Learning to Generate Fast Signal Processing Implementations

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

Abstract

A single signal processing algorithm can be represented by many mathematically equivalent formulas. However, when these formulas are implemented in code and run on real machines, they have very different running times. Unfortunately, it is extremely difficult to model this broad performance range. Further, the space of formulas for real signal transforms is so large that it is impossible to search it exhaustively for fast implementations. We approach this search question as a control learning problem. We present a new method for learning to *generate fast formulas*, allowing us to intelligently search through only the most promising formulas. Our approach incorporates signal processing knowledge, hardware features, and formula performance data to learn to construct fast formulas. Our method learns from performance data for a few formulas of one size and then can construct formulas that will have the fastest run times possible across many sizes.

1. Introduction

Signal processing algorithms take as an input a *signal*, as a numerical dataset, and output a *transformation* of the signal that highlights specific aspects of the dataset. Many signal processing algorithms can be represented by a transformation matrix A which is multiplied by an input data vector X to produce the desired output vector $Y = A\,X$ (Rao & Yip, 1990). Naïve implementations of this matrix multiplication are too slow for large datasets or real time applications. However, the transformation matrices can be factored, allowing for faster implementations.

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). Interestingly, when these formulas are implemented in code and executed, they have very different running times. The complexity of modern processors makes it difficult to analytically predict or model by hand the performance of formulas. Further, the differences between current processors lead to very different optimal formulas from machine to machine. Thus, a crucial problem is finding the formula that implements the signal processing algorithm as efficiently as possible (Moura et al., 1998).

A few researchers have addressed similar goals. FFTW (Frigo & Johnson, 1998) uses binary dynamic programming to search for an optimal FFT implementation. We have previously shown that we can effectively learn to predict running times of Walsh-Hadamard Transform (WHT) formulas (Singer & Veloso, 2000), and we have also developed a stochastic evolutionary algorithm for finding fast implementations (Singer & Veloso, 2001). Other learning researchers select the optimal algorithm from a few algorithms. For example, Brewer (1995) uses linear regression to predict running times for four different implementations, and Lagoudakis and Littman (2000) use reinforcement learning for selecting between two algorithms to solve sorting or order statistic selection problems. We consider thousands of different algorithms in this work.

Accurate prediction of running times (Singer & Veloso, 2000) still does not solve the problem of searching a very large number of formulas for a fast, or the fastest, one. At larger sizes, it is infeasible to just enumerate all possible formulas, let alone obtain predicted running times for all of them in order to choose the fastest. In this paper we present a method that learns to *generate formulas* with fast running times. Out of the very large space of possible formulas, our method learns how to *control* the generation of formulas to produce the formulas with the fastest running times. Remarkably, our new method can be trained on data from a particular sized transform and still construct fast formulas across many sizes. Thus, our method can generate fast formulas for many sizes, even when not a single formula of those sizes has been timed yet.

Our work relies on a number of important signal processing observations. We have successfully integrated this domain knowledge into our learning algorithms. With these observations, we have designed a method for learning to accurately predict the number of cache misses incurred by a formula. Additionally, we can train this predictor using only data for formulas of one size while still accurately predicting the number of cache misses incurred by formulas of many different sizes. We then use this predictor along with a number of concepts from reinforcement learning to generate formulas that will have the fastest run times possible.

2. Signal Processing Background

This section presents some signal processing background necessary for understanding the remainder of the paper. For this work, we have focused on the WHT since it is one of the simpler and yet still important transforms (Johnson & Püschel, 2000). We plan to extend this approach to a wide variety of transforms.

2.1 Walsh-Hadamard Transform

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

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

and $\otimes$ is the tensor or Kronecker product (Beauchamp, 1984). 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

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

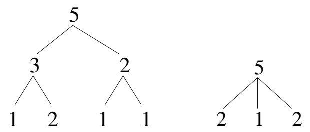
where I_k is the $k \times k$ identity matrix. This break down rule can then be recursively applied to each of these new smaller WHTs. Thus, $WHT(2^n)$ can be implemented as any of a large number of different but mathematically equivalent formulas.

2.2 Split Trees

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:

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

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.



