Fast Automatic Generation of DSP Algorithms

Markus Püschel[1], Bryan Singer[2], Manuela Veloso[2], and José M.F. Moura[1]

[1] Carnegie Mellon University, Pittsburgh
Department of Electrical and Computer Engineering
{moura,pueschel}@ece.cmu.edu
[2] Department of Computer Science
{bsinger,veloso}@cs.cmu.edu

Abstract. SPIRAL is a generator of optimized, platform-adapted libraries for digital signal processing algorithms. SPIRAL's strategy translates the implementation task into a search in an expanded space of alternatives. These result from the many degrees of freedom in the DSP algorithm itself and in the various coding choices. This paper describes the framework to represent and generate efficiently these alternatives: the formula generator module in SPIRAL. We also address the search module that works in tandem with the formula generator in a feedback loop to find optimal implementations. These modules are implemented using the computer algebra system GAP/AREP.

1 Introduction

SPIRAL, [1], is a system that generates libraries for digital signal processing (DSP) algorithms. The libraries are generated at installation time and they are optimized with respect to the given computing platform. When the system is upgraded or replaced, SPIRAL can regenerate and thus readapt the implementations. SPIRAL currently focuses on DSP transforms including the discrete trigonometric transforms, the discrete Fourier transform, and several others. Other approaches to similar problems include for DSP transforms [2] and for other linear algebra algorithms [3–6].

SPIRAL generates a platform-adapted implementation by searching in a large space of alternatives. This space combines the many degrees of freedom associated with the transform and the coding options.

The architecture of SPIRAL is displayed in Figure 1. The DSP transform specified by the user is input to a formula generator block that generates one out of many possible formulas. These formulas are all in a sense equivalent: barring numerical errors, they all compute the given transform. In addition, they all have basically the same number of floating point operations. What distinguishes them is the data flow pattern during computation, which causes a wide range of actual runtimes. The output of the formula generator is a formula given as a program in a SPIRAL proprietary language called SPL (signal processing language). The SPL program is input to the SPIRAL-specific formula translator block that compiles it into a C or Fortran program [7]. This program, in turn, is compiled by a standard C or Fortran compiler. The runtime of the resulting code is then fed back through a search module. The search module controls the generation of the next formulas to be tested using search and learning techniques (Section 4). Iteration of this loop yields a platform-adapted implementation.

This paper focuses on the formula generator and its interplay with the search module. It explains the underlying mathematical framework (Section 2) and its implementation (Section 3) using the computer algebra system GAP/AREP [8].

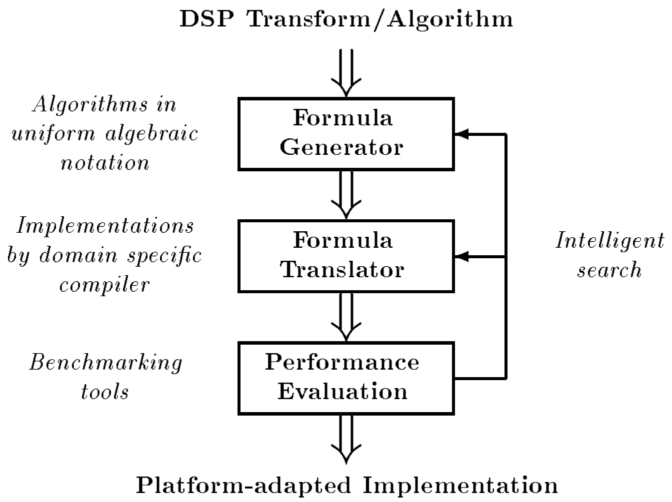

Fig. 1. The architecture of SPIRAL.

2 DSP Transforms and Algorithms

In this section we introduce the framework used by SPIRAL to describe linear DSP (digital signal processing) transforms and their fast algorithms. A similar approach has been used in [9] for the special case of FFT algorithms. We start with an introductory example.

2.1 Example: DFT, Size 4

The DFT (discrete Fourier transform) of size 4 is given by the following matrix DFT_4, which is then factored as a product of sparse structured matrices.

