DFT Performance Prediction in FFTW

Liang Gu and Xiaoming Li

University of Delaware

Abstract. Fastest Fourier Transform in the West (FFTW) is an adaptive FFT library that generates highly efficient Discrete Fourier Transform (DFT) implementations. It is one of the fastest FFT libraries available and it outperforms many adaptive or hand-tuned DFT libraries. Its success largely relies on the huge search space spanned by several FFT algorithms and a set of compiler generated C code (called codelets) for small size DFTs. FFTW empirically finds the best algorithm by measuring the performance of different algorithm combinations. Although the empirical search works very well for FFTW, the search process does not explain why the best plan found performs best, and the search overhead grows polynomially as the DFT size increases. The opposite of empirical search is model-driven optimization. However, it is widely believed that model-driven optimization is inferior to empirical search and is particularly powerless to solve problems as complex as the optimization of DFT.

In this paper, we propose a model-driven DFT performance predictor that can replace the empirical search engine in FFTW. Our technique adapts to different architectures and automatically predicts the performance of DFT algorithms and codelets (including SIMD codelets). Our experiments show that this technique renders DFT implementations that achieve more than 95% of the performance with the original FFTW and uses less than 5% of the search overhead on four test platforms. More importantly, our models give insight on why different combinations of DFT algorithms perform differently on a processor given its architectural features.

1 Introduction

Adaptive libraries usually apply application-specific knowledge to define a search space for potential implementation of their target routines such as BLAS and DFT. Furthermore, empirical search is used to find the best performing version from that search space. Well-known adaptive libraries include FFTW [8,10], ATLAS [18] generating the Basic Linear Algebra Subprograms(BLAS), SPIRAL [14] performing linear digital signal processing transforms, as well as Sparskit [16] and Sparsity [11] focusing on sparse numerical computation. These libraries usually outperform the best hand-tuned implementation, and sometimes achieve peak performance on some platforms.

It is generally acknowledged that the good performance of those libraries depends on the generators' extensive search process. A typical example is SPIRAL, which employs empirical search for both algorithm and implementation optimizations. However, there has been a long-standing question: why empirical search seems to be indispensable to the generation of high quality code. In some cases, empirical search explores implementation forms that are hard to deduct from models. Other times, it selects better value for the parameter of the transformation that is applied to the routine. Finally, the advantage of empirical search might be its capabilities of applying optimizing transformations in different orders so that it can solve the so-called "phase-ordering problem" [12]. It is helpful to compare empirical search with its opposite, model-driven optimization. There have been some previous efforts to compare these two approaches in ATLAS. Yotov et al. [19] show that the empirical search in ATLAS can be successfully replaced by an architectural model while still maintaining its good performance. Furthermore, researchers in the signal processing domain have accumulated a set of heuristics for the selection of best DFT algorithms for a particular problem [1]. More recently, Fraguela et al. [6] show that a properly built memory model can successfully select the best performing DFT algorithms in SPIRAL even though its runtime prediction is far from accurate. Overall, it is still unclear how architectural features of a processor such as instruction latency and SIMD(Single-Instruction Multiple Data) instructions affect the performance of a DFT.

This paper presents a quantitative evaluation of the empirical search strategy of FFTW and a new model-driven optimization. It replaces the original FFTW empirical search engine and produces equally high-quality code. The performance modeling of the empirical search in FFTW differs from many other library generators, like ATLAS, which have relatively fixed formulas of implementation and limited number of parameters to be tuned. FFTW, however, has some pre-generated codelets together with other algorithms that can potentially compose the best implementation for a DFT problem. The empirical search engine in FFTW tries to find the best combination of problem decomposition strategies, which is called "a plan" in FFTW. Each of the decomposition strategies is a complex manipulation of the target FFT problem using codelets or other algorithms. Therefore, the fundamental degree of freedom in FFTW search space is not the parameter values but the algorithm structure of a solution.

This paper makes two major contributions. The first one is the building of performance prediction models for several frequently used DFT algorithms and DFT codelets. Particularly, we present how to automatically determine the parameter values of these models on different computer architectures. The second contribution is the composition of these individual DFT models and the building of a model-driven optimization engine. It produces plans that have comparable quality to those generated by exhaustive search in FFTW while eliminating most overhead of empirical search.

The rest of this paper is organized as follows. Section 2 describes the existing empirical search engine of FFTW. In Section 3, we quantitatively analyze and model the performance of several DFT algorithms and DFT codelets used in FFTW. Section 4 presents the model-driven optimization engine for FFTW. In Section 5, we comprehensively compare a modified FFTW using our model-driven optimization engine with the original FFTW on four different platforms. Finally, we conclude and suggest future directions of research in Section 6.

2 Empirical Searching in FFTW

FFTW has a code generation phase and a runtime phase. In its code generation phase, FFTW generates and compiles some C subroutines for small DFTs, while in its runtime phase, FFTW conducts *all* of its empirical search. All FFTW's capabilities of adapting to different computer architectures is the result of its runtime empirical search. As a result, the efficiency and accuracy of FFTW's runtime empirical search are relatively more important than that of other libraries. In this section, we overview these two phases and some important factors that contribute to the high performance of FFTW.

