Automatic Generation of Customized Discrete Fourier Transform IPs

Grace Nordin, Peter A. Milder, James C. Hoe, and Markus Püschel
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA, U.S.A.

{yhn, pam, jhoe, pueschel}@ece.cmu.edu

ABSTRACT

This paper presents a parameterized soft core generator for the discrete Fourier transform (DFT). Reusable IPs of digital signal processing (DSP) kernels are important time-saving resources in DSP hardware development. Unfortunately, reusable IPs, however optimized, can introduce inefficiencies because they cannot fit the exact requirements of every application context. Given the well-understood and regular computation in DSP kernels, an automatic tool can generate high-quality ready-to-use IPs customized to user-specified cost/performance tradeoffs (beyond basic parameters such as input size and data format). The paper shows that the generated DFT cores can match closely the performance and cost of DFT cores from the Xilinx LogiCore library. Furthermore, the generator can yield DFT cores over a range of different performance/cost tradeoff points that are not available from the library.

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

General Terms: Algorithm, Design

Keywords: Discrete Fourier transform, IP, design generator, FPGA

1. INTRODUCTION

To improve productivity and time-to-market, hardware designers are increasingly using ready-to-use components from IP (intellectual property) libraries. However, a shortcoming of reusable IPs is that the end-designers cannot dictate application-specific customizations. An alternative is to use customized IPs generated by a parameterized design generator that can be tailored for application-specific tradeoffs in performance, size, power consumption, and numerical accuracy. We argue that parameterized IP generation is particularly well suited for DSP kernels such as transforms and filters because of their well-understood structure and regularity.

This paper focuses on the generation of discrete Fourier transform (DFT) cores, which are among the most important building block in DSP applications. However, the assumptions and the techniques in this paper are not restricted to the DFT and can be extended to other linear DSP transforms.

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

To support end-user customization, the DFT generator presented in this paper accepts a scalar parameter p that enables a designer to control the degree of concurrency in the microarchitecture of the resulting core. This control allows the designer to achieve a custom tradeoff between minimizing cost (e.g., area and power) and maximizing performance (e.g., latency and throughput). In addition to microarchitectural-level parameterization, our DFT generator (currently developed for Xilinx Virtex2-Pro FPGAs) also accepts a low-level parameter to reflect the designer's preference for spending memory versus logic resources in the resulting core.

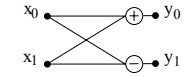
The output of the soft core generator is a synthesizable RTL-level Verilog description. Our evaluation shows that the generated DFT cores can match closely the performance and cost of DFT cores from the Xilinx LogiCore library. More importantly, we show that by varying p, our parameterized generator can yield DFT cores over a range of different performance/cost tradeoff points that are not available from the Xilinx library.

Paper outline. Section 2 provides a brief background on DFT and the Pease algorithm. Section 3 explains our framework for extracting logic structures from mathematically represented DSP algorithms. Section 4 next elaborates on the implementation and operation of the generated DFT cores. Section 5 reports a comparative evaluation of our generated DFT cores against the Xilinx LogiCore library. Section 6 discusses related work in the area of DFT IP design and optimization. Section 7 provides a summary and our conclusions.

2. BACKGROUND: DFT AND PEASE FFT

The DFT of a complex-valued input vector x of length n is the matrix-vector product $y = \mathrm{DFT}_n\, x$, where DFT_n is an n-by-n complex matrix. In particular,

$$\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad \text{dataflow:}$$



is the so-called *butterfly* matrix, which we denote by F_2. A direct evaluation of $\mathrm{DFT}_n\, x$ requires $O(n^2)$ arithmetic operations. Practical implementations, based on fast Fourier transform (FFT) algorithms, achieve $O(n \log(n))$ operations by factoring DFT_n into a series of multiplications by structured sparse matrices [5]. For a given n, there is a large number (exponential in n) of different FFT algorithms.