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

This factorization is an example of a fast algorithm for DFT_4. Using the Kronecker (or tensor) product of matrices, $\otimes$, and introducing symbols L^4_2 for the permutation matrix (right-most matrix), and $\mathrm{T}^4_2 = \mathrm{diag}(1,1,1,i)$, the algorithm can be written in the very concise form

$$\mathrm{DFT}_4 = (\mathrm{DFT}_2 \otimes \mathrm{I}_2) \cdot \mathrm{T}^4_2 \cdot (\mathrm{I}_2 \otimes \mathrm{DFT}_2) \cdot \mathrm{L}^4_2 \,. \tag{1}$$

The last expression is an instantiation of the celebrated Cooley-Tukey algorithm [10], also referred to as the fast Fourier transform (FFT).

2.2 Transforms, Rules, Ruletrees, and Formulas

Transforms: A (linear) DSP transform is a multiplication of a vector x (the sampled signal) by a certain $(n \times n)$-matrix M (the transform), $x \mapsto M \cdot x$. The transform is denoted by a symbol having the size n of the transform as a subscript. Fixing the parameter "size" determines a special instance of the transform, e.g., DFT_4 denotes a DFT of size 4. For arbitrary size n, the DFT is defined by

$$\mathrm{DFT}_n = [w_n^{k\ell} \mid k, \ell = 0..n-1],$$

where $w_n = e^{2\pi j/n}$ denotes an nth root of unity. In general, a transform can have other determining parameters rather than just the size. Transforms of interest include the discrete cosine transforms (DCT) of type II and type IV,

$$\begin{aligned}\mathrm{DCT}_n^{(\mathrm{II})} &= [\cos((\ell+1/2)k\pi/n) \mid k, \ell = 0 \ldots n-1], \text{ and}\\ \mathrm{DCT}_n^{(\mathrm{IV})} &= [\cos((k+1/2)(\ell+1/2)\pi/n) \mid k, \ell = 0 \ldots n-1],\end{aligned}$$

which are used in the current JPEG and MPEG standards, [11], respectively (given here in an unscaled version). Further examples with different areas of application are the other types of discrete cosine and sine transforms (DCTs and DSTs, type I – IV), the Walsh-Hadamard transform (WHT), the discrete Hartley transform (DHT), the Haar transform, the Zak transform, the Gabor transform, and the discrete wavelet transforms.

Breakdown Rules: All of the transforms mentioned above can be evaluated using $O(n \log n)$ arithmetic operations (compared to $O(n^2)$ operations required by a straightforward implementation). These algorithms are based on sparse structured factorizations of the transform matrix. For example, the Cooley-Tukey FFT is based on the factorization

$$\mathrm{DFT}_n = (\mathrm{DFT}_r \otimes \mathrm{I}_s) \cdot \mathrm{T}_s^n \cdot (\mathrm{I}_r \otimes \mathrm{DFT}_s) \cdot \mathrm{L}_r^n, \tag{2}$$

where $n = r \cdot s$, L_r^n is the stride permutation matrix, and T_s^n is the twiddle matrix, which is diagonal (see [12] for details). We call an equation like (2) a *breakdown rule*, or simply *rule*. A breakdown rule

- is an equation that factors a transform into a product of sparse structured matrices;
- may contain (possibly different) transforms of (usually) smaller size;
- the applicability of the rule depends on the parameters (e.g., size) of the transform.

Examples of breakdown rules for $\mathrm{DCT}^{(\mathrm{II})}$ and $\mathrm{DCT}^{(\mathrm{IV})}$ are

$$\begin{aligned}\mathrm{DCT}_n^{(\mathrm{II})} &= P_n \cdot (\mathrm{DCT}_{n/2}^{(\mathrm{II})} \oplus \mathrm{DCT}_{n/2}^{(\mathrm{IV})}) \cdot P_n' \cdot (\mathrm{I}_{n/2} \otimes \mathrm{DFT}_2) \cdot P_n'', \text{ and}\\ \mathrm{DCT}_n^{(\mathrm{IV})} &= S_n \cdot \mathrm{DCT}_n^{(\mathrm{II})} \cdot D_n,\end{aligned}$$

where P_n, P'_n, P''_n are permutation matrices, S_n is bidiagonal, and D_n is a diagonal matrix (see [13] for details).

