Fast and Accurate Resource Estimation of Automatically Generated Custom DFT IP Cores

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

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

ABSTRACT

This paper presents an equation-based resource utilization model for automatically generated discrete Fourier transform (DFT) soft core IPs. The parameterized DFT IP generator allows a user to make customized tradeoffs between cost and performance and between utilization of different resource classes. The equation-based resource model permits immediate and accurate estimation of resource requirements as the user considers the different generator options. Furthermore, the fast turnaround of the model allows it to be combined with a search algorithm such that the user could query automatically for an optimal design within the stated performance and resource constraints.

Following a brief review of the DFT IP generator, this paper presents the development of the equation-based models for estimating slice and hard macro utilizations in the Xilinx Virtex-II Pro FPGA family. The evaluation section shows that an average error of 6.1% is achievable by a model of linear equations that can be evaluated in sub-microseconds. The paper further offers a demonstration of the automatic design exploration capability.

Categories and Subject Descriptors

B.6.3 [**Hardware**]: Design Aids—*Automatic synthesis*

General Terms

Algorithm, Design

Keywords

Discrete Fourier transform, FPGA resource estimation, IP, design generator

1. INTRODUCTION

Ready-to-use IP designs of complex functionalities greatly reduce the time and effort of hardware development. However, IP designs in the form of static library modules have the limitation that they cannot match the optimal cost/performance tradeoff encountered in every application scenario. As a solution to this problem, for certain domains, important IP functionality can be generated and customized automatically to satisfy application requirements. In previous work, we have described an example of this approach: a parameterized IP generator for the discrete Fourier transform (DFT) [1]. Using this generator, the user can control microarchitectural tradeoffs between cost and performance, as well as the tradeoff between utilization of different classes of resources (e.g., slices vs. block RAM when targeting Xilinx FPGAs). We showed that by setting the control parameters accordingly, the generator can produce DFT cores comparable to the Xilinx LogiCore library as well as cores spanning a wide range of design tradeoffs not available in that library.

An opportunity exists to couple such a parameterized design generator with a feedback-driven search algorithm to make possible automatic design space exploration and optimization. The end user could then request the desired customized IP block in simpler terms, such as performance requirements and available resources, and the search would find the optimal solution. This way, the end user would be freed from understanding and configuring the parameters of the generator.

The main obstacle in realizing this vision is the latency of the evaluation feedback loop. For example, starting from the synthesizable RTL-level description produced by the DFT IP generator, the Xilinx ISE synthesis flow can take tens of minutes to produce the post-map resource utilization report for a typical 1024-point DFT design (and exponentially longer on larger designs). Consider that an exhaustive search of the design space for a 1024-point DFT involves evaluating 138 different design variations (already assuming that the parameters controlling numerical accuracy are fixed), the latency of synthesis-based estimation is prohibitively expensive even if the design exploration process is fully automated.

To circumvent the prohibitive cost of post-map resource estimation, this paper proposes an accurate equation-based resource utilization model for the automatically generated DFT cores. The model reduces the time for resource evaluation per design to the order of microseconds, and thus makes it practical to exhaustively explore the design space covered by the DFT IP generator. The model presented in this paper is tuned to target Xilinx Virtex-II Pro FPGAs. However, the modeling approach is extensible to other synthesis targets by recalibrating the technology-related coefficients in the equations.

In Section 2, we review the DFT IP generator, including the parameterized microarchitecture of the DFT datapath and the degrees of freedom controllable by the end user. We first present the complete resource modeling equations for the generated DFT cores in

FPGA'06, February 22–24, 2006, Monterey, California, USA.

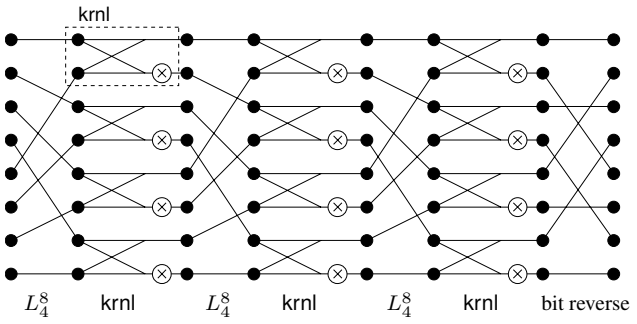

Figure 1: The dataflow graph of the Pease DFT for size $n = 8$.

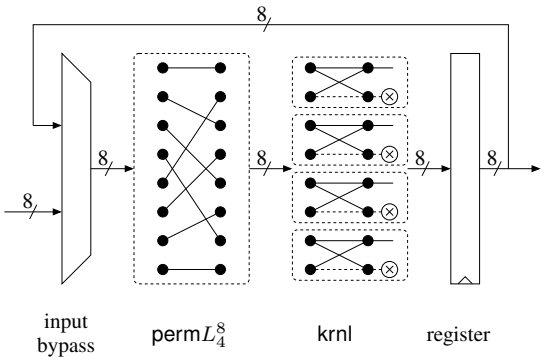

Figure 2: A fully horizontally-folded Pease DFT, size $n = 8$ ($p = n/2$).

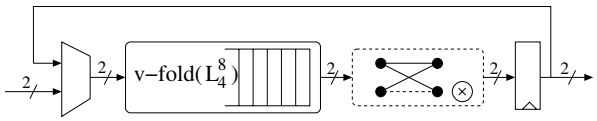

Figure 3: A fully horizontally and vertically-folded Pease DFT, size $n = 8$ ($p = 1$).

an overview in Section 3. Then, Section 4 expands on the details of the models that require clarification. We offer models for estimating the utilization of both generic logic resources (i.e., slices) and hard macros (i.e., block RAM and multipliers). Due to the different synthesis and mapping algorithms employed for slices and hard macros, they are handled differently. The slice utilization is modeled by empirically developed linear equations parameterized by coefficients, which are then found through least squares fitting; the utilization of block RAMs and multiplier macros is modeled by exact equations. Section 5 evaluates the proposed resource model in terms of accuracy (relative to post-map report produced by Xilinx ISE) and evaluation latency. The results show that the slice estimation has an average error of 6.1%. To illustrate the capability of this approach, we apply the resource model in an exhaustive search to find, for DFT sizes ranging from 16 point to 4096, the best feasible implementation for each FPGA in the Xilinx Virtex-II Pro family. Lastly, Section 6 surveys related prior work in high-level resource estimation, and Section 7 offers concluding remarks.