(a) (b)

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

Implicit in the split tree is the *stride* at which nodes compute their respective WHTs. A node's stride determines how it accesses data from the input and output vectors. The exact nature of how nodes access data depending on their stride is not important for this paper. However, it is important that stride can greatly impact the performance of the cache. Two nodes of the same size but that have different strides can have very different cache performance. Further, the stride of a node depends on its location in the split tree and the size of the nodes to its right.

2.3 WHT Timing Package

For the results presented in this paper we used a WHT package, (Johnson & Püschel, 2000), which can implement in code and run WHT formulas passed to it. By using hardware performance counters, the package can count a number of different performance measures, including the number of cycles needed to execute the given formula or the number of cache misses incurred by the formula. The package can return performance measures for entire formulas or for each node in the split tree. 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, while internal nodes are implemented as recursive calls to their children.

We will be discussing the level 1 data cache which is the smallest and fastest memory cache closest to the CPU that contains data. Many processors also have a level 1 instruction cache to hold program instructions and a larger and slower level 2 cache closer to memory. The WHT package can specifically count the number of level 1 data cache misses. To simplify the discussion, we will just use the term "cache" throughout rest of this paper while we really mean "level 1 data cache."

2.4 Search Space

There is a very large number of possible formulas for a WHT of any given size. $WHT(2^n)$ has on the order of $\theta((4+\sqrt{8})^n/n^{3/2})$ different possible formulas (Johnson

& Püschel, 2000). For example, $WHT(2^8)$ has 16,768 different split trees. Thus, it is infeasible to exhaustively search through all possible split trees of even modest sizes.

In this work, we learn to generate fast formulas, allowing us to search through only the most promising formulas. However, evaluating our approach is difficult since the total space of formulas cannot be feasibly exhausted and thus the fastest formula is not known. We evaluate our approach by exhausting over a limited subset of the space of formulas that we have observed to be the most promising.

We have observed that the fastest binary split trees are just as fast as the fastest non-binary ones. However, there are still on the order of $\theta(5^n/n^{3/2})$ binary split trees (Johnson & Püschel, 2000), making it infeasible to search through all binary split trees for transforms larger than size of the cache. We have found that the fastest formulas never have leaves of size 2^1 since it is beneficial to use unrolled code of larger sizes. Searching over all split trees with no leaves of size 2^1 greatly reduces the search space, being feasible for formulas of sizes larger than the cache. Unfortunately, it still becomes infeasible to exhaust over this limited space for transforms much larger than 2^{16}. We have observed for Pentium machines that the best split trees are always rightmost (trees where every left child is a leaf). This limited space can be searched for larger sizes.

3. Key Signal Processing Observations

This section discusses several important observations that we made about the WHT that directed our research. We have incorporated this domain knowledge into our learning algorithms.

All the data presented here are for a Pentium III running Linux. The ideas and methods described here are general and should work across different architectures where similar observations can be made.

Figure 2 shows a scatter plot of run times versus cache misses for all binary $WHT(2^{16})$ split trees with no leaves of size 2^1. For each WHT formula, a dot appears in the plot corresponding to that formula's run time and cache misses. The plot shows that while there is a complete spread of run times, there is a grouping of formulas with similar numbers of cache misses. Both run times and cache misses vary considerably differing by about a factor of 6 and 10 respectively from the smallest to the largest. Further, as the number of cache misses decreases so does the minimal and maximal run times for formulas with the same number of cache misses. The formula with the fastest run time also has the minimal number of cache misses. So, if we can generate all of the formulas with minimal cache misses, we would have a much smaller set of formulas to time to determine the one with the fastest run time.

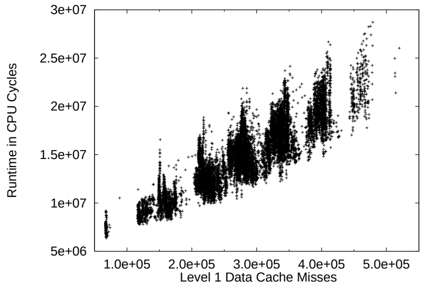


Figure 2. Run times vs. cache misses for $WHT(2^{16})$ formulas.