A transform usually has several different rules. Rules for the DFT that we can capture from fast algorithms as they are given in literature, include the Cooley-Tukey rule ($n = r \cdot s$ composite), Rader's rule (n prime), Good-Thomas rule ($n = r \cdot s$, $\gcd(r, s) = 1$), and several others (see [12]).

Besides breakdown rules, SPIRAL includes also rules for base cases, such as

$$\mathrm{DFT}_2 = \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right],$$

which shows that a DFT_2 can be computed with 2 additions/subtractions.

Formulas and Ruletrees: A formula is a mathematical expression that represents a sparse structured factorization of a matrix of fixed size. A formula is composed of mathematical operators (like $\cdot, \oplus, \otimes$), basic constructs (permutation, diagonal, plain matrix), symbolically represented matrices (like I_5 for an identity matrix of size 5), and transforms with fixed parameters. An example is

$$(\mathrm{DFT}_4 \otimes \mathrm{diag}(1, 7)) \cdot \left(\mathrm{I}_6 \oplus \left[\begin{smallmatrix} 1 & 4 \\ 2 & -1 \end{smallmatrix}\right]\right). \tag{3}$$

We call a formula *fully expanded* if it does not contain any transforms.

Expanding a transform M of a given size using one of the applicable rules creates a formula, which (possibly) contains transforms of smaller size. These, in turn, can be expanded further using the same or different rules. After all transforms have been expanded, we obtain a formula that represents in a unique way a fast algorithm for M. Since the formula is uniquely determined by the rules applied in the different stages, we can represent a formula, and hence an algorithm, by a tree in which each node is labeled with a transform (of size) and the rule applied to it. A node has as many children as the rule contains smaller transforms (e.g., the Cooley-Tukey rule (2) gives rise to binary trees). We call such a tree a *ruletree*. The ruletree is fully expanded, if all its leaves are base cases. Thus, within our framework,

$$\text{fully expanded ruletree} = \text{fully expanded formula} = \text{algorithm}.$$

An example for a fully expanded ruletree for a $\mathrm{DCT}_8^{(\mathrm{IV})}$ is given in Figure 2 (we omitted the rules for the base cases). The rules' identifiers used are not of significance.

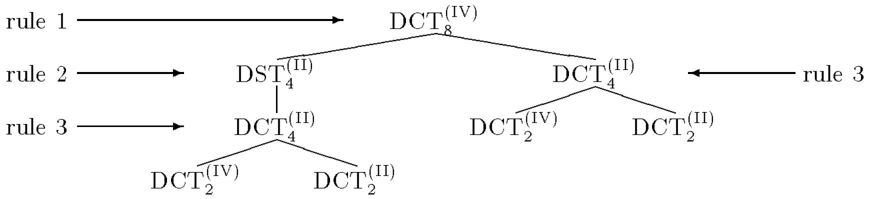

Fig. 2. A ruletree for $\mathrm{DCT}^{(\mathrm{IV})}$, size 8

2.3 The Formula Space

Applying different rules in different ways when expanding a transform gives rise to a surprisingly large number of mathematically equivalent formulas. Applying only the Cooley-Tukey rule (2) to a DFT of size $n = 2^k$ gives rise to $\Theta(5^k/k^{3/2})$ many different formulas. This large number arises from the degree of freedom in splitting 2^k into 2 factors. Using different rules and combinations thereof leads to exponential growth (in n) in the number of formulas. As an example, the current implementation of the formula generator contains 13 transforms and 31 rules and would produce about 10^{153} different formulas for the $\mathrm{DCT}_{512}^{(\mathrm{IV})}$.