2. DFT IP CORE GENERATOR

This section provides an overview of the DFT IP generator. First, we introduce the Pease DFT algorithm, which underlies all DFT cores synthesizable by the generator. Then we explain wthe DFT IP generator's user-controlled parameters, which affect functionality and resource usage of the generated DFT cores.

2.1 Pease DFT algorithm

The DFT generator presented in [1] outputs DFT cores for complex input vectors based on the Pease FFT algorithm [2][1]. An example dataflow of the Pease algorithm for a DFT of size 8 is given in Figure 1. The computation kernel in the dataflow is a 2-input/2-output submodule (identified as the krnl module) that comprises a "butterfly" and a multiplication by a complex-valued constant (the so-called twiddle factor). The butterfly computes the sum and the difference of the two complex-valued inputs. For a DFT of two-power size n, $n\log_2(n)/2$ such kernels are performed. In a dataflow representation, the $n\log_2(n)/2$ krnl modules can be arranged into an array of $n/2$ rows and $\log_2(n)$ columns as shown in Figure 1 for $n = 8$. A notable feature of the Pease algorithm is that an identical *stride permutation* (defined later) shuffles the data after each of the $\log_2(n)$ computation stages; the stride permutation in Figure 1 is called L_4^8. After the final computation stage of the dataflow in Figure 1 is a "bit-reverse" permutation. As with many DFT implementations, our generator omits this stage in the generated cores. Thus, the generated cores accept input vectors in natural order and produce output vectors in bit-reversed order; an alternative option generates cores for DFTs with bit-reversed ordered input vectors and natural-ordered output vectors.

The regular structure of the Pease DFT dataflow allows our generated DFT core to instantiate only a single permutation and a single column of krnl modules to be reused iteratively $\log_2(n)$ times, as shown in Figure 2. This results in resource savings of approximately a factor of $\log_2(n)$ without a latency penalty. One should note that in this so-called horizontally folded implementation, the lower output of the butterfly in the krnl module must be multiplied by a different twiddle factor in each of the $\log_2(n)$ iterations. These factors are stored in tables of size $n\log_2(n)/2$ or $n/2$, depending on a user-specified parameter.

The regular structure of the Pease DFT dataflow further allows reusing the krnl modules in the vertical dimension. Figure 3 shows a maximally folded DFT datapath with only a single krnl module that is iteratively reused $n\log_2(n)/2$ times per DFT. Our DFT generator can produce the full range of vertically folded implementations where the number of krnl modules is a two-power p between 1 (minimum resource, minimum performance) and $n/2$ (maximum resource, maximum performance). Ideally, both performance and resource usage scale linearly with p, the number of krnl modules used. Notice that in Figure 3—and in vertically folded implementations in general—the data vector is presented as a data stream sequenced in time. Therefore, to permute a vector, a folded implementation of the stride permutation must buffer the data stream to shuffle the data elements in time. Our generator folds the stride permutation using the method in [3].

2.2 DFT generator parameters

The DFT IP generator produces synthesizable RTL-level Verilog descriptions of DFT cores [4]. To support application-specific customizations, many details of the generated core are controllable by the end-user. The user-controlled parameters can be grouped into two categories: those that control the functionality of the DFT core and those that the control hardware implementation choices. The parameters are described below and summarized for convenience in Table 1.

[1]FFT or fast Fourier transform refers generally to the class of $O(n\log(n))$ algorithms for the DFT.

The following parameters control the functionality of the generated IP cores:

- n: The generated core computes the DFT of an n-element complex-valued input vector and outputs a complex-valued n-element output vector. n must be a two-power and $n \geq 4$.
- w: w specifies the precision of the fixed-point representation of the input, the output, and the intermediate data values. $2 \cdot w$ bits are used to represent both the real and imaginary components.
- t: t specifies the precision of the fixed-point representation for the precomputed twiddle factors. The real and the imaginary components are each represented using 1-bit sign, 1-bit integer, and $t-2$-bit fraction. We require $2 < t \leq w$.
- dir: This boolean parameter selects whether the generated core accepts natural-ordered input vectors and outputs bit-reversed-ordered output vectors ($dir = 0$), or vice versa ($dir = 1$).
- $scale$: This boolean parameter determines whether the fixed-point data representation of intermediate values is scaled by a factor 2 after each krnl module to avoid overflow.

The following parameters control implementation choices:

- p: This parameter specifies the number of krnl modules instantiated in the DFT implementation. This is the primary microarchitectural parameter that allows the end-user to customize the tradeoff between performance and resource use. The input and output data vectors are streamed $2p$ elements per cycle over $n/(2p)$ cycles. Parameter p must be a two-power and $1 \leq p \leq n/2$.
- $twid$: This parameter controls whether the twiddle factor tables use the 16-Kbit block RAM (BRAM) macros in the Xilinx Virtex-II Pro FPGA architecture ($twid = 0$), or if one of two distributed RAM methods are used ($twid = 1$ or 2). If the twiddle factor tables are greater than 16-Kbit in size, BRAM mode is selected automatically (see Section 4.2).
- thr: If $p < n/4$, the stride permutation module requires FIFO queues ranging in depth from 2 to size $n/(4p)$. FIFOs deeper than thr use BRAMs for storage. FIFO queues requiring more than 16-Kbit of storage are automatically set to use BRAM.

Given a fixed DFT size n and fixed numerical accuracy (bitwidth w and twiddle bitwidth t), the generator can produce

$$3[(\log_2(n) - 1)\log_2(n)/2 + 1] \quad (1)$$

distinct DFT cores with differing resource and performance characteristics. Equation (1) arises from the different available choices of p, $twid$, and thr.

3. RESOURCE MODELING: OVERVIEW

This section presents our model to quickly and accurately estimate the resource requirements for any generated DFT core after synthesis and mapping to a Xilinx Virtex-II Pro FPGA. The model for the general-purpose logic resources (i.e., slices) consists of a hierarchical set of empirical equations, which take as input the same parameters as the DFT generator (summarized in Table 1). The utilization of hard macros in Xilinx FPGAs—block RAMs (BRAMs) and 18x18 multipliers—is expressed separately by exact equations. Although the presented models are developed in the context of Xilinx Virtex-II Pro FPGAs, the factors specific to the Xilinx Virtex-II Pro architecture are captured as empirically determined coefficients that can be meaningfully recalibrated for other synthesis target technologies.

	meaning	range
n	DFT size	2^2 to 2^{14}
w	datapath bitwidth	6 to 36
t	twiddle bitwidth	6 to w
dir	location of bit-reversal	input or output
$scale$	arithmetic mode	scaled or unscaled
p	degree of parallelism	1 to $n/2$
$twid$	twiddle storage	three twiddle storage options
thr	FIFO of size $\geq thr$ is implemented in BRAM	2 to $n/(2p)$

Table 1: DFT IP generator parameters

3.1 Slice utilization

A slice is the general purpose logic element in Xilinx Virtex-II Pro FPGAs. A slice contains two 4-to-1 function tables (that can alternatively be used as RAM, ROM, or FIFO), two bits of storage (either as registers or latches), and a small number of simple gates with hardwired functionalities (such as multiplexing, carry-propagation, etc.) [5]. To simplify the presentation, we first present our slice model for the cases where the generated DFT core is parameterized to use only slices and not BRAMs (i.e., when $twid \neq 0$, $thr > n/(4p)$).

