Generating FPGA-Accelerated DFT Libraries

Paolo D'Alberto [*]
Yahoo!
{*pdalbert@yahoo-inc.com*}

Peter A. Milder, Aliaksei Sandryhaila, Franz Franchetti
James C. Hoe, José M. F. Moura, and Markus Püschel
Department of Electrical and Computer Engineering
Carnegie Mellon University
{*pam,asandryh,franzf,jhoe,moura,pueschel*}*@ece.cmu.edu*

Jeremy R. Johnson
Department of Computer Science
Drexel University
jjohnson@cs.drexel.edu

Abstract

We present a domain-specific approach to generate high-performance hardware-software partitioned implementations of the discrete Fourier transform (DFT) in fixed point precision. The partitioning strategy is a heuristic based on the DFT's divide-and-conquer algorithmic structure and fine tuned by the feedback-driven exploration of candidate designs. We have integrated this approach in the Spiral linear-transform code-generation framework to support push-button automatic implementation. We present evaluations of hardware–software DFT implementations running on the embedded PowerPC processor and the reconfigurable fabric of the Xilinx Virtex-II Pro FPGA.

In our experiments, the 1D and 2D DFT's FPGA-accelerated libraries exhibit between 2 and 7.5 times higher performance (operations per second) and up to 2.5 times better energy efficiency (operations per Joule) than the software-only version.

1 Introduction

The goal of a hardware-software partitioned implementation is to achieve the fast execution time of a hardware implementation while retaining the flexible programmability of a software implementation. Typically, the most computation-intensive kernels that are conducive to hardware acceleration are extracted from an algorithm and realized as hardware, while the remaining computations are carried out in software. In such a scenario, the hardware kernels are utilized in different contexts under software control. Thus, the hardware–software implementation combines the flexibility of software and the performance benefits of hardware.

In general, the determination of an optimal partitioning strategy while satisfying a set of constraints is an NP-hard problem [1] and several heuristic methods for its solution have been proposed (e.g., [8,9]). In this paper, we present a domain-specific approach to generating high-performance hardware–software implementations of fast Fourier transform algorithms (FFT). The partitioning strategy is based on heuristics derived from the FFT's divide-and-conquer algorithmic structure and further refined by feedback-driven exploration of candidate designs.

The discrete Fourier transform (DFT) is an important primitive underlying many DSP applications. Fast algorithms to compute the DFT—called FFT algorithms—have been studied extensively and they are known to exhibit a regular structure (i.e., an FFT algorithm decomposes a large DFT into many smaller DFTs recursively). From this general structure, we infer that the hardware accelerated kernels must be in the form of throughput-optimized DFT cores for small problem sizes. When considering two-power problem sizes (i.e., DFTs on 2^n points), we only need to consider two-power sized DFT kernels (i.e., DFT_{2^k}). By offloading the appropriate kernels into hardware, the software receives the benefit of hardware acceleration and yet can still compute arbitrary (sized) DFTs on top of the available kernels. Different kernels synthesized in hardware yield different performance (e.g., operations per second) and necessitate different amounts of resources (e.g., logic or number of BRAM).

As a consequence, the DFT partitioning problem becomes the problem of selecting the appropriate set of

[*] The author worked on this project during his post-doctorate fellowship at the ECE department in CMU

0-7695-2940-2/07 $25.00 © 2007 IEEE
DOI 10.1109/FCCM.2007.58

IEEE computer society

throughput-optimized two-power sized DFT cores to satisfy a given resource constraint (logic, power, energy) while maximizing a scalar metric, such as performance. We present our solution to this problem in two parts. First, in what we call the forward design problem, we make use of the Spiral [14] generator framework to automatically produce a hardware–software implementation given a prespecified partitioning strategy, in our case defined by the set of DFT cores available in hardware. These hardware cores are generated using Spiral [12, 13] and are often faster and smaller than other available implementations. Second, we solve the inverse design problem: given the desired constraints and an objective function, select the optimal set of two-power-sized DFT cores to include in hardware. Together, the overall design generation problem is solved by first solving the inverse design problem by emulating the hardware cores (i.e., without synthesizing any DFT cores) and then the resulting forward design problem, given the candidate hardware cores. In solving the inverse design problem, a simple scalar optimization metric (e.g., runtime) is used to maximize the performance for a single DFT problem size. The scalar metric can also be defined as the average performance over multiple problem sizes (to optimize a DFT library) or a composite function that takes into account a combination of performance, power, and logic cost simultaneously.

As a demonstration, we present experimental results of applying our automatic generator to create hardware–software implementations of the DFT for the Xilinx XUP2VP development board with a Virtex-II Pro XC2VP30 FPGA. The 1D and 2D DFT problems of two-power and non-two-power sizes are partitioned into software (running on one of the two PowerPC cores in the FPGA) and hardware (comprised of the DFT cores instantiated in the reconfigurable fabric). The specific partitioning strategy to be decided in these experiments is which two two-power DFT kernels, ranging between size 2^5 to 2^{10}, must be synthesized in hardware to maximize a single problem's performance or the average library performance. Our evaluation includes the optimization of runtime, energy and power, and thus their Pareto tradeoff. In this paper, we focus solely on hardware–software solutions for embedded-system implementations; for example, see [14] for high-performance software solutions or [12, 13] for custom hardware solutions.

Synopsis. In Section 2, we briefly survey the related work. In Section 3, we present the necessary background for the DFT, Spiral and our evaluation platform. In Section 4, we first present the forward design problem of how to generate a concrete implementation from a DFT formula and a partitioning decision. In Section 5, we present the inverse design problem of arriving at the optimal partitioning strategy. We present our experimental results in Section 6 and conclude in Section 7.

2 Related Work

Companies like XtremeData and DRC are positioning FPGAs on the fast memory interconnects of high-performance PC workstations. These technologies promote a new computation paradigm with FPGAs as first-class processing elements alongside of traditional microprocessors.

An algorithm will nevertheless need to be partitioned—ideally with performance-critical kernels in hardware and control-intensive kernels in software—to take advantage of these new hybrid hardware–software platforms; an algorithm needs this partitioning because 1) not all sections benefit from hardware acceleration and 2) hardware accelerators may require *new* hardware data paths that are difficult to synthesize onto an FPGA.

The hardware–software partitioning problem is based on the ability to determine and isolate the part of a computation that could be realized into specialized hardware, for which we could improve performance, energy, size or any other composed measure. The general hardware–software partitioning problem has been shown to be NP-hard [1]. Efficient heuristic partitioning procedures have been studied (e.g., [8, 9]). In these works, the most difficult challenge is in choosing the appropriate granularity of representation in the computation graph; for example, a node can represent an instruction, a loop, a function call, or a module. This issue is addressed by system level design languages such as *SpecSyn* [5, 6]. The development frameworks for these languages deploy search techniques and implementation strategies based on the ability to represent and manipulate specific solution features at various levels of abstraction and where the user is always welcome to interact during the design process.

