A SEARCH OPTIMIZATION IN FFTW

by

Liang Gu

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Winter 2010

© 2010 Liang Gu
All Rights Reserved
A SEARCH OPTIMIZATION IN FFTW

by

Liang Gu

Approved: _______________________________________
Xiaoming Li, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _______________________________________
Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _______________________________________
Michael J. Chajes, Ph.D.
Dean of the College of Engineering

Approved: _______________________________________
Debra Hess Norris, M.S.
Vice Provost for Graduate and Professional Education
ACKNOWLEDGEMENTS

First, I want to thank my advisor, Dr. Xiaoming Li, for his guidance during this research. Without his strong support and encouragement, this work would not be possible. My lab mates, Jakob Siegel and Murat Bolat also helped me a lot during the writing of this thesis. I would like to express my gratitude to them and all others who helped me. Last but not least, special thanks and blessing go to my wife, Jiani Guan, my parents, my brother and other friends. I am heartily grateful to all your support and care for me. Thank you all.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>vii</td>
</tr>
</tbody>
</table>

## Chapter

1 **A BACKGROUND OF FAST FOURIER TRANSFORM AND ITS LIBRARIES** ........................................... 1

1.1 Discrete Fourier Transform and Fast Fourier Transform ............... 1
1.2 Adaptive Fast Fourier Transform Libraries ........................................... 2

2 **ANALYTICAL MODEL IN ADAPTIVE LIBRARIES** .......................... 6

2.1 Previous Works of Performance Model .................................. 6
2.2 Challenge in Building Models for DFT .................................. 8

3 **FFTW FRAMEWORK** ........................................ 10

3.1 Codelets in FFTW ........................................ 10
3.2 Other Algorithms in FFTW ........................................ 11

4 **PERFORMANCE MODELS FOR DFT ALGORITHMS** ......................... 13

4.1 Target of Performance Model ........................................ 13
4.2 Explicit Abstract Machine Model .................................... 15
4.3 Implicit Abstract Machine Model .................................... 17
4.4 Segmented Memory Access Cost ...................................... 18
4.5 Model for Generic DFT Algorithm ................................... 20
4.6 Prediction Results of Individual DFT algorithms ................... 22
# PERFORMANCE MODELS FOR CODELETS

5.1 Scalar Codelets Model ........................................ 24
5.2 SIMD Codelets Model ........................................ 26
5.3 Prediction results of Codelet Model ......................... 28

# MODEL TRAINING AND EVALUATION

6.1 Training of Performance Models .............................. 30
6.2 Evaluation Environment Setup ................................ 31
6.3 Overall Evaluation Results Comparison ..................... 33

6.3.1 Runtime Comparison ....................................... 33
6.3.2 Search Time Comparison .................................... 35

# CONCLUSION AND FUTURE WORKS

7 CONCLUSION AND FUTURE WORKS ......................... 38

BIBLIOGRAPHY .................................................. 40
LIST OF FIGURES

3.1 FFTW installation and runtime ........................................... 12
4.1 An FFTW plan in tree structure ......................................... 13
4.2 One decomposition step of DFT 35 using Cooley-Tukey ............. 14
4.3 Memory access profile of Bluestein on Intel Xeon ................... 19
4.4 Performance profile of Generic DFT on Intel Xeon ................. 21
4.5 Prediction result of individual DFT algorithms on Intel Xeon ... 23
5.1 Runtime of n25 with different strides on Intel Xeon ............... 25
5.2 Runtime of n1fv25 with different strides on Intel Xeon .......... 28
5.3 Prediction of n25 and n1fv25 with different strides on Intel Xeon 29
6.1 Scalar runtime comparison among Exhausive, Estimate and Model-driven mode ...................................................... 34
6.2 SIMD runtime comparison among Exhausive, Estimate and Model-driven mode ..................................................... 35
6.3 Search time comparison among Exhausive, Estimate and Model-driven mode ..................................................... 36
ABSTRACT

Generating high performance fast Fourier transform (FFT) libraries for different computer architectures is an important task. Architecture vendors sometimes have to rely on dedicated experts to tune FFT implementation on each new platform. Fastest Fourier transform in the West (FFTW) replaces this tedious and repeated work with an adaptive FFT library. It automatically generates FFT code that are comparable to libraries provided by vendors. Part of its success is due to its highly efficient straight-line style code for small DFTs, called codelets. The other part of its success is the result of a large and carefully chosen search space of FFT algorithms. FFTW mainly traverses this space by empirical search, otherwise a simple heuristic is used. However, both methods have their downside. The empirical search method spends a lot of search time on large DFT problems and the simple heuristic often delivers implementation that is much worse than optimum. An ideal approach should find a reasonably good implementation within the FFT search space in a small amount of time.

Model-driven optimization is often believed to be inferior to empirical search. It is very hard to capture all the performance features of an adaptive library on many modern architectures. No one has implemented an adaptive performance model to automatically assist the search of FFT algorithms on multiple architectures. This thesis presents an implicit abstract machine model and a codelet performance model that can be used in the current FFTW framework. With the performance prediction given by these models, the empirical search engine of FFTW can be replaced without serious hurt of performance. This technique also helps to break down the runtime
in FFTW and explain the performance of complex combination of DFT algorithms. As a good trade-off, this optimization achieves 95% of the performance of FFTW empirical search and uses less than 5% of the search overhead on four test platforms.
Chapter 1

A BACKGROUND OF FAST FOURIER TRANSFORM AND ITS LIBRARIES

1.1 Discrete Fourier Transform and Fast Fourier Transform

Fourier transform (FT), both analog and digital, is one of the most important tools in signal processing. It transforms a function or input set in time domain to frequency domain and the inverse transform does the opposite. Due to the advantage of digital system over analog, Discrete Fourier Transform (DFT) and its fast implementation, fast Fourier transform (FFT) has become more and more important ever since the 1960s. Their implementation covers data compression and spectrum analysis and processing for audio signal, image/video signal and other multi-dimensional signals. The definition of DFT is shown in equation 1.1

\[ X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i kn/N} \quad k = 0, 1, ..., N - 1 \]  

(1.1)

The definition of inverse DFT (IDFT) is shown in equation 1.2

\[ X_n = (1/N) \sum_{k=0}^{N-1} x_n e^{2\pi i kn/N} \quad n = 0, 1, ..., N - 1 \]  

(1.2)

Both definition involve \(O(N)\) complex number computation for each output data. The overall computation for size N DFT or IDFT is \(O(N^2)\).

