# Learning to Predict Performance from Formula Modeling and Training Data

**Bryan Singer** BSINGER+@CS.CMU.EDU
**Manuela Veloso** MMV+@CS.CMU.EDU
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA

## Abstract

This paper reports on our work and results framing signal processing algorithm optimization as a machine learning task. A single signal processing algorithm can be represented by many different but mathematically equivalent formulas. When these formulas are implemented in actual code, they have very different running times. Signal processing optimization is concerned with finding a formula that implements the algorithm as efficiently as possible. Unfortunately, a correct mapping between a mathematical formula and its running time is unknown. However empirical performance data can be gathered for a variety of formulas. This data offers an interesting opportunity *to learn* to predict running time performance. In this paper we present two major results along this direction: (1) Different sets of features are identified for mathematical formulas that distinguish them into partitions with significantly different running times, and (2) A function approximator can learn to accurately predict the running time of a formula given a limited set of training data. Showing the impact of selecting different features to describe the input, this work contributes an extensive study on the role of learning for this novel task.

## 1. Introduction

In simple terms, signal processing includes the study of algorithms that take as an input a *signal*, as a numerical dataset and output a *transformation* of the signal that highlights specific aspects of the dataset. For example, the Fast Fourier Transform (FFT) takes as an input the values of a signal over time and returns the corresponding frequency variations.

In general, signal processing algorithms can be represented by a transform matrix which is multiplied by an input data vector to produce a desired output vector (Rao & Yip, 1990; Tolimieri et al., 1997). Signal processing is particularly challenging for very large datasets for which an implementation of the transform as a straightforward matrix multiplication would require expensive numerical manipulations. However, the transform matrices often can be factored into a product of structured matrices, allowing for faster implementations of signal processing algorithms. Furthermore, these factorizations can be represented by mathematical formulas and a single signal processing algorithm can be represented by many different, but mathematically equivalent, formulas (Auslander et al., 1996; Singer & Veloso, 2000). Interestingly, when these formulas are implemented in actual code and executed, they often have very different running times. Thus, a crucial problem is finding the formula that implements the signal processing algorithm as efficiently as possible (Moura et al., 1998).

Unfortunately, a correct mapping between a mathematical formula and its running time is unknown. But a large amount of data can be easily gathered on the empirical performance of a variety of formulas. This data offers an interesting opportunity to study the *role of learning* in performance prediction.

This paper reports on our work in formulating and representing formula performance prediction as a machine learning task. A major step involves the selection of features to represent the data. We present several possible sets of features extracted from the mathematical formulas. We show the significant impact of these different choices both in the partition of the performance data sets and in performance prediction. We support our work with extensive empirical studies that process real performance data from a complex signal processing transform, the Walsh-Hadamard Transform.

In summary, this paper contributes two main results in our work towards the ambitious goal of learning to predict performance of signal processing algorithms:

- Simple features describing formulas can be used to distinguish formulas with significantly different running times.
- A function approximator can learn to accurately predict the running time of a formula given a limited set of training data.

## 2. Walsh-Hadamard Transform

In this paper, we present results from our work with the Walsh-Hadamard Transform (WHT), one of several important and fundamental signal processing transforms (Johnson & Püschel, 2000). We had previously explored the Fast Fourier Transform (FFT) and obtained results similar to those presented here (Singer & Veloso, 2000).

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

$$WHT(2^n) = \bigotimes_{i=1}^{n} DFT(2),$$

$$DFT(2) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

and $\otimes$ is the tensor or Kronecker product (Johnson & Püschel, 2000). If $A$ is a $m \times m$ matrix and $B$ a $n \times n$ matrix, then $A \otimes B$ is the block matrix product

$$\begin{bmatrix} a_{1,1}B & \cdots & a_{1,m}B \\ \vdots & \ddots & \vdots \\ a_{m,1}B & \cdots & a_{m,m}B \end{bmatrix}.$$

For example,

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

While this provides a potential algorithm for computing the WHT, more efficient algorithms usually take advantage of the fact that larger WHTs can be calculated by combining smaller WHTs appropriately. Computing tensor products with large matrices can be very computationally expensive, to the point that doing more tensor products of smaller matrices is often considerably faster than computing fewer tensor products with larger matrices. So, signal processing optimization involves searching for optimal methods of decomposing large transforms into smaller ones.