In the Xilinx synthesis flow, RTL-level structural descriptions are reduced to slices indirectly through several steps of complex transformations. Since the exact transformations employed are not known, an accurate estimation of the slice utilization by direct inspection of the RTL-level description is not possible. Although we can develop accurate equations to model the slice utilization of some simple primitive modules, estimating the slice utilization of a higher level module as the sum of its constituent submodules leads to erratic outcomes. Furthermore, aggressive optimizations during synthesis can make estimations of even seemingly simple primitives unreliable. For example, the number of slices necessary to realize a read-only constant table is highly dependent on the values stored and how "compressible" they are.

Our estimation relies on an empirically developed model of linear combinations of terms with fitted coefficients. For example, for a primitive module (without submodules) whose number of slices $S = S(w, p)$ depends on w and p , we may begin with a model of the form

$$S(w, p) = c_0 \cdot pw + c_1 \cdot p + c_2 \cdot w + c_3. \quad (2)$$

The coefficients $c_0, \ldots, c_3$ in the linear model are then determined by standard least-squares fitting from a sufficiently large number of synthesized reference designs for representative values of w and p. Terms in the equation with negligible coefficients can be eliminated from the model. If the fitted model remains in poor agreement with the synthesized references, this is an indication that additional terms may be needed to account for the differences.

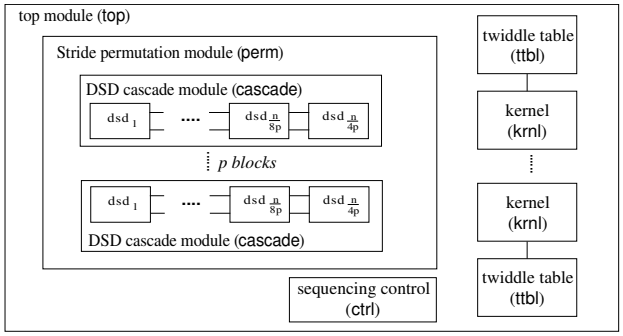

Figure 4: Hierarchical diagram of the generated DFT core, illustrating the modules: top, perm, cascade, dsd, ctrl, ttbl, and krnl.

constant	value
$c_{\text{perm},0}$	0.93
$c_{\text{perm},1}$	19.00
$c_{\text{perm},2}$	3.86
$c_{\text{perm},3}$	-138.46
$c_{\text{ctrl},0}$	0.76
$c_{\text{ctrl},1}$	2.73

(a)

constant	value
$c_{\text{cascade},0}$	0.50
$c_{\text{dsd},0}$	3.00
$c_{\text{dsd},1}$	0.03
$c_{\text{dsd},2}$	1.00

(b)

	$twid = 0$	$twid = 1$	$twid = 2$
$c_{\text{ttbl},0}$	0	0.024	0.027
$c_{\text{ttbl},1}$	0	0.147	-3.194
$c_{\text{ttbl},2}$	0	-0.397	0.131
$c_{\text{ttbl},3}$	0	67.261	129.709

(c)

Table 2: Coefficient values used in: (a) perm and ctrl, (b) cascade and dsd, and (c) ttbl models.

Similarly, we build a slice usage estimate S_{mod} for a higher-level module recursively from the estimates $S_{\text{submod},j}$ of its submodules (indexed by j) as a linear combination:

$$S_{\text{mod}}(\ldots) = c_{\text{mod},0} \sum S_{\text{submod},j} + c_{\text{mod},1} \cdot \text{term}_1 + \cdots + c_{\text{mod},k} \cdot \text{term}_k + c_{\text{mod},k+1}. \quad (3)$$

The first term of (3) is a weighted sum of the submodule estimates. The latter terms $\text{term}_1, \ldots, \text{term}_k$, suitably chosen, account for overhead and glue logic necessary to form the current hierarchy

Figure 4 graphically shows the hierarchical module structure of the generated DFT cores, and Figure 5 summarizes the recursive slice model we have constructed. Each node in Figure 5 corresponds to a module in the hierarchy and states a slice estimate in the form of (2) or (3) above. The empirically determined coefficients (when the synthesis target is a Xilinx Virtex-II Pro FPGA) have been compiled in Tables 2–4. In Section 4, we highlight key details of this model that require additional attention.

	$w > 18$		$w \leq 18$
	$t \leq 18$	$t > 18$	$t \leq 18$
$dir = 1$			
$c_{\text{krnl},0}$	14.0	19.5	8.0
$c_{\text{krnl},1}$	9.5	13.2	1.4
$c_{\text{krnl},2}$	6.7	77.0	8.4
$dir = 0$			
$c_{\text{krnl},0}$	17.0	20.5	6.2
$c_{\text{krnl},1}$	9.8	11.7	0.8
$c_{\text{krnl},2}$	-26.7	78.0	14.8

Table 3: Coefficient values used in the krnl model.

3.2 Block RAM utilization

Xilinx Virtex-II Pro FPGAs contain dedicated on-chip memory elements called block RAMs (BRAMs). The aspect ratio of this 16K-bit memory primitive can be optionally configured to be between 16Kx1-bit and 512x32-bit [5]. Unlike the algorithms for mapping logic to slices, the algorithm for inferring BRAM from RTL-level "array" constructs is straightforward. The number of BRAMs consumed in a generated DFT core thus can be stated exactly in a parameterized equation.

Our DFT generator can be parameterized to employ BRAM instead of slices to store twiddle constants (when $twid = 0$) and to implement FIFO queues (when $thr \leq n/(4p)$). Their modeling equations are derivable directly from our parameterized DFT core architecture. Below, we first offer the equations without justification. Section 4.1 and Section 4.2 provide further details.

