Automating the Modeling and Optimization of the Performance of Signal Processing Algorithms

Bryan W. Singer

CMU-CS-01-156

December 2001

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by DARPA Grant No. DABT63-98-1-0004 and by a National Science Foundation Graduate Fellowship.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **DEC 2001**	2. REPORT TYPE	3. DATES COVERED **00-00-2001 to 00-00-2001**

4. TITLE AND SUBTITLE **Automating the Modeling and Optimization of the Performance of Signal Processing Algorithms**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**213**	

Keywords: Machine learning, signal processing, signal transform optimization, automatic performance tuning

Abstract

Many applications require fast implementations of signal processing algorithms to analyze data in real time or to effectively process many large data sets. Fast implementations of a signal transform need to take advantage of structure in the transformation matrix to factor the transform into a product of structured matrices. These factorizations compute the transform with fewer operations than the naïve implementation of matrix multiplication. Signal transforms can have a vast number of factorizations, with each factorization of a single transform represented by a unique but mathematically equivalent formula. Interestingly, when implemented in code, these formulas can have significantly different runtimes on the same processor, sometimes differing by an order of magnitude. Further, the optimal implementations differ significantly between processors. Therefore, determining which formula is the most efficient for a particular processor is of great interest.

This thesis contributes methods for automating the modeling and optimization of performance across a variety of signal processing algorithms. Modeling and understanding performance can greatly aid in intelligently pruning the huge search space when optimizing performance. Automation is vital considering the size of the search space, the variety of signal processing algorithms, and the constantly changing computer platform market.

To automate the optimization of signal transforms, we have developed and implemented a number of different search methods in the SPIRAL system. These search methods are capable of optimizing a variety of different signal transforms, including new user-specified transforms. We have developed a new search method for this domain, STEER, which uses an evolutionary stochastic algorithm to find fast implementations.

To enable computer modeling of signal processing performance, we have developed and analyzed a number of feature sets to describe formulas representing specific transforms. We have developed several different models of formula performance, including models that predict runtimes of formulas and models that predict the number of cache misses formulas incur.

Further, we have developed a method that uses these learned models to *generate* fast implementations. This method is able to construct fast formulas, allowing us to intelligently search through only the most promising formulas. While the learned model is trained on data from one transform size, our method is able to produce fast formulas across many transform sizes, including larger sizes, even though it has never timed a formula of those other sizes.

Acknowledgments

Whatever you do, work at it with all your heart, as working for the Lord, not for men. — Colossians 3:23

First, I would like to acknowledge that my ability and patience to complete this work comes from God. He has blessed me with the intellectual ability, the yearning for knowledge, and the environments to allow those to grow. It is for Him that I work and live.

Many thanks to my wife Kelly. She has stood by me and encouraged me throughout this process. She endured three years of long distance dating before I was able to move to Baltimore. Many, many thanks for being my faithful editor of nearly every paper I have written, including this one. Thanks also for patiently listening to so many practice talks.

My family has always been there for me. Mom, Dad, and Jason have encouraged me in my endeavors and always provided a place where I could be myself. Without their love, care, and encouragement, I could not be where I am today.

Manuela, thanks for all of your guidance and encouragement. Thanks for constantly encouraging me to write papers, to prepare my thesis proposal, and now to prepare for my thesis defense. It was by your guidance that I began looking at this interesting research area.

I would also like to thank rest of my thesis committee, Jeremy, Scott, and John. You have been very encouraging about this work and always willing to listen and to help.

It has been a joy to work with the SPIRAL research group. They have been very encouraging about my work and have been extremely helpful. A special thanks to Markus for all of his help with the formula generator and many other details, and to Jianxin for his many quick fixes and major improvements to the SPL compiler. Thanks also to José for his guidance and encouragement.

What would lunch at CMU be without Mike and Peter? Thanks guys for being great friends. Thanks also for all the miscellaneous help, from answering TeX questions to allowing me to bounce research ideas around.

Finally, I would like to thank Luis Torgo for his help in using RT4.0. He went out of his way to add some unusual functionality that I needed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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 Fourier Transform takes as an input the values of a signal over time and returns the corresponding frequency variations. Many signal processing algorithms can be represented by a transformation matrix which is multiplied by an input data vector to produce a desired output vector. That is, for an input vector X representing the signal, a signal transform produces the output vector $Y = A\,X$, where A is the transformation matrix (Nussbaumer, 1982; Rao and Yip, 1990; Tolimieri et al., 1997).

Signal processing is particularly challenging for large datasets for which an implementation of the transform as a straightforward matrix-vector multiplication would require $O(n^2)$ operations. However, the transformation matrices for signal transforms often can be factored into a product of structured matrices, allowing for faster implementations with $O(n \log n)$ operations. 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).

The number of different formulas for a given transform is often very large and grows with transform size. For example, with just a few different methods of factorization, the Fast Fourier Transform (FFT) has 258,400 different formulas for size 2^5 and 1.8×10^{13} for size 2^6. Again with just a few methods of factorization, the Discrete Cosine Transform (DCT) of type IV has 2.2×10^{306} different formulas for size 2^{10}.

Clearly, as the transform size increases, it becomes infeasible to even enumerate all possible formulas let alone time them.

Interestingly, when these formulas are actually implemented in code and executed, their runtimes can vary by a factor of 2 to 10. While many of the factorizations may produce the exact same number of arithmetic operations, the different orderings of the operations that the factorizations produce can greatly impact the performance of the formulas on modern processors. For example, different operation orderings can greatly impact the number of cache misses and register spills that a formula incurs or its ability to make use of the available execution units in the processor. The complexity of modern processors makes it difficult to analytically predict or model by hand the performance of formulas.

Figure 1.1 shows a histogram of runtimes for a set of 70,376 different FFT formulas of size 2^{18}. All of these formulas were run on the same Pentium III 450 MHz running Linux. The histogram shows a significant spread of runtimes, almost a factor of 4 from fastest to slowest. Further, it shows that there are relatively few formulas that are among the fastest.

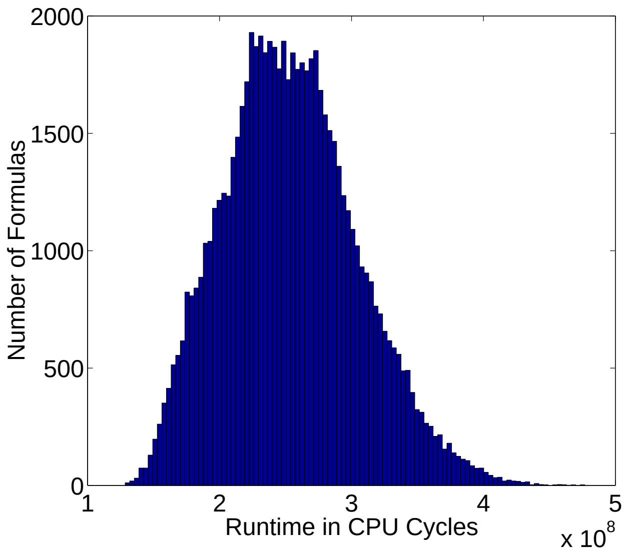


Figure 1.1: Histogram of runtimes for 70,376 different $FFT(2^{18})$ formulas.

The differences between current processors lead to very different optimal formulas from machine to machine. The optimal formula on one machine is very suboptimal on another machine. To make this point, Püschel et al. (2001b) searched for fast $FFT(2^{20})$ formulas on four different platforms. Then we timed these implementations on each of the other platforms. The results are displayed in Table 1.1. Each row corresponds to a timing platform; each column corresponds to a fast formula found for a particular machine. For example, the runtime of the formula found for the Pentium 4, timed on an Athlon is in row 3 and column 2. The fastest runtime for each machine corresponds to the formula found for that machine. Furthermore, for a given machine, the other formulas run significantly slower.

Table 1.1: Comparing fast $FFT(2^{20})$ implementations generated for different machines. The entries are runtimes given in seconds. (Püschel et al., 2001b)

		fast formula for			
		PIII	P4	Athlon	Sun
timed on	Pentium III 900 MHz	0.83	1.08	0.99	1.10
	Pentium 4 1.4 GHz	0.97	0.63	0.73	1.23
	Athlon 1.1 GHz	1.23	1.23	1.07	1.22
	Sun UltraSparc II 450 MHz	0.95	1.67	1.42	0.82

Given this complexity, hand tuning signal transform implementations for a given architecture is very difficult and time consuming for humans to perform. The size of the search space of possible formulas for a single transform is very large, and it is not easy to understand why one formula runs faster than another. As different transforms require different operations to be performed and data to be accessed differently, each transform requires a separate effort to hand tune. Compounding this problem is the fact that each new computer platform requires a completely new effort to understand the new architecture and to tune code for that platform.

1.1 Thesis Problem and Approach

The thesis question is:

> How can machine learning techniques automate the optimization of the performance of signal transform implementations?

And specifically how can machine learning techniques help identify what influences performance?

This thesis investigates how machine learning techniques can effectively analyze runtime performance data for different signal transform implementations to then aid in optimizing the signal transform. While it is difficult for a human to analyze and understand performance by hand, it is easy to collect runtime performance data for specific implementations of a given signal transform on a specific computing platform. This data provides an opportunity both for learning to model and predict runtime performance and for runtime feedback while searching for fast implementations. By collecting runtime performance data for different signal transform implementations, machine learning techniques can be used to automatically analyze this data and construct performance models that can predict runtime performance for implementations. By timing specific signal transform implementations, search methods can use this runtime performance data to guide the search towards faster implementations.

Thus, the problem that this thesis addresses is to find the best implementation for:

- a signal transform of interest,
- a size of interest for that transform,
- a computing platform for which the code is to be tuned, and
- a performance metric to be optimized.

We constrain this problem by assuming the following are given:

- a set of break down rules to factor the given transform, defining the space of formulas that can be considered,
- a method for implementing mathematical formulas representing transform factorizations into machine code for the given platform, possibly with parameters that influence the exact method of implementing the formulas, allowing another degree of freedom, and
- a method of obtaining the runtime performance for specific implementations on the given computing platform.

Instead of trying to optimize a single implementation for a signal transform across all possible sizes, we consider each different transform size to be a different problem. Each transform size has a different space of formulas for factoring a transform of that size. Further, the transform size can have a big impact on how different types of implementations may perform, particularly as the transform size crosses the sizes of different cache levels. However, we use machine learning techniques to learn performance models that can accurately predict across a range of sizes.

While most of the results presented in this thesis have concentrated on optimizing the runtime of implementations, the methods are general and can be used with any performance metric that can be measured. For example, in hardware design, other metrics such as power consumed or chip size may be of interest.

We have taken three different approaches to address this problem:

- Optimizing performance by searching for fast implementations. Our search methods generate a number of different implementations and run them on the given computing platform to determine their runtimes. The search methods then use this runtime information to determine new implementations to time, and the process is repeated.

- Modeling performance of different formulas. We have developed methods that are able to learn to predict performance of different formulas on a given computing platform.

- Generation of optimized implementations by using learned performance models. We have developed a method that is able to construct fast implementations of signal transforms without performing a search that requires timing formulas. Instead, our method uses learned performance models to guide it in controlling the construction of fast formulas.

Thus, we can optimize the performance of signal transform implementations by either performing a search in the space of implementations or by using our learned models of performance to guide the generation of fast implementations.

This research has been conducted as part of a larger research effort by the SPIRAL (Signal Processing algorithms Implementation Research for Adaptable Libraries) research group (Moura et al., 1998). The ultimate goal of the SPIRAL group is to develop adaptable, optimized libraries for signal processing algorithms. This thesis focuses on how artificial intelligence and particularly machine learning techniques can be used to further this goal.

1.2 Infrastructure

Instead of considering every arbitrary implementation of a signal transform, we have constrained our problem in two ways. First, we fixed a set of break down rules that our methods could use in factoring any given transform. This defines the space of possible formulas that can be considered. As we have already discussed, the number of possible formulas is huge, in some cases exceeding 10^{300} for a transform size of 2^{10}. Second, we have used software developed by others to translate mathematical formulas representing a transform factorization into machine code. This software defines how a formula is implemented on the computing platform of interest.

Some other members of the SPIRAL group produced a Walsh-Hadamard Transform (WHT) package (Johnson and Püschel, 2000). This package takes any WHT formula as input and is able to implement the WHT according to the factorization specified by the formula. The package is then able to run and time this implementation. Much of the work with the WHT in this thesis has used this package.

More recently, the SPIRAL group has also developed a system that can implement and time a wide variety of different signal transforms, including new user-specified transforms (Püschel et al., 2001b). Figure 1.2 gives an overview of the SPIRAL system. This system consists of four main components:

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, but also comes with a number of common transforms and break down rules already defined.

2. **Formula Generation.** The second step is to apply these break down rules repeatedly to produce a complete factorization of a given transform as a mathematical formula (Püschel et al., 2001a).

3. **Code Generation.** Given a formula, the third step is to implement it in executable code. This is done by compiling the formula into Fortran or C which is in turn compiled using the native compiler. This step allows for different portions of the code to be optionally unrolled into straight-line code (code without loops or function calls). This step can also measure the performance of the resulting implementation. (Xiong, 2001)

4. **Optimization.** The final step is to search for a fast implementation. This

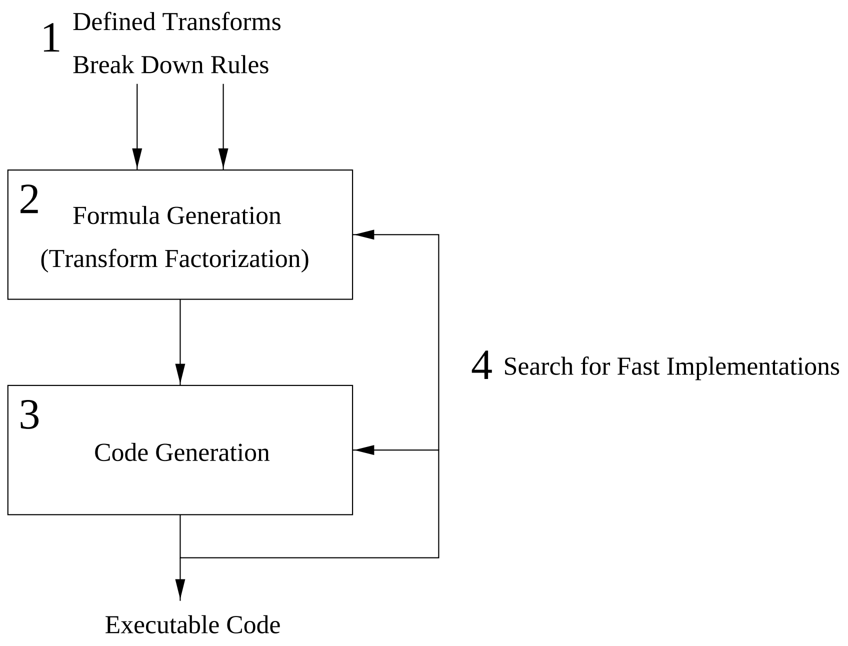


Figure 1.2: Overview of the SPIRAL system.

thesis contributes the development and implementation of the search engine in the SPIRAL system. While the formula and code generation modules were designed by others, this thesis contributes the entirety of this final optimization step.

The SPIRAL system has formed the infrastructure for rest of the thesis that explores transforms beyond just the WHT.

This infrastructure then defines the space of possible implementations considered for a given transform. It provides two different sources of degrees of freedom. First, there are the different possible formulas for a given transform. Second, with the SPIRAL system there are also options in what portions of the code are unrolled. Outside of these degrees of freedom, our work is constrained by this infrastructure. For example, if the code generation step in the SPIRAL system always produced very slow code, our work would never be able to find a really fast implementation, but only the fastest implementation that the code generation step allows. So, the goal of this thesis is to find the fastest implementations possible in the search space defined by the used infrastructure.

1.3 Performance Optimization by Searching

One of the major components of the SPIRAL system is the search engine which contains search methods for finding fast implementations of given transforms using the infrastructure that rest of the SPIRAL system provides. The search methods control the formula generation and code generation steps to produce an implementation which is run to determine its performance. This performance information is then used by the search engine to determine new implementations to generate and time. Since the user can specify new transforms and break down rules to the SPIRAL system, the search engine must be able to handle arbitrary transforms and break down rules.

We have developed a number of different search methods in the search engine, including exhaustive search, dynamic programming, random search, hill climbing search, STEER, and timed search. All of these search methods can optimize new user-specified transforms using user-specified break down rules. Not only do all of these search methods search over different factorizations for a given transform, but almost all of these search methods are also able to search over different code unrolling parameters that the SPIRAL system's code generation method allows.

Each of these search methods have a different bias that directs how it searches the space of possible implementations and how it uses feedback from timing implementations. Thus, a user can explore a variety of different search methods when trying to optimize a particular transform. Further, the timed search method that we developed allows the user to use multiple search methods while specifying a time limit for search.

We have developed a stochastic evolutionary search method called STEER for searching for fast implementations in this domain. STEER is similar to a genetic algorithm except that it uses a richer representation for individuals, namely a compact tree representation of a factorization. STEER generates a population of random implementations and then evolves this population using two operators. Mutation makes small changes to a factorization to produce another factorization. Crossover works between two different factorizations to exchange subformulas.

We have tested each of these search methods and have found that no one search method tends to outperform all of the others for all transforms and sizes. One of the advantages of the search engine is that many different search methods are provided and can be tried, allowing for faster implementations to be found than if a single search method was provided. However, we have found that for small sized Discrete

Trigonometric Transforms (DTTs), STEER outperforms the other search methods, finding formulas with runtimes up to 20% faster than the formulas found by the standard dynamic programming search method.

1.4 Performance Modeling

Given the infrastructure provided by the SPIRAL system or the WHT package, it is possible to generate many different implementations of a given transform and to obtain runtime performance data for those implementations on a given platform. While we have implemented a number of search methods for trying to find a fast implementation using this runtime data as feedback, we have also used this runtime data to automatically train machine learning methods to model performance of different formulas for a given transform. These performance models can then predict the performance of new formulas more quickly than an accurate measurement of runtime performance can be obtained.

One of the most difficult parts of using machine learning techniques to learn to predict performance for signal transform formulas was developing a good set of features to describe formulas. We have contributed a number of different feature sets in this thesis and have evaluated them. One very important step in developing good features was to to view the mathematical formulas using a compact tree representation of a factorization that we have called a split tree. This representation highlights only the most important aspects of the factorization while hiding some of the mathematical details. Thus, we focused on developing features that described split trees.

Another important problem was framing the exact machine learning task. We began by trying to train machine learning methods to predict performance for entire formulas and thus developed features for entire split trees. While having good success in doing this for transform sizes of about 2^{10} and smaller, we found that it did not work as well at larger sizes. Shifting our focus, we considered making predictions for subportions of formulas, specifically individual nodes in split trees. We found that by changing the machine learning task to predicting performance for individual nodes, we could still accurately predict for entire formulas by simply summing our predictions over all of the nodes. This change allowed us to predict accurately for much larger sizes such as 2^{20}.

While we have learned models for specific transforms and computing platforms, we have been able to learn models that accurately predict across transform sizes. Further, we can train these models using data from only one transform size and be able to accurately predict for both smaller and larger sizes. Since runtime performance data can be gathered more quickly for smaller sized transforms, being able to accurately predict for larger sizes while training on data from smaller sized transforms is particularly exciting.

1.5 Generating Fast Implementations

Given that we have been able to develop accurate performance models, the next step is to be able to use those performance models to aid in the optimization of signal transform implementations. While optimization by search requires implementing and timing each formula that it considers, the performance models offer the possibility to obtain predicted performance values for formulas without actually implementing and timing them. Unfortunately, there are still so many different formulas for a given transform that predicting for all them would be infeasible.

Ideally, we would like a method to be able to generate a formula with the fastest predicted runtime possible without enumerating all possible formulas. Specifically, we would like a method to learn how to *control* the generation of formulas so as to produce fast ones. Producing a formula for a signal transform involves a series of choices in how to factor that transform and the resulting factors recursively. Thus, we wish to devise a method that learns to control the generation of fast implementations by making the best choices possible in factoring the transforms.

By borrowing concepts from reinforcement learning, we have been able to develop a method for controlling the generation of formulas. Our method uses learned performance models to guide its choices, allowing it to generate fast formulas. Our method achieves excellent results, often producing the previously fastest known formula for a given transform and size within the first 50 formulas generated. Further, the runtime of the first formula that our method generates is often within 6% of the fastest known runtime.

Since the models being used can predict well across many transform sizes, our generation method can produce fast formulas also across many sizes. While some formulas of one size were timed to collect data to train the performance models, our method does not see timings for transforms of any other size and still can produce fast formulas for those sizes. Thus, our method pays a one time cost to collect data for one transform size to train a performance model and then can construct fast formulas for many different sizes, including larger sizes, without timing a single formula of those other sizes.

1.6 Thesis Contributions

This thesis makes three major contributions:

1. **Several search methods for finding fast implementations of a variety of signal transforms.**
   We have developed and implemented a variety of different search methods in the SPIRAL system, namely exhaustive search, dynamic programming, random search, hill climbing search, STEER, and timed search. These methods are able to automatically optimize any transform that can be specified to the system, including new user-specified transforms. We have specifically developed a new search method for this domain, namely an evolutionary stochastic search algorithm named STEER. Further, we have developed a meta-search algorithm that uses the other search algorithms to try to find the best implementation given a limited amount of time to search. In this thesis, we describe the development and implementation of these algorithms as well as present a comparison of their performance.

2. **Automatic methods for modeling and predicting performance of signal transforms.**
   This thesis presents a number of methods for automatically learning to predict performance of signal transforms. We show results for predicting both runtime and cache misses. Most of the techniques can be immediately used with any other performance measure as well. Two of the most difficult problems here were determining a good set of features to use and defining a good task for the machine learning algorithms to address. We have contributed several different feature sets and two very different approaches to defining signal transform performance prediction as a machine learning task.

3. **A method for automatically generating fast implementations.**
   We have developed a method that uses learned models of performance to gen-

erate fast WHT and FFT implementations. By using learned models of performance, our method is able to construct fast formulas for a given transform size, even though the method never times a single formula of that size. By paying a one time cost to time a few formulas of one particular size to train the performance model, our method is able to generate fast formulas for many different transform sizes, including larger sizes.

1.7 Document Outline

Following this introduction, Chapter 2 details some of the necessary background for understanding the remainder of this document. In particular, this chapter provides background on signal processing and describes the SPIRAL system.

The next two chapters describe our work in optimizing the performance of signal processing algorithms. Chapter 3 describes our development and implementation of different search algorithms in the SPIRAL system. Further, it compares their relative performance. Chapter 4 then provides more details about STEER, the evolutionary stochastic search algorithm that we developed for this domain.

The next two chapters describe our work in modeling the performance of signal processing algorithms. Chapter 5 describes techniques that attempt to directly model the performance of entire formulas. Chapter 6 then describes techniques that model the performance of individual nodes in split trees and how this can be used to predict performance for entire formulas.

Chapter 7 details our method for generating fast implementations of both the WHT and the DFT. This method uses the learned performance models of Chapter 6 to guide its construction of fast formulas.

Finally, Chapter 8 discusses related work, and Chapter 9 presents conclusions and future work.

Chapter 2

Background

This chapter presents some necessary background for understanding the remainder of the thesis. Section 2.1 gives a brief overview of signal transforms. Section 2.2 introduces the Walsh-Hadamard Transform (WHT), giving concrete examples of the concepts first presented in Section 2.1. Section 2.2 also describes break down rules, split trees, stride, and the search space for the WHT. Then, Section 2.3 describes the WHT timing package that has been used in this thesis. Next, Section 2.4 highlights some of the significant differences and similarities between the WHT and some other transforms that have been investigated. Finally, Section 2.5 describes the SPIRAL system.

2.1 Signal Transforms

A linear signal transform takes as an input a signal X as a vector and produces the transformation $Y = AX$. The matrix A is called the transformation matrix, and it defines the transform. A transform of size n operates on an input vector of length n and is represented by a transformation matrix of size $n \times n$. The output vector Y then also has length n. (Tolimieri et al., 1997; Nussbaumer, 1982; Rao and Yip, 1990)

Throughout this thesis, we have tested our methods with real or complex valued input vectors and transformation matrices. It is possible to perform integer transforms or even transforms on arbitrary groups (Maslen and Rockmore, 1995). The work presented here is not restricted to real or complex transforms, but no other types of transforms have been tested as the infrastructure used does not currently allow for other types of transforms.

A naïve implementation of the matrix-vector multiplication $Y = AX$ would require $O(n^2)$ operations. Fortunately, the transformation matrices in signal processing are usually highly structured, allowing them to be factored into a product of structured matrices. These factorizations usually lead to algorithms with $O(n \log n)$ operations, resulting in a substantial reduction in execution time. Further, there are often many possible ways to factor a transform, each which can be represented by a different but equivalent mathematical formula (Auslander et al., 1996).

While most of these different formulas lead to $O(n \log n)$ algorithms with the exact same arithmetic operation counts, they perform the operations in different orders and access data differently. This causes different formulas to have widely different runtimes on the same machine and the same formula to have widely different runtimes across different machines. Different operation orderings can greatly impact on cache performance as well as register and execution unit usage, causing the variance in runtimes between formulas. Since computers have differing cache architectures and different numbers of registers and execution units, the same formula can have widely varying runtimes between computers.

Figure 2.1 shows a histogram of the runtimes of 13,175 different formulas for the Walsh-Hadamard Transform of size 2^{16}. All of these formulas were run on the same Pentium III 450 MHz running Linux. The histogram shows a significant spread of runtimes, almost a factor of 6 from fastest to slowest. Further, it shows that there are relatively few formulas that are amongst the fastest.

This thesis is interested in finding the formula that implements the signal transform with the fastest runtime on a given machine. That is, we wish to find the best factorization of the transformation matrix for a given architecture.

It is possible that further optimizations could be performed if it was known that the input vector had particular structure. However, for this thesis we will not consider this case. We will assume that we wish to find the optimal implementation of a given transform for any arbitrary input vector. That is, our resulting implementations will correctly perform the specified transformation on any arbitrary input vector of the given length. However, many of the techniques and methods presented here should be easily extended to also consider structured input vectors.

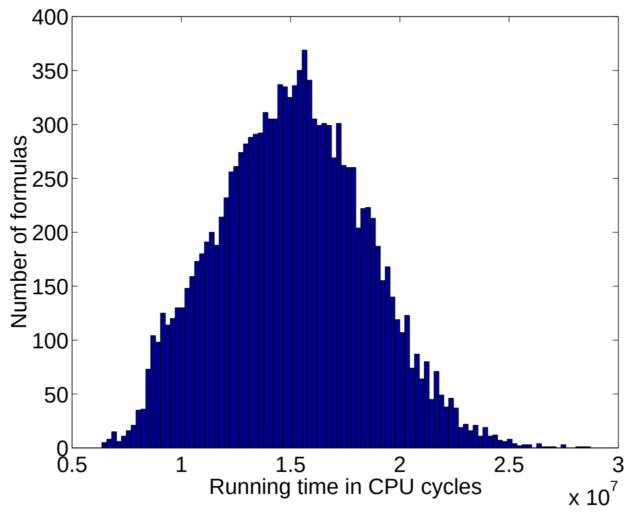


Figure 2.1: Histogram of runtimes some $WHT(2^{16})$ formulas run on the same machine.

2.2 Walsh-Hadamard Transform

Now we consider the Walsh-Hadamard Transform (WHT) as an example of a specific transform. The concepts introduced have close analogs for other transforms. Our research began with the WHT because it is similar to the widely used Fast Fourier Transform (FFT), but is a simpler transform.

2.2.1 WHT Definition

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 (Beauchamp, 1984). 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,

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

2.2.2 Break Down Rules

Clearly the matrix in the example above is highly structured. By calculating and combining smaller WHTs appropriately, this structure can be leveraged to produce 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 the product

$$\begin{gathered} [WHT(2^{n_1}) \otimes I_{2^{n_2+\cdots+n_t}}] \cdot \\ \left[\prod_{i=2}^{t-1} (I_{2^{n_1+\cdots+n_{i-1}}} \otimes WHT(2^{n_i}) \otimes I_{2^{n_{i+1}+\cdots+n_t}})\right] \cdot \\ [I_{2^{n_1+\cdots+n_{t-1}}} \otimes WHT(2^{n_t})], \end{gathered}$$

where I_k is the $k \times k$ identity matrix. We call the above formula a "break down rule" as it specifies how to factor a transform of one size into other (in this case smaller) transforms. This 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.

2.2.3 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 2.2(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. The split tree in Figure 2.2(b) corresponds to the formula:

$$WHT(2^5) = [WHT(2^2) \otimes I_{2^3}][I_{2^2} \otimes WHT(2^1) \otimes I_{2^2}][I_{2^3} \otimes WHT(2^2)]$$

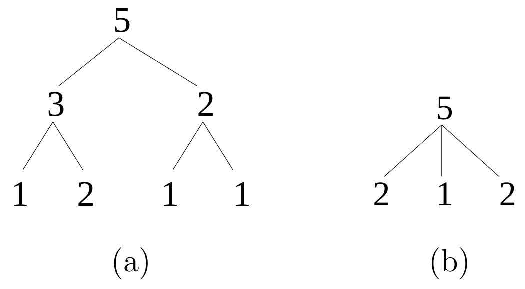


Figure 2.2: Two different split trees for $WHT(2^5)$.

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. In the above example, the representation was simplified since it only used one break down rule which only involved WHTs.

More formally, the root node of a split tree specifies the transform being factored including the transform's size. The break down rule used to factor this transform is also specified at the root. The children of the root node then specify the transforms resulting from the application of this break down rule. In general, a node in a split tree specifies the transform being factored and the break down rule being used. The children of a node represent the resulting transforms from applying the node's break down rule to the node's transform.

Split trees capture the order and sizes of the computations that are performed in a computer. Any change to the split tree means a change in the order and/or size of the computations performed. Unlike simple scalar multiplication where $a * b$ runs just as fast as $b * a$, the different orderings and different sized subcomponents of these matrix operations can often produce very different runtimes on a computer.

2.2.4 Stride

Implicit in a WHT 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. A stride of 1 means that a node accesses data from the input and output vectors sequentially. Such a node would access data items x_0, x_1, x_2, ..., up to the size of the transform being computed at that node, assuming no initial offset. A stride of s means that a node accesses every s-th data item from the input and output vectors. So, such a node would access data items x_0, x_s, x_{2s}, ..., with no initial offset, and data items x_i, x_{s+i}, x_{2s+i}, ..., with an initial offset of i.

Stride arises from the tensor products of the smaller WHTs with identity matrices in the mathematical formulas for factored WHTs. For example, $(WHT(2^2) \otimes I_{2^3})x$ can be calculated by performing 2^3 $WHT(2^2)$'s all at stride 2^3 but changing their initial offset into the input vector x. That is, WHTs of size 2^2 are performed on the four data items x_i, x_{8+i}, x_{16+i}, and x_{24+i}, with i taking on a different value for each WHT performed $(0 \leq i < 8)$.

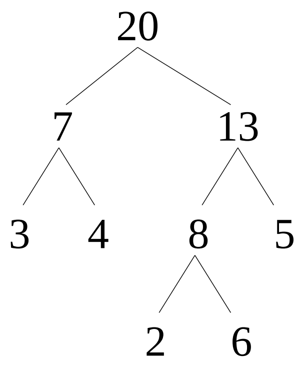
Thus, the stride of a WHT node is determined by its location in the split tree. The root node of a split tree has a stride of 1. If a node has a stride of s, then its right child has a stride also of s. Further, if the right child is of size r, then the next child to the left has stride sr. An arbitrary child of this node has a stride of s times the sizes of all siblings to the right of the given child.

Figure 2.3 gives an example WHT split tree along with the stride of each of nodes. Note that in the figure all values are given as base two logarithms of their size or stride.

The stride of a node can greatly impact its cache performance. Two nodes of the same size but that have different strides can have very different cache performance.

2.2.5 Search Space

There is a 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 $O((4+\sqrt{8})^n/n^{3/2})$ different possible split trees (Johnson and Püschel, 2000). The column marked "All" in Table 2.1 shows the total number of possible WHT formulas for sizes 2^1 to 2^{20}. For example, $WHT(2^{10})$ has 551,613 different split trees. Since it requires more than a second to obtain an accurate runtime for a single formula, it



$\log_2$ Size	$\log_2$ Stride
20	0
13	0
5	0
8	5
6	5
2	11
7	13
4	13
3	17

Figure 2.3: Example of stride for WHT nodes.

would require more than 6 days to time all possible $WHT(2^{10})$ formulas and over a million years to time all possible $WHT(2^{20})$ formulas.

To slightly reduce the search space, it is possible to only consider binary WHT split trees. However, there are still $O(5^n/n^{3/2})$ possible binary $WHT(2^n)$ split trees (Johnson and Püschel, 2000). The column marked "Binary" in Table 2.1 gives the actual number of binary WHT split trees. Even with this reduction, it would require over a week and a half to time all formulas of size 2^{12}.

By conducting a number of different searches, we have observed that the fastest formulas never have leaves of size 2^1. This confirms that it is more effective to use unrolled code of sizes larger than 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, there are 51,819 binary trees of size 2^{10} but only 101 binary trees of size 2^{10} with no leaves of size 2^1, as shown in Table 2.1. This restriction allows us to exhaustively search this limited space for formulas of sizes greater than the level 1 data cache.

Unfortunately, it still requires at least several hours to exhaustively search this limited space for transforms much larger than 2^{16}. Again through conducting a number of different searches in this space, we have observed that the best formulas are always rightmost split trees (trees where every left child is a leaf). Exhaustively searching this restricted space is then possible for even larger sizes than 2^{16}.

Table 2.1: Number of WHT formulas for different subsets of formulas.

Size	All	Binary	Binary No 2^1 leaves	Binary No 2^1 leaves Rightmost
2^1	1	1	0	0
2^2	2	2	1	1
2^3	6	5	1	1
2^4	24	15	2	2
2^5	112	51	3	3
2^6	568	188	6	5
2^7	3,032	731	11	8
2^8	16,768	2,950	23	13
2^9	95,199	12,234	46	20
2^{10}	551,613	51,819	101	33
2^{11}	3.2×10^6	223,180	218	52
2^{12}	1.9×10^7	974,382	488	84
2^{13}	1.2×10^8	4.3×10^6	1,092	134
2^{14}	7.1×10^8	1.9×10^7	2,489	215
2^{15}	4.4×10^9	8.6×10^7	5,696	344
2^{16}	2.7×10^{10}	3.9×10^8	13,175	551
2^{17}	1.7×10^{11}	1.8×10^9	30,620	882
2^{18}	1.1×10^{12}	8.1×10^9	71,664	1,413
2^{19}	6.6×10^{12}	3.7×10^{10}	168,468	2,262
2^{20}	4.2×10^{13}	1.7×10^{11}	398,041	3,623

2.3 WHT Timing Package

For most of the results with the WHT presented in this document we used a WHT package, (Johnson and Püschel, 2000), that can implement and run WHT formulas passed to it. The package can also time the execution, giving a runtime for the specified formula. For some computer architectures, the package can use performance counters to count the number of cycles needed to execute the given WHT formula.

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. Unrolled or straight-line code is a sequence of instructions with no loops, jumps, or function calls so that each instruction is executed sequentially. The code for these leaves efficiently implements a WHT of the given size, accessing its input and output at any specified stride and initial offset. Internal nodes of the split tree are implemented as recursive calls to their children. For example, suppose an internal node of size n and stride s has two children, a right child of size r and a left child of size l (where $n = rl$). This node is then computing:

$$[\ (WHT(l) \otimes I_r)\ (I_l \otimes WHT(r))\ ] \otimes I_s.$$

When this node is called, it recursively calls the right child l times in a row. Every call specifies a stride of s but each call has a different initial offset so that the right child accesses n different data items all together. When called, this right child will recursively call its children if it is also an internal node. Then, the original node calls its left child r times, all at stride sr, but again with different initial offsets so that it accesses n different data items.

The straight-line code used in the leaves has the advantage that it does not have loop or recursion overhead but the disadvantage that large code blocks will overfill the instruction cache and can be difficult for compilers to optimize. Thus, it is often better to *not* continue splitting down every node possible until every leaf is of size 2^1. Having leaves of size larger than 2^1 has the advantage of avoiding some recursive calls and loop overhead. However, it is also usually better to continue splitting nodes of size 2^8 as its straight-line code is often too large.

Besides its usual timing method, the WHT package also uses the Performance Counter Library (PCL) (Berrendorf and Mohr, 2000) to use hardware performance counters to obtain different performance measures for formulas or even subportions of formulas. PCL can be used on a variety of popular platforms, and the WHT package can use PCL anywhere that PCL can be properly installed. This allows the

WHT package to potentially collect a number of different performance measures for formulas, such as cache misses incurred (for different caches), TLB misses, CPU stalls, number of instructions executed from different instruction classes, and so on.

Further, since PCL uses hardware performance counters, it is possible to get reasonably accurate timings for subportions of WHT formulas. The WHT package can return timings for each subtree in an entire split tree, indicating the total amount of time (or other performance measure) spent in computing each portion of the formula. Adding calls to obtain performance measurements during the computation of a formula can reduce the accuracy of the measurements; so, this is only used when timings for subportions of formulas are needed, and not when wanting to obtain a runtime for the entire formula.

