Automating the Modeling and Optimization of the Performance of Signal Transforms

Bryan Singer and Manuela Veloso*

Computer Science Department

Carnegie Mellon University

5000 Forbes Avenue

Pittsburgh, PA 15213-3891

Phone: 412-268-1474

Fax: 412-268-4801

Email: {bsinger+, mmv+}@cs.cmu.edu

Abstract

Fast implementations of discrete signal transforms, such as the discrete Fourier transform, the Walsh-Hadamard transform, and the discrete trigonometric transforms, can be viewed as factorizations of their corresponding transformation matrices. A given signal transform can have many different factorizations, with each factorization represented by a unique but mathematically equivalent formula. When implemented in code, these formulas can have significantly different running times on the same processor, sometimes differing by an order of magnitude. Further, the optimal implementations on various processors are often different. Given this complexity, a crucial problem is automating the modeling and optimization of the performance of signal transform implementations.

To enable computer modeling of signal processing performance, we have developed and analyzed more than 15 feature sets to describe formulas representing specific transforms. Using some of these features and a limited set of training data, we have successfully trained neural networks to learn to accurately predict performance of formulas, with error rates less than 5%. In the direction of optimization, we have developed a new stochastic evolutionary algorithm, STEER, for finding fast implementations of a variety of signal transforms. STEER is able to optimize completely new transforms specified by a user. We present results that show STEER can find discrete cosine transform formulas that are 10–20% faster than what a dynamic programming search finds.

I. Introduction

Many signal transforms can be represented by a transformation matrix A which is multiplied by an input data vector X to produce the desired output vector $Y = A\,X$. To allow for fast implementations, the transformation matrices often can be factored into a product of structured matrices. Further, these factorizations can be represented by mathematical

formulas and a single transform can be represented by many different, but mathematically equivalent, formulas [1]. As an example of the number of different formulas, we consider 51,819 formulas for a Walsh-Hadamard transform of size 2^{10} and 31,242 formulas for a discrete cosine transform of size 2^4.

Interestingly, when these formulas are implemented in code and executed, they have very different running times, differing by as much as an order of magnitude. While many of the factorizations produce the exact same number of operations, the different orderings of the operations that the factorizations produce can greatly impact the performance of the formulas. For example, different operation orderings can greatly impact the number of cache misses and register spills that a formula incurs. The complexity of computing platforms makes it difficult to analytically predict or model by hand the performance of formulas. Further, the differences between computing platforms lead to different optimal formulas from machine to machine. To address this problem, we have developed methods that automate the modeling and optimization of performance across a variety of signal transforms.

To enable the use of neural networks to automatically learn performance models, we have characterized formulas in terms of numerical features, that is, numerical values describing different aspects of the formula such as the transform size. We have explored over 15 feature sets to describe mathematical formulas of particular transforms, identifying feature sets with different abilities to partition formulas according to their running times. By describing formulas with features, we can train a neural network to learn to predict the running times of formulas. We have shown that a neural network can learn to accurately predict with less than 5% error the faster of two formulas or the running time of a formula given a limited set of training data.

Signal processing optimization presents a challenging search problem as there is a large number of formulas that represent the same signal processing algorithm. Exhaustive search is only possible for very small transform sizes or over limited regions of the space of formulas. Dynamic programming offers a more effective search method. By assuming independence among substructures, dynamic programming searches for a fast implementation while only timing a few formulas. However, this independence assumption does not always hold for this domain and there is no bound as to how much slower the implementation found by dynamic programming is compared to the optimal implementation. We present a stochastic evolutionary algorithm, STEER, for searching through this large space of possible formulas. STEER searches through many more formulas than dynamic programming, covering a larger portion of the search space, while still timing a tractable number of formulas as opposed to exhaustive search. We initially developed STEER specifically for the Walsh-Hadamard Transform and then extended it to work across a wide variety of transforms. Through empirical comparisons, we show that STEER can find discrete trigonometric transform formulas that run 10–20% faster than what

dynamic programming finds. Further, for the discrete cosine transform of type IV and of size 2^4 on a Pentium III, STEER finds a formula that runs 17% faster than that found by dynamic programming and almost equally fast as that found by an exhaustive search, but STEER times about 2 orders of magnitude less formulas than exhaustive search.

This article is arranged as follows. Section II gives an overview of the background necessary for understanding the rest of the article. Section III presents our work in modeling performance of signal transforms and Section IV describes our work in optimizing the performance. The performance modeling work in Section III is independent of the optimization work in Section IV, and either section can be read separately from the other. Finally, we conclude in Section V.

II. Background

This section presents some necessary background for understanding the remainder of the article. Section II-A describes the Walsh-Hadamard Transform (WHT) in detail. Section II-B describes the SPIRAL system that we have been developing in collaboration with others to optimize the performance of a wide variety of signal transforms. Section II-C then highlights some of the significant differences and similarities between the WHT and some of the other transforms being considered. Finally, we discuss some related work in Section II-D.

A. Walsh-Hadamard Transform

The Walsh-Hadamard Transform of a signal x of size 2^n is the product $WHT(2^n) \cdot x$ where

$$WHT(2^n) = \bigotimes_{i=1}^{n} DFT(2), \quad DFT(2) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

and $\otimes$ is the tensor or Kronecker product [2]. For example,

$$WHT(2^2) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}.$$

By calculating and combining smaller WHTs appropriately, the structure in the WHT transformation matrix can be leveraged to produce more efficient algorithms. Let $n = n_1 + \cdots + n_t$ with all of the n_j being positive integers. Then, $WHT(2^n)$ can be rewritten as

$$\prod_{i=1}^{t} (I_{2^{n_1+\cdots+n_{i-1}}} \otimes WHT(2^{n_i}) \otimes I_{2^{n_{i+1}+\cdots+n_t}})$$

where I_k is the $k \times k$ identity matrix. This formula or break down rule can then be recursively applied to each of these new smaller WHTs. Thus, $WHT(2^n)$ can be rewritten as any of a large number of different but mathematically

equivalent formulas.

Any of these formulas for $WHT(2^n)$ can be uniquely represented by a tree, which we call a "split tree." For example, suppose $WHT(2^5)$ was factored as:

$$\begin{aligned} & WHT(2^5) \\ = \; & [WHT(2^3) \otimes I_{2^2}][I_{2^3} \otimes WHT(2^2)] \\ = \; & [\{(WHT(2^1) \otimes I_{2^2})(I_{2^1} \otimes WHT(2^2))\} \otimes I_{2^2}][I_{2^3} \otimes \{(WHT(2^1) \otimes I_{2^1})(I_{2^1} \otimes WHT(2^1))\}] \end{aligned}$$

The split tree corresponding to this final formula is shown in Figure 1(a). Each node in the split tree is labeled with the base two logarithm of the size of the WHT at that level. The children of a node indicate how the node's WHT is recursively computed.

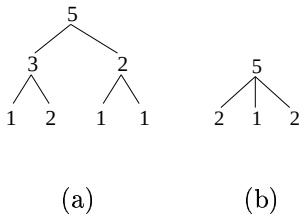


Fig. 1. Two different split trees for $WHT(2^5)$

In general, each node of a split tree should contain not only the size of the transform, but also the transform at that node and the break down rule being applied. Recall that a break down rule specifies how a transform can be computed from smaller or different transforms. In Figure 1, the representation was simplified since it only used one break down rule which only involved WHTs.

There is a very large number of possible split trees for a WHT of any given size, and thus there is a large number of formulas equal to that WHT. Specifically, a WHT of size 2^n has on the order of $\theta((4+\sqrt{8})^n/n^{3/2})$ different possible split trees. For example, $WHT(2^8)$ has 16,768 different split trees. To slightly reduce the search space, it is possible to only consider binary WHT split trees. There are still on the order of $\theta(5^n/n^{3/2})$ possible binary split trees [3].