The subject of this paper is the domain-specific partitioning of the DFT where high-level algorithmic knowledge greatly simplifies the viable implementation space. There have been other scenario-specific partitioning methods developed for image processing (e.g., [16]), scalability of a design (e.g., [19]), reactive systems (e.g., [17]) and custom processor applications (e.g., [7]).

3 Background

We provide the necessary background on the 1D and 2D DFT, FFTs, the program generator Spiral, and the Virtex-II Pro platform.

DFT and FFT. The DFT is a matrix-vector multiplication $x \mapsto y = \mathrm{DFT}_n\, x$, where x, y are the input and output vector, respectively, of length n, and DFT_n is the $n \times n$ DFT matrix, given by $\mathrm{DFT}_n = [\omega_n^{k\ell}]_{0 \le k,\ell < n}$, $\omega_n = \exp(-2\pi j/n)$, $j = \sqrt{-1}$.

Algorithms for the DFT are sparse structured factorizations of the transform matrix [15]. For example, the Cooley-Tukey fast Fourier transform algorithm (FFT) can be written as

$$\mathrm{DFT}_{mn} \rightarrow (\mathrm{DFT}_m \otimes I_n) D_{m,n} (I_m \otimes \mathrm{DFT}_n) L_m^{mn}. \quad (1)$$

Here, I_m is the $m \times m$ identity matrix; $D_{m,n}$ is a diagonal matrix, and L_m^{mn} is the stride permutation matrix, both depending on m and n (see [15] for details). The Kronecker, or tensor product is defined as

$$A \otimes B = [a_{k,\ell} B]_{k,\ell} \text{ for } A = [a_{k,\ell}]. \quad (2)$$

For example,

$$I_m \otimes \mathrm{DFT}_n = \begin{bmatrix} \mathrm{DFT}_n & & \\ & \ddots & \\ & & \mathrm{DFT}_n \end{bmatrix}. \quad (3)$$

Similarly, the 2D DFT of an input of size $m \times n$ is given by a matrix $\mathrm{DFT}_{m \times n}$ that, using the row-column method, can be broken down as

$$\mathrm{DFT}_{m \times n} \rightarrow (\mathrm{DFT}_m \otimes I_n)(I_m \otimes \mathrm{DFT}_n). \quad (4)$$

Both (1) and (4) represent divide-and-conquer algorithms. For example, (1) asserts that $\mathrm{DFT}_{mn}\, x$ can be computed in four steps by first permuting x with L_m^{mn}, dividing the computation into m consecutive DFT_n subvectors of length n (see (3)), scaling with $D_{m,n}$, and finally dividing the computation into n DFT_m to subvectors at stride n.

In fixed-point implementations, scaling is often used to avoid overflow. This is formally captured by replacing above DFT_n with $\frac{1}{n}\mathrm{DFT}_n$ in the formulas above. In particular, every so-called *butterfly* DFT_2 is then succeeded by a scaling of $\frac{1}{2}$ (i.e., each vector element is shifted right by one bit).

Spiral. Spiral [14] is a program generation and optimization system for transforms. In Spiral, the formalism above is called SPL (signal processing language); a decomposition like (1) is called a *rule*. For a given transform, Spiral recursively applies these rules until all transforms have reached a pre-defined basic problem size (often 2). These rules are then compiled to generate one algorithm represented as a matrix formula. There are many formulas for each transform due to the choices of expansion. For example, in (1) different factorizations can be chosen (i.e., different m and n). The formula is then structurally optimized using a rewriting system, which performs formula level vectorization, parallelization, and loop optimizations, if needed. Finally, the resulting formula is compiled into actual C code or Verilog. The *performance* of the implementation (e.g., runtime, power, energy, error) is measured or estimated and fed back into a search engine, which decides how to modify the algorithm, using a dynamic programming search. Eventually, this feedback loop terminates and outputs the *best* implementation found. The entire process is depicted in Figure 1.

Spiral takes a similar approach to generate hardware implementations of transforms, currently restricted to the DFT [12]. In this case a model is used to aid the feedback driven optimization [11].

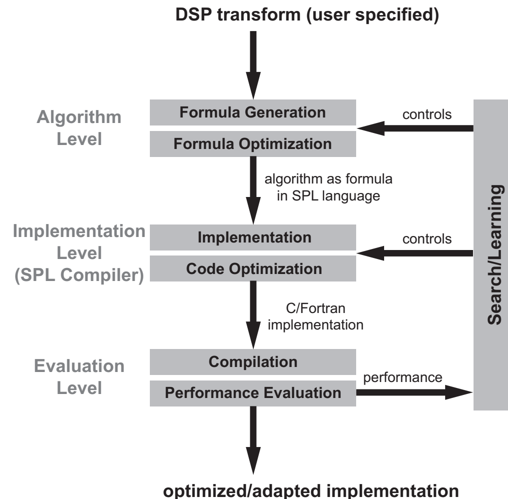


Figure 1. The code generator Spiral

Platform: Virtex-II Pro. We demonstrate and evaluate our approach by generating hardware-software partitioned implementations of the DFT for the Xilinx XUPV2P development board, which contains the Xilinx Virtex-II Pro XC2VP30 FPGA. The software portion of the partitioned implementation is executed on one of the two PowerPC405 processor cores embedded in the FPGA; the hardware data path of the partitioned implementation is mapped onto the FPGA's reconfigurable fabric. The hardware and software communicate through the DSOCM (data-side onchip memory) interface in the Virtex-II Pro architecture. A block diagram of this arrangement is shown in Figure 2. We briefly elaborate on the relevant platform parameters.

The embedded PowerPC 405 processor is a 300MHz in-order pipelined processor supporting integer operations only. The processor can access 512 MByte of DRAM on the XUPV2P board with a bandwidth of approximately 80MByte/second. Each PowerPC processor is also served by separate 1-cycle 16-KByte instruction and data caches. We run embedded Linux 2.4.27 on the PowerPC with the peripherals necessary to enable remote telnet through Ethernet (i.e., *openssh*). In this setup, the main Spiral engine is running on a separate workstation host. Executables for the PowerPC processor are generated, compiled, and linked on the remote host and communicated to the XUP board via Ethernet.

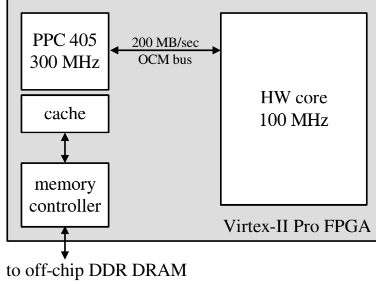


Figure 2. Virtex II Pro: hardware–software platform for experiments.