Fast Fourier transforms, on the other hand, is a group of fast DFT algorithms with complexity of \(O(N\log(N))\). Some FFT algorithms rely on divide and conquer method, like Cooley-Tukey algorithm [1] that can break down composite
size $N = N_1 \times N_2$ recursively. Other algorithms [2] can decompose the problem size once, such as Prime-Factor(Good-Thomas) [3] decomposing size $N = N_1 \times N_2$ where $N_1$ and $N_2$ are co-prime. Rader’s algorithm [4] and Bluestein’s algorithm [5] can re-express a prime-size DFT as convolutions. In practice, each of the above algorithm may have versions with different implementation details. Cooley-Tukey algorithm has decimate in time (DIT), decimate in frequency (DIF), radix-2, mixed-radix, split-radix [6] cases. Moreover, these algorithms can be applied in a combined way, therefore greatly increasing the possible choices of algorithm in solving a DFT problem. Because of the large number of available FFT algorithms, the question of choosing to implement which one or ones of the above will achieve the most performance on different platforms becomes a real challenge.

1.2 Adaptive Fast Fourier Transform Libraries

Since FFT is such a widely used signal processing library for both scientists and engineers, building an efficient library on computers becomes an important task. However, modern processors have quite different execution models and their features change rapidly over time. On the other hand, there are a considerable numbers of FFT algorithms available and they have even more versions with different implementation detail. Finding the right FFT algorithm and the efficient implementation on a particular architecture is not an easy job. Traditionally, vendors of a new processor sometimes provide a hand-tuned FFT library. These native libraries often approach the peak performance of that machine. But performance comes from hard work and some detailed knowledge of the processor. Once a new processor comes out, the hand-tuning process has to be done again.

In order to avoid such kind of repeated tuning work, some adaptive FFT library has been developed over the past few years. Among all adaptive DFT libraries, SPIRAL [7] and FFTW [8, 9] are the most popular ones. SPIRAL is a software/hardware generation tool for DSP algorithms and FFTW is a dedicated
library for DFT. SPIRAL has two levels of optimization, i.e. algorithm level and implementation level. Algorithm level optimization takes user specified transforms as input, generates formula and optimizes them. Optimized formula coded in Signal Processing Language(SPL) is passed on to implementation level optimizer. Implementation details are decided and some common compiler techniques are applied on this level. In the following evaluation phase, the generated source code is compiled, executed and evaluated. The evaluation is feedback to control the formula generator and code optimization. Dynamic programming or evolution algorithms are used to reach the optimal on available control parameters. The search process of SPIRAL is crucial to its performance but also extremely time consuming. It may take hours for a work station to generate a DFT implementation for the size of several thousand. There is a present effort [10] to reduce the search time by building analytical models and guiding the search process. However, the model relies heavily on a complicated probability based cache model and requires a lot of detail about the platform’s memory system.

FFTW on the other hand is dedicated to solve DFT problems of any dimension and size on a wide range of platforms in C language. FFTW has a code generation phase and a runtime phase. In its code generation phase, some highly optimized C code, called codelet, is generated for small size DFT problems. In its runtime phase, other architectural decomposition FFT algorithms are available. A DFT solution plan is an architectural decomposition scheme of codelet and FFT algorithm combination. Due to the large number of different FFT algorithms and all sizes of codelet, various choices are available at almost each level of problem decomposition. A similar dynamic search and execution approach is applied here to automatically find the best solution. This solution found is usually a reasonably good one, however, it may not be the optimal because the underlying assumption of dynamic programming is not true. The overall performance of FFTW relies heavily
on the efficiency of its codelet, which is generated using an unique FFT compiler, called genfft [11]. Genfft is written in Ocaml(Objective Caml), a dialect of static type metal language. On the code generation phase, a couple of FFT algorithms are available and one of them is chosen for each codelet size, depending on which one is fast on most computers. Some general compiler techniques and some unique ones are applied on the acyclic graph before it is unparsed to output C code. A considerable amount of knowledge of FFT is used in this framework, therefore FFTW is less likely to be extended to other signal processing problems like SPIRAL is.

Both FFTW and SPIRAL have much better performance than other older FFT libraries and vendor provided hand-tuned FFT libraries. Both of them supports single instruction multiple data(SIMD) extensions, such as SSE and Altivec, as well as IBM Cell processor [12, 9]. Both libraries keep updating their latest versions and they are similarly fast for the time being. Specifically, for scalar implementation FFTW is faster than SPIRAL and parallel implementation SPIRAL is slightly better than FFTW. However, besides microprocessor, SPIRAL also provides code generation in Verilog on FPGA and other platforms. FFTW, on the other hand, supports DFT problem of any dimension, size with real/complex and odd/even input data. Both SPIRAL and FFTW have the possibility to discover new FFT algorithms and particularly the author of FFTW has published a new split-radix algorithm [13] that reduces the number of arithmetic operations on top of best previous work.

One of the problems these two library share is that the search overhead in finding the best solution. Because SPIRAL has feedback on both algorithm level and implementation level search, its search process becomes extremely time consuming even on a powerful work station. The work in [10] can reduce the search time considerably but it is verified only on one Intel machine and is largely limited by its requirement of detailed memory system knowledge. This problem is less sever for
small problem sizes on FFTW because of some heuristic methods are incorporated in FFTW to bring down the exhaustive search time. However these heuristics seriously compromise DFT performance and the search overhead is still large for large problem sizes. This thesis is the first effort that addresses this problem using an automatically adaptive method on multiple platforms.
Chapter 2

ANALYTICAL MODEL IN ADAPTIVE LIBRARIES

An adaptive library usually applies application-specific knowledge to define a search space and chooses the best performing version out of it. The most simple and used evaluation method is to execute the code and time it. However, this straightforward method can be extremely time consuming. In a sophisticated signal processing library, the search space is usually maximized in order to cover the optimal solution. This huge search space directly worsen the situation. The authors of these libraries come up with all kinds of ways to mitigate the problem. Dynamic programming is often used to reduce redundant measurements and total number of possibilities. It assumes any local optimum should also be part of the global optimum solution, which is not true in all cases. This assumption is a typical trade-off between absolute optimum and search overhead.

2.1 Previous Works of Performance Model

Auto-tuning is a very popular trend in library generation and there is often some kind of analytical model behind these tuning tools. [14] has built an analytical model to auto select the optimal parameters for ATLAS without executing different versions of the program. Not only for programs on CPU, people have extend auto-tuning to other platforms. [15] has recently used a simple model to speed up FFT programs on GPU by tuning radix sequence to choose and number of threads to use. Such a simple tuning model involves only a couple of key parameters in the implementation, such as, loop unrolling times, tile sizes etc. However more
complicated models can be used to predict the run time of a program directly, like [10].