A special case of an FFT is the Pease algorithm given by the factorization

$$\mathrm{DFT}_{2^k} = R_{2^k} \left(\prod_{i=0}^{k-1} T_i (I_{2^{k-1}} \otimes F_2) L^{2^k}_{2^{k-1}} \right). \qquad (1)$$

Table 1: Matrix primitives and formulas M interpreted as multiplication $x \mapsto M \cdot x$.

$M = A \cdot B$	apply B, then A
$M = I_n \otimes A$	apply A, n times, in parallel
M is a permutation	permute element of x
M is diagonal	scale the elements of x by elements on the diagonal

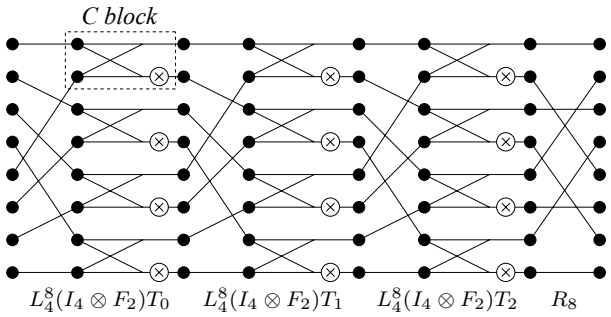


Figure 1: The dataflow graph of the Pease FFT for size $n = 8$. The formula terms appear in reverse order to match the left-to-right flow of the dataflow graph.

The equation above formulates the Pease FFT as a product of sparse matrices (that are to be multiplied to the input vector x starting with the rightmost factor).[1] The product $\prod$ in (1) indicates that the Pease algorithm for DFT_{2^k} comprises k iterations of multiplying by $T_i(I_{2^{k-1}} \otimes F_2)L^{2^k}_{2^{k-1}}$; these stages are identical apart from the scaling step T_i. For example,

$$\mathrm{DFT}_8 = R_8 \left(T_0(I_4 \otimes F_2)L_4^8\right) \left(T_1(I_4 \otimes F_2)L_4^8\right) \left(T_2(I_4 \otimes F_2)L_4^8\right). \quad (2)$$

3. ALGORITHM TO HARDWARE

The matrix formula framework provides an easy visualization of an algorithm's dataflow for datapath synthesis. For example, the dataflow for $M = A \cdot B$ is the dataflow of B followed by the dataflow of A. Table 1 gives the rules for mapping matrix formulas to dataflow for the other constructs relevant to this paper.

For example, using the interpretations in Table 1, the Pease FFT in (2) translates into the dataflow graph in Figure 1. In accordance with the formula, the DFT_8 dataflow comprises three successive stages of "$T(I_4 \otimes F_2)L_4^8$," followed by a R_8 bit-reversal permutation. In general, the Pease FFT computes the DFT of 2-power size n in $\log_2(n)$ stages of "$T(I_{n/2} \otimes F_2)L^n_{n/2}$." Let C, shown in Figure 1, be the computation block comprising a butterfly F_2 followed by a twiddle multiplier[2]. Then we can view the Pease dataflow as a $\log_2(n)$-by-$n/2$ grid of C blocks, where the columns are separated by the $L^n_{n/2}$ stride permutations. The regularity and structure in the dataflow is directly reflected by the product and tensor product in the Pease FFT formula (1) and is conducive to the concurrency parameterization employed by our core generator. In contrast, the more commonly used Cooley-Tukey algorithm [5] has identical cost in terms of the number of C blocks but does not exhibit a similar degree of regularity.

The dataflow graph for the Pease FFT can be directly mapped to a combinational implementation, but except for very small n the cost would be prohibitive. A practical DFT implementation requires a sequential implementation where the logic resources, e.g., C, are reused iteratively. The grid-like organization of the Pease DFT dataflow graph enables resources to be reused iteratively in two dimensions: *horizontally* and *vertically*, as we explain next.