2.1 FFTW Code Generation Phase

FFTW's low-level optimization is performed within codelets [7]. Some of the codelets are architecture independent(scalar codelets), while others specially take advantage of data level parallelism by using SIMD instruction extensions such as SSE, SSE2, Altivec etc. All these highly optimized straight-line style codelets are generated using a special-purpose compiler called *genfft* written in OCaml. For each small DFT size, *genfft* implements only one DFT algorithm which is expected to be efficient for most architectures. DFT algorithms used in codelets include the Cooley-Tukey algorithm [4], the Prime Factor algorithm [13, p619], the Generic DFT algorithm and the Split-radix algorithm [5]. For each small size DFT codelet, *genfft* first implements one of the above DFT algorithms. Then it simplifies the initial code with a handful of common compiler optimizations such as constant folding and common expression elimination. Two DFT-specific optimizations, i.e. making all numerical constants positive and applying network transposition, are also used. *genfft* schedules the instructions using a cache oblivious algorithm to achieve the asymptotic optimum for register spilling. Finally, the internal representation of the algorithm, written in OCaml, is un-parsed to straight-line C code.

Most users do not need to use the codelet generator during installation. The downloaded FFTW package includes pre-generated codelets for some DFT problem sizes from 2 to 64. Users can generate codelets of other sizes according to their need using *genfft*.

2.2 FFTW Runtime Phase

FFTW performs empirical search in the runtime phase. When the FFTW library is invoked by a user program, the FFTW search engine will apply a wide range of DFT algorithms to the target problem, so as to span a huge enough search space to cover the best solution. Among all $O(nlog(n))$ FFT algorithms, FFTW implements those that are efficient and widely used on modern architectures, including Cooley-Tukey for composite size DFT, Rader's [15] and Bluestein [2] for prime size DFT problems. Generic DFT (or Direct DFT) $O(n^2)$ algorithm is the most straightforward solution to DFT problems and is able to solve all DFT sizes. The Generic DFT algorithm is generally much slower than FFT algorithms. However, FFTW still implements it because it may be beneficial for some small size problems. Each of the above algorithm may have several variants (referred as different *solvers*). For instance, Cooley-Tukey has Decimation In Time (DIT) and Decimation In Frequency (DIF), buffered and non-buffered solvers. By using these different solvers, FFTW can generally decompose a large size DFT problem into smaller size sub-problems and recursively solve those sub-problems. One thing worthy noting is that even a prime size DFT can be decomposed into composite size one by applying Rader or Bluestein. This decompose process stops when the problem size is small enough and FFTW can solve it directly with a codelet or a Generic DFT solver.

For a specific DFT problem, the search space of FFTW grows like a tree. Search space can become extremely huge for large size DFT problems. Decision of which solver to choose on each level generally relies on both this solver's performance and its child solvers' performance. A fundamental question raised by this scenario is how we should traverse this huge search space to find the best solution. An easy answer to this question is exhaustive empirical search, which guarantees the best solution within the search space. Indeed, FFTW applies four strategies in its search engine, namely, *Exhaustive*, *Patient*, *Measure* and *Estimate*, in the order of decreasing sizes of search space explored. The first three strategies apply different possible solvers, run each version of the code, measure the performance and select the best, which generally takes a long time to return the best solution for a large DFT problem. *Estimate* mode uses simple heuristics to estimate performance and hence predict the best solver combination, which is often not accurate but is the fastest strategies. An ideal optimization technique for FFTW should generate plans that perform comparably with the plans generated by *Exhaustive* strategy but have much less overhead.

3 Program Analysis and Prediction Models

A specific DFT implementation, i.e. *plan*, is a hierarchical combination of individual solvers. In order to predict the combination's performance, we start from individual DFT algorithms. FFTW employs both Generic DFT algorithm and several FFT algorithms that divide a DFT into child DFT problems. A recursive implementation of an FFT algorithm has a complexity of $O(nlog(n))$ but the kernel part has a complexity of $O(n)$ if child problems are excluded. An example of such kernel part is the twiddle factor multiplication in Cooley-Tukey. Our models predict the performance of FFT kernels recursively and the performance of Generic DFT as a whole. Codelets are highly efficient components in FFTW and the correct choice of them is crucial to performance. Since each of them has fixed number of instructions but frequently different strides. We predict its performance with respect to the different strides.

Generally, it is very hard to predict the performance of a program on any modern architecture. However some efforts have been made on some special programs, for example, various benchmark performance was predicted in [17] by using an abstract machine model. Models for memory hierarchy can be combined with empirical search to improve the performance of dense-matrix computations as shown in [3]. More recently, a highly effective model-driven optimization engine [19] was developed for ATLAS to predict the relative performance between code versions that have different values for transformation parameters. These above works have inspired us to propose an adaptive model-driven DFT performance prediction technique in FFTW. Our model-driven search engine is developed in three steps: (1) Program analysis and performance modeling using a fractional abstract machine model and a codelet model. (2) Training models on the target computers to determine their architecture dependent parameters using regression. (3) Recursive performance prediction to choose the optimum.