Table 2.1: Classification of performance tuning models

<table>
<thead>
<tr>
<th>Model Feature</th>
<th>Input</th>
<th>Need training</th>
<th>Tuning knob</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>include architecture features</td>
<td>Yes</td>
<td>limited number of parameter values</td>
<td>execution time</td>
</tr>
<tr>
<td></td>
<td>no architecture features</td>
<td>No</td>
<td>more complicated</td>
<td>non execution time</td>
</tr>
</tbody>
</table>

These performance models can be classified by different features as shown in table 2.1. The input of a model can include architecture features or not. Including features, like cache size, instruction delay etc. as input definitely helps to predict the performance of the tuned program. However, on the other hand, doing so makes the tuning model less automatic largely decreasing the portability. Modern architecture of microprocessors is considerably complex, especially cache structure and instruction schedule. Therefore a lot of information and expertise is needed to provide enough details of the platform for the model. For example, [10] needs to know not only the total size, line size, associativity of the cache but also some of the physical layout of cache to assist the modeling.

If little or no architecture feature is provided, the performance model will probably need a training process. In this case, a couple of sample programs are executed on the targeted platform. Execution time is often fed back to the model to adjust the model parameters. The model and training algorithms should be designed to quickly determine its parameters without too many training steps. Usually the more tuning knob you have, you are more likely to achieve a better performance. However more tuning parameters also means increasing complexity of the search space and complexity of your model that explores it. This feature of the tuning model is largely decided by the nature of the problem and the size of the search space.
Typical tuning parameters include loop unrolling factor and tile size. However for the search space of FFTW, the situation is very different and much more complex as I will show in the next section. The output of the a performance model can be the execution time of different program versions and let the search engine to pick the minimum as it walks through the search space. The other kind of output is the optimal parameters within the search space, which is usually the case of simple search space and few tuning knobs.

2.2 Challenge in Building Models for DFT

Building performance models for DFT is very different from other models for much simpler applications, like matrix multiplication. There are primarily three challenges. The first is the large available algorithms of FFT. Each of these FFT algorithm has a couple of different versions with unique implementation details. On one hand, this makes the search space really huge for large DFT problems. While, on the other hand, this means the performance model must be general enough to be able to predict the performance of these algorithms. The second challenge is the complexity of each algorithm. An FFT program, especially the data access pattern, is much more complex than that of ATLAS. A direct result of this complexity is the interaction between program and microprocessor becomes hard to predict. Lastly, a optimal solution to a DFT problem is often a combination of different algorithms. Therefore, it is not enough to provide a couple of key parameter as the output. Instead, a combination of different algorithms in a well structured way is desired as a solution.

To meet this challenge, I have built a general DFT performance model to predict the runtime of some high level FFT algorithms and a different model for low level codelets in FFTW. The search engine recursively uses these models and provides a close-to-optimal algorithm structure for any size of DFT. A training process is necessary for the models to get parameters adapting to the underlying platform.
The training needs to be done for once taking only a couple of seconds typically. To provide as much portability as possible, I do not include any architecture parameters as input. However, there is an assumption that the underlying platform has two-level-cache structure, which is true for most modern architectures. Details of these models will be described in the following chapters.
Chapter 3

FFTW FRAMEWORK

FFTW is one of the most widely used DFT libraries today. It is written in portable C language and is adaptive to most hardware architectures and operating systems. FFTW computes DFT of all sizes and dimensions in $O(n\log(n))$ time. It supports input/output data with constant access strides and real/symmetric features, known as discrete cosine/sine transform. The user interface of FFTW has two stages. First, a user specify a DFT problem size to transform and FFTW will find a optimal solution for that size. Then the user can execute that plan as many times as he wants on different data input. Obviously, the first step is more interesting and an empirical search takes place here. Two groups of DFT implementations are available during this search. One is high level decomposition algorithms and the other is low level codelets for small size DFTs.

3.1 Codelets in FFTW

Codelets are important to the performance of FFTW because most of the computation is done by them at the low level after decomposed by other FFT algorithms. Codelets are dedicated to solve some small size DFT problems and are generally not architecture dependent (except SIMD codelets). Codelets are generated in four steps: creation, simplification, scheduling and un-parsing. For each small DFT size (less than 64), Genfft chooses one of the following algorithms: Cooley-Tukey, prime-factor, split-radix, Rader and Generic DFT algorithm depending on which one if more likely to be fast on most architectures. Generic DFT algorithm
follows the definition of DFT which has a complexity of $O(n^2)$, but it still can be efficient for some really small DFT sizes. The algorithm written in OCaml is expressed in the form of a directed acyclic graph (DAG). Then some common compiler techniques together with some DFT specific optimization, such as making constants positive and network transposition [11], are applied in the simplification phase. In the scheduler, the simplified DAG is scheduled using a cache-oblivious algorithm therefore minimizing the register pressure asymptotically. Then the OCaml code is unparsed to C code. Un-looping a small DFT implementation in straight-line code makes many optimizations and optimal schedule possible. Since Cooley-Tukey algorithm is frequently used in DFT problem decomposition and a DFT of size $r \times m$ will be completed in three steps: computing $m$ DFTs of size $r$; multiplying the output by twiddle factors; computing $r$ DFTs of size $m$. Therefore, besides these direct codelets designed for single small DFTs, each of them has a counterpart combining twiddle factor multiplication into it. Those codelets are designed for Cooley-Tukey algorithm by combining two of the three steps for efficiency. Other flavors of codelets exist to especially take advantage of real input or architecture specific SIMD extensions.

### 3.2 Other Algorithms in FFTW

FFTW has a installation phase and runtime phase, as shown in figure 3.1. Codelets are generated before installation and users are not required to generate more unless they want to. Codelets and other algorithms in plain C are provided in FFTW download package. During the runtime, both codelets and other algorithms are used together to maximize the possible search space.

All algorithms implemented in Codelets, except for prime factor algorithm, are also included at high level to decompose a large DFT problem into smaller ones. Prime factor algorithm [2] solves DFT of size $N = N_1 \times N_2$, where $N_1$ and $N_2$ are coprime. Compared with Cooley-Tukey, prime factor eliminates the twiddle factor
multiplication at the cost of re-index the data. I actually tried to add prime factor algorithm to FFTW but its performance is always worse than Cooley-Tukey. For modern architectures, data movement, especially complex re-index, is much more costly than multiplication.