2.4 Other Transforms

We have investigated 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 2.2 gives a few example break down rules for some of these transforms. Some break down rules show how to compute a transform of one size from several smaller transforms of the same type, while other break down rules show how to compute a transform from several smaller transforms of different types. Further, some break down rules show how to translate a transform of one type into a transform of a different type but of the same size.

The discrete trigonometric transforms (DTTs), DCT and DST, (Rao and Yip, 1990; Chan and Ho, 1990; Wang, 1984; Strang, 1999) are considerably different from the WHT. Specifically, the following differences are of importance:

- While we have used just one basic break down rule for the WHT, there are several very different break down rules for most of the different types of DCTs and DSTs. Currently we have as many as ten different break down rules for some of the DTTs while others have two or three.

- While the break down rule for the WHT allowed for many possible sets of children, most of the break down rules for the DTTs specify exactly one set of children.

Table 2.2: A few example break down rules for some specific transforms. These equations are not meant to be exact but simply to give the flavor of the 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.

$$\begin{aligned}
WHT(2^n) &= (WHT(2^{n_1}) \otimes I_{2^{n_2+\cdots+n_t}}) \\
&\quad \left[\prod_{i=2}^{t-1} (I_{2^{n_1+\cdots+n_{i-1}}} \otimes WHT(2^{n_i}) \otimes I_{2^{n_{i+1}+\cdots+n_t}})\right] \\
&\quad (I_{2^{n_1+\cdots+n_{t-1}}} \otimes WHT(2^{n_t})) \\
DCT\_IV(n) &= M_n * DCT\_II(n) * D_n \\
DCT\_IV(n) &= M_n * P_n * (DCT\_II(n/2) \oplus DST\_II(n/2)^{P_{n/2}}) * R_n \\
DST\_I(n) &= P_n * (DST\_III(n/2) \oplus DST\_I(n/2)^{P_{n/2}}) * M_n \\
DFT(rs) &= (DFT(r) \otimes I_s)\, T_s^{rs}\, (I_r \otimes DFT(s))\, L_r^{rs} \\
DFT(n) &= CosDFT(n) + i * SinDFT(n) \\
CosDFT(n) &= M_n * (CosDFT(n/2) \oplus DCT\_II(n/4)) * M_n * P_n
\end{aligned}$$

- 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. Most of the DTTs are interrelated, with one DTT being a child of another DTT in a factorization of the second DTT. Thus, when specifying a split tree for a DTT, it is necessary to not only label the nodes with the size of the transform, but also with which transform is present at each node and which break down is applied at each node.

- Most of the rules for the DTTs either translate a DTT of one type to another type without a change in size or produce two children each of half the size of the parent. Thus, the sum of the sizes of the children add up to the size of the parent for the DTTs. Whereas, with WHT, the product of the sizes of the children is the size of the parent (sum of the base two logarithm). Therefore, 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 the current set of break down rules.

The DFT also has many different break down rules. Note that a fast implementation of the DFT is called a Fast Fourier Transform (FFT). The standard Cooley-Tukey decomposition (Cooley and Tukey, 1965) for the FFT is:

$$DFT(rs) = (DFT(r) \otimes I_s)\, T_s^{rs}\, (I_r \otimes DFT(s))\, L_r^{rs}$$

The structure of this break down rule is similar to the one for the WHT in that it factors a DFT of one size into two DFTs where the product of the sizes of the resulting DFTs is equal to the size of the original DFT. Another break down rule for the DFT splits it into real and imaginary parts which each can be decomposed using the DTTs (Vetterli and Nussbaumer, 1984).

Table 2.3 lists the number of different split trees for different transforms using all of the default break down rules available in the SPIRAL system as of version 3.1. The table shows that the number of formulas grows even more quickly for the DFT and the DCTs than for the WHT. Further, the number of formulas for the DCTs grow even more quickly than for the DFT. Clearly, the space of all possible formulas for these transforms can not be exhaustively searched at sizes above 2^5.

Table 2.3: Number of possible split trees for different transforms.

Size	DFT	DCT1	DCT2	DCT3	DCT4
2^1	1	2	1	1	1
2^2	6	4	1	1	10
2^3	40	8	10	10	126
2^4	360	160	1,260	1,260	31,242
2^5	258,400	403,200	3.9×10^7	3.9×10^7	1.9×10^9
2^6	1.8×10^{13}	3.2×10^{13}	7.6×10^{16}	7.6×10^{16}	7.3×10^{18}
2^7	7.2×10^{13}	4.8×10^{30}	5.6×10^{35}	5.6×10^{35}	1.1×10^{38}
2^8	7.2×10^{14}	5.4×10^{66}	6.0×10^{73}	6.0×10^{73}	2.3×10^{76}
2^9	1.5×10^{16}	6.4×10^{140}	1.4×10^{150}	1.4×10^{150}	1.1×10^{153}
2^{10}	2.3×10^{17}	1.8×10^{291}	1.5×10^{303}	1.5×10^{303}	2.2×10^{306}

2.5 SPIRAL

This section briefly describes the SPIRAL system (Püschel et al., 2001b; Moura et al., 1998) that has been developed by the SPIRAL research group. The development of the SPIRAL system has been a major collaborative effort spanning several universities and many researchers. This section, however, does not provide much detail about the search engine, as that is the major focus of Chapter 3. The search engine in the SPIRAL system is one of major contributions of this thesis. To understand this contribution, it is necessary to understand the larger SPIRAL system framework in which the search engine works.

The SPIRAL system automatically produces code for performing a given transform of a given size. Different factorizations can be specified to the system as break down rules, allowing the system to produce different implementations of the same transform. The system is general in that new transforms and new break down rules can be specified to the system and it can immediately produce code for these new transforms using the new break down rules. Further, the system can automatically produce platform adapted, highly optimized code by performing a search over the different degrees of freedom available (e.g., applying different break down rules). The SPIRAL system can currently produce C or Fortran code for performing the specified transform, allowing the code to be easily integrated into applications.

2.5.1 System Design

Figure 2.4 provides a basic overview of the major components of the SPIRAL system. 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. A large number of transforms, including the DFT, WHT, and four types of discrete cosine and sine transforms (DCT/DST), and a large number of break down rules are already specified to the system by default.

2. **Formula Generation.** The second step is to apply these break down rules repeatedly to produce a complete factorization of a given transform. This factorization is represented by a mathematical formula. The formula generator uses the transform and break down rule specifications to be able to produce

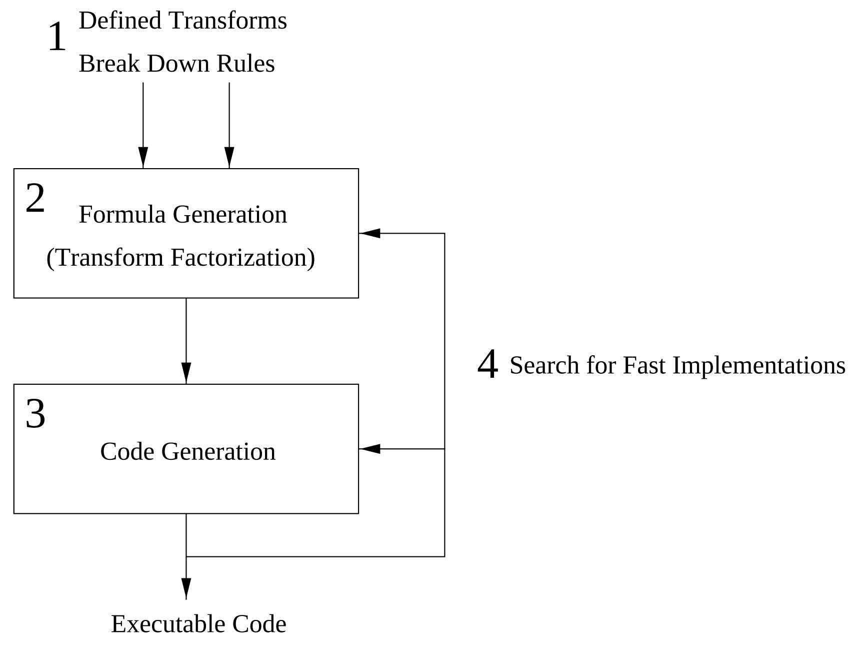


Figure 2.4: The SPIRAL system.

a formula that implements the given transform efficiently. In general, there are many different ways to apply the break down rules leading to a very large number of different formulas that can be obtained. (Püschel et al., 2001a)

3. **Code Generation.** Given a formula, the third step is to implement it in executable code. To perform this step, a compiler has been developed to translate formulas into a low-level language such as Fortran or C. This resulting Fortran or C program can then be compiled by a standard compiler for the given language. 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. (Xiong, 2001; Xiong et al., 2001; Johnson et al., 2000)

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. Further, this executable code can be run to obtain a runtime for that implementation. The final but key step in the process, then, is to use this machinery to search for

a fast implementation for a given transform. Research and implementation of this final step is one of the major contributions of this thesis and is discussed in Chapters 3 and 4.

The SPIRAL system has an additional module that can be used to test and verify the code that it generates. Given that a user may specify a new break down rule for a transform, it is beneficial to be able to verify that the factorizations produced using this new break down rule actually calculate the specified transform. Further, given that there is several steps from transform specification to executable code, having a method to verify the correctness of the generated code is desirable. The SPIRAL verifier can compare a given factorization against the definition of a transform (implemented as a matrix multiplication), or it can compare two different factorizations. These comparisons can be made on a set of basis input vectors or on a set of random input vectors.

2.5.2 Representation

The SPIRAL system uses several different representations for transforms and their factorizations.

A transform can be specified to the system by providing:

1. A name for the transform as a string,
2. A function for checking its parameters, and
3. A function for producing its defining transformation matrix.

A transform specification usually defines the transform across all possible sizes, and thus most transforms take a size as a parameter. Some of the discrete trigonometric transforms also have scaled or unscaled versions which can be specified as a parameter. The system is general and allows new transforms to take arbitrary parameters.

A break down rule can be specified to the system by providing:

1. The transform it applies to,
2. A function for checking whether it is applicable for a given transform and its parameters,

3. A function for returning the resulting (often smaller) transforms from applying the break down rule once, and
4. A function for returning the entire mathematical formula from applying the break down rule once.

The function in item 3 returns the transforms that would appear in the resulting mathematical formula from applying the break down rule and that could be further factored. Since many break down rules specify several different factorizations, this function actually returns all possible sets of resulting transforms.

The formula generator uses a representation which we have called a "ruletree" extensively as an intermediate representation. The search engine also heavily uses this representation and it is a common interface between the two modules. A ruletree is very similar to a split tree which was introduced in Section 2.2.3, and most of the remaining document will use the two terms interchangeably.

A ruletree is an efficient representation of a factorization. A ruletree consists of nodes that specify the following:

1. The transform and its parameters represented by the node,
2. The break down rule that has been applied to this transform,
3. The children of the node, if any, and
4. Any formula compiler options specific to this node and its subtree.

Figure 2.5 shows an example ruletree.

There is a one to one correspondence between ruletrees and mathematical formulas representing the transform factorizations. The formula generator can take a fully specified ruletree and transform it into a mathematical formula for exporting to the formula generator. The formulas are written in a specially designed language called SPL (Signal Processing Language) (Xiong, 2001; Xiong et al., 2001; Auslander et al., 1996). Hence the formula compiler is also called the SPL compiler. The SPL language also supports certain directives to the SPL compiler, such as indicating that a particular portion of the formula should be implemented as unrolled code.

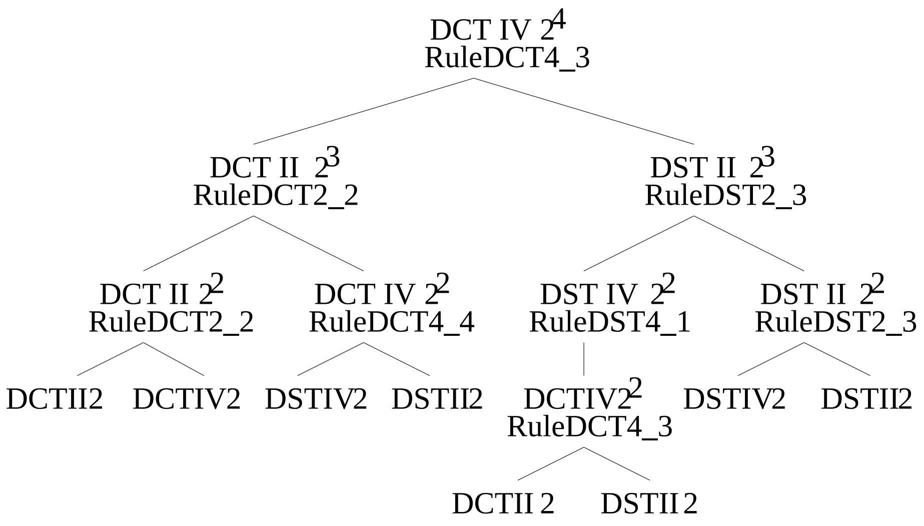


Figure 2.5: Example ruletree.

2.5.3 Implementation

The formula generator and the search engine have been implemented in the computer algebra system GAP (Groups, Algorithms, and Programming) (Schönert et al., 1995; GAP) with the share package AREP (Egner and Püschel, 1998). Some useful code for manipulating structured matrices was already available in AREP, and GAP also provides a powerful, high-level programming language. For both of these reasons, the formula generator was implemented in GAP. Since the search engine must heavily interact with the formula generator, it was also written in GAP.

Specifications for transforms and break down rules are currently given as records in GAP. However, even more user-friendly methods for specifying new transforms or break down rules may be developed in the future.

The formula compiler is written in C and can be used stand alone if desired. Code has been written in GAP to write out a formula to a file and to call the formula compiler externally. Further, timing information can be returned to the search engine through some additional interface code in GAP.

The verifier has been written as a number of shell scripts. These scripts call the formula compiler and run the resulting code. Further, there has been code developed in GAP to interface with the verifier.

2.5.4 User Interface

The SPIRAL system is available for download at `http://www.ece.cmu.edu/~spiral`. Installation is a simple, few step process. Upon starting up SPIRAL, a command line prompt is presented. GAP provides a powerful command line interface, including history and completions. SPIRAL simply uses the standard GAP command line interface.

A user can then simply specify a transform with a simple command such as:

```
spiral> t := Transform("DFT",2^20);
```

which specifies `t` to be a DFT of size 2^{20}. Then by running:

```
spiral> Implement(t);
```

the user can obtain a fast implementation for this DFT. In particular, the system performs a number of searches trying to find the fastest implementation possible in 30 minutes. This generates a C or Fortran (depending on the system default) file in the user's current directory containing the fast implementation of this DFT. Of course, many other options are available.

The SPIRAL system comes with a considerable amount of documentation. This documentation is provided in a number of forms including text, postscript, and at the SPIRAL prompt.

Further, the command line interface provides researchers with an easy interface for experimentation. For example, transforms can be specified, particular split trees grown, runtimes obtained, and search algorithms tested, all with rather simple function calls.

Several functions are also provided for testing the SPIRAL system. These include verification of the specified break down rules and of the generated code. They also include basic tests that the installation was performed correctly and that the native C or Fortran compiler works correctly. Many of the major components of the SPIRAL system have their own test and verification routines.

2.6 Summary

This chapter described the Walsh-Hadamard Transform in detail, explaining how it can be factored. It compared the WHT with several other transforms that we have investigated. We presented split trees as a simple but important representation for the factorization of signal transforms. We discussed the search spaces for signal transforms and introduced specific subspaces for the WHT. The chapter introduced the concept of stride for nodes in WHT split trees. Also, we described a package for timing WHT formulas that is used in this thesis. Finally, we presented the SPIRAL system, providing the necessary background for understanding how the search engine fits within the overall system.

Chapter 3

Optimizing Performance with the Search Engine

The search engine is one of the major components of the SPIRAL system and a major contribution of this thesis. The other major components of the SPIRAL system generate and implement in code different factorizations of arbitrary, user-specified signal transforms. Many different factorizations, and thus possible implementations, can be generated for a given transform. By running these implementations, it is possible to collect runtimes for the different implementations on a given machine. The search engine then uses this machinery and the runtimes of different implementations to search for the fastest implementations possible for given transforms.

Search plays a critical role in the SPIRAL system for two main reasons:

1. There is a very large number of different implementations that the SPIRAL system can produce for any one transform.
2. These different implementations have a very wide variance in runtimes.

Thus, it is crucial to have an intelligent method to search through this space of implementations to find the fastest one possible.

We have implemented a number of different search methods in the SPIRAL system. All of these methods are capable of optimizing arbitrary, user-specified transforms. Not only do these methods search over different formulas (or equivalently, different factorizations) for a given transform, but most are also capable of searching over different formula compiler options that control how the formula is compiled into code.

We begin with a brief overview of the different search methods available in the SPIRAL system in Section 3.1. In Section 3.2, we describe the two different search spaces that are considered and how the search methods search over these spaces in general. Next, Section 3.3 describes the search engine's user interface and available options. Section 3.4 returns to the individual search methods and provides more details about the implementation of each method. Finally, Section 3.5 provides results from using the search engine.

The reader is encouraged to review Section 2.5 for additional details about the SPIRAL system. Details about the search method STEER have been omitted from this chapter since they are the topic of Chapter 4. Further, the user manual for the search engine appears in Appendix A.

3.1 Overview of the Search Methods

This section briefly discusses each of the search methods available in the SPIRAL system. For each of the search methods, this section presents the basic idea behind the method without going into details about how the method is implemented in the SPIRAL system. After further discussion, Section 3.4 then provides more details about each of the search methods and their implementation in the SPIRAL system.

Exhaustive Search

Exhaustive search is the most basic search method. It generates all possible formulas for a given transform and exhaustively times each one to determine the fastest one. This search method is only possible for very small transform sizes, as the number of possible formulas for most transforms grows very large as the transform size increases.

Dynamic Programming

Dynamic programming has been one of the most common approaches to search in this type of domain (Johnson and Burrus, 1983; Frigo and Johnson, 1998a; Haentjens, 2000; Sepiashvili, 2000). Dynamic programming builds up a table of the best formulas found for each transform and size. For a particular transform and its applicable rules, dynamic programming considers all possible sets of children. For each child, dynamic programming substitutes the best split tree found for that transform. Dynamic programming makes the assumption that the fastest split tree for a particular transform is also the best way to split a node of that transform in a larger tree. For many transforms, dynamic programming times very few formulas and still is able to find reasonably fast formulas.

Random Search

A very different approach is to generate a set of random formulas, time them and choose the fastest. This approach assumes that while there is a wide variance of runtimes between formulas, there are enough fast formulas that at least one can be found randomly. This approach has the advantage that it can time as few or as many formulas as the user desires, but the disadvantage of blindly generating random formulas.

Hill Climbing Search

As a refinement of random search, hill climbing search tries to successively modify an initially random implementation to find faster implementations. A number of random restarts are performed to generate a new random implementation occasionally.

STEER

As a refinement of random and hill climbing search, STEER (Split Tree Evolution for Efficient Runtimes) uses a stochastic, evolutionary search approach (Singer and Veloso, 2001b). STEER's approach is very similar to genetic algorithms (Goldberg, 1989) except that instead of using a bit vector representation, it uses split trees as its representation. STEER uses evolutionary operators to stochastically guide its search toward more promising portions of the space of formulas. STEER has the advantage of searching a larger portion of the space of formulas than dynamic programming while still remaining tractable unlike exhaustive search.

Timed Search

We have also developed a meta-search method that uses the above search methods to find the fastest implementation possible given a limited amount of time to search. This method takes advantage of the strengths of the different search methods to produce a fast implementation while also limiting the search to only take the specified length of time. The idea is for a user to be able to ask for the best DFT of size 2^{20} that the system can find in 30 minutes, for example.

3.2 Search Spaces

The search methods actually perform searches over two different spaces:

- The space of possible factorizations, and
- The space of possible options to the formula compiler.

These two different spaces are searched simultaneously as a transform's implementation is determined by both the chosen formula and the options passed to the formula compiler. However, we will now consider each search space separately.

3.2.1 Search Over Factorizations

Usually, the search methods begin by generating one or a few different factorizations of a given transform. These formulas are then implemented in code and timed. Based on this runtime information, the search method currently being used then generates a new formula or set of formulas to be timed. This process is then repeated until the search method reaches a stopping condition or until the space of formulas that the search method considers is exhausted.

To perform a search over possible factorizations, these search methods must control the formula generator and how it applies break down rules to produce formulas. Further, this must be done irrespective of the transform or available break down rules as new user-specified transforms and break down rules can be specified to the system.

First, a suitable representation for factorizations needed to be chosen to be used by both the formula generator and the search engine as a common interface. The formula generator takes a transform definition and a set of break down rules and produces a mathematical formula written in SPL (see Section 2.5.2) that represents a given factorization of the desired transform. However, these formulas were not an ideal representation for interfacing between the search engine and the formula generator for a couple reasons:

- The mathematical formula for a factorization obscures the important aspects of the factorization. Given just a formula, it is not easy to determine which break down rules have been applied or what intermediate transforms occurred during the recursive application of break down rules. Further, although the transforms at the leaves appear in the formula, they are intermixed with the necessary mathematical structures for combining these leaves appropriately.
- Formulas can require a large amount of memory storage. The mathematical formulas not only include the leaf transforms to be computed, but also all the necessary mathematics to recombine these leaves to compute the overall transform. These extra mathematical structures often require a large amount of memory to store.

Instead, we decided to use split trees (see Section 2.5.2) as a representation for interfacing between the search engine and the formula generator. This was chosen for several reasons:

- The particular break down rules used and the intermediate transforms were are all immediately accessible from the split tree. This allows the search methods to easily inspect and manipulate the split trees to generate new factorizations to be considered.
- A split tree can be easily converted into an SPL formula.
- A split tree usually requires significantly less memory storage in comparison to the corresponding SPL formula.

Second, the formula generator needed to provide methods for generating valid split trees or portions of valid split trees as the search engine required. Random search, hill climbing search, and STEER all require the ability to generate random split trees for a given transform. Exhaustive search requires the ability to generate all possible split trees. Dynamic programming requires the ability to generate all possible immediate children of a given node in a split tree. All three of these different split tree generation methods were implemented as functions in the formula generator and are used by the search engine.

3.2.2 Search Over Options to the Formula Compiler

Most of the search methods not only search over different factorizations, but also can search over options to the formula compiler. At the time of this writing, the only option to the formula compiler that significantly changes its generated code is the level of code unrolling. It is also possible to specify different options to the native C or Fortran compiler which could change the actual generated executable. The search engine allows for searches over different levels of code unrolling. Currently, the search engine does not allow for searches over different options to the native C or Fortran compiler; however, different options can be specified with each call to a search method.

All of the code that the formula compiler generates is either straight-line code (that is simply a linear sequence of statements) or such code contained in loops. Straight-line code is very efficient in that it specifies exactly what operations are to be performed with no overhead. However, it can be very difficult for C and Fortran compilers to optimize very long blocks of straight-line code. Further, straight-line code is particularly bad for instruction cache performance. Thus, it is often preferred to generate code with some loops. However, this has the disadvantage of requiring overhead for maintaining the loop variables and for calculating indices into the input and output vectors based on these loop variables.

Given an SPL formula, the formula compiler begins by generating code that contains many loops and only very short straight-line code sections. Then, based on the unrolling options specified, some of these loops are unrolled to generate longer sections of straight-line code (also called unrolled code). The formula compiler actually allows for code unrolling to be specified with two different methods:

1. A global option can be set, specifying a size. All loops whose data accesses are of the specified size or smaller are unrolled. This allows for very reasonable code to be generated by setting just one single option.
2. Local options can be set specifying particular portions of the SPL formula that are to be implemented as straight-line code. This allows a much greater degree of freedom than the global option in that one piece of code can be unrolled while another left with its loops even though they may access they same amount of data.

Thus, the search engine allows for two different types of searches over code unrolling based on these two methods for specifying the degree of unrolling to the formula compiler:

1. A search can be made over the size to be set for the global unrolling option. The user can specify a minimum and maximum size over which to search, limiting the search to reasonable portions of the search space.
2. A search can also be performed over placement of local unrolling specifications. Any node in a split tree can be marked for local unrolling, indicating that the entire subtree rooted at that node should be unrolled. Thus, if a node is marked for unrolling, then all nodes in its subtree are implicitly marked for unrolling. The user can specify a minimum and maximum size over which to search. Then, any node of the minimum size and smaller are always marked for unrolling and no node larger than the maximum size is ever marked for unrolling. Search is then performed over whether to mark for unrolling nodes of sizes between the minimum and maximum.

Exactly how these searches are implemented is dependent on the search method being used. It is not possible to simultaneously search over global and local unrolling options as this does not make sense. By default, most of the search methods do not perform either of these searches but simply use a default global unrolling size.

3.3 User Interface

Each of the search methods can be directly called. While many of the search methods have a large number of options available, reasonable defaults are provided for all options. So, for example, using STEER to search for a fast implementation of a DFT of size 2^{15} can be achieved by the simple command:

```
spiral> STEER( Transform("DFT", 2^15) );
```

Most of the search methods return the split tree corresponding to the fastest implementation that it found, as well as the formula compiler options used.

The user may specify two different sets of options, namely options for the search method and options that are directly passed to the formula compiler. The user only needs to specify the options for which a non-default value is desired. The available options for each of the search methods as well as for the formula compiler may be listed by calling the appropriate function. For example, the following call uses STEER to search for fast C implementations of the DFT of size 2^{15} while using a population size of 500 and setting the random number generator seed to 7:

```
spiral> STEER( Transform("DFT", 2^15),
>              rec(popSize:=500, seed:=7),
>              rec(language:="c") );
```

Note that the population size and random seed are options to the STEER search method which are grouped together in a GAP record, while the desired implementation language is an option to the formula compiler.

The search engine also provides a simple, high-level function for implementing a transform, called “Implement”. This function will automatically search for a fast implementation of the given transform and then produce a file containing the C or Fortran code for that fast implementation. Thus, by the simple call:

```
spiral> Implement( Transform("DFT", 2^15) );
```

a user can have a file created in the current directory that contains a fast implementation of a DFT of size 2^{15}. Implement can also take a number of options so that the user can specify the implementation language, the filename to be used, the search method to be used, or a number of other options. By default, Implement obtains a fast implementation by one of two methods:

1. If the search engine has already found a fast implementation of the desired transform, the corresponding C or Fortran code is generated.

2. Otherwise, the timed search method is used, allowing 30 minutes to search by default. The corresponding C or Fortran code for the fastest implementation that this search finds is then generated.

The search engine automatically keeps track of the best implementations that it has found so far for all of the transforms considered. These implementations are stored in a hash table as a split tree along with its corresponding formula compiler options. By using the formula generator and the formula compiler, Implement can easily take such a split tree and formula compiler options and create the corresponding

C or Fortran code. Further, these best found implementations may be saved to a file and then reloaded during a different session with the SPIRAL system.

Many of the major components of the SPIRAL system provide test and verification routines. These allow the user to test that the system has been installed correctly and that it is working properly. The search engine provides a test routine that tests each the different search methods. This is particularly useful in that for a search method to work correctly most of the other infrastructure in the SPIRAL system must also work correctly in order to produce runtimes, and thus testing the search engine also tests most of the SPIRAL system.

Further, the search engine also has a substantial user manual. This manual is accessible both at the SPIRAL prompt and in a text file. A user can type at the SPIRAL prompt:

```
spiral> ?Search
```

to begin reading the search engine user manual. Further, the user can request at the SPIRAL prompt information about any of the search methods or most of the other functions available in the search engine simply by typing a question mark followed by the function name. The user manual is duplicated in Appendix A.

3.4 Details about Search Methods

This section details the different search methods provided in the search engine.

3.4.1 Exhaustive Search

The exhaustive search method begins by requesting the formula generator to produce all possible split trees for the given transform. This is only feasible for very small sized transforms. Then exhaustive search times each of these split trees and returns the one that runs the fastest.

Currently, exhaustive search does not perform a search also over unrolling parameters, largely because this would greatly increase the search space of an already very large one. However, it is possible to manually conduct a search over global unrolling by calling exhaustive search multiple times, each time with a different global unrolling size set in the formula compiler options.

It is also possible to limit the time exhaustive search uses. If the time limit expires before exhaustive search has timed every formula, it simply returns the fastest split tree it found thus far.

3.4.2 Dynamic Programming

Dynamic programming builds a table of the best implementations that it has found for different transforms and transform sizes. For efficiency reasons, this table is implemented in the search engine as a hash table. A key to the hash table is a transform and its size, and the stored data is the fastest split tree and formula compiler options found for implementing that transform and size.

Given a transform and size, dynamic programming begins by checking whether it has a fast implementation already stored in its table. If it does, this implementation is returned. Otherwise, dynamic programming requests all possible sets of immediate children of the given transform from the formula generator. Each set of immediate children corresponds to a different split tree that dynamic programming will grow and time, before choosing the fastest.

For each possible immediate child, dynamic programming recursively calls itself to determine the fastest split tree that dynamic programming can find for that transform and size. Once this is determined, dynamic programming substitutes this fast split tree for the immediate child of the originally given transform. Thus, given a set of immediate children, dynamic programming recursively calls itself on each of these children to determine fast split trees to grow under these immediate children. The entire new split tree then is timed.

Dynamic programming then has timings for one split tree for each of the possible sets of immediate children of the given transform. The fastest one is chosen, stored in dynamic programming's table, and returned.

Thus, dynamic programming not only finds a fast implementation for the given transform and size but also for any of the possible descendents of that transform. However, dynamic programming makes a very strong 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.

That is, dynamic programming does not consider the context in which the transform is being computed — the same subtree is always grown for the same transform and size irrespective of its location in a larger split tree. One might expect this assumption not to hold for at least a couple reasons: (1) the state of the cache may be very different at the time a subtree is computed depending on its location within the larger split tree, and (2) the stride at which some subtrees are computed is dependent upon their location within the larger split tree.

For many transforms dynamic programming times relatively few formulas compared to other search methods. However, if the number of possible sets of immediate children grows linearly or faster with the size of the transform, dynamic programming becomes infeasible for larger transform sizes. This is true for the WHT if no restrictions are placed on the standard break down rule. Because the break down rule for $WHT(2^n)$ can create any set of children such that the product of the sizes of the children is 2^n, the number of possible sets of children is 2^{n-1}, the number of ordered partitions of n. Thus, dynamic programming would time 2^{n-1} split trees for $WHT(2^n)$, assuming it already determined the best split trees for all smaller sizes. However, by restricting the break down rule to produce just binary WHT split trees, dynamic programming becomes very efficient, timing at most $n-1$ formulas to determine the best formula of size 2^n given the best formulas for all previous sizes.

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 an exhaustive search. While exhaustive search can not be used at larger sizes, 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. Thus, exploring other search techniques besides just dynamic programming has the advantage of either producing better results or validating the use of dynamic programming as a fast method.

As a generalization of the dynamic programming approach, k-best dynamic programming not only keeps track of the best formula for each size but also the best k formulas (Haentjens, 2000; Sepiashvili, 2000). This softens the assumption that the best formula of a given size is the best way to split a node of that size in a larger split tree, allowing for the fact that a sub-optimal formula at a given size might be the optimal way to split a node of that size in a larger tree. Unfortunately, moving from standard 1-best to just 2-best dynamic programming more than doubles the number of formulas that must be timed.

It is also possible to specify a time limit for dynamic programming to search. However, it is possible that when the time limit expires that dynamic programming has not yet timed a single formula of the given transform (rather it has only been timing smaller transforms). In this case, no split tree can be returned. However, if at least one split tree for the given transform has been timed, then the fastest one found thus far is returned.

In the search engine, dynamic programming can search over global unrolling or over local unrolling parameters. To search over global unrolling settings, dynamic programming is essentially called multiple times, once for each possible global unrolling setting. Searching over local unrolling settings is slightly more complicated. The following cases occur:

- If dynamic programming is called with a node of size equal to or smaller than the minimum local unrolling size, then the node is set to be locally unrolled.
- If dynamic programming is called with a node of size larger than the maximum local unrolling size, then the node is *not* set for local unrolling.
- Otherwise, one of the following two cases applies:
  - If both children are marked for local unrolling, then the split tree is timed both with the node marked and with it not marked for local unrolling.
  - Otherwise, the node is *not* set for local unrolling. (This is required since marking this node for local unrolling would mean that all of its children would be unrolled by virtue of the parent being unrolled.)

3.4.3 STEER, Random Search, and Hill Climbing Search

STEER is discussed extensively in Chapter 4 and so a detailed discussion of STEER is omitted here. Random search and hill climbing search both use much of the same code as STEER.

Random search is simply implemented as a call to STEER with the number of generations set to one. That is, random search generates a number of random implementations which are timed. The fastest implementation is then returned. As random search is just a front end to STEER, all of the options that are available in STEER are also available to random search (as long as they do not pertain to the evolutionary operators). If the same implementation is randomly generated multiple times, the implementation is only timed once.

Random search has the advantage that the user can decide exactly how many formulas are to be timed by the search method before it returns the best implementation it found. Further, specifying a time limit is equally meaningful, allowing the user to specify how long random search takes before returning the fastest implementation it found. However, random search does not take advantage of any information provided by knowing the timings of different split trees.

Hill climbing search in the search engine is a refinement of random search, but not as sophisticated as STEER. Hill climbing search begins by generating a random implementation and timing it. Then it randomly applies a mutation to this implementation, using the mutations defined for STEER (see Section 4.3.3), and times this new implementation. If the new implementation is faster, hill climbing continues by applying a mutation to this new implementation and the process is repeated. Otherwise if the original implementation was faster, hill climbing applies another random mutation to the original implementation and the process is repeated. After a certain number of mutations, the process is restarted with a completely new random implementation. See Table 3.1 for pseudo-code of the hill climbing search method.

The user can specify how many mutations are to be performed before doing a restart. In particular, an absolute maximum number of mutations before restarting can be specified, as well as a maximum number of mutations without making an improvement in the runtime (that is, without accepting the new implementation) before restarting. Further, the user can also specify how many restarts are to be performed in a similar manner. An absolute maximum number of restarts can be specified, as well as a maximum number of restarts without making an improvement in runtime of the best implementation found thus far.

If the same implementation is derived again during the process, it does not need to be timed again since the hill climbing search method stores all timed implementations and their runtimes for quick access. If a time limit is specified, then the best implementation found thus far is returned when the time limit expires. Further, hill climbing search can search over global and local unrolling options since much of its code is based on parts of STEER's code.

Table 3.1: Pseudo-code for the hill climbing search method.

```
HillClimb( Transform )
    BestTime = $\infty$
    While ContinueRestarting do
        CurrentImp = RandomImplementation( Transform )
        CurrentTime = Time( CurrentImp )
        While ContinueMutating do
            NewImp = Mutate( CurrentImp )
            NewTime = Time( NewImp )
            If NewTime < CurrentTime then
                CurrentImp = NewImp
                CurrentTime = NewTime
        done
        if CurrentTime < BestTime then
            BestImp = CurrentImp
            BestTime = CurrentTime
    done
    return BestImp
```

3.4.4 Timed Search

The timed search method is really a meta-search algorithm in that it calls the other search methods to find a fast implementation of the given transform. The idea is to find the best possible implementation by using different search methods, constrained to searching for only a specified length of time. However, if the time limit is set unusually large, then timed search can be used simply as a meta-search algorithm for calling multiple other search algorithms to find a fast implementation.

All the other search methods allow for a time limit to be specified. Many of them can always return the best implementation found thus far when the time limit expires. However, some search methods like dynamic programming may not have timed a full split tree of the given transform by when the time limit expires, and thus can not return an implementation. It should be noted that it is not possible to set interrupts in GAP; so, most of these methods check the time at strategic places in their code. Thus, it is possible for the search methods to actually take longer than allowed. Unfortunately, this is not avoidable, but should not be a significant problem in most cases as checks are made fairly often.

The user can specify which search methods should be called by timed search including the order in which they are called. For each search method, the user can also specify what options should be passed to the search method and what the maximum time limit should be for the search method. Further, the user can specify an overall time limit which is not to be exceed. This overall time limit may cause some of the later search methods not to be called at all, and some of the search methods to be limited to less time than specified for those particular search methods.

By default, timed search will search for a fast implementation for 30 minutes. It begins by calling random search, limiting it to only generate 10 formulas. This default ensures that at least some implementation is quickly found. Then, dynamic programming and next STEER are called, both with the default options. Dynamic programming often can find a good implementation in relatively little time, while STEER can sometimes find faster implementations given more time. If time still remains then dynamic programming is called again, keeping the 4-best formulas this time and also doing a search over local unrolling options. Finally, STEER is again called with any remaining time, allowing it to search over local unrolling options, to use a larger population, and to perform mutations and cross-overs on more implementations each generation.

3.5 Results

In this section, we present some results from the search engine. Section 3.5.1 compares the different search methods across many different transforms on a Pentium. Section 3.5.2 shows results when the search methods are also allowed to search over local unrolling parameters. Finally, Section 3.5.3 shows results collected on a Sun.

3.5.1 Search Method Comparison

Figures 3.1 through 3.6 compare several of the different search methods at finding fast implementations of different transforms across many sizes. All of these runs were conducted on a Pentium III 900 MHz machine running Linux 2.2.17, using the 3.1

version of the SPIRAL system. For all of these runs, C code was generated by the SPL compiler. Except the timed search method, none of the other search methods searched over unrolling parameters but instead used a default global unrolling of all nodes of size 2^5 and smaller. There are two different types of plots shown in these figures:

- Plots showing the runtimes of the fastest formulas found. The runtime of the fastest formula found by the given search method for each size is divided by the runtime of the fastest formula found for that size by 1-best dynamic programming (that is, the default dynamic programming that only keeps the single best formula for each transform and size). This ratio of runtimes is then plotted, and thus lower points correspond to finding faster formulas.