For the results with the WHT, we used a WHT package, [3], which can implement in code, run, and time WHT formulas passed to it. The WHT package allows leaves of the split trees to be sizes 2^1 to 2^8 which are implemented as unrolled straight-line code. This introduces a trade-off since straight-line code has the advantage that it does not have loop or recursion overhead but the disadvantage that large code blocks will overfill the instruction cache. The package performs the transform using double-precision floating point.

B. SPIRAL System

As part of the SPIRAL (Signal Processing algorithms Implementation Research for Adaptable Libraries) research group [4], we are developing a system to optimize a wide variety of signal transforms. The SPIRAL system consists of four main steps:

1. **Transform and Break Down Rule Specification.** The system begins by allowing the user to specify new transforms and new break down rules to factor the transforms. These break down rules specify how a given transform can be factored into other smaller components. Several transforms, including the DFT, WHT, and four types of discrete cosine and sine transforms (DCT/DST), and over 35 break down rules have already been specified.

2. **Formula Generation.** The second step in the system is to apply the break down rules repeatedly to produce a factorization of a given transform. The resulting formula represents an algorithm for computing the given transform. There are many different ways to apply the break down rules leading to a very large number of different formulas that can be obtained.

3. **Code Generation.** Given a formula, the third step is to implement it in code. To perform this step, a compiler has been developed to translate formulas into a low-level language such as Fortran or C [5]. The formula compiler has a number of parameters that can be adjusted to change how it implements a formula in code. For example, a parameter can be specified to indicate how much of the code should be unrolled and how much should be left as loops. This provides another dimension that must be optimized when searching for fast implementations. By default, the formula compiler implements the transform using double-precision floating point.

4. **Optimization.** With all of this machinery in place, it is possible to generate executable code for any possible factorization specified by the break down rules for any possible transform that has been defined to the system. The final but key step in the system, then, is to use this machinery to search for a fast implementation for a given transform. Our contribution to this system has been in research and implementation of this final step. Specifically, we have developed and implemented several search algorithms for this system.

C. Other Transforms

We are investigating a number of transforms besides the WHT, including the discrete Fourier Transform (DFT), four types of the discrete cosine transform (DCT), and four types of the discrete sine transform (DST). Table I gives a few example break down rules for some of these. The $\oplus$ operator denotes matrix direct sum and exponentiation denotes matrix conjugation.

The discrete trigonometric transforms (DTTs), DCT and DST, [6], [7], [8], [9] are considerably different from the

TABLE I

A FEW EXAMPLE BREAK DOWN RULES. DIAGONAL MATRICES ARE REPRESENTED BY D, PERMUTATIONS BY P AND L, ROTATION MATRICES BY R, AND OTHER SPARSE MATRICES BY M AND T.

$$
\begin{aligned}
WHT(2^n) &= \prod_{i=1}^{t} (I_{2^{n_1+\cdots+n_{i-1}}} \otimes WHT(2^{n_i}) \otimes I_{2^{n_{i+1}+\cdots+n_t}}) \\
DCT\_IV(n) &= M_n * DCT\_II(n) * D_n \\
DCT\_IV(n) &= M_n * P_n * (DCT\_II(n/2) \oplus DST\_II(n/2)^{P_{n/2}}) * R_n \\
DST\_I(n) &= P_n * (DST\_III(n/2) \oplus DST\_I(n/2)^{P_{n/2}}) * M_n \\
DFT(rs) &= (DFT(r) \otimes I_s)\, T_s^{rs}\, (I_r \otimes DFT(s))\, L_r^{rs} \\
DFT(n) &= CosDFT(n) + i * SinDFT(n) \\
CosDFT(n) &= M_n * (CosDFT(n/2) \oplus DCT\_II(n/4)) * M_n * P_n
\end{aligned}
$$

WHT. Specifically, the following differences are of importance:

- While we have used just one basic break down rule for the WHT, there are several different break down rules for most of the different types of DTTs.
- While the break down rule for the WHT allowed for many possible sets of children, most of the break down rules for the DCTs specify exactly one set of children.
- While the WHT factored into smaller WHTs, the break down rules for the DTTs often factor one transform into two transforms of different types or even translate one DTT into another DTT or into a discrete sine transform. Thus, a split tree for a DTT labels the nodes not only with the size of the transform, but also with the transform and the applied break down rule.
- The number of factorizations for the DTTs grows even quicker than that for the WHT. For example, DCT type IV already has about 1.9×10^9 different factorizations at size 2^5 and about 7.3×10^{18} factorizations at size 2^6 with our current set of break down rules.

D. Related Work

Much of the work in signal transform optimization has been concerned with minimizing the number of arithmetic operations necessary to compute a transform. Our work builds upon this in that we use break down rules to generate formulas with the minimal number of arithmetic operations. However, we are also concerned with how we can perform these arithmetic operations as quickly as possible and not just how to minimize their number. There are many formulas with the same minimal number of arithmetic operations but that have a wide variance in run times. We are concerned

with optimizing the run time performance of transforms on real machines.

Also addressing this issue, Frigo and Johnson have developed FFTW [10], [11], an adaptive package that produces efficient Fast Fourier Transform (FFT) implementations across a variety of machines. FFTW performs a constrained dynamic programming search to search for a fast FFT implementations. In this article, we look at several search methods beyond just dynamic programming and at optimizing performance across a wide variety of transforms beyond just the FFT.

In the related field of linear algebra optimization, PHiPAC [12], [13] and ATLAS [14] have developed a set of parameterized linear algebra algorithms. For each algorithm, a pre-specified search is made over the possible parameter values to find the optimal implementation. Their work uses a fair bit of human intelligence to determine which portions of the parameter space is to be explored.

A few researchers have addressed similar goals in optimizing other numerical algorithms. However, most of these approaches only search among a few algorithms instead of the space of thousands of different formulas we consider in our work. Lagoudakis and Littman [15] used reinforcement learning to learn to select between algorithms for solving sorting or order statistic selection problems. Brewer [16] uses linear regression to learn to predict running times for four different implementations of two different applications: iterative partial differential equation solving and sorting. By learning to predict running times across different input sizes, his algorithm was able to quickly predict which of the four implementations should run fastest given a new input size.

III. Modeling Performance

Empirical performance data can be gathered for a variety of formulas. This data offers an interesting opportunity *to learn* to predict running time performance. This section describes our work in automating the modeling of the performance of signal transforms [17]. This work has involved formulating and representing formula performance prediction as a machine learning task. Section III-A describes and evaluates several sets of features that we introduced for binary WHT split trees. We show that the choice of feature set significantly impacts the ability to distinguish between formulas with different running times. Section III-B demonstrates how a neural network can learn to accurately predict the running time of a formula given a limited set of training data.

A. Features for WHT Split Trees

To automatically learn to predict running times of formulas by using neural networks, one of the major steps is to be able to represent formulas with numerical features. The problem of feature selection in machine learning is important.

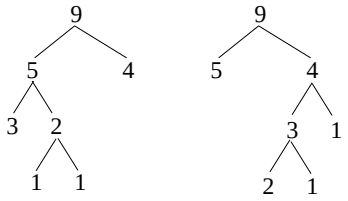


Fig. 2. Two split trees with the same All Nodes counts but different Leaf Nodes counts

The introduction and careful analysis of different feature sets for formula prediction represents a significant part of our work.

In selecting and evaluating features, an important question is what aspects of the formulas determine their running times. Or, equivalently, what are good features for predicting a formula's running time? To answer these questions, we introduce several different feature sets to describe WHT formulas. We then compare the feature sets along three measures to see how well the features can differentiate formulas with different running times.