For positive integers $n_i$ such that $n = n_1 + \cdots + n_t$, $WHT(2^n)$ 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 (Johnson & Püschel, 2000). This formula can then be recursively applied to each of the WHTs that appear on the right side. 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. See Figure 1 for two example split trees. Each node in the split tree indicates the size of the WHT (given simply as the exponent) at that level, and the children of a node indicate how the node's WHT is recursively computed. As an example, the split tree shown in (a) can be derived as follows:

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

with the split tree representing the final formula.

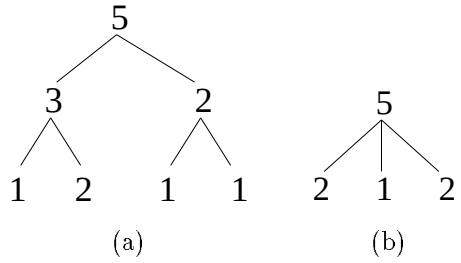


*Figure 1.* Two different split trees for $WHT(2^5)$

There are a very large number of possible split trees for a WHT of any given size, and thus there are large number of formulas equal to that WHT. Specifically, a WHT of size $2^n$ has on the order $\theta((4+\sqrt{8})^n/n^{3/2})$ different possible split trees (Johnson & Püschel, 2000). For example, $WHT(2^8)$ has 16,768 different split trees (Johnson & Püschel, 2000).

Since we had previously investigated the Cooley-Tukey expansion for the FFT which can be visualized with binary split trees (Singer & Veloso, 2000), this work reports only on binary WHT split trees. This helps to slightly reduce the total number of possible split trees, but there are still on the order of $\theta(5^n/n^{3/2})$ possible binary split trees (Johnson & Püschel, 2000).

The formulas that the different split trees express can have very different running times when implemented in code. Even what may seem like subtle changes such as swapping the left and right children of a node in the split tree changes the formula that the tree represents and can significantly change the running time. Split trees capture the order and sizes of the computations that are performed in a computer. Any change to the split tree will mean a change in the order and/or size of the computations performed. Unlike simple addition or multiplication, the different orderings and different sized subcomponents of these matrix operations can often produce very different running times on a computer. Ideally, we would like to be able to predict which of these formulas will have the fastest running time for any given computer on which it is run.

## 3. Relevant Features for Predicting Running Time

Given that there are many different expansions of large WHTs with different running times, we would like to find the one with the fastest running time. One simple approach would be to produce all of the formulas and to time each one on each different machine that we might be interested in. Then the formula with the fastest time can be determined for each machine.

There are two problems with this approach: (1) each formula may take a non-trivial amount of time to run, and (2) there are a very large number of formulas that need to be run. These problems make the approach intractable for WHTs of even fairly modest sizes.

In this paper, we present an approach to help solve the first problem. In particular, our approach is as follows:

- Generate a small set of formulas automatically and time each of these formulas.
- Describe the formulas by appropriate features.
- Use this data to learn to quickly and accurately predict the running times of the other formulas.

Elsewhere (Singer & Veloso, 2000), we present a method for generating formulas automatically in a principled way. In this section, we discuss and evaluate feature sets that will enable us to perform the learning experiments discussed in the next section.

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 begin by introducing several different feature sets to describe WHT formulas. After each of these different feature sets have been described, we then compare them along several measures to see how well the features can differentiate formulas with different running times.

### 3.1 Feature Sets

We begin by introducing a simple set of features to describe formulas, as well as a number of successive refinements of these features. In almost all of the features, we take advantage of the fact that we can visualize WHT formulas as split trees. Note that these features are not unique to the WHT and many have been used in our early study of the FFT (Singer & Veloso, 2000). However, we do take advantage of the fact that we have limited ourselves to 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 were 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.

#### 3.1.1 Counting Nodes

A general category of formula features we have explored is counting the number of nodes of various types. The following paragraphs describe several such features and Table 1 gives an example of each feature set for the split tree shown in Figure 1(a).

One simple and yet important set of features of a WHT split tree is the number and sizes of the leaves. These leaves correspond to the WHTs that must actually be computed directly and that appear in the formula represented by the split tree. Specifically, we count 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. We call this feature set "Leaf Nodes."

One modification of the above 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). We call this feature set "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 the previous two feature sets is to include both. We call this feature set "Leaf and All Nodes."