The number of BRAMs consumed by a twiddle table is

$$B_{\text{ttbl}}(n,t,p) = \begin{cases} 0 & \text{if } twid \neq 0 \\ p & \text{else if } t'h \leq 2^{13} \text{ and } t' \leq 16 \\ 2p & \text{else if } t'h \leq 2^{14} \\ \ell p & \text{else} \end{cases} \quad (4)$$

where

$$t' = 2^{\lceil \log_2(t) \rceil}$$

is the effective twiddle word width,

$$h = \begin{cases} \frac{n}{2} & \text{if } p \leq \log_2(n) \\ \frac{n}{2} \cdot \frac{\log_2(n)}{p} & \text{if } p > \log_2(n) \end{cases}$$

is the number of words per table, and

$$\ell = 2^{\lceil \log_2(2t'h/2^{14}) \rceil}$$

is the number of BRAMs per table.

The vertically folded stride permutation comprises $2p$ FIFOs of sizes 2^0 through $2^{\log_2(n/(4p))}$. For FIFO depth d greater than thr, the FIFO is implemented using BRAMs.

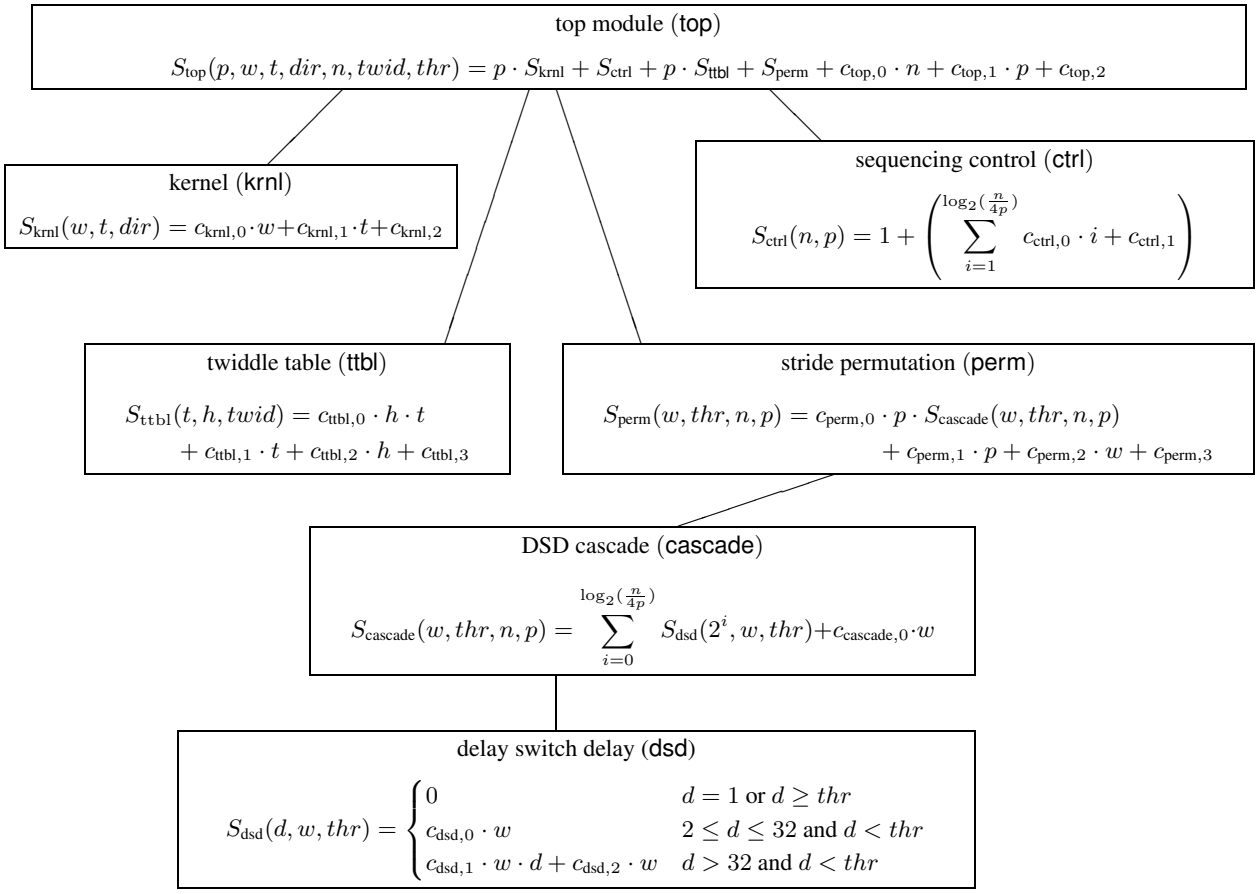

Figure 5: The complete hierarchical slice model

	$w = 8$			$w = 16$			$w = 32$		
	$twid = 0$	$twid = 1$	$twid = 2$	$twid = 0$	$twid = 1$	$twid = 2$	$twid = 0$	$twid = 1$	$twid = 2$
$dir = 1$									
$c_{\text{top},0}$	0.0767	0.1709	0.2887	0.3045	0.8350	1.9663	-0.7310	0.8465	-28.4821
$c_{\text{top},1}$	3.3968	16.9921	7.9950	-132.0714	-319.5112	-156.8810	1.9041	10.4357	73.8951
$c_{\text{top},2}$	49.0620	-44.9364	-45.1627	316.5599	692.0835	205.7261	-109.2496	-496.8318	40.3796
$dir = 0$									
$c_{\text{top},0}$	0.0397	0.1166	0.2198	0.2968	0.7855	0.9067	-0.9366	0.5265	-4.8235
$c_{\text{top},1}$	-18.4841	-10.3651	8.6668	-153.7381	-249.2857	-209.2381	-7.3549	39.2352	-3.3765
$c_{\text{top},2}$	111.2902	9.6899	-91.1419	288.5831	460.7778	379.2339	-13.2356	-927.9078	-128.4907

Table 4: Coefficient values used in the top model

The number of BRAMs consumed by a FIFO of depth d is

$$B_{\text{FIFO}}(d, w, thr) = \begin{cases} 0 & \text{if } d < thr \\ 1 & \text{else if } w'd \leq 2^{13},\ w' \leq 16 \\ 2 & \text{else if } w'd \leq 2^{14} \\ 2^{\lceil \log_2(2w'h/2^{14}) \rceil} & \text{else} \end{cases} \quad (5)$$

where $w' = 2^{\lceil \log_2(w) \rceil}$ is the effective data width.