The second key observation is that all of the run time and cache misses occur in computing the leaves. Further, the total run time and number of cache misses of a formula is simply the sum of the run time and cache misses at each of the leaves. Thus, a prediction of the run time or cache misses of leaves leads to a prediction of the run time or cache misses of entire formulas.

Figure 3 shows a histogram of the number of cache misses incurred by *leaves* of all binary $WHT(2^{16})$ split trees with no leaves of size 2^1. For all the WHT formulas, the number of cache misses incurred by each leaf was measured, and a histogram was generated over all these leaves. The spikes in the histogram show that the number of cache misses incurred by leaves takes on only a few possible values. Thus, it is not necessary to predict a real valued number of cache misses, but just which of only a few groups of cache misses.

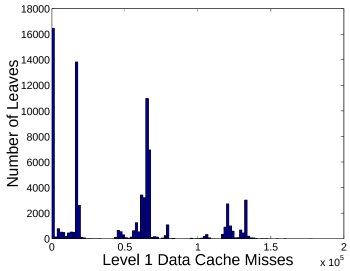


Figure 3. Histogram of the number of cache misses incurred by *leaves* of $WHT(2^{16})$ formulas.

Finally, we have observed both in Figure 3 and in a number of other similar histograms for different sized WHTs that the number of cache misses incurred by leaves occurs in specific fractions of the size of the transform. In an overall transform of size s, the number of cache misses a leaf will incur is $0, s/4, s, 2s$, or more. These particular fractions correspond to particular features of the cache. For example, it is possible to have $s/4$ cache misses since a cache line holds exactly four data items on this particular Pentium.

Thus, the number of cache misses incurred by a leaf comes in only a few specific fractions of the size of the transform being computed. This suggests the possibility to learn across different sized WHTs by predicting cache misses in terms of fractions of the transform size.

In summary, we observed:

- For a given size, the WHT formula with the fastest run time has the minimal number of cache misses. So minimizing cache misses produces a group of formulas containing the fastest one.
- All of the computational time and cache misses occur in the leaves of the split trees. So being able to predict for leaves allows predicting for entire formulas.
- The number of cache misses incurred by a leaf is only one of a few possible values. So we can learn categories instead of real-valued numbers of cache misses.
- The number of cache misses incurred by leaves are fractions of the transform size. So it may be possible to learn across different sizes.

4. Predicting Leaf Cache Misses

We now present our method to learn to predict cache misses for WHT leaves. We discuss the features used for the leaves, then we present the learning algorithm, and finally we evaluate our approach.

4.1 Features for WHT Leaves

To learn to predict cache misses for WHT leaves, we need features that will distinguish leaves with different cache misses. Our methods incorporate domain knowledge about the WHT by using good features.

Clearly the size of the leaf is important, but so is its position in the split tree. However, it is not as easy to capture the position of a leaf in a split tree as its size. A leaf's stride provides some information about its position in the split tree, as discussed in Section 2.2. The size and stride of the parent of a leaf indicates how much data the leaf will share with its siblings and how the data is laid out in memory.

Further, given a particular leaf l, the leaf p computed immediately before l gives information about l's position in the tree. Specifically, this "previous" leaf p is the leaf just to the right of l along the fringe of the tree. However, the size and stride of the common parent between l and p (i.e., the first common node in the parent chains of both leaves) provide information about how much data is currently in the cache because of p and how it is laid out in memory. Thus, we use the size and stride of the "common parent" in our features and not that of the previous leaf.

In summary, we use the following features:

- Size and stride of the given leaf
- Size and stride of the parent of the given leaf
- Size and stride of the common parent.

4.2 Learning Algorithm

Given these features for leaves, we can now use standard classification algorithms to learn to predict cache misses for WHT leaves. Our algorithm is as follows:

1. Run several different WHT formulas, collecting the number of cache misses for each of the leaves.
2. Divide the number of cache misses by the size of the transform, and classify them as:
   - near-zero if less than 1/8
   - near-quarter if less than 1/2
   - near-whole if less than 3/2
   - large otherwise.
3. Describe each of the leaves with the features outlined in the previous subsection.
4. Train a decision tree to predict one of the four classes of cache misses given the leaf features.