Other algorithms implemented on the hand-coded high level includes Bluestein and other flavors of Cooley-Tukey. The main purpose of these hand-coded algorithm is to decompose a larger (can be prime) DFT problem into smaller (or composite) ones. Usually a byproduct of the decomposition, like twiddle factor multiplication may be computed on the high level. While the small DFT problems will be recursively decomposed until small enough for codelets to solve. Each algorithm has a couple of variants, referred as solvers, with different implementation details, like decimation in time (DIT) or decimation in frequency (DIF). There are solvers that can downgrade a high dimensional to looped problem or perform extra copy and buffer in solving a DFT problem. Generic DFT algorithm is also included in the choices but it is limited to be within the size of 173 due to its quadratic complexity.
Chapter 4

PERFORMANCE MODELS FOR DFT ALGORITHMS

As I have shown before, FFTW’s major computation is done in two parts: DFT algorithms and codelets. Prediction of the two is a bit different and will be addressed separately. In this chapter I will focus on DFT algorithms’ performance modeling first.

4.1 Target of Performance Model

FFTW generally decompose a DFT problem recursively using available FFT algorithms in a well structured hierarchy, called plan. A plan is generated when a user specifies the DFT problem size and FFTW is done with its search process. A plan can be saved as what they call wisdom and they are based on runtime test of the system. Plans can be reused when user specifies the local path to the wisdom file. Figure 4.1 is an example of real DFT plan generated by FFTW. Instead of showing the plan in a list of FFTW’s highly encrypted descriptions, I transformed it into a clear decomposition tree. DFT problem of size 113 can be solved by Rader
algorithm, which is dedicated to solving prime size DFTs. The child level problems of Rader are three DFTs of size 112 and a small extra computation kernel. The first two DFT of size 112 are identical while the third one should be computed in-place unlike the other two. Each of them will be solved by a twiddle codelet of size 7 in a Cooley-Tukey way and eventually finished by a direct codelet of size 16. To be clear, there are many different algorithms to choose from at each problem node, therefore the whole search space is considerably large. However if I can recursively model the performance of codelets and the small computation kernel in algorithms like Rader and Cooley-Tukey, I can model the whole runtime.

Therefore, the target of our performance model for DFT algorithms is primarily the small computation kernel left when generating child problems. In the case of Rader, it re-indexes a prime size DFT into a cyclic convolution, which can be solved by two forward DFT, array multiplication and an inverse DFT. The computation kernel in this case is the array multiplication of the two child DFT outputs. Another example is the well known Cooley-Tukey algorithm. Figure 4.2 shows the decomposition of DFT size 35. It can be solved by computing 5 DFTs of size 7 first, then multiplying twiddle factors and performing 7 DFTs of size 5 at the end. Excluding child problems, twiddle multiplication therefore becomes the target of this performance model. All child problems will be modeled recursively either by this DFT algorithm model or by codelet model at the end.

Besides FFT kernels, another case needs to be covered is the Generic DFT algorithm because there is not child problem and it solve the problem all in one
step. The performance model need to predict the performance of Generic DFT as a whole at least for DFT size less than 173.

4.2 Explicit Abstract Machine Model

It is widely believed that program runtime prediction is extremely hard if not impossible at all. Part of the reason is the complexity of application itself and compiler techniques that applied before runtime. The other part is the complexity of modern architectures that the program runs on. The amount of hardware resources and cache system will determine the actual performance in an non-obvious way. However there are still efforts trying to model the performance of some particular programs. The performance of some benchmarks was modeled in [16] by using an abstract machine model. For the computation of dense-matrix [17], models of memory hierarchy were combined with empirical search to improve performance. More recently, a highly effective model-driven optimization engine [14] was developed for ATLAS to predict the relative performance between code versions that have different transformation parameters. These previous work has inspired me to propose an adaptive performance model for FFT algorithms.

I start with the modeling of FFT algorithm kernels and Generic DFT algorithm by assuming an ideal machine environment where all data is in L1 cache and there is no pipeline and overlap of computation. Large memory access delay is eliminated if all data is in L1 and memory access time is unified to be L1 access time. Assuming no pipeline and overlap of different operations means all computations are executed one by one in a serialized way. These two assumptions follow the work of Saavddral, et.al in [16], where an abstract machine model based on Fortran was used to predict the performance of some benchmarks. With these assumption, the runtime of any simple program kernel is a weighted linear combination of instructions as shown in equation 4.1. By simple program, I mean program segments that are similar to basic blocks or loops of basic blocks. There is no function call,
not many branches except for branch conditions, which is exactly the case of FFT
algorithm kernels in FFTW. Consequently, unlike a common purpose compiler, the
DFT performance model does not need to consider complicated function evocation
or a lot of branch prediction.

\[ T_{A,M} = \sum_{i=1}^{n} C_{A,i} P_{M,i} = \vec{C}_A \vec{P}_M. \]  (4.1)

\( T_{A,M} \) is the runtime of algorithm \( A \) on machine \( M \). Suppose there are \( n \) types of
instruction on machine \( M \), \( C_{A,i} \) is the count of instruction type \( i \) in algorithm \( A \)
and \( P_{M,i} \) is the performance cost of that instruction on machine \( M \). If the equation
is expressed in a vector form, \( \vec{C}_A \) is the program characteristic vector and \( \vec{P}_M \) is the
instruction cost vector on machine \( M \).

Theoretically, the instruction set of size \( n \) depends on the instruction set
architecture (ISA) of the local processor and are quite different from one platform
to another. Therefore a complete program character vector \( \vec{C}_A \) includes all instruc-
tion types that appear in the assembly which is not available before compile time.
Even after compiling, the assembly of the program is machine dependent and using
them for adaptive performance modeling across different architectures is impossi-
ble. However despite of the difference in ISAs, a majority of them actually share
some similar basic instructions like load, store, multiplication and addition with
only minor variance. More importantly, the most time consuming instructions are
floating point instructions and memory related operations if there are few branches.
So instead of trying to touch every single instruction, I focus on some of the most
time consuming instructions and they can be estimated by studying the source code
directly.

In the DFT performance model, vector \( \vec{C}_A \) is obtained from a static analysis
of the source code. Special care needs to be taken when calculating the number
of different operations in source code because the local compiler will optimize the
program considerably. The \( \vec{C}_A \) is only an estimation of the number of operations
in a compiled program. Number of floating point operations is reduced if some compiler technologies like, common subexpression elimination, strength reduction are taken into account. In FFTW source code, part of this performance vector estimation is already provided. FFTW intends to use this information to compute a total instruction count so as to assist a fast while naive search strategy, called Estimate, as I will explain in later chapters. $\vec{P}_M$, on the other hand, corresponds to the latency of each instruction and can be obtained either from processor manual or measured by a set of micro-benchmarks.

4.3 Implicit Abstract Machine Model

The model discussed above is an explicit model because each instruction clearly has its own cost and the overall runtime is the sum of them. However, a real modern architecture has instruction mix and pipelining and the runtime of a batch of instructions is usually less than the sum of their individual runtime. For example, floating point division is an expensive operation which may take 40 cycles according to the processor manual. In one case, the data dependence may force other instructions to wait for the result of these division, it may take about 40 cycles on average to complete each division. While in another case, parallelism is available for other instructions to overlap with division, the actual cost can be much smaller. That is to say, the values of performance vector $\vec{P}_M$ can not be treated as global constant if instruction parallelism is taken into account.