By using only the best rules available (regarding the number of additions and multiplications), the algorithms that can be derived all have about the same arithmetic cost. They differ, however, in their data access during computation, which leads to very different runtime performances. As an example, Figure 3 shows a histogram of runtimes for all 31,242 formulas generated with our current set of rules for a $\mathrm{DCT}_{16}^{(\mathrm{IV})}$. The histogram demonstrates that even for a transform of small size, there is a significant spread of running times, more than a factor of two from the fastest to the slowest. Further, it shows that there are relatively few formulas that are amongst the fastest.

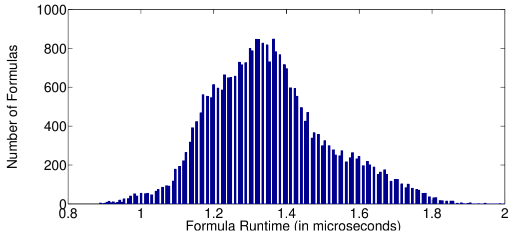

Fig. 3. Histogram of running times for all 31,242 $\mathrm{DCT}^{(\mathrm{IV})}$, size 2^4, formulas generated by SPIRAL's formula generator on a Pentium III running Linux.

3 The Formula Generator

Briefly, the formula generator is a module that produces DSP algorithms given as formulas for a user specified transform of given size. The formula generator is coupled with a search module that uses a feedback loop of formula generation and evaluation to optimize formulas with respect to a given performance measure.

Formula generation and formula manipulation fall into the realm of symbolic computation, which lead us to choose the language and computer algebra system GAP [8], including AREP [14], as an implementation platform. GAP provides the infrastructure for symbolic computation with a variety of algebraic objects. The GAP share package AREP is particularly focused on structured matrices and their symbolic manipulation. A high level language like GAP with its readily available functionality facilitates the implementation of our formula generator. It provides, as an additional advantage, *exact* arithmetic for square roots, roots of unity, and trigonometric expressions that make up the entries of most DSP transforms and formulas. The current implementation of the formula generator has about 12,000 lines of code.

The main objectives for the implementation of the formula generator are

- **efficiency:** it should generate formulas *fast* and store them efficiently; this is imperative since the optimization process requires the generation of many formulas;
- **extensibility:** it should be easy to expand the formula generator by including new transforms and new rules.

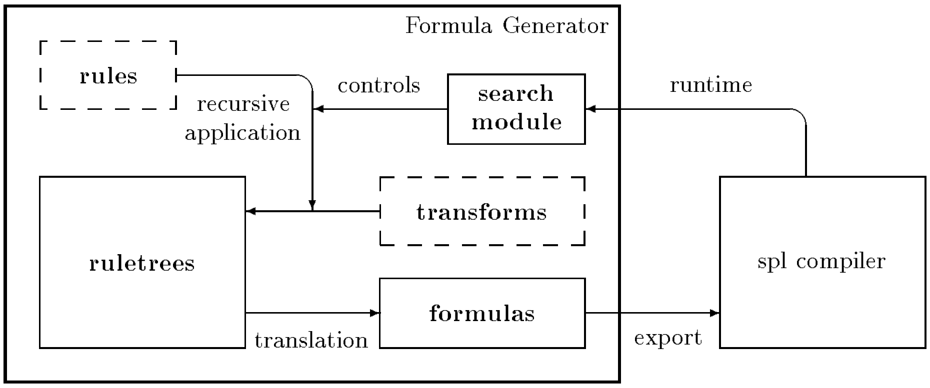

Fig. 4. Internal architecture of the formula generator including the search module. The main components are recursive data types for representing ruletrees and formulas, and extensible data bases (dashed boxes) for rules and transforms.

The architecture of the formula generator and the process of formula generation is depicted in Figure 4. We start with a transform with given parameters as desired by the user, e.g., a $\mathrm{DCT}_{64}^{(\mathrm{II})}$. The transform is recursively expanded into a ruletree. The choice of rules is controlled by the search module (see Section 4). The ruletree then is converted into a formula, which, in turn, is exported as an SPL program. The SPL program is compiled into a Fortran or C program (see [7]). The runtime of the program is returned to the formula generator. Based on the outcome, the search module triggers the derivation of different ruletrees.