The Virtex-II Pro XC2VP30 FPGA has 13,969 reconfigurable slices, 136 18-bit hard multipliers and 136 2KByte BlockSelect RAMs (BRAMs). The DSOCM interface in the Virtex-II Pro architecture allows the PowerPC processor to memory-map up to 16 BRAMs in the fabric with 158 MByte/sec and 198 MByte/sec in read and write bandwidth. The reconfigurable fabric is clocked at the frequency of the DSOCM interface (100MHz). All of our FPGA configurations are created using Xilinx EDK tools.

4 Automatic Partitioning of Transforms

In this section, we explain the automatic partitioning of the DFT for a given set of hardware accelerators. Furthermore, we explain how we map the (partitioned) formula into inter-operating hardware and software. In this paper, we restrict the discussion to the 1D and 2D DFT, but the methodology is applicable to the domain of linear transforms.

To generate partitioned implementations for a given transform, the Spiral system searches over a space of candidate algorithms that compute the transform. Different algorithms lead to different hardware–software partition boundaries and thus different final performance.

4.1 Partitioning: Idea

Fast divide-and-conquer algorithms for the DFT are equivalent to the recursive application of breakdown rules like (1) and (4) (for 1D and 2D DFTs, respectively). Both rules compute multiple smaller 1D DFTs recursively and combine their results. The structure of these breakdown rules directly suggests a good partitioning strategy, namely to compute the subproblems DFT_m and DFT_n in hardware, and to let the software orchestrate the conquer step. Depending on the size of m and n, DFT_m and DFT_n may need to be further factorized to use smaller DFT hardware kernels that can fit within the available FPGA logic resources.

In all of the factorizations, the DFT kernels occur in the context of a tensor product with an identity matrix. In particular, $I_m \otimes \mathrm{DFT}_n$ (see (3)) indicates m data parallel applications of DFT_n to a vector of length mn. In hardware, for a wide vector that is presented sequentially as a stream of data flits, we can instantiate a single pipelined implementation of DFT_n and reuse it m times for the complete vector. Using the known identity

$$\mathrm{DFT}_m \otimes I_n = L^{mn}_m (I_n \otimes \mathrm{DFT}_m) L^{mn}_n, \qquad (5)$$

we can convert $\mathrm{DFT}_m \otimes I_n$ into the same form plus two data permutations to be handled in software; these permutations do not allocate extra memory space because they manage data movement through the temporary buffers used by the hardware cores.

The Spiral hardware core generator currently only generates DFT cores of two-power problem sizes. However, using (1), we can decompose the problem for other sizes into a two-power portion and a non-two-power portion. Then, the hardware can accelerate the former, while the latter portion must be implemented in software. We will present experimental results for this type of problem.

To summarize, based on an analysis of the DFT formula structure, we can restrict our choices of hardware cores to different sized streaming pipelined DFT kernel implementations. In this restricted partitioning problem, the key degree of design freedom is in choosing the appropriate set of DFT kernel sizes to be synthesized into the available FPGA resources in order to maximize the desired metric. For most performance metrics, this means choosing the set of kernel sizes that maximizes the computation off-loaded from the PowerPC processor into the FPGA fabric.

4.2 Partitioning: Formal method

We use a rewriting system [3] that constructs partitioned FFTs for a given DFT. The partitioning algorithm is encoded as a set of *rewrite rules* that operate in tandem with breakdown rules such as (1). The input to the rewrite system is a transform tagged "to be partitioned", and a list of available hardware cores. The output is one or a set of partitioned formulas.

Tags. Throughout the rewriting process, partitioning information is propagated using *tags*. We introduce two tags:

$$\underbrace{A}_{\text{partition}} \quad \text{and} \quad \underbrace{A}_{\text{HW}}.$$

A formula A tagged with *partition* needs to be partitioned; however, its partitioning is not yet known. For a formula A that is not tagged no further rewriting needs to be done and it is implemented in software. For a formula A tagged with *HW*, this decision has been made and A is mapped to hardware cores.

Rewriting rules. In order to extract hardware-mappable sub-formulas, we utilize a set of rewriting rules summarized

Table 1. Rewriting rules for hardware–software partitioning.

$$\underbrace{A}_{\text{partition}} \to A \quad \text{Software only} \qquad (6)$$

$$\underbrace{A}_{\text{partition}} \to \underbrace{BC}_{\text{partition}} \quad \text{Break down} \qquad (7)$$

$$\underbrace{A\,B}_{\text{partition}} \to \underbrace{A}_{\text{partition}}\,\underbrace{B}_{\text{partition}} \quad \text{Distribution} \qquad (8)$$

$$\underbrace{\mathrm{DFT}_m \otimes I_n}_{\text{partition}} \to L^{mn}_m (\underbrace{I_n \otimes \mathrm{DFT}_m}_{\text{HW}}) L^{mn}_n \quad \text{If } \mathrm{DFT}_m \text{ in HW} \qquad (9)$$

$$\underbrace{\mathrm{DFT}_m \otimes I_n}_{\text{partition}} \to \underbrace{\mathrm{DFT}_m}_{\text{partition}} \otimes I_n \quad \text{If } \mathrm{DFT}_m \text{ not in HW} \qquad (10)$$

$$\underbrace{I_m \otimes DFT_n}_{\text{partition}} \to \underbrace{I_m \otimes \mathrm{DFT}_n}_{\text{HW}} \quad \text{If } \mathrm{DFT}_n \text{ in HW} \qquad (11)$$

$$\underbrace{I_m \otimes DFT_n}_{\text{partition}} \to I_m \otimes \underbrace{\mathrm{DFT}_n}_{\text{partition}} \quad \text{If } \mathrm{DFT}_n \text{ not in HW} \qquad (12)$$

in Table 1. For example, rule (8) distributes the partition tag across factors of a product; rule (9) implements the identity (5), effectively mapping $\mathrm{DFT}_m \otimes I_n$ to the form (3) compatible with streaming pipelined DFT cores. Note that any rule where the left side is untagged (as (1) and (4)) can be applied "inside" a tagged expression, leaving the tag unchanged.

Base cases. When the desired set of hardware DFT kernel sizes is known, those DFT kernels are encoded as *base cases* which cannot be broken down further. The base case rule (11) matches subformulas of the form $I_m \otimes \mathrm{DFT}_n$ where DFT_n is a hardware supported DFT kernel size. This is because a streaming pipelined DFT core efficiently handles any number of consecutive, data-parallel applications to an input stream.