We will not declare these models can always give a very accurate runtime prediction for complex DFT plans on all architectures, nor do we need to do that in FFTW. What we really care about is the *relative performance* between different solvers, and we can tolerant some performance prediction errors while still being able to pick a *good* solution.

3.1 Fractional Abstract Machine Model

As we mentioned before, FFTW has 4 major $O(nlog(n))$ FFT algorithms, Cooley-Tukey, Bluestein, Rader and the Prime Factor algorithm. Among them, Prime Factor, which involves less computation but more memory operation comparing to Cooley-Tukey, is not beneficial for modern architectures and is not implemented in the high level optimization. The Cooley-Tukey implementation is limited to the case where both factors are larger than 16. Otherwise, its kernel, i.e. the twiddle factor multiplication, is incorporated into some special codelets used in one of its child problems. We use a fractional abstract machine model to predict the performance of the three FFT kernels and Generic DFT.

We start modeling the algorithms' performance by putting them in an ideal environment where all data are in L1 cache(there is no memory stalls) and hence the performance is only determined by the type and the number of operations. This assumption follows the work of Saavddral, et.al. [17], in which an abstract machine model based on Fortran language was used to predict the performance of benchmarks. Under the no-stall and no-parallelism assumption, the run time of a program is a weighted linear combination of operations as shown in (1)

$$T_{A,M} = \sum_{i=1}^{n} C_{A,i} P_{M,i} = \boldsymbol{C_A P_M}. \tag{1}$$

$\boldsymbol{C_A}$ is the program characteristic vector and $C_{A,i}$ is the number of instruction type i in algorithm A. A complete $\boldsymbol{C_A}$ should include all instruction types that appear in the code. Vector $\boldsymbol{C}_A$ is obtained from a static analysis of the source code. and part of it is already provided by FFTW. $\boldsymbol{P_M}$ is the machine performance vector that describes the cost of each instruction type on a specific computer architecture.

This model is an explicit model where each element of machine performance vector is the latency of the corresponding instruction and the machine performance vector can be explicitly measured from a set of micro-benchmarks or collected from the processor manuals. This model is simplified by including only the most expensive instructions such as floating point addition, multiplication, division and memory accesses, which take the majority of the execution time. Furthermore, we do not consider a very limited number of function call and branch instructions in the kernel code. For the loop condition branches, most processors can predict them quite well, and they incur almost no more overhead than an integer operation.

Next we want to remove the no-parallelism and no-stall assumption. To do so, we need to calculate the effective machine performance vector, that is, the effective latencies for various operations. The effective latency depends on the schedule of instructions and might be different from the absolute operation delay defined in the processor manual. However, it is usually very difficult to deduct effective operation latency from source code given the intrinsic latency of instructions because different instruction set architectures (ISA) and the complex interaction between instructions, like instruction level parallelism (ILP). Therefore, instead of treating the machine performance vector as a global invariant , we make it local to each FFT algorithm because it is determined by the unique instruction schedule, stalls and ILP. As a result, each algorithm has its own local performance vector on a particular computer architecture and it is empirically determined using regression methods. The empirical determination of algorithm-specific performance vectors consider both the intrinsic latency of the corresponding instruction and the unique instruction mix and schedule of a FFT algorithm kernel.

In summary the effective performance vector of an algorithm is determined by the tuple $(algorithm, architecture)$ and can be learned by first measuring a number of $T_{A,M}$, each being the performance of a specific application of the algorithm A on machine M. The program characteristic vector $\boldsymbol{C_A}$ can be obtained by analyzing the code using common compiler parsing techniques. Finally the performance vector $\boldsymbol{P_M}$ is regressed from $T_{A,M}$ and $\boldsymbol{C_A}$ using (1). The details of the exact regression method are discussed in Section 4. One thing worth noting is that all loop bounds in the algorithms can be statically determined, which makes it easy to compute $\boldsymbol{C_A}$.

Finally, we take into consideration of different memory access latency for our model. The typical two-level cache hierarchy of modern architecture has a very important impact on performance of large size DFTs. A careful study of the relation between cache misses and runtime reveals that we need to implement the above model in three DFT size segments separately according to L2 cache size. Figure 1 (a) shows the profile of average stalled cycles per memory access and the total number of memory accesses in Bluestein kernel. These numbers are collected using hardware counters. The L1 and L2 cache of the tested platform are 64K and 6M bytes. When the Bluestein's number of memory footprint is less than roughly half of L2 size, the average stalled cycles per memory access is