[1] In this notation, the upper-case letters represent well-known structured sparse matrices. R_{2^k} denotes the *bit-reversal* permutation matrix that reorders a vector by bit-reversing the indices. T_i is a diagonal matrix that multiplies (scales) the elements of the input vector by complex constants (known as the twiddle factors). I_n is an n-by-n identity matrix, and, more importantly, the *Kronecker* (or tensor) product $I_n \otimes F_2$ is a $2n$-by-$2n$ block-diagonal matrix that is zero everywhere except for F_2's along the diagonal. Lastly, L^n_m is the *stride* permutation matrix that reorders a vector according to $L^n_m : i \mapsto mi \mod n-1\ (0 \le i < n-1),\ n-1 \mapsto n-1$. Most pertinent to this paper, $L^n_{n/2}$ is the "perfect-shuffle" permutation (e.g., L_4^8 reorders [0,1,2,3,4,5,6,7] as [0,4,1,5,2,6,3,7]).

[2] Only a single twiddle multiplier is needed in each C block because one of the two twiddle factors is always 1.

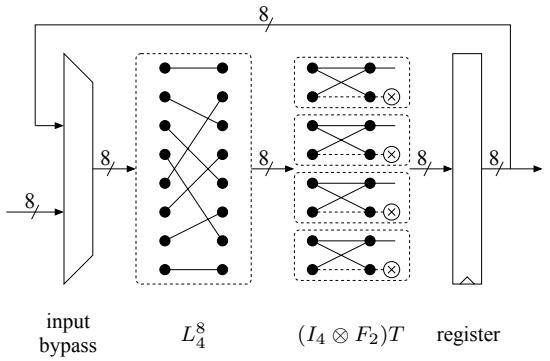


Figure 2: A fully h-folded Pease FFT, size $n = 8$ ($p = n/2$ many C blocks).

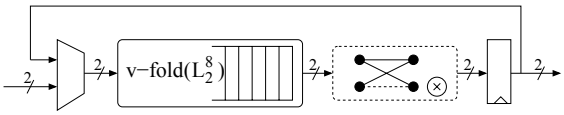


Figure 3: A fully h- and v-folded Pease FFT, size $n = 8$ ($p = 1$ C block).

3.1 Folding the datapath horizontally

The Pease dataflow graph can be trivially folded horizontally into a single column of $n/2$ C blocks. Figure 2 shows the resulting implementation for $n = 8$ including the necessary multiplexers and registers. A vector iterates over the feedback datapath $\log_2(n)$ times to compute the DFT_n. In each iteration, a control signal selects, from a table, the appropriate set of twiddle factors as operands to the $n/2$ twiddle multipliers. In this fully horizontally folded configuration, the resource requirements are reduced by approximately a factor of $\log_2(n)$. The throughput and latency of the folded implementation is practically unchanged from the combinational implementation. There is no advantage in considering a partial degree of horizontal folding; therefore we always use "h-folded" to mean "fully h-folded" in this paper.

3.2 Folding the datapath vertically

Starting from an h-folded datapath, the $n/2$ C blocks within the column can be folded vertically to differing degrees to achieve different levels of concurrency and therefore different performance and cost tradeoffs. In the extreme, the $n/2$ C blocks are fully v-folded onto a single C block, as shown in Figure 3, again for size $n = 8$. This C block is now reused iteratively $(n \log_2(n))/2$ times to compute the complete DFT_n.

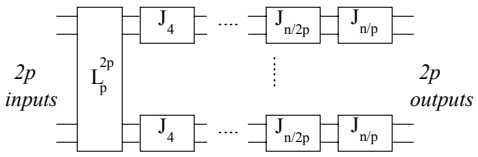


Figure 4: A v-folded $L^n_{n/2}$ permutation with $2p$ ports.

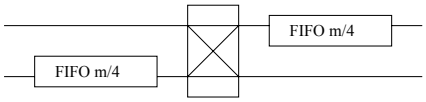


Figure 5: The J_m block in Figure 4.

4. IMPLEMENTATION DETAILS

In our DFT generator, a user-specified parameter p controls the degree of v-folding; $p = n/2$ signifies the original h-folded configuration with $n/2$ C blocks (Figure 2); $p = 1$ signifies the fully v-folded configuration with just one C block (Figure 3). Relative to a baseline configuration with h-folding only, v-folding by a degree p reduces resource requirements by roughly a factor of $n/2p$; the performance, in terms of both latency and throughput, degrades by the same factor of $n/2p$ commensurately. In our DFT generator, p is the main microarchitectural parameter that controls the tradeoff between performance and cost.