Lastly, the software termination rule (6) is the rule indicating that everything that cannot be further rewritten is implemented in software. This rule is used when no other rules apply, ensuring that a subformula is mapped to hardware whenever beneficial (e.g., determined by the feedback system).

Rewriting process. The input to the rewriting system is a tagged transform, for instance

$$\underbrace{\mathrm{DFT}_n}_{\text{partition}} \quad \text{or} \quad \underbrace{\mathrm{DFT}_{m\times n}}_{\text{partition}}.$$

In a final partitioned formula, the tag "partition" has been removed, and all subformulas are either untagged or tagged as "HW".

The rewriting system contains three rule sets: 1) *breakdown rules* (1) and (4), 2) *partitioning rules* (8)–(12), and 3) the *cleanup rule* (6).

Example. We show a partitioning example of a DFT_{mnr}, where r is a small odd prime number (for instance 3 or 5), and m and n are two-powers (e.g., $m = 2^\ell$). We assume the availability of two streaming hardware cores, $I_k \otimes \mathrm{DFT}_m$, and $I_k \otimes \mathrm{DFT}_n$. The input to our rewriting system is the tagged problem specification,

$$\underbrace{\mathrm{DFT}_{mnr}}_{\text{partition}}.$$

The rewriting system first applies (1), producing subproblems of size mn and r. Then, the partitioning rule set is applied. In the next breakdown step, the system applies (1), reducing the problem of size mn into m and n. The partitioning rule set is applied again, and finally, the cleanup rule (6) is applied, leading to the partitioned formula (13) (seen in Figure 3). The search space of all partitioned formulas is obtained by enumerating all possible choices of parameterizations whenever (1) is applied.

Next we explain how a partitioned formula like (13) is mapped to hardware and software.

4.3 Software and Hardware Generation

Mapping a partitioned formula like (13) to an FPGA-accelerated program for the PowerPC requires three steps: 1) generating software for the untagged parts, 2) generating the required hardware designs, and 3) interfacing with the accelerators from within the generated software.

Software. We use the standard fixed-point code generation process of Spiral [4,14] to generate software implementations for the untagged parts of a formula. The hardware portions of the formula are dispatched to the DFT cores in the FPGA fabric via a specialized hardware function-call interface.

Hardware. We use Spiral's DFT IP core generator [12,13] to generate hardware cores implementing a streaming version of $I_m \otimes \mathrm{DFT}_n$ for a two-power n. For a DFT core of a given size, it is possible to compute the DFT of a smaller two-power size by interleaving the smaller vector's elements with zero elements (i.e., up-sampling in time); the results are in the leading elements of the output vector (i.e., periodic in frequency). In other words, each instantiated DFT_n core actually provides a set of *virtual cores* for the smaller DFTs of sizes $n/2, n/4$, etc. The disadvantage of using the virtual cores is that they have the same latency as computing the larger native-sized DFTs. Fortunately, our primary performance concern is in the throughput of repeated DFT computations. By overlapping multiple DFT

$$\underbrace{\mathrm{DFT}_{mnr}}_{\text{partition}} \rightarrow \big(\mathrm{DFT}_r \otimes I_{mn}\big) D_{r,mn} \big\{\big[I_r \otimes \big(L_m^{mn} \underbrace{(I_n \otimes \mathrm{DFT}_m)}_{\text{HW}} L_n^{mn}\big)\big] D_{m,n} \underbrace{(I_m \otimes \mathrm{DFT}_n)}_{\text{HW}} L_m^{mn}\big\} L_r^{mnr} \qquad (13)$$

Figure 3. DFT_{mnr} **partitioned for streaming hardware cores** $I_k \otimes \mathrm{DFT}_m$ **and** $I_k \otimes \mathrm{DFT}_n$**.** m **and** n **are two-powers and** r **is a small prime.**

calculations in flight, we can hide the effect of the extra latency completely and see no throughput penalty when using a DFT_n core to compute problem sizes down to $n/4$.

Interface. Figure 4 shows the architecture of two hardware cores accelerating the PowerPC. The PowerPC is in control, processing data residing in the main memory. It communicates with the hardware cores via the DSOCM bus, which is used to send and receive data, as well as control information. To initiate a DFT calculation, the PowerPC writes the DFT input data into BRAM and sets a bit in the control registry to select the desired core size (virtual or real) and initiate processing. The selected DFT core begins streaming the data from the input BRAM, through the streaming DFT pipeline and returns the processed data into the output BRAM. Once all data is processed, the DFT core sets a ready status bit in the control registry polled by the PowerPC. The PowerPC begins retrieving the data from the return BRAM buffer.

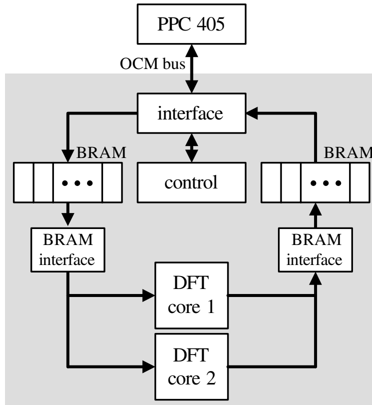


Figure 4. Architecture of the generated hardware DFT IP cores.

In order to obtain full streaming utilization, the PowerPC loads data into the next buffer and retrieves the previous result while the hardware core is processing the current data set. The BRAM interface implements a ring buffer for 4 sets of data, allowing 4 DFT calculations to be in flight concurrently to hide latency.

5 Optimizing an Entire Library

So far, we have shown, in a forward design problem, how to generate a partitioned implementation for a given DFT and a given set of hardware cores. Next we discuss the inverse design problem of determining the optimal set of hardware cores based on a performance metric and resource constraints. The overall optimized design generation problem is solved by first solving the inverse design problem and then the resulting forward design problem.

The straightforward approach to an inverse design problem is to solve many forward design problems and search for the best solution. This approach has been used in Spiral to generate optimized software-only implementations. It is possible since software code generation and evaluation are fast enough to enable a feedback-driven optimization loop. However, it is impractical to let Spiral generate and try out all sets of hardware cores admissible under a specified area/power budget since synthesis takes on the order of hours for each trial.

Fortunately, the timing behavior of DFT cores is extremely predictable, resembling a delay buffer of a precisely known delay (a function of the size of the DFT). For performance evaluation in Spiral's search-based feedback loop, a special version of the DFT software is used on the PowerPC processor. The timing software faithfully performs all software instructions including loads and stores to memory (through caches) and the BRAM-based hardware interface. However, the expected computation delay of the DFT core invocations is emulated in software using cycle counters. This enables us to explore the performance space of different DFT core sizes without actually synthesizing or downloading any actual hardware cores. Using this approximation, we can fully explore the available hardware core choices in typically a few hours (up to 30 minutes per hardware-core configuration).