By replacing the spl compiler block in Figure 4 by another evaluation function, the formula generator becomes a potential optimization tool for DSP algorithms with respect to other performance measures. Examples of potential interest include numerical stability or critical path length.

As depicted in Figure 4, and consistent with the framework presented in Section 2, the main components of the formula generator are formulas, transforms, rules, and ruletrees. Formulas and ruletrees are objects meant for computation and manipulation, and are realized as recursive data types. Transforms and rules are merely collections of information needed by the formula generator. We elaborate on this in the following. The search module is explained in Section 4.

Formulas: Formulas are implemented by the recursive data type SPL. We chose the name SPL since it is similar to the language SPL understood by the formula translator (see Section 1). A formula is an instantiation of SPL and is called spl. An spl is a GAP record with certain fields mandatory to all spls. Important examples are the fields `dimensions`, which gives the size of the represented matrix, and the field `type`, which contains a string indicating the type of spl, i.e, node in the syntax tree. Basic types are `diag` for diagonal matrices or `perm` for permutation matrices. Examples for composed types are `tensor` or `directSum`. The type `symbol` is used to symbolically represent frequently occurring matrices such as identity matrices I_n. The list of symbols known to the formula generator can be extended. A complete overview of all types is given in Table 1.

```
<spl> ::= <matrix>                              ; "mat"
        | <diagonal>                            ; "diag"
        | <permutation>                         ; "perm"
        | Symbol( <name>, <params> )            ; "symbol"
        | NonTerminal( <name>, <params> )       ; "nonTerminal"
        | <spl> * .. * <spl>                    ; "compose"
        | DirectSum(<spl>, .., <spl>)           ; "directSum"
        | TensorProduct(<spl>, .., <spl>)       ; "tensor"
        | <scalar> * <spl>                      ; "scalarMultiple"
        | <spl> ^ <"perm"-spl>                  ; "conjugate"
        | <spl> ^ <positive-int>                ; "power"
```

Table 1. The data type SPL in Backus-Naur form as the disjoint union of the different types. The string identifying the type is given in double quotes

The data type SPL mirrors the language SPL (Section 1) with the exception of the type `nonTerminal`. A `nonTerminal` spl represents a transform of fixed size, e.g., DFT_{16}, within a formula. The non-terminal spls available depend on the global list of transforms, which is explained below.

Other fields are specific to certain types. For example, an spl of type `diag` has a field `element` that contains the list of the diagonal entries; an spl of type `compose` has a field `factors` containing a list of spls, which are the factors in the represented product. For each of the types a function is provided to construct the respective spls. As an example, we give the spl corresponding to the formula in (3) as it is constructed in the formula generator.

```
ComposeSPL(TensorSPL(SPLNonTerminal("DFT", 4), SPLDiag([1, 7])),
          DirectSumSPL(SPLSymbol("I", 6), SPLMat([[1, 4], [2, -1]])))
```

Transforms: All transforms known to the formula generator are contained in the global list `NonTerminalTable`. Each entry of the list is a record corresponding to one transform (e.g., DFT). The record for a transform M stores the necessary information about M. Important fields include (1) `symbol`, a string identifying M (e.g, "DFT"); (2) `CheckParams`, a function for checking the validity of the parameters used to create an instantiation of M, usually the parameter is just the size, but we allow for arbitrary parameters; (3) `TerminateSPL`, a function to convert an instantiation of M into a plain matrix (type `mat`), used for verification. An instantiation of a transform (e.g., a DFT_{16}) is created as an spl of type `nonTerminal` as explained in the previous paragraph. The transform table can easily be extended by supplying this record for the new transform to be included.

Rules: All breakdown rules known to the formula generator are contained in the global list `RuleTable`. Each entry of the list corresponds to one rule (e.g., Cooley-Tukey rule). Similar to the transforms, rules are records storing all necessary information about the rule. Important fields of a rule R include (1) `nonTerminal`, the symbol of the transform R applies to (e.g., "DFT"); (2) `isApplicable`, a function checking whether R is applicable to a transform with the given parameters (e.g., Cooley-Tukey is applicable iff n is not prime); (3) `allChildren`, a function returning the list of all possible children configurations for R given the transform parameters, children are non-terminal spls; (4) `rule`, the actual rule, given the parameters for transform, returns an spl.