4.1 Folding the $L^n_{n/2}$ permutation

To vertically fold the Pease datapath, the main problem is in folding the $L^n_{n/2}$ permutation. Without v-folding, $L^n_{n/2}$ is a simple permutation in space implemented by wires. In a v-folded configuration with $p < n/2$, the input and the intermediate vectors in the datapath are streamed in segments of $2p$ elements per cycle. Thus, a v-folded $L^n_{n/2}$ permutation becomes a sequential logic block that buffers and reorders the vector elements in space and time.

Takala et al. [4] describe an efficient v-folded implementation of general stride permutations. Specifically, $L^n_{n/2}$ can be decomposed into the structure shown in Figure 4 for any choice of $0 < p < n$. In the figure, the L^{2p}_p block remains a simple wiring. The occurring 2-input, 2-output J_m blocks are implemented sequentially as shown in Figure 5. Each J_m block requires two $m/4$-entry synchronous FIFOs and a programmable switch that allows the two data values either to pass-through for $m/4$ cycles or to criss-cross for $m/4$ cycles. The switching state alternates every $m/4$ cycles repeatedly to permute a continuous stream of input vectors. An $L^n_{n/2}$ permutation v-folded to have $2p$ inputs and $2p$ outputs has a pipeline delay of $n/2p - 1$.

4.2 Basic operation and latency

The DFT computation of a v-folded Pease FFT with parameter p begins with loading the input vector, $2p$ elements per cycle, into the datapath through the input-bypass multiplexer (refer back to Figures 2 and 3 for a visual reference). After $n/2p$ cycles, the vector is fully loaded and buffered in the feedback register (for $p = n/2$) or in the FIFOs of the folded $L^n_{n/2}$ (for $p < n/2$). Once loaded, the vector cycles through the pipeline $\log_2(n)$ times. In the last iteration, the transformed vector is read out instead of being fed back into the datapath. When processing a stream of input vectors, the last iteration of one vector is overlapped with loading of the next vector.

If the C blocks are not pipelined, a vector of length n passes through p concurrent C blocks in $n/2p$ cycles. This matches the latency of the v-folded $L^n_{n/2}$ permutation. Therefore, a new iteration can start every $n/2p$ cycles. In the actual implementation, the C block (with an internal critical path comprising one 16-bit fixed-point multiplication and two 16-bit fixed-point additions) is pipelined into three stages to increase the clock-rate; therefore the actual iteration time is $(\frac{n}{2p} + 3)$ cycles. Thus, the latency of our Pease DFT core is $\log_2(n)(\frac{n}{2p} + 3)t$ where t is the cycle time (which varies with n and p). The steady-state throughput is $1/\text{latency}$.

The generated DFT core does not implement the bit-reversal permutation at the end of $\log_2(n)$ iterations because in most applications, this bit-reversal permutation is absorbed with another permutation in a subsequent computation. Thus, the generated DFT core has a "natural-in, bit-reversed-out" data ordering interface.[3] This is a common interface option found in many DFT cores.

5. EVALUATION

We compare our generated DFT cores to their counterparts from the Xilinx LogiCore library [6]. The Xilinx DFT cores are based on an in-place radix-4 (and optionally radix-2) Cooley-Tukey algorithms. The radix-4 algorithm is scheduled to serially reuse a single pipelined DFT_4 block $n\log_2(n)/8$ times. Input and intermediate vectors are stored in a multi-ported memory array. Although the Xilinx LogiCore library does not provide parameterized control over cost/performance, it does offer three distinct choices of DFT microarchitectures, with tradeoffs between performance and cost. The library also offers a selection of data ordering, data format and rounding modes.