While the decision tree predicts one of the four categories for any leaf, this can be translated back into cache misses. In a transform of size s, a leaf is predicted to have:

- 0 cache misses, if near-zero is predicted;
- $s/4$ cache misses, if near-quarter is predicted;
- s cache misses, if near-whole is predicted;
- $2s$ cache misses, if large is predicted.

Further, the number of cache misses incurred by an entire formula can be predicted by summing over all the leaves.

4.3 Evaluation

There are several measures of interest for evaluating our learning algorithm. The simplest is to measure the accuracy at predicting the correct category of cache misses for leaves. Since we want to predict cache misses for an entire tree, another measure is to evaluate the accuracy of using this predictor for entire WHT formulas. Further, we are most interested in whether it accurately predicts the fastest formulas to have the fewest number of cache misses.

Table 1 evaluates the accuracy of our method at predicting the correct category of cache misses for leaves. In particular, we trained the decision tree on a random 10% of the leaves of all binary $WHT(2^{16})$ split trees with no leaves of size 2^1. We then tested this decision tree on leaves from different sized formulas, using all of the formulas of the different limited formula spaces discussed in Section 2.4. The error rate shown is the percentage of the total number of leaves tested for which the decision tree predicted the wrong category. Clearly, there are very few errors, less than 2% in all cases shown. This is surprisingly good in that while training only on a small fraction of the total leaves of one size, the learned decision tree can accurately predict across a wide range of sizes.

Table 1. Error rates for predicting cache miss category incurred by leaves.

Binary No-2^1-Leaf Size	Errors	Binary No-2^1-Leaf Rightmost Size	Errors
2^{12}	0.5%	2^{17}	1.7%
2^{13}	1.7%	2^{18}	1.7%
2^{14}	0.9%	2^{19}	1.7%
2^{15}	0.9%	2^{20}	1.6%
2^{16}	0.7%	2^{21}	1.6%

We used the same decision tree as in the previous experiments to predict cache misses for *entire* formulas by summing its predictions for each leaf within a split tree. We then calculated an average percentage error over a test set of formulas of a particular size as:

$$\frac{1}{|TestSet|} \sum_{i \in TestSet} \frac{|a_i - p_i|}{a_i},$$

where a_i and p_i are the actual and predicted number of cache misses for formula i.

Table 2 shows the error on predicting cache misses for entire formulas. Tables 1 and 2 cannot be directly compared, since Table 1 shows the number of leaves for which an error is made, while Table 2 shows the average amount of error between the real and predicted number of cache misses. Further, we would expect a larger error when predicting the actual number of cache misses for an entire formula instead of just one of four categories for a single leaf.

Except for the extreme sizes shown in Table 2, the learned decision tree is able to on average predict within 10% of the real number of cache misses. This is surprisingly good especially considering that Figure 2 shows that there is about a factor of 10 difference in the number of cache misses incurred by different formulas of the same size. Further, this result is very good considering that this is predicting for entire formulas and not just leaves and that the decision tree was only trained on data from formulas of size 2^{16}.

Table 2. Average percentage error for predicting cache misses for entire formulas.

Binary No-2^1-Leaf Size	Errors	Binary No-2^1-Leaf Rightmost Size	Errors
2^{12}	12.7%	2^{17}	8.2%
2^{13}	8.6%	2^{18}	8.2%
2^{14}	6.7%	2^{19}	7.9%
2^{15}	5.2%	2^{20}	8.1%
2^{16}	4.6%	2^{21}	10.4%

Ultimately we are only concerned with whether the fastest formulas are predicted to have the least number of cache misses. To test this, we have plotted the actual running times of formulas against the predicted number of cache misses. Figure 4 shows these plots for all the formulas within a restricted space for two different sized WHTs. The plots clearly show that the fastest formulas in both cases also have the fewest number of predicted cache misses. In addition, as the predicted number of cache misses increases, so do the running times of those formulas.

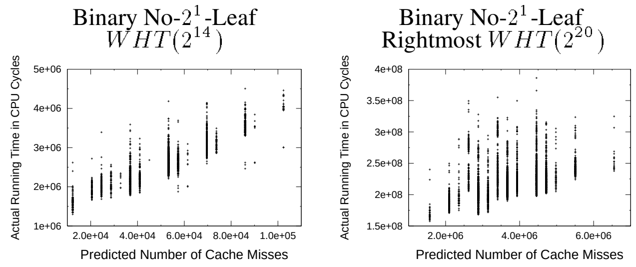


Figure 4. Runtime vs. predicted cache misses for entire formulas.