A.1 Feature Sets

This section describes several feature sets for WHT formulas. In all of the features, we take advantage of the fact that WHT formulas can be represented as split trees. These features are not unique to the WHT and many have been used in our early study of the FFT [18]. However, some of the features sets can only be used to describe binary split trees. We consider feature sets from two broad categories, *node count* features and features corresponding to the *shape* of the split tree. These features are chosen to capture both the size of the computations being performed as well as the ordering of those computations and thus to hopefully capture the running time.

A.1.a Counting Nodes. The following formula features sets count the number of nodes of various types:

- **Leaf Nodes.** The Leaf Nodes feature sets counts the number of different sized leaves in the WHT split tree. The leaves of a WHT split tree correspond to the WHTs that must actually be computed directly and that appear in the formula represented by the split tree. Specifically, this feature set counts the number of $WHT(2^1)$'s, the number of $WHT(2^2)$'s, the number of $WHT(2^3)$'s, and so on that appear in the formula.
- **All Nodes.** One modification of the Leaf Nodes features is to count all of the nodes of the split tree instead of just the leaves. This not only indicates what size WHTs must be directly computed but also what intermediate sizes are combined from smaller ones (although this feature set can not distinguish leaf from internal nodes).
- **Leaf and All Nodes.** For sufficiently large split trees, it is possible for two different formulas to have the exact same All Nodes counts, but to have different Leaf Nodes counts. For example, see Figure 2. So, a simple refinement of those two feature sets is to include both.

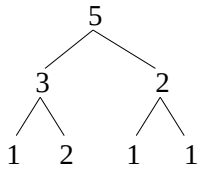


Fig. 3. Example Split Tree.

TABLE II

Example values of the different node count features for the tree shown in Figure 3.

Features:	leaf			all			right leaf			left leaf			right all			left all		
WHT size:	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Counts:	3	1	0	3	2	1	1	1	0	2	0	0	1	2	0	2	0	1

• **Left/Right Leaf Nodes.** Consider again the Leaf Nodes features which simply counted all of the leaf nodes. A different refinement of this is to separate nodes that are right children of their parents in the tree from those that are left children. This provides more information about the structure of the split tree which is an important factor in the performance of a split tree. In particular, the Left/Right Leaf Nodes feature set counts the number of left $WHT(2^1)$'s, the number of right $WHT(2^1)$'s, the number of left $WHT(2^2)$'s, and so on in the split tree.

• **Left/Right All Nodes.** Combining the previous idea along with the idea of counting all the nodes in the split tree produces yet another set of features. In particular, the Left/Right All Nodes feature set counts the number of different sized left and right nodes appearing in the tree, excluding the root node.

• **Left/Right Leaf and Left/Right All Nodes.** Once again, counting Left/Right All Nodes can not always distinguish two trees that counting Left/Right Leaf Nodes can distinguish. Thus, this feature set combines the two for a large set of features that include all those in the previous two sets.

Table II gives an example of these features for the split tree shown in Figure 3.

A.1.b Features of the Shape of the Tree. All of the above features count the number of various kinds of nodes of different sizes. Another feature category pertains to the general shape of the tree. Since the position of nodes within the split tree can influence the amount of time spent computing those nodes, the shape of the split tree is an important factor in the running time of the split tree.

A simple feature is the "leftness" or "rightness" of a tree. The path from the root node to any given node is a sequence of choosing either the left or right child for each node along the path. Let the *leftness of a node* in a tree be the number of left children chosen minus the number of right children chosen along the path from the root to the given node. Then the *leftness of the tree* is defined to be the sum of the leftness of all of the tree's nodes. The single number feature

Leftness is the leftness of a tree.

This single number feature can be expanded to provide the Vertical Profile feature set. In particular, the Vertical Profile feature set is an array of numbers, with each number indicating how many nodes have a particular leftness value.

For example, the split tree shown in Figure 3 has nodes with leftness as shown in Figure 4, and a total leftness of 0 since it is balanced.

- 3 nodes with leftness 0
- 1 node with leftness 1
- 1 node with leftness 2
- 1 node with leftness -1
- 1 node with leftness -2

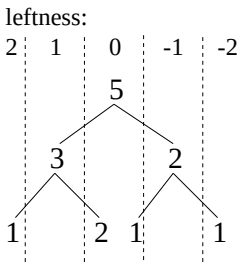


Fig. 4. Leftness of nodes in tree of Figure 3

There are several possible single numbers that capture some aspect of a tree's depth. The Total Path Length feature is the sum of the path lengths of every node to the root. The Average Path Length feature divides the total path length by the total number of nodes in the tree. The Horizontal Profile feature is constructed by counting the number of nodes at each possible depth.

For example, the split tree shown in Figure 3 has the following features:

- A total path length of 10
- An average path length of 10/7

and the horizontal profile is:

- 1 node at depth 0
- 2 nodes at depth 1
- 4 nodes at depth 2

A.2 Evaluating Features

In this section, the defined feature sets are evaluated.

A.2.a Number of Partitions. Because several different formulas can have the same set of feature values, the features can be thought of as generating a set of equivalence classes or partitions. Under a set of features, formulas are indistinguishable if they have the same set of feature values, while formulas are distinguishable if they have different feature values. For example, the two split trees shown in Figure 2 have the same feature values under the All Nodes feature set but have different feature values under the Leaf Nodes feature set. So, a partition consists of all formulas that have the

TABLE III

NUMBER OF PARTITIONS GENERATED BY FEATURE SETS FOR ALL BINARY TREES OF WHTS

Features	WHT size 2^5	2^6	2^7	2^8	2^9	2^{10}
Node Count:						
Leaf	7	11	15	22	29	40
All	13	31	68	168	384	947
Leaf & All	13	31	68	168	385	954
L/R Leaf	23	44	81	142	240	395
L/R All	45	149	523	1832	6584	23548
L/R Leaf & L/R All	49	170	617	2262	8472	31711
Shape:						
Leftness	11	19	29	41	55	71
Vert Prof	20	44	96	204	428	888
Tot Path Len	8	13	19	26	33	42
Avg Path Len	8	12	20	32	47	67
Horz Prof	8	13	22	38	65	115
Vert & Horz Prof	21	54	143	394	1087	3043
Composite:						
All Nodes & Leftness	36	117	373	1222	3878	12394
All Nodes & Vert Prof	36	122	409	1463	5183	18966
All Nodes & Tot Path Len	14	35	83	220	563	1533
All Nodes & Horz Prof	14	35	83	220	565	1544
All Nodes, Vert & Horz Prof	36	123	420	1535	5586	21140
All Formulas	51	188	731	2950	12234	51819

same feature values under a particular feature set.

Ideally, we would like all of the formulas that fall into the same partition to have very close running times. One straightforward method for achieving this is to create a large number of partitions causing few formulas to fall into any one partition. Thus, a very simple measure of the effectiveness of a set of features is the number of partitions it creates for a set of formulas. Some results are shown in Table III. For each of the sizes of the WHT in the table, all possible binary trees were generated. The bottom line of the table shows the number of different formulas produced. The remaining lines show how many different partitions or equivalence classes are generated by the different features for each set of formulas.

First, consider the top portion of Table III with node count features. The feature sets that are refinements of other feature sets have more partitions. For example, the All Nodes feature set has many more partitions than the Leaf Nodes feature set, and likewise all of the Left/Right feature sets have more partitions than their corresponding plain feature sets. The final feature set in this group, the Left/Right Leaf Nodes and Left/Right All Nodes features, is able to almost, but not quite, uniquely identify all the formulas. However, as the size of WHT grows, this feature set is less and less able to uniquely identify formulas.

The middle portion of the table considers features pertaining to the shape of the split trees. The leftness feature and the vertical profile produce more partitions than the path length features or the horizontal profile. However, none of these features produce as many partitions as some of the node count feature sets.