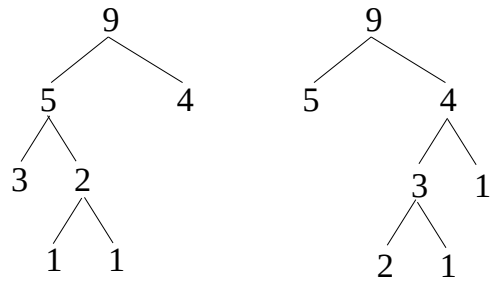


*Figure 2.* Two split trees with the same All Nodes counts but different Leaf Nodes counts

Consider again the first set of 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. In particular, we count 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 formula. We call this feature set "Left/Right Leaf 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, we can count the number of different sized left and right nodes appearing in the tree, excluding the root node. We call this feature set "Left/Right All Nodes."

*Table 1.* Example values of the different node count feature sets for the tree shown in Figure 1(a). Each row corresponds to a feature set and each column corresponds to a particular feature. For example, the column marked "leaf 1" is the feature representing the number of leaf nodes of size 1 (corresponding to $WHT(2^1)$) in the split tree. The entries in the table are the count of nodes of the given feature or an X if the feature is not present in the feature set.

| | 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 |
| Leaf | 3 | 1 | 0 | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| All | X | X | X | 3 | 2 | 1 | X | X | X | X | X | X | X | X | X | X | X | X |
| Leaf & All | 3 | 1 | 0 | 3 | 2 | 1 | X | X | X | X | X | X | X | X | X | X | X | X |
| L/R Leaf | X | X | X | X | X | X | 1 | 1 | 0 | 2 | 0 | 0 | X | X | X | X | X | X |
| L/R All | X | X | X | X | X | X | X | X | X | X | X | X | 1 | 2 | 0 | 2 | 0 | 1 |
| L/R Leaf & L/R All | X | X | X | X | X | X | 1 | 1 | 0 | 2 | 0 | 0 | 1 | 2 | 0 | 2 | 0 | 1 |

Once again, counting Left/Right All Nodes can't always distinguish two trees that counting Left/Right Leaf Nodes can distinguish. Thus, we can combine the two for a large set of features that include all those in the previous two sets. We call this feature set "Left/Right Leaf and Left/Right All Nodes."

#### 3.1.2 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.

A simple feature is the "leftness" or "rightness" of a tree. More formally, let the *leftness of a node* in a tree be the number of left children minus the number of right children 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. We call this single number feature "Leftness."

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

For example, the split tree shown in Figure 1(a) has nodes with leftness as shown in Figure 3, 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

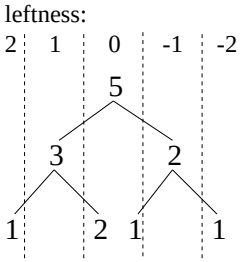


*Figure 3.* Leftness of nodes in tree of Figure 1(a)

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

For example, the split tree shown in Figure 1(a) 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

As has been suggested by previously described feature sets, it is possible to combine multiple feature sets to produce other feature sets. A few of these combinations will be shown in the following sections, and we have investigated many more than are reported here.

### 3.2 Evaluating Features

We now turn to evaluating the defined feature sets.

#### 3.2.1 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.

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.

*Table 2.* Number of partitions generated by different feature sets for all binary trees of different sized WHTs

| | WHT size | | | | | |
|---|---|---|---|---|---|---|
| Features | $2^5$ | $2^6$ | $2^7$ | $2^8$ | $2^9$ | $2^{10}$ |
| **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 |

Some results are shown in Table 2. 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 the table with node count features. In general, 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 a couple of the node count feature sets.

The lower portion of the table combines the All Nodes features with some of the shape features. The following sections will make it clear why the All Nodes features were chosen instead of some of the other node count features. Combining the leftness feature or the vertical profile greatly increases the number of partitions over the simple 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.

### 3.2.2 Relative Standard Deviation

While being able to partition a set of formulas into a large set of equivalence classes is important, ultimately we are only concerned that all of the formulas within a partition have roughly the same running time. 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 roughly the same running time. As a measure of this, we consider both the "weighted average relative standard deviation" and the maximum relative standard deviation.

For each partition we calculate the standard deviation of the running times of all the formulas that fall into that partition. We then calculate the relative standard deviation for each partition by dividing the standard deviation by the mean running time for that partition. To calculate the weighted average relative standard deviation, we then take a weighted average over all partitions, weighting each relative standard deviation by the number of formulas in the partition. See Table 3. The weighted average relative standard deviation indicates *on average* how close 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.