We select Xilinx radix-4 DFT cores with a scaled fixed-point (16-bit) data format, burst I/O interface, and natural-in bit-reverse-out data ordering. This configuration matches the interface specification of our generated DFT cores. All results in this study are based on synthesis for Xilinx Virtex2-Pro XC2VP100 FPGAs using the Xilinx ISE version 6.1. The slice and BRAM utilizations are extracted after synthesis and mapping.[4] The cycle times used to compute the latency are extracted after place-and-route.

Figures 6 and 7 report the results for DFT_{64} and DFT_{1024}, respectively. For each DFT size, the graph reports, as a function of p, the slice utilization, the BRAM utilization, and the relative speedup[5]. Our generator allows the user to specify whether storage structures (twiddle tables and FIFOs of varying sizes) are implemented using logic slices or BRAMs. Results corresponding to three exemplary resource settings are given: 1) minimize the use of slices; 2) minimize the use of BRAMs; and 3) store twiddle tables and large FIFOs in BRAMs.[6] The dashed horizontal line in each graphs reflects the Xilinx reference value.

For small values of p (e.g., 2 or 4), our soft cores with balanced storage assignments (option 3) closely match the absolute cost and performance of the Xilinx DFT cores. For DFT_{64} in particular, at $p = 2$, our generated core matches the performance of the Xilinx core while using 38% less slices and 75% less BRAMs. More

[3]The generator can also produce a "bit-reversed-in, natural-out" variation.

[4]A slice is the basic logic building block in Virtex FPGAs. Each slice has two SRAM-based 4-to-1 lookup tables and two 1-bit registers. A BRAM block is a 2-KByte memory block embedded in the Virtex2-Pro architecture. The Xilinx XC2VP100 FPGA has 44,096 slices and 444 BRAMs.

[5]speedup= $\frac{\text{latency}_{Xil}}{\text{latency}_{gen}}$ where latency_{Xil} is 0.61 μsec and 8.59 μsec for DFT_{64} and DFT_{1024}, respectively

[6]Note that for DFT_{64} at $p > 8$, the minimum-slice option is indistinct from option 3.

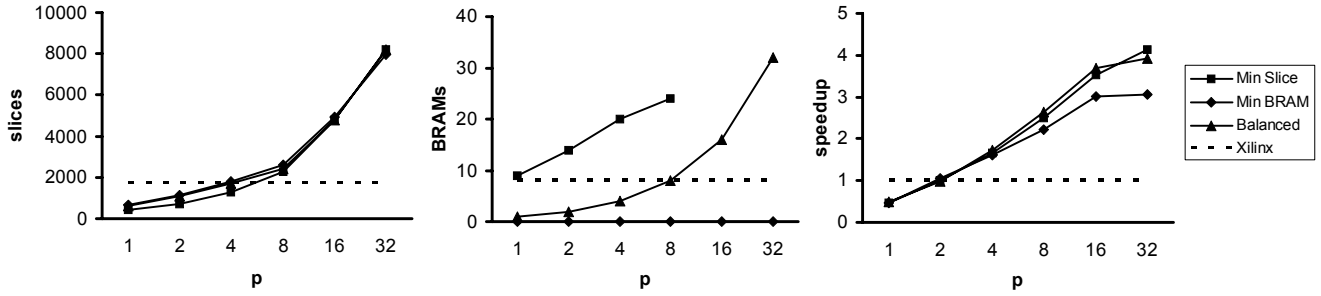


Figure 6: Synthesis Results for DFT_{64}. From left to right: slice utilization, BRAM utilization, and throughput performance.