- Plots showing the number of formulas timed by the different search methods. A search algorithm spends almost all of its time running different formulas to collect runtimes. Thus, the number of formulas timed is a good measure of the total time spent by the search algorithm. Note the logarithm scale along the y-axis. The random search method is not displayed in these plots as it always generated 100 random formulas (although it is possible, particularly at smaller sizes, that fewer were timed since duplicates are not timed repeatedly). An additional line is included for these plots representing the total number of possible formulas using the default rules in the SPIRAL system. Timed search may time the same formula multiple times through the various search methods it calls, and thus at very small sizes may time more formulas than shown for the "All Formulas" line. Further, the lines for dynamic programming include the number of formulas timed of smaller sizes as well as the current size (since these formulas must be timed), and thus at small sizes, dynamic programming may time more formulas than shown for the "All Formulas" line.

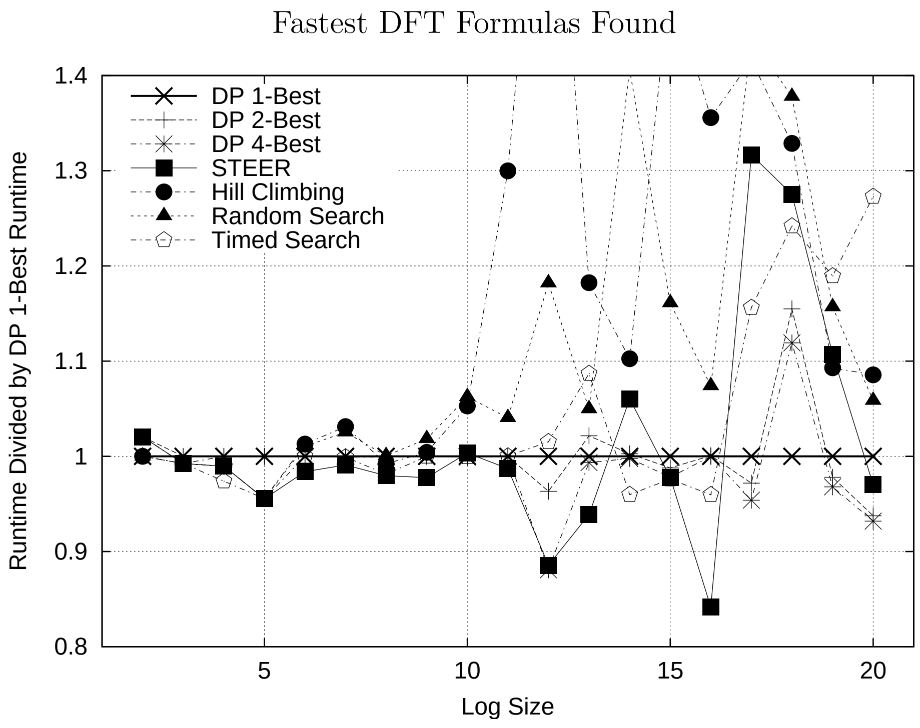
Particularly for the DFT and also for the WHT, both random search and hill climbing search find rather slow formulas in comparison to the other search methods at many of the larger sizes. While it is not surprising that random search would perform poorly, it is surprising that hill climbing search performs so poorly, sometimes even considerably worse than random search. However, hill climbing search does not generate as many completely random split trees as does random search. So, if the initial completely random split trees were all poor, it may have difficulty finding mutations that produce good split trees. Thus, STEER has an important advantage over hill climbing in that it generates and uses a large population, and also STEER has an advantage over random search in that it uses evolutionary operators to intelligently search for fast implementations.

For the WHT of size 2^6, timed search finds a formula with a significantly faster runtime than those found by the other tested search methods. This is because timed search was the only search method that was allowed to search over unrolling parameters. Timed search begins by calling random search, DP, and STEER without any search over unrolling parameters, and then if time remains it calls DP and STEER again allowing them to search over local unrolling parameters. By default all the other search methods unrolled all nodes of size 2^5 and smaller. The best formula that timed search found was completely unrolled.

For the DFT and WHT, there is no one search method that always finds faster formulas than all of the others. In fact, plain 1-best dynamic programming never performs extremely worse than the others, but is sometimes 10% slower than some of the other search methods. Generally, either STEER, timed search, or 4-best dynamic programming finds the fastest formula for a given transform and size. One advantage of the search engine is that so many different search methods are available.

However, for most of the Discrete Trigonometric Transforms (DTTs) and many of the sizes shown, STEER finds faster formulas than all of the other search methods. Further, STEER rarely finds a slower formula than the fastest found by all of the other search methods. Often for size 2^7, STEER is able to find a formula that is 5–20% faster than that found by 1-best dynamic programming.

These plots show that the number of possible formulas grows very quickly for very small sized transforms, forcing the search algorithms to only consider a very small portion of the space of formulas. For dynamic programming, clearly increasing the number of best formulas kept for each transform and size increases the number of formulas that must be timed. For small sizes, timed search usually times the most formulas as it calls several search algorithms. Since timed search is only allowed 30 minutes, it begins to time slightly fewer formulas at larger sizes at it requires more time to run larger sized formulas. For larger sizes of the DFT and WHT, 4-best dynamic programming times the most formulas of all the search algorithms. For sizes 2^5 to 2^7 for the DTTs, STEER times the most formulas.



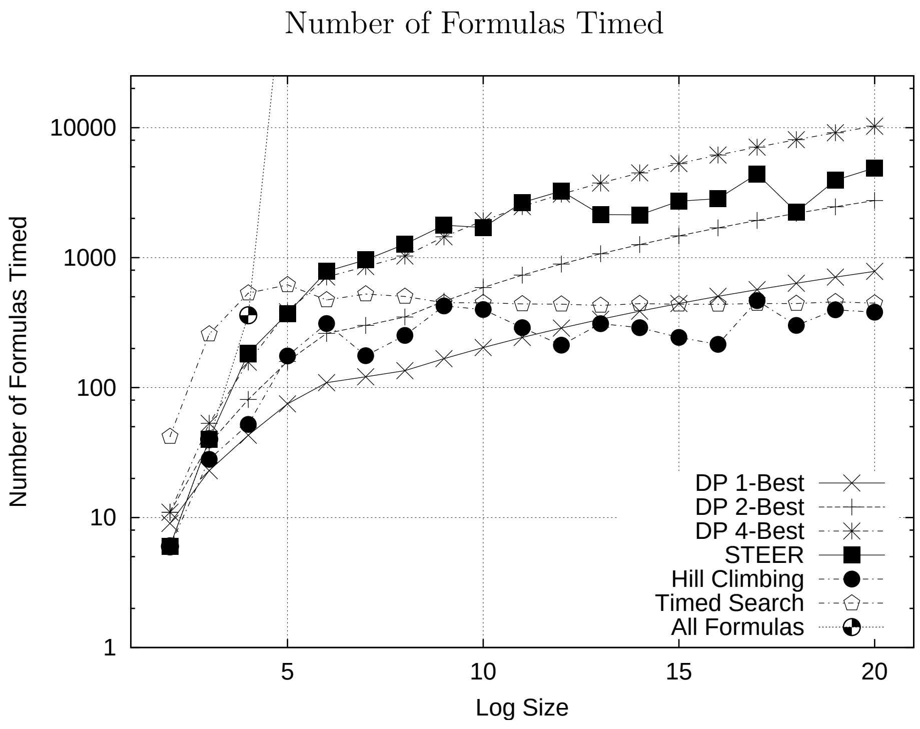
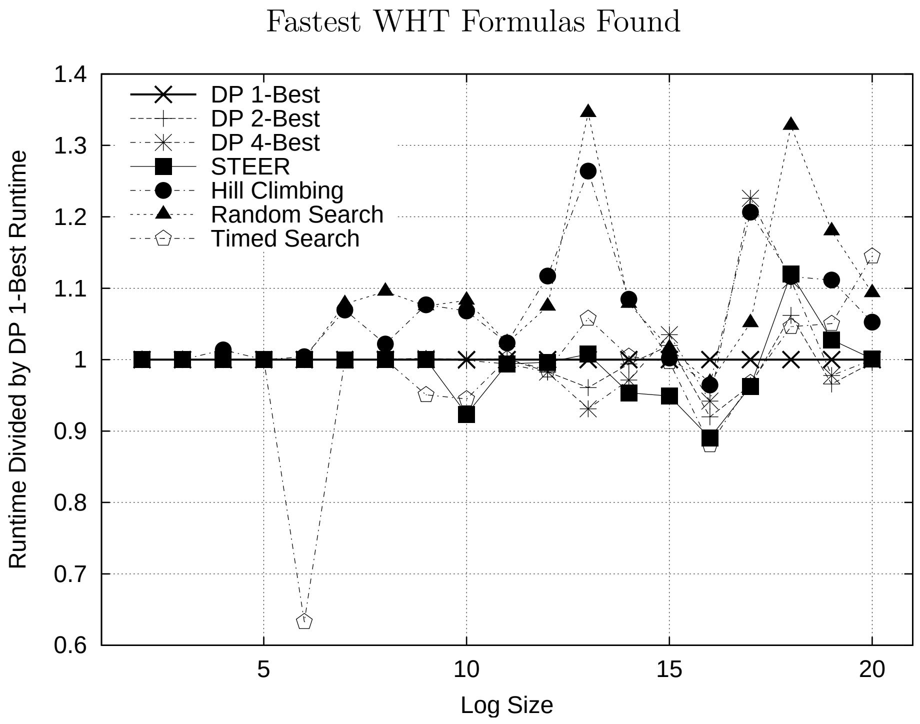


Figure 3.1: Comparison of search methods for the DFT on a Pentium.



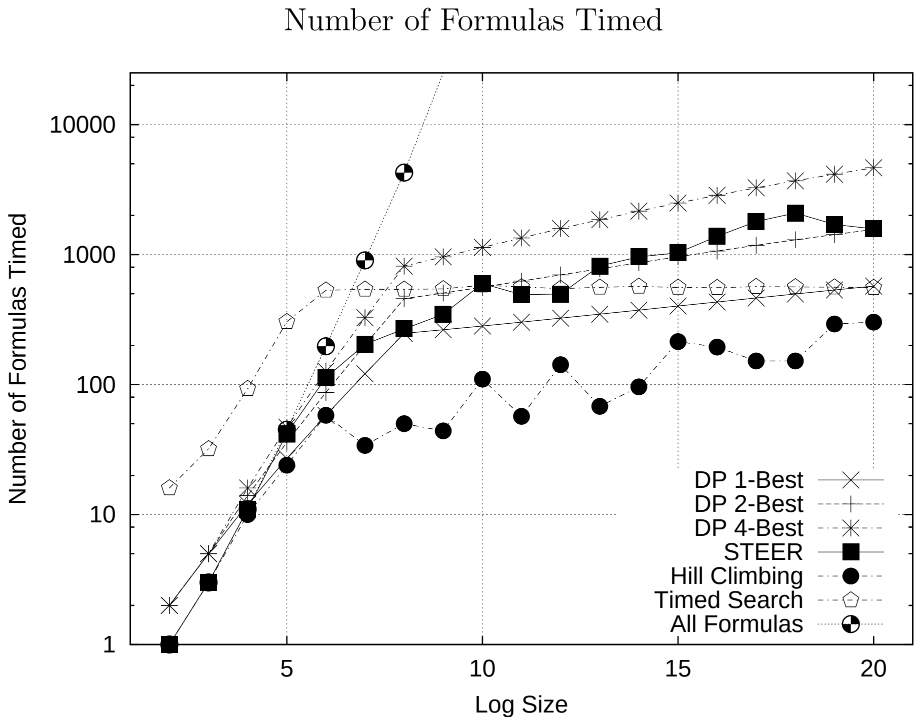


Figure 3.2: Comparison of search methods for the WHT on a Pentium.

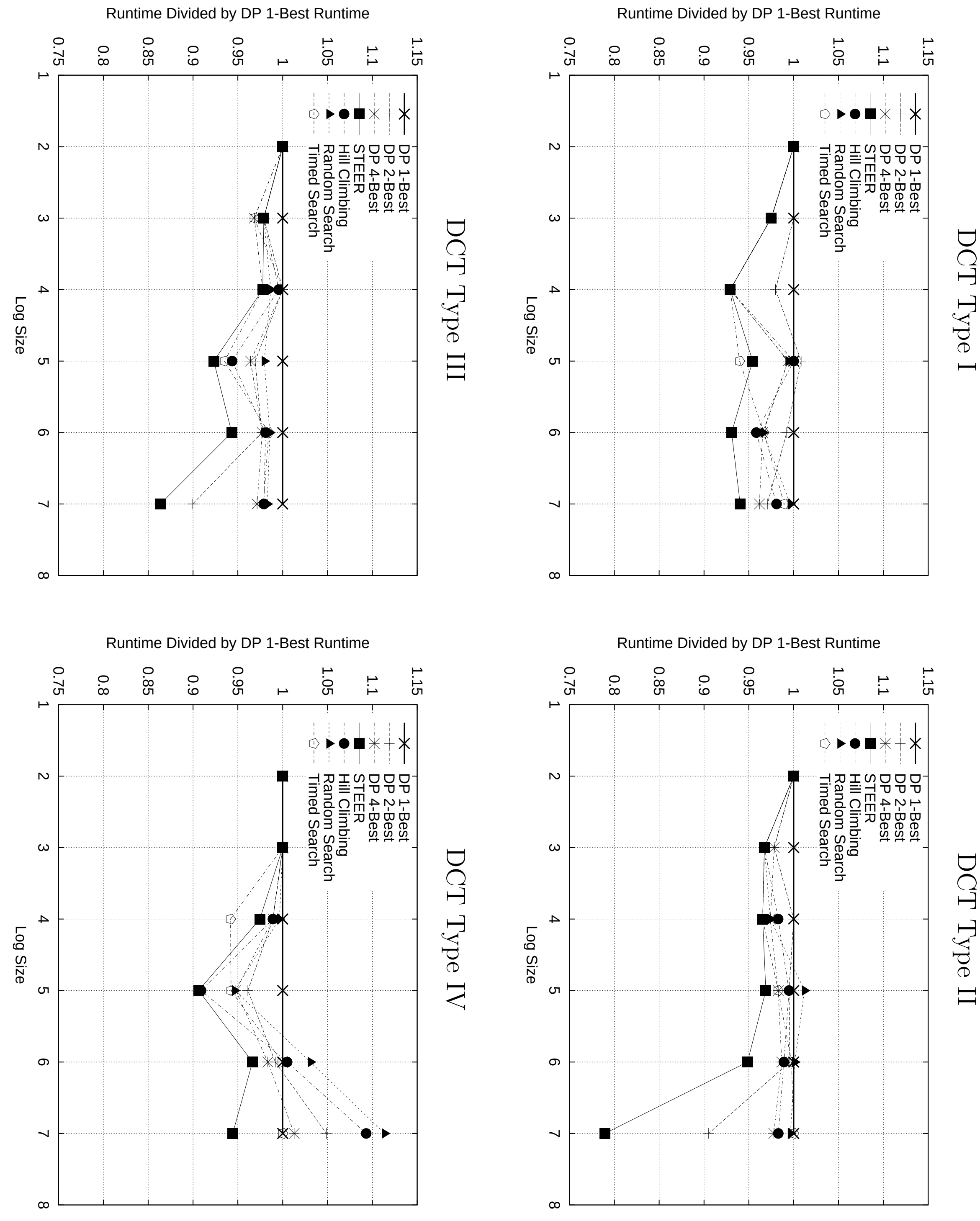


Figure 3.3: Comparison of the runtimes of the fastest formulas found by different search methods for small sized DCTs on a Pentium.

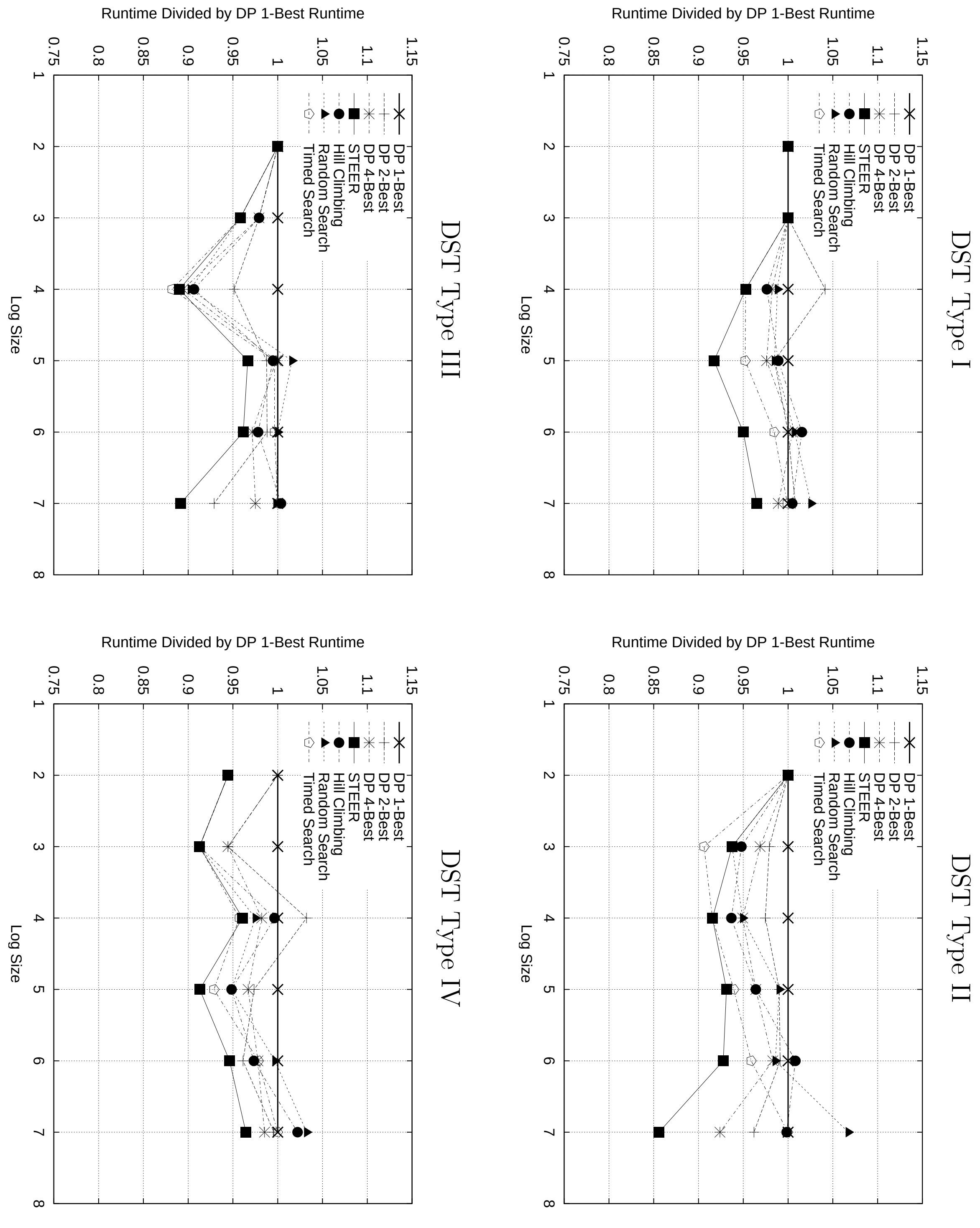


Figure 3.4: Comparison of the runtimes of the fastest formulas found by different search methods for small sized DSTs on a Pentium.

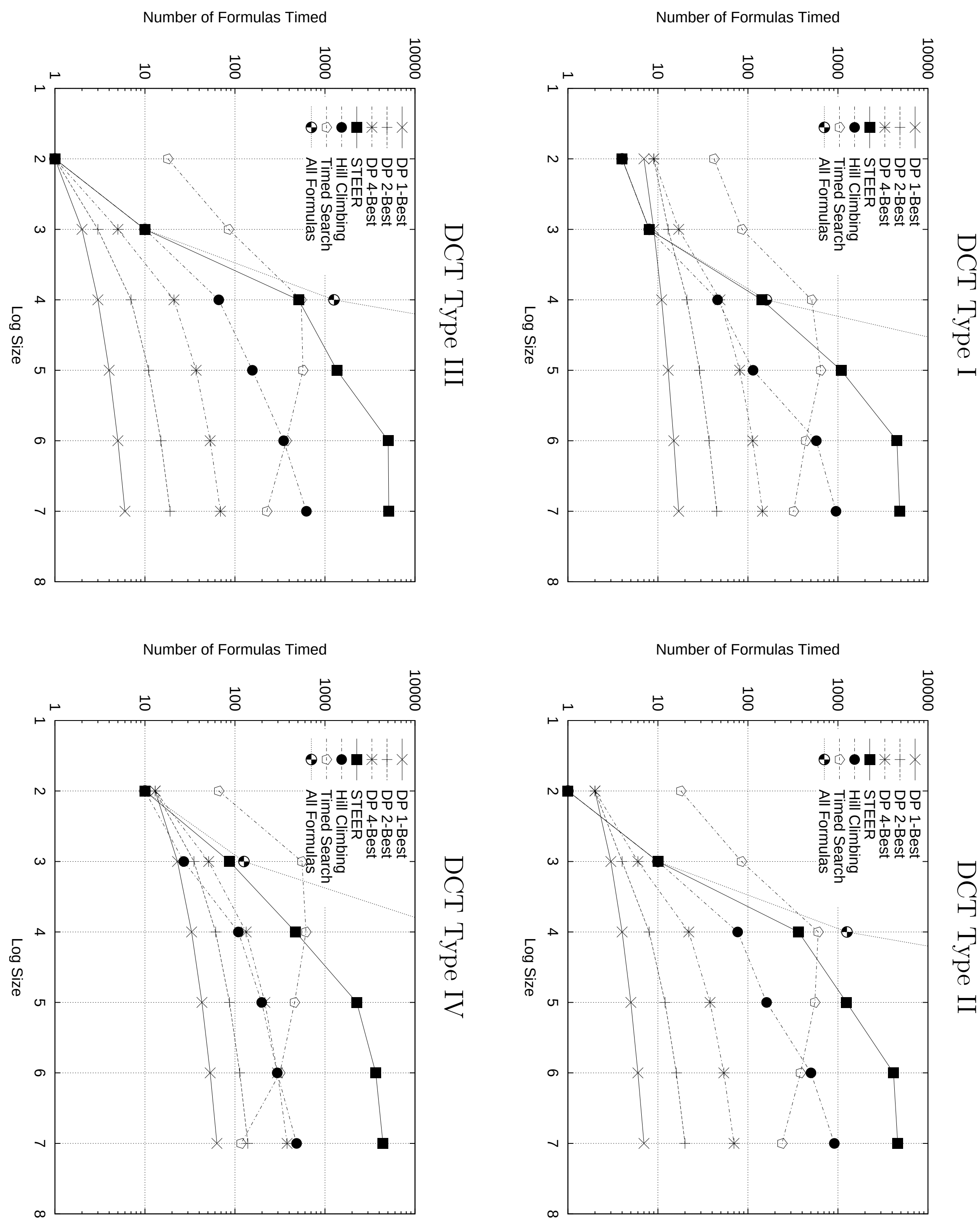


Figure 3.5: Number of formulas timed by different search methods for small sized DCTs on a Pentium.

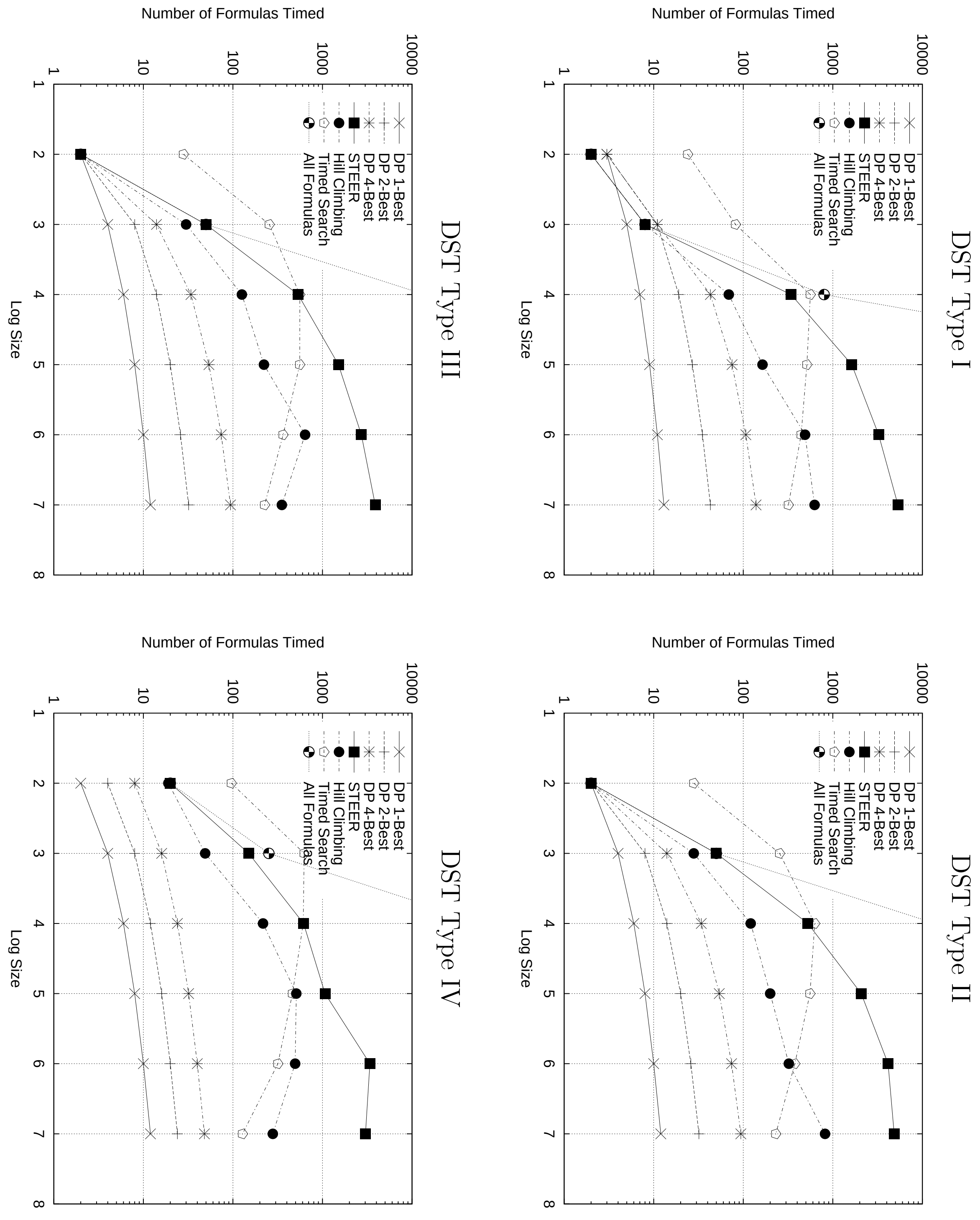


Figure 3.6: Number of formulas timed by different search methods for small sized DSTs on a Pentium.

3.5.2 Local Unrolling Search

Figures 3.7 and 3.8 compare several of the search methods when also searching over local unrolling settings for the DFT and WHT. STEER and 1-best and 4-best dynamic programming were run on the same Pentium machine as the previous experiments but this time allowed to search over local unrolling settings (see Section 3.2.2). The plots compare these new runs against the previous runs that used a fixed global unrolling setting.

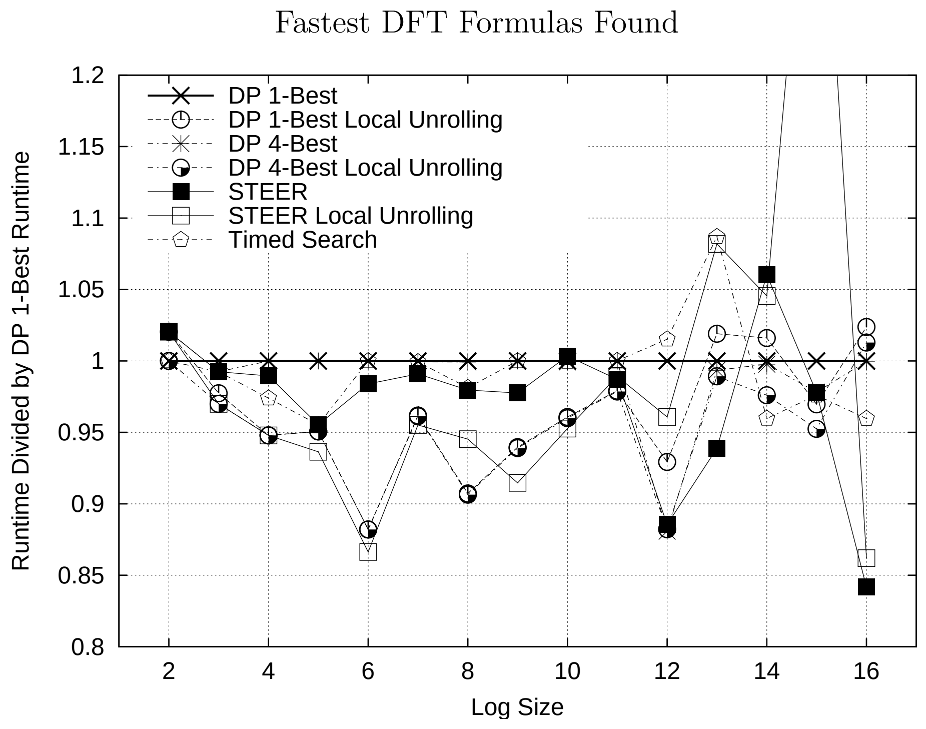
For most sizes of the WHT and many of the smaller sizes of the DFT, the searches that were allowed to search over local unrolling settings found faster implementations than those that used a fix global unrolling setting. However, each search method normally timed more formulas when searching over local unrolling settings than if the same search method used a fixed global unrolling setting. Thus, if time is available to search longer, allowing a search over local unrolling settings can produce faster implementations.

3.5.3 Sun Results

Figure 3.9 compares several of the different search methods finding fast FFT implementations for a Sun UltraSparc IIi 300 MHz. Again, no one search method outperforms all of the others, but usually either STEER or 4-best dynamic programming performs the best. As this is a slower machine, timed search tends to perform poorly at smaller sizes than on the Pentium since it again was only allowed thirty minutes to search.

3.6 Summary

This chapter has presented the SPIRAL system's search engine which we have designed and implemented. The search engine includes several different search methods, including exhaustive search, dynamic programming, random search, hill climbing search, STEER, and timed search. This variety of search methods allows the user to search different portions of the very large search space. Each of these search methods are implemented in the SPIRAL system with an easy user interface. They have many options that can be set, but reasonable defaults are provided for all options. Most of



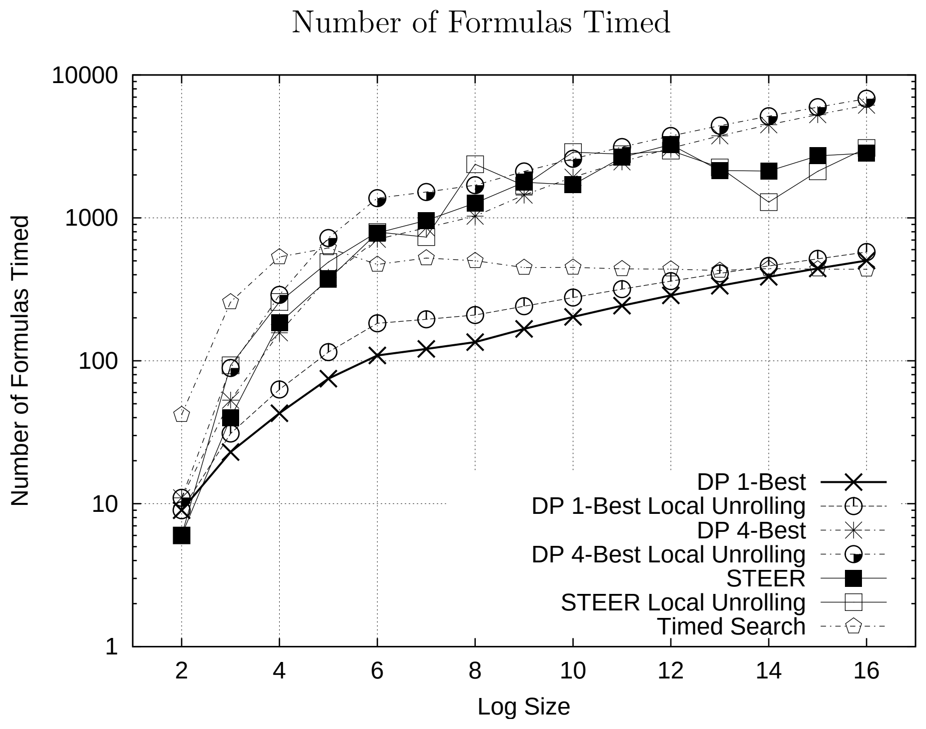
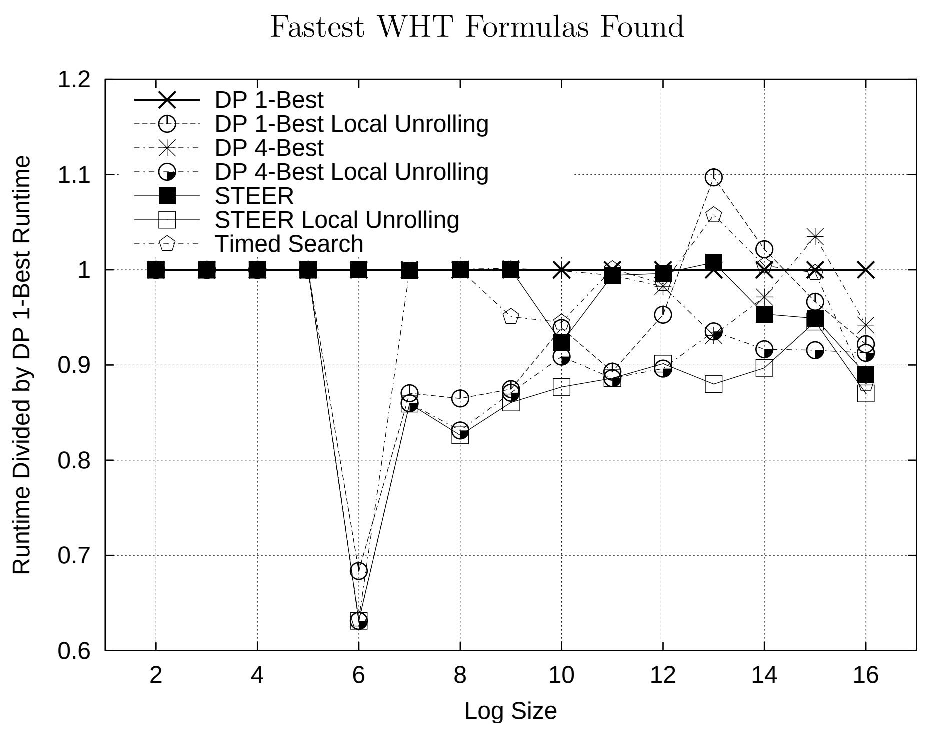


Figure 3.7: Comparison of search methods searching over local unrolling settings for the DFT on a Pentium.



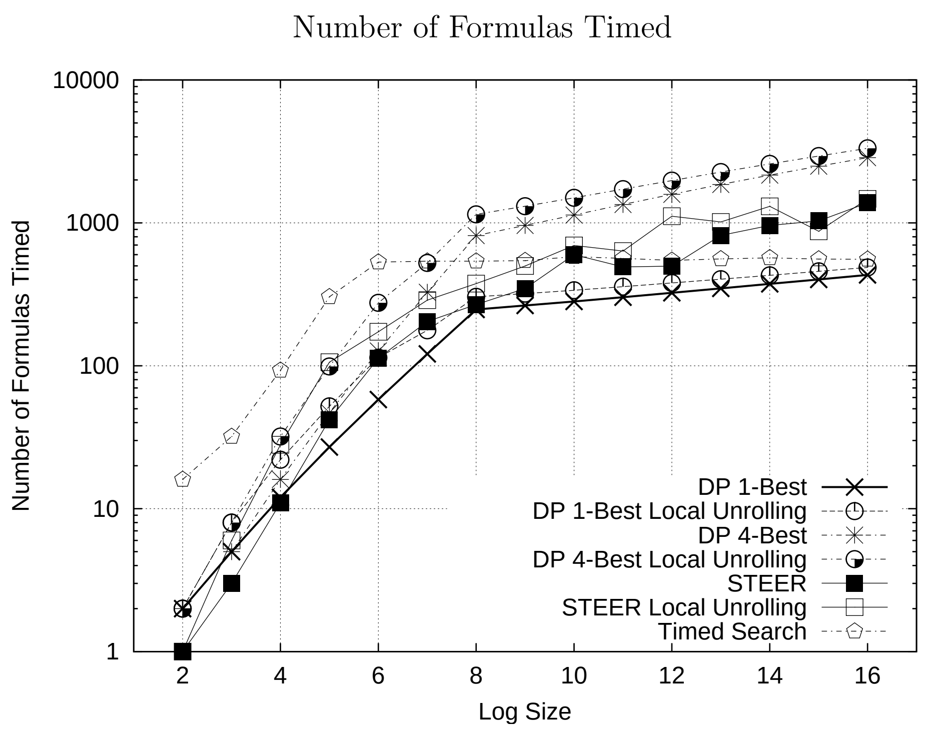


Figure 3.8: Comparison of search methods searching over local unrolling settings for the WHT on a Pentium.