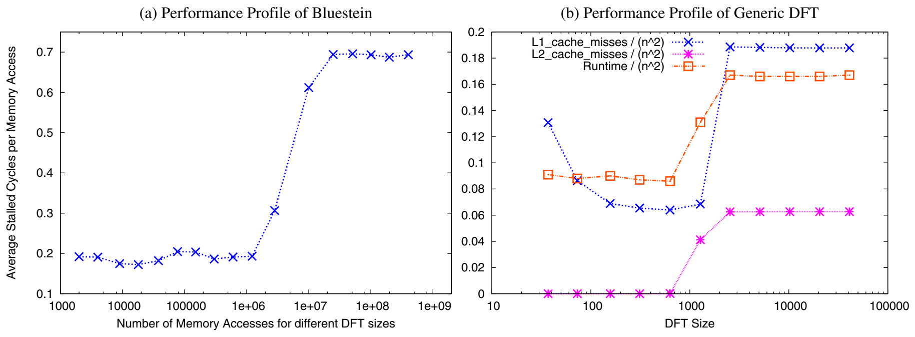


Fig. 1. Profile of Bluestein and Generic DFT on Intel Xeon

relatively small because most are caused by L1 cache misses. With the increase of total number of memory accesses, hence the L2 miss rate, there is a significant increase on average stalled cycles per memory access. The increase stops when the memory footprint is larger than roughly four times of L2 cache size. More profiles of other FFT solver kernels show similar change of average memory access cost on different architectures. Since the cost of other instructions remain unchanged for different DFT size, we only need to adjust the cost of memory accesses according to DFT sizes. Unlike the work by Fraguela et.al. [6] that uses an analytic memory model, we develop an empirical model to compute the cost of memory accesses in the algorithms.

For Bluestein, Cooley-Tukey and Rader kernels, the average cost of memory access can be treated as a constant for DFT sizes that are much smaller or much larger than L2 cache, where L1 or L2 cache misses dominate the cost. A linear interpolation of average memory access cost is used for the transition segments where DFT memory footprint is around L2 cache size. The L2 transition region can be determined by using hardware counter as shown in Figure 1 or by empirical ways, i.e. capturing the nonlinear turning point in the runtime over DFT sizes data sets. Our experiment results shown in Section 5 demonstrate that our implicit abstract machine models achieve on average less than 10% error in predicting the performance of FFT algorithms.

The only $O(n^2)$ algorithm used in FFTW is the Generic DFT algorithm. It follows the direct definition of DFT. For a size n DFT problem, Generic DFT works n times on a size $2n$ complex input/output data and accesses n^2 complex twiddle factors. Figure 1 (b) shows L1 and L2 cache misses and runtime over n^2 of Generic DFT on Intel Xeon for different problem size n. As indicated in the analysis of Generic DFT and verified in Figure 1 (b), all major arithmetic operations and memory accesses increase quadratically with n.

With the above observation, our fractional abstract machine model is simplified to a fractional quadratic model: $t = q * n^2$. Quadratic coefficient q is detected separately on three regions. q is obtained using regression when memory footprints is smaller than half or larger than four times of L2 cache. Interpolation method is used for transition part around L2 cache size. Actually, our fractional performance model is an overkill for the application of Generic DFT in FFTW. Because of the algorithm's quadratic complexity, it can only be beneficial for small problem sizes in which case all data is in L2 cache. FFTW hardcodes to constrain the applicability of Generic DFT only for problem size $n <= 173$. Although the first segment of our fractional quadratic model is enough for FFTW, we still address the full version of this Generic DFT performance prediction model for completeness.

3.2 Performance Model for Codelets

As described before, codelets are highly optimized straight-line style C code for DFTs of small sizes or sizes of small factor. There are primarily two kinds of codelets. One is direct codelets(n codelet) that solve size n DFT directly and the other is twiddle codelets(t codelet) that solves DFTs of size $n * m$ following Cooley-Tukey. FFTW version 3.2.1 includes codelets for size 2-16,20,25,32,64. A direct codelet and each of the m iteration of a twiddle codelets have a fixed number of instructions. Unlike other FFT algorithms, which are more likely to be implemented on high level of a DFT decomposition and have smaller strides, codelets can be implemented at any level of decomposition and may have large strides because of Cooley-Tukey's shuffle effect.

Figure 2 (a) shows the performance of direct codelet n1_25 with different strides. We can make a couple of observations from the figure. Firstly, with the increase of stride, the runtime of the codelet increase rapidly around the strides of 2000-30000. In this region, the memory footprint of the codelet are roughly from half to four times of L2 cache. Similar to other FFT algorithms kernels, the performance of codelets decreases in this L2 transition region, as the ratio of L2 cache misses per memory access increases. Performance of codelets are relatively stable for regions where memory footprint is smaller than half of L2 cache or larger than four times of L2 cache because of relatively fixed ratio of L1/L2 cache misses ratio per memory access. The second observation is that the codelet with power-of-two strides performs up to 100% worse than with other strides of similar sizes. This is not hard to understand because all cache feature sizes are power-of-two and such a stride will easily result in extra conflict cache