The weighted average and maximum relative standard deviations are shown in Table 4. For each WHT size shown in the tables, formulas for all possible split trees

*Table 3.* Calculating weighted average relative standard deviation

- Let $P_k$ be the set of formulas in partition $k$.
- Let $t_i$ be the running time of formula $i$.
- Let $m_k$ be the mean running time of the formulas in $P_k$. Then, $m_k = \frac{1}{|P_k|} \sum_{i \in P_k} t_i$.
- Let $\sigma_k$ be the standard deviation of the running times of the formulas in $P_k$. Then $\sigma_k = \sqrt{\frac{1}{|P_k|} \sum_{i \in P_k} (t_i - m_k)^2}$.
- Let $r_k$ be the relative standard deviation of the running times of the formulas in $P_k$. Then $r_k = \frac{\sigma_k}{m_k}$.
- Then, the Weighted Average Relative Standard Deviation is $\frac{\sum_k |P_k| r_k}{\sum_k |P_k|}$.

*Table 4.* 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 | 12.0 | 20.3 | 14.0 | 26.4 | 15.3 | 26.2 | 16.0 | 25.7 | 15.9 | 25.8 | 15.6 | 23.6 |
| All | 3.3 | 5.9 | 3.2 | 6.0 | 3.2 | 6.2 | 3.3 | 14.7 | 3.2 | 7.8 | 3.0 | 10.9 |
| Leaf & All | 3.3 | 5.9 | 3.2 | 6.0 | 3.2 | 6.2 | 3.3 | 14.7 | 3.2 | 7.8 | 3.0 | 10.9 |
| L/R Leaf | 9.0 | 20.5 | 12.0 | 25.5 | 13.8 | 25.8 | 14.8 | 25.5 | 14.9 | 26.5 | 14.8 | 24.6 |
| L/R All | 0.3 | 2.2 | 0.4 | 2.7 | 0.6 | 5.3 | 0.8 | 10.8 | 0.9 | 16.7 | 0.9 | 23.7 |
| L/R Leaf & L/R All | 0.1 | 2.1 | 0.2 | 2.0 | 0.3 | 5.3 | 0.4 | 10.2 | 0.5 | 19.2 | 0.6 | 26.0 |
| **Shape:** | | | | | | | | | | | | |
| Leftness | 31.9 | 55.4 | 35.0 | 55.7 | 33.8 | 52.9 | 32.9 | 40.1 | 31.8 | 41.2 | 30.4 | 37.2 |
| Vert Prof | 10.1 | 19.5 | 12.9 | 23.3 | 14.2 | 25.8 | 14.9 | 28.1 | 14.9 | 40.1 | 14.7 | 39.7 |
| Tot Path Len | 10.3 | 18.5 | 12.1 | 22.2 | 15.6 | 27.6 | 18.4 | 30.8 | 20.2 | 40.1 | 21.1 | 39.7 |
| Avg Path Len | 10.3 | 18.5 | 13.5 | 22.4 | 13.6 | 24.4 | 13.1 | 26.5 | 12.3 | 40.1 | 11.8 | 39.7 |
| Horz Prof | 10.3 | 18.5 | 12.1 | 22.2 | 12.4 | 24.4 | 12.4 | 26.5 | 11.9 | 40.1 | 11.4 | 39.7 |
| Vert & Horz Prof | 9.6 | 19.5 | 11.5 | 23.3 | 11.9 | 25.8 | 12.0 | 28.1 | 11.5 | 40.1 | 11.1 | 39.7 |
| **Composite:** | | | | | | | | | | | | |
| All Nodes & Leftness | 1.6 | 5.4 | 1.3 | 5.2 | 1.5 | 4.7 | 1.7 | 26.0 | 1.7 | 18.4 | 1.8 | 26.9 |
| All Nodes & Vert Prof | 1.6 | 5.4 | 1.3 | 5.2 | 1.3 | 4.7 | 1.4 | 26.0 | 1.4 | 18.4 | 1.4 | 19.0 |
| All Nodes & Tot Path Len | 3.2 | 5.9 | 3.2 | 6.0 | 3.1 | 6.2 | 3.2 | 14.7 | 3.1 | 13.6 | 2.9 | 13.1 |
| All Nodes & Horz Prof | 3.2 | 5.9 | 3.2 | 6.0 | 3.1 | 6.2 | 3.2 | 14.7 | 3.1 | 13.6 | 2.9 | 13.1 |
| All Nodes, Vert & Horz Prof | 1.6 | 5.4 | 1.3 | 5.2 | 1.3 | 4.7 | 1.3 | 26.0 | 1.3 | 18.4 | 1.3 | 19.0 |
| All Formulas | 43.0 | 43.0 | 40.0 | 40.0 | 36.5 | 36.5 | 34.7 | 34.7 | 32.7 | 32.7 | 31.1 | 31.1 |

were generated. These formulas were timed using a WHT package (Johnson & Püschel, 2000) on a Pentium II running Linux.