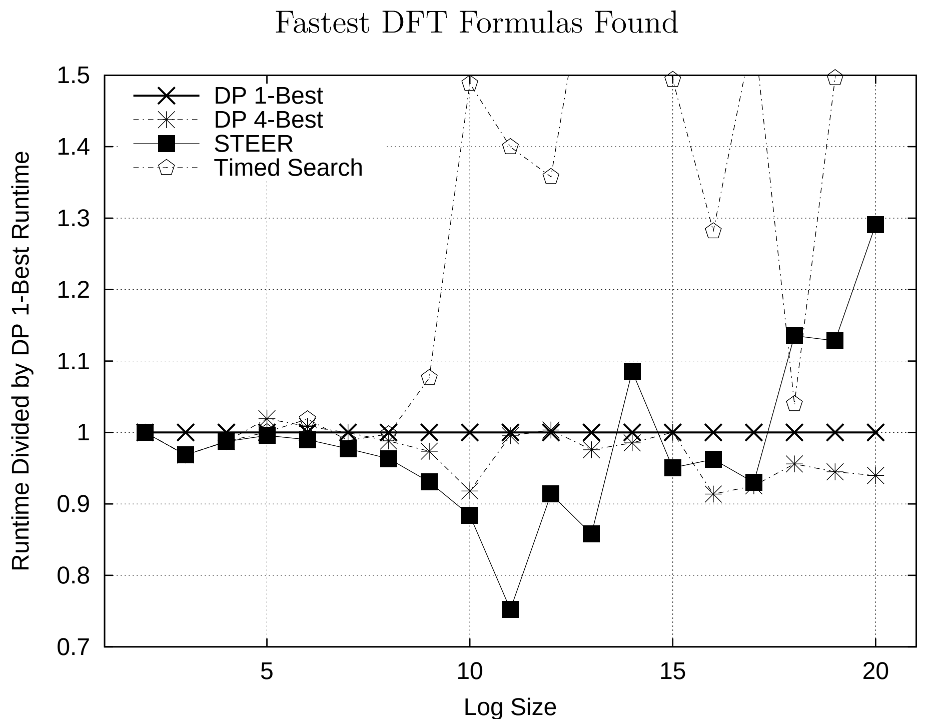


Figure 3.9: Comparison of search methods for the DFT on a Sun.

the search methods can not only search over different formulas but also over options to the formula compiler.

We have also presented results comparing the different search methods. We have found that for the FFT and the WHT on a Pentium and on a Sun no one search method outperforms all of the others, but for any given transform and size either STEER or 4-best dynamic programming often finds the fastest formula. However, for small sized DTTs, STEER outperforms all of the other search methods at finding the fastest implementations. We also found that allowing the search methods to search over local unrolling parameters usually improved the performance of the best found implementations.

Chapter 4

Optimizing Performance with STEER

This chapter presents STEER, an evolutionary algorithm for searching for the optimal implementations of signal transforms. STEER stands for Split Tree Evolution for Efficient Runtimes. We initially developed STEER specifically for the WHT. Then, as the SPIRAL system developed, we re-implemented STEER in the search engine of the SPIRAL system (see Chapter 3), extending STEER to work with arbitrary, user-defined transforms.

We begin in Section 4.1 with a description of STEER as it was implemented to optimize WHTs. Section 4.2 compares STEER against other search methods for optimizing WHTs. Then, Section 4.3 presents how we modified STEER to optimize arbitrary signal transforms in the SPIRAL system. This section includes many details about STEER's implementation in the SPIRAL system and its many features.

4.1 STEER for the WHT

Our first implementation of STEER explicitly searched for optimal WHT formulas. When STEER was first implemented, the SPIRAL system had not yet been designed. So, STEER also used the WHT package (see Section 2.3) to implement and time WHT formulas.

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 very similar to a standard genetic algorithm (Goldberg, 1989; Mitchell, 1996) 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 runtime.
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 runtime is expensive, runtimes are cached and only new split trees in P at step 2 are actually run.

4.1.1 Tree Generation and Selection

Random tree generation produces the initial population from which the algorithm searches. Note that legal split trees are not just arbitrary trees but rather have specific structure. In a WHT split tree, all of the base two logarithms of the sizes of the children of any node must sum to the base two logarithm of the size of the parent. Random tree generation must always produce legal split trees. While the method STEER uses does not uniformly generate random split trees over the space of possible split trees, it is able to quickly generate a random split tree. Recall that each node in the split tree is labeled with the base two logarithm of the size of the WHT at that level. Then, the sum of the labels of the leaves of a split tree is the root node's label since the sum of the labels of the children of any node in the tree must be that node's label. Thus, random leaves can be generated until the sum of the labels of all the generated leaves is the desired size. Then, a random subset of the leaves can be chosen and made to be children of a new node. A random subset of the remaining leaves and this new subtree can be chosen and the process repeated until just a single tree with all the originally generated leaves has been created.

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 runtime. This method weights trees with faster runtimes more heavily, but allows slower trees to be selected on occasion. The faster a tree's runtime, the more likely more copies of that tree will appear in P_{new}.

4.1.2 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 (Goldberg, 1989).

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 4.1 produces the trees (c) and (d).

4.1.3 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

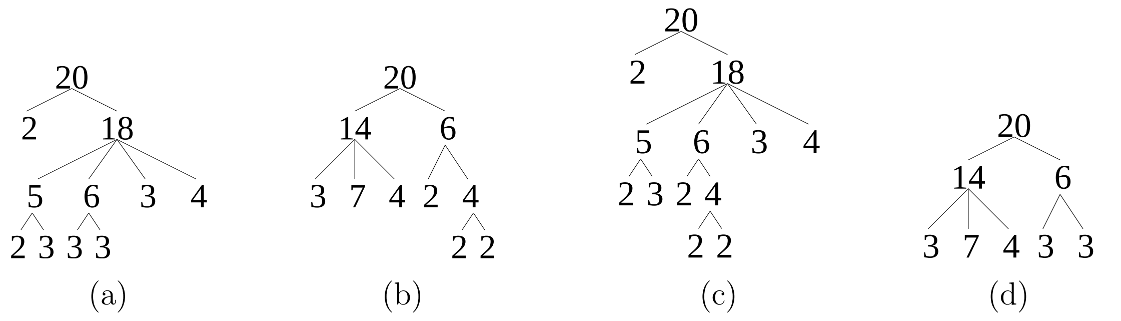


Figure 4.1: Crossover of trees (a) and (b) at the node of size 6 produces trees (c) and (d) by exchanging subtrees.

perform even better. Mutations provide a way to search the space of similar split trees (Goldberg, 1989).

We present the mutations that STEER uses with the WHT. A subset of the mutations listed here could be used to potentially move from any split tree of a given size to any other of that same size. Except for the first mutation, all of them come in pairs with one essentially doing the inverse operation of the other. Figure 4.2 shows one example of each mutation performed on the split tree labeled "Original." The mutations are:

- Flip: Swap two children of a node. The nodes of size 4 and 5 have been flipped in the example.
- Grow: Add a subtree under a leaf, giving it children. The node of size 4 in the example has had a subtree grown underneath it.
- Truncate: Remove a subtree under a node that could be a leaf, making the node a leaf. The node of size 6 in the example has been truncated.
- Up: Move a node up one level in depth, causing the node's grandparent to become its parent. If this leaves a node with a single child, replace that node with the single child. The node of size 5 in the example has been moved up.
- Down: Move a node down one level in depth, causing the node's sibling to become its parent. If this leaves a node with a single child, replace that node with the single child. The node of size 6 in the example has been moved down so that its old sibling of size 5 became its parent.

- Join: Join two siblings into one node which has as children all of the children of the two siblings. To prevent this mutation from changing the depths of nodes, STEER limits it to siblings which are not leaves and which have a third sibling. The nodes of size 5 and 6 have been joined in the example.

- Split: Break a node into two siblings, dividing the children between the two new siblings. To prevent this mutation from changing the depths of nodes, STEER limits it to non-root nodes with at least 4 children so that the new siblings can have at least 2 children each. The node of size 18 has been split into two nodes in the example.

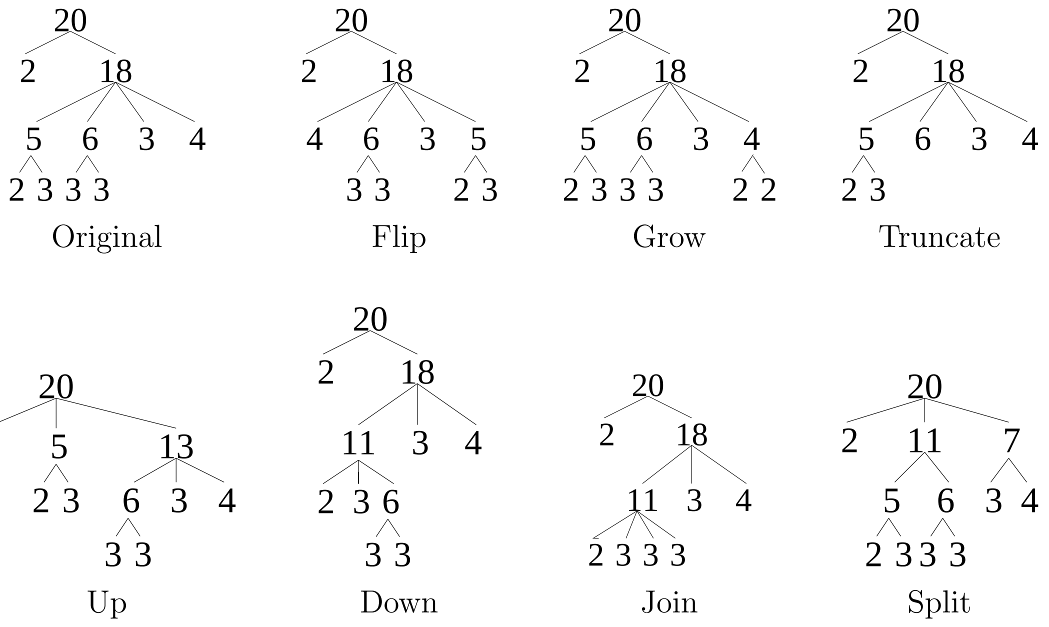


Figure 4.2: Examples of each kind of mutation, all performed on the tree labeled "Original."

The following theorem shows that from any valid split tree, the entire search space of WHT split trees can be explored through a sequence of mutations.

Theorem: From any arbitrary valid WHT split tree, it is possible to obtain any other arbitrary valid WHT split tree of the same size through a sequence of Grow, Truncate, Up, and Down mutations.

Proof: It suffices to show that a sequence of the specified mutations can transform any node with arbitrary children and subtrees in a valid split tree to a node of the same size with any specified valid children ignoring the subtrees under the new children and without changing rest of the split tree. Given such a transformation, the desired transformation on the entire split tree can be obtained by first producing the desired children of the root and then recursing on each of the new children of the root.

First, if the desired transformation is for the node to become a leaf with no children, then Truncate performed on that node would produce the desired result. Second, if the original node has no children, then a Grow mutation can produce the desired children. Otherwise, the following sequence of mutations would allow such a transformation:

1. Generate all 1's as leaves. A sequence of Grow mutations performed on all of the leaves of the subtrees of the original node can produce subtrees with 1's as the only leaves.
2. Let the original node only have 1's as children. A sequence of Up mutations performed on every leaf 1 that is not an immediate child of the original node but that is in the subtree rooted at the original node will eventually cause the original node to only have leaf 1's as children. Note that when an Up mutation is performed on a leaf 1 that only has a single sibling, then this only child also becomes a child of its grandparent with the parent disappearing.
3. Produce desired children. A sequence of Down mutations performed on the leaf 1 children of the original node will produce any desired valid set of children. Specifically, for each child of size n, a sequence of $n-1$ Down mutations that move a leaf 1 child of the original node onto the growing sibling produces a child of the original node of size n which has n immediate children all of which are leaf 1's. The first Down mutation that needs to be performed for each desired child requires that a leaf 1 child of the original node be made a child of a sibling leaf 1, causing both leaf 1's to become children of a new node of size 2.

While this theorem shows that any split tree in the space of possible split trees is reachable through a sequence of mutations from any other split tree, the sequence of mutations presented in the proof would be very long and would be likely to generate many trees with very poor runtimes along the way. The other mutations that STEER uses allow for other similar split trees to be reached more quickly from a given split tree, even though these additional mutations do not increase the space of possible split trees that are attainable through a sequence of mutations.

4.1.4 Running STEER

There is a large number of parameters that can be adjusted in STEER, including the population size, the number of fastest formulas to be kept, the number of formulas to be mutated or crossed over, and the number of generations to be run can all be adjusted. For most of the experiments presented in Section 4.2, the following values were used. A population of size 100 was used with 20 of the fastest formulas kept after each generation, 10 pairs of formulas crossed, and 20 formulas mutated. The algorithm was run for 200 generations or stopped earlier if 75 generations passed without it finding a faster formula than its current best.

Figure 4.3 shows a typical plot of the runtime of the best formula (solid line) and the average runtime 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. The y-axis displays the runtime in cycles and the x-axis displays the generations as the population evolves. The average runtime of the first generation that contains random formulas is more than twice the runtime of the best formula at the end, verifying the wide spread of runtimes of different formulas. Further, both the average and best runtimes decrease significantly over time, indicating that the evolutionary operators are finding better formulas.

4.2 Search Algorithm Comparison for WHT

To evaluate STEER's performance on finding fast WHT formulas, we have compared it against a number of different search methods. Some of these search methods have been specifically adapted to searching for fast WHT formulas. However, both STEER and the random search method search through the full space of possible WHT formulas.

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 many cases, the fastest WHT formulas never have leaves of size 2^1. By

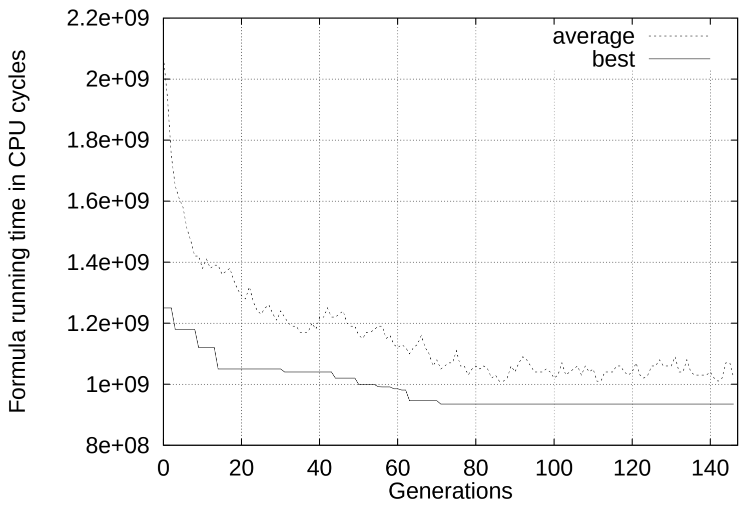


Figure 4.3: Typical plot of the best and average runtime of formulas as STEER evolves the population.

searching over just binary split trees with no leaves of size 2^1, the total number of trees that need to be timed can be greatly reduced, but still becomes intractable at larger sizes.

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 very 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 to be timed.

Figure 4.4 compares the best runtimes found by a variety of search techniques on a Pentium III 450 MHz running Linux. 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 4.5 compares the number of formulas timed by each of the search methods. A logarithm scale is used along 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

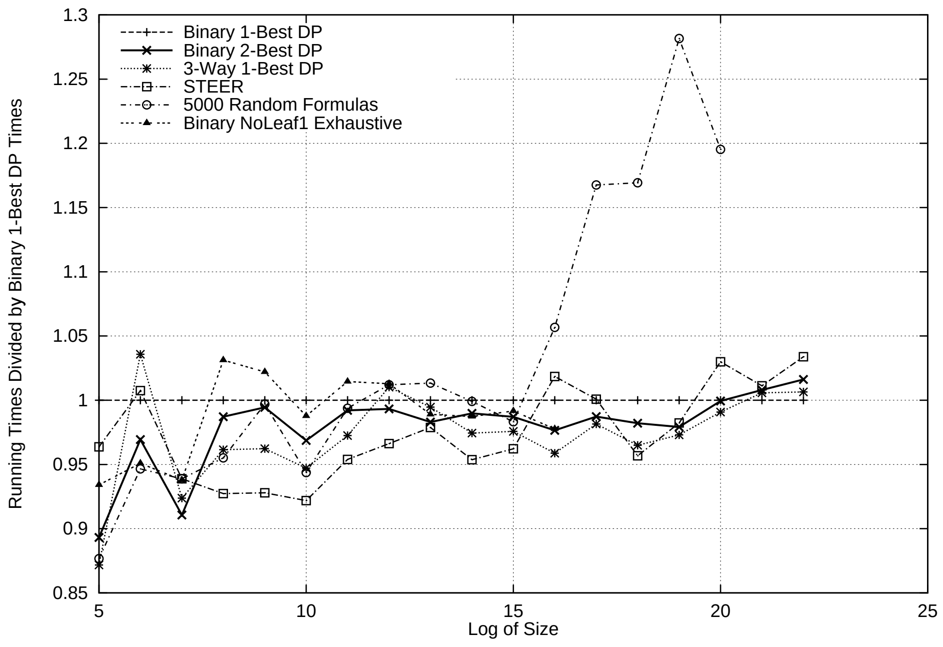


Figure 4.4: Comparison of best WHT runtimes found by several search methods.

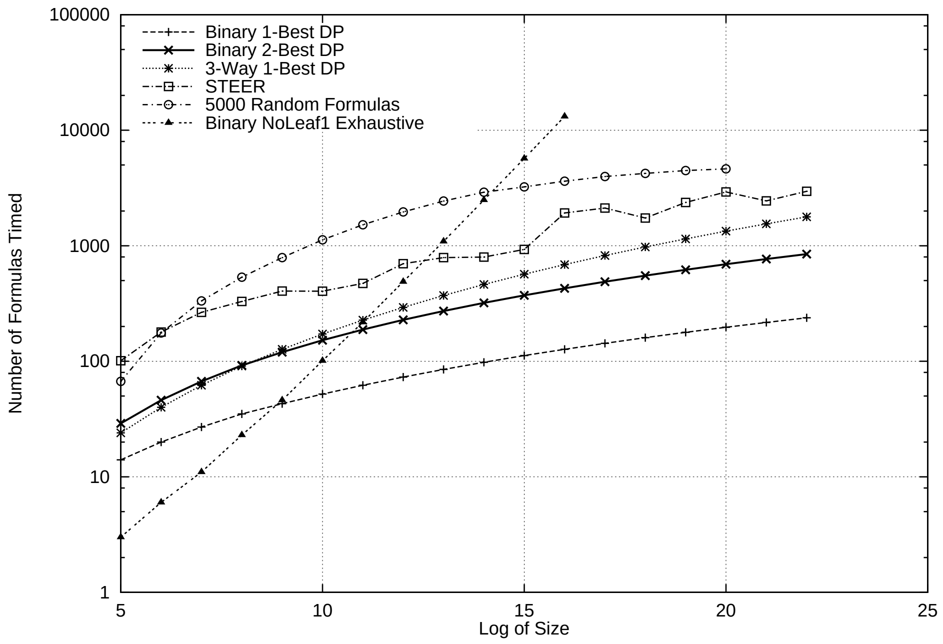


Figure 4.5: Comparison of the number of WHT formulas timed by several search methods.

by the exhaustive search method grows much faster than all of the other techniques, indicating why it quickly becomes intractable for larger sizes. Except for the exhaustive search, the random formula generation method times many more formulas than all the other methods and still does not find nearly as fast formulas as the other methods. Clearly plain binary dynamic programming has the advantage that it times the fewest formulas.

However, while binary dynamic programming found fast formulas while timing relatively few formulas in the previous results, sometimes dynamic programming does not perform as well. Figure 4.6 shows two different runs of binary dynamic programming on the same Pentium machine. 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.

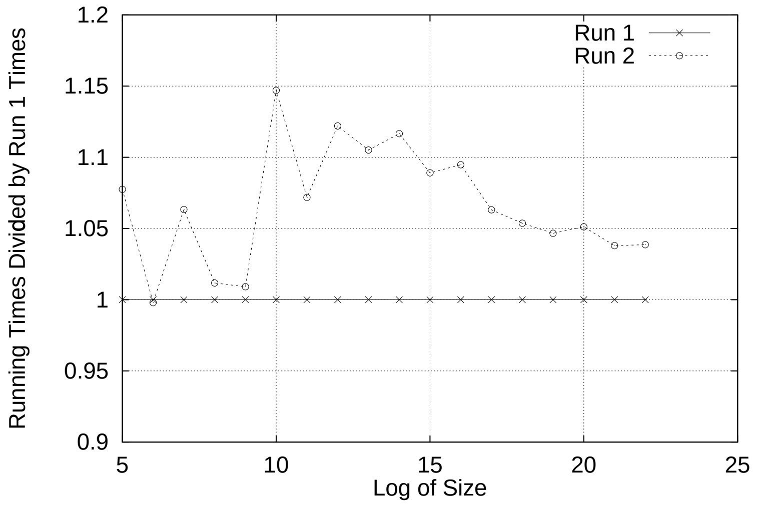


Figure 4.6: Two runs of dynamic programming.

An analysis of these two runs and several other runs on this same machine shows that the major difference depends on the split tree chosen for size 2^4. The two fastest split trees for that size have close runtimes. 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. This strongly argues for using a k-best dynamic programming instead of a plain dynamic programming. Unfortunately, even if k is only 2, this more than doubles the number of formulas that must be timed. Further, a similar problem can arise for any chosen k if the dynamic programming assumption does not hold, but a larger k can help simple cases like the one discussed above.

While this specific result is particular to the machine we were using, it demonstrates a general problem with dynamic programming. There may be several formulas for small sizes that all run about equally fast. However, one formula may run considerably faster as part of a larger split tree than the others. So, if dynamic programming happens to choose poorly for smaller sizes early in its search, then it can produce significantly worse results at larger size than it would if it had choose the right formulas for smaller sizes.

Of the search methods compared, 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. STEER also finds fast formulas but is not as strongly impacted by poor initial choices.

4.3 STEER in the SPIRAL System

With the development of the SPIRAL system, we ported STEER to work in the SPIRAL system. Not only did this involve a rewrite of the code from C to GAP, but also several significant modifications to the evolutionary operators. STEER for the WHT relied heavily on several properties of the WHT. We modified STEER so that it could work with any arbitrary, user-specified transform that could be specified to the SPIRAL system. Further, we extended STEER to take advantage of many of the features available in the SPIRAL system, including unrolling search and time limits.

4.3.1 Tree Generation and Selection

A new method for generating a random split tree was needed that could work with arbitrary transforms. For a given transform, all applicable break down rules are found, and a random one is chosen to be used. This break down rule can potentially produce many different sets of possible children, and so a random set is chosen. This is then repeated recursively for each of the transforms of the children.

This random tree generation method also does not uniformly generate random split trees over the space of possible split trees. For example, any applicable rule is equally likely to be chosen for a transform as another while some applicable rules may generate only one set of children while others potentially generate many different sets of children. However, this method is quick and works for any transform and set of break down rules.

If the user specifies to search over global unrolling settings, then a random global unrolling setting is generated for each split tree that is randomly generated. Currently, this is a power of two size between the specified minimum and maximum values inclusive (chosen uniformly over such two power sizes).

If the user specifies to search over local unrolling settings, then the random split tree generation method must also mark nodes in the split tree for local unrolling. In particular, any node of the specified minimum local unrolling size or smaller is marked for local unrolling. Further, recursing from the root, any node encountered that is not larger than the maximum local unrolling size has a 50% chance of randomly being marked for local unrolling (and thus causing the entire subtree under that node to be locally unrolled).

The selection process remains the same as the selection process for the WHT.

4.3.2 Crossover

Generally, crossover remains the same except the definition of equivalent nodes. Now instead of looking for split tree nodes of the same size, crossover must find nodes with the same transform and size. Now for crossover, the subtrees beneath two nodes of the same transform and size are swapped between two different split trees.

However, if the user specifies to search over global unrolling values, then a crossover of two implementations randomly could swap their global unrolling values instead of subtrees. Crossover randomly chooses between crossing over subtrees and crossing over global unrolling values when both choices are available, favoring crossing over subtrees 4 to 1.

No extra work is performed by crossover if the user specifies to search over local unrolling values. Since local unrolling is specified in the split tree, when subtrees are swapped, their local unrolling values are also swapped. However, if a subtree does not have local unrolling specified for its root node, but the new parent of the subtree does, then by default the entire subtree becomes locally unrolled.

4.3.3 Mutation

We developed a new set of mutations since 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 4.7 shows an example of each of these mutations.

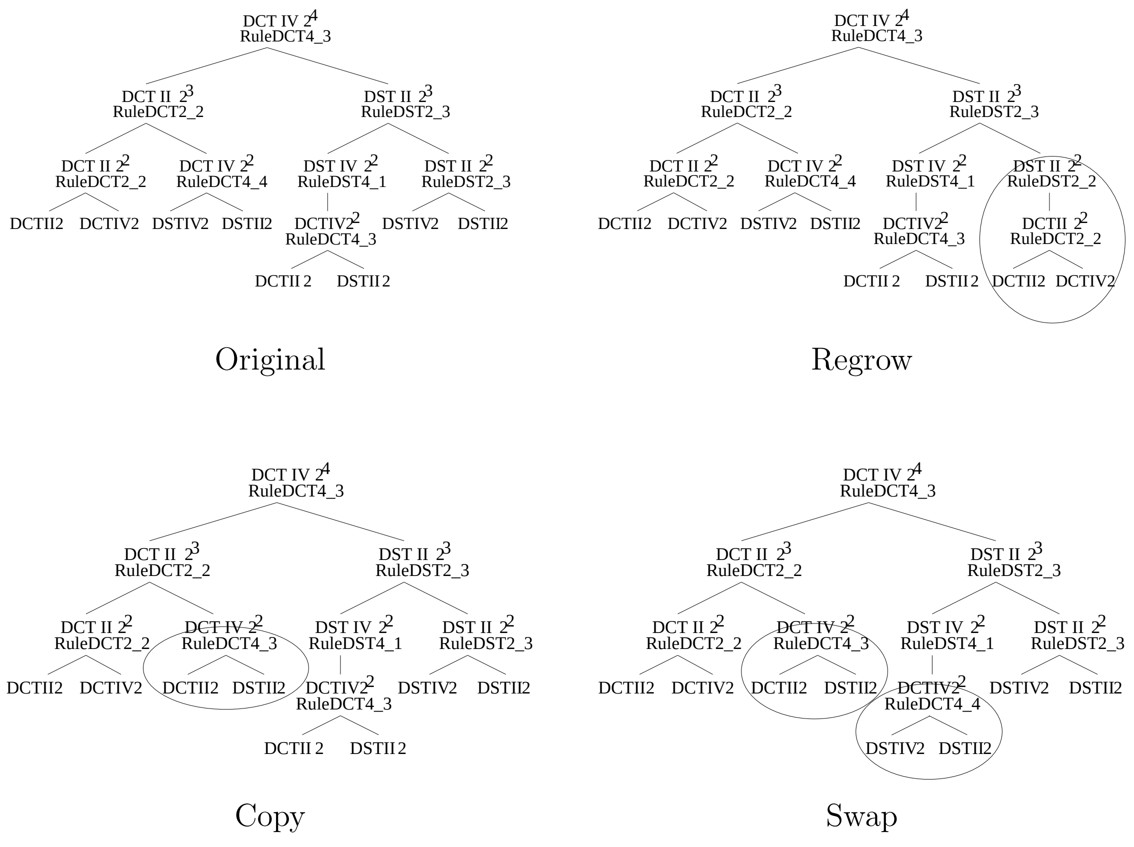


Figure 4.7: Examples of each general mutation, all performed on the tree labeled “Original.” Areas of interest are circled.

If the user specifies to search over global unrolling values, then a fourth mutation is added to the main three. This mutation randomly doubles or halves the global unrolling setting associated with the given split tree (as long as this keeps the global unrolling setting within the specifies minimum and maximum values).

If the user specifies to search over local unrolling parameters, then a different fourth mutation is added to the main three. This mutation randomly moves a local unrolling specification up or down one level in the split tree. More specifically, it randomly does one of the following:

- Randomly chooses a node that is locally unrolled but is larger than the minimum local unrolling size and whose parent is not locally unrolled. It then removes the local unrolling for this node.
- Randomly chooses a node that is not locally unrolled but that could be and who has at least one child that is locally unrolled. It then marks this node for local unrolling.

4.3.4 Other User Options

In the SPIRAL system, there is a large number of options that the user can specify to control STEER, including:

- the population size
- the number of generations to run as well as the maximum number of generations to run without an improvement in the best found implementation
- the number of implementations to cross-over, to mutate, and to randomly generate each generation as well as the number of best implementations to keep from generation to generation
- the local and global unrolling search parameters
- a time limit — STEER can be given a time limit which causes STEER to return the best implementation that it has found thus far when that time limit expires

Any subset of these options can be specified easily, allowing the other options to retain their default values.

See the search engine user manual in Appendix A for further details.

4.4 Results Using STEER in the SPIRAL System

Many results from using STEER in the search engine have already been presented in Section 3.5 and so no duplicated here. For small sized DTTs, STEER outperformed all of the other search methods tested. However, for larger sized DFTs and WHTs, STEER's results were mixed, sometimes outperforming the other search methods but sometimes not performing as well.

4.5 Summary

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 in the SPIRAL system. This later form of STEER is able to optimize arbitrary transforms, even user-defined transforms that it has never before seen.

Chapter 5

Modeling Performance of Entire Formulas

Modeling performance of signal processing algorithms is important. Performance models can restrict the space of implementations to be searched to those that are most promising or provide information about portions of the space that need not be searched because they are the least promising. Further, performance models can provide confidence that the fastest formulas have been found.

However, it is very difficult and time consuming for a human to develop an accurate model of performance for signal processing algorithms. The complexity of modern processors makes it very hard to understand the performance of even simple pieces of code. Further, even if a good performance model could be developed for one architecture and transform, it would still be difficult and time consuming to develop models for other transforms and particularly other architectures.

Fortunately, empirical performance data can be gathered for a variety of formulas. This data offers an interesting opportunity to use machine learning techniques to automatically learn to predict performance. The methods that we have developed can automatically generate a performance model given some empirical performance data gathered on a machine of interest.

This chapter discusses our initial work in automating the modeling of performance of signal transforms. The chapter focuses on the WHT, but similar results were obtained for the FFT (Singer and Veloso, 2000a). Unfortunately, this work with the FFT used a package for timing FFTs that implemented them in less than the most optimal ways, and thus this work with the FFT is not reported here. However, in part of Chapter 6, we discuss modeling performance of FFTs, using the SPIRAL package to produce good implementations.

Chapter 6 reports on some more recent modeling work that takes a different approach. In the current chapter, we focus on modeling the performance of entire formulas. In Chapter 6, we focus specifically on modeling performance of subportions of a formula, and then using performance predictions for the subportions to build a prediction for the entire formula.

A major step in automating the modeling of performance of signal transforms involves the selection of good features to represent a formula. In Section 5.1, 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. In Section 5.2, we describe how a function approximator can be used to learn to predict runtimes for formulas. We show that for small sizes this method can produce excellent results.

5.1 Features for WHT Split Trees

To automatically learn to predict runtimes of formulas by using machine learning techniques, one of the major steps is to be able to represent formulas with features. The problem of feature selection in machine learning is important. 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 runtimes? Or, equivalently, what are good features for predicting a formula's runtime? To answer these questions, we introduce several different feature sets to describe WHT formulas. We then compare the feature sets along several measures to see how well the features can differentiate formulas with different runtimes.

5.1.1 Feature Sets

This section describes several feature sets for WHT formulas. In almost 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 (Singer and Veloso, 2000a). 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 runtime.

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 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).

- **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 5.1. So, a simple refinement of the previous two feature sets is to include both.

- **Left/Right Leaf Nodes.** 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, 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.

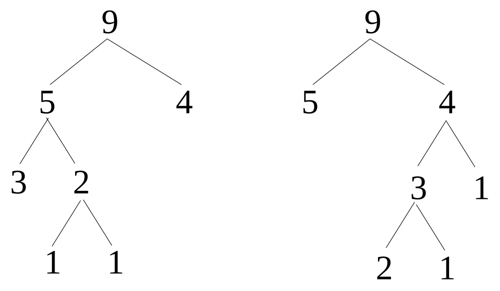


Figure 5.1: Two split trees with the same All Nodes counts but different Leaf Nodes counts.

- **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 5.1 gives an example of each feature set for the split tree shown in Figure 5.2. Each row in the table 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.

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. The single number feature Leftness is the leftness of a tree.

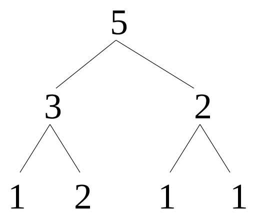


Figure 5.2: Example Split Tree.

Table 5.1: Example values of the different node count feature sets for the tree shown in Figure 5.2.

	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

- 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

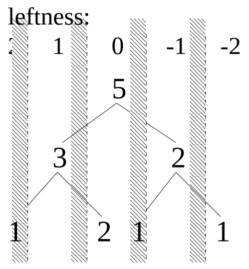


Figure 5.3: Leftness of nodes in example tree of Figure 5.2.

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 5.2 has nodes with leftness as shown in Figure 5.3, and a total leftness of 0 since it is balanced.

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 5.2 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

5.1.2 Evaluating Features

Given that we have defined a number of different feature sets, an important task is to evaluate them. We have used a few different methods to perform these evaluations.

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 5.1 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 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 runtimes. 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 5.2. 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 Table 5.2 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 5.2 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.

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 runtimes. A good set of features can separate formulas with significantly different

Table 5.2: 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

runtimes into different partitions so that all formulas within a single partition have close to the same runtime. 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 t_i be the runtime of formula i.
- Let m_k be the mean runtime of the formulas in P_k. Then, $m_k = \frac{1}{|P_k|}\sum_{i\in P_k} t_i$.
- Let σ_k be the standard deviation of the runtimes 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 runtimes 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 runtimes of formulas in the same partition are. The maximum relative standard deviation indicates how far apart runtimes of formulas are in the *worst* partition.

Evaluating the feature sets, the weighted average and maximum relative standard deviations are shown in Table 5.3. For each WHT size shown in the table, formulas for all possible binary split trees were generated. These formulas were timed using a WHT package (Johnson and Püschel, 2000) on a Pentium III 450 MHz running Linux.

Looking at the top portion of Table 5.3, we see that using the Left/Right features tends to improve the standard deviation. 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}.

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

Table 5.3: 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

surprisingly impressive. Not only do these feature sets produce good relative standard deviation results, but they do so with relatively few partitions.

5.2 Learning to Predict WHT Performance

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 runtimes of new formulas. Note that this still does not solve the problem of searching through a large space of potential formulas. However, a predicted runtime can now be obtained much more quickly than we could have obtained an actual runtime.

While accurately predicting a formula's runtime 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 runtime if it can accurately predict which of two formulas runs faster.

5.2.1 Experimental Setup

The results that are presented in this section are for $WHT(2^8)$ and are similar to those collected for other sizes, up through 2^{10}. All 2950 possible formulas corresponding to binary trees of $WHT(2^8)$ were generated and timed using a WHT package (Johnson and Püschel, 2000) on a Pentium III 450 MHz running Linux.

We used a back-propagation neural network as the function approximator. For all of the results presented, we used 50 hidden 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 (runtime 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, 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 runtimes of formulas, and (2) to predict which of two formulas would run faster.

5.2.2 Results

Results are shown in Table 5.4. The column marked "Cost" reports the runtime prediction error on the test set:

- Let c_i be the actual runtime of formula i.
- Let p_i be the predicted runtime 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:

- Let m_i be 1 if the network incorrectly predicts which of pair i's formulas run faster and 0 otherwise.
- Then the percent error on predicting the faster of two formulas is $\frac{\sum_{i \in formula-pairs} m_i}{|formula-pairs|}$

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 runtimes 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 runtimes 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.

Table 5.4: Neural network prediction accuracy for $WHT(2^8)$ with node counting features. The column marked "Cost" is the average percent error on predicting runtime. 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

5.3 Summary

To model performance of signal processing algorithms, we have explored using machine learning to learn to predict runtimes of formulas. In order to use standard machine learning techniques, we have developed feature sets to describe split trees representing signal transform formulas. We have explored a variety of feature sets, identifying different feature sets with different abilities to partition formulas according to their runtimes. Further, some simple feature sets do well at partitioning the space of formulas according to their runtimes. By describing formulas with features, we can present formulas to a function approximator. 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 runtime of a formula given a limited set of training data.

Chapter 6

Modeling Performance of Individual Nodes

In Chapter 5, we discussed methods for modeling performance of entire WHT formulas. In this chapter we take a different approach, looking at how we can model performance of individual nodes in both WHT and DFT (FFT) split trees. For the WHT, we consider only the leaves of the split trees, but for the FFT, we must consider the internal nodes as well. By first learning to successfully predict performance for individual nodes, these predictors can then be used to accurately predict for entire formulas by summing their predictions for all of the nodes in a formula's split tree.