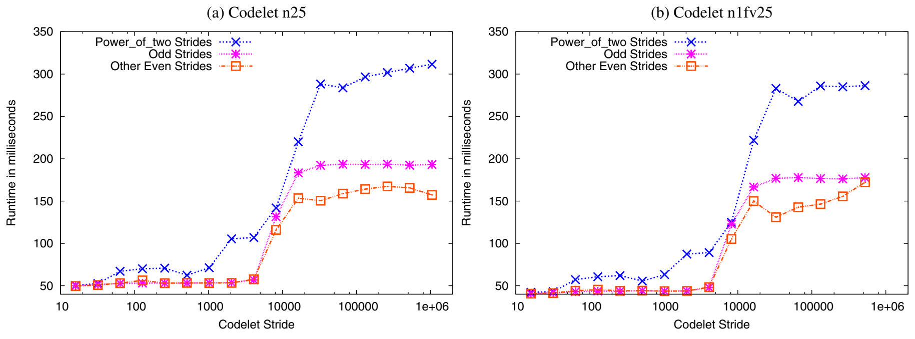


Fig. 2. Runtime of two codelets with different strides on Xeon

misses. This effect is not obvious for stride regions where memory footprint is smaller than L1 cache. Power-of-two strides that are smaller than L1 will not be mapped to the same L1 cache set causing less conflict misses and the cost of L1 cache misses is relative small. Finally, the even but non-power-two strides performance better than similar odd strides, this is probably because even stride has better data alignment resulting in less number of physical memory access in different memory levels.

Besides what we discussed above, strides with large power-of-two factors also affect the performance of codelets. For example, if $strideA = n = L2\_size/4$, where n is power-of-two, every other 4 elements accessed will be mapped to the same L2 cache set. If $strideB = 5 * n/4$, then $strideB$ 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 (2).

$$T = T_e + \alpha * (T_p - T_e), \tag{2}$$

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.

3.3 Performance Model for SIMD Codelets

The current version of FFTW, fftw-3.2.1, supports SIMD instruction extensions such as SSE, SSE2 and Altivec. FFTW takes advantage of SIMD instructions at the codelet level by including direct and twiddle SIMD codelets. FFTW uses two implementation schemes, SIMD with vector length of two and SIMD with vector length of four. SIMD with vector length of four takes advantage of parallelism between different iterations of the 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, i.e. $DFT(A + iB) = DFT(A) + iDFT(B)$, and is applicable to more problems. The input/output data in memory is required to be aligned correctly for SIMD instructions before computation.

Figure 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.

4 Model-Driven Optimization Engine for FFTW

In this section, we describe the training of the performance prediction models and the replacement of the original empirical search engine in FFTW with our model-driven optimization engine.

4.1 Training of DFT Performance Models

The integration of the model-driven optimization engine begins with the training of performance models for a specific processor so that the values of parameters depending on architectural features can be determined. The training is done only once for a computer when each solver is registered into FFTW planner during installation. We measure the performance of each solver for different sizes or strides and get the model parameters using linear regression.

For the abstract machine model of each FFT algorithm kernel, we train on 10-20 randomly selected sizes. 10-20 is more than the number of independent parameters in the model because some numbers of instructions are linearly dependent, e.g. $2n$ multiplications and $3n$ additions in Cooley-Turkey algorithm, which decreases the degrees of freedom of the model. For each training size, we estimate the number of different instructions and measure the execution time of the algorithm kernel. Then, we use weighted linear least square regression to determine the optimal model parameters. 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 (3) is minimized for the optimal solution.

$$\sum_{j=1}^{m} \left| \frac{\sum_{i=1}^{n} C_{A,i,j} P_{M,i} - T_{A,M,j}}{T_{A,M,j}} \right|^2 \tag{3}$$

A solution to this weighted linear least square problem is given the matrix form in (4), where W is a diagonal matrix and $w_{i,i} = \frac{1}{T_{A,M,j}^2}$

$$\boldsymbol{P} = (C^T W C)^{-1} C^T W \boldsymbol{T} \tag{4}$$

For the fractional quadratic model of Generic DFT and codelet performance model, similar weighted linear least square regression is used for different size or stride segments of the model that are dominated by L1 or L2 misses. Only one parameter, i.e. the quadratic coefficient q_n or the runtime, is extracted in each case from the regression. Linear interpolation of q_n or runtime is used in the transition segment of each model where the regression does not apply. One last thing to mention is the variance of measurements between multiple executions of a kernel or a codelet. The variance is typically less than 5% of the total runtime and we minimize this effect by repeat each measurement several times and pick the minimum. The training of all models typically takes just several seconds on each experiment platform we use.

4.2 Replacement of the Original Empirical Search Engine

Our model-driven optimization engine still follows the workflow of recursive search in FFTW. Only the performance measurement part is replaced by performance prediction using our models. For each solver, given the specific DFT size and stride, an estimated cost is generated by performance models. FFT solvers having child DFT problems will return a sum of its kernel performance prediction and that of the child problems which is obtained recursively. One thing to note is that solvers that are derived from the same FFT algorithm but with different implementation details, like buffered or non-buffered, will have separate models. Like the original empirical search engine, dynamic programing technique is still used to reduce redundant performance prediction and solution search. By the end of the search, a plan with the least predicted cost is returned as the optimum.