The lower portion of Table III combines the All Nodes features with some of the shape features. Combining the leftness feature or the vertical profile greatly increases the number of partitions over the All Nodes features while adding path lengths or the horizontal profile does not. This indicates that the All Nodes features incorporate more of the horizontal features than the vertical ones.

A.2.b Relative Standard Deviation. While being able to partition a set of formulas into a large set of equivalence classes is important, ultimately we want all of the formulas within a partition to have close running times. A good set of features can separate formulas with significantly different running times into different partitions so that all formulas within a single partition have close to the same running time. As a measure of this, we consider both the "weighted average relative standard deviation" and the maximum relative standard deviation.

The weighted average relative standard deviation and the maximum relative standard deviation are calculated as follows:

- Let P_k be the set of formulas in partition k.
- Let m_k be the mean running time of the formulas in P_k.
- Let σ_k be the standard deviation of the running times of the formulas in P_k.
- Let r_k be the relative standard deviation of the running times of the formulas in P_k.

Then $r_k = \frac{\sigma_k}{m_k}$.

- The Weighted Average Relative Standard Deviation is $\frac{\sum_k |P_k| r_k}{\sum_k |P_k|}$.
- The Maximum Relative Standard Deviation is $\max_k r_k$.

The weighted average relative standard deviation indicates *on average* how far apart running times of formulas in the same partition are. The maximum relative standard deviation indicates how far apart running times of formulas are in the *worst* partition.

Evaluating the feature sets, the weighted average and maximum relative standard deviations are shown in Table IV. For each WHT size shown in the table, formulas for all possible binary split trees were generated. These formulas were timed using the WHT package [3] on a Pentium III 450 MHz running Linux 2.2.5-15. The WHT package was compiled using gcc version egcs-2.91.66 with the options "`-O6 -fomit-frame-pointer -malign-double`." The last row in the table shows the relative standard deviation for all possible formulas, as though they were all placed in a single partition. Note that the maximum relative standard deviation for some feature sets can be larger than the relative standard deviation for all formulas because some partitions may have a significantly smaller mean than that of all formulas.

Looking at the top portion of Table IV, we see that using the Left/Right features tends to improve both standard

TABLE IV

WEIGHTED AVERAGE AND MAXIMUM RELATIVE STANDARD DEVIATION OF DIFFERENT FEATURE SETS FOR ALL BINARY TREES OF DIFFERENT SIZED WHTS. ENTRIES IN EACH CELL ARE PERCENTAGES, WEIGHTED AVERAGE FOLLOWED BY MAXIMUM.

	WHT size											
Features	2^5		2^6		2^7		2^8		2^9		2^{10}	
Node Count:												
Leaf	11.0	16.5	12.4	22.7	13.6	23.9	14.2	25.9	14.3	24.1	13.9	20.4
All	5.3	9.9	5.2	10.1	5.0	9.5	4.9	9.5	4.7	9.3	4.6	22.6
Leaf& All	5.3	9.9	5.2	10.1	5.0	9.5	4.9	9.5	4.7	9.3	4.6	22.6
L/R Leaf	7.7	15.2	10.1	22.1	11.8	24.5	12.7	25.2	13.0	24.7	12.8	21.6
L/R All	0.3	2.0	0.5	3.8	0.7	5.8	0.9	8.3	1.0	8.0	1.1	17.0
L/R Leaf& L/R All	0.1	1.7	0.2	3.1	0.3	5.8	0.5	8.3	0.6	8.0	0.7	19.6
Shape:												
Leftness	31.6	55.7	33.7	54.6	32.6	51.7	32.1	39.4	31.3	40.5	29.8	36.7
Vert Prof	9.0	18.5	11.5	23.1	12.7	26.3	13.4	26.7	13.5	36.0	13.2	34.3
Tot Path Len	9.9	16.5	11.5	20.9	14.8	25.4	17.5	28.6	19.1	36.0	19.9	34.2
Avg Path Len	9.9	16.5	12.8	21.6	13.0	24.4	12.6	25.4	11.8	36.0	11.3	34.2
Horz Prof	9.9	16.5	11.5	20.9	11.9	24.4	11.9	25.4	11.4	36.0	10.9	34.2
Vert& Horz Prof	8.6	18.5	10.4	23.1	10.9	26.3	11.0	26.7	10.6	36.0	10.2	34.3
Composite:												
All Nodes& Leftness	2.5	9.9	2.3	8.8	2.3	7.2	2.5	9.2	2.6	10.2	2.7	32.4
All Nodes& Vert Prof	2.5	9.9	2.2	8.8	2.1	7.2	2.2	8.3	2.2	9.4	2.3	32.4
All Nodes& Tot Path Len	5.2	9.9	5.1	10.1	4.9	9.5	4.8	9.5	4.6	9.3	4.5	22.6
All Nodes& Horz Prof	5.2	9.9	5.1	10.1	4.9	9.5	4.8	9.5	4.6	9.3	4.5	22.6
All Nodes, Vert& Horz Prof	2.5	9.9	2.2	8.8	2.1	7.2	2.1	8.3	2.1	9.4	2.1	32.4
All Formulas	42.9	42.9	39.0	39.0	35.6	35.6	34.1	34.1	32.3	32.3	30.6	30.6

deviation measures. The features that look at all nodes significantly outperform those just using the leaves. The middle portion of the table shows that the shape features are significantly poorer than the All Nodes features in both measures. However, the lower portions of the tables show that the leftness feature and vertical profile can help the weighted average of All Nodes. Overall, there several feature sets that produce very good weighted average results and even reasonable maximum results for sizes smaller than 2^{10}.

As a point of comparison, we chose 12 random WHT formulas for each of the sizes from 2^5 to 2^{10} and timed them repeatedly 100 times. For each formula, we then computed the relative standard deviation of its 100 timings. The average relative standard deviation over all formulas was 0.57%.

When considering both the relative standard deviation results along with the number of partitions, the All Nodes features and the Leaf and All Nodes features are surprisingly impressive. Not only do these feature sets produce good relative standard deviation results, but they do so with relatively few partitions.

B. Learning to Predict WHT Performance

With the features discussed in the previous section and with training data obtained by timing a set of formulas, we can use neural networks to learn to quickly predict the running times of new formulas. Note that such a neural network still does not solve the problem of searching through a large space of potential formulas. However, a predicted running time can now be obtained much more quickly than we could have obtained an actual running time.

While accurately predicting a formula's running time allows the fastest formula to be determined through exhaustive search over all formulas, it is actually more than necessary. In particular, accurately predicting which of two formulas runs faster would also allow the fastest formula to be determined through exhaustive search over all formulas. Thus, a learning algorithm need not learn the exact running time if it can accurately predict which of two formulas runs faster.

B.1 Experimental Setup

The results that are presented in this section are for $WHT(2^8)$ and are similar to those collected for other sizes. All 2950 possible formulas corresponding to binary trees of $WHT(2^8)$ were generated and timed using the WHT package on the same Pentium III running Linux.

We used a back-propagation neural network with 50 hidden sigmoid units, a learning rate 0.01 and a momentum of 0.001. These parameters are not highly tuned due to the fact that they were used across several different input feature sets (of varying number of inputs) and across desired output (running time or faster of two formulas). For predicting running times, a single linear output unit was used, while for predicting the faster of two formulas a single sigmoid output unit was used. The training minimized the mean squared error.

The various node count feature sets were used as inputs to the neural network. The set of formulas were partitioned into training, validation, and testing sets of different sizes. Except in the cases where all of the formulas are used for both the training and testing sets, the results presented are averages over four random splits into training, validation, and testing sets. The neural network was allowed to train for 5000 epochs, but every 100 epochs the network was tested against the validation set and if this produced the lowest seen error then the network was saved. So, the final saved network was the one with lowest error on the validation set during training. This saved network was then tested against the data in the test set.