The question immediately follows is how can we get the value of the performance vector $\vec{P}_M$ if it is not global and is different from the processor manual. A short answer is training and regression. For each DFT algorithm kernel, I measure the runtime of a couple of different DFT sizes and use the runtime to retrieve the performance vector using a linear regression method. Because there are $n$ performance variable $P_{M,i}$ in equation 4.1, $n$ training point is needed at least. To make the
solution more stable, I measure more than \( n \) DFT points and it becomes an over-determined system. Linear least square method is used to compute the performance vector as I will show in more detail later.

Some times, equation 4.1 needs less than \( n \) measurements to solve, because a couple of instruction counts may be dependent on each other making it impossible to solve them separately.

\[
5N \times \text{mult} + 10N \times \text{add} = 5N \times \text{fused mult add} \tag{4.2}
\]

For example, one part of an FFT kernel of size \( N \) may have \( 5N \) multiplications and \( 20N \) additions. It is impossible to solve for the runtime of multiplication and additions separately from runtime. Instead, we can get the fused single runtime of one multiplication and two addition as shown in equation 4.2.

By localizing the performance cost vector for each DFT algorithms kernels, the performance model is turned into an implicit model. The cost of each instruction is no longer the exact value shown in the processor manual, but it becomes a combined cost with other instructions after being compiled and scheduled uniquely. This performance vector will take resource stalls (other than memory stall) and instruction level parallelism (ILP) into consideration. Memory stalls will be discussed in the next section.

### 4.4 Segmented Memory Access Cost

With the increase of DFT size, the cost of all other fused instructions in implicit abstract machine model increase linearly with the number of such instructions except memory accesses. Modern microprocessors has typically a two-level cache structure, i.e. L1 cache and L2 cache besides main memory. Most FFTW input are continuous (stride equals to one) and at the top of FFT decomposition data strides are small. Consequently, DFT problems with small data access amount will fall in L1 or L2 because of good data spacial locality and has a small access cost
on average. With the increase of DFT problem size, data accessed will have larger portion of cache misses and larger cost for each memory access. Figure 4.3 shows the runtime profile of Bluestein FFT kernel on an Intel Xeon workstation which has a L2 cache of 6M bytes. In this figure X axis is the number of memory accesses of different DFT sizes and Y axis is the average stalled cycles per memory access. This stalled cycle profile is generated using the hardware counter tool, performance application programming interface (PAPI). As we can see, the curve is three-fold along X axis. For small DFT sizes, the average stalled cycle is only 0.2 cycles per access. Then there is huge increase in the middle range and average stalled cycle remain about 0.7 cycles per access afterwards. The dramatic increase in the figure coincide with L2 cache size. A more detailed study of cache misses using PAPI shows that the average memory access cost is small for the first segment because there is only limited L1 cache miss. After exceeding the L2 cache size, the program kernel receives more and more L2 cache misses per access until the rate eventually becomes flat.

With the above observation, a segmented memory access cost is used in the implicit abstract machine model. When the memory footprint of an FFT algorithm

![Figure 4.3: Memory access profile of Bluestein on Intel Xeon](image)
kernel is less than L2 cache size, the small memory access cost in figure 4.3 is used. When the accessed area is much larger than L2 cache, the large memory access cost is used. Linearly interpolated values are used for sizes around L2. The turning point around L2 can be determined before runtime by check the machine’s memory configuration. Empirically the turning points in figure 4.3 are roughly $0.5 \times L2_{\text{size}}$ and $4 \times L2_{\text{size}}$. Or without any parameter input, the training on a series of increasing DFT sizes can detect the start and end of this dramatic increase in cost of memory accesses. This way, the training of the models is more automatic.

4.5 Model for Generic DFT Algorithm

The above model is used to predict to performance of FFT algorithm kernels. The well known FFT complexity, $O(n \log(n))$, is a result of recursive implementation of FFT decomposition. The kernel part we models each time, however, only has complexity of $O(n)$. Generic DFT algorithm solve a DFT problem without any decomposition and its complexity is $O(n^2)$. According to the definition of DFT, Generic DFT works n times on a size $2n$ complex input/output data and access $n^2$ complex twiddle factors. BTW, twiddle factor are computed before hand in FFTW because its value can be computed with just knowledge of the DFT size but not the input data.

In Generic DFT algorithm, all operations including memory accesses grows quadratically with the increasing of problem size. Hence, if the performance cost of all operations is constant, the overall runtime will be a quadratic function of n. However, memory access cost is a function of DFT sizes as is the case of FFT algorithm kernels. As we can see in figure 4.4, memory accesses largely dominate the performance of Generic DFT. Since number of memory access is a linear function of $n^2$, we can view $/(n^2)$ in the figure as per memory access. With the increase of DFT sizes, L1 cache misses and L2 cache misses experience a similar three-fold curve. L2
cache misses coincide with the runtime on turning points much better than L1 cache misses.

With a similar profile on other two-level-cache platforms, I simplified the implicit abstract machine model to a fractional quadratic model as is shown in equation 4.3

\[
T_{\text{generic}} = \begin{cases} 
q_1 n^2, & \text{if within L2;} \\
q_{\text{interpolate}} n^2, & \text{if within L2/2 and 4*L2;} \\
q_2 n^2, & \text{if much larger than L2}
\end{cases}
\]  

(4.3)

In this equation, \( q_1 \) is the average of quadratic coefficient if problem memory footprint is less than half of L2 cache and \( q_2 \) is the one for problems memory footprint much larger than L2 cache size. \( q_{\text{interpolate}} \) is the interpolation of the above two in the transaction segment.

As we mentioned before, FFTW limits the applicability of Generic DFT algorithm to DFT sizes less than 173 because of its \( O(n^2) \) complexity. It may only be fast for some small prime size DFT problems when the data is within L2 cache. Practically, Only the first segment of the model is needed. Or alternatively, the runtime of limited prime sizes less than 173 is recorded and used directly later on.

Figure 4.4: Performance profile of Generic DFT on Intel Xeon
4.6 Prediction Results of Individual DFT algorithms

In this section I will show the prediction result of individual DFT algorithm kernels using the implicit abstract machine model with segmented memory access cost and result of Generic DFT using fractional quadratic model. There are three major FFT algorithm kernels in high level decomposition, they are Bluestein, Rader and Cooley Tukey. Bluestein and Rader are both applicable to prime size DFT problems. Together with Generic DFT, they are the major potential solution for prime size DFT problems. Correctly modeling the performance of them will help to make a right decision when decomposing a prime size problem. Accurately predict the performance of Generic DFT will facilitate the right choice of small problems and that of Cooley-Tukey is important to composite large composite DFT problems.