FFTW search engine has some internal search flags to constrain the search space. These flags are mapped from user interface patience levels, namely *Exhaustive*, *Patient*, *Measure* and *Estimate*[9]. *Exhaustive* mode traverses the whole 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.

5 Evaluation

In this section, we describe the results of our experiment by comparing FFTW version 3.2.1 with a modified version using our model-driven search engine. 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 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. Furthermore, the best DFT plan's performance for different sizes are compared among different search strategies. Finally, the search time spent by different strategies for different DFT sizes are compared.

5.1 Performance Prediction of Individual Algorithms

In this section, we show the accuracy of the fractional abstract machine model, fractional quadratic model and codelet performance model by comparing the

Table 1. Test Platforms Configuration

	Athlon64 X2	**Xeon 5405**	**PowerPC 970**	**UltraSparc IIIi**
Frequency	2 GHz	2 GHz	2.3 GHz	1.06 GHz
L1 Data Cache	64 KB	64 KB	32 KB	64 KB
L2 Cache	1 MB	6 MB	512KB	1 MB
OS	linux 2.6.24	linux 2.6.23	linux 2.6.24	SunOS 5.10
SIMD	3DNow!(not supported)	SSE2	Altivec	N.A.

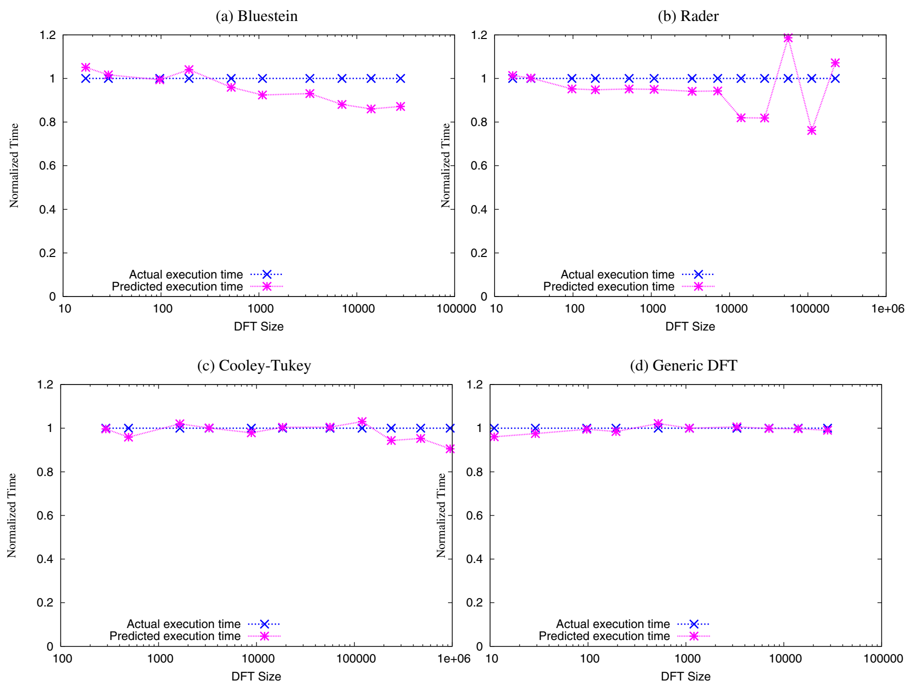


Fig. 3. Performance prediction of individual algorithms

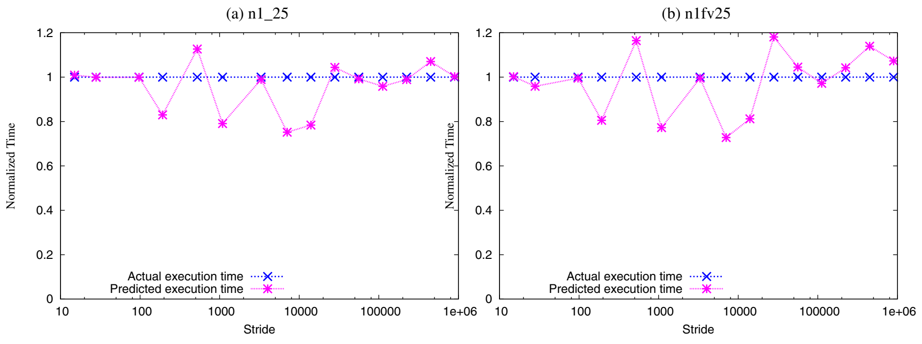
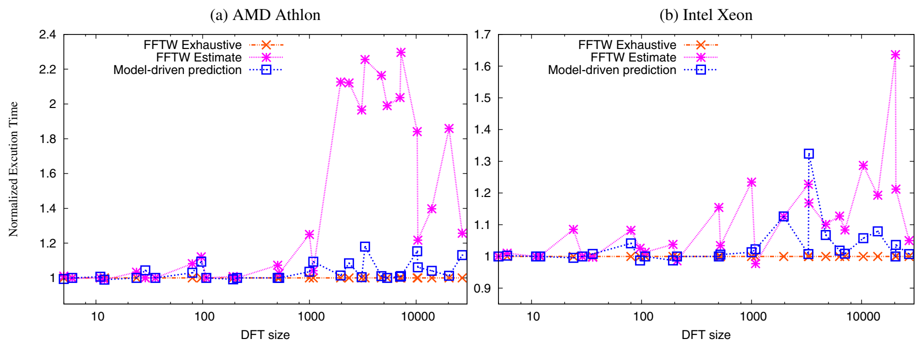