Therefore, the total number of BRAMs used in a DFT core is:

$$B(n, w, p, thr) = B_{\text{ttbl}}(n, w, p) + 2p \left(\sum_{i=0}^{\log_2(\frac{n}{4p})} B_{\text{FIFO}}(2^i, w, thr) \right). \quad (6)$$

When twiddle tables and FIFOs are implemented using BRAMs, their corresponding contributions in the slice model need to be subtracted. Further, the amount of overhead and glue logic (in slices) to incorporate a submodule is also affected by whether the submodule is constructed from slices or BRAMs. These effects are reflected in the slice model by 1) requiring an alternative set of coefficients or 2) requiring different equations to be applied at a given hierarchy depending on the setting of DFT generator parameters that control the BRAM usage.

3.3 Multiplier utilization

Xilinx Virtex-II Pro FPGAs provide hardwired 18x18 multiplier macros which are used automatically by the synthesis tool to realize multiplications in RTL-level descriptions. Like BRAM usage, the utilization of multipliers follows a simple formula.

$$m(p, w, t) = \begin{cases} 4p & \text{if } w \leq 18 \text{ and } t \leq 18 \\ 8p & \text{if } w > 18 \text{ and } t \leq 18 \\ 16p & \text{if } w > 18 \text{ and } t > 18 \end{cases} \quad (7)$$

A total of p complex multipliers of width w by t are required per DFT core. In the most basic case, each complex multiplier is implemented with 4 real multipliers and two real adders. However, when $w > 18$ or $t > 18$, additional 18x18 bit multipliers are needed. This problem is discussed further in Section 4.3.

4. MODEL DETAILS

This section expands on the details in the resource utilization model that require more in-depth knowledge of the Pease DFT datapath. The modules covered in this section are the stride permutation (perm module) in Section 4.1, the twiddle table (ttbl module) in Section 4.2, and the kernel (krnl module) in Section 4.3.

4.1 Stride permutation

The perm module implements the stride permutation (also called perfect shuffle or corner turn) in the Pease DFT. A stride permutation, denoted as L^n_m, reorders an input vector such that the element at position $i \cdot (n/m) + j$ in a vector becomes the element at position $j \cdot m + i$ (for $0 \leq j < n/m$ and $0 \leq i < m$). The Pease DFT requires either $L^n_{n/2}$ or L^n_2, depending on the setting of the interface ordering parameter dir.

When the DFT core is not vertically folded (i.e., $p = n/2$), the stride permutation is simply a wired reordering in space. When the datapath is vertically folded (i.e., $p < n/2$), the data vector is presented as a data stream of $2p$ elements per cycle over $n/2p$ cycles. Thus, the folded perm module requires memory to buffer and reorder the data stream.

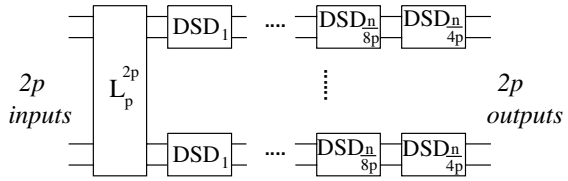

Figure 6: A vertically folded $L^n_{n/2}$ permutation with $2p$ ports.

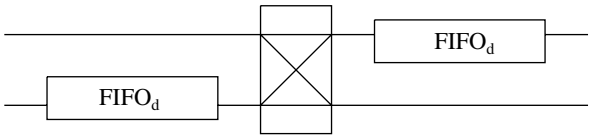

Figure 7: The dsd block.

Takala, *et al.* [3] describe an efficient construction to vertically fold a stride permutation for any p. For $1 \leq p < n/2$, $L^n_{n/2}$ can be realized as shown in Figure 6. (L^n_2 is the left-right mirror image.) On each cycle, a $2p$-element subvector enters the perm module and first undergoes an L^{2p}_p permutation by wire reordering. For the next $n/(2p) - 1$ cycles, the $2p$-element vector passes through $\log_2(n/8p)$ successive stages of *delay-shift-delay* (dsd) modules. A dsd block, shown in Figure 7, consists of two FIFOs of depth d and a programmable switch that either allows its inputs to pass through or criss-cross. A chain of dsd modules where d increases from 1 to $n/4p$ constitutes the "DSD cascade" module (cascade) in the hierarchical slice model (Figure 5).

FIFOs in slices. The slice model of the perm module is developed recursively from the bottom up using the approach explained in Section 3.1. We obtain an exact expression (shown at the bottom of Figure 5) for the number of slices in a dsd module when the FIFOs in the DSD cascade are built using slices (i.e., $d > thr$). Notice that for $d < 32$, the number of slices used is constant because any FIFO of depth less than 32 still consumes the same number of 32-entry FIFO primitives (based on lookup tables) in its construction. Moving up the hierarchy in Figure 5, the cascade model sums the slice estimates for the instantiated dsd modules and adds an overhead dependent on the bitwidth w. Next, the model for perm sums the slice estimate for p parallel cascade modules and adds a constant and an overhead that is a linear combination of p and the bitwidth w.

FIFOs in BRAM. FIFOs deeper than the user-set threshold thr are implemented using BRAMs instead of slices. The aspect ratio of the 16-Kbit BRAM is configurable but only for data widths that are two-powers. Therefore, to determine the number of BRAMs utilized, the user-specified data word width w first is rounded up to the effective word width $w' = 2^{\lceil \log_2(w) \rceil}$. The number of BRAMs needed by a FIFO of depth d can now be easily calculated using (5) (from Section 3.2) based on its effective storage requirement $2w'd$.

In Equation (5), the first case, $d < thr$, corresponds to when a FIFO is to be implemented using slices. The second case corresponds to all implementations where the FIFO only partially utilizes one 16-Kbit BRAM. The third case is another transitional case where the FIFO is bigger than one but smaller than two BRAMs. In this case, the imaginary and the real components of the data values are split across two BRAMs. Larger FIFO sizes are captured by the final case which rounds up the effective FIFO storage requirement to correspond to the storage capacity of a two-power number of BRAMs.

4.2 Twiddle factor table

The krnl modules in the generated DFT core are reused iteratively for different instances of the krnl module in the unrolled dataflow. Thus, the multiplier in a krnl module must be supplied with a different twiddle factor (a complex constant) in each iteration of reuse. The DFT IP generator pre-computes these twiddle factors and includes them as constant tables in the generated RTL Verilog description.

For a given n, there are $n\log_2(n)/2$ complex multiplications to be performed by p krnl modules; each krnl module is thus responsible for $n\log_2(n)/(2p)$ multiplications. In a naive approach (i.e., $twid = 2$), each krnl module maintains a private table of $n\log_2(n)/(2p)$ twiddle factors that it needs. The twiddle factors are stored in this table in the order they are needed to simplify the indexing logic. The total number of twiddle constants stored in this case is $n\log_2(n)/2$, independent of p.