We have empirically verified that this modeling approach is sufficiently accurate to support meaningful design space exploration (e.g., Figure 5). The model predicts execution very well especially for in-cache DFT sizes less than 512. For the larger, out-of-cache sizes, we have seen a maximum error of 15%, but the model accurately estimates the performance/energy trends.

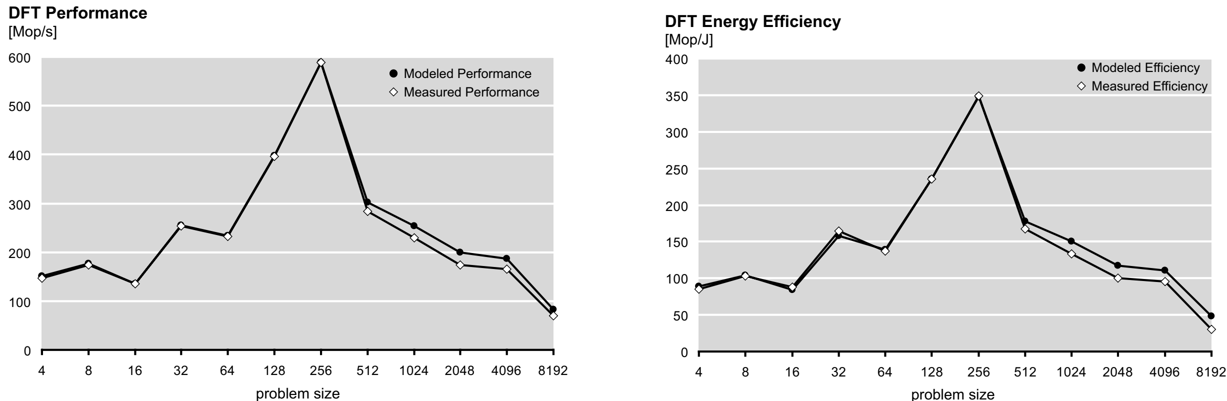


Figure 5. Quality of our performance and energy efficiency model. The closer the lines the better.

5.1 Pruning The Search Space

For a given transform size, there are exponentially many formulas. In addition, allowing any combination of hardware cores leads to a combinatorial explosion. To make our approach feasible we need to prune this search space in addition to speeding up evaluation. We employ two pruning ideas.

Choice of cores. Due to the overlap in functionality, latency, and energy consumption arising from virtual cores, some combinations of DFT kernel sizes are not beneficial for joint mapping onto an FPGA. Specifically, the possibility of using the virtual cores associated with each real DFT core dramatically reduces the number of sensible hardware configurations. This claim is made specific in Section 6.2, where a performance analysis of virtual cores is given.

Dynamic programming. We reduce the algorithm search space by taking advantage of the recursive nature of transform algorithms. Experiments show that *dynamic programming* (DP) is a viable method to optimize these algorithms. The DP methodology is based on the assumption that the performance of a subproblem does not depend on its context. For instance, this means that the performance of DFT_n is assumed the same in the context of $I_m \otimes \mathrm{DFT}_n$ or $\mathrm{DFT}_n \otimes I_m$.

To find the best solution for a given problem (say, DFT_n), DP tries all applicable breakdown rules (in this paper, Cooley-Tukey FFT (4) only). For each applicable rule, DP first recursively finds the best solutions for its subproblems and all rule parameterization (in our example, all DFT_k and DFT_m with $km = n$). It then finds the best solution for the original problem (DFT_n) by evaluating all parameterizations of the breakdown rule and plugging in the best subproblem solutions that it has already found. All problems are cached after they are evaluated, accelerating the search for any future overlapping problems.

6 Experimental Results

In this section, we evaluate our approach on the Xilinx Virtex-II Pro FPGA with embedded PowerPC processor by generating FPGA-accelerated DFT libraries optimized for both performance and energy efficiency. After specifying the experimental setup, we first briefly consider the necessary library components: Spiral generated DFT software libraries, Spiral generated hardware cores, and virtual cores. Then, we evaluate a specific example of FPGA-accelerated DFT libraries and discuss the tradeoffs between using one core versus two cores. Finally, we evaluate in detail DFT libraries accelerated with two hardware cores; we discuss the tradeoffs of the hardware core choice in the performance/area/power space. Again we stress that in all accelerated libraries both the software components and the hardware cores are Spiral-generated ("push-button").

In this work, we present DFT implementations in fixed-point precision (16 bits with 14 bits fractions). We implement scaled DFTs to avoid overflow. We do not address the minimization of the error by using different algorithms in finite precision; however, in Section 6.4, we will experimentally address the issues of finite precision errors arising from different algorithms and partitioning.

Performance metrics. We assume an operation count of $5n\log_2 n$ for the 1D DFT_n and $5mn\log_2 mn$ for the 2D $\mathrm{DFT}_{m\times n}$. We measure *runtime performance* in pseudo Mop/s (mega-operations per second), computed as operations [op]/runtime [μs], and *energy efficiency* in pseudo Mop/J (mega-operations per Joule), computed as operations [op]/energy [μJ]. These metrics are scaled inverses of runtime and energy, respectively, and thus preserve runtime and energy relations. In Pareto plots we use *normalized runtime* [ns/op] (inverse of runtime performance) vs. power [W] and *area* [slices].

Physical measurements. To obtain the required measurement resolution, we perform the same computation

Table 2. Performance [Mop/s], for problems of size 16—1024 on hardware cores of size 32—1024.

	problem size						
Core size	1024	512	256	128	64	32	16
32						460	313
64					603	460	296
128				739	603	439	230
256			866	739	585	313	135
512		980	866	726	396	178	74
1024	1092	980	858	476	220	96	

multiple times. We measure runtime using the PowerPC's cycle counter, and the power supplied to the board is determined by measuring the supply current (at 5V) using an Agilent 34401A digital multi meter. Both measurements are acquired automatically and fed back to the Spiral search engine, closing the performance tuning loop (see Fig. 1). We compute the energy as the product of measured runtime and power.

6.1 Performance Evaluation of Software and Hardware

Software. Spiral-generated floating-point and fixed-point software code is competitive with the best available DFT software implementations across many platforms [2, 14]. The 300 MHz PowerPC typically achieve on average 100 pseudo Mop/s on in-cache DFT calculations.[1]

Hardware cores. We have shown Spiral-generated DFT cores are competitive in performance and size with the Xilinx LogiCore DFT library for DFT sizes up to 1024 [12,13]. Table 2 reports the throughput performance (Mops/s) of the streaming pipelined DFT cores, including when the cores are applied to smaller problem sizes in the virtual mode. We observe that in each case for $n/2$ and $n/4$ there is virtually no performance penalty compared to the real core of that size. The reason is that in these cases the performance bottleneck is the data copy bandwidth of the DSOCM bus. The extra latency incurred by virtual cores is fully hidden. As a consequence, any two DFT hardware cores of size n and m ($n > m$) used for acceleration in larger problems (i.e., when used in throughput mode) should be chosen to satisfy $n/m > 4$.