Fig. 4. Performance prediction of Scalar and SIMD Codelets

actual runtime and the predicted runtime of the Bluestein , Rader, Cooley-Tukey, the Generic DFT and scalar/SIMD codelets. Figure 3 shows the performance prediction for the four algorithms on Xeon. All predicted runtime is normalized with respect to the corresponding actual runtime. We compare the performance prediction for DFTs and codelets with sizes/strides up to 10^6. From the plots we can see these performance prediction models are very accurate. On Xeon, the average relative errors are less than 10% and similar results are got from other platforms. Figure 4 shows the performance prediction for scalar and SIMD codelets on Xeon with different strides. Generally, we have about 10% average prediction error and the worst is around 25%. As we will show later, the accuracy of our prediction model is good enough to distinguish *good* from *bad* plans in most cases.

5.2 Overall DFT Plan Performance

Performance comparison is made among our model-driven optimization engine, FFTW *Exhaustive* and *Estimate* mode. We compare the runtime of the best DFT plans found by these optimization methods. Because we only care about relative performance, all runtime is normalized with respect to the runtime of *Exhaustive* mode. Figure 5 shows the performance comparison among different DFT plans on four test architectures with SIMD disabled. All three search engines 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. 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 complex DFTs.



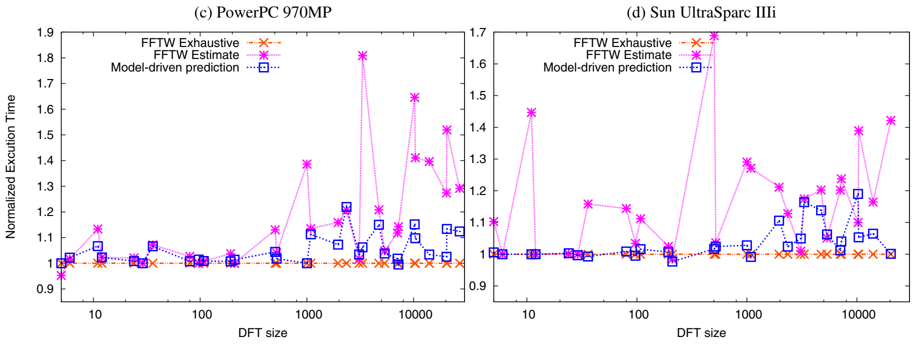


Fig. 5. Scalar performance comparison among FFTW Exhaustive, Estimate mode and model-driven prediction

Figure 6 shows the performance of the three search strategies 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

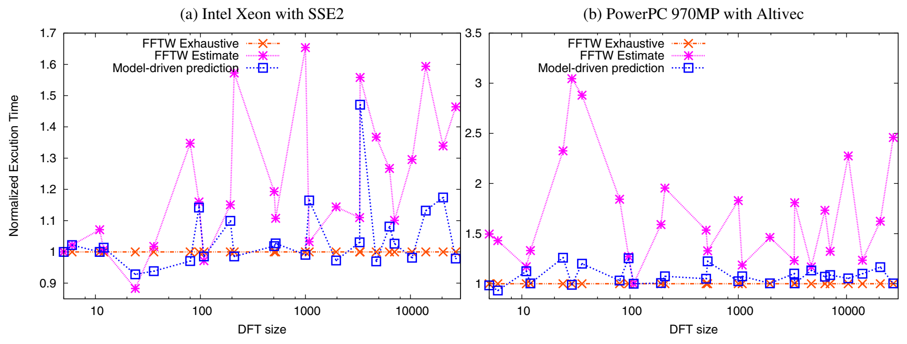


Fig. 6. SIMD performance comparison among FFTW Exhaustive, Estimate mode and model-driven prediction

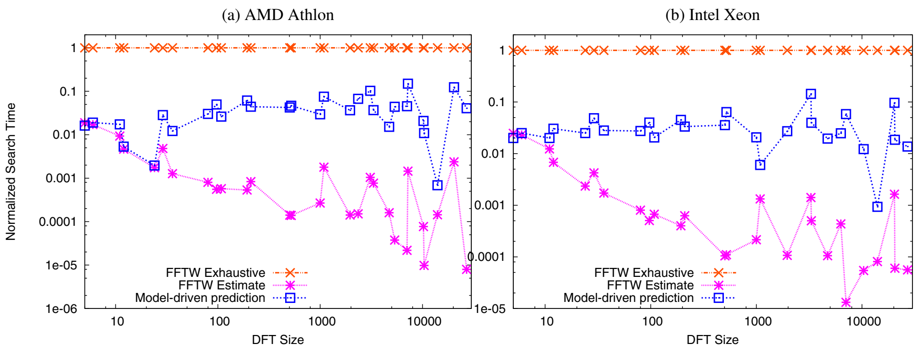


Fig. 7. The comparison of optimization time using FFTW Exhaustive, Estimate and model-driven optimization

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.