We begin in Section 6.1 with a number of key observations that we made for the WHT on a Pentium. These observations led us to develop methods for predicting cache misses for WHT leaves as discussed in Section 6.2. Next, Section 6.3 discusses our observations for the WHT on a Sun. These new observations directed us to develop methods for predicting actual runtimes for WHT leaves as discussed in Section 6.4. Then, Section 6.5 discusses some of the difficulties in extending these methods to the FFT. Finally, Section 6.6 presents methods for predicting runtimes for FFT leaves and internal nodes.

6.1 Pentium Observations

This section discusses several important observations that we made about the WHT that directed our research. Specifically, these observations were made on a Pentium III

450 MHz running Linux 2.2.5-15.

Figure 6.1 shows a scatter plot of runtimes versus level 1 data cache misses for all binary $WHT(2^{16})$ split trees with no leaves of size 2^1. Each point in the scatter plot corresponds to a different WHT formula. The placement of this point corresponds to that formula's runtime and cache misses. The plot shows that while there is a complete spread of runtimes, there is a grouping of formulas with similar numbers of cache misses. Both runtimes 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 runtimes for formulas with the same number of cache misses. The formula with the fastest runtime also has the minimal number of cache misses.

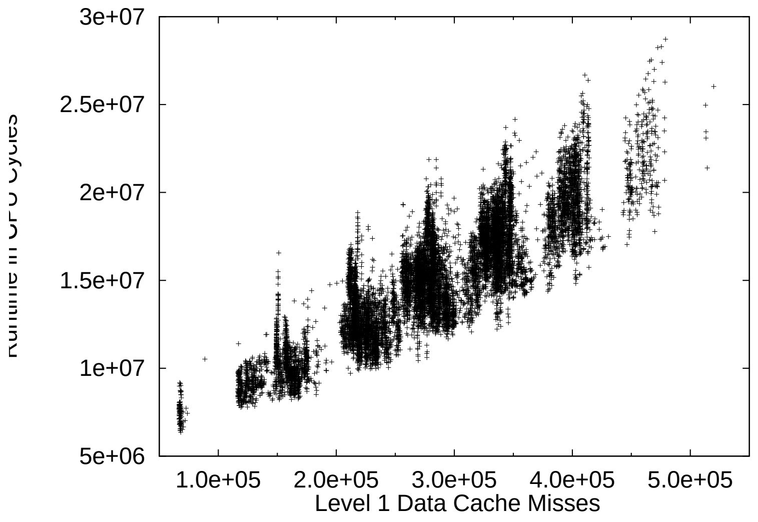


Figure 6.1: Runtimes vs. cache misses for entire $WHT(2^{16})$ formulas on a Pentium.

This observation indicates that minimizing level 1 data cache misses produces a group of fast formulas. Further, the overall fastest formula lies within this group. Again, if we can generate all of the formulas with minimal cache misses, then we will have a much smaller set of formulas to time to determine the one with the fastest runtime.

The second key observation is that all of the runtime and cache misses occur in computing the leaves. The WHT package we are using (Johnson and Püschel, 2000) implements leaf WHTs as unrolled, straight-line code. There is no work necessary to combine the leaf WHTs. The recursive calls to children from an internal node in the

WHT package simply specify which portions of the input and output data vectors are to be operated on when calculating a smaller WHT. Additionally, the total run time and number of cache misses of a formula is simply the sum of the runtime and cache misses at each of the leaves.

This observation allows us to consider simply modeling performance of individual leaves. The performance of an entire formula could then be predicted by summing up the individual predictions for the leaves.

Figure 6.2 shows a histogram of the number of level 1 data cache misses incurred by *leaves* of all binary $WHT(2^{16})$ split trees with no leaves of size 2^1. For all of 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.

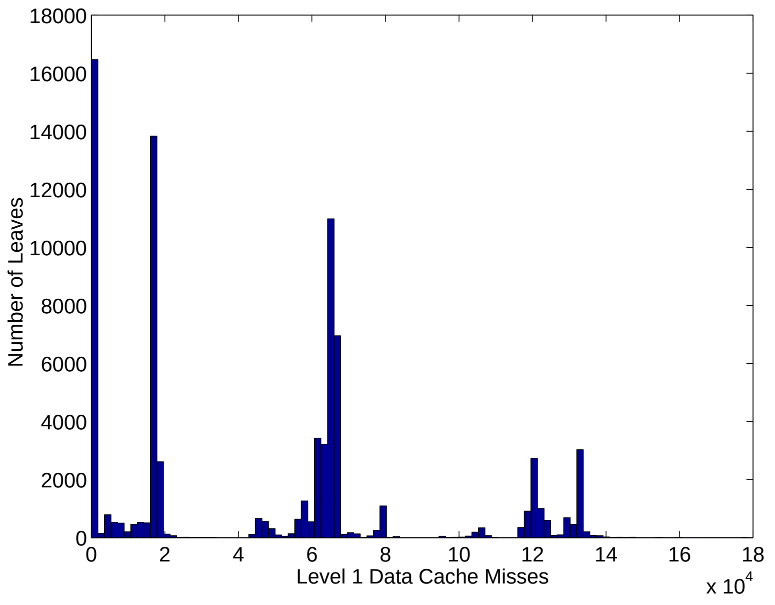


Figure 6.2: Histogram of the number of cache misses incurred by *leaves* of $WHT(2^{16})$ formulas on a Pentium.

Thus, it is not necessary to predict a real valued number of cache misses, but rather only to predict the correct group of cache misses out of only four groups. This observation indicates that simple classification algorithms can be used to predict cache misses for leaves instead of needing to use a function approximator.

Finally, we have observed both in Figure 6.2 and in a number of other similar histograms for different sized WHTs that the number of level 1 data 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 incurs is:

- 0
- $s/4$
- s
- $2s$ or more.

These particular fractions correspond to particular features of the cache. To have no cache misses means that all of the data that the leaf needed was already in cache. To have as many cache misses as the size of the transform means that the leaf incurred one cache misses for each data item. On a Pentium machine, it is possible to have as many cache misses as a 1/4 of the size of the transform since a cache line holds exactly four data items. Further, it is possible to have more cache misses than data items, since a single data item may need to be accessed multiple times during a computation.

Thus, the number of level 1 data 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 runtime has the minimal number of level 1 data 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 predicting leaf cache misses allows predicting for entire formulas.
- The number of level 1 data 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 level 1 data cache misses incurred by leaves are fractions of the transform size. So learning may be able to generalize across different sizes.

6.2 Predicting WHT Leaf Cache Misses

Given the observations discussed in the previous section, we now turn to modeling level 1 data 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.

6.2.1 Features for WHT Leaves

To use standard machine learning methods to predict cache misses for WHT leaves, we need to describe the leaves with features. These features need to be able to distinguish leaves with different cache misses. The use of good features provides a source of domain knowledge about the WHT to our methods.

The features that we have decided to use came about from trying to model cache misses for leaves by hand. Trying to understand cache misses is difficult, and we were only able to understand a few simple cases by hand. However, after the attempt, we were able to write down a number of the features that we were considering when trying to model the cache misses.

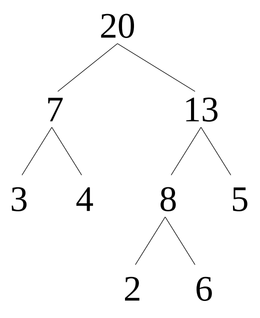
Clearly the size of the leaf is important in determining a leaf's number of cache misses, as the size indicates the size of the problem the leaf computes and the amount of data it needs to access on each call. A leaf's position in a split tree is also very important in determining its number of cache misses. The position of a leaf in a split determines at what stride it accesses its input and output data as well as the state of the cache when the leaf is called. However, it is not as easy to capture the position of a leaf in a split tree with numeric features as it is for the size of a leaf.

A leaf's stride provides some information about its position in the split tree and describes how a leaf accesses its input and output data. Cache performance is clearly effected by the stride at which data is accessed. A leaf's stride can be easily determined by its position in the split tree. The stride is simply the product of the sizes of the leaves to the right of the given leaf along the fringe of the tree. The rightmost leaf has a stride of 1. See Section 2.2.4 for further details about stride.

To provide more context of the position of a leaf in its split tree, the size and stride of the parent of the leaf can also be considered. These features indicate 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. This is due to the fact that before l is called p has been called enough times so that it has accessed as much data as the size of the common parent. Further, p has accessed its data always at a multiple of the common parent's stride, but at the appropriate initial offsets so that the total data brought in before l is called is exactly at the common parent's stride. Thus, we use the size and stride of the "common parent" in our features and not that of the previous leaf.

Figure 6.3 gives an example split tree along with the features for each of the leaves. Each line of the table corresponds to features for one leaf. The nodes in the split tree are labeled by the base two logarithms of their sizes (for convenience, no two nodes have the same size for this example). The features are actually given as base two logarithms of the sizes and strides. Further, the rightmost leaf has no common parent since no leaf is computed before it, and this is indicated by a "-1" value for the common parent features.



Leaf		Parent		Common Parent	
Size	Stride	Size	Stride	Size	Stride
5	0	13	0	-1	-1
6	5	8	5	13	0
2	11	8	5	8	5
4	13	7	13	20	0
3	17	7	13	7	13

Figure 6.3: Example leaf features for all of the leaves in the given split tree.

In summary, we use the following six 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.

6.2.2 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 shown in Table 6.1.

Table 6.1: Algorithm for learning to predict WHT cache miss categories.

1. Run a subset of WHT formulas, collecting the number of level 1 data 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 sub-section.
4. Train a classification algorithm to predict one of the four classes of cache misses given the leaf features.

While the classification algorithm predicts one of the four categories for any leaf, this can be translated back into actual 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.

6.2.3 Training

We have specifically used a C4.5 decision tree (Quinlan, 1992) as the classification algorithm in the experiments that follow, but other classification algorithms should be usable. A decision tree was used as it allows rules that are somewhat human readable to be extracted and analyzed.

We trained a C4.5 decision tree on a random 10% of the leaves of all binary $WHT(2^{16})$ split trees with no leaves of size 2^1. The actual number of level 1 data cache misses were collected for these leaves by running their trees on a Pentium.

The resulting decision tree was large with 155 nodes, and so the number of rules that C4.5 extracted from this tree was also large with 46 rules. Some of the rules are quite understandable and intuitive, but as a whole it is hard to understand the entire collection. Table 6.2 shows a few example rules, one for each category. For example, Rule C says that if the leaf is the rightmost leaf (that is, it has no common parent), then it will incur near-quarter number of cache misses. This is to be expected on a Pentium where a cache line holds 4 data items and since the rightmost leaf accesses its data at a stride of 1.

Table 6.2: A few example learned rules for predicting cache misses.

```
Rule A:
  IF
    leaf_size > 2 AND
    leaf_stride > 8 AND
    parent_stride > 0
  THEN large

Rule B:
  IF
    leaf_size <= 2 AND
    leaf_stride > 9 AND
    parent_stride > 0
  THEN near-whole
```

```
Rule C:
  IF
    common_parent_size <= 0
  THEN near-quarter

Rule D:
  IF
    common_parent_size > 0 AND
    common_parent_size <= 8 AND
    common_parent_stride <= 2
  THEN near-zero
```

6.2.4 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 6.3 evaluates the accuracy of our method at predicting the correct category of cache misses for leaves. We tested the decision tree trained from $WHT(2^{16})$ data discussed in the previous section. Specifically, we tested on leaves from different sized formulas, using all of the formulas of the different limited formula spaces discussed in Section 2.2.5. 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 6.3: Error rates for predicting cache miss category incurred by leaves.

Binary No-2^1-Leaf

Size	Errors
2^{12}	0.5%
2^{13}	1.7%
2^{14}	0.9%
2^{15}	0.9%
2^{16}	0.7%

Binary No-2^1-Leaf Rightmost

Size	Errors
2^{17}	1.7%
2^{18}	1.7%
2^{19}	1.7%
2^{20}	1.6%
2^{21}	1.6%

We used the same decision tree as in the previous experiments to predict cache misses for *entire* formulas. This was done by using the decision tree to predict a category for each leaf in the split tree. These categories were transformed into an actual number of cache misses as discussed in Section 6.2.2. Then the final prediction for the entire formula was made by summing the predicted number of cache misses incurred by each leaf within the 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 6.4 shows the error on predicting cache misses for entire formulas. Tables 6.3 and 6.4 cannot be directly compared, since Table 6.3 shows the number of leaves for which an error is made, while Table 6.4 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 6.4, the learned decision tree is able to predict within 10% of the real number of cache misses on average. This is surprisingly good especially considering that Figure 6.1 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 6.4: Average percentage error for predicting cache misses for entire formulas.

Binary No-2^1-Leaf

Size	Errors
2^{12}	12.7%
2^{13}	8.6%
2^{14}	6.7%
2^{15}	5.2%
2^{16}	4.6%

Binary No-2^1-Leaf Rightmost

Size	Errors
2^{17}	8.2%
2^{18}	8.2%
2^{19}	7.9%
2^{20}	8.1%
2^{21}	10.4%

We are also concerned with whether the fastest formulas are predicted to have the least number of cache misses. To test this, we have plotted the actual runtimes of formulas against the predicted number of cache misses. Figure 6.4 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 runtimes of those formulas.

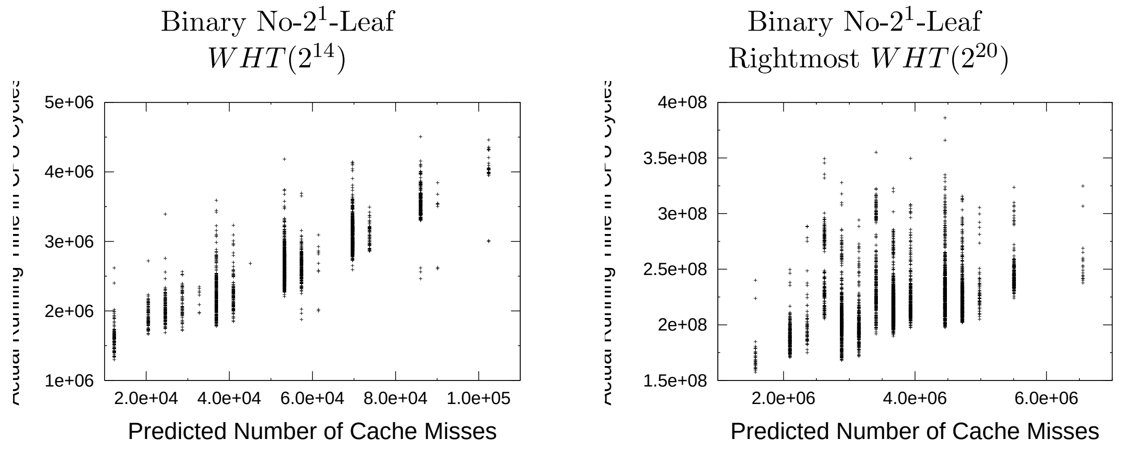


Figure 6.4: Runtime vs. predicted cache misses for entire formulas.

6.2.5 Summary

In summary, we have presented a method for predicting a formula's number of level 1 data 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 a variety of sizes, including larger sizes, even when only trained on one particular size. This learned decision tree serves as a model of the cache performance of formulas.

6.3 Sun Observations

Given the excellent results achieved for predicting cache misses on a Pentium, we wished to see if the same methods would work on other architectures. In particular, we began by collecting data on a Sun UltraSparc IIi 300 MHz machine to see if the same observations described in Section 6.1 for the Pentium would carry over to the Sun.

Figure 6.5 shows a scatter plot of runtimes versus level 1 data cache misses for all binary rightmost $WHT(2^{18})$ split trees with no leaves of size 2^1. Unlike on the Pentium, the fastest formula on the Sun does not also have the minimum number of level 1 data cache misses. This is unfortunate, in that even if we could learn to predict level 1 data cache misses for a Sun, this would not aid in finding the fastest formula.

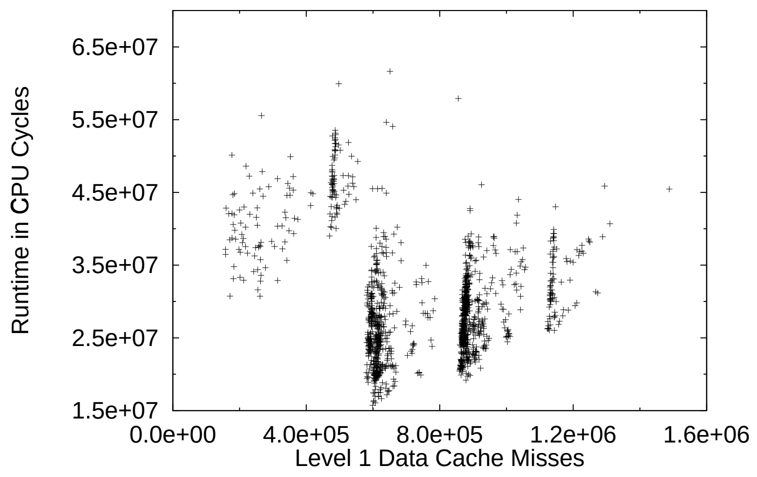


Figure 6.5: Runtimes vs. cache misses for entire $WHT(2^{18})$ formulas on a Sun.

There is a great deal of structure in Figure 6.5. By also collecting the level 2 cache misses for these same formulas, some of this structure becomes understandable. The slower formulas with few level 1 data cache misses have a large number of level 2 cache misses which cause the slower runtimes. Further, all of the formulas with about 6×10^5 level 1 data cache misses and larger have relatively few level 2 cache misses.

This suggests that by minimizing the correct linear combination of level 1 data cache misses and level 2 cache misses, it may be possible to generate a small group of formulas that includes the fastest formula on a Sun. However, this adds an extra level of difficulty in that the correct linear combination must be determined.

Instead of pursuing this direction, we have decided to directly model runtimes of leaves. While learning to model level 1 data cache misses on a Pentium provided a number of advantages in terms of learning, actually learning runtimes for leaves has the advantage of directly modeling the performance measure for which we are most interested. Further, modeling runtimes of leaves does not depend on specific observations of how one performance measure correlates with runtime.

Since runtimes do not come in discrete values as did cache misses, we will have to use a function approximation method instead of a classification algorithm to model runtimes for leaves. By dividing the runtimes by the overall transform size, we will hope to continue to learn across different transform sizes.

6.4 Predicting WHT Leaf Runtimes

We now present our work in modeling runtimes of WHT leaves and demonstrate the performance of the learned models for both a Pentium and a Sun.

6.4.1 Learning Algorithm and Training

Our algorithm for learning to predict runtimes for WHT leaves is shown in Table 6.5.

Table 6.5: Algorithm for learning to predict WHT leaf runtimes.

1. Run a subset of WHT formulas, collecting the runtimes for each of the leaves.
2. Divide each of these runtimes by the size of the overall transform.
3. Describe each of the leaves with the features outlined in Section 6.2.1.
4. Train a function approximation algorithm to predict for leaves the ratio of their runtime to the overall transform size.

This algorithm is very similar to the algorithm presented for learning to predict cache misses for WHT leaves in Section 6.2.2. Again we use the same features to describe WHT leaves. However, instead of learning one of a four categories, this algorithm learns to predict a real valued ratio, namely the runtime divided by the overall transform size.

In the results presented, we have used a regression tree learner, RT4.0 (Torgo, 1999), as the function approximation algorithm. Regression trees are very similar to decision trees except that they can predict real valued outputs instead of categories. However, other function approximators could have been used.

We trained regressions trees from data on a Pentium and also from data on a Sun. Like when learning to predict cache misses, we trained using a random 10% of the leaves of all binary $WHT(2^{16})$ split trees with no leaves of size 2^1. Further, we trained different regression trees on the leaves from 500 random binary $WHT(2^{16})$ split trees with no leaves of size 2^1. These random split trees were generated uniformly over all possible such split trees. While there was some variation in the results from these two different training sets, the results were largely similar. So, only results from the regression trees trained on 500 random split trees is presented.

6.4.2 Evaluation

Again, there are several different methods for evaluating the performance of the learning algorithm at predicting runtimes. We begin by evaluating the performance of the learned regression tree at predicting runtimes for individual leaves. Tables 6.6 and 6.7 show this performance for a Pentium and a Sun. The errors reported are an average percentage error over all leaves in all formulas in the given test set, calculated 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 runtimes for leaf i.

Table 6.6: Error rates for predicting runtimes for leaves for a Pentium.

Binary No-2^1-Leaf

Size	Errors
2^{13}	13.0%
2^{14}	13.8%
2^{15}	15.8%
2^{16}	14.6%

Binary No-2^1-Leaf Rightmost

Size	Errors
2^{17}	11.4%
2^{18}	12.9%
2^{19}	12.6%
2^{20}	12.7%

Table 6.7: Error rates for predicting runtimes for leaves for a Sun.

Binary No-2^1-Leaf

Size	Errors
2^{13}	8.7%
2^{14}	8.7%
2^{15}	10.9%
2^{16}	7.3%

Binary No-2^1-Leaf Rightmost

Size	Errors
2^{17}	16.5%
2^{18}	16.9%
2^{19}	18.9%
2^{20}	20.0%

In all cases, the average error rate for predicting runtimes for leaves was not greater than 20%. This is good considering that formulas can have runtimes that vary by a factor of 2 to 10. Also, this task is considerably more difficult than predicting categories for cache misses since the regression tree is predicting a real valued runtime instead of just one of a few categories.

Next, we evaluate our trained regression trees by determining their accuracy in predicting runtimes for entire formulas. This is done by using the regression trees to predict for each leaf in a split tree and summing these prediction to determine the predicted runtime for the entire formula. Again we calculate an average percentage error, but this time over entire formulas. Tables 6.8 and 6.9 show this performance.

Table 6.8: Error rates for predicting runtimes for entire formulas for a Pentium.

Binary No-2^1-Leaf

Size	Errors
2^{13}	20.1%
2^{14}	22.6%
2^{15}	25.0%
2^{16}	18.1%

Binary No-2^1-Leaf Rightmost

Size	Errors
2^{17}	14.4%
2^{18}	14.1%
2^{19}	12.5%
2^{20}	10.1%

Table 6.9: Error rates for predicting runtimes for entire formulas for a Sun.

Binary No-2^1-Leaf

Size	Errors
2^{13}	23.5%
2^{14}	17.6%
2^{15}	25.8%
2^{16}	36.5%

Binary No-2^1-Leaf Rightmost

Size	Errors
2^{17}	13.3%
2^{18}	15.2%
2^{19}	19.8%
2^{20}	21.2%

Unfortunately, the error rates in some cases are fairly large. Further, the error rates for predicting runtimes for entire formulas on a Pentium are considerably larger than the error rates for predicting cache misses for entire formulas. However, runtimes for formulas take on a whole range of values whereas cache misses for formulas were much more concentrated at specific values, making cache misses easier to predict.

Fortunately, to aid in optimization, a predictor only needs to accurately order formulas according to their runtimes. We do not need to be able to accurately predict the actual runtime of formulas, but just to predict which formula will run faster than another.

To evaluate this, we have plotted the actual runtimes of formulas against their predicted runtimes. Figures 6.6 and 6.7 show these plots for two particular sizes. Each dot in the scatter plots correspond to one formula in the test set with the placement corresponding to the formula's actual and predicted runtimes. The plots show that as the predicted runtimes decrease so do the corresponding actual runtimes. Further, the formulas with the fastest runtimes also have the fastest predicted runtime or nearly the fastest predicted runtime.

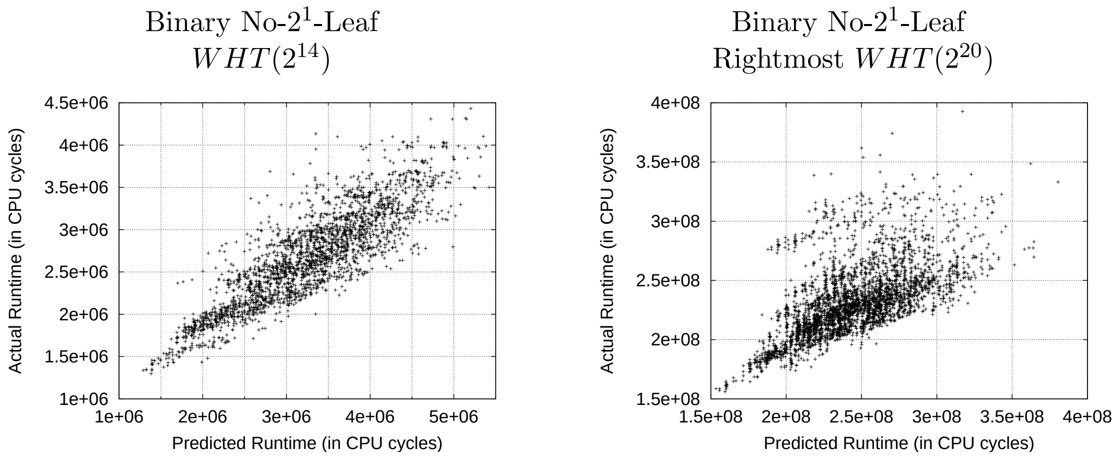
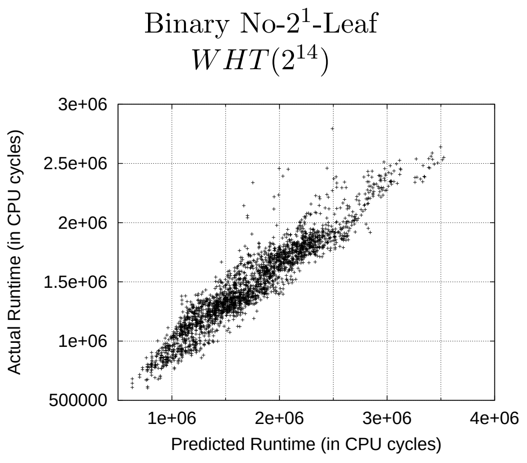


Figure 6.6: Actual vs. predicted runtimes for entire formulas for a Pentium.

Ideally, these plots should be simply a segment along the straight line $y = x$. However, the scatter plots show that there is some spread in that formulas with the same predicted runtimes have different actual runtimes and vice versa. Further, the slope of the plots seems not to be perfectly one. For example, in Figure 6.7, the $WHT(2^{14})$ formulas on the Sun predicted to take about 2.5×10^6 CPU cycles actually take about 2×10^6 CPU cycles. This systematic skew may account for much of the error shown in Tables 6.8 and 6.9.

These plots show that the learned regression trees perform well at ordering formulas according to their runtimes. The formulas with actual faster runtimes are predicted to have faster runtimes. While the error rates at predicting runtimes were larger than we may have liked, these plots show that the learned regression trees could still be used to find fast formulas. This is surprisingly good considering that the regression trees were trained only on data from formulas of one particular size.



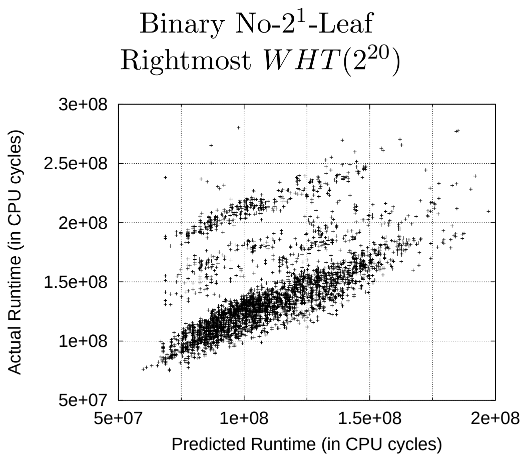


Figure 6.7: Actual vs. predicted runtimes for entire formulas for a Sun.

6.5 Extending to the FFT

The good results of the previous sections for the WHT raise the possibility of extending these techniques to other transforms. We chose to consider the FFT next because it is one of the most important and widely used transforms and also because it is similar to the WHT.

The previous results for the WHT used a package for timing different WHT formulas. The SPIRAL system (see Section 2.5) provides the ability to implement in code, run, and time a wide variety of different transforms including the FFT; thus, it was the obvious choice to replace the WHT package. However, this introduced a number of new issues:

- The SPIRAL system has several different rules for the FFT that break it down in very different ways.
- The SPIRAL system has no predefined leaves, but instead allows more flexibility as the formula can specify the tree completely to the base cases of size 2^1 and can also specify which portions of the tree are to be implemented as unrolled code.
- The SPIRAL system had to be extended to allow it to keep track of the amount of time spent in computing different subportions of a formula. Unfortunately, the accuracy of these timings decrease as the size of the subportions being timed decreases (it is particularly poor for transforms of size 2^1).

To simplify the transition from the WHT to the FFT, we decided to use just one single break down rule for the FFT, namely the Cooley-Tukey factorization (Cooley and Tukey, 1965):

$$DFT(rs) = (DFT(r) \otimes I_s)\, T_s^{rs}\, (I_r \otimes DFT(s))\, L_r^{rs}$$

which is probably the most widely used factorization for the FFT. Further, this factorization is very similar to that used by the WHT in that it factors a DFT of one size into two smaller sized DFTs.

Given that the SPIRAL system has no predefined leaves, ideally we would like to just consider the base cases of size 2^1 as our leaves. However, the fact that accurate timings can not be obtained for such small pieces of code forced us to consider other approaches. Instead, we decided to construct efficient leaves. We conducted an exhaustive search over all possible DFT split trees of sizes 2^2 to 2^7, implementing all of them as entirely unrolled pieces of code. The fastest split tree for each size was then chosen as our new leaf of that size.

With these new leaves, we then added to the SPIRAL system a rule that takes any node of size 2^2 to 2^7 and constructs no children in the split tree. However, when this split tree is then exported to the SPL compiler and one of these leaves are reached, the SPL corresponding to the fastest split tree found for that leaf's size is exported. That is, in the search engine and formula generator, the split trees have leaves of sizes 2^2 to 2^7. However, the SPL compiler actually views a formula that has the fastest split tree substituted for each of the leaves.

Further, the Cooley-Tukey rule was modified so that it never constructed a leaf of size 2^1 so that all split trees must have leaves of sizes 2^2 to 2^7. Nodes of size 2^4 to 2^7 can be made into leaves using our new rule and thus be implemented as unrolled code, or can be further factored using the Cooley-Tukey rule and thus be implemented with loops.

In summary, we have assumed the following space of FFT implementations:

- Only FFT formulas derived from the Cooley-Tukey break down rule are considered.
- We have searched for optimal split trees of sizes 2^2 to 2^7 on the given architecture. These are then available to use as leaves and are implemented as unrolled code.

- All leaves must be of sizes 2^2 to 2^7.
- All internal nodes are implemented in code with loops.

This setup allows us to view modeling performance of FFT formulas in a very similar manner as was done for the WHT. However, there is one significant remaining difference. With the WHT, all of the runtime is spent in computing the leaves while basically no time is spent in computing the internal nodes. However, this is not true for the FFT. Significant computation is performed for each internal FFT node (this arises from the twiddle factors, T_s^{rs}, in the Cooley-Tukey factorization). Thus, the total runtime of an FFT formula is more than just the sum of the runtimes spent computing each of the leaves.

So, we decided to follow a similar approach as before and now model runtime performance of not only leaves but also internal nodes. As indicated earlier, the SPIRAL system can return the time spent in computing different subportions of a split tree. However, the runtime returned for an internal node includes the time spent computing the entire subtree rooted at that node and not just the time associated with the individual node. While we could have trained a model to predict the runtime for the entire subtree under an internal node, this seemed likely to fail as it would mean that the model should be able to predict an entire formula's runtime by simply making a prediction for the root node.

Instead, we calculated the amount of runtime spent in computing a given internal node by taking the runtime for its entire subtree and subtracting the runtimes obtained for its children's subtrees. This then provides a measure of the amount of time spent in computing the work associated just with the internal node.

This method also has an extra benefit of removing some of the error introduced by overhead associated with obtaining the timings. In particular, sometimes the overhead associated with obtaining timings for small sized leaves would cause the total time of two leaves to be greater than the total runtime obtained for the parent of the leaves. Thus, our method would calculate a negative value for the time spent in computing the internal node. While this may seem undesirable, it actually works well in that it cancels out the error introduced by the timing overhead.

6.6 Predicting FFT Runtimes

To predict runtimes for FFTs, we trained up two separate predictors: one for predicting leaf runtimes and one for predicting internal node runtimes. Thus, a runtime prediction for an entire FFT split tree can be made by summing the predictions made for all of its nodes, both leaves and internal nodes, using the appropriate predictor for each node.

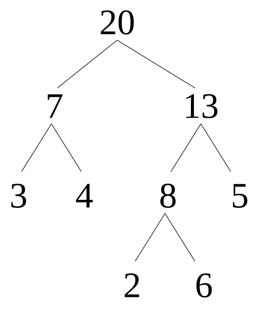
6.6.1 Learning Algorithm, Features, and Training

A runtime predictor for FFT leaves was trained exactly as was done in Section 6.4.1 for WHT leaves using the same leaf feature set. The same method can also be used to train a predictor for FFT internal nodes as well, but the feature set needs to be expanded to describe internal nodes. We have also explored using some additional features for internal nodes that are not applicable for leaves.

Clearly the size and stride of an internal node or an internal node's parent still are easy to obtain. Since the root node has no parent, the corresponding feature values are set to "-1" for the root node's parent. The concept of an internal node's common parent needs to be slightly expanded. We define the previous leaf of a node to be the leaf computed immediately before reaching the given node during computation, and we define a node's common parent to be the first node in common between the parent chains of the node and of the node's previous leaf.

Figure 6.8 gives an example of the features for all the nodes in the given tree. Since all of the nodes have unique sizes in this example, the figure also provides an example of what are the common parents for nodes in a split tree.

We have also explored additional features for internal nodes. Specifically, the sizes and strides of the immediate children of an internal node add four additional features. One might expect that the way an internal node is split may impact on the runtime performance of the work associated with that internal node. Further, the sizes and strides of the four grandchildren could be used as additional features, further defining the internal node's subtree. If any of the children are leaves, then the features corresponding to the missing grandchildren are set to -1 to indicate that the nodes do not exist. We present results using the following three different feature sets:



Node		Parent		Common Parent	
Size	Stride	Size	Stride	Size	Stride
20	0	-1	-1	-1	-1
13	0	20	0	-1	-1
5	0	13	0	-1	-1
8	5	13	0	13	0
6	5	8	5	13	0
2	11	8	5	8	5
7	13	20	0	20	0
4	13	7	13	20	0
3	17	7	13	7	13

Figure 6.8: Example node features for all of the nodes in the given WHT split tree.

- **Original.** The original six features as described earlier.
- **Children.** The original six features plus features describing the immediate children of the internal node.
- **Grandchildren.** The original six features plus features describing the immediate children and grandchildren of the internal node.

Again we used RT4.0 to train regression trees on data collected on the same Pentium as used in the previous experiments in this chapter. For the leaf predictor, we used a random 10% of the leaves from all of the formulas generated for $DFT(2^{16})$. Likewise we trained regression trees using each of the different feature sets on a random 10% of the internal nodes from all of the formulas generated for $DFT(2^{16})$.

6.6.2 Evaluation

We used two different methods to evaluate these predictors for the FFT. First, we considered the average error rate for predicting runtimes for entire formulas. Second, we plotted the predicted runtime against the actual runtime for individual formulas. To do the evaluations, we generated all possible formulas for sizes 2^{12} to 2^{18} and timed them. We were not confident that rightmost trees were optimal, and so without that additional limitation it was impossible to exhaust up to size 2^{20}. Results are shown using all three different feature sets to describe internal nodes, but in all cases the Original feature set was used to describe the leaves (as leaves clearly do not have children or grandchildren).

Table 6.10 shows the average error rates for predicting FFT formula runtimes across the three different feature sets introduced in the previous subsection. The error rates are quite good, outperforming those for the WHT in Table 6.8. For many of the larger sizes, the error rates are less than 10%, and for no size are they greater than 20%. Further, increasing the number of features used through the Children and Grandchildren feature sets tends to improve the overall performance of the predictor while slightly decreasing the performance for a few of the smaller sizes.

Table 6.10: Error rates for predicting FFT formula runtimes on a Pentium using different feature sets.

Original		Children		Grandchildren	
Size	Errors	Size	Errors	Size	Errors
2^{12}	9.8%	2^{12}	11.8%	2^{12}	19.3%
2^{13}	15.5%	2^{13}	15.2%	2^{13}	9.3%
2^{14}	11.8%	2^{14}	9.6%	2^{14}	10.7%
2^{15}	8.7%	2^{15}	7.6%	2^{15}	7.3%
2^{16}	6.4%	2^{16}	5.6%	2^{16}	5.0%
2^{17}	8.2%	2^{17}	7.8%	2^{17}	7.3%
2^{18}	8.9%	2^{18}	8.8%	2^{18}	7.9%

Figures 6.9 and 6.10 are scatter plots of the actual versus predicted runtimes for FFT formulas on the same Pentium machine. The plots show predictions for formulas of several different sizes and using the three different features sets. Each point in the plots corresponds to one formula with the point placed according to the formula's actual and predicted runtimes. Ideally all the points would fall along the line $y = x$ indicating that the predicted runtime equaled the actual runtime.

While there is some spread, most of the points fall very close to the ideal line $y = x$. Further, the formulas with the fastest predicted runtimes are those formulas with the fastest actual runtimes. Visually, the major difference between the different features sets is that increasing the number of features seems to improve the overall slope so that the predicted runtimes of the fastest formulas are more accurate.

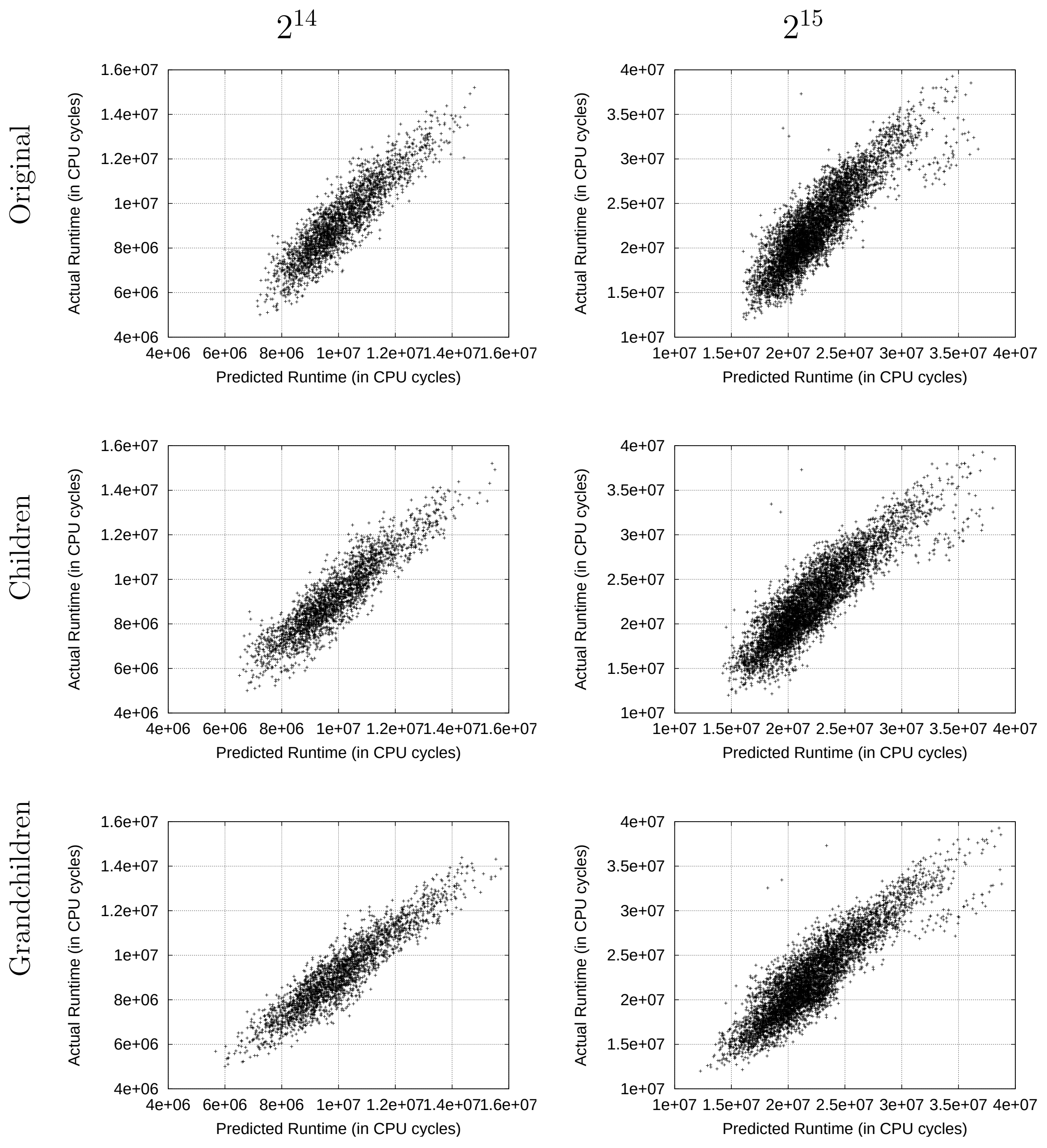


Figure 6.9: Predicted FFT runtimes versus actual runtimes for sizes 2^{14} and 2^{15}.

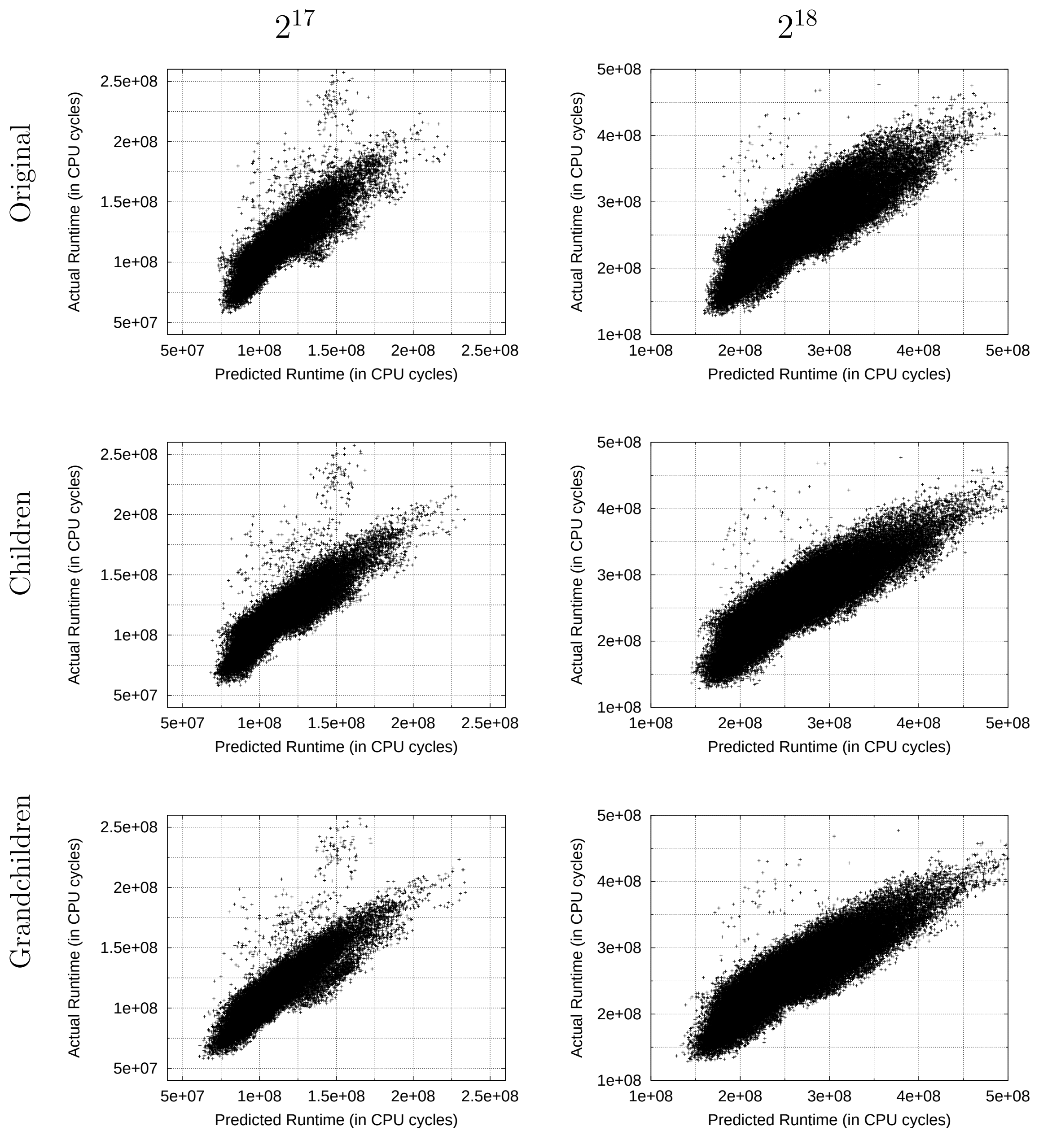


Figure 6.10: Predicted FFT runtimes versus actual runtimes for sizes 2^{17} and 2^{18}.

The learned regression trees perform well at ordering formulas according to their runtimes, predicting faster formulas to be faster. Once again, these are particularly excellent results considering that the learned regression trees were only trained on data from transforms of size 2^{16} and here are predicting for both smaller and larger sizes.

6.7 Summary

We have presented a method for predicting a WHT formula's number of cache misses on a Pentium 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.

We have presented a similar method for predicting runtimes instead of cache misses for WHT formulas. This method can be used on any machine and we demonstrated its performance on two very different architectures. The learned regression trees are able to perform very well at ordering formulas according to their runtimes, especially considering that the regression trees were trained on data of one size and used to predict across many transform sizes including larger sizes.

Further, we have extended this method to predict runtimes for FFT formulas. This required training two predictors, one for leaves and one for internal nodes. Using data for a Pentium, we obtained very excellent results, with the regression trees accurately predicting runtimes for FFT formulas. While the predictors were only trained on data from one transform size, they accurately predicted for both smaller and larger transform sizes.

Chapter 7

Generating Optimal Implementations

Accurate prediction of runtimes (Chapters 5 and 6) still does not solve the problem of determining the fastest formula from a very large number of candidates. At larger sizes, it is infeasible to just enumerate all possible formulas, let alone obtain predicted runtimes for all of them in order to choose the fastest. While our work in predicting performance for formulas allowed for runtimes to be predicted much more quickly than the formulas could be timed, it still did not provide a method that quickly produced formulas with fast runtimes.

In this chapter, we present a method that learns to *generate formulas* with fast runtimes. 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 runtimes. 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.

The learned decision and regression trees of Chapter 6 that predict performance of WHT and FFT nodes are used by this work as the sole source of runtime information for a given platform. By using a number of concepts from reinforcement learning combined with information from these predictors, our method is able to generate formulas that have the fastest known runtimes.

We begin by describing our initial approach to the problem and how it led to the algorithm we have designed and implemented. After discussing this algorithm, we then provide other views of our method. Finally, we evaluate our method's performance for two different machines and two different transforms.

7.1 Approach

We approach the question of generating fast formulas as a control learning problem. Given a transform and size, we want our algorithm to grow a split tree for that transform that runs as fast as possible. Figure 7.1 illustrates this process for the WHT. We begin in (a) with a root node of the desired size. Next, we grow in (b) the best possible children. Here there are choices for which children are to be grown and our method needs to learn to make the choice that will produce the fastest possible split tree. Then, we recurse on each of the children, which is started in (c).

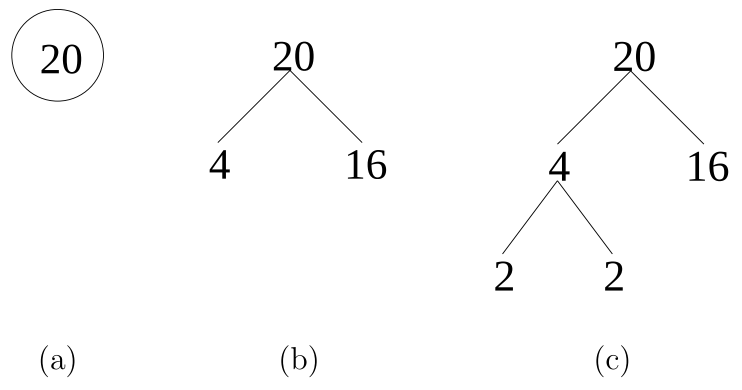


Figure 7.1: Example of growing a fast WHT split tree.

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.

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

We now turn to trying to formulate this problem in terms of MDPs. Let the states in the reinforcement learning problem be nodes in a split tree that have no children but have not been decided to be leaves. Then the start state is just a root node of the given transform and 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:

- the leaf's runtime (or cache misses) when making a node a leaf, and
- the internal node's runtime when giving children to a node.

Note that for the WHT, there is no runtime associated with internal nodes, and thus the cost function would be zero when giving children to a node. This causes the total cost of growing an entire split tree to simply be the runtime of the entire formula. The goal is then 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.

7.1.2 Details and Difficulties

There is a number of details that need to be filled in with respect to this basic formulation. Further, one significant difficulty also becomes apparent upon trying to fill in these details.

State Representation

We need a state representation for the nodes within a split tree so that we can use standard reinforcement learning algorithms. For the WHT, we have used a modified form of the leaf features described in Section 6.2.1 that expands the feature set to describe any node in a split tree. Specifically, we have used 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. Note that the root node has no parent, and thus the corresponding feature values are set to "-1" for the root node.

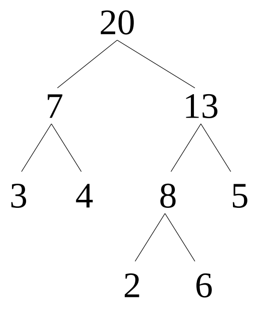
The concept of a node's common parent is more difficult to understand in this setting. We define the previous leaf of a node to be the leaf computed immediately before reaching the given node during computation. However, the previous leaf of a given node is not always known when constructing a fast split tree. 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. Consider the following cases:

- The root node has no common parent, as there is no previous leaf computed before reaching the root node. Thus, the common parent feature values are set to "-1" for the root node.
- If the given node is *not* the rightmost child of its parent, then some descendant of the sibling to the right of the given node will be the given node's previous leaf. Thus, it must be the case that the parent of the given node is also its common parent.
- If the given node *is* the rightmost child of its parent, then the previous leaf for this given node is also the previous leaf for its parent since the rightmost child is called first from a parent. So, in this case, the node's common parent is its parent's common parent. Note that if its parent's common parent is not defined (for example, its parent is the root), then the node's common parent is likewise undefined and the corresponding feature values are set to "-1".

Figure 7.2 gives an example of the features for all the nodes in the given tree. Since all of the nodes have unique sizes in this example, the figure also provides an example of what are the common parents for nodes in a split tree.

For the FFT, we have used the same features as above, but also the two additional feature sets used to describe internal nodes that were introduced in Section 6.6.1.



Node		Parent		Common Parent	
Size	Stride	Size	Stride	Size	Stride
20	0	-1	-1	-1	-1
13	0	20	0	-1	-1
5	0	13	0	-1	-1
8	5	13	0	13	0
6	5	8	5	13	0
2	11	8	5	8	5
7	13	20	0	20	0
4	13	7	13	20	0
3	17	7	13	7	13

Figure 7.2: Example node features for all of the nodes in the given WHT split tree.

These feature sets expand states to also account for the sizes and strides of the given node's immediate children and grandchildren.

Cost Function

Ideally, we would use the actual runtime of leaves and internal nodes to determine the cost function. However, this requires determining the runtime for these nodes even when the split tree is not fully grown. Unfortunately, it is not possible to run a partially grown split tree with the SPIRAL system or with the WHT package we are using.

However, Chapter 6 discussed how we can learn to predict cache misses or runtimes for leaves and internal nodes. So, we can approximate our desired cost function by using these learned predictors. In particular, we will use the following cost function:

- the leaf's predicted performance when making a node a leaf,
- zero when giving children to a WHT node, and
- the internal node's predicted performance when giving children to a FFT node.

The predictors have several advantages:

- They can make predictions much quicker than we could time a formula (even if we had a formula to time in this case).
- Since they are using the same features to describe nodes as we are using to describe nodes in our MDP state space, no extra work needs to be done to translate features.

This change causes our method to rely on the predictors and to only be as good as they are. Further, if a cache miss predictor is used, then our method will construct formulas with minimal cache misses, instead of explicitly the fastest formulas. However, on a Pentium we have shown that the fastest WHT formulas have the minimal number of cache misses.

Also the choice of predictor can influence how many formulas are generated with the same expected performance. Since there are many formulas with the same number of cache misses and our cache miss predictor produces discrete valued outputs, it would be expected that using a cache miss predictor will cause our algorithm to generate several formulas with the same predicted performance. On the other hand, runtimes tend to be much more continuous, and so it is likely using a runtime predictor will lead to many fewer formulas with the exact same predicted performance.

Transition Function

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 a learned predictor. We will define a value function over our states and show how it can be computed off-line.

7.1.3 Value Function

First consider a value function over WHT nodes in a fully specified split tree. The value of a leaf node is simply its predicted performance, that is, the cost of making that node a leaf. The value of an internal node is then the sum of the performance measures of all the leaves in the subtree rooted at the internal node. More formally,

$$V(node) = \sum_{leaf \in subtree} PredictedPerformance(leaf).$$

Thus, the value of the root node is the performance of the entire split tree. We can rewrite this value function recursively in terms of the values of the children of the given node. The value of a node is the sum of the values of its children if it is an internal node or the predicted performance if it is a leaf. So,

$$V(node) = \begin{cases} PredictedPerformance(node), \text{ if node is a leaf} \\ \sum_{children} V(child), \text{ otherwise} \end{cases}.$$

The value function over FFT nodes is similar but must take into account that work must be performed at the internal nodes and not just the leaves. So, we have

$$\begin{aligned} V(node) &= \sum_{leaf \in subtree} PredictedPerformance(leaf) \\ &+ \sum_{internalNode \in subtree} PredictedPerformance(internalNode) \end{aligned}$$

and then recursively

$$V(node) = \begin{cases} PredictedPerformance(node), \text{ if node is a leaf} \\ PredictedPerformance(node) + \sum_{children} V(child), \text{ otherwise} \end{cases}.$$

Thus, the value of an internal node is the predicted work associated with that node plus the values of its children.

Now we will define the *optimal* value function over the specified state space for the WHT. If a state must be a leaf, then its value is its predicted performance. 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 minimizes the predicted performance summed over all the leaves. That is,

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

Again, we can rewrite this value function recursively in terms of the values of the possible 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 predicted performance if the optimal subtree for the state is to be a leaf. Mathematically, let the leaf performance of a state be:

$$LeafPerf(state) \;=\; \begin{cases} PredictedPerformance(state), & \text{if state can be a leaf} \\ \infty, & \text{if state cannot be a leaf} \end{cases}$$

and the splitting value of a node be:

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

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

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

For the FFT, we again must account for the runtime cost associated with internal nodes. So, we have:

$$\begin{aligned} V^*(state) \;=\; & \min_{subtrees} \sum_{leaf \in subtree} PredictedPerformance(leaf) \\ & + \sum_{internalNode \in subtree} PredictedPerformance(internalNode). \end{aligned}$$

When rewriting this recursively, *LeafPerf* remains the same as for the WHT. Assuming the Original feature set is used, we have:

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

and then

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

If the Children or Grandchildren feature sets are used, then the state space already captures the immediate children. In these cases, *SplitV*'s minimum must be taken over all possible grandchildren or great-grandchildren. While the child is determined from the state of the given node, the child's full state feature values must account for the chosen grandchildren or great-grandchildren.

7.2 Algorithm

This recursive 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 predictor to get its predicted performance. 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. The algorithm is shown in Table 7.1. For the WHT, let the `PredictedPerformance` of internal nodes be zero.

Table 7.1: Algorithm for computing values of states.

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

Note that for the FFT feature sets Children and Grandchildren, this algorithm must be modified. Since the state space also describes a state's immediate children, no loop over possible sets of children is needed. However, to determine a child's state description, it is necessary to consider the possible children or grandchildren of this child. Thus, the algorithm remains very similar but loops over possible sets of grandchildren or great-grandchildren of the given state instead of possible sets of immediate children.

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 best ways to split the root node. This procedure is then repeated for each child, building up a set of fast split trees.

The algorithm for generating fast split trees is shown in Table 7.2. For simplicity, the algorithm is shown only for binary 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. As before, the loop over possible children would need to be modified to loop over possible grandchildren or great-grandchildren if the FFT feature sets Children or Grandchildren are used.

Table 7.2: Algorithm for generating fast split trees.

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

Note that ComputeValues cannot easily keep track of the best split tree as there may be several formulas with the best predicted performance. This is particularly true when using a cache miss predictor since many formulas have the exact same 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, we want the ability to not only produce the formula or formulas that all have the best predicted performance, but also a larger set of formulas that all have close to the optimal predicted performance. In our code, the FastTree algorithm has been extended to allow for a tolerance, producing split trees that have up to the tolerance more runtime or cache misses than what is predicted to be optimal.

7.3 Other Views

Section 7.1 presented our original approach to the problem. This section expounds on how our final algorithm can be viewed.

The algorithm we have presented is similar to solving an MDP. However, we did not give a well defined transition function for this problem, and thus did not actually frame it as an MDP. We do have a model of the system in that we know what actions do and we know the cost function since it is represented by the learned predictor. This allows us to compute the value function off-line.

Since a node's value only depends on the possible subtrees that could be grown underneath it, dynamic programming is an efficient method for computing the value function. While both this method and the dynamic programming search method discussed in Chapter 3 use a variation of the basic concept of dynamic programming, there are significant differences between the two methods:

- The dynamic programming algorithms are performed over different feature spaces. ComputeValues uses at least six features to describe a state in the dynamic programming. The earlier dynamic programming search method only used the transform and its size. That is, ComputeValues distinguishes between nodes of the same size but in different locations in the split tree, whereas the dynamic programming search method does not. Thus, ComputeValues uses a much richer representation and makes a much weaker assumption in using dynamic programming.

- Values in the dynamic programming algorithms are obtained in different ways. ComputeValues uses a learned predictor to determine the values of nodes, while the dynamic programming search method actually times split trees to determine values. Thus, the method presented in this chapter pays a one time cost to run some formulas to gather data to train a performance predictor. Once this is done, the learned predictor can be used for many different transform sizes and to make predictions much more quickly than actually timing formulas.

- ComputeValues only uses the learned predictor for getting values for leaves and the runtime associated with internal nodes. The values of internal nodes are computed by summing the values of the children. The dynamic programming search method actually times all subtrees that it considers.

While ComputeValues considers all possible sets of children of a given state, it is considerably more efficient than exhaustively constructing all possible split trees and making performance predictions for them. By computing and memoizing *values of states*, ComputeValues saves significant computation over an exhaustive search approach. For example, suppose we split $WHT(2^{20})$ into $WHT(2^8)$ and $WHT(2^{12})$. Then if we were performing exhaustive search, we would have to recompute the value for all possible subtrees of $WHT(2^8)$ every time we choose a different possible subtree for $WHT(2^{12})$ (and there are many possible trees for both). Since ComputeValues memoizes its results, it only has to compute a value once for $WHT(2^8)$ as the left child of the root and once for $WHT(2^{12})$ as the right child of the root.

Thus, this problem is well suited for using dynamic programming. The problem has overlapping subproblems in that the value for a particular state is repeatedly needed as illustrated in the above example. Further, by using a powerful state space representation that captured information about the location of a node in a split tree, we could assume optimal substructure — that is, optimal solutions to the problem contain optimal solutions to subproblems. For example, the best subtree for the $WHT(2^8)$ node in the previous example is largely independent of the subtree grown for the $WHT(2^{12})$ node, but it is dependent on the fact that the $WHT(2^8)$ node was the left child of the $WHT(2^{20})$ root node. The features chosen capture this later information but not the former, allowing optimal substructure to be assumed.

7.4 Evaluation

Evaluating our method is difficult since the optimal formula for larger sizes is unknown. However, for the WHT, we can compare our algorithm against the best formulas found by searches over limited portions of the space as we did to evaluate the predictors in Chapter 6. That is, for sizes 2^{16} and smaller we exhaustively time all binary WHT formulas with no leaves of size 2^1, and for sizes 2^{17} and larger we exhaustively time all rightmost binary WHT formulas with no leaves of size 2^1. These subspaces of WHT formulas contain the fastest formulas found from performing a variety of different searches. For the FFT, we exhaustively time the defined space and compare our results, but this is only possible up to about size 2^{18} (which took a considerable amount of time to collect the data).

We have used the decision tree learned in Section 6.2 that predicted cache misses for WHT leaves and the regression trees learned in Section 6.4 that predicted runtimes for WHT leaves. Further, we used the regression trees learned in Section 6.6 that predicted runtimes for FFT nodes. Since these decision and regression trees were 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 or regression tree on a broader class of formulas.

7.4.1 Using the WHT Cache Miss Predictor

We begin with the results from using the learned decision tree that predicted cache misses for WHT leaves on a Pentium. Since many split 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 7.3 displays three different results for different sizes.

The first column gives the transform size. The second 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.

The third 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 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 fastest 16 formulas found by a limited

Table 7.3: Results from generating fast WHT formulas using Pentium data and a cache miss predictor.

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

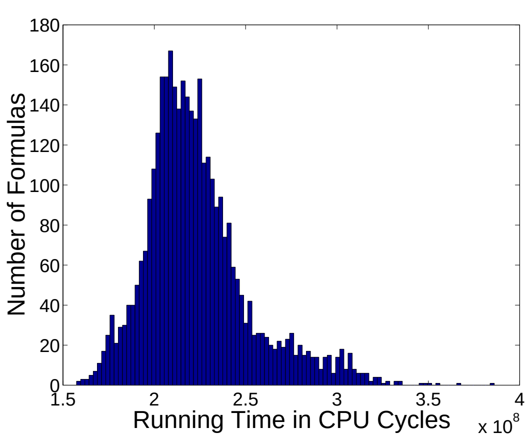
exhaustive search, but did not generate the 17th fastest formula. For all sizes, our method generates the fastest formula as well as many of the formulas that are very close to the fastest.

Figure 7.3 compares the histograms of runtimes 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 runtimes amongst the fastest found by the more exhaustive method.

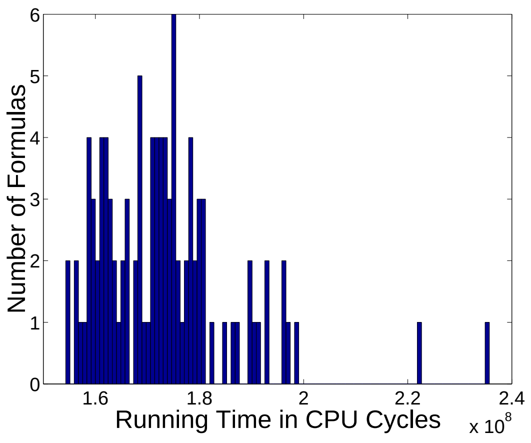
7.4.2 Using WHT Runtime Predictors

Next, we evaluate our method while it is using the learned regression trees that predicted runtimes for WHT leaves. Here we evaluate the approach both on Pentium and Sun data. We have used the same learned regression trees as were described and evaluated in Section 6.4 and that were trained using data from 500 random $WHT(2^{16})$ split trees with no leaves of size 2^1.

Because the learned regression trees predict real-valued runtimes instead of only one of a few categories, most formulas have different predicted runtimes. So, our generation method normally would only produce a single formula which it believes to



(a)



(b)

Figure 7.3: Histograms of runtimes 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 $WHT(2^{20})$ using a cache miss predictor on Pentium data.

run as fast as possible. To evaluate, we have given our generation method a tolerance and had it generate all formulas that it believes will have up to that tolerance more runtime than the predicted fastest one. We began with a small tolerance and increased it as necessary to generate as many formulas as necessary to perform the evaluations described below.

Tables 7.4 and 7.5 show the results of using our method to generate fast WHT formulas for a Pentium and for a Sun. The first column shows the size of the WHT transform being generated. The second column shows when during the generation process the best known formula (through a limited exhaustive search) is generated. For example, on the Pentium (Table 7.4) the fourth $WHT(2^{20})$ formula generated by our method is the fastest known formula, while the first three formulas generated by our method were actually slower than this fourth one.

The third column shows how much slower the first generated formula is compared to the fastest known formula. The fourth column shows how many of top 100 best known formulas are generated if we allow our method to only generate 100 formulas. Finally the fifth column shows the first best known formula that was not in the top 100 formulas generated.

These results are quite excellent, especially considering the fact that our method only uses training data from one size (2^{16} in this case). Our method always generated

Table 7.4: Evaluation of generation method using a WHT runtime predictor for a Pentium.

Size	Generated formula X is best known formula	First generated formula is X% slower than best known formula	Number of top 100 best known formulas in top 100 generated formulas	First best known formula not in top 100 generated formulas
2^{13}	5	3.4%	69	19
2^{14}	4	3.0%	63	19
2^{15}	3	2.1%	68	16
2^{16}	4	1.7%	63	18
2^{17}	5	0.1%	54	36
2^{18}	4	2.0%	60	24
2^{19}	1	0.0%	44	36
2^{20}	4	1.7%	64	24

the fastest known formula within the first five formulas it generated for the Pentium and within the first 50 formulas it generated for the Sun. This is excellent considering the literally thousands or tens of thousands of formulas that are possible. Further, except for a few sizes on the Sun, the very first formula our method generated had a runtime within 6% or less of the fastest known formula. Again this is great considering that the formulas of the same size can have a factor of 2 to 10 spread in runtimes.

Also, our method was able to generate about 50 of the top 100 fastest known formulas even when our method was limited to generating only 100 formulas. For the Pentium, our method was also able generate all of the top 15 formulas for each size (often many more) while being limited to generating only 100 formulas. For the Sun, our method was sometimes unable to generate the fourth fastest known formula for a few particular sizes when being limited to generating only 100 formulas.

Overall, our method more easily generates fast formulas for the Pentium than for the Sun, but still for the Sun our method is able to generate the fastest known formula in the first 50 or less formulas that it produces. Thus, by timing a few random formulas of one particular size to be used as training data, our method can be used to generate very fast formulas for different transform sizes including larger sizes. Often the very first formula it produces has a runtime very close to the fastest known formula. Further, if one is willing to time a few additional formulas for each

Table 7.5: Evaluation of generation method using a WHT runtime predictor for a Sun.

Size	Generated formula X is best known formula	First generated formula is X% slower than best known formula	Number of top 100 best known formulas in top 100 generated formulas	First best known formula not in top 100 generated formulas
2^{13}	14	77.7%	20	6
2^{14}	20	12.8%	70	24
2^{15}	1	0.0%	68	38
2^{16}	2	4.3%	70	20
2^{17}	7	18.0%	47	10
2^{18}	38	5.9%	46	7
2^{19}	17	3.3%	46	4
2^{20}	47	1.4%	52	4

size, the fastest known formula can be found. While we have evaluated our method on a Pentium and a Sun, our method should work across any architecture. Our method does not depend on any observations specific to those machines.

7.4.3 Using FFT Runtime Predictors

Finally, we used the learned regression trees that predicted runtimes for FFT leaves and internal nodes to generate fast FFT implementations. We use a similar evaluation as used with the WHT runtime predictors, but this time evaluate across the three different feature sets. We evaluate against an exhaustive search of all possible formulas from the space we considered (see Section 6.5) for sizes 2^{12} to 2^{18}. Recall that the trained regression trees were trained only on data for FFTs of size 2^{16}.

Table 7.6 displays the results of using our generation method to construct fast FFT implementations. With any of the feature sets, our method was able to construct the fastest known FFT formula of sizes 2^{14} to 2^{16} within the first 10 formulas that it generated. However, the Grandchildren feature set tends to outperform the other feature sets. Using the Grandchildren feature set, our method was able to generate the fastest known formula as its first produced formula for three of the sizes. Further, for the other sizes, either the first generated formula was nearly as fast as the best known or the best known formula was generated within the first several formulas generated.

Table 7.6: Evaluation of generation method using FFT runtime predictors for a Pentium. "Place" = generated formula X is best known formula. "Slower" = first generated formula is $X\%$ slower than best known formula.

	Original		Children		Grandchildren	
Size	Place	Slower	Place	Slower	Place	Slower
2^{12}	58	37.5%	203	21.1%	16	14.3%
2^{13}	6	40.0%	86	49.8%	1	0.0%
2^{14}	3	12.4%	8	13.6%	2	13.6%
2^{15}	7	24.9%	6	20.6%	1	0.0%
2^{16}	9	24.5%	2	16.3%	1	0.0%
2^{17}	217	77.4%	533	18.7%	82	3.6%
2^{18}	165	2.4%	27	18.4%	11	6.5%

Again these results are quite excellent considering the huge search space of formulas and the wide spread of runtimes. Further, our method has only seen timings for formulas of size 2^{16} and yet can construct fast formulas for both smaller and larger sizes.

Figure 7.4 compares our generation method against the search engine's dynamic programming search method for sizes 2^{12} to 2^{20}. The search engine's dynamic programming used the default options (and thus was a 1-best DP) and searched over the same space of possible formulas as our generation method. In this plot, we used the Grandchildren feature set with our generation method. We plot separate lines for the fastest timing out of the first one, twenty, and hundred formulas our method generated. Exhaustive search is also displayed for sizes up to 2^{18}.

From the figure, it can be seen that our generation method is able to find faster formulas than the dynamic programming method in the search engine for sizes up through 2^{19}, even when our method is limited to only generating 20 formulas. While performance of our generation method tends to degrade at size 2^{20}, it is still able to find a formula within the first 100 generated that had a runtime within 5% as fast as that found by dynamic programming. Further, this is still surprisingly good that while only being trained on data from size 2^{16} our method is able to still perform well at size 2^{20}.

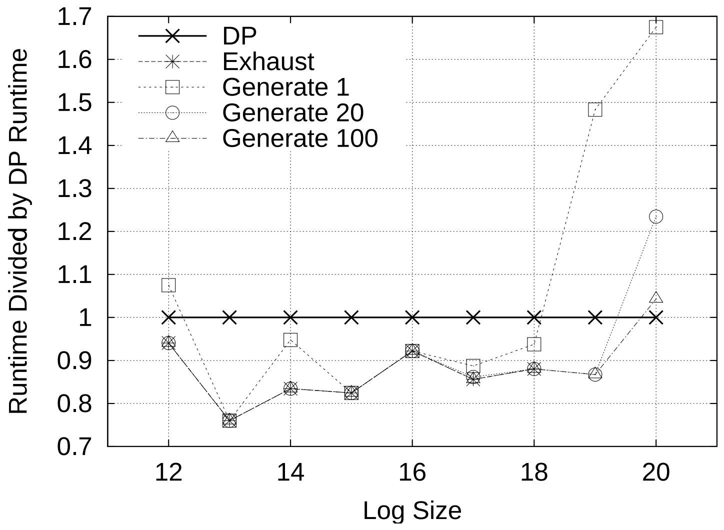


Figure 7.4: Comparison of our generation method against dynamic programming and exhaustive search.

7.5 Summary

We have introduced a method for learning to generate fast implementations of signal processing algorithms. We demonstrated its effectiveness along several dimensions:

- It can be applied to different transforms (FFT and WHT).
- It can be applied across different machines (Pentium and Sun).
- It can be used with different performance measures (runtime and cache misses).

The results that we obtained were excellent, constructing the fastest known formula within the first few generated formulas and often the first formula constructed was very fast. Further, this method can generate fast formulas across many sizes including larger sizes while only being trained on data from one particular size.

This method provides an alternative to searching as we described in Chapter 3. Our method constructs fast formulas of a given transform size without even timing a single formula of that particular size. Often the very first formula generated had a runtime very close to the fastest known formula, but a small search over the top few formulas generated could be performed to determine the fastest one.

Chapter 8

Related Work

This discussion of related work begins in Section 8.1 with some other related work within the SPIRAL group. Section 8.2 focuses on other signal transform optimization work. Then, Section 8.3 describes some related work in the similar area of linear algebra. Broadening the scope, Section 8.4 discusses some related work in optimization and modeling of more general numerical algorithms. Optimization has been of great importance in compiler design, and Section 8.5 discusses some of the work in automatically tuning the optimizations that a compiler applies. Next, Section 8.6 discusses work in using machine learning to guide search in combinatorial problems. Finally, Section 8.7 discuss more general work in applying artificial intelligence to signal processing.

8.1 Other SPIRAL Related Work

There has been a considerable amount of related work produced by others within the SPIRAL research group. Much of this related work has already been discussed in this thesis, much of it in Section 2.5 on the SPIRAL system. However, there is some other related work in the SPIRAL research group that has not already been described.

Kumhom et al. (2000; 2001) describe a method for designing, optimizing and implementing dimensionless FFTs (Auslander et al., 1997) on FPGAs. They follow a process very similar to the SPIRAL system, but adapted for designing FPGA hardware instead of implementations on standard desktop uniprocessors. They outline different possible implementations in a mathematical framework and provide a method for mapping these onto the FPGA hardware. Further, they conduct an exhaustive search over a particular limited subspace of possible implementations. Since obtaining actual runtimes would require reconfiguring the FPGA for each implementation and doing so would be very time consuming, they use a model to determine the performance of different implementations.

Park et al. (2000; 2001) introduce dynamic data layout as a method for improving cache performance by reorganizing data layout in the middle of computations. While performing a data reorganization during computation can be expensive, it can improve cache performance so significantly that the overall runtime is improved. Determining when to apply these reorganizations introduces a new degree of freedom and thus another potential area to be searched. Results were shown for both the FFT and the WHT.

8.2 Signal Transform Optimization

Most of the early work in signal transform optimization was concerned with minimizing the number of arithmetic operations necessary to compute a transform. However, others have also considered optimizations that yield more efficient code on real computer platforms while not changing the number of arithmetic operations.

8.2.1 Minimizing Arithmetic Operations

There has been a large amount of work in designing efficient algorithms for signal processing transforms that minimize the number of arithmetic operations; for example, (Heideman, 1988; Burrus, 1997; Tolimieri et al., 1997; Nussbaumer, 1982; Rao and Yip, 1990). Most of these efficient algorithms factor a signal transform in a specific way that leads to efficient computation in terms of the number of arithmetic operations that must be performed. These factorization methods can be represented by break down rules that describe how a signal transform can be computed by combining smaller or different signal transforms. This thesis builds off this work in that we use these break down rules to generate formulas representing a given transform.

One interesting variant to these approaches has been to use binary decision diagrams and related structures to efficiently implement the WHT and other transforms (Clarke et al., 1997; Zhao, 1996).

Most of the prior signal processing optimization work has concentrated on simply minimizing the number of arithmetic operations without consideration of how fast those operations could be performed on a real machine. Most of the algorithms efficient in arithmetic operations really represent a whole class of algorithms that can be represented by different formulas. There is a large number of different formulas with the exact same number of operations that represent the same transform. Because of the complexity of modern processors, these formulas can have very different runtimes despite having the same number of operations. This thesis is concerned with optimizing and modeling performance on real machines — not just minimizing the number of operations but also adapting to the computer platform.

8.2.2 Optimizing Signal Transforms for Real Computers

There are many efficient implementations of the FFT that are available today, some of which are listed at (Frigo and Johnson, 1998b). Some of these implementations are hand tuned for particular platforms while others simply try to provide efficient C or FORTRAN code. However, most of these implementations do not automatically adapt their code to the particular platform that it is being run on.

Probably the most similar work to ours is that of Frigo and Johnson on FFTW (Frigo and Johnson, 1998a; Frigo, 1999). FFTW is an adaptive package that produces efficient Fast Fourier Transform (FFT) implementations across a variety of machines. FFTW consists of two major components. The first component is a “codelet” generator that produces optimized sequences of unrolled code (called codelets) for specific small sized FFTs. The codelet generator deterministically uses a set of well known algorithms to generate the specified FFT. This is then simplified as much as possible to reduce the number of operations, and a few transformations are performed that lead to more efficient code across platforms. The second major component is the planner and executor that determines and executes a strategy for combining the generated codelets to compute the desired FFT. The codelets are combined using only the Cooley-Tukey FFT algorithm (Cooley and Tukey, 1965). A constrained dynamic programming search is used to find a fast way to factor the desired FFT according to the Cooley-Tukey. Dynamic programming assumes that the optimal implementation of a particular sized FFT is still optimal if used as a subpart of a larger FFT. Adaptation to the particular machine being used is performed at this stage as actual implementations are timed during the dynamic programming search.

There is a number of similarities and differences between FFTW and our work:

- The WHT package described in Section 2.3 that we have used in some of the experiments presented in this thesis is similar to FFTW in its use of small, optimized, unrolled pieces of code. However, the more general SPIRAL system does not use this notion, and allows the space of possible small, unrolled formulas to be searched as well. The codelets generated by FFTW are the same across all platforms and are not adapted to the particular platform as the general SPIRAL framework allows. When we did construct leaves for the FFT in the SPIRAL system in Section 6.5, we used the SPIRAL system to search for optimized leaves tuned for the particular architecture being used.
- Both FFTW and our work have used search to find fast implementations adapted to the particular machine being used. FFTW uses only dynamic programming to search for optimal implementations. We have not only used a variety of dynamic programming variations, but have also developed other search methods.
- FFTW only uses the Cooley-Tukey algorithm for combining its codelets. This is similar to the WHT package which only uses one specific rule for decomposing WHTs into smaller WHTs. However, the general SPIRAL framework has a number of different break down rules that may be applied when searching for an optimal FFT.
- FFTW constrains dynamic programming to only search in the space of rightmost split trees, while our work has not been limited to this much smaller set of split trees.
- As its name implies, FFTW explicitly considers just FFTs. While many other transforms can be constructed with FFTs and thus computed with FFTW (Vuduc and Dremmel, 2000), the general SPIRAL system has the advantage of explicitly describing different transforms (even user specified ones) and different break down rules specific to those transforms.

Other researchers have extended FFTW in different ways. Vuduc and Dremmel (2000) extended FFTW to also be able to compute DCT Type II.

Gatlin and Carter (2000) extended FFTW to allow it to unroll and interleave loops implementing the butterfly operation. Finding the best amount of unrolling and interleaving for each node in the split tree introduces a new degree of freedom in the algorithm. A search over a set of reasonable levels of interleaving is made at each node while holding the level of interleaving at all other nodes fixed to some predefined default. Then, the best level of interleaving found for each node individually is used in one split tree as the best possible. Thus, Gatlin and Carter assume that the global optimum can be found by doing local searches at each node. We have not made this assumption and in fact provide a wide variety of search methods for trying to find an optimal implementation.

UHFFT is another package for computing FFTs (Mirković and Johnsson, 2001). While not sharing any code, UHFFT is basically FFTW with a few extensions. UHFFT extends the break down rules that can be used to combine codelets beyond Cooley-Tukey to two others: split-radix and prime factor. They also seem to use a model to find fast implementations based on performance data of the codelets, but this is unfortunately not explained in detail.

8.3 Linear Algebra Algorithm Optimization

In the closely related field of linear algebra, optimization has followed a somewhat similar development as in signal processing. The Basic Linear Algebra Subprograms (BLAS) (Lawson et al., 1979; Dongarra et al., 1988, 1990) has become a standard library of basic linear algebra routines, consisting of vector and matrix operations. Computer platform vendors are encouraged to develop highly tuned implementations of the BLAS for their particular platform, allowing applications that use the BLAS to be efficient across a variety of platforms. However, there has been some more recent work in developing automatic methods for finding fast implementations for the BLAS.

Both PHiPAC (Bilmes et al., 1997, 1998) and ATLAS (Whaley and Dongarra, 1998) search for optimal implementations of matrix multiplication, and PHiPAC can also search for optimal implementations of a number of other linear algebra algorithms. Both approaches have developed a set of parameterized linear algebra algorithms. This constitutes a significant portion of their contribution and is roughly similar to the known factorizations of signal processing algorithms. PHiPAC and ATLAS then perform optimization by searching over the space of possible parameters to these algorithms. Generally, several "exhaustive" searches are run on small portions of the search space, guided by a pre-specified overall search. While their work requires a fair bit of intelligent selection of portions of the parameter space to search over, our work has concentrated more on developing good search algorithms that consider the entire space of possible implementations.

Using PHiPAC, Vuduc et al. (2001; 2000) use machine learning and statistical modeling to optimize matrix multiplication. Specifically, they use PHiPAC to generate three different implementations of matrix multiplication. Then they wish to choose the best implementation given the sizes of the matrices to be multiplied. They train up three different methods to predict the fastest implementation: (1) a support vector machine, (2) a regression model, and (3) a novel cost minimization method. Their use of machine learning to develop models of performance is similar to ours. However, we consider literally thousands of different implementations, while they only choose between three. On the other hand, they learn across all of the dimensions of the input matrices, while we only learn across a single transform size.

Also, Vuduc et al. (2001; 2000) present an interesting early stopping criterion for random search based on a statistical test. This statistical test allows random search to be stopped early when it is sufficiently likely that the search has already found a sufficiently fast implementation (where sufficiently likely and sufficiently fast can be user defined). This test could potentially be implemented in the SPIRAL system's random search method. Unfortunately, the method assumes a uniform random distribution over generated random implementations, which is not the case with the SPIRAL system's random search method.

SPARSITY is a system for optimizing code for sparse matrix multiplication (Im, 2000; Im and Yelick, 2001). In their optimization for register reuse, they have developed by hand a performance model which requires a few parameters to be set based on performance data and an analysis of matrices typical to the problem at hand.

8.4 Numerical Algorithm Optimization and Modeling

Beyond the signal processing and linear algebra fields, considerable work has been invested in optimizing a broad range of numerical algorithms, for example (Press et al., 1992; Dongarra and Grosse, 2000). Again, much of this work has focused on developing efficient algorithms by hand. However, there has been some effort in developing adaptable libraries, some of which are outlined in (Veldhuizen and Gannon, 1998).

Lagoudakis and Littman (2000) use reinforcement learning to learn to select between algorithms for solving sorting or order statistic selection problems. For each of these problems, they consider using two different algorithms that perform better in different portions of the input space. Again, this is considerably different from our problem where there is a very large number of possible implementations. Their method depends on being able to select a different algorithm at each recursive step and to gather runtimes as the algorithm is being computed. They use reinforcement learning to learn which algorithm to select based on the input parameters for the current recursive call. In the SPIRAL framework, it is not easy to gather runtimes and choose break down rules in the middle of computing a particular transform. However, we likewise have borrowed concepts from reinforcement learning to construct fast implementations of signal transforms, but we have done this off-line by using learned performance models.

Brewer (1995; 1994) uses statistical modeling to optimize algorithms for parallel machines. In Brewer's framework, the user provides a few different implementations to solve the same problem. This is in sharp contrast to the very large number of different signal processing implementations available. Given a few different implementations, Brewer's system then runs the implementations with different input parameters. The system then uses linear regression to predict that implementation's runtime across different inputs. The terms or features that are used in the linear regression must be specified by the user and are based on the problem inputs. Then, given a specific set of problem inputs, the system chooses to use the implementation whose model predicts the fastest runtime. While we also have developed models of performance, we have developed the models along different dimensions. With so few different implementations, Brewer develops a separate model for each implementation, but the model predicts along different problem inputs. Likewise, we have also developed models that predict well across different transform sizes. However, our work has also developed models of performance across the different implementations of the same transform so that a single model could predict the runtime of all the different implementations.

Gatlin and Carter (1999) describe what they call an architecture-cognizant approach to divide and conquer algorithms. They specifically give results for matrix multiply and 2D Point Jacobi. They consider two or three different alternatives that can be chosen at each level in the divide and conquer algorithm. They assume that at least one of these alternatives is what they call an isolator variant which allows searching for fast implementations of any subtree below the variant without considering alternatives for other portions of the tree. To search for a fast implementation, they then use a modified form of dynamic programming that takes into account these isolator variants. Unfortunately, in our problem, we do not have isolator variants since runtime and even the set of legal split trees are dependent on the parent chain of any node in our split trees.

Mitchell et al. (2001) run a number of benchmarks on a given machine. Using this performance information, regression is then used to instantiate the parameters of pre-defined performance models. Each of these models correspond to a different pattern of memory access. Given a piece of code to be optimized, a static analysis of its array reference patterns is performed. These array reference patterns are matched to their corresponding performance models. Guided by the performance models, legal transformations of the access patterns are considered. In the experiments shown they only choose between two different transformations.

8.5 Tuning Compiler Optimizations

While we have focused on optimizing implementations from a particular field, some similar research has been conducted in trying to find the right compiler optimizations to apply to arbitrary code so as to produce the most efficient executable. Some of the same issues are raised in that there are many different compiler optimizations or transformations that can be applied. Different choices of these optimizations can produce code with significantly different runtimes. This section surveys some of the efforts that have attempted to adapt their choice of compiler optimizations to a given architecture based on runtime feedback.

To learn to schedule straight-line code for an Alpha 21064 architecture, Moss et al. (1997) describe a method that uses supervised learning and McGovern and Moss (1998) describe a related method that uses reinforcement learning. They consider different legal re-orderings of the instructions within a single basic block, using a greedy scheduler that tries to find the best next instruction to add to the partially grown schedule. They define the learning task to determine a preference between two candidate instructions to be added next to the schedule. With such a predictor learned, they could find the best next instruction to schedule by asking it to predict for each pair of possible instructions that could be scheduled next.

This work is similar to ours in that they use machine learning techniques to try to optimize a piece of code. However, their framing the learning task to learn a preference between pairs of alternatives differs from our approach to learn to predict performance for alternatives. They assume that at any point in the schedule there are relatively few possible instructions to consider, while we are forced to consider many different ways to split a node in a split tree. Like our work, one of the most important and challenging problems they face is to develop a feature set for this scheduling task.

One further contrast to our work is that these code schedulers are both trained and tested on runtime information collected from a simulator. This simulator does not account for any stalls due to cache or TLB misses. We have always evaluated our work against actual runtimes collected on real machines. Cache performance is of great concern in our application area and can not be ignored.

GAPS uses a genetic algorithm approach to optimizing the compilation of Fortran programs for parallel architectures (Nisbet, 1998). The genetic algorithm specifically considers different optimization transformations that can be applied and can be applied in different orders. Similar to STEER, GAPS uses domain specific knowledge in its evolutionary operators. Further, GAPS actual compiles code and runs it to evaluate individuals just as STEER does.

Iterative compilation is used to optimize which transformations are applied during complication of different applications for an embedded processor (Kisuki et al., 2000; Bodin et al., 1998a). This work compiles different implementations of the same source code, searching for the fastest one. Specifically, a grid search is conducted over possible values for parameters to each of the transformations. This type of search was not feasible in our problem domain since their was no smooth space of parameters to be tuned but rather simply a set of different formulas to be searched through.

GCDS chooses between different transformations to apply to different blocks of code, trying to optimize for both speed and code size for an embedded processor (Bodin et al., 1998b). To do this, GCDS uses a pre-defined model which has parameters that are set from code profiling. While we have looked at optimizing different performance measures, we have not tried to optimize multiple performance measures simultaneously like GCDS.

Chow and Wu (1999) use fractional factorial design to control their search over different compiler options to use on an IA64. They consider nine compiler options which can either be turned on or off. This provides them a larger search space to consider than many other approaches consider, but still is relatively small (2^9) in comparison to our many thousands of different implementations.

8.6 Combinatorial Problems and Machine Learning

Combinatorial problems such as satisfiability, the traveling salesman problem, and graph coloring present an interesting challenge to design algorithms that are as efficient as possible for as many instances as possible. We are not aware of any approaches to combinatorial problems that try to tune themselves to the given computer architecture. Instead, there has been a considerable amount of work in trying to reduce the total number of nodes visited in the search tree explored since such a reduction can have a huge impact on runtime. While much of the work in this area has focused on developing good heuristics, this section focuses on work that automatically tunes an approach to the combinatorial problem for a given instance or class of instances.

STAGE learns to improve search performance for local search algorithms for a wide variety of optimization problems (Boyan and Moore, 1998; Boyan, 1998). Local search algorithms such as hill climbing and simulated annealing impose a neighborhood relation on different potential solutions or states. Local search algorithms begin with one state and then consider other neighbors, looking for either real solutions or better quality solutions. STAGE learns an evaluation function that predicts the outcome of applying a particular local search algorithm starting from different states. A hill climbing search then uses this evaluation function to find a good state in which to start the original local search algorithm. This process is repeated, each time retraining the evaluation function on the new trajectory taken by the local search algorithm starting from the newly found good start state. Two important steps are left to the user: (1) defining a good set of features to use in learning the evaluation function, and (2) choosing a learning method to learn the evaluation function. It would be interesting to see how STAGE would perform at improving the SPIRAL system's hill climbing search method, but defining a good set of features may be difficult.

Zhang and Dietterich (1995; 1996) use reinforcement learning to learn how to control search for job-shop scheduling. To solve the job-shop scheduling problem, they use an iterative improvement algorithm that has only 2 main improvements that can be applied to the current potential solution. However, these improvements can be applied to different subparts of the potential solution. They use $TD(\lambda)$ to learn a value function on different states or potential solutions. This value function is then used with a one-step look-ahead random-sample greedy search to determine the next improvement to make. One of the important steps in their work is to define a good set of features to describe a state or potential solution in this domain. They perform training on smaller sized instances and are able to get good results applying their learned value function to larger sized instances.

Moll et al. (1999; 2000) describe two different variations on the previous two works (those of Boyan and Moore and of Zhang and Dietterich). One of these variations tries to combine the best of both of the previous two methods.

Adapting their previous work in algorithm selection (Lagoudakis and Littman 2000; also discussed in Section 8.4), Lagoudakis and Littman (2001) use reinforcement learning to choose between seven different branching rules when trying to solve #SAT, the problem of finding the number of satisfying assignments for a given boolean formula. Branching rules select which literal to assign a value next during the search and thus can greatly impact the size and shape of search tree. Again one of the challenging problems they faced was determining features to use to describe the current state of the search. Unfortunately, they had difficulty in this direction and choose to only use one simple feature, the number of variables remaining to be assigned. They then use reinforcement learning to learn to choose between the different branching rules given the number of variables.

8.7 Artificial Intelligence and Signal Processing

Artificial intelligence has been widely used in the field of signal processing. However, most of this work has been applying artificial intelligence techniques to signal processing applications instead of optimizing signal transforms. For example, artificial intelligence has been used in filtering, imaging processing, speech processing, and pattern recognition. Neural networks are a commonly used technique as evidenced by the ongoing IEEE Workshop on Neural Networks for Signal Processing and (Luo and Unbehauen, 1997). Additionally, evolutionary algorithms have been applied as well, for example (Sharman et al., 1995).

Chapter 9

Conclusions and Future Work

The goal of this thesis has been to develop methods to automate the modeling and optimization of the performance of signal transforms. We have been particularly interested in developing methods that use learned performance models to aid in optimizing performance so that a fast implementation can be obtained more quickly than with standard search methods. To do this, we have focused on learning to control the generation of formulas, learning to make the best choices in how to factor a transform.

We have developed a powerful technique for generating optimized implementations of signal transforms by using automatically learned performance models. Without timing a single formula of a given transform size, our method is able to construct fast formulas for a given transform by making good choices in how that transform should be factored. These constructed formulas usually have runtimes within 6% of the fastest formula ever found and often this fastest formula is one of the first 50 formulas constructed. Thus, this generation method eliminates the need of search methods to time a large number of formulas. Instead, for a given transform our method times a few formulas of one size to train machine learning techniques to automatically learn to predict performance. With these performance models, our method then learns to control the generation of formulas of the given transform across both smaller and larger transform sizes.

We have tested this generation method in several different situations, but there also remains work to extend it to others. For each new transform, a new performance model must be automatically learned. We have done this for two different transforms, namely the WHT and the FFT. Likewise, for each new computing platform, a new performance model must also be automatically learned. We have tested our method on both a Pentium and a Sun and would expect that it should immediately work on any uniprocessor. Given a performance model for a given transform and computing platform, our method is able to immediately generate fast implementations across many sizes.

To provide performance models for this generation method, we used machine learning techniques to automatically learn to predict performance of signal transform implementations. With performance data being easily collected by simply running a set of formulas, the difficult problems were defining the appropriate machine learning task and choosing a good set of features to describe signal transform factorizations. We contributed the idea of framing the machine learning task to learn to predict performance for individual nodes in split trees. While the predictors were trained on this task, they could still accurately predict for entire formulas. Also, this task allowed them to learn to predict accurately across transform sizes. Further, we have contributed several different feature sets to describe signal transform split trees and individual nodes within those split trees.

Prior to developing our generation method that constructs fast signal transform implementations, we had implemented the search engine and its variety of different search methods. These provided an opportunity to explore the search space of several different transforms. While the generation method has the advantage of being able to construct fast implementations for many transform sizes without performing a search, the search engine has the advantage currently of being able to optimize many different transforms including new, user-defined transforms. The generation method has only been tested with the FFT and WHT, and other transforms may require developing new features.

9.1 Contributions

This thesis makes three major contributions:

1. **A search engine with several different search methods for finding fast implementations of a variety of signal transforms.**
   We have developed a search engine within the SPIRAL system for optimizing a wide variety of signal transforms, including new user-specified transforms. The

search engine includes several different search methods, including exhaustive search, dynamic programming, random search, hill climbing search, STEER, and timed search. The variety of search methods allow the large search space to be explored in different ways, with different search methods outperforming the others depending on the chosen transform and size. Most of these search methods can search over both formulas and compiler options, allowing the search methods to fully tune the resulting implementations.

We developed a new search technique for this domain, STEER, which uses a stochastic evolutionary approach to optimize signal transforms. STEER outperforms all other search methods at finding fast implementations of small sized DTTs. The timed search method allows a user to easily apply several of the different search methods to finding a fast implementation while limiting the search to a specified length of time.

2. **Automatic methods for modeling and predicting performance of signal transforms.**

   We have developed methods to automatically model the performance of signal transforms. We looked at trying to explicitly model both the performance of entire formulas and then the performance of individual nodes within split trees. We found that modeling performance of entire formulas only worked well for small sizes, while modeling performance of individual nodes worked well at larger sizes. Our performance models for individual nodes could be used to predict performance for entire formulas by summing predictions made for individual nodes.

   These models were automatically learned by providing machine learning techniques with performance data collected on a real machine by running different implementations. We have developed and analyzed a number of different feature sets for describing FFT and WHT formulas and their split tree nodes.

   By modeling performance of individual split tree nodes, we have obtained excellent prediction results. We have shown this to be effective for (1) different transforms, (2) different machines, and (3) different performance measures. Further, this method can be trained on data from one transform size and still predict very well across many sizes, both larger and smaller.

3. **A method for automatically generating fast implementations.**

   We have developed a method that can automatically *generate* fast FFT and

WHT implementations. This method uses the learned performance models for individual split tree nodes. The very first formula that this method produces often has a runtime that is within 6% of the runtime of the fastest known formula. Further, this method is able to construct the fastest implementations known usually within the first 50 formulas that it generates. These results are particularly good for several reasons:

- There is a very wide spread of runtimes for a transform, often a factor of 2 to 10, and yet this method can often get within 6% of the fastest known with its very first formula constructed.
- There are literally tens of thousands of formulas in the search spaces our method considers for some transform sizes, and yet this method can usually generate the fastest known formula within the first 50 that it produces (often within much fewer than 50).
- This method never sees a timing for formulas other than those of size 2^{16} and yet it can construct fast formulas for both larger and smaller sizes, from 2^{12} to 2^{20}.

We have demonstrated that this generation method can work well on different machines (Pentium and Sun), with different transforms (FFT and WHT), and with different performance measures (runtime and cache misses).

9.2 Future Directions

There are many interesting ways in which this work could be extended. We begin with some general extensions to the overall contributions of this thesis. Many of these extensions would influence both the modeling and optimization as well as the entire SPIRAL system. Then, we discuss more specific extensions toward particular contributions of this thesis.

9.2.1 General Extensions

In this thesis, we have focused on standard uniprocessor workstations. It would be interesting to extend this work to multiprocessors or to specialized hardware design. On multiprocessors, there may be many significantly different ways to implement the same factorization in code. These new degrees of freedom may require extending our optimization and modeling techniques to account for them. When optimizing signal transforms for special purpose hardware, it may be necessary to not only consider runtime, but also a number of other performance measures such as code size, energy consumed, number of gates used, or amount of wiring necessary.

All of this work has assumed that the user has an arbitrary input vector. However, for many applications, the input vectors used may be highly structured allowing for even faster implementations. It is not clear how this might influence the optimization and modeling of transforms tuned to particular classes of input vectors.

This work has focused on signal transforms without respect to the context in which they are run in real applications. The state of the machine and particularly the state of the cache at the time the transform starts computing could potentially make a significant impact on the performance of different implementations. Analyzing real applications may also provide an opportunity to extend this work to other signal processing algorithms beyond just transforms.

Chapter 8 on related work discussed many different fields in which automated optimization have been applied. It would be interesting to see how easily this work could be extended to any of those application fields or many others.

9.2.2 Search Engine Extensions

There are many different extensions that could be made to the search engine. We begin with some extensions to the existing search methods and then discuss broader issues about the entire search engine.

Random split tree generation does not produce trees uniformly across all possible trees. Having a uniform random generation method may be preferable to the current method. However, this requires determining the number of all possible subtrees for each of the children of a given transform and size. With a uniform random tree generation method, it would then be interesting to extend our random search method with the early stopping method suggested by Vuduc et al. (2001).

While we have implemented a hill climbing search method, we have not spent much time tuning its default options. Further, it could be fairly easily extended to be simulated annealing, allowing for split trees with worse runtimes to be chosen occasionally. Extending the hill climbing search method to use STAGE (Boyan and

Moore, 1998) to aid in optimization would also be interesting, although developing the right set of features might be difficult.

In the search engine, STEER has a very limited set of mutations. One of the difficulties of trying to find new or better mutations is that they need to work with any transform or break down rule that is defined to the SPIRAL system. One method around this would be to design mutations that only could be applied when certain conditions were met for the transform or break down rule in question. STEER's default options also have not been well tuned within the search engine. It would be interesting to see how changing options such as population size, number of mutations and cross-overs, and number of generations influences STEER's performance.

An interesting question would be whether seeding STEER's initial population with good split trees would be beneficial. These good split trees could come from a previous run of dynamic programming or from the generation method. However, sometimes seeding a genetic algorithm with good individuals can cause it to perform poorly because diversity is quickly eliminated from the population as the good individuals strongly dominate the random ones.

Another possible extension to STEER would be to allow it to evolve several populations simultaneously. These different populations could consist of the same transform but across different sizes, or even different transforms and sizes. This would allow a good split tree found for one transform and size to be crossed-over with a subtree in a larger sized transform, for example.

While the search engine already contains quite a few different search methods, there may still be others worth trying. One idea would be to profile different rules while generating random split trees. Several random split trees could be generated and timed such that every possible rule and set of immediate children is generated for the root. After enough data is collected, then split trees could be generated such that the root is more likely to be split with rules and into immediate children that have been found through the profiling to have faster runtimes. At this stage, then profiling could continue at the level of the immediate children of the root, and the process repeated.

While the search engine in the SPIRAL system has only been tested for optimizing runtimes, it should work immediately for any other performance measure. However, it is not clear how this might influence the relative performance of the different search methods. Further, it would be interesting to extend the system to try to optimize multiple performance measures simultaneously.

Currently most of the search methods can search over both local and global unrolling options to the SPL compiler. Further, the user can specify a set of options to pass to the native C or Fortran compiler. However, it is non-trivial to extend the search methods to search over other SPL compiler options or to search over any options to the native compiler. One of the difficulties in doing this is that different compiler options require different types of searches by the various search methods. For example, how dynamic programming searches over local unrolling is very different from how it searches over global unrolling, and both of these are different than how STEER searches over each of them. It may be possible, though, to come up with categories that most options fall into and then augment each of the search methods to correctly search over any option specified for each of the different categories.

9.2.3 Modeling and Generation Extensions

Many of the extensions that would be desirable for our modeling work would likewise be desirable for our generation of fast implementations, largely due to the fact that the generation method is heavily dependent on the predictors that it uses.

All of our previous modeling and generation work has focused on particular transforms and took advantage of particular properties of those transforms. Extending this work to handle a broader range of transforms or even arbitrary user-specified transforms would be desirable. However, this presents some challenges. For example, the DTTs are considerably different from the FFT and the WHT in that they have many different break down rules for one transform and yet those break down rules have no degrees of freedom in application (such as how to factor the size). This would require investigating into different feature sets.

Since the SPIRAL system has more break down rules for the FFT than just the Cooley-Tukey, it would be desirable to extend our modeling work to consider these other rules. This would require incorporating features that indicate what rule is used at different nodes. Further, at least one of the break down rules for the FFT factor it into DTTs. Our feature sets have assumed binary split trees, but both the WHT and the FFT can have non-binary split trees.

Extending our modeling and generation methods to work without the use of specially designed leaves would be desirable. One of the reasons that we created leaves for the FFT was that it was difficult to get accurate timings for very small transforms, particularly size 2^1. It may be possible to get accurate times by other methods or to compensate in some way for the inaccurate timings that are obtained. Changing exactly what is modeled may also alleviate the need for timings of such small sizes; for example, with the Grandchildren feature set, nodes with grandchildren that are of size 2^1 could account for the runtime of the node's entire subtree.

All of the models discussed in this thesis were trained on and predicted for one particular transform and machine. Learning across different transforms and machines so that the learned model could still predict well given a new transform and/or new machine would be very exciting but probably rather difficult. This would require new features that describe transforms and their associated break down rules and new features that would describe different architectures. Being able to collect enough data to learn across different transforms or different machines could be difficult. To collect data across different architectures, it may be possible to use a good simulator where different architectural parameters can be changed (e.g., cache size). Having a model that could predict well across different architectural parameters could also be useful in guiding the design of new architectures.

In this thesis, we used basically one training methodology without exploring how variations could impact performance for the predictors or the generation method. It would be good to minimize the amount of data collected for training, and so studying the impact of training set size on performance would be useful. Further, it would be interesting to know if training with data across several transform sizes could allow the predictor to perform well with less training data. It could also be useful to have a method that could design the particular formulas that it timed, having it attempt to maximize the information gained with each formula timed.

Finally, as an engineering task, integrating the learning and generation method into the SPIRAL system would allow these methods to be more widely used.

9.3 Concluding Remarks

This thesis has shown the usefulness of machine learning techniques in optimizing signal transform implementations. One of the major contributors to this success is the fact that runtime performance data can be easily collected for a set of different signal transform implementations. This ability to obtain runtime performance data allowed for feedback during search and provided data to train machine learning techniques to be able to automatically learn to predict performance for different implementations. Further, defining a good machine learning task and a good set of features were key to being able to accurately model performance. Finally, we have developed a novel method for controlling the generation of formulas to immediately produce the fastest ones without even timing a single formula of the given size.

Bibliography

L. Auslander, Jeremy R. Johnson, and R. W. Johnson. Automatic implementation of FFT algorithms. Technical Report 96-01, Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA, June 1996.

L. Auslander, Jeremy R. Johnson, and R. W. Johnson. Dimensionless fast Fourier transforms. Technical Report DU-MCS-97-01, Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA, 1997.

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

Rudolf Berrendorf and Bernd Mohr. PCL — The Performance Counter Library, 2000. `http://www.fz-juelich.de/zam/PCL/`.

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*, pages 340–347, July 1997.

Jeff Bilmes, Krste Asanović, C. Chin, and Jim Demmel. The PHiPAC v1.0 matrix-multiply distribution. Technical Report TR-98-35, International Computer Science Institute, Berkeley, CA, 1998.

F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation, in conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT98)*, 1998a.

Francois Bodin, Zbigniew Chamski, Christine Eisenbeis, Erven Rohou, and André Seznec. GCDS: A compiler strategy for trading code size against performance in embedded applications. Technical Report RR-3346, Institut National de Recherche en Informatique et en Automatique (INRIA), 1998b.

J. A. Boyan. *Learning Evaluation Functions for Global Optimization*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1998.

J. A. Boyan and A. W. Moore. Learning evaluation functions for global optimization and boolean satisfiability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI)*, 1998.

Eric A. Brewer. *Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization*. PhD thesis, Massachusetts Institute of Technology, 1994.

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*, pages 80–91, 1995.

C. S. Burrus. Notes on the FFT, 1997. `http://www-dsp.rice.edu/research/fft/fftnote.ps.gz`.

S.C. Chan and K.L. Ho. Direct methods for computing discrete sinusoidal transforms. *IEE Proceedings*, 137(6):433–442, 1990.

K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *Proceedings of the 2nd workshop on Feedback-Directed Optimization*, 1999.

E. M. Clarke, M. Fujita, and W. Heinle. Hybrid spectral transform diagrams. Technical Report CMU-CS-97-149, Computer Science Department, Carnegie Mellon University, 1997.

J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematical Computation*, 19:297–301, 1965.

J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subroutines. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

Jack Dongarra and Eric Grosse. Netlib, 2000. `http://www.netlib.org`.

S. Egner and M. Püschel. *AREP – Constructive Representation Theory and Fast Signal Transforms*. GAP share package, 1998. `http://www.ece.cmu.edu/~smart/arep/arep.html`.

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*, volume 3, pages 1381–1384, 1998a.

Matteo Frigo. A fast Fourier transform compiler. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180, 1999.

Matteo Frigo and S. G. Johnson. benchFFT FFT software, 1998b. `http://www.fftw.org/benchfft/doc/ffts.html`.

GAP. *GAP – Groups, Algorithms, and Programming*. The GAP Team, University of St. Andrews, Scotland, 1997. `http://www-gap.dcs.st-and.ac.uk/~gap/`.

Kang Su Gatlin and Larry Carter. Architecture-cognizance divide and conquer algorithms. In *Proceedings of Supercomputing '99*, 1999.

Kang Su Gatlin and Larry Carter. Faster FFTs via architecture-cognizance. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2000)*, 2000.

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

Gavin P. Haentjens. An investigation of Cooley-Tukey decompositions for the FFT. Master's thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, May 2000.

M. T. Heideman. *Multiplicative Complexity, Convolution, and the DFT*. Springer-Verlag, New York, 1988.

Eun-Jin Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, Department of Computer Science, University of California at Berkeley, Berkeley, CA, 2000.

Eun-Jin Im and Katerhine Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Science, Lecture Notes in Computer Science 2073*, pages 127–136. Springer-Verlag, 2001.

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*, volume 31, pages 378–387, April 1983.

J. Johnson, R. Johnson, D. Padua, and J. Xiong. Searching for the best FFT formulas with the SPL compiler. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing*, 2000.

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*, pages 3347–3350, 2000.

T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and H.A.G. Wijshoff. Iterative compilation in program optimization. In *Proceedings of the 8th International Workshop on Compilers for Parallel Computers (CPC2000)*, pages 35–44, 2000.

Pinit Kumhom. *Design, Optimization, and Implementation of a Universal FFT Processor.* PhD thesis, Drexel University, Philadelphia, PA, 2001.

Pinit Kumhom, Jeremy R. Johnson, and Prawat Nagvajara. Design, optimization, and implementation of a universal FFT processor. In *Proceedings of the 13th Annual IEEE International ASIC/SOC Conference*, pages 182–186, 2000.

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

Michail G. Lagoudakis and Michael L. Littman. Learning to select branching rules in the dpll procedure for satisfiability. In *Electronic Notes in Discrete Mathematics (ENDM), Volume 9, LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, 2001.

C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

Fa-Long Luo and Rolf Unbehauen. *Applied Neural Networks for Signal Processing.* Cambridge University Press, 1997.

D. Maslen and D. Rockmore. Generalized FFTs – A survey of some recent results. In *Proceedings of the 1995 DIMACS Workshop in Groups and Computation*, pages 183–238, 1995.

Amy McGovern and Eliot Moss. Scheduling straight-line code using reinforcement learning and rollouts. In *Proceedings of the 11th Neural Information Processing Systems Conference (NIPS '98)*, 1998.

Dragan Mirković and Lennart Johnsson. Automatic performance tuning in the uhfft library. In *Proceedings of the International Conference on Computational Science, Lecture Notes in Computer Science 2073*, pages 71–80. Springer-Verlag, 2001.

M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, 1996.

Nick Mitchell, Larry Carter, and Jeanne Ferrante. A modal model of memory. In *Proceedings of the International Conference on Computational Science, Lecture Notes in Computer Science 2073*, pages 81–96. Springer-Verlag, 2001.

R. Moll, A. G. Barto, and T. J. Perkins. Learning instance-independent value functions to enhance local search. In *In Advances in Neural Information Processing Systems 11 (NIPS-11)*, pages 1017–1023, Cambridge, MA, 1999. MIT Press.

R. Moll, T. J. Perkins, and A. G. Barto. Machine learning for subproblem selection. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 615–622, San Francisco, 2000. Morgan Kaufmann.

J. Eliot B. Moss, Paul E. Utgoff, John Cavazos, Doina Precupand Darko Stefanovic, Carla Brodley, and David Scheeff. Learning to schedule straight-line code. In *Proceedings of Advances in Neural Information Processing Systems 10 (NIPS '97)*, 1997.

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/`.

Andy P. Nisbet. GAPS: Iterative feedback directed parallelisation using genetic algorithms. In *Workshop on Profile and Feedback-Directed Compilation, in conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT98)*, 1998.

H. J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer-Verlag, Heidelberg, Germany, 2nd edition, 1982.

N. Park, D. Kang, K. Bondalapati, and V. K. Prasanna. Dynamic data layouts for cache-conscious factorization of DFT. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2000.

N. Park and V. K. Prasanna. Cache conscious Walsh-Hadamard transform. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 2001.

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 2nd edition, 1992.

Markus Püschel, Bryan Singer, Manuela Veloso, and J. Moura. Fast automatic generation of DSP algorithms. In *Proceedings of the International Conference on Computational Science, Lecture Notes in Computer Science 2073*, pages 97–106. Springer-Verlag, 2001a.

Markus Püschel, Bryan Singer, Jianxin Xiong, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing Applications*, 2001b. Submitted.

John Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1992.

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

Martin Schönert et al. *GAP – Groups, Algorithms, and Programming.* Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, fifth edition, 1995.

David Sepiashvili. Performance models and search methods for optimal FFT implementations. Master's thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, May 2000.

Ken C. Sharman, Anna I. Esparcia Alcazar, and Y. Li. Evolving signal processing algorithms by genetic programming. In *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, pages 473–480, 1995.

Bryan Singer and Manuela Veloso. Automated formula generation and performance learning for the FFT. Technical Report CMU-CS-00-123, Computer Science Department, Carnegie Mellon University, 2000a.

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*, pages 887–894, San Francisco, 2000b. Morgan Kaufmann.

Bryan Singer and Manuela Veloso. Learning to generate fast signal processing implementations. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 529–536, San Francisco, 2001a. Morgan Kaufmann.

Bryan Singer and Manuela Veloso. Stochastic search for signal processing algorithm optimization. In *Proceedings of the ACM/IEEE SC2001 Conference*, 2001b.

Gilbert Strang. The discrete cosine transform. *SIAM Review*, 41(1):135–147, 1999.

R. Tolimieri, M. An, and C. Lu. *Algorithms for Discrete Fourier Transforms and Convolution*. Springer-Verlag, New York, 2nd edition, 1997.

L. Torgo. *Inductive Learning of Tree-based Regression Models*. PhD thesis, Department of Computer Science, Faculty of Sciences, University of Porto, 1999.

Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compiler and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*. SIAM Press, 1998.

Martin Vetterli and Henri J. Nussbaumer. Simple FFT and DCT algorithms with reduced number of operations. *Signal Processing*, 6:267–278, 1984.

Richard Vuduc, Jeff Bilmes, and James W. Demmel. Statistical modeling of feeback data in an automatic tuning system. In *MICRO-33: Workshop on Feedback-Directed Dynamic Optimization*, 2000.

Richard Vuduc, James W. Demmel, and Jeff Bilmes. Statistical models for automatic performance tuning. In *Proceedings of the International Conference on Computational Science, Lecture Notes in Computer Science 2073*, pages 117–126. Springer-Verlag, 2001.

Richard Vuduc and James W. Dremmel. Code generator for automatic tuning of numerical kernels: Experiences with FFTW. In *ICFP 2000: Workshop on Semantics, Application, and Implementation of Program Generators*, 2000.

Z. Wang. Fast algorithms for the discrete w transform and for the discrete fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-32(4):803–816, 1984.

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.

Jianxin Xiong. *Automatic Optimization of DSP Algorithms*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, 2001.

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)*, pages 298–308, 2001.

W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1114–1120, 1995.

Wei Zhang. *Reinforcement Learning for Job-Shop Scheduling*. PhD thesis, Computer Science Department, Oregon State University, Corvallis, Oregon, 1996.

Xudong Zhao. *Verification of Arithmetic Circuits*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1996.

Appendix A

Search Engine User Manual

This appendix contains the user manual for the search engine. This manual is available as a text file when downloading the SPIRAL system. Also, the manual is available at the SPIRAL prompt by typing:

```
spiral> ?Search
```

to begin reading the search engine user manual. Further, the user can request at the SPIRAL prompt information about any of the search methods or most of the other functions available in the search engine simply by typing a question mark followed by the function name.

A.1 The Search Engine

This chapter describes the search module. The search module uses different search algorithms to find fast implementations of given transforms (or more generally SPLs).

The first sections begin with information about a couple high level functions for interacting with the search module (see "Implement" and "TestSearch"). "Implement" allows an SPL to be quickly implemented. "Implement" can be used to search for a fast implementation, but only uses the default search options for the different search methods. "TestSearch" runs a number of quick tests of the search module.

The next sections give a much more detailed explanation of the different search methods provided and how they can be called separately (see “Search Method Overview”, “General Search Options”, “ExhaustiveSearch”, “DP”, “RandomSearch”, “HillClimb”, “STEER”, and “TimedSearch”). “ImplementRuleTree” can then be used to implement the best ruletrees that these search methods find.

The next sections give some more details about how the search module keeps track of the best found implementations and how you can retrieve these (see “BestFoundTable”, “BestFoundLookup”, “BestFoundSave”, “BestFoundRead”).

The final sections discuss HashTables and how you can use them to store partial results from the different search methods and then later re-use those results (see “HashTables”, “HashSave”, “HashRead”, “HashTable Creation”, and “HashTable Example”).

Type “?>” to read the next section of the search module documentation.

A.2 Implement

```
Usage:  Implement( <spl> [, <Implement-search-options> ] )
```

Implements the <spl> in code, creating a file containing the code.

Returns: a record with the ruletree implemented, its runtime in nanoseconds, the SPL Options used, and the file name where the code was written.

By default, Implement will use the BestFound Implementation when available and create the corresponding code. If no BestFound Implementation is available for the specified spl, Implement will then conduct, by default, a 30-minute search for a fast implementation. These defaults can be changed by passing an options record.

Optionally, an options record can be passed containing any of the following fields:

```
   spiral> PrintSpecImplementOptionsRecord();
   rec(
    file := <filename>,
    language := "f77" | "c",
    search :=   "BestOrTimedSearch" | "BestFound"
                 | "DP" | "TimedSearch"
                 | "OneRandom" | "RandomSearch"
                 | "HillClimb" | "STEER"
                 | "ExhaustiveSearch",
    timeLimit := false | <minutes>,
    searchOpts := <searchOptionsRecord>,
    SPLOpts := <SPLOptionsRecord>,
    verbosity := <non-negative integer>
  );
```

Specifying file changes the name of the file to be created with the resulting code. This filename should not contain an extension (that is no ".c" or the like). The file name should be passed as a string. The default file name is created by "DefaultFileName-SPL". Generally this is a file in the current directory with the file name containing the transform name and its parameters.

Specifying the language changes what language the resulting code is implemented in. Note that currently search methods BestOrTimedSearch, BestFound, and Timed-Search only return the best found implementation which may be in a different language (this will be fixed in the future). A different set of languages may be available on your machine, call PrintSpecImplementOptionsRecord() to see which languages are available. This setting overrides any setting of language in the SPLOpts.

Valid search methods for finding fast implementations are:

```
  "BestOrTimedSearch" : uses BestFound implementation,
                        or a 30-minute "TimedSearch" if none
  "BestFound"         : uses BestFound implementation, if it exists
  "OneRandom"         : creates a single random implementation
  "TimedSearch"       : uses several search methods for a specified length
                        of time, returns BestFound implementation
  "RandomSearch"      : picks best out of a number of
                        random implementations
```

```
"DP"                : dynamic programming search
"STEER"             : genetic algorithm
"HillClimb"         : performs a hill climbing search
"ExhaustiveSearch"  : exhaustive search
```

The search method "BestOrTimedSearch" is used by default.

Specifying an integer for the timeLimit causes the search algorithm to be limited to (roughly) that many minutes. This option overrides any given in searchOpts.

Specifying the searchOpts provides the opportunity to provide the search method with extra options. This must be a valid options record for the specified search method.

Specifying the SPLOpts changes the options passed to the SPL compiler for producing the implementation. Note that search methods that return a BestFound implementation may override these options.

Verbosity levels:

```
0 = Print nothing, tell search algorithm to print nothing
1 = Print nothing
2 = Print final best
3 = Print methods used to determine final implementation
```

Example calls:

```
Implement( Transform("DFT", 8) );
   Will implement a DFT of size 8 in the file named DFT_8.f77 (if f77
   is the default language).  Will use the BestFound Implementation
   if available.  Otherwise, will conduct a 30-minute search.

Implement( Transform("DFT", 8), rec(timeLimit:=5) );
   Same as above, but will only conduct a 5-minute search if no
   BestFound implementation is available.

Implement( Transform("DFT", 8), rec(search:="BestFound") );
   Only implement the transform in code if a BestFound Implementation
```

```
     is available.

  Implement( Transform("DCT4",32),
             rec(search:="DP", language:="c", file:="mydct") );
     Implement a DCT type 4 of size 32 in the file "mydct.c" using the
     C programming language.  Search for a fast implementation using
     dynamic programming (DP).

  Implement( Transform("WHT",10),
             rec(search:="STEER",
                 searchOpts:=rec(localUnrolling:=true),
                 SPLOpts:=rec(dataType:="complex")) );
     Implement a WHT of size 2^10 that works on complex input data.
     Use STEER to search for a fast implementation and allow it to
     search over localUnrolling.
```

A.3 TestSearch

Call “TestSearch()” with no arguments to run quick tests of the search module. This is used to check that your installation is correct and that the search module is working correctly (at least for a few simple cases, the functions return reasonable values).

Note: If “TestSearch()” causes a fatal error while testing BestFound Implementation, and if you have some implementations stored in the BestFoundTable that you want, then type:

```
BestFoundTable := savedBestFoundTable;
```

to restore the BestFoundTable to its state prior to “TestSearch()” being called.

A.4 Search Method Overview

There are five main search methods provided:

```
'ExhaustiveSearch' -- exhausts over all possible ruletrees.
'DP'               -- performs dynamic programming.
'RandomSearch'     -- generates random ruletrees,
                      searching for the fastest.
'HillClimb'        -- performs hill climbing search.
'STEER'            -- a stochastic evolutionary search algorithm
                      (similar to a genetic algorithm).
'TimedSearch'      -- a meta-search algorithm that calls others with
                      specific time limits.
```

Each algorithm takes an "SPL" as its first argument. This can be a transform or a more general SPL for which a fast implementation is to be searched.

All of these algorithms take a search options record as an optional second argument. This record contains various options that modify how the search algorithm work. Call "`PrintSpec<SearchAlg>OptionsRecord()`" to get a list of the options available for the specific search algorithm. See the file "search/config.g" or run "`Merge<SearchAlg>OptionsRecord( rec() )`" for defaults.

All of these algorithms take an optional final argument that specifies options to the SPL compiler. Call "PrintSpecSPLOptionsRecord()" to find out more about the SPL options.

All of the algorithms return some form of data structure containing the best implementation it found.

In general exhaustive search is only possible for very small transforms. Dynamic programming usually times relatively few formulas (except when a transform has a rule with a high branching factor), but still achieves good results. STEER times more formulas than DP, but often finds faster ones. RandomSearch can be used to time as many or few formulas as one would like. STEER or HillClimb is to be preferred over RandomSearch unless you only want to time a small number of formulas.

Each of these search methods are described in detail in the following sections (see "ExhaustiveSearch", "DP", "RandomSearch", "HillClimb", and "STEER"), after a description of some of the general search options (see "General Search Options"). In the following examples, options are given for illustration not because they are good options to provide; further, runtimes were gathered on different loaded machines and thus are not representative.

A.4.1 General Search Options

The following search options are common to many of the algorithms:

```
    timeLimit := false | <minutes>,
    localUnrolling := true | false,
    localUnrollingMin := <positive int>,
    localUnrollingMax := <positive int>,
    globalUnrolling := true | false,
    globalUnrollingMin := <positive int>,
    globalUnrollingMax := <positive int>,
    bestFound := "save" | "none"
```

Time limit specifies the maximum number of minutes that a search algorithm should continue searching. Setting this to false allows the search algorithm to continue searching until it is finished. Most, but not all, search algorithms (notably not DP) can give back a reasonable result after any amount of time. Since GAP does not have an interrupt mechanism, it is quite possible for a search algorithm to take considerably more time than specified. A check on time is only conducted at certain points in the loops/recursion of the search algorithms.

Local unrolling specifies to the SPL compiler exactly which portions of the ruletree are to be unrolled. If a specific node in the ruletree is marked for unrolling, then the SPL compiler will unroll the code for that node and its subtree.

Global unrolling specifies to the SPL compiler that ALL nodes of the given size and smaller are to be unrolled.

Setting "localUnrolling" to "false" causes the search algorithms to only search through ruletrees without any nodes marked for unrolling. Setting "localUnrolling" to "true" causes the search algorithms to search through the space of ruletrees with nodes marked for unrolling. Specifically, all nodes of size "localUnrollingMin" and smaller are marked for unrolling, while those of size greater than "localUnrollingMin" and less than or equal to "localUnrollingMax" are considered both with and without unrolling. Turning on search over local unrolling turns off all global unrolling specified to the SPL compiler.

Setting "globalUnrolling" to "true" causes the search algorithms to search over different settings for the global unrolling. Specifically they search from "globalUnrollingMin" to "globalUnrollingMax", inclusive, going by factors of 2 (that is, they consider globalUnrollingMin, globalUnrollingMin$\times$2, globalUnrollingMin$\times$4, etc ... until globalUnrollingMin$\times 2^k >$ globalUnrollingMax).

It is not allowable to search over both local and global unrolling.

Setting "bestFound" to "save" causes the search modules to try to save their best found implementations for SPLs into the BestFoundTable. If they did not find one faster than previously found, then nothing happens. Setting "bestFound" to "none" causes the search modules to do nothing with the BestFoundTable. (See also "BestFoundTable".)

A.4.2 ExhaustiveSearch

```
Usage: ExhaustiveSearch( <spl>
          [, <Exhaustive-Search-options-record>, <SPLOptionsRecord> ] )
```

Note that if you wish to specify an Exhaustive-options-record or a SPL-options-record, then you must specify both. To leave one blank, just pass "rec()" in its place.

Searches over all possible ruletrees and returns the one with the fastest runtime. Currently does NOT search over local or global unrolling parameters, just different ruletrees.

Returns: fastest ruletree

Search Options:

```
spiral> PrintSpecExhaustiveSearchOptionsRecord();
rec(
  timeLimit := false | <minutes>,
  bestFound := "save" | "none",
  verbosity := <non-negative integer>
);
```

Run “MergeExhaustiveSearchOptionsRecord(rec());” to determine defaults.

Verbosity levels:

```
0 = Print nothing
1 = Pretty print final best tree
2 = Also print how many measurements to do
3 = Print all trees as measured
```

Examples:

```
spiral> W := SPLNonTerminal("WHT", 3);;
spiral> ExhaustiveSearch(W);
#I no. trees: 1 3 3 3 3 3
#I 3 measurements to do

Best Rule Tree:
WHT(3)     {RuleWHT_1}
 |--WHT(1)     {RuleWHT_0}
 `--WHT(2)     {RuleWHT_1}
     |--WHT(1)     {RuleWHT_0}
     `--WHT(1)     {RuleWHT_0}
 ! 215

RuleTree(
  RuleWHT_1,
      SPLNonTerminal( "WHT", 3 ), [
      RuleTree(RuleWHT_0, SPLNonTerminal( "WHT", 1 )),
      RuleTree(
        RuleWHT_1,
        SPLNonTerminal( "WHT", 2 ), [
        RuleTree(RuleWHT_0, SPLNonTerminal( "WHT", 1 )),
        RuleTree(RuleWHT_0, SPLNonTerminal( "WHT", 1 ))
      ] )
    ] )
    spiral> ExhaustiveSearch( W, rec(verbosity:=1), rec(language:="c") );;
    #I no. trees: 1 3 3 3 3 3

    Best Rule Tree:
    WHT(3)     {RuleWHT_1}
     |--WHT(1)     {RuleWHT_0}
     `--WHT(2)     {RuleWHT_1}
         |--WHT(1)     {RuleWHT_0}
         `--WHT(1)     {RuleWHT_0}
     ! 216

    spiral>
```

A.4.3 DP

```
Usage: DP( <spl> [, <DP-options-record>, <SPL-options-record> ] )
```

Note that if you wish to specify a <DP-options-record> or a <SPL-options-record>, then you must specify both. To leave one blank, just pass "rec()" in its place.

"DP()" runs dynamic programming on the given spl. Specifically, dynamic programming considers all applicable rules to the given transform, and all possible sets of children generated by those rules. For each child, it looks up in its table the fastest implementation of that transform. If no such entry exists yet in its table, then DP recursively calls itself on the child. Once DP has found the best implementation for the child, it substitutes this ruletree in as a subtree of the root in place of the child.

DP then times all such trees, determines the fastest one, and enters that in its table.

Returns: List of records. Each record contains a ruletree and the measured time for that ruletree. Fastest ruletree is first entry in the list. The list has <nBest> many entries, from the fastest to the <nBest> fastest formulas found.

Search Options:

```
spiral> PrintSpecDPOptionsRecord();
rec(
  timeLimit := false | <minutes>,
  localUnrolling := true | false,
  localUnrollingMin := <positive int>,
  localUnrollingMax := <positive int>,
  globalUnrolling := true | false,
  globalUnrollingMin := <positive int>,
  globalUnrollingMax := <positive int>,
  bestFound := "save" | "none",
  nBest := <positive integer>,
  optimize := "minimize" | "maximize",
  hashTable := <hashTable>,
  verbosity := <non-negative integer>
);
```

Run “MergeDPOptionsRecord(rec());” to determine defaults.

An nBest of 1 is the standard DP. Increasing nBest causes DP to not only keep in its list the best implementation for each SPL and size, but the nBest such implementations.

By setting optimize to “maximize” it is possible to cause DP to maximize the measured event instead of the usual minimization.

DP uses a hash table to store its list of best implementations. It is possible to pass a hash table to DP for it to use and so that you can keep the resulting hash table when DP finishes. Note that any entries in the hashTable will be assumed to the best implementations by DP. You can create a new hashTable for use with DP by calling

"HashTableDP()".

Verbosity levels:

```
0 = Print nothing
1 = Show recursive calls to DP
2 = Show best trees found found each recursive call to DP
3 = Show how many trees must be fully expanded and how many were timed
4 = Show formulas that are being timed.
```

Examples:

```
spiral> DP( SPLNonTerminal("DCT2", 4) );
DP called on SPLNonTerminal( "DCT2", 4 )
1 tree(s) to fully expand
  DP called on SPLNonTerminal( "DCT2", 2 )
  1 tree(s) to fully expand
  1 tree(s) timed at this level
  Best Trees:
     DCT2(2)     {RuleDCT2_0} ! 67
  DP called on SPLNonTerminal( "DCT4", 2 )
  1 tree(s) to fully expand
  1 tree(s) timed at this level
  Best Trees:
     DCT4(2)     {RuleDCT4_0} ! 84
1 tree(s) timed at this level
Best Trees:
   DCT2(4)     {RuleDCT2_2}
    |--DCT2(2)     {RuleDCT2_0}
    `--DCT4(2)     {RuleDCT4_0} ! 124
3 total trees timed
[ rec(
      ruletree := RuleTree(
          RuleDCT2_2,
          SPLNonTerminal( "DCT2", 4 ), [
          RuleTree(RuleDCT2_0, SPLNonTerminal( "DCT2", 2 )),
          RuleTree(RuleDCT4_0, SPLNonTerminal( "DCT4", 2 ))
        ] ),
      measured := 124 ) ]
spiral> DP( SPLNonTerminal("DCT4", 4), rec(nBest:=2), rec() );
DP called on SPLNonTerminal( "DCT4", 4 )
10 tree(s) to fully expand
  DP called on SPLNonTerminal( "DCT2", 4 )
  1 tree(s) to fully expand
    DP called on SPLNonTerminal( "DCT2", 2 )
    1 tree(s) to fully expand
    1 tree(s) timed at this level
    Best Trees:
       DCT2(2)     {RuleDCT2_0} ! 67
    DP called on SPLNonTerminal( "DCT4", 2 )
    1 tree(s) to fully expand
    1 tree(s) timed at this level
    Best Trees:
       DCT4(2)     {RuleDCT4_0} ! 83
  1 tree(s) timed at this level
  Best Trees:
     DCT2(4)     {RuleDCT2_2}
      |--DCT2(2)     {RuleDCT2_0}
      `--DCT4(2)     {RuleDCT4_0} ! 124
  DP called on SPLNonTerminal( "DST2", 2 )
  1 tree(s) to fully expand
  1 tree(s) timed at this level
  Best Trees:
     DST2(2)     {RuleDST2_0} ! 65
  DP called on SPLNonTerminal( "DST4", 2 )
  1 tree(s) to fully expand
  1 tree(s) timed at this level
  Best Trees:
     DST4(2)     {RuleDST4_0} ! 83
  DP called on SPLNonTerminal( "DCT3", 4 )
  1 tree(s) to fully expand
```

```
    DP called on SPLNonTerminal( "DCT3", 2 )
    1 tree(s) to fully expand
    1 tree(s) timed at this level
    Best Trees:
       DCT3(2)     {RuleDCT2_0 ^ T} ! 73
  1 tree(s) timed at this level
  Best Trees:
     DCT3(4)     {RuleDCT2_2 ^ T}
      |--DCT3(2)     {RuleDCT2_0 ^ T}
      `--DCT4(2)     {RuleDCT4_0} ! 135
  DP called on SPLNonTerminal( "DST3", 2 )
  1 tree(s) to fully expand
  1 tree(s) timed at this level
  Best Trees:
     DST3(2)     {RuleDST2_0 ^ T} ! 70
10 tree(s) timed at this level
Best Trees:
   DCT4(4)     {RuleDCT4_5 ^ T} ! 162
   DCT4(4)     {RuleDCT4_3}
    |--DCT2(2)     {RuleDCT2_0}
    `--DST2(2)     {RuleDST2_0} ! 170
18 total trees timed
[ rec(
      ruletree := RuleTree(RuleDCT4_5, "T", SPLNonTerminal("DCT4", 4)),
      measured := 162 ), rec(
      ruletree := RuleTree(
          RuleDCT4_3,
          SPLNonTerminal( "DCT4", 4 ), [
          RuleTree(RuleDCT2_0, SPLNonTerminal( "DCT2", 2 )),
          RuleTree(RuleDST2_0, SPLNonTerminal( "DST2", 2 ))
        ] ),
      measured := 170 ) ]
spiral>
```

A.4.4 RandomSearch

```
Usage: RandomSearch( <spl>
          [, <RandomSearch-options-record>, <SPL-options-record> ] )
```

Note that if you wish to specify a RandomSearch-options-record or a SPL-options-record, then you must specify both. To leave one blank, just pass "rec()" in its place.

RandomSearch generates random ruletrees and times them, keeping track of the fastest one it has found so far. Technically, RandomSearch() is just a front in to STEER(), passing the correct options to only generate and time random formulas.

Returns: The fastest implementation found as an "individual" which is a record consisting of fields for a ruletree, SPLOptions, the measured event (usually runtime), and a few other fields not of importance.

SearchOptions:

```
spiral> PrintSpecRandomSearchOptionsRecord();
rec(
  numFormulas := <positive integer>
  seed        := <integer>
  timeLimit := false | <minutes>,
  localUnrolling := true | false,
  localUnrollingMin := <positive int>,
  localUnrollingMax := <positive int>,
  globalUnrolling := true | false,
  globalUnrollingMin := <positive int>,
  globalUnrollingMax := <positive int>,
  bestFound := "save" | "none",
  verbosity   := <non-negative integer>
);
```

Run "MergeRandomSearchOptionsRecord(rec());" to determine defaults.

numFormulas specifies how many random formulas to generate. Note that is possible that RandomSearch will not time this many formulas as it will not time the exact same formula twice.

seed specifies a random seed for use with the random number generator.

Verbosity levels:

```
0 = Print nothing
1 = Print final best
2 = Print formulas being timed
```

Example:

```
spiral> RandomSearch( SPLNonTerminal("DFT",16), rec(numFormulas:=10),
> rec(globalUnrolling:=8) );

Summary:
  Indiv 1: DFT(16)     {RuleDFT_1}
             |--DFT(4)     {RuleDFT_1 ^ T}
             |   |--DFT(2)     {RuleDFT_0}
             |   `--DFT(2)     {RuleDFT_0}
             `--DFT(4)     {RuleDFT_1}
                 |--DFT(2)     {RuleDFT_0}
                 `--DFT(2)     {RuleDFT_0} ! 2861

rec(
  IsIndiv := true,
  operations := IndivOps,
  ruletree := RuleTree(
      RuleDFT_1,
      SPLNonTerminal( "DFT", 16 ), [
      RuleTree(
        RuleDFT_1, "T",
        SPLNonTerminal( "DFT", 4 ), [
        RuleTree(RuleDFT_0, SPLNonTerminal( "DFT", 2 )),
          RuleTree(RuleDFT_0, SPLNonTerminal( "DFT", 2 ))
        ] ),
        RuleTree(
          RuleDFT_1,
          SPLNonTerminal( "DFT", 4 ), [
          RuleTree(RuleDFT_0, SPLNonTerminal( "DFT", 2 )),
          RuleTree(RuleDFT_0, SPLNonTerminal( "DFT", 2 ))
        ] )
      ] ),
    SPLOpts := rec(
        dataType := "complex",
        globalUnrolling := 8,
        language := "fortran",
        compflags := "'-O -fomit-frame-pointer -malign-double'" ),
    measured := 2861,
    fitness := 1/2861 )
  spiral>
```

A.4.5 HillClimb

```
Usage: HillClimb( <spl>
                  [, <HillClimb-options-record>, <SPL-options-record> ] )
```

Note that if you wish to specify a HillClimb-options-record or a SPL-options-record, then you must specify both. To leave one blank, just pass “rec()” in its place.

HillClimb performs a hill climbing search. It begins by generating a random implementation and timing it. Next, it applies a random mutation to generate a new implementation which is then timed. If this new implementation is faster, then a random mutation is applied to the new implementation and otherwise a random mutation is applied to the original implementation. Mutations are applied a specified number of times, searching for a fast implementation. Then the process is restarted with a new random implementation.

Returns: The fastest implementation found as an “individual” which is a record consisting of fields for a ruletree, SPLOptions, the measured event (usually runtime), and a few other fields not of importance.

SearchOptions:

```
spiral> PrintSpecHillClimbOptionsRecord();
rec(
  timeLimit := false | <minutes>,
  localUnrolling := true | false,
  localUnrollingMin := <positive int>,
  localUnrollingMax := <positive int>,
  globalUnrolling := true | false,
  globalUnrollingMin := <positive int>,
  globalUnrollingMax := <positive int>,
  bestFound := "save" | "none",
  numRestart  := <positive integer>
  quitRestart := <positive integer>
  numMutate   := <positive integer>
  quitMutate  := <positive integer>
  seed        := <integer>
  hashTable   := <hashTable>
  verbosity   := <non-negative integer>
);
```

Run “MergeHillClimbOptionsRecord(rec());” to determine defaults.

numRestart specifies the maximum number of times HillClimb will restart with a random implementation.

quitRestart specifies the maximum number of restarts without seeing an improvement in the best found implementation.

numMutate specifies the maximum number of mutates HillClimb will perform before restarting.

quitMutate specifies the maximum number of mutates (before restarting) without seeing an improvement in the current implementation.

seed specifies a random seed for use with the random number generator.

HillClimb keeps track of all the formulas it has timed so far (to avoid duplicating timings). This information is kept in a hash table. It is possible to pass a hash table to HillClimb for it to use and so that you can keep the resulting hash table when HillClimb finishes. Note that any entries in the hashTable will be assumed to have correct timings. You can create a new hashTable for use with HillClimb by calling HashTableHillClimb().

Verbosity levels:

```
0 = Print nothing
1 = Print final best
2 = Print restart count
3 = Print restart best found so far
4 = Print formulas being timed
```

Example:

```
spiral> HillClimb( Transform("DST2",8), rec(numRestart:=3), rec() );;
Restart 1
   10 mutations tried
   New Best Found:
      DST3(8)     {RuleDST2_2 ^ T}
       `--DCT3(8)     {RuleDCT2_2 ^ T}
           |--DCT3(4)     {RuleDCT2_2 ^ T}
           |   |--DCT3(2)     {RuleDCT2_0 ^ T}
           |   `--DCT4(2)     {RuleDCT4_0}
           `--DCT4(4)     {RuleDCT4_3 ^ T}
               |--DCT3(2)     {RuleDCT2_0 ^ T}
               `--DST3(2)     {RuleDST2_0 ^ T} ! 336
Restart 2
   15 mutations tried
   New Best Found:
         DST3(8)     {RuleDST2_3 ^ T}
          |--DST4(4)     {RuleDST4_1}
          |   `--DCT4(4)     {RuleDCT4_1}
          |       `--DCT2(4)     {RuleDCT2_2}
          |           |--DCT2(2)     {RuleDCT2_0}
          |           `--DCT4(2)     {RuleDCT4_0}
          `--DST3(4)     {RuleDST2_3 ^ T}
              |--DST4(2)     {RuleDST4_0}
              `--DST3(2)     {RuleDST2_0 ^ T} ! 307
   Restart 3
      11 mutations tried
      No better finds than previous 307

   7 total trees timed
   Best Found Implementation:
   DST3(8)     {RuleDST2_3 ^ T}
    |--DST4(4)     {RuleDST4_1}
    |   `--DCT4(4)     {RuleDCT4_1}
    |       `--DCT2(4)     {RuleDCT2_2}
    |           |--DCT2(2)     {RuleDCT2_0}
    |           `--DCT4(2)     {RuleDCT4_0}
     `--DST3(4)     {RuleDST2_3 ^ T}
        |--DST4(2)     {RuleDST4_0}
        `--DST3(2)     {RuleDST2_0 ^ T} ! 307
```

A.4.6 STEER

STEER stands for Split Tree Evolution for Efficient Runtimes

```
Usage: STEER( <spl> [, <STEER-options-record>, <SPL-options-record> ] )
```

Note that if you wish to specify a STEER-options-record or a SPL-options-record, then you must specify both. To leave one blank, just pass “rec()” in its place.

STEER is a stochastic evolutionary search algorithm for finding fast implementations. STEER is very similar to a genetic algorithm.

Returns: The fastest implementation found as an "individual" which is a record consisting of fields for a ruletree, SPLOptions, the measured event (usually runtime), and a few other fields not of importance.

SearchOptions:

```
spiral> PrintSpecSTEEROptionsRecord();
rec(
  timeLimit := false | <minutes>,
  localUnrolling := true | false,
  localUnrollingMin := <positive int>,
  localUnrollingMax := <positive int>,
  globalUnrolling := true | false,
  globalUnrollingMin := <positive int>,
  globalUnrollingMax := <positive int>,
  bestFound := "save" | "none",
  popSize  := <positive integer>
  numGens  := <positive integer>
  quitGen  := <positive integer>
  bestKept := <non-negative integer>
  crossed  := <non-negative integer>
  mutated  := <non-negative integer>
  injected := <non-negative integer>
  seed     := <integer>
  fitnessFun := ReciprocalFitness | MeasuredFitness
                                  | <user-specified-function>
  hashTable := <hashTable>,
  verbosity := <non-negative integer>
);
```

Run "MergeSTEEROptionsRecord(rec());" to determine defaults.

popSize specifies the size of the population. That is, popSize formulas are present in the population each generation.

numGens specifies the maximum number of generations that STEER will be allowed to run.

quitGen specifies that after the given number of generations without finding a faster formula than the current best, STEER should stop.

bestKept specifies the number of distinct fastest formulas to keep from generation to generation.

crossed specifies the number of formulas to cross-over each generation. Note that crossed/2 pairs of formulas are crossed-over.

mutated specifies the number of formulas to be mutated each generation.

injected specifies the number of new random formulas to be injected into the population each generation after the first.

Note that it must be that popSize >= bestKept + crossed + mutated + injected.

seed specifies a random seed for use with the random number generator.

fitnessFun specifies a function used to calculate the fitness of a formula given its measured event. The higher the fitness, the better the formula.

STEER keeps track of all the formulas it has timed so far (to avoid duplicating timings). This information is kept in a hash table. It is possible to pass a hash table to STEER for it to use and so that you can keep the resulting hash table when STEER finishes. Note that any entries in the hashTable will be assumed to have correct timings. You can create a new hashTable for use with STEER by calling HashTableSTEER().

Verbosity levels:

```
   0 = Print nothing
   1 = Print final best
```

```
2 = Print generation count
3 = Print population stats for each generation
4 = Print formulas being timed
```

Example:

```
spiral> STEER(  SPLNonTerminal("DCT1",4),
> rec( seed := 348, bestKept := 2, verbosity := 1 ), rec() );

Summary:
  Indiv 1: DCT1(4)     {RuleDCT1_3 ^ T}
           |--DCT1(2)     {RuleDCT1_3}
           |   |--DCT1(1)     {RuleDCT1_0}
           |   `--DCT1(1)     {RuleDCT1_0}
           `--DCT1(2)     {RuleDCT1_3}
               |--DCT1(1)     {RuleDCT1_0}
               `--DCT1(1)     {RuleDCT1_0} ! 149
  Indiv 2: DCT1(4)     {RuleDCT1_3}
           |--DCT1(2)     {RuleDCT1_3 ^ T}
           |   |--DCT1(1)     {RuleDCT1_0}
           |   `--DCT1(1)     {RuleDCT1_0}
           `--DCT1(2)     {RuleDCT1_3 ^ T}
               |--DCT1(1)     {RuleDCT1_0}
               `--DCT1(1)     {RuleDCT1_0} ! 153

rec(
  IsIndiv := true,
  operations := IndivOps,
  ruletree := RuleTree(
      RuleDCT1_3, "T",
      SPLNonTerminal( "DCT1", 4 ), [
      RuleTree(
        RuleDCT1_3,
        SPLNonTerminal( "DCT1", 2 ), [
        RuleTree(RuleDCT1_0, SPLNonTerminal( "DCT1", 1 )),
        RuleTree(RuleDCT1_0, SPLNonTerminal( "DCT1", 1 ))
        ] ),
        RuleTree(
          RuleDCT1_3,
          SPLNonTerminal( "DCT1", 2 ), [
          RuleTree(RuleDCT1_0, SPLNonTerminal( "DCT1", 1 )),
          RuleTree(RuleDCT1_0, SPLNonTerminal( "DCT1", 1 ))
        ] )
      ] ),
    SPLOpts := rec(
        dataType := "real",
        globalUnrolling := 32,
        language := "fortran",
        compflags := "'-O -fomit-frame-pointer -malign-double'" ),
    measured := 149,
    fitness := 1/149 )
  spiral>
```

A.4.7 TimedSearch

```
Usage: TimedSearch( <spl>
         [, <TimedSearch-options-record>, <SPL-options-record> ] )
```

Note that if you wish to specify a TimedSearch-options-record or a SPL-options-record, then you must specify both. To leave one blank, just pass “rec()” in its place.

TimedSearch is a meta-search algorithm; that is, it calls other search algorithms to do the real search. It runs for a specified length of time, limiting the called search algorithms to certain amounts of time. The idea is that this algorithm can be used to find the best ruletree possible in say 30 minutes (or any specified length of time).

Returns: Best Found Implementation (this is a record containing a ruletree, the SPL Options used, and the measured amount of time). Note that TimedSearch does a BestFoundLookup and returns the resulting implementation; so, it is possible for TimedSearch to return a ruletree that none of the called search algorithms timed during the execution of TimedSearch.

Search Options:

```
spiral> PrintSpecTimedSearchOptionsRecord();
rec(
  timeLimit := <minutes>,
  searches  := [ "<searchMethod1>", <SearchOpts1>, <timeLimit1>,
                 ...
                 "<searchMethodN>", <SearchOptsN>, <timeLimitN> ]
  verbosity := <non-negative integer>
);
```

Run “MergeTimedSearchOptionsRecord(rec());” to determine defaults.

timeLimit specifies the overall time limit for the entire search. This can not be set to false as in the other search algorithms.

Setting searches determines which search algorithms are called from TimedSearch, with which search options, and for how long. Each search method is called in turn, passed its respective search options. Each search method has its time limit set to be the minimum of timeLimitK and of the remaining global time.

searchMethodK must be a string and the name of a valid search algorithm.
searchOptsK must be a valid search options record for searchMethodK.
timeLimitK must be an integer representing the maximum time in minutes, or false if searchMethodK is to be allowed to run up to the maximum global amount of remaining time.

The default is to run RandomSearch on 10 formulas, then to run DP, next STEER, and then to run 4-best DP over localUnrolling and finally STEER over localUnrolling with a larger population (see config.g).

Verbosity levels:

```
0 = Print nothing, tell search algorithms to print nothing
1 = Print nothing
```

```
   2 = Print final best
   3 = Print search algorithms being called
```

A.5 BestFoundTable

The BestFoundTable keeps track of the best found implementations for the different SPLs. The search algorithms will save the best implementations that they find to the BestFoundTable. Currently, none of the search algorithms use the table to guide their search.

You can interact with the BestFoundTable using the functions “BestFoundLookup”, “BestFoundSave”, and “BestFoundRead”.

A.5.1 BestFoundLookup

```
Usage: BestFoundLookup( <spl> [, <SPL-options-record>] )
```

Looks up the best found implementation for the given spl.

Returns: a list of BestFoundImpl records consisting of the ruletree, the actual full SPLOptions, and the measured time.

If no SPL-options-record is passed, then BestFoundLookup returns a list of the (equally) fastest implementations across all possible SPL Options (such as language or dataType). That is, if there is a BestFound Implementation for the given spl in both C and Fortran, it will only return the one that is fastest, unless they are equally fast in which case both will be returned.

If an SPL-options-record IS passed, then BestFoundLookup returns a list of the (equally) fastest implementations across only those implementations having the same values for the SPL Options listed in the variable BestFoundDifferFields. By default, BestFoundDifferFields includes the fields dataType and language.

So, if you want to get the BestFound Implementation for a DFT of size 16 implemented in C (but irrespective of the used dataType), try:

```
   BestFoundLookup( Transform("DFT",16), rec(language:="c") );
```

A.5.2 BestFoundSave

```
Usage: BestFoundSave( <filename> )
```

Saves the BestFoundTable to the specified filename (given as a string). This allows later retrieval of the Table.

A.5.3 BestFoundRead

```
Usage: BestFoundRead( <filename> )
```

Reads the BestFoundTable in the specified filename (given as a string). This overwrites (without saving) the current BestFoundTable. In case of error in reading the file, the current BestFoundTable is not overwritten.

A.6 HashTables

Hash tables are efficient ways to store certain types of data. They are used in several places in the search module. For example, the BestFoundTable is implemented as a HashTable with particular wrapper functions. Also, DP and STEER use HashTables to store partial results. These hash tables can be saved to files and later reused to avoid duplicating work already done by the search algorithms.

In particular, DP uses a HashTable to store the nBest ruletrees that it has found for a given spl. STEER uses a HashTable to store entire implementations to avoid re-running the same implementation multiple times.

The most common reason why you would want to use HashTables is to save DP's partial results. This is particularly useful, if, for example, you run DP on a transform of a particular size, but later may want to run DP on a larger size of that transform. By saving off DP's HashTable and then later reusing it, DP can avoid duplicating its earlier work. An example is shown in the section "HashTable Example".

The following sections describe how to save, restore, and create HashTables as well as give an example (see "HashSave", "HashRead", "HashTable Creation", and "HashTable Example").

A.6.1 HashSave

`Usage: HashSave( <HashTable>, <filename> )`

Saves the given HashTable to the specified filename (given as a string). This allows later retrieval of the HashTable.

A.6.2 HashRead

`Usage: HashRead( <filename> )`

Returns the HashTable that was stored in the specified file (filename should be passed as a string).

A.6.3 HashTable Creation

HashTableDP() creates a HashTable for use with DP.
HashTableSTEER() creates a HashTable for use with STEER.

A.6.4 HashTable Example

```
  spiral> myHashTable := HashTableDP();
```

```
HashTable
spiral> DP( Transform("DFT",32), rec(hashTable:=myHashTable), rec() );
...
spiral> HashSave( myHashTable, "DP_DFT32.hash" );
spiral> quit;
$
...
spiral> myHashTable := HashRead( "DP_DFT32.hash" );
HashTable
spiral> DP( Transform("DFT",1024), rec(hashTable:=myHashTable), rec() );
...
```