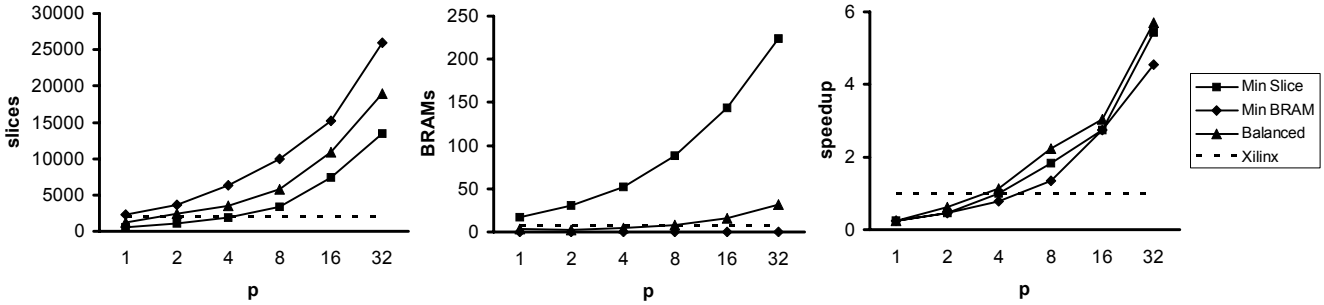


Figure 7: Synthesis Results for DFT_{1024}. From left to right: slice utilization, BRAM utilization, and throughput performance.

importantly, by varying p, our generator can further produce DFT cores of higher or lower performance in exchange for more or less resources. For DFT_{1024} at $p = 4$, our generated core with balanced resource assignments also matches the performance of the Xilinx core but requires 82% more slices. However, if surplus BRAMs are available, the difference in slices can be eliminated by opting to minimize slice usage. The complete set of results corresponding to option 1 (minimizing slices) and option 2 (minimizing BRAMs) further delineates the additional range of customization flexibility afforded by parameterizing user preference for spending logic slices versus BRAMs.

6. RELATED WORK

Creating an optimized hardware implementation of a DSP transform is a non-trivial undertaking since it requires combined expertise in both transform mathematics and hardware design. There are companies that specialize in creating custom hardware implementations of DSP transforms for customers who do not have the necessary expertise in-house. Commercially, besides the Xilinx LogiCore library, ready-to-use DFT cores are supplied by nearly every technology vendor's IP library and numerous third party IP developers. The radix-2 and radix-4 Cooley-Tukey based DFT cores are the current mainstay designs.

Optimized hardware implementations of DFTs remain an area of active interest and research. Kumhom, et al. [2] propose a universal DFT processor that is scalable in the number processing elements (arithmetic unit, local memory and address generator) connected by an on-chip reconfigurable network. Choi, et al. [1] present a radix-4 Cooley-Tukey based FFT datapath that can be scaled by a factor of up to 4 to trade off between energy and throughput. In their study, their comparison to the Xilinx cores assumes all cores are fixed at 100 MHz operation, and they do not show that their range of scaling includes the cost/performance tradeoff point occupied by the Xilinx cores.

7. CONCLUSIONS

We presented a parameterized soft core generator for DFTs based on the Pease FFT. Besides standard parameters such as input size and data format, the generator accepts a microarchitectural input parameter that controls the degree of concurrency in the generated DFT cores. Different cost/performance tradeoff points are achieved by varying the hardware mapping of the Pease FFT. The results show that our approach can yield DFT cores that are comparable in quality to DFT cores from the Xilinx LogiCore library. We further show that by varying the user controlled parameters, the generator returns ready-to-use DFT cores that are customized to user-specific requirements. This work is part of the Spiral project [3], which aims to automate the implementation and optimization of DSP functionality in software and hardware.

8. ACKNOWLEDGMENTS

This work was supported by NSF through awards ACR-0234293, SYS-0310941, and ITR/NGS-0325687.

9. REFERENCES

[1] 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*, 2003.

[2] P. Kumhom, J. Johnson, and P. Nagvajara. Design, optimization, and implementation of a universal FFT processor. In *Proc. 13th IEEE ASIC/SOC Conference*, 2000.

[3] Spiral project. www.spiral.net.

[4] J. Takala, T. Jarvinen, P. Salmela, and D. Akopian. Multi-port interconnection networks for radix-r algorithms. In *Proc. IEEE Intl. Conf. Acoustics, Speech, Signal Processing*, 2001.

[5] C. Van Loan. *Computational Framework of the Fast Fourier Transform*. SIAM, 1992.

[6] Xilinx, Inc. *Xilinx LogiCore: Fast Fourier Transform v3.1*, November 2004.