Training of the performance model for each algorithm is done on a list of DFT sizes that are separated roughly by 2 in log scale. The training data ranges from 10 to 30000. Test DFT sizes are picked from a different set of data in the same range after training. Figure 4.5 show the comparison of actual runtime and predicted runtime using the implicit abstract model. The blue lines with x points are the actual runtime of the algorithm kernels and the red lines with * points are the predicted time given by the model. The predicted runtime is normalized with respect to the actual runtime. As we can see, the prediction is very accurate for most cases and there is only a slightly bigger error on Rader when DFT size is larger than 10000. On average, the model achieves less than 10% prediction error for all individual DFT algorithms.
Figure 4.5: Prediction result of individual DFT algorithms on Intel Xeon
Chapter 5

PERFORMANCE MODELS FOR CODELETS

Each codelet has a fixed number of operations for a particular problem size. Therefore there is no need to apply abstract machine model on it. However, codelets often have different strides depending on the problems it is applied to. I built a model to specifically model the effect of strides on the performance change of codelets.

5.1 Scalar Codelets Model

Codelets are generated to solve small DFT problems of those with small factors in Cooley-Tukey method. The first class of codelets are called direct codelets and the other is called twiddle codelets which combine twiddle multiplication and size $N_1$ DFT together. FFTW version 3.2.1 includes codelets of size 2-16, 20, 25, 32 and 64. For each size, the direct and each iteration of twiddle codelets have fixed number of operations. Codelets are more likely to be implemented at the bottom of decomposition and have a large stride because of Cooley-Tukey’s shuffle effect. The runtime profile of a direct codelet $n25$ of different strides is shown in figure 5.1 Stride size along X axis roughly doubles from one to the next point. Three stride types along the same vertical line are not of the same size but of very close sizes. A couple of observations can be made in this figure. First of all, the overall curve of performance versus memory access range is somewhat similar to that of abstract machine model, despite of different stride types. The performance has a huge decrease around L2 cache size and both small strides and large strides have a flat performance curve. All three stride types, power-of-two, odd and other even
strides, has similar curve in figure 5.1. However among them, we can see power-of-two strides has the worst performance of all. This is not hard to understand because all cache feature sizes are of power two and such a stride will easily incur large numbers of conflict cache misses. The penalty of L1 cache misses is much smaller than that of L2 cache misses therefore this effect is not too serious for strides much smaller than L2. For the non power two cases, we can still see some performance difference between odd and other even strides. Codelets with even but non power two strides performance better than those with odd stride. A possible explanation of this is even strides have better alignment than odd ones and this leads to fewer numbers of physical memory accesses.

Besides what we discussed above, strides with large power-of-two factors also affect the performance of codelets. For example, if $stride_A = n = L2\_size/4$, where $n$ is power-of-two, every other 4 elements accessed will be mapped to the same L2 cache set. If $stride_B = 5 \times n/4$, then $stride_B$ has a large power-of-two factor $n/4$ and every other 16 elements accessed will be mapped to the same L2 cache set. With similar compulsory cache misses but less conflict misses, we can expect the performance of even strides with large power-of-two factor will be better than that
of power-of-two strides but worse than other cases.

Accordingly, the performance model for codelets divides codelets strides into three cases: power-of-two, odd and even strides. If memory access region is smaller than half of L2 or larger than four times of L2, an averaged runtime is used for that segment and each stride type. Interpolation method is used in the L2 cache transition segment for each stride type. However, when stride $n$ is even and has a large power-of-two factor, it is treated differently. Assume stride $n = 2^p * m$, where $m$ is an odd number, and $2^q < m < 2^{(q+1)}$, then a coefficient $\alpha = p/(p+q)$ is adopted to represent the percentage of power-two part in $n$. Predicted runtime is adjusted according to equation 5.1.

$$T = T_e + \alpha * (T_p - T_e),$$

(5.1)

where $T_p$ and $T_e$ are the runtime of the codelet with close power-of-two strides and even non-power-two strides. When the stride is not close to power-of-two or numbers that has large power-of-two factor and it is smaller than L1 or much larger than L2 cache size, an averaged runtime of that stride type is used. otherwise, interpolation is used on odd/even strides depending on the stride type.

5.2 SIMD Codelets Model

SIMD is very important to FFTW on architectures that support such instruction extensions. The performance of FFT plans with SIMD enabled can be 50% faster than the scalar counterpart. The current version of FFTW, fftw-3.2.1, supports SIMD instruction extensions including SSE, SSE2 and Altivec. It is turned off by default in the download package and users need to set the flag, like ”–enable sse2” while compiling to enable the extension. Codelets is the only part in FFTW that uses SIMD extension and all other FFT algorithms will still be executed in a scalar fashion. In these high level algorithms, problem size is not fixed and it is
more difficult to achieve the correct data alignment and data level parallelism that is required for SIMD instructions.

SIMD codelets are generated by genfft and C macros for different SIMD intrinsics are used in these codelets, therefore the code is portable to some extend. There are two implementation scenarios for SIMD codelets in FFTW. One is SIMD with vector length of two and the other of four. SIMD with vector length of four takes advantage of parallelism in vector loops of the same DFT problem. It batches four iterations of the identical DFT problem together. Despite of the high efficiency, this implementation is limited to batched DFT problems or loops of DFTs generated as child problems of a single top level DFT problem. On the other hand, SIMD with vector of length-2 relies on the natural parallelism between real and imaginary parts of the complex data as is shown in equation 5.2.

\[
DFT(A + iB) = DFT(A) + iDFT(B)
\]  

(5.2)

One complex input DFT problem is split into two identical real input DFT problem and the result of the original one is a complex addition of the two individual outputs. This general case is applicable to all complex input DFT problems and is used much more often than the case where vector length is four. Besides these, the input/output data in memory is required to be aligned correctly depending on whether the input is single or double precision and the vector length of the SIMD instructions. Figure 5.2 (b) shows the performance of a SIMD direct codelet n1fv_25 on Intel Xeon processor with SIMD extension SSE2 enabled. Despite that SIMD codelets perform better than the corresponding scalar versions, the performance pattern of SIMD codelet for different strides types is similar. Therefore we still measure performance of each SIMD codelet with power-of-two, odd and even strides separately. When memory footprint of a codelet with certain stride is smaller than half of L2 or larger than 4 times of L2, an averaged measurement is used directly. Otherwise, the same interpolation method as what is used for scalar codelets is applied for the estimation
of the runtime for codelets with arbitrary strides. These three cases are shown in equation 5.3 and 5.4.