[1]Note that the operation count does not comprise loads/stores, index computations, loop bound computations and scaling. Thus, we use an under-estimation of the number of operations actually issued.

6.2 Accelerating Software with Hardware Cores

We now consider the first example of an FPGA-accelerated DFT library generated by Spiral. We choose two hardware cores: DFT_{64} and DFT_{512}. Figure 6 shows the performance impact of using one or both hardware cores compared to a generated software DFT library.

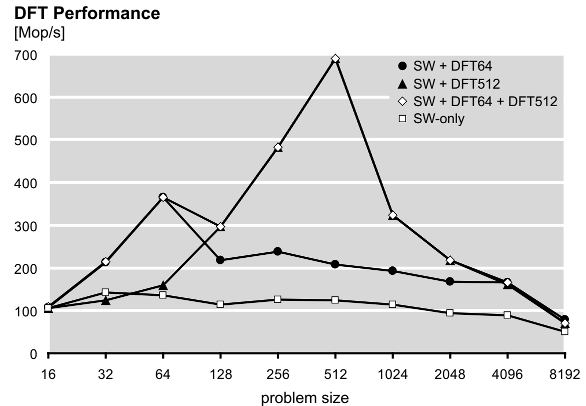


Figure 6. Performance of Spiral-generated 1D DFT software accelerated by one or two hardware cores DFT_{64} and DFT_{512}. Higher is better.

Single core. Accelerating the software implementation with a single core shows the same characteristic trend for each of the choices. For very small sizes, employing the hardware core does not speed up computation due to overhead and, there, the software-only library is faster. This is the case for $n = 16$ for the DFT_{64} core and for $n \leq 64$ for the DFT_{512} core.

When the problem size is between this "break-even" point and the size of the hardware core, we observe a ramp-up in which the highest performance is reached at the core's native size, for a speed-up of 2.6 times for $n = 64$ and 5.6 times for $n = 512$. In this region all computations are done in hardware (via real or virtual cores), and software is only used to route data in and out of the cores.

We observe a drop in performance for the first problem size larger than the core size ($n = 128$ for DFT_{64}, and $n = 1024$ for DFT_{512}) to about two times the software performance. For this size and larger sizes a significant amount of the computation is done in software. Nevertheless, the hardware acceleration provides a speed-up of at least two times for large in-cache problem sizes. Once data does not fit into cache at $n = 8192$, memory bandwidth becomes the main bottleneck and practically reduces all possible speedups.

Two cores. Both single core configurations have weak spots: the DFT_{64} core speeds up small sizes but provides

only moderate speed-up for large sizes, and the DFT_{512} core provides high speed-up for medium and large sizes but cannot speed up small sizes. By employing a system with *both* hardware cores, we leverage the positive aspects of both cores. Figure 6 shows that a system with two cores achieves the maximum performance of both single core systems for each problem size.

6.3 Optimizing an Entire DFT Library

Due to the nature of the 1D and 2D DFT algorithms (1) and (4), we can speed up an entire library of two-power and non-two-power sizes using two (or any other number of) hardware cores. However, there are multiple possible choices for the sizes of the hardware cores, which give different performance characteristics and thus tradeoffs for the entire DFT library.

We investigate these tradeoffs with respect to both performance and energy efficiency. To compare the different libraries (i.e., different choices of cores) across all sizes, we use *average normalized runtime* [ns/op] and *average power* as metrics. This choice weighs all problem sizes equally.

Choice of hardware cores: Details. Figure 7 displays the performance and energy efficiency behavior of a two-power 1D and 2D DFT library for different choices of two hardware cores.

Figure 7 (a) and (b) show that for 1D DFT, there is clearly a best configuration for each fixed problem size. However, across all problem sizes, each configuration is the best at least twice. Choosing the best choice across a library is not straightforward and depends on the targeted application context.

The shape of the energy efficiency plot is similar to the performance plot, with the notable difference that for small problem sizes software is the most energy-efficient choice.

Figure 7 (c) and (d) show the same evaluation for 2D DFTs. The 2D $\mathrm{DFT}_{m\times n}$ has the same memory footprint as a 1D DFT_{mn}. All sizes larger than or equal to 64×128 do not fit into cache, which leads to a performance degradation for all choices of cores. As in the 1D case, smaller problem sizes are accelerated by hardware, but software-only is more energy efficient. The latter effect is more pronounced than in the 1D case. Thus, the possible speed-up through FPGA acceleration is smaller and the variance across different core choices is less pronounced than in the 1D case.

Using two-power hardware cores we can also accelerate libraries of non-two-power sizes by computing two-power factors in hardware and the remainder in software. Figure 8 displays the performance for non-two-power problems DFT_n with $n = 3 \cdot 2^k$ and $n = 5 \cdot 2^k$. In this situation, a significant amount of computation—the radix-3 and radix-5 kernels—is done in software. Nevertheless, using two hardware cores we obtain up to 2.5 times speed-up over a software-only implementation.

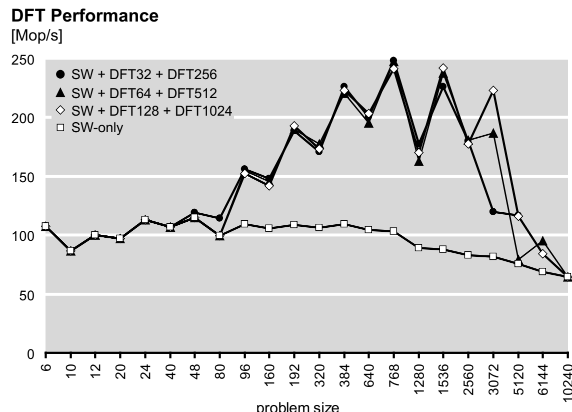


Figure 8. Performance of Spiral-generated 1D DFT_n with $n = 3 \cdot 2^k$ and $n = 5 \cdot 2^k$ accelerated by different two-core configurations. Higher is better.

Trade-offs and Pareto analysis. Choosing different hardware core pairs allows an exploration of trade-offs between performance and power or FPGA area across a whole DFT library. For instance, one can choose the set of hardware cores that yields the fastest DFT library (averaged over all sizes) for a given slice or power budget.