Looking at the top portion of the table, we see that using the Left/Right features tends to improve the weighted average over the plain features while causing mixed results for the maximum. The features that look at all nodes significantly outperform those just using the leaves. The middle portions of the tables show 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 but at the expense of sometimes increasing the maximum. Note that the plain All Nodes features and the Leaf and All Nodes features consistently have amongst the lowest maximums.

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 some of the best relative standard deviation results, but they do so with relatively few partitions.

## 4. Learning to Predict Running Times

With the features discussed in the previous section and with some training data obtained by timing a few formulas, we can use machine learning techniques to produce a function approximator that can quickly predict the running times of new formulas. Note that this still does not solve the problem of searching through a large space of potential formulas. However, we can now obtain a predicted running time 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.

### 4.1 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 a WHT package (Johnson & Püschel, 2000) on a Pentium II running Linux.

We used a back-propagation neural network as the function approximator. For all of the results presented, we used 25 hidden units, a learning rate 0.01 and a momentum of 0.001. These parameters obviously 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).

The various node count feature sets were used as inputs to the neural network. The set of formulas were partitioned into training 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 and testing sets.

As was suggested earlier, neural networks were trained on two different tasks. In the first task, neural networks were trained to predict the running times of formulas. In the second task, different neural networks were trained to predict which of two formulas would run faster.

### 4.2 Results

Results are shown in Table 5. The column marked "Cost" reports the running time prediction error on the test set. In particular, it is calculated by dividing the absolute difference between predicted running time and actual running time by the actual running time, and then averaging over all formulas in the test set:

- Let $c_i$ be the actual running time of formula $i$.
- Let $p_i$ be the predicted running time of formula $i$.
- Then the average percent error on predicting cost is $\frac{\sum_{i \in test-set} \frac{|c_i - p_i|}{c_i}}{|test-set|}$.

The column marked "Faster" corresponds to predicting the faster of two formulas. This column reports the prediction error on a random sampling of pairs of formulas in the test set. In particular, the number of samplings was 100 times the number of formulas in the test set. The percentage was calculated by taking the number of pairs of formulas the network predicted incorrectly which ran faster and dividing it by the total number of pairs of formulas tested. The "Cost" and "Faster" columns should not be directly compared as they report different measures of performance.

*Table 5.* 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. Note that the "Cost" and "Faster" columns should not be directly compared as they report different measures of performance. The size of the training and test sets are shown in percentages.

| Features | Train | Test | Cost | Faster |
|---|---|---|---|---|
| Leaf | 100 | 100 | 14.21 | 28.72 |
| | 75 | 25 | 14.69 | 28.32 |
| | 50 | 50 | 14.61 | 28.63 |
| | 25 | 75 | 14.61 | 28.40 |
| | 10 | 90 | 14.94 | 28.33 |
| All | 100 | 100 | 2.74 | 5.33 |
| | 75 | 25 | 3.04 | 5.31 |
| | 50 | 50 | 3.15 | 5.39 |
| | 25 | 75 | 3.40 | 5.41 |
| | 10 | 90 | 3.95 | 5.71 |
| Leaf & All | 100 | 100 | 2.79 | 4.79 |
| | 75 | 25 | 2.97 | 5.12 |
| | 50 | 50 | 3.15 | 5.31 |
| | 25 | 75 | 3.60 | 5.20 |
| | 10 | 90 | 4.32 | 5.45 |
| L/R Leaf | 100 | 100 | 13.69 | 25.92 |
| | 75 | 25 | 13.93 | 25.03 |
| | 50 | 50 | 14.18 | 25.06 |
| | 25 | 75 | 14.58 | 25.42 |
| | 10 | 90 | 15.52 | 24.98 |
| L/R All | 100 | 100 | 1.30 | 2.88 |
| | 75 | 25 | 1.68 | 3.00 |
| | 50 | 50 | 1.98 | 3.31 |
| | 25 | 75 | 2.71 | 3.51 |
| | 10 | 90 | 4.20 | 4.18 |
| L/R Leaf and L/R All | 100 | 100 | 1.22 | 2.50 |
| | 75 | 25 | 1.63 | 3.02 |
| | 50 | 50 | 1.79 | 3.04 |
| | 25 | 75 | 2.69 | 2.90 |
| | 10 | 90 | 3.91 | 3.57 |