The neural networks were trained on two different tasks: (1) to predict the running times of formulas, and (2) to predict which of two formulas would run faster.

B.2 Results

Results are shown in Table V. The column marked "Cost" reports the percent error on predicting the running times for the test set. The column marked "Faster" corresponds to predicting the faster of two formulas. This column reports the percentage of random pairs of formulas in the test set that our method mispredicts which is faster. In particular, the number of samplings was 100 times the number of formulas in the test set. The "Cost" and "Faster" columns should not be directly compared as they report different measures of performance.

The Left/Right All Nodes feature set and the "Left/Right Leaf Nodes and Left/Right All Nodes" feature set yield the best learning results. These results were quite good with about 4% error on predicting the faster of two formulas and about 3% error on predicting the running times even when trained on only 10% of the formulas. The All Nodes feature set and the Leaf and All Nodes feature set, which were discussed earlier for their excellent performance at partitioning the formulas, also perform well here, obtaining about 5% error on predicting the running times and less than 8% error on predicting the faster of two formulas. The Leaf Nodes and Left/Right Leaf Nodes feature set both perform significantly worse than all of the other feature sets.

C. Summary

To model performance of signal transforms, we have explored using neural networks to learn to predict running times of formulas. In order to use neural networks, we have developed feature sets to describe split trees representing signal transform formulas. We have explored a variety of feature sets, identifying feature sets with different abilities to partition formulas according to their running times. Further, some simple feature sets do well at partitioning the space of formulas according to their running times. By describing formulas with features, we can present formulas to a neural network. We showed that performance varied according to what set of features were used. With several feature sets such as All Nodes or Left/Right All Nodes, we showed that a neural network can learn to accurately predict the faster of two formulas or the running time of a formula given a limited set of training data.

IV. Optimizing Performance

This section describes our work in developing search methods for finding fast implementations of signal transforms. Section IV-A overviews several search methods for this domain. Section IV-B describes the evolutionary algorithm STEER for searching for fast WHT implementations. Then, Section IV-C compares these search techniques for the WHT. Finally, Section IV-D describes modifications to these search algorithms to allow them to work on arbitrary transforms and results using them.

A. Search Techniques

There are several approaches for searching for fast implementations of signal transforms, including exhaustive search, dynamic programming, random search, and evolutionary algorithms.

One simple approach to optimization is to exhaust over all possible formulas of a signal transform and to time each one on each different machine that we are interested in. There are three problems with this approach: (1) each formula may take a non-trivial amount of time to run, (2) there is a very large number of formulas that need to be run, and

TABLE V

NEURAL NETWORK PREDICTION ACCURACY FOR $WHT(2^8)$ WITH NODE COUNTING FEATURES. THE COLUMN MARKED "COST" IS THE AVERAGE PERCENT ERROR ON PREDICTING RUNNING TIME. THE COLUMN MARKED "FASTER" IS THE PERCENT MISTAKES ON PREDICTING THE FASTER OF TWO FORMULAS. THE SIZE OF THE TRAINING, VALIDATION, AND TEST SETS ARE SHOWN IN PERCENTAGES.

Features	Train	Val.	Test	Cost	Faster
Leaf	100	100	100	12.67	24.14
	75	10	15	13.08	25.49
	50	15	35	12.92	25.44
	25	15	60	13.13	25.63
	10	15	75	13.24	24.27
All	100	100	100	4.46	7.41
	75	10	15	4.60	7.49
	50	15	35	4.70	7.46
	25	15	60	4.87	7.56
	10	15	75	5.10	7.79
Leaf & All	100	100	100	4.32	7.62
	75	10	15	4.61	7.21
	50	15	35	4.61	7.37
	25	15	60	4.85	7.43
	10	15	75	5.23	7.66
L/R Leaf	100	100	100	11.71	21.33
	75	10	15	12.17	21.32
	50	15	35	12.13	21.56
	25	15	60	12.42	21.18
	10	15	75	12.66	21.80
L/R All	100	100	100	1.40	3.04
	75	10	15	1.79	3.06
	50	15	35	1.91	3.17
	25	15	60	2.26	3.44
	10	15	75	2.87	4.13
L/R Leaf and L/R All	100	100	100	1.14	2.86
	75	10	15	1.79	2.96
	50	15	35	1.84	3.20
	25	15	60	2.24	3.18
	10	15	75	3.02	3.76

(3) just enumerating all of the possible formulas may be impossible. These problems make the approach intractable for transforms of even small sizes.

With the WHT, there are several ways to limit the search space. One such limitation is to exhaust just over the binary split trees, although there still are many binary split trees. In practice we have found that the fastest WHT formulas never have leaves of size 2^1. By searching just over split trees with no leaves of size 2^1, the total number of trees that need to be timed can be greatly reduced (for example, from 51,819 to 101 trees for size 2^{10}), but still becomes intractable at larger sizes.

A common approach for searching the very large space of possible implementations of signal transforms has been to use dynamic programming [19], [10], [20], [21]. This approach maintains a list of the fastest formulas it has found for each transform and size. When trying to find the fastest formula for a particular transform and size, it considers all possible splits of the root node. For each child of the root node, dynamic programming substitutes the best split tree found for that transform and size. Thus, dynamic programming makes the following assumption:

Dynamic Programming Assumption: The fastest split tree for a particular transform and size is also the best way to split a node of that transform and size in a larger tree.

While dynamic programming times relatively few formulas for many transforms, it would need to time an intractable number of formulas for large WHTs. However, by restricting to just binary WHT split trees, dynamic programming becomes efficient. Between the two extremes, k-way dynamic programming considers split trees with at most k children at any node. Unfortunately, increasing k can significantly increase the number of formulas that must be timed.

As another generalization, k-best dynamic programming keeps track of the k best formulas for each transform and size [20], [21]. This softens the dynamic programming assumption, allowing for the fact that a sub-optimal formula for a given transform and size might be the optimal way to split such a node in a larger tree. Unfortunately, moving from standard 1-best to just 2-best more than doubles the number of formulas that must be timed.

While dynamic programming has been frequently used, it is not known how far from optimal it is at larger sizes where it can not be compared against exhaustive search. Other search techniques with different biases will explore different portions of the search space. This exploration may find faster formulas than dynamic programming finds or provide evidence that the dynamic programming assumption holds in practice.

A different search technique is to generate a fixed number of random formulas and time each. This approach assumes that while the running times of different formulas may vary considerably, there is still a sufficiently large number of formulas that have running times close to the optimal. Evolutionary techniques provide a refinement to the previous approach [22]. Evolutionary algorithms add a bias to random search directing it toward better formulas.

B. Evolutionary Algorithm STEER for the WHT

We developed an evolutionary algorithm named STEER (Split Tree Evolution for Efficient Runtimes) to search for optimal signal transform formulas. Our first implementation of STEER explicitly only searched for optimal WHT formulas. This section describes STEER for the WHT, while Section IV-D describes our more recent implementation of STEER that will work for a variety of transforms.

Given a particular size, STEER generates a set of random WHT formulas of that size and times them. It then proceeds through evolutionary techniques to generate new formulas and to time them, searching for the fastest formula. STEER is similar to a standard genetic algorithm [22] except that STEER uses split trees instead of a bit vector as its representation. At a high level, STEER proceeds as follows:

1. Randomly generate a population P of legal split trees of a given size.
2. For each split tree in P, obtain its running time.