A common optimization to reduce twiddle storage cost leverages the fact that only $n/2$ unique constants are used. In this option (i.e., $twid = 1$), each krnl module maintains a private table of all $n/2$ twiddle factors because there is no natural way to partition the factors. Also, the krnl module requires a somewhat involved index generator to read out from the table the required twiddle factors in the order used. The number of twiddle constants stored in this case is $p\cdot(n/2)$. Under idealized assumptions, this option yields savings when $p < \log_2(n)$.

Table in Slices. With either storage method, the storage cost, in terms of the number of twiddle factors stored times the number of bits per twiddle factor, is known exactly. However, when these tables of constants are mapped to use lookup tables in slices—a preferable option when the tables are small or when BRAMs are scarce—the number of slices actually consumed is much smaller then expected. We have determined by examination that Xilinx ISE does not instantiate these tables of constants literally but rather spends considerable effort to minimize the redundancies in the table at the bit level. The degree of compression varies with the storage method and the table size.

Thus, the slice model for the twiddle factor table (given in Figure 5, repeated below) is based on the linear combination of four terms.

$$S_{\text{ttbl}}(t,h,twid) = c_{\text{ttbl},0}\cdot ht + c_{\text{ttbl},1}\cdot t + c_{\text{ttbl},2}\cdot h + c_{\text{ttbl},3}. \quad (8)$$

In this equation, h corresponds to the number of entries in the table and t is the bitwidth of the twiddle factors. The first term proportional to ht is the expected size of the table in bits. The latter terms attempt to capture the dependencies on h and t alone. This equation works for both $twid = 1$ and $twid = 2$, but the values of the weighting coefficients $c_{\text{ttbl},0}$, $c_{\text{ttbl},1}$, $c_{\text{ttbl},2}$, and $c_{\text{ttbl},3}$ (given in Table 2(c)) depend on the twiddle storage method used.

Table in BRAM. For large twiddle tables, it make sense—both in terms of resource utilization and in terms of performance due to routing delay—to utilize the more compact BRAMs for storage (i.e., $twid = 0$). The calculation of BRAMs used follows a simple formula analogous to the calculation of the BRAM usage in the dsd module's FIFOs.

The height of the twiddle tables to be stored in BRAM is given by

$$h = \begin{cases} \frac{n}{2} & \text{if } p \le \log_2(n) \\ \frac{n}{2}\cdot\frac{\log_2(n)}{p} & \text{if } p > \log_2(n) \end{cases},$$

where the two cases correspond to the two twiddle table options (i.e., $n\log_2(n)/2p$ vs. $n/2$ twiddle factors per krnl module). This decision is made automatically when $twid = 0$ because the relative advantage of the two methods is decidable by $p > \log_2(n)$ exactly.

The effective table height $h' = 2^{\lceil\log_2(\frac{2w'\cdot h}{2^{14}})\rceil}$ and the effective twiddle bitwidth, $t' = 2^{\lceil\log_2(t)\rceil}$ are used in determining the effective BRAM storage requirement. Based on $h'\cdot t'$, Equation (4) (given in Section 3.2) gives the number of BRAMs consumed by p twiddle tables. The four cases are exactly analogous to the four cases when calculating BRAM usage in dsd FIFOs in Section 4.1.

4.3 Nonlinear dependence on bitwidth

Slice utilization generally varies smoothly with data bitwidth w. However, as seen from the BRAM equations (4), (5), and (6), slice utilization can also vary in a nonlinear fashion when using hard macros with fixed native bitwidths or capacity.

Another example of this nonlinearity is in the implementation of the krnl module, which utilizes the FPGA's hardwired 18-bit by 18-bit multipliers. If the data bitwidth or twiddle bitwidth are greater than the native width of the multipliers, a twiddle multiplier needs to synthesized from several 18-by-18 multiplier hard macros. Furthermore, because these multipliers are in the critical delay path, the DFT generator readjusts the optimal pipeline depth of the krnl module accordingly. To handle this type of non-linearity associated with bitwidth in our linear modeling approach, the slice model requires different sets of weighting coefficients for different bitwidth regions (shown in Table 3). For the slice model reported in this paper, the weighting coefficients are separately tuned for bitwidths of 8, 16, and 32 bits. For intermediate bitwidths, the coefficients are interpolated.

5. RESULTS

This section reports the accuracy and runtime of the resource model. Only the slice model is evaluated for accuracy because the equations for BRAM and multiplier usage are exact. We also present the outcomes of a design space exploration experiment.

5.1 Slice model accuracy

For the purpose of this evaluation, we consider a design space of 8388 distinct DFT cores, corresponding to all possible combinations of the following parameters[2]:

$$\begin{aligned} n &\in \{2^3 \ldots 2^{14}\} \\ p &\in \{2^0 \ldots n/2\} \\ thr &\in \{2^1 \ldots n/(2p)\} \\ w &\in \{8, 16, 32\} \\ t &\in \{8, 16, 32\} \\ twid &\in \{0, 1, 2\} \\ dir &\in \{0, 1\} \end{aligned}$$

The resource estimates reported in this section are produced by the hierarchical slice model (Figure 5) whose coefficients derive from a training set of 162 selected DFT cores. These coefficients are reported in Tables 2, 3, and 4. This training set comprises less than 2% of the design space. To select the set, the parameter space is divided into 18 regions: (3 settings for w) $\times$ (3 settings for $twid$) $\times$ (2 settings for dir). For each of these 18 regions, we select 3 values for n and three values for p that are evenly spaced within their

[2]The current slice model is able to handle cases when $w \neq 8$, 16, or 32. In those cases, the coefficients listed in Table 4 are interpolated from existing values.