\[ T_{\text{power-of-2}} = \begin{cases} 
  t_{1, \text{power-of-2}}, & \text{if stride less than } 0.5 \times L_2; \\
  t_{\text{power-of-2, interpolate}}, & \text{if stride within } 0.5 \times L_2 \text{ and } 4 \times L_2; \\
  t_{2, \text{power-of-2}}, & \text{if stride larger than } 2 \times L_2 
\end{cases} \]  

(5.3)

\[ T_{\text{odd}} = \begin{cases} 
  t_{1, \text{odd}}, & \text{if stride less than } 0.5 \times L_2; \\
  t_{\text{odd, interpolate}}, & \text{if stride within } 0.5 \times L_2 \text{ and } 4 \times L_2; \\
  t_{2, \text{odd}}, & \text{if stride larger than } 2 \times L_2 
\end{cases} \]  

(5.4)

\[ T_{\text{even}} = \begin{cases} 
  t_{1, \text{even}}, & \text{if stride less than } 0.5 \times L_2; \\
  t_{\text{even, interpolate}}, & \text{if stride within } 0.5 \times L_2 \text{ and } 4 \times L_2; \\
  t_{2, \text{even}}, & \text{if stride larger than } 2 \times L_2 
\end{cases} \]  

(5.5)

5.3 Prediction results of Codelet Model

Figure 5.3 shows prediction results of one scalar codelet, n25, and one SIMD codelet, n1fv25. The training of our prediction model is done on three sets of input
strides, power-of-two, odd and even. Each set has about ten different sizes ranging from 10 to 10000. After the model is established, a new set of input is tested on the codelets. On those new sizes, our predicted results are close to the actual runtime. Again all runtime is normalized against actual runtime. From the figure, we can see, for small sizes and large sizes, the prediction is more accurate. For middle sizes, the prediction error can be as large as 20%. In deed, the cache of modern architectures is highly sensitive to different strides and base addresses. It presents some kind of randomness and its influence on performance is very hard to precisely predicted. Only some simple stride cases, like odd, even, power-of-two and large power-of-two factors can be well handled.

Despite the prediction error in the middle problem size range, the codelet model still achieves within 10% prediction error on average. More importantly, the purpose of the codelet prediction is to choose fast codelets from others ones. Frequently, the error we present does not affect the final correct choice of codelets. Our overall result in the next chapter will show that the precision achieved on individual codelets is enough to provide a reasonably good DFT plan at the end of the FFTW search process.
Chapter 6

MODEL TRAINING AND EVALUATION

In this section I will describe the training of the performance models presented above. The overall result of a search engine using these models will be compared with the original FFTW in two aspects.

6.1 Training of Performance Models

Training of the model is necessary for the models to get the parameter they need according to the local machine features. It is done only once during the FFTW installation when each solver is registered to the planner. The performance of DFT kernels with different sizes and codelets with different strides are measured during this time. Model parameters are extracted using the following linear regression method. For each FFT algorithm kernels and the Generic DFT algorithm, about 10-20 training sizes are picked from the range of 10 to 30000. The instruction counts of each algorithm kernel are estimated from the source code and actual runtime is recorded. For the case of codelets model, about 10 different strides for each codelet are chosen and their runtime is measured.

After getting the above information, the following weighted linear regression method shown in equation 6.1 is used to obtain the parameters in implicit abstract machine model. Given $n$ instruction types and $m$ training points, the $j_{th}$ actual runtime is $T_{A,M,j}$. Since we care about the relative runtime error instead of absolute error, the weighted residual in equation 6.1 is minimized for the optimal solution.
\[
\sum_{j=1}^{m} \left| \sum_{i=1}^{n} \frac{C_{A,i,j}P_{M,i} - T_{A,M,j}}{T_{A,M,j}} \right|^2
\]  \hspace{1cm} (6.1)

A solution to this weighted linear least square problem is given the matrix form in equation 6.2, where \( W \) is a diagonal matrix and \( w_{i,i} = \frac{1}{T_{A,M,j}^2} \)

\[
\bar{P} = (C^TWC)^{-1}C^TW\bar{T}
\]  \hspace{1cm} (6.2)

In the case of fractional quadratic model of Generic DFT and fractional model of codelets, a direct average method is used. The average quadratic coefficient \( q_n \) is extracted from smaller than and larger than L2 segments. The average runtime of codelets with strides much smaller than L2 or larger than L2 is computed. The middle range of both case is covered by linear interpolation method. All FFT kernels and codelets with different implementation details are discriminated by different model parameters. The measured runtime of any code segment has a little variance of about 5%. To alleviate this effect, Firstly, I batch the runtime of small kernel into longer runtime. Looping is not used because all other iterations except the first have a hot cache and have much less cache misses. Secondly, I repeat such measurements four times any pick the one with smallest runtime, which will minimize the effect of different machine status.

### 6.2 Evaluation Environment Setup

I set up the model-driven search engine base on FFTW version 3.2.1. The work flow of recursive search in decomposing a DFT problem is the same. Only performance measurement on each level of search is replace with runtime prediction generated by performance models. Dynamic programing is still used to reduce redundant performance prediction. By doing recursive search, the plan with the least overall predicted cost is returned as \textit{optimal}.

FFTW search engine has some internal search flags to constrain the search space. These flags are mapped from user interface patience levels, namely \textit{Exhaustive}, \textit{Patient}, \textit{Measure} and \textit{Estimate}[18]. \textit{Exhaustive} mode traverses the whole
Table 6.1: Test Platforms Configuration

<table>
<thead>
<tr>
<th></th>
<th>Athlon 64 X2</th>
<th>Xeon 5405</th>
<th>PowerPC 970</th>
<th>UltraSparc III</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>2 GHz</td>
<td>2 GHz</td>
<td>2.3 GHz</td>
<td>1.06 GHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64 KB</td>
<td>64 KB</td>
<td>32 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1 MB</td>
<td>6 MB</td>
<td>512KB</td>
<td>1 MB</td>
</tr>
<tr>
<td>OS</td>
<td>linux 2.6.24</td>
<td>linux 2.6.23</td>
<td>linux 2.6.24</td>
<td>SunOS 5.10</td>
</tr>
<tr>
<td>SIMD</td>
<td>3DNow!(not supported)</td>
<td>SSE2</td>
<td>Altivec</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

FFTW search space and takes the longest time to return a plan, e.g. 503 seconds for a DFT of size 27000 on a 2GHz Athlon desktop. *Patient* and *Measure* modes exclude solvers that are unlikely to lead to the optimal solution, and hence reduce the search space. They generally find plans that perform worse than the *Exhaustive* but spending less search time. Their search overhead, however, is still huge for large problems and increases polynomially with the problem size. *Estimate* mode further reduces the search space and uses the estimated total number of floating point operations as the metric of best plan. Clearly, such a strategy oversimplifies a machine model by treating different operations with the same delay and neglecting other factors such as ILP and memory access stalls. For each of the above cases, if SIMD extension is enabled, the search space will become larger because extra SIMD codelets are included and therefore it will take longer to return a plan.