3. Let $P_{fastest}$ be the set of the b fastest trees in P.

4. Randomly select from P, favoring faster trees, to generate a new population P_{new}.

5. Cross-over c random pairs of trees in P_{new}.

6. Mutate m random trees in P_{new}.

7. Let $P \leftarrow P_{fastest} \cup P_{new}$.

8. Repeat step 2 and following.

All selections are performed with replacement so that P_{new} may contain many copies of the same tree. Since obtaining a running time is expensive, running times are cached and only new split trees in P at step 2 are actually run.

B.1 Tree Generation and Selection

Random tree generation produces the initial population of legal split trees from which the algorithm searches. To generate a random split tree, STEER generates a set of random leaves and then combines these randomly to generate a full tree.

To generate the new population P_{new}, trees are randomly selected from P using fitness proportional reproduction which favors faster trees. Specifically, STEER selects from P by randomly choosing any particular tree with probability proportional to one divided by the tree's running time. This method weights trees with faster running times more heavily, but allows slower trees to be selected on occasion.

B.2 Crossover

In a population of legal split trees, many of the trees may have well optimized subtrees, even while the entire split tree is not optimal. Crossover provides a method for exchanging subtrees between two split trees, allowing for one split tree to potentially take advantage of a better subtree found in another split tree [22].

Crossover on a pair of trees t_1 and t_2 proceeds as follows:

1. Let s be a random node size contained in both trees.

2. If no s exists, then the pair can not be crossed-over.

3. Select a random node n_1 in t_1 of size s.

4. Select a random node n_2 in t_2 of size s.

5. Swap the subtrees rooted at n_1 and n_2.

For example, a crossover on trees (a) and (b) at the node of size 6 in Figure 5 produces the trees (c) and (d).

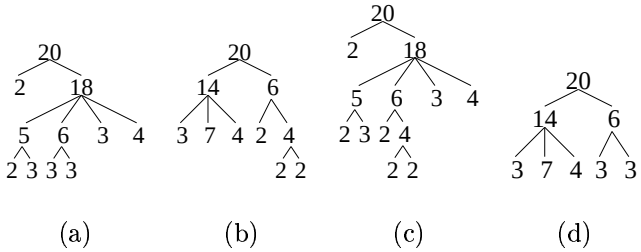


Fig. 5. Crossover of trees (a) and (b) at the node of size 6 produces trees (c) and (d) by exchanging subtrees.

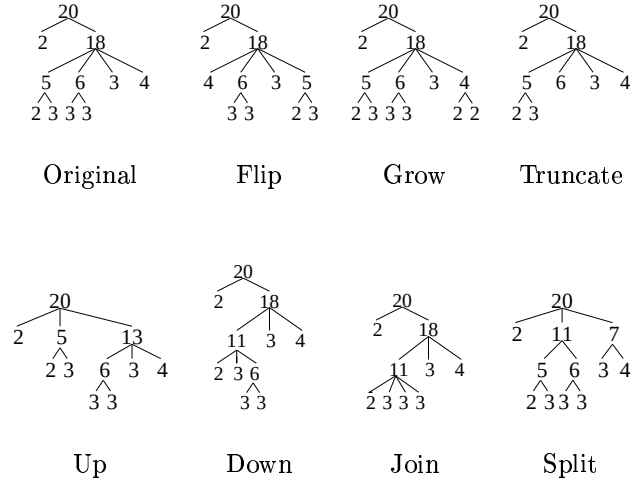


Fig. 6. Examples of each kind of mutation, all performed on the tree labeled "Original."

B.3 Mutation

Mutations are changes to the split tree that introduce new diversity to the population. If a given split tree performs well then a slight modification of the split tree may perform even better. Mutations provide a way to search the space of similar split trees [22].

We present the mutations that STEER uses with the WHT. Except for the first mutation, all of them come in pairs with one essentially doing the inverse operation of the other. Figure 6 shows one example of each mutation performed on the split tree labeled "Original." The mutations are:

- Flip: Swap two children of a node.
- Grow: Add a subtree under a leaf, giving it children.
- Truncate: Remove a subtree under a node that could be a leaf, making the node a leaf.
- Up: Move a node up one level in depth, causing the node's grandparent to become its parent.
- Down: Move a node down one level in depth, causing the node's sibling to become its parent.
- Join: Join two siblings into one node which has as children all of the children of the two siblings.
- Split: Break a node into two siblings, dividing the children between the two new siblings.

B.4 Running STEER

Figure 7 shows a typical plot of the running time of the best formula (solid line) and the average running time of the population (dotted line) as the population evolves. This particular plot is for $WHT(2^{22})$ on a Pentium III 450 MHz running Linux 2.2.5-15. The WHT package was compiled using gcc version egcs-2.91.66 with the options "`-O6 -fomit-frame-pointer -malign-double`." The average running time of the first generation that contains random formulas is more than twice the running time of the best formula at the end, verifying the wide spread of running times of different formulas. Further, both the average and best running times decrease significantly over time, indicating that the evolutionary operators are finding better formulas.

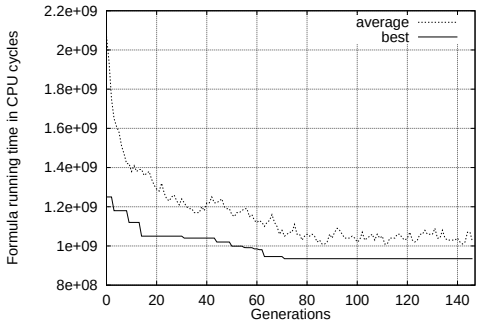


Fig. 7. Typical plot of the best and average running time of formulas as STEER evolves the population.

C. Search Algorithm Comparison for WHT

Figure 8 shows two different runs of binary dynamic programming on the same Pentium III. For sizes larger than 2^{10}, many of the formulas found in the second run are more than 5% slower than those found in the first run. An analysis of this and several other runs shows that the major difference is what split tree is chosen for size 2^4. The two fastest split trees for that size have close running times. Since the timer is not perfectly accurate, it times one split tree sometimes faster and sometimes slower than the other from run to run. However, one particular split tree is consistently faster than the other when used in larger sizes. Thus, a poor choice early in the search can produce significantly worse results at larger sizes.

Figure 9 compares the best running times found by a variety of search techniques on the same Pentium III. In this particular run, plain binary dynamic programming chose the better formula for size 2^4 and performs well. All of the search techniques perform about equally well except for the random formula generation method which tends to perform significantly worse for sizes larger than 2^{15}, indicating that some form of intelligent search is needed in this domain and that blind sampling is not effective.

Figure 10 compares the number of formulas timed by each of the search methods. A logarithm scale is used along

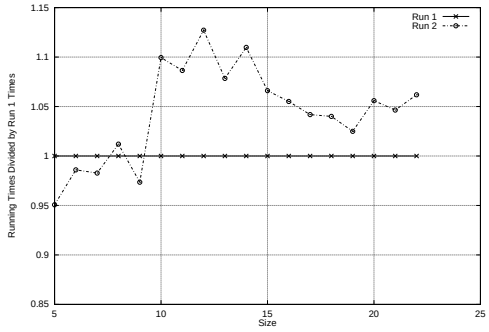


Fig. 8. Comparison of two runs of dynamic programming.

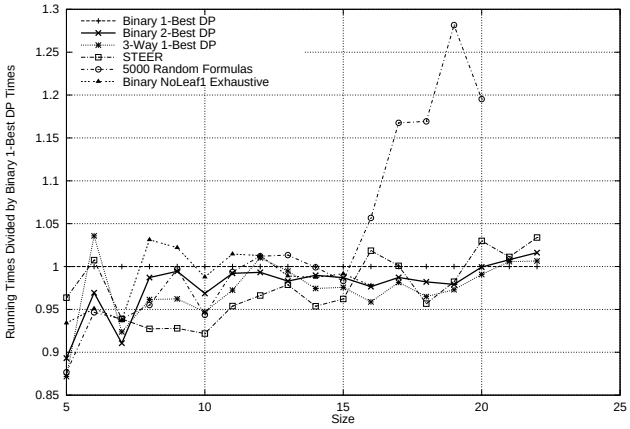


Fig. 9. Comparison of best WHT running times.

the y-axis representing the number of formulas timed. Effectively all of the time a search algorithm requires is spent in running formulas. The random formula generation method sometimes times less formulas than were generated if the same formula was generated twice. The number of formulas timed by the exhaustive search method grows much faster than all of the other techniques, indicating why it quickly becomes intractable for larger sizes. Clearly plain binary dynamic programming has the advantage that it times the fewest formulas.