In summary, we have presented a method for predicting a formula's number of cache misses by training a decision tree to predict a leaf's number of cache misses to be one of only a few categories. We have also shown that this method produces very good results across sizes even when only trained on one particular size. This learned decision tree serves as a model of the cache performance of formulas.

5. Generating Fast Formulas

Now that we have a method for predicting cache misses for WHT formulas, we still have the problem that the space of all formulas is too large to exhaust over. We now use the prediction algorithm to learn to *generate fast formulas*. We describe and evaluate our control learning approach.

5.1 Approach

We approach the question of generating fast formulas as a control learning problem. We first try to formulate the problem in terms of a Markov decision process (MDP) and reinforcement learning. In the end, our formulation is not an MDP but does borrow many concepts from reinforcement learning.

5.1.1 Basic Formulation

An MDP is a tuple $(\mathcal{S}, \mathcal{A}, T, C)$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $T: \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a transition function that maps the current state and action to the next state, and $C: \mathcal{S} \times \mathcal{A} \to \Re$ is a cost function that maps the current state and action onto its real valued cost. Reinforcement learning provides methods for finding a policy $\pi: \mathcal{S} \to \mathcal{A}$ that selects the best action at each state that minimizes the (possibly discounted) sum of costs incurred.

Given a size, we want our algorithm to grow a WHT split tree for that size that runs as fast as possible. Let the states in the reinforcement learning problem be nodes in a split tree that have no children. Then the start state is just a root node of the given size with no children. The available actions in each state are the different ways to grow children for that node or to leave the node as a leaf (if the node's size is small enough). Ideally, the cost function should be:

- zero when giving children to a node, and
- the leaf's run time when making a node a leaf.

Then, the goal is to minimize the sum of the undiscounted costs over building an entire split tree. We want to use undiscounted costs since constructing a split tree only requires a finite number of steps.

5.1.2 Details and Difficulties

We need a state representation for the nodes within a split tree. We use a modified form of the leaf features described in Section 4.1 that expands the feature set to describe any node in a split tree. Specifically, we use the features:

- Size and stride of the given node
- Size and stride of the parent of the given node
- Size and stride of the common parent to this node.

The first two pairs of features are the same as for leaves except now they pertain to any node within the split tree. The concept of a node's common parent is more difficult to understand in this setting. The previous leaf of a node (that is the leaf computed immediately before computing the given node) is not always known in this setting. For example, if we expand the root node into two children, then the previous leaf of the left child is not known since the right child of the root may still be expanded.

However, it is still possible, without expanding the entire tree, to know what the common parent would be between any given node and its previous leaf that will be later constructed. In particular, the common parent of a node is its parent if the node is not the rightmost child, and otherwise the common parent of a node is its parent's common parent. The root node has no common parent, as there is no previous leaf computed before reaching the root node. A node's common parent will also be the common parent of the rightmost leaf in the given node's subtree.

Ideally, we would use the run time of leaves to determine the cost function. However, the previous sections discussed how we can easily learn to predict cache misses for leaves. We can approximate our desired cost function by using the learned classifier to predict cache misses for leaves. This change causes our method to construct formulas with minimal cache misses, instead of explicitly the fastest formulas. However, we have shown that the fastest formulas have the minimal number of cache misses.

Defining a transition function for this formulation is difficult. If two children of the root node are grown, then several questions arise, such as: which node is the next state, when will we transition back to the sibling node, and what should the transition function be from a leaf node? It is possible to answer these questions in specific ways, but then the Markov property may no longer hold. Lagoudakis and Littman (2000) discuss one approach for coping with this difficulty. They determine the Monte Carlo return for all but one of the next states, fixing the current policy. Then they continue learning on the one remaining next state. However, we can take a different approach, departing from the MDP framework, since we can formulate our problem to be deterministic and off-line.

Clearly actions are deterministic in that a node will always be given the children, if any, specified by the action. Further, the cost function is deterministic and known if we use the learned classifier to predict cache misses of leaves. We will define a value function over our states and show how it can be computed off-line.