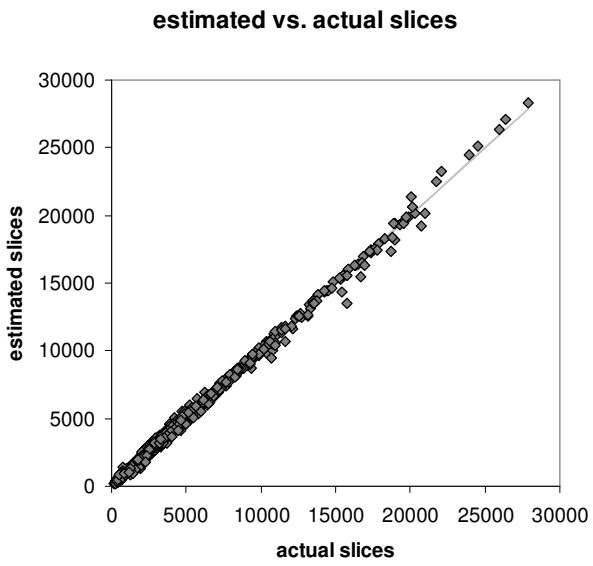

Figure 8: Actual vs. estimated slices for various generated DFT cores.

respective allowed[3] range. The choice of p and n are not fixed across regions because, for example, the allowed maximum values for n and p are larger when $twid = 2$ than when $twid = 1$. The coefficients for primitive submodules with simple parameterization, such as FIFOs, are calibrated independently. Due to the simplicity of these primitive submodules, relatively few reference points are needed to converge to the final coefficient values.

From the total design space outlined above, we select an evaluation sample of 962 DFT cores for synthesis and mapping using Xilinx ISE 7.1.03i. This sample includes a comprehensive coverage for 64, 1024 and 2048-point DFT. The set is constructed from the following parameters:

$$\begin{aligned} n &\in \{64, 1024, 2048\} \\ p &\in \{2^0 \dots 2^5\} \\ thr &\in \{2^1 \dots n/(2p)\} \\ w &\in \{8, 16, 32\} \\ t &\in \{8, 16, 32\} \\ twid &\in \{0, 1, 2\} \\ dir &\in \{0, 1\} \end{aligned}$$

(The parameter p is limited to 32 due to synthesis and mapping time of larger designs.) In addition, we sample 9 cores for $n = 2^4$ through $n = 2^{12}$ while fixing the other parameters ($p = 1$, $thr = n/2$, $w = 16$, $twid = 0$, and $dir = 1$). This evaluation sample represents 11.5% of the overall design space.

To evaluate the accuracy of the slice model, we compare the slice estimates calculated by the model to post-map estimates reported by Xilinx ISE for these 962 synthesized DFT cores. We present the results visually in a scatter plot in Figure 8 where the X-axis indicates the post-map slice estimate and the Y-axis indicates the model-based slice estimate. Each point corresponds to a DFT core in the test sample.

The fact that the points in Figure 8 are clustered tightly around the line through the origin with a slope of 1 gives indication that there is a high degree of correspondence between the estimated and the actual slice utilization. However, these results require a more careful interpretation. The average absolute error over the test sample is 6.12%. What is not obvious from the plot is that the maximal error is 74.98%. The practical significance of this error is small because the class of large percentage errors occur only in very small designs—the observed maximal error corresponds to a difference of only 584 slices.[4] There is an analogous caveat when considering the maximal error in terms of slices. The highest error observed is 2287 slices. This class of large errors only occurs in large DFT cores where they correspond to small percentage errors.[5] To clarify this analysis, we divide the test sample into four bins according the number of slices used. Table 5 separately reports for each bin the average absolute error and maximal error, in terms of slices and as a percentage of the complete design. Errors of these magnitudes and distribution do not interfere with the usefulness of this model in practice.

slices	abs. error (%)		abs. error (slices)	
	avg.	max.	avg.	max
< 5000	7.4	75.0	118	835
5000–10000	2.4	14.4	162	756
10000–15000	2.0	11.5	232	1235
> 15000	2.3	14.5	438	2287

Table 5: Average absolute error in slices and percentage, by size of implementation.

5.2 Evaluation speed

The main reason for introducing the resource usage models in this paper is to enable the user to quickly determine which DFT core to choose under given resource constraints. For example, for a DFT of size 2^{14}, the DFT IP core generator can produce 276 possible design (see (1)), provided the bitwidths w and t are fixed. If the latter are included in the search space, the number of possible designs is considerably larger. The synthesis and mapping of just one possible configuration of $\mathrm{DFT}_{2^{14}}$ takes on the order of hours using Xilinx ISE 7.1.03i. Using our model the time for evaluation becomes negligible. For example, the evaluation of our model for all 276 possible designs of $\mathrm{DFT}_{2^{14}}$ can be performed in less than a millisecond on a 2.8 GHz Pentium 4 system.

5.3 A possible application

We apply the resource model to search over the settings of p, $twid$, and thr to determine the *fastest* DFT_n (given fixed w and t) for $n \in \{16, 64, 256, 1024, 4096\}$ that fits within the resource budget of each part in the Xilinx Virtex-II Pro family. For this experiment, we fix $w = t = 16$ and $dir = 1$. For the purpose of this experiment, we evaluate performance in terms of execution latency in clock cycles.

[3]Such that the resulting design could still fit in the largest Xilinx Virtex-II Pro FPGA available.

[4]These large percentage errors occur because the real system that we model by linear system is highly nonlinear at these smallest sizes.

[5]These errors tend to occur in large DFTs with twiddle tables stored in slices. Furthermore, this area-intensive approach is not commonly used in DFT designs.

The latency in clock cycles for the generated DFT cores is inversely proportionally to p by the equation

$$\text{cycles} = \log_2(n)\left(\frac{n}{2p} + \text{pipe}\right), \quad (9)$$

where pipe is the krnl module's pipeline depth. Thus, this experiment reduces to finding the settings of thr and $twid$ to maximize p in a given FPGA. Although a true performance comparison must take into consideration clock frequency, we are able to simplify the problem in this way for a several reasons. The generated core's critical path delay is fixed by the arithmetic operations in the krnl module, except in cores that occupy a very large number of slices. On these large designs, routing delay begins to dominate. However, in our tests, the minimum cycle time never changes by more than a factor of 2. Since the latency in clock cycle reduces by (close to) a factor of 2 for each increment of p, we know the fastest implementation of a DFT at a given n must have the largest feasible p. From the experimental data in [1], we know varying $twid$ and thr have only a small impact on the critical path delay. Thus, all feasible designs with the maximal p have nearly the same maximum clock frequency.

The final results of this experiment are tabulated in Table 6. For each intersection of DFT size and part choice, the table reports the latency of the DFT in cycles and the generator parameters used. The data to compile this table is generated in a few milliseconds on a 2.8 GHz Pentium 4.

6. RELATED WORK

Fast and accurate cost estimation is a pivotal component of any high-level hardware design or automatic design exploration framework. Several prior research projects have developed resource modeling in those contexts.