5.3 Optimization Time

In this section, we compare the search time that the three optimization engines use to find the best DFT plan. Again, the time is normalized with respect to the search time of *Exhaustive* mode. This search time excludes the initializing and training time of our model, which is about several seconds and spent only once for each platform instead of each DFT problem.

Figure 7 shows the search time spent by using model-driven, *Exhaustive* and *Estimate* optimization engines on two architectures. For limited space, we do not show the similar results on the other two platforms. Our model-driven optimization engine spends only about 0.1% to 10% of the time spent by *Exhaustive* mode. On the other hand, compared with the *Estimate* mode, our model-driven engine spends about 100 times more on average. One of the main reasons is that the *Estimate* engine runs on a much smaller search space than the *Exhaustive* search space, which our optimization engine needs to walk through. However, the absolute value of the search time using our search strategy is small and is within several seconds even for extremely large DFT sizes.

In summary, our model-driven search engine achieves about 94% of the performance of *Exhaustive* search engine and uses only less than 5% of its search time. Our 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.

6 Conclusion

In this paper, we propose a model-driven optimization engine for FFTW. This optimization has successfully reduced the empirical search time by developing performance prediction models for several DFT algorithms and codelets used in

FFTW and integrating them into a model-driven search engine. The most important conclusion we can draw from this work is that model-driven optimization can be effectively applied to a complex problem such as the generation of highly efficient implementation of DFT. This work also provides insight on why a DFT solution found by FFTW is the fastest by breaking down the overall performance into some architectural-dependent components. Besides that, reducing searching time means more for FFTW than other libraries, because search time is spent on every DFT size and every runtime in FFTW. Our model-driven optimization engine achieves more than 95% of the performance of the code found by exhaustive search while using less than 5% of the optimization time.

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 solvers 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.

References

1. Discussion with Franz Franchetti (2009)
2. Bluestein, L.: A linear Filtering approach to the computation of discrete Fourier transform. IEEE Transactions on Audio and Electroacoustics 18(4), 451–455 (1970)
3. Chen, C., Chame, J., et al.: Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In: Proceedings of CGO, Washington, DC, USA, pp. 111–122. IEEE Computer Society, Los Alamitos (2005)
4. Cooley, J.W., Tukey, J.W.: An algorithm for the machine computation of complex Fourier series. Mathematics of Computation 19(90), 297–301 (1965)
5. Duhamel, P., Vetterli, M.: Fast fourier transforms: a tutorial review and a state of the art. Signal Processing 19(4), 259–299 (1990)
6. Fraguela, B.B., Voronenko, Y., et al.: Automatic tuning of discrete fourier transforms driven by analytical modeling. To appear in PACT (2009)
7. Frigo, M.: A fast Fourier transform compiler. ACM SIGPLAN Notices 34(5), 169–180 (1999)
8. Frigo, M., Johnson, S.G.: The Fastest Fourier Transform in the West (1997)
9. Frigo, M., Johnson, S.G.: FFTW manual version 3.1–The Fastest Fourier Transform in the West. Massachusetts Institute of Technology, Massachusetts (2004)
10. Frigo, M., Johnson, S.G.: The design and implementation of fftw3. Proceeding of the IEEE 93(2), 216–231 (2005)
11. Im, E.-J.: Optimizing the performance of sparse matrix-vector multiplication. PhD thesis (2000); Chair-Katherine A. Yelick
12. Kulkarniand, P.A., Whalley, D.B., et al.: In search of near-optimal optimization phase orderings. SIGPLAN Not. 41(7), 83–92 (2006)
13. Oppenheim, A.V., Schafer, R.W., et al.: Discrete-Time Signal Processing (1999)
14. Püschel, M., Moura, J.M.F., et al.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation 93(2), 232–275 (2005)
15. Rader, C.M.: Discrete Fourier transforms when the number of data samples is prime. Proceedings of the IEEE 56(6), 1107–1108 (1968)

16. Saad, Y.: Research Institute for Advanced Computer Science (US). Sparskit: A Basic Tool Kit for Sparse Matrix Computation (1994)
17. Saavedra, R.H., Smith, A.J.: Analysis of benchmark characteristics and benchmark performance prediction. ACM Transactions on Computer Systems (TOCS) 14(4), 344–384 (1996)
18. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing 27(1-2), 3–35 (2001)
19. Yotov, K., Li, X., et al.: Is Search Really Necessary to Generate High-Performance BLAS? Proceedings of the IEEE 93(2), 358–386 (2005)