Figure 9 shows that there is indeed room to trade runtime for area or power. We assume a 1D DFT library for two-power sizes $n = 64, \ldots, 2048$. Each point represents the average performance/power/area values for the whole library for different pairs of hardware cores. The software-only configuration is marked with a different symbol. Note that in the power data we subtracted the XUP2VP development board's idle power consumption (3.8W).

Figure 9 (a) shows that there is a 4 times variation in both area consumption and normalized runtime across all possible configurations. Figure 9 (b) shows that there is also a 3 times variation in the power consumed by the DFT calculations. In other words, by allowing up to 3 times more power (or 4 times more area) to be consumed, one can speed up a whole library up to 4 times (averaged across the library). As there are many points between these extremes, we provide a fine-grain choice for adapting the performance and resource usage of a whole DFT library to application-specific needs.

6.4 Error Analysis

Finally, we show experiments detailing the error behavior of our generated hardware–software libraries. We follow the error models for fixed-point DFTs [10, 18], specialized to our case of 16-bit precision with 14 fraction bits. We compare the software-only library to the hardware-software

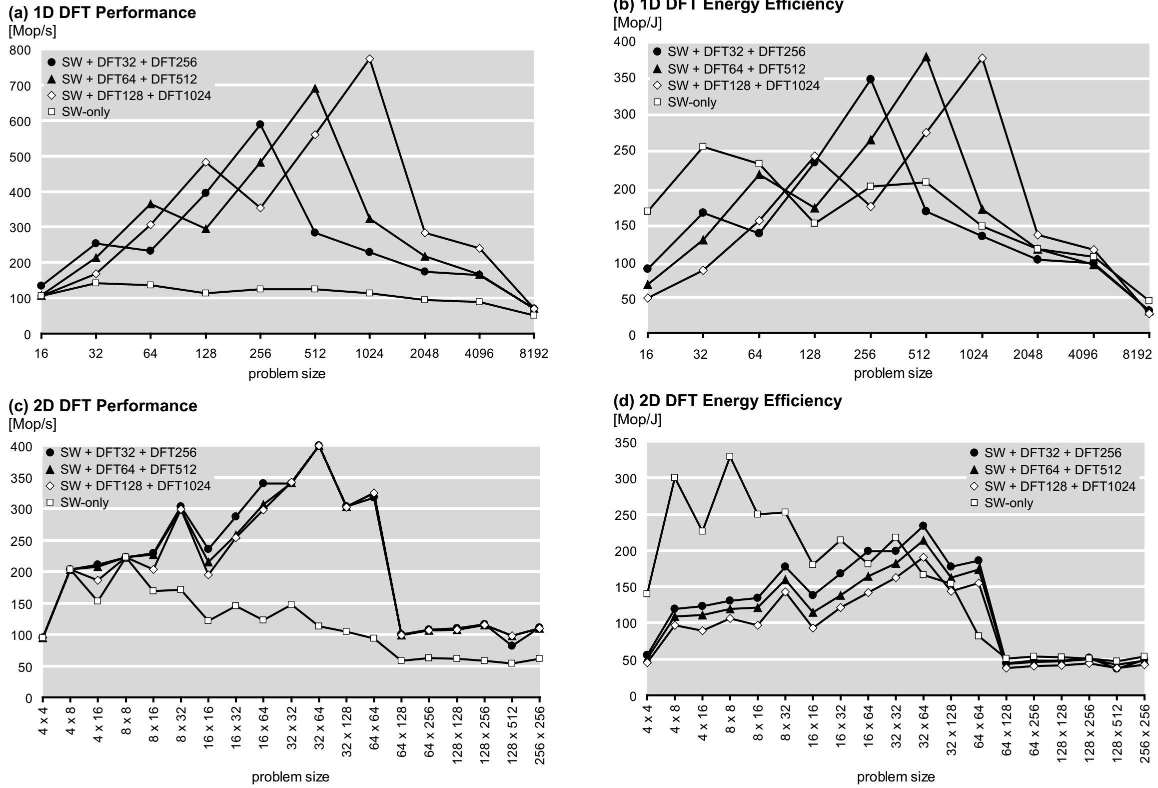


Figure 7. Performance and energy efficiency of 1D and 2D DFT software accelerated by different two-core configurations. Higher is better in all plots.

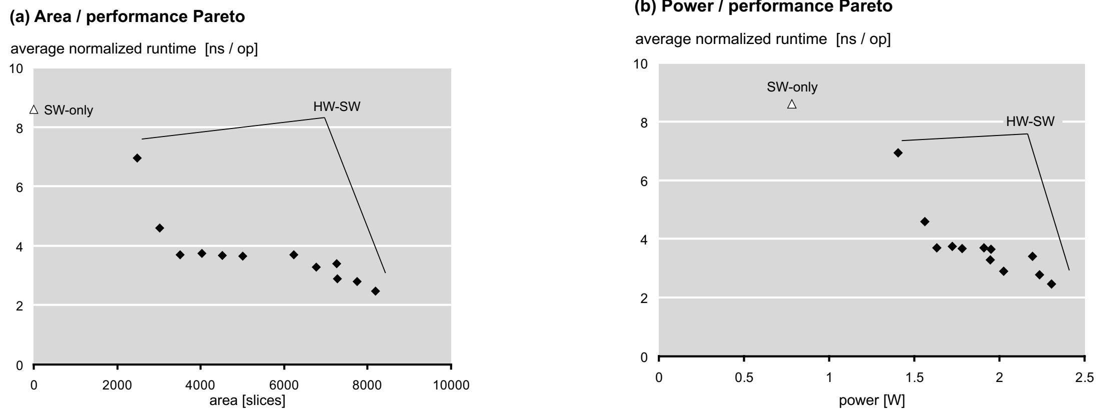


Figure 9. 1D DFT Area/performance and power/performance trade-offs (one point is a normalized performance for a whole DFT library and a specific set of hardware cores). Points closer to the origin are better in both plots.

libraries considered in the performance evaluation in Figure 7.

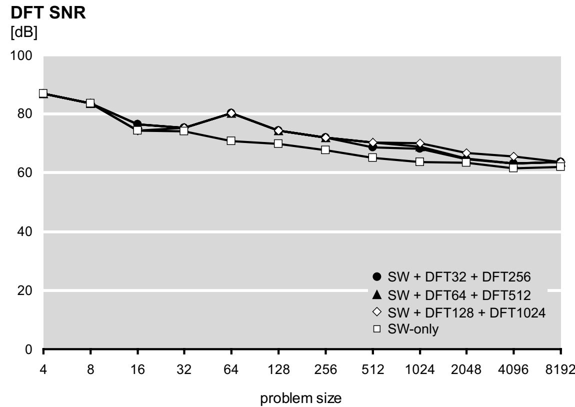


Figure 10. Signal-to-noise ratio (SNR) of Spiral-generated scaled fixed-point DFT implementations.