The rule table can also easily be extended by supplying this record for the new rule to be included.

Ruletrees: A ruletree is a recursive data type implemented as a record. Important fields include (1) `node`, the non-terminal spl expanded at the node; (2) `rule`, the rule used for expansion at the node; (3) `children`, ordered list of children, which again are ruletrees. In addition, we allow for a field `SPLOptions` that controls implementation choices that cannot be captured on the formula, i.e., algorithmic, level. An example is the code unrolling strategy. Setting `SPLOptions` to `"unrolling"` causes the code produced from the entire subtree to be unrolled.

There are two main reasons for having ruletrees as an additional data structure to formulas (both represent DSP algorithms): (1) ruletrees require much less storage than the corresponding formulas (a ruletree only consists of pointers to rules and transforms) and can be generated very fast, thus moving the bottleneck in the feedback loop (Figure 4) to the spl compiler; and (2) the search algorithms (see Section 4) use the ruletree data structure to easily derive variations of algorithms in the optimization process.

Infrastructure: In addition to these data types, the formula generator provides functionality for their manipulation and investigation. Examples include functions that (1) convert ruletrees into formulas; (2) export formulas as SPL programs; (3) convert formulas into plain matrices; (4) verify rules (for given transforms) and formulas using exact arithmetic where possible; (5) compute an upper bound for the arithmetic cost of an algorithm given as a formula.

4 Search

In this section, we discuss the search module shown in Figure 4 and how it interfaces with the formula generator. Given that there is a large number of formulas for any given signal transform, an important problem is finding a formula that runs as fast as possible. Further, the runtimes of formulas for a given transform vary widely as shown in Figure 3. Unfortunately, the large number of formulas for any given signal transform makes it infeasible to exhaustively time every formula for transforms of even modest sizes. Thus, it is crucial to *intelligently* search the space of formulas. We have implemented the following search methods.

Exhaustive Search: Determines the fastest formula, but becomes infeasible even at modest transform sizes since there is a large number of formulas.

Dynamic Programming: A common approach has been to use dynamic programming (DP) [15]. DP maintains a list of the fastest formulas it has found for each transform and size. For a particular transform and its applicable rules, DP considers all possible sets of children. For each child, DP substitutes the best ruletree found for that transform. DP makes the assumption that the fastest ruletree for a particular transform is also the best way to split a node of that transform in a larger tree. For many transforms, DP times very few formulas and still is able to find reasonably fast formulas.

Random Search: A very different approach is to generate a fixed number of random formulas and time each. This approach assumes that there is a sufficiently large number of formulas that have runtimes close to the optimal.

STEER: As a refinement to random search, we have developed an evolutionary stochastic search algorithm called STEER [16]. STEER is similar to standard genetic algorithms [17] except it uses ruletrees instead of a bit representation. For a given transform and size, STEER generates a population of random ruletrees and times them. Through evolutionary techniques, STEER produces related new ruletrees and times them, searching for the fastest one. STEER times significantly less formulas than exhaustive search would but usually searches more of the formula space than dynamic programming.

These search algorithms must interface with the formula generator to produce the formulas that they wish to time. Ruletrees were specifically designed to be an efficient representation equivalent to a formula and a convenient interface between the search module and the formula generator. The search algorithms can very easily manipulate ruletrees without needing to parse through long formulas. Further, the search algorithms can interface with the formula generator to expand or change ruletrees as the search algorithms need. Dynamic programming needs the ability to apply all breakdown rules to any given transform and size, producing all possible sets of children for each applicable rule. A ruletree is a convenient data structure as dynamic programming will substitute for each of these children the ruletree that it has found to be fastest for that child's transform and size. STEER and random search requires the ability to choose a random applicable rule, and to choose randomly from its possible sets of children. For crossover, STEER takes advantage of the ruletree data structure to allow it to easily swap two subtrees between two ruletrees.

