Dynamic Data Layouts for Cache-conscious Factorization of DFT*

Neungsoo Park, Dongsoo Kang, Kiran Bondalapati and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089-2562, USA.
{nspark,dskang,kiran,prasanna}@ceng.usc.edu

Abstract

Effective utilization of cache memories is a key factor in achieving high performance in computing the Discrete Fourier Transform (DFT). Most optimization techniques for computing the DFT rely on either modifying the computation and data access order or exploiting low level platform specific details, while keeping the data layout in memory static. *In this paper, we propose a high level optimization technique,* dynamic data layout *(DDL). In DDL, data reorganization is performed* between *computations to effectively utilize the cache. This cache-conscious factorization of the DFT including the data reorganization steps is automatically computed by using efficient techniques in our approach. An analytical model of the cache miss pattern is utilized to predict the performance and explore the search space of factorizations. Our technique results in up to a factor of 4 improvement over standard FFT implementations and up to 33% improvement over other optimization techniques such as copying on SUN UltraSPARC-II, DEC Alpha and Intel Pentium III.*

1. Introduction

Discrete Fourier Transform (DFT) is an integral component of many signal processing applications on several classes of platforms. It is one of the key stages in embedded signal processing applications such as Automatic Target Recognition, Synthetic Aperture Radar, Space-Time Adaptive Processing etc. [9, 11]. Algorithmic techniques for efficient computation of DFT have been explored on different architectural platforms [5, 14]. Most of these algorithms have been developed to efficiently compute the DFT by reducing the number of floating point operations. In current architectures, the effective bandwidth between the memory hierarchy and the processor is the bottleneck for achieving high performance. The effective memory bandwidth is improved by using cache memories which exploit the spatial and temporal locality in application data.

The data access pattern of a computation severely impacts the performance of the memory hierarchy [2]. DFT, like many other scientific applications, exhibits data access patterns which do not possess spatial locality. The number of memory words between successive data accesses is called the stride. Computation of DFT using factorization such as Cooley-Tukey [4], results in non-unit stride in accessing the data in the computation of the Fast Fourier Transform (FFT). This results in significantly more cache misses than accessing data with unit stride, reducing the effective memory bandwidth. There have been various approaches to improve the cache memory performance for such access patterns. Most of these techniques attempt to modify the computation and the data access order to improve the spatial and temporal locality. In such approaches, the data layout in memory is *static*, i.e. it does not change during the computation.

In this paper we develop the *dynamic data layout* (DDL) technique to dynamically reorganize the data during the computation. Cache-conscious algorithmic design techniques are utilized to determine a factorization of the DFT and the data layout to make effective use of the cache. The data layout is modified between the computation stages of FFT to improve the effective bandwidth. We show that modifying the data layout *dynamically* yields performance benefits greater than the data reorganization overhead. Analytical modeling of the memory hierarchy behavior is utilized to predict the performance of an FFT computation stage. The analytical model is validated by performing experiments using the cache simulator in SUN's Shade software package [13]. The factorization of the FFT including the data reorganization steps is automatically computed by using efficient techniques in our approach.

We achieve up to a factor of 4 improvement over straightforward implementations and up to 33% improvement over

*Supported by the DARPA/DSO OPAL Program through the Carnegie Mellon University under subcontract number 1-541704-50296.

FFT using other optimizations such *copying* on several platforms including SUN UltraSPARC-II, DEC Alpha and Intel Pentium III. We also show that the *static* approach, with a fixed data layout, yields factorizations which are not optimal and alternate factorizations with *dynamic data layout* result in higher performance. Our cache-conscious design approach yields a portable high performance FFT which is competitive with other high performance implementations without using any low level optimizations. DDL is a high level optimization based on data reorganization and not on platform specific algorithm or computation restructuring.

In Section 2 we describe previous approaches to improving the performance of memory hierarchy in computing the FFT and identify the shortcomings of these approaches in effectively utilizing available memory bandwidth. Section 3 provides the motivation for *dynamic data layout* based on an analytical model of the cache behavior of FFT computation. We describe our *dynamic data layout* approach and determination of the optimal factorization in detail in Section 4. The performance improvements obtained using our approach are illustrated in Section 5. Section 6 draws conclusions and gives an overview of the SPIRAL project [12] framework which encapsulates our approach.

2. Related Work

Optimizing memory hierarchy performance has been widely studied for computations in general [10], and FFTs in particular [1, 6, 7, 16]. General techniques for improving memory hierarchy performance attempt to improve the spatial and temporal locality. These techniques fall into two classes: access reordering and *static* data layout modification(i.e. before the computation begins). FFT performance has been optimized on various platforms by exploiting low level architectural and compiler features.

2.1. Memory Access Optimizations

Various manual and automated memory access optimization techniques have been developed for uniprocessors and parallel systems [8, 10, 15]. Access reordering involves modification of the computation to change the order in which data is accessed. Compiler techniques such as blocking (tiling), loop interchange and copying improve the spatial and temporal locality properties of the data being accessed. *Copying* optimization [10] utilizes temporary arrays into which the original array data is copied. This temporary array exhibits high spatial and temporal locality. But, the improvement in temporal locality due to *copying* does not alleviate the access overheads in constructing the temporary array. The proposed *dynamic data layout* is a global optimization which reduces the actual data access overheads. The second class of techniques involve static modification of data organization in memory such as nonlinear data layouts [3]. In DFT computation using several stages, a single data layout is not necessarily optimal for all the computation stages. For large size DFTs, the data is accessed with different strides in different stages. Any static data layout results in large number of cache misses for a high percentage of these strides.

2.2. FFT Performance Optimizations

Improving the performance of FFT by using various optimization techniques has also been considered by various researchers [1, 6, 7, 16]. The MIT FFTW project utilizes composition of code for FFTs of small sizes called codelets to compute FFTs of larger sizes [6]. The codelets are special pieces of code optimized based on various low level techniques. We show that our approach can achieve performance improvement without low level optimizations. The codelets are executed on a specific target to measure the performance of various optimizations. The larger size FFT is then composed using the codelets and the optimizations which perform best on the given target architecture. *This approach assumes that all FFTs of the same size have the same performance*. As we show in Section 3, for the same data size, FFT performance also depends on the data access pattern. Our high level optimization approach can potentially be integrated with the low level optimizations of FFTW to yield higher performance.

Bailey [1] and Gannon et. al [7] have studied the performance of FFTs in external memory to develop techniques for computing the FFT on vector and parallel computers. The six-step approach by Bailey attempts to reduce the number of accesses to the external memory or the Solid State Disk (SSD) by restructuring the computation and performing efficient transpose of the matrix. Wadleigh [16] developed a seven-step approach enhancing the six-step approach by performing cache-based optimizations. The number of cache misses are reduced by blocking the computation and the transpose operations. Gannon et. al. study the performance of a shared memory hierarchy in vector multi-processors using