5.1.3 Value Function

If a state must be a leaf, then its value is the predicted number of cache misses of this leaf. However, the optimal value of a node that could be an internal node or a leaf must consider both possibilities. If a state can have children, then we wish to find the subtree (possibly the subtree that simply makes the node a leaf) that has the minimal number of cache misses summed over all the leaves. That is,

$$V^*(state) = \min_{subtrees} \sum_{leaf \in subtree} CacheMisses(leaf).$$

We can rewrite this value function recursively in terms of the values of the children of the state. The optimal value of a state is the sum of the values of all the children in the optimal subtree rooted at this state, or the number of cache misses incurred if the optimal subtree is for the state to be a leaf. Mathematically, let the cache misses of a state be:

$$\begin{aligned} & LeafCM(state) \\ & \quad = \begin{cases} CacheMisses(state), & \text{if state can be a leaf} \\ \infty, & \text{if state cannot be a leaf} \end{cases} \end{aligned}$$

and the splitting value of a node be:

$$SplitV(state) \quad = \quad \min_{splittings} \sum_{child \in splitting} V(child),$$

where the minimum over splittings minimizes over all possible sets of children of a state and has a value of infinity if the state cannot have children. Then,

$$V^*(state) = \min\{LeafCM(state), SplitV(state)\}.$$

5.1.4 ALGORITHM

This later formulation of the value function suggests dynamic programming for computing it. For any state that could be a leaf, we can determine its value as a leaf by querying the classifier to get a predicted number of cache misses. For any state that could have children, the dynamic programming routine can then recursively call itself with each of the possible children, memoizing computed values for efficiency. By computing and memoizing *values of states*, dynamic programming performs significantly less computation than exhaustively constructing all possible split trees. The algorithm is as follows:

```
ComputeValues(State)
    if V(State) already memoized
        return V(State)
    Min = ∞
    if State can be a leaf
        Min = CacheMisses(State)
    for SetOfChildren in PossibleSetsOfChildren(State)
        Sum = 0
        for Child in SetOfChildren
            Sum += ComputeValues(Child)
        if Sum < Min
            Min = Sum
    V(State) = Min
    return Min
```

With the value function determined for all relevant states, the next step is to produce fast formulas. For a given size, the algorithm looks up the value of a root node of that size. It then considers all possible sets of children of the root node and determines their values. Any set of children whose sum of values equals the root node's value is then predicted to be one of the fastest way to split the root node. This procedure is then repeated for each child, building up a set of fast split trees. For simplicity, the algorithm is shown below only for binary trees:

```
FastTrees(State)
    Trees = {}
    if State can be a leaf
        if V(State) == CacheMisses(State)
            Trees = { Leaf(State) }
    for RightChild in PossibleRightChildren(State)
        LeftChild = MatchingChild(State,RightChild)
        if V(LeftChild) + V(RightChild) == V(State)
            for RightSubtree in FastTrees(RightChild)
                for LeftSubtree in FastTrees(LeftChild)
                    Trees = Trees ∪
                        { Node(LeftSubtree,RightSubtree) }
    return Trees
```

Leaf() creates a leaf node from the given state, Node() creates a split tree node with the corresponding subtrees as children, and MatchingChild() creates the state for the left child of the specified state when given the right child. Note that ComputeValues cannot easily keep track of the best split tree as there may be several formulas with the minimal number of cache misses. Due to the fact that we have made a number of approximations in our learning algorithms, it is possible that some error has been introduced. Thus, the above algorithm can be extended to allow for a tolerance, producing split trees that have up to the tolerance more cache misses than what is predicted to be optimal.

5.2 Evaluation

Evaluating our method is difficult in that it is not known what the optimal formula is for larger sizes. However, we can compare our algorithm against the best formulas found by searches over limited portions of the space as we did to evaluate the cache miss predictor. That is, for sizes 2^{16} and smaller we exhaust over all binary formulas with no leaves of size 2^1, and for sizes 2^{17} and larger we exhaust over all rightmost binary formulas with no leaves of size 2^1.