We conclude with a comparison of the different search strategies. Figure 5 shows the runtimes of the fastest formulas found by several search methods across several transforms. In general, STEER performs the best, outperforming DP for many of the transforms. However, STEER often times the most formulas; for example, DP times 156 formulas and STEER 1353 formulas for the DFT of size 2^{10}. We have also compared SPIRAL against FFTW 2.1.3. At size 2^4, FFTW is about 25% slower than SPIRAL probably due to the overhead caused by FFTW's plan data structure. Thus, we omitted this data point in the diagram. At size 2^{10}, SPIRAL is performing comparable with FFTW.

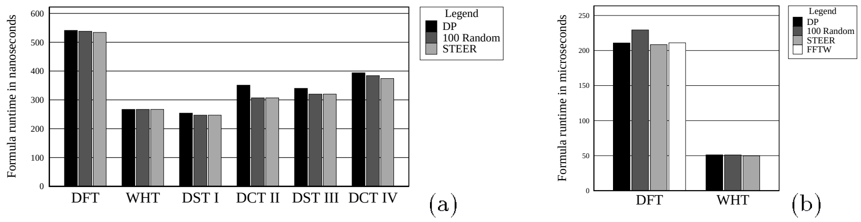

Fig. 5. Runtimes of the fastest formulas, implemented in C, found on a SUN UltraSparc-IIi 300 MHz for various transforms of size (a) 2^4 and (b) 2^{10}.

References

1. J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. Prasanna, M. Püschel, and M. M. Veloso, " SPIRAL: Portable Library of Optimized Signal Processing Algorithms," 1998, `http://www.ece.cmu.edu/~spiral`.
2. Matteo Frigo and Steven G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *ICASSP 98*, 1998, vol. 3, pp. 1381–1384, `http://www.fftw.org`.
3. C. Überhuber et.al., "Aurora," `http://www.math.tuwien.ac.at/~aurora/`.
4. M. Thottethodi, S. Chatterjee, and A. R. Lebeck, "Tuning Strassen's Matrix Multiplication for Memory Efficiency," in *Proc. SC98: High Performance Networking and Computing*, 1998.
5. J. Demmel et.al., "PHIPAC,"`http://www.icsi.berkeley.edu/~bilmes/phipac/`.
6. R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS project," Tech. Rep., University of Knoxville, Tennessee, 2000, `http://www.netlib.org/atlas/`.
7. J. Xiong, D. Padua, and J. Johnson, "SPL: A Language and Compiler for DSP Algorithms," in *Proc. PLDI*, 2001, to appear.
8. The GAP Team, University of St. Andrews, Scotland, *GAP – Groups, Algorithms, and Programming*, 1997, `http://www-gap.dcs.st-and.ac.uk/~gap/`.
9. J. Johnson and R. W. Johnson, "Automatic generation and implementation of FFT algorithms," in *Proc. SIAM Conf. Parallel Proc. for Sci. Comp.*, 1999, CD-Rom.
10. J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. of Computation*, vol. 19, pp. 297–301, 1965.
11. K. R. Rao and J. J. Hwang, *Techniques & standards for image, video and audio coding*, Prentice Hall PTR, 1996.
12. R. Tolimieri, M. An, and C. Lu, *Algorithms for discrete Fourier transforms and convolution*, Springer, 2nd edition, 1997.
13. Z. Wang, " Fast Algorithms for the Discrete W Transform and for the Discrete Fourier Transform ," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. ASSP-32, no. 4, pp. 803–816, 1984.
14. S. Egner and M. Püschel, *AREP – Constructive Representation Theory and Fast Signal Transforms*, GAP share package, 1998, `http://www.ece.cmu.edu/~smart/arep/arep.html`.
15. H. W. Johnson and C. S. Burrus, "The design of optimal DFT algorithms using dynamic programming," *IEEE Trans, on Acoustics, Speech, and Signal Processing*, vol. ASSP-31, pp. 378–387, 1983.
16. B. Singer and M. Veloso, "Stochastic search for signal processing algorithm optimization," in *Conf. on Uncertainty in Artificial Intelligence*, 2001, submitted.
17. David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, 1989.