We use the model-driven search engine on the search space of *Exhaustive* mode and it greatly reduces the search time. Similarly, search time will be reduced proportionally if we use our model in *Patient* or *Measure* mode. We conduct the comparison on four platforms: AMD Athlon, Intel Xeon, IBM PowerPC and Sun SPARC. The configurations of the four architectures are shown in Table 6.1. The comparison is made in three aspects. We first show the accuracy of the performance prediction of three FFT algorithm kernels, Generic DFT algorithm, a scalar codelet and a SIMD codelet.
6.3 Overall Evaluation Results Comparison

An FFTW version with the model-driven search engine and the orginal fftw-3.2.1 version are compared in two aspects. First, the runtime of best plan returned by each version is compared. Second, I compared the search time spent by each mode.

6.3.1 Runtime Comparison

As described before, FFTW has four different search strategies. I pick out two representative ones, Estimate and Exhaustive to compare with my model-driven search engine. These three methods are different not only in their evaluation mechanism but also in the sizes of search space. The model-driven search engine runs on almost the same search space as Exhaustive. The runtime of best performing DFT plans returned by each search method are compared first. Figure 6.1 shows the performance comparison among different DFT plans on four test architectures with their SIMD extension disabled. All three search engines engines, model-driven, Exhaustive and Estimate, perform good (at most 10% – 20% slower than the best) for most small size DFTs. While for large problem sizes on Xeon, Estimate plans generally run 10% – 20% slower than Exhaustive plans, with occasional 50% slowdown. Large size DFT plans on AMD found by the Estimate mode run 20% – 130% slower than the plans found by the Exhaustive mode. Our model-driven optimization engine achieves comparable performance with the Exhaustive mode. With the exception of two cases, our model-driven engine finds plans that are at worst 10% – 20% slower than best plan. On average, our model-driven optimization engine achieves 94.4%, 94.8%, 93.6% and 94% of the performance of FFTW Exhaustive on these four platforms. For the cases where our search engine does not performance well, it is either because our model fail to give an accurate runtime prediction or because our actual search space is a bit smaller than Exhaustive mode. We have not extended our work to real(non-complex) DFT solvers which sometimes are used in
Figure 6.1: Scalar runtime comparison among Exhaustive, Estimate and Model-driven mode
complex DFTs. Figure 6.2 shows the performance of the three search strategies

Figure 6.2: SIMD runtime comparison among Exhaustive, Estimate and Model-driven mode

with SIMD extension enabled. Among the four platforms we tested, Intel Xeon supports SSE2 for double precision DFT and PowerPC970 supports Altivec for single precision. 3DNow of AMD is no longer supported in the current version of FFTW. Our performance model outperforms the Estimate mode over the whole test region. The model-driven strategy performs comparably with Exhaustive mode for most sizes but the performance of Estimate mode is about 20% to 30% slower than the best. This is because the Estimate mode relies on instruction counts to optimize the performance of codelets, which is inapplicable in the case of SIMD.

6.3.2 Search Time Comparison

Compared to Exhaustive search, the model-driven search engine loses a little bit performance. While the benefit is the significant decrease of search time as a trade-off. Figure 6.3 shows the comparison of search time spent by three strategies on different DFT sizes. The search time is in log scale and normalized against search time spent by Exhaustive mode. Of all three strategies, Exhaustive search
mode spend the most time, model-driven search mode spend about 1% to 10% of Exhaustive’s search time. This is considerable amount of time in the case of large DFT sizes. Estimate mode, on the other hand, spend only 1% of the search time spent by model-driven mode. There are mainly two reasons contributing to this gap. First of all, Estimate mode runs on a much smaller search space than the other strategy. Second, the heuristic of minimizing instruction count used in Estimate mode is considerably simpler than the performance model I use. However, the absolute amount of time spent by the model-driven model is still small, not larger than a couple of seconds even for the largest DFT size. Here I only listed the comparison on AMD Athlon and Intel Xeon, but PowerPC and Sparc also have similar results.

Overall, the model-driven search engine achieves about 95% of the performance of Exhaustive search engine and uses only less than 5% of its search time. The model-driven optimization engine achieves the goal of model based optimization, that is, delivering performance comparable to that of exhaustive search but using much smaller amount of search time.
Chapter 7

CONCLUSION AND FUTURE WORKS

In the thesis, I present a set of performance models for DFT algorithms and codelets and use these model to optimize the search engine in FFTW. This work has successfully replaced the empirical search engine while keeping a reasonably good performance in FFTW. It achieves 95% of the performance of exhaustive search and uses 5% of the optimization time on average. It is a good trade-off between performance and search time. This work also provides a method to break down the runtime of different DFT algorithms. It helps to find out the crossover points of performance curve among different candidate algorithms and helps to understand why some algorithms are faster than others on a given architecture.

In these models, individual FFT algorithm kernels are the modeled targets. But the method may be potentially valuable for other transformations and signal processing algorithms with small kernels having little branch instructions. The number of different operations and memory architecture penalties can be modeled similarly for performance prediction. Indeed, this model-driven performance prediction technique may be applied to other linear transformation libraries that rely on empirical search.

An important conclusion we can draw here is that model-driven methods are valuable tools in helping reducing empirical search time. In some cases, they can replace the empirical search method and reach a good trade-off between search time and implementation performance. Reducing search time means a lot to DFT
libraries like SPIRAL and FFTW. To generate the code for a DFT of size several thousand, SPIRAL can spend several hours on a powerful workstation. A major part of the time is spent on performance evaluation for different versions. In FFTW, this issue is not as serious as that in SPIRAL, but it still takes more than ten minutes to generate a plan for DFT of size 30000 in Exhaustive mode. This thesis is an important step to address this issue.

Sampling and linear interpolation is used both in an abstract machine model and codelet performance model. An unified and general adaptive performance model without data training or interpolation is still an open research topic. This work will be more complete if it is extended to real( non-complex) DFT algorithms. It will also be interesting if choices of best plan on each search node can be directly given by some rules that are learned from a one-time performance training. Furthermore, it is still a challenge to generalize this work and apply model-driven optimization on other complicated scientific libraries.
BIBLIOGRAPHY