The "Left/Right Leaf Nodes and Left/Right All Nodes" model yields the best learning results. These results were quite good with less than 4% error on predicting the faster of two formulas and less than 4% error on predicting the running times even when trained on only 10% of the formulas. The All Nodes model and the Leaf and All Nodes model, which were discussed earlier for their excellent performance at partitioning the formulas, also perform well here.

The Leaf Nodes and Left/Right Leaf Nodes models both perform significantly worse than all of the other models. This is not surprising, given that these two models had much larger weighted average relative standard deviations.

## 5. Conclusions

We have explored a wide variety of feature sets to describe mathematical formulas. We identified different feature sets with different abilities to partition formulas according to their running times. By describing formulas with features, we can present formulas to a function approximator. 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.

A few researchers have addressed similar goals. FFTW (Frigo & Johnson, 1998) uses dynamic programming to search for an optimal FFT implementation, assuming that the optimal implementation of a particular size FFT is still optimal if used as a subpart of a larger FFT. Outside the signal processing field, Brewer (1995) learned to predict the running times of various implementations of an algorithm. He only considers four different implementations as opposed to the thousands of formulas that we have considered. However, for each implementation, Brewer uses linear regression to predict that implementation's running time across different input sizes. In Brewer's framework, the user specifies the "terms" or features that are used in the linear regression. PHiPAC (Bilmes et al., 1997) and ATLAS (Whaley & Dongarra, 1998) use 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. Adding to these successful approaches, we have introduced the use of machine learning in this domain.

We are currently pursuing several lines of research that build upon the work presented in this paper, including:

- Determining how well a function approximator can interpolate and extrapolate to different size WHTs. Trained with expansions of $WHT(2^7)$ and $WHT(2^9)$, could a function approximator predict well for $WHT(2^8)$ or $WHT(2^{10})$?
- Investigating learning across machines and compilers. Can a function approximator learn to predict running times for different machines and compilers, given features of the machine and compiler?
- Investigating other signal processing algorithms besides the WHT and the FFT as well as considering non-binary WHTs.
- Finding a method to search through the extremely large number of possible formulas representing signal processing algorithms. Since it is not feasible to exhaustively generate all possible formulas for large transforms, we are developing heuristic methods for searching the space of formulas.

Further, this work could be extended to other mathematical algorithms outside of signal processing or even general algorithms.

## Acknowledgements

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

This research was sponsored by the DARPA Grant No. DABT63-98-1-0004. 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. The first author, Bryan Singer, is partly supported by a National Science Foundation Graduate Fellowship.

## References

Auslander, L., Johnson, J. R., & Johnson, R. W. (1996). *Automatic implementation of FFT algorithms* (Technical Report 96-01). Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA.

Bilmes, J., Asanović, K., Chin, C., & Demmel, J. (1997). Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. *Proceedings of the 1997 International Conference on Supercomputing* (pp. 340–347).

Brewer, E. A. (1995). High-level optimization via automated statistical modeling. *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (pp. 80–91).

Frigo, M., & Johnson, S. G. (1998). FFTW: An adaptive software architecture for the FFT. *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing* (pp. 1381–1384).

Johnson, J., & Püschel, M. (2000). In search of the optimal Walsh-Hadamard transform. *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing.* (In press).

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

Rao, K. R., & Yip, P. (1990). *Discrete cosine transform.* Boston: Academic Press.

Singer, B., & Veloso, M. (2000). *Automated formula generation and performance learning for the FFT* (Technical Report CMU-CS-00-123). Computer Science Department, Carnegie Mellon University.

Tolimieri, R., An, M., & Lu, C. (1997). *Algorithms for discrete Fourier transforms and convolution* (2nd edition). New York: Springer-Verlag.

Whaley, R. C., & Dongarra, J. J. (1998). Automatically tuned linear algebra software. *SC'98: High Performance Networking and Computing: Proceedings of the 1998 ACM/IEEE SC98 Conference.*