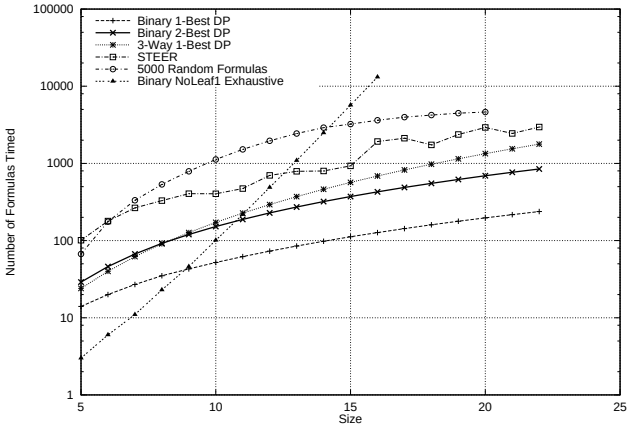


Fig. 10. Comparison of number of WHT formulas timed.

Of all the search methods compared here, dynamic programming both finds fast WHT formulas and times relatively

few formulas. However, we have also shown that dynamic programming can perform poorly if it chooses a poor formula for one of the smaller sizes. In addition, we have shown that STEER can also find fast WHT formulas.

D. Optimization for Arbitrary Transforms

We now turn to searching for optimal formulas for arbitrary transforms, including new user specified transforms. To do this, we have implemented the different search algorithms in the SPIRAL system (see Section II-B).

Dynamic programming and k-best dynamic programming are fairly straightforward to implement in this new setting. Given a transform to optimize, dynamic programming uses every applicable break down rule to generate a list of possible sets of children. For each of these children, it then recursively calls dynamic programming to find the best split tree(s) for the children, memoizing the results. Each of these possible sets of children are used to form an entire split tree of the original transform. These new split trees are then timed to determine the fastest.

STEER as described above used many operators that heavily relied on properties of the WHT. We have adapted STEER to be applicable across a variety of different signal processing transforms. The following changes were made:

- Random Tree Generation. A new method for generating a random split tree was developed. For a given transform, a random applicable break down rule is chosen, and then a random set of children are generated using the break down rule. This is then repeated recursively for each of the transforms represented by the children.
- Crossover. Crossover remains the same except the definition of equivalent nodes changes. Now instead of looking for split tree nodes of the same size, crossover must find nodes with the same transform and size.
- Mutation. We developed a new set of mutations since all of the previous ones were specific to the WHT. We have developed three mutations that work in this general setting without specific knowledge of the transforms or break down rules being considered. They are:
  - Regrow: Remove the subtree under a node and grow a new random subtree.
  - Copy: Find two nodes within the split tree that represent the same transform and size. Copy the subtree underneath one node to the subtree of the other.
  - Swap: Find two nodes within the split tree that represent the same transform and size. Swap the subtrees underneath the two nodes.

Figure 11 shows an example of each of these mutations. These mutations are general and will work with arbitrary user specified transforms and break down rules.

We ran dynamic programming and STEER on the same Pentium III to find fast implementations of DCTs. The formula compiler in the SPIRAL system generated Fortran code which was then compiled using g77 version egcs-2.91.66

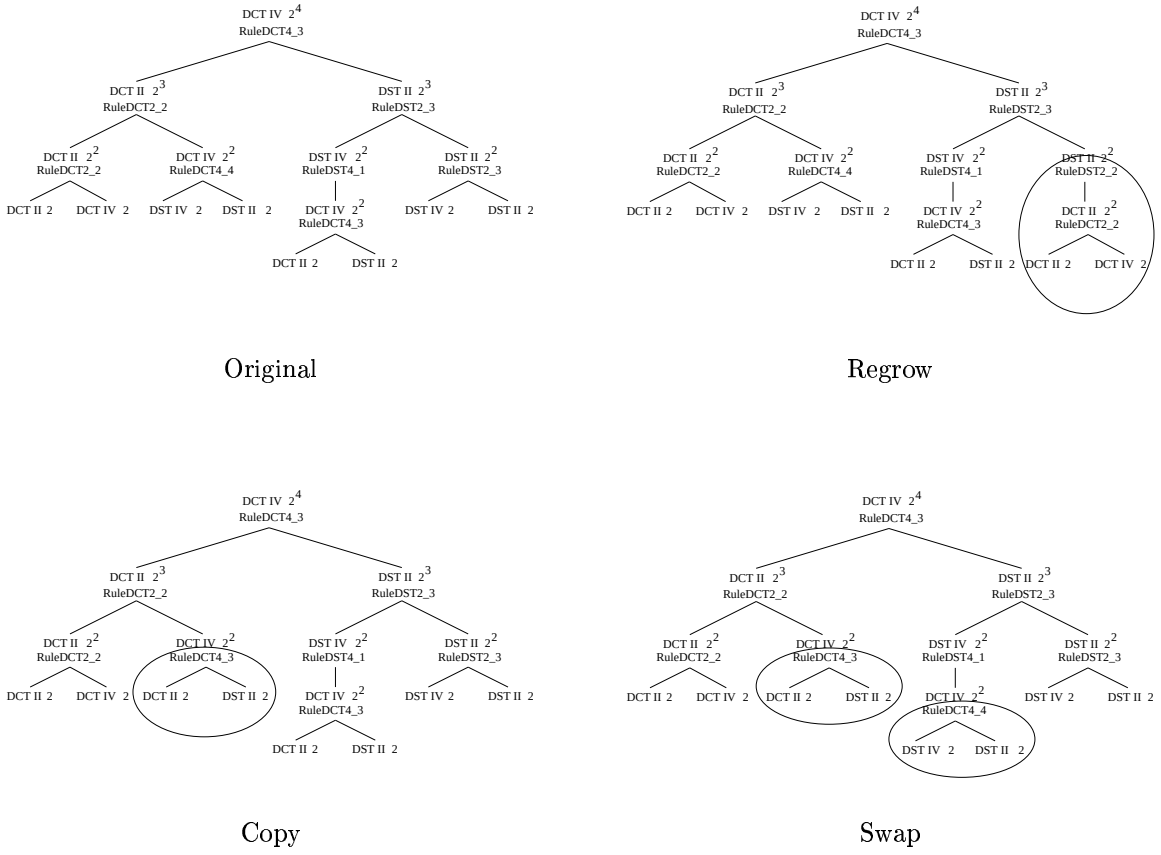


Fig. 11. Examples of each kind of general mutation, all performed on the tree labeled "Original." Areas of interest are circled.

with the option "`-O`." Figure 12 shows the running times of the best formulas found by each algorithm for DCT IV across three sizes. Figure 13 shows the same but for the four types of DCTs all of size 2^5. Both diagrams indicate that STEER finds the fastest formulas, finding formulas that are 10–20% faster than that found by 1-best dynamic programming.

