_2 = 1$ . From there, we continue along the Pareto optimal set by first increasing the  $d$  parameters while keeping  $w = 2$  (white diamonds in Figure 7). Increasing  $w$  yields designs in the Pareto optimal set only after several values of  $d_1$  and  $d_2$  have been included.

This observation—that it is preferable to first increase  $d_1$  and  $d_2$  before increasing  $w$ —is supported by our theoretical understanding of streaming and iterative reuse as outlined in Section 3.3. However, it is not obvious which parameter combinations will yield designs in the Pareto optimal set, and it is difficult to determine the “crossover” points where one design parameter becomes more important than another. This highlights the importance of an automatic generation system; it would be exceedingly difficult to complete such a design exploration by hand.

### 4.4 Generality

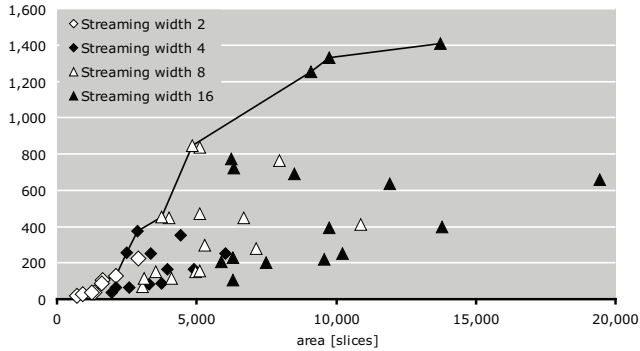
The datapath concepts considered in this paper (streaming reuse of  $I_m \otimes A_n$  and iterative reuse of  $\prod_k A_n$ ) apply to algorithms for transforms other than those already discussed. In this section, we present several problems that fit within these structures.

For example, the Walsh-Hadamard transform (WHT) can be computed with an algorithm of the form

$$\text{WHT}_{r,t} = \prod_{k=0}^{t-1} ((I_{r,t-1} \otimes \text{WHT}_r) P_{r,t}),$$

and the real discrete Fourier transform (RDFT) can be com-

**DFT 2D 64x64 (16 bit fixed point) on Xilinx Virtex-5 FPGA**  
throughput [million samples per second]



**Figure 7: Throughput versus area for 2D DFT<sub>64×64</sub>. The streaming width  $w$  is indicated by the data marker.**

puted using an algorithm of the form

$$\text{RDFT}_{4m} = P_{4m} \left( \prod_{k=0}^{\log_2(m)} (I_m \otimes_{\ell} \text{RDFT}_4(\ell, k)) P_{4m} \right) P_{4m}.$$

The WHT algorithm is completely expressible in the language we consider, and the RDFT requires only a small addition. Both algorithms contain iterative product  $\prod$  and tensor product  $I \otimes A$ , which means that iterative and streaming reuse can be applied to each. We have implemented both of these algorithms in our framework and have generated and evaluated datapaths for both.

Other fast linear transform algorithms can be written using  $\prod$  and  $I \otimes A$ , meaning that streaming and iterative reuse naturally apply. For example, [1] shows algorithms of this sort for discrete sine and cosine transforms (DST and DCT).

Lastly, we point out that streaming and iterative reuse can apply to other numerical problems outside of the domain of linear transforms. For example, Viterbi decoding is performed using a dataflow quite similar to the discrete Fourier transform, and many matrix-matrix multiplication algorithms exhibit parallelism which can be expressed by the tensor product. By extending our framework beyond linear transforms, we may be able to efficiently describe datapaths for these types of problems.

## 5. RELATED WORK

Although we do not know of any other instances of the tensor formula language being extended to support a general class of hardware implementations in this manner, it has been used in the process of designing special purpose hardware (e.g., an FFT processor in [7] and FFT cores in our previous work [8]). The important distinctions are that neither approach extends the formula language to describe datapath structure and that neither compiles from the formula to hardware; the formula is used to describe the algorithm only.

Many methods have been proposed to compile hardware from a software-like level of abstraction. This work differs from ours in the level of representation (typically C or Matlab code) and in scope.

Lastly, many special purpose FFT implementations have been proposed in the literature that have features that correspond to the datapath structures we are interested in. To name just a few, [6] is an example of a design with stream-

ing reuse, and cores with streaming and iterative reuse are developed in [2, 4, 8].

## 6. CONCLUSIONS

Linear DSP transforms and their algorithms are well understood and can be formally described in a compact manner with the tensor product formulation. In this work, we extended this framework to allow the representation of the datapath concepts of streaming and iterative reuse. This enables a domain-specific, formula-level view of hardware design and allows datapath manipulation to take place automatically at the mathematical level. We have implemented these ideas in an automatic design flow that manipulates a formula based upon high-level directives and produces a design in RTL Verilog. Lastly, we have presented results that demonstrate the breadth of these techniques and have established the quality of the generated designs.

## 7. ACKNOWLEDGMENTS

This work was supported by NSF through awards 0325687 and 0702386 and by DARPA through Department of Interior grant NBCH1050009 and ARO grant W911NF0710416.

## 8. REFERENCES

- [1] J. Astola and D. Akopian. Architecture-oriented regular algorithms for discrete sine and cosine transforms. *IEEE Transactions on Signal Processing*, 47(4):1109–1124, 1999.
- [2] D. Cohen. Simplified control of FFT hardware. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 24(6):577–579, 1976.
- [3] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90), 1965.
- [4] N. Dave, M. Pellauer, S. Gerding, and Arvind. 802.11a transmitter: a case study in microarchitectural exploration. In *MEMOCODE*, 2006.
- [5] J. Granata, M. Conner, and R. Tolimieri. The tensor product: a mathematical programming language for FFTs and other fast DSP operations. *Signal Processing Magazine, IEEE*, 9(1):40–48, 1992.
- [6] S. He and M. Torkelson. A new approach to pipeline FFT processor. In *Proc. International Parallel Processing Symposium*, 1996.
- [7] P. Kumhom, J. Johnson, and P. Nagvajara. Design, optimization, and implementation of a universal FFT processor. In *Proc. 13th IEEE ASIC/SOC Conference*, 2000.
- [8] G. Nordin, P. A. Milder, J. C. Hoe, and M. Püschel. Automatic generation of customized discrete Fourier transform IPs. In *Design Automation Conference (DAC)*, pages 471–474, 2005.
- [9] M. C. Pease. An adaptation of the fast Fourier transform for parallel processing. *Journal of the ACM*, 15(2), April 1968.
- [10] 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*, 93(2):232–275, 2005.
- [11] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM, 1992.