In Figure 10, we evaluate the signal-to-noise ratio (SNR). We see that the software library and hardware–software libraries have very similar SNR and hardware acceleration does not introduce further error.

Figure 11 shows the maximum absolute error and the average absolute error. Figure 11 (a) shows that all considered libraries behave similarly and stay well below the theoretical upper bound for the maximum absolute error of a fixed-point DFT, given in [18]. We see up to 20% variation in the error. This could be exploited to obtain low-error implementations by using error as search metric in Spiral's feedback loop. Figure 11 (b) shows that the average absolute error saturates at about $2^{-15} \approx 3 \cdot 10^{-5}$, which is the system precision.

7 Conclusion

Architectures with tightly integrated FPGAs and general purpose processors are starting to play an important role in both embedded and high performance computing settings. The tight integration makes it possible to offload fine and coarse grain functionalities from processors to the FPGA fabric, combining the strengths of both components.

In this paper we introduce an extension to the program and hardware design generation system Spiral, that automatically partitions DFT kernels across software and hardware, and generates both components. In addition, our extension finds—under user-supplied resource constraints and performance metrics—a partition choice that optimizes an entire library, not a single problem instance.

In our experiments on a Xilinx Virtex-II Pro, the automatically partitioned and generated FPGA-accelerated library has between 2 and 7.5 times higher performance and up to 2.5 times better energy efficiency than the software-only version.

Acknowledgment

This work was supported by DARPA through the Department of Interior grant NBCH1050009 and by NSF through awards 0234293 and 0325687.

References

[1] P. Arató, Z. Mann, and A. Orbán. Algorithmic aspects of hardware/software partitioning. *ACM Trans. Des. Autom. Electron. Syst.*, 10(1):136–156, 2005.

[2] P. D'Alberto, F. Franchetti, and M. Püschel. Performance/energy optimization of DSP transforms on the XScale processor. In *Proceeding of the 2007 International Conference on High Performance Embedded Architectures and Compilers*, Lecture Notes in Computer Science, Ghent, Belgium, Jan 2007. Springer.

[3] N. Dershowitz and D. A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 9, pages 535–610. Elsevier, 2001.

[4] F. Franchetti, Y. Voronenko, and M. Püschel. Formal loop merging for signal transforms. In *Proc. of Programming Language Design and Implementation*, Chicago, Jun. 2005.

[5] D. Gajski, F. Vahid, and S. Narayan. SpecSyn: an environment supporting the specify-explore-refineparadigm for hardware/software system design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(1):84–100, Mar 1998.

[6] D. Gajski, F. Vahid, S. Narayan, and J. Gong. SpecSyn: an environment supporting the specify-explore-refine paradigm for hardware/software system design. pages 108–124, 2002.

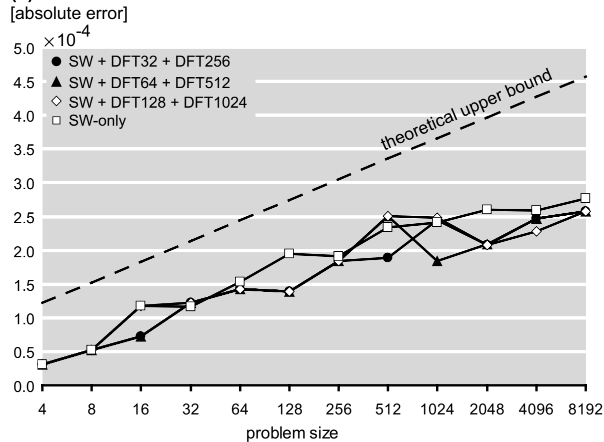
[7] J. R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 12, Washington, DC, USA, 1997. IEEE Computer Society.

[8] A. Kalavade and E. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *CODES '94: Proceedings of the 3rd international workshop on Hardware/software co-design*, pages 42–48, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[9] B. Knerr, M. Holzer, and M. Rupp. Improvements of the GCLP algorithm for SW/HW partitioning of task graphs. In *Prooceedings of the Fourth IASTED International Conference on Circuits, Signals, and Systems*, pages 107–113, San Francisco, CA, USA, Nov. 2006.

[10] W. Knight and R. Kaiser. A simple fixed-point error bound for the fast fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 27(6):615–620, Dec 1979.

[11] P. A. Milder, M. Ahmad, J. C. Hoe, and M. Püschel. Fast and accurate resource estimation of automatically generated custom DFT IP cores. In *Proc. FPGA*, 2006.



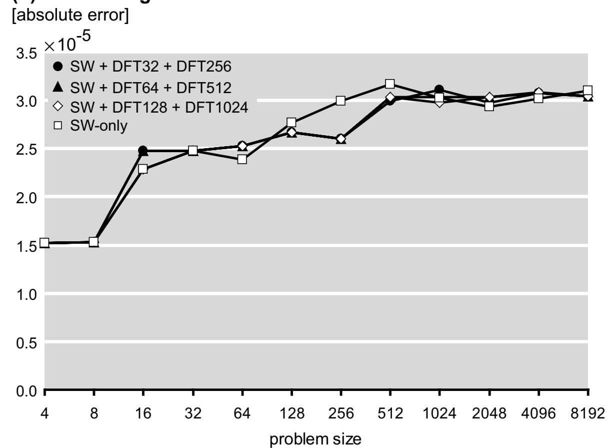


Figure 11. Error behavior of Spiral-generated scaled fixed-point DFT implementations.

[12] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel. Discrete Fourier transform compiler: From mathematical representation to efficient hardware. Technical Report CSSI 07-01, Center for Silicon System Implementation, Carnegie Mellon University, 2007.

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

[14] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2), 2005.

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

[16] C. Vicente-Chicote, A. Toledo, and P. Snchez-Palma. Image processing application development: From rapid prototyping to sw/hw co-simulation and automated code generation. In *Pattern Recognition and Image Analysis*, volume 3522 of *Lecture Notes in Computer Science*, pages 659–666. Springer Berlin / Heidelberg, 2005.

[17] K. Weis;, T. Steckstor, G. Koch, and W. Rosenstiel. Exploiting FPGA-features during the emulation of a fast reactive embedded system. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 235–242, New York, NY, USA, 1999. ACM Press.

[18] P. Welch. A fixed-point fast fourier transform error analysis. *IEEE Transactions on Audio and Electroacoustics*, 17(2):151–157, Jun 1969.

[19] C. Zhang, Y. Long, and F. Kurdahi. A scalable embedded JPEG2000 architecture. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 3553 of *Lecture Notes in Computer Science*, pages 334–343. Springer Berlin / Heidelberg, 2005.