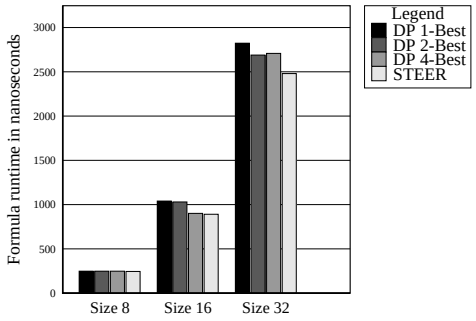


Fig. 12. Best DCT type IV times across several sizes.

For DCT type IV size 2^4, exhaustive search was also performed over the space of 31,242 different formulas that can be generated with our current set of break down rules. Figure 14(a) shows the running times of the fastest formulas found by each search algorithm. Figure 14(b) shows the number of formulas timed by each of the search algorithms. The formulas found by the exhaustive search and by STEER are about equally fast while the one found by dynamic

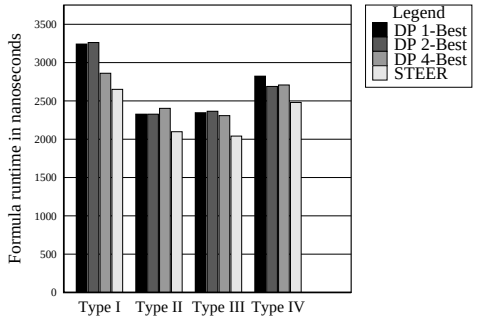
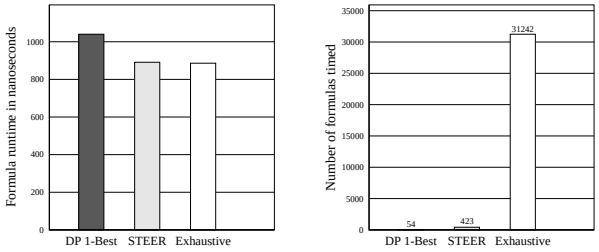


Fig. 13. Best DCT size 2^5 times across 4 types.



(a) (b)

Fig. 14. Search comparison for DCT type IV size 2^4.

programming is 17% slower. However, dynamic programming times very few formulas, and STEER times about 2 orders of magnitude less than the exhaustive search.

We have been able to develop search algorithms that can work in a general environment with a variety of transforms and break down rules. Simple algorithms such as dynamic programming do reasonably well, but STEER does better. Further, STEER was able to find a formula as fast as the one found by exhaustive search while timing significantly less formulas.

E. Summary

This section described different search methods for optimizing signal transforms, including a new evolutionary algorithm STEER. We have compared the different methods for the WHT and found that dynamic programming can find good formulas while timing relatively few. STEER does equally well as dynamic programming, but times more formulas than dynamic programming. However, we have also shown that sometimes a poor early choice can lead dynamic programming to do significantly worse. STEER does not rely on a few critical decisions early in the search process as does dynamic programming.

Further, we have discussed modifications to these algorithms to allow them to be applied to a variety of signal transforms. We have shown that STEER finds faster DCT formulas than dynamic programming while still timing significantly less formulas than exhaustive search. In at least one case, STEER was able to find a DCT formula that

runs about equally as fast as the optimal one found by exhaustive search.

V. Conclusions

We have developed automated methods for modeling and optimizing the performance of signal transforms. To model performance, we have explored over 15 feature sets to describe mathematical formulas. We identified feature sets with different abilities to partition formulas according to their running times. By describing formulas with features, we can present formulas to a neural network. While performance varied, as expected, according to what set of features were used, we showed that a neural network can learn to accurately predict the faster of two formulas or the running time of a formula given a limited set of training data.

Further, we have introduced a stochastic evolutionary search approach, STEER, for finding fast signal transform implementations. We have described the development of STEER both specifically for the WHT and for a wide variety of transforms. We have shown that STEER finds faster DCT formulas than dynamic programming while still timing significantly less formulas than exhaustive search. In at least one case, STEER was able to find a DCT formula that runs about equally as fast as the optimal one found by exhaustive search. While we have shown that a poor early choice can cause dynamic programming to perform sub-optimally, we have also given evidence that dynamic programming can sometimes perform well when searching for fast WHT formulas.

Acknowledgements

We would especially like to thank José Moura, Jeremy Johnson, Markus Püschel, and rest of the SPIRAL research group for their many helpful discussions on this research.

This research was sponsored by the DARPA Grant No. DABT63-98-1-0004 and by a National Science Foundation Graduate Fellowship. The content of the information in this publication does not necessarily reflect the position or the policy of the Defense Advanced Research Projects Agency or the US Government, and no official endorsement should be inferred.

References

[1] L. Auslander, Jeremy R. Johnson, and R. W. Johnson, "Automatic implementation of FFT algorithms," Tech. Rep. 96-01, Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA, June 1996.

[2] K. G. Beauchamp, *Applications of Walsh and Related Functions*, Academic Press, 1984.

[3] Jeremy Johnson and Markus Püschel, "In search of the optimal Walsh-Hadamard transform," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 2000, pp. 3347–3350.

[4] J. M. F. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, and M. M. Veloso, "SPIRAL: Portable Library of Optimized Signal Processing Algorithms," 1998, `http://www.ece.cmu.edu/~spiral/`.

[5] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua, "SPL: A language and compiler for DSP algorithms," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001, pp. 298–308.

[6] K. R. Rao and P. Yip, *Discrete Cosine Transform*, Academic Press, Boston, 1990.

[7] S.C. Chan and K.L. Ho, "Direct methods for computing discrete sinusoidal transforms," *IEE Proceedings*, vol. 137, no. 6, pp. 433–442, 1990.

[8] Z. Wang, "Fast algorithms for the discrete w transform and for the discrete fourier transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-32, no. 4, pp. 803–816, 1984.

[9] Gilbert Strang, "The discrete cosine transform," *SIAM Review*, vol. 41, no. 1, pp. 135–147, 1999.

[10] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 1998, vol. 3, pp. 1381–1384.

[11] Matteo Frigo, "A fast Fourier transform compiler," in *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999, pp. 169–180.

[12] Jeff Bilmes, Krste Asanović, C. Chin, and Jim Demmel, "Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology," in *Proceedings of International Conference on Supercomputing*, July 1997, pp. 340–347.

[13] Jeff Bilmes, Krste Asanović, C. Chin, and Jim Demmel, "The PHiPAC v1.0 matrix-multiply distribution," Tech. Rep. TR-98-35, International Computer Science Institute, Berkeley, CA, 1998.

[14] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *SC'98: High Performance Networking and Computing: Proceedings of the 1998 ACM/IEEE SC98 Conference*, 1998.

[15] Michail G. Lagoudakis and Michael L. Littman, "Algorithm selection using reinforcement learning," in *Proceedings of the Seventeenth International Conference on Machine Learning*, San Francisco, 2000, pp. 511–518, Morgan Kaufmann.

[16] Eric A. Brewer, "High-level optimization via automated statistical modeling," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995, pp. 80–91.

[17] Bryan Singer and Manuela Veloso, "Learning to predict performance from formula modeling and training data," in *Proceedings of the Seventeenth International Conference on Machine Learning*, San Francisco, 2000, pp. 887–894, Morgan Kaufmann.

[18] Bryan Singer and Manuela Veloso, "Automated formula generation and performance learning for the FFT," Tech. Rep. CMU-CS-00-123, Computer Science Department, Carnegie Mellon University, 2000.

[19] Howard W. Johnson and C. Sidney Burrus, "The design of optimal DFT algorithms using dynamic programming," in *IEEE Transactions on Acoustics, Speech, and Signal Processing*, April 1983, vol. 31, pp. 378–387.

[20] Gavin P. Haentjens, "An investigation of Cooley-Tukey decompositions for the FFT," M.S. thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, May 2000.

[21] David Sepiashvili, "Performance models and search methods for optimal FFT implementations," M.S. thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, May 2000.

[22] David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, 1989.