We have used our method to successfully generate fast formulas. In particular, we used the same decision tree learned in Section 4. Since that decision tree was trained on leaves from binary trees with no leaves of size 2^1, our algorithm only constructs binary trees with no leaves of size 2^1. This can be easily extended by training a decision tree on a broader class of formulas. Since many trees can have the same number of cache misses, it is not surprising that many states have the same value, and thus our algorithm produces several trees that it predicts to be fast.

Table 3 displays three different results for different sizes.

Table 3. Results from fast formula generation.

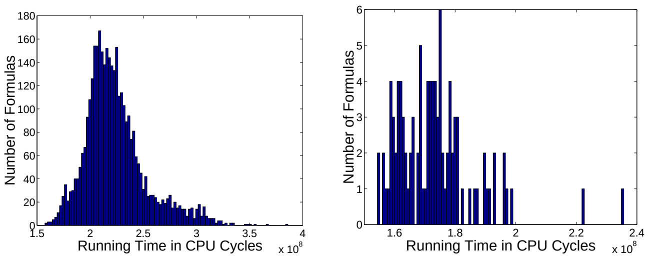
Size	Number of Formulas Generated	Generated Included the Fastest Known	Top N Fastest Known Formulas in Generated
2^{12}	101	yes	77
2^{13}	86	yes	4
2^{14}	101	yes	70
2^{15}	86	yes	11
2^{16}	101	yes	68
2^{17}	86	yes	15
2^{18}	101	yes	25
2^{19}	86	yes	16
2^{20}	101	yes	16

The first column shows how many formulas our method generated that it predicted to have the minimal number of cache misses. All of the formulas constructed have a very similar structure, allowing for the same number of formulas to be generated across many sizes. Note that this is a very small number compared to the thousands of formulas of the complete search space (see Section 2.4).

The second column checks whether the fastest formula found by a limited exhaustive search was among those constructed by our algorithm. We can see that, remarkably, the learning algorithm generates the fastest known formulas for all sizes, including sizes larger than the training size.

The third and last column shows the largest n where all n of the fastest formulas found by a limited exhaustive search were also generated by our method. For example, for size 2^{20}, our method constructed all of the first fastest 16 formulas found by a limited exhaustive search, but did not generate the 17th fastest formula. For all sizes, our method generates the fastest formula as well as many of the formulas that are very close to the fastest.

Figure 5 compares the histograms of running times for $WHT(2^{20})$ formulas generated by our method and for all rightmost binary $WHT(2^{20})$ formulas with no leaves of size 2^1. Notice the different scales along both axes. Clearly our method is constructing formulas with run times amongst the fastest found by the more exhaustive method.



(a) (b)

Figure 5. Histograms of run times for (a) all rightmost binary $WHT(2^{20})$ formulas with no leaves of size 2^1 and (b) the formulas generated by our method for $WHT(2^{20})$.

6. Conclusions

We have introduced a method for learning to generate fast implementations of signal processing algorithms. We have shown that this method can effectively learn to construct fast WHT formulas. Further, this method can generate fast WHT formulas across many sizes while only being trained on data from one particular size.

To support our control learning approach, we have also developed an accurate predictor of the number of cache misses that WHT formulas incur. This predictor is trained on data from one size and predicts well across many sizes.

We contributed an approach that, for the first time, allows the automatic generation of fast implementations of a particular signal processing transform (WHT) on a given computer (Pentium). We are extending this approach to work for other transforms and for other machines.

Acknowledgements

We would especially like to thank Jeremy Johnson, José Moura, and Markus Püschel for their many helpful discussions. 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, was partly supported by a National Science Foundation Graduate Fellowship.

References

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

Beauchamp, K. (1984). *Applications of walsh and related functions*. Academic Press.

Brewer, E. (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. (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* (pp. 3347–3350).

Lagoudakis, M., & Littman, M. (2000). Algorithm selection using reinforcement learning. *Proceedings of the Seventeenth International Conference on Machine Learning* (pp. 511–518).

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

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

Singer, B., & Veloso, M. (2000). Learning to predict performance from formula modeling and training data. *Proceedings of the Seventeenth International Conference on Machine Learning* (pp. 887–894).

Singer, B., & Veloso, M. (2001). Stochastic search for signal processing algorithm optimization. *Uncertainty in Artificial Intelligence*. Submitted.