Most closely related to our approach, Brandolese *et al.* [6] describe an approach to estimate FPGA resources (in terms of flip-flops and LUTs) for designs described in SystemC. Their approach analyzes the SystemC constructs in the description to generate a system of equations with fitted coefficients. As in our approach, these constant coefficients are pre-determined off-line against synthesized reference designs. They report an error ranging from 9 to 37% over six benchmarks. Kulkarni *et al.* [7] reduce a high-level hardware description (written in SA-C) to a dataflow graph and estimate its design costs by an equation that sums the cost over the node types. The coefficients corresponding to different node types are determined also by regression against synthesized reference designs. Bilavarn *et al.* [8] describe a similar high-level approach to high-level resource modeling.

Alternatively, Xu and Kurdahi [9] more directly estimate FPGA resources by predicting the mapping of designs from the netlist-level to lookup tables in FPGAs. This approach requires that a design is synthesized first into a netlist and furthermore the analysis is more time consuming than equation-based estimations.

In addition, others have developed coarse-grain cost estimation in terms of datapath units (e.g., number of adders, multipliers and registers). For example, Nayak *et al.* [10] and Bjreus *et al.* [11] both describe extraction of register and functional unit usage of MATLAB descriptions.

Whereas these other approaches all support a more general class of design inputs, our goal is to provide very fast and very accurate estimation for a restricted set of designs. Although the generated DFT cores span a large space of design tradeoffs, they nonetheless utilize a common microarchitecture and a common set of primitives. Thus, with a very high-degree of control and knowledge over

		DFT size				
FPGA		16	64	256	1024	4096
xc2vp2	cycles	28	114	536	5150	—
	thr	4	16	64	256	—
	$twid$	0	0	0	0	—
	p	2	2	2	1	—
xc2vp4	cycles	20	66	280	2590	—
	thr	2	8	32	256	—
	$twid$	0	0	0	0	—
	p	4	4	4	2	—
xc2vp7	cycles	16	42	152	1310	12324
	thr	2	4	16	128	512
	$twid$	1	1	1	0	0
	p	8	8	8	4	2
xc2vp20	cycles	16	30	88	350	6180
	thr	2	2	8	8	512
	$twid$	0	1	1	1	0
	p	8	16	16	16	4
xc2vp30	cycles	16	24	56	190	3108
	thr	2	2	4	4	2
	$twid$	0	1	1	1	1
	p	8	32	32	32	8
xc2vp40	cycles	16	24	56	190	1572
	thr	2	2	4	16	128
	$twid$	0	0	0	0	0
	p	8	32	32	32	16
xc2vp50	cycles	16	24	56	190	1572
	thr	2	2	4	16	128
	$twid$	0	0	0	0	0
	p	8	32	32	32	16
xc2vp70	cycles	16	24	40	110	804
	thr	2	2	2	8	32
	$twid$	0	0	1	1	0
	p	8	32	64	64	32
xc2vp100	cycles	16	24	40	110	420
	thr	2	2	2	8	32
	$twid$	0	0	0	0	0
	p	8	32	64	64	64

Table 6: Fastest generated DFT cores for Xilinx Virtex-II Pro FPGA series. $w = t = 16, dir = 1$

the generated DFT IP cores, we are able to build a simple model for our purpose.

In contrast, without the advantage of special domain restrictions, the prior work generally cannot achieve the level of accuracy we desire. The approaches that are intended for high-level design abstractions [7, 8, 10, 11] do not have the resolution in their design abstraction to capture the many low-level but important optimizations in the generated DFT cores. Lastly, approaches such as [6] and [9] are also not fast enough to enable real-time design space exploration.

7. CONCLUSIONS

This paper presented an equation-based model for estimating the slice usage and exact models for calculating the block RAM and multiplier usage of generated DFT IP cores. The model is accurate for practical purposes: in our tests, we estimate slices with an average absolute error of 6.1%. Due to its simplicity, the evaluation of the model is very fast, thus enabling exhaustive search over the available design parameters.

The model can be decomposed into two components: a set of parameterized equations that capture our understanding of the synthesis and mapping procedure, and a standard least squares fitting that determines the best choices of these coefficients from a given training set of generated designs. This structure should enable easy adoption to FPGAs with different logic resources by simply refitting the model starting from the same structure.

We believe that more important than the specific result is the general approach taken in [1] and this paper: replacing static IP design by flexible, domain-specific IP generators coupled with corresponding domain-specific resource and performance models. The IP generator encapsulates the degrees of freedom in implementing a well-known kernel functionality and is capable of generating the corresponding implementations. The resource and performance models, together with a simple search, enable the user to quickly find and instantiate the optimal design for her application constraints. In this paper we focused on modeling resource usage. To complete the approach we will consider cycle time, numerical accuracy, and power in future work.

We invite the reader to try the DFT IP generator at [4].

8. ACKNOWLEDGMENTS

This work was supported by DARPA under DOI grant NBCH-1050009 and by NSF awards ACR-0234293 and ITR/ACI-0325687.

9. REFERENCES

[1] G. Nordin, P. Milder, J. Hoe, and M. Püschel. Automatic generation of customized discrete Fourier transform IPs. In *Proceedings of the 42nd Annual Conference on Design Automation*, 2005.

[2] M. C. Pease. An adaptation of the fast Fourier transform for parallel processing. *ACM*, 15(2), April 1968.

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

[4] Spiral DFT IP generator. www.spiral.net/hardware/dftgen.html.

[5] Xilinx, Inc. *Xilinx Virtex-II Pro Platform FPGA Data Sheet*, June 2005.

[6] C. Brandolese, W. Fornaciari, and F. Salice. An area estimation methodology for FPGA based designs at SystemC-level. In *Proceedings of the 41st Annual Conference on Design Automation*, 2004.

[7] D. Kulkarni, W. Najjar, R. Rinker, and F. Kurdahi. Fast area estimation to support compiler optimizations in FPGA-based reconfigurable systems. In *Proceedings of the 10th Annual Symposium on Field-Programmable Custom Computing Machines*, 2002.

[8] S. Bilavarn, G. Gogniat, and J. L. Phillipe. Area time power estimation for FPGA based designs at a behavioral level. In *Proceedings of the 7th IEEE International Conference on Electronics, Circuits, and Systems*, 2000.

[9] M. Xu and F. Kurdahi. Area and timing estimation for lookup table based FPGAs. In *Proc. IEEE European Design and Test Conference*, 1996.

[10] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee. Accurate area and delay estimators for FPGAs. In *Proc. IEEE Design, Automation and Test in Europe Conference*, 2002.

[11] P. Bjuréus, M. Millberg, and A. Jantsch. FPGA resource and timing estimation from Matlab execution traces. In *Proc. of the 10th International Symposium on Hardware/software Codesign (CODES)*, 2002.