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

Bryan W. Singer

December 13, 2001

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 $Y = AX$, 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 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.

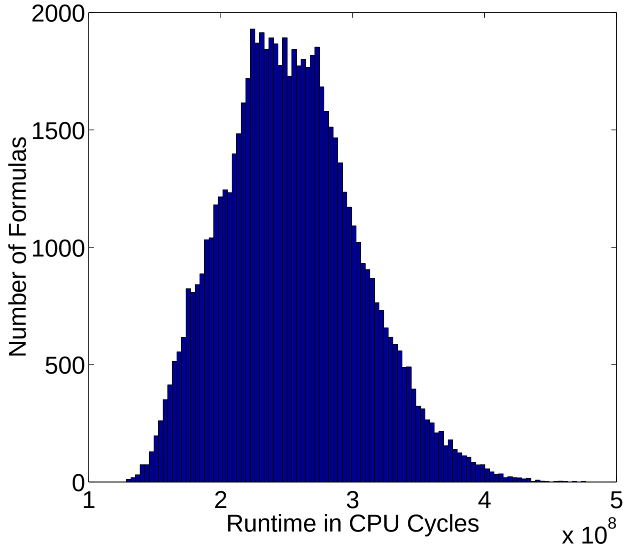

Figure 1: Histogram of runtimes for 70,376 different $FFT(2^{18})$ formulas.

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. 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: 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 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.

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 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)

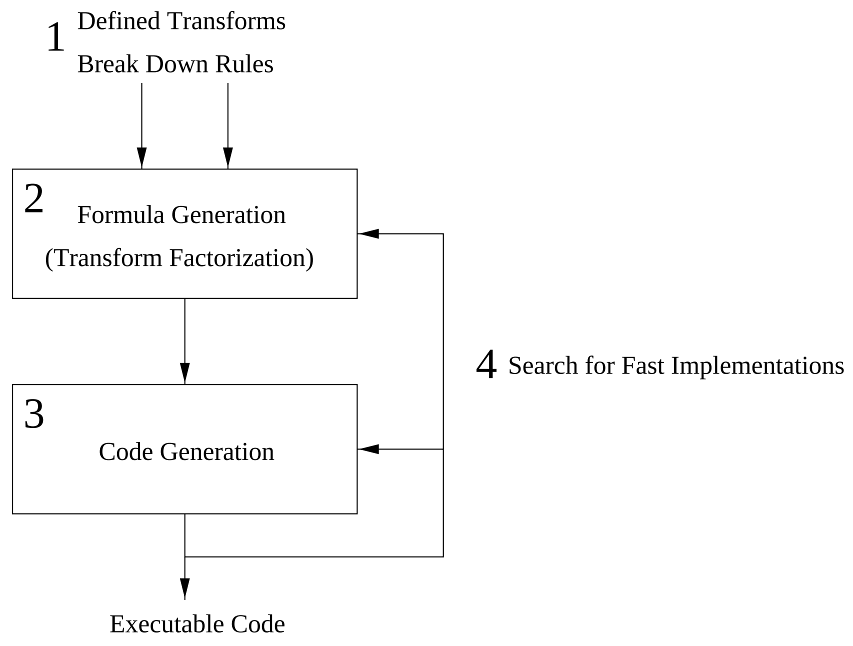

Figure 2: Overview of the SPIRAL system.

4. **Optimization.** The final step is to search for a fast implementation and one of the main focuses of this thesis.

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.

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 up to 20% faster than the standard dynamic programming search method.

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.

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.

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 generate 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.

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.

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.

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/`.

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

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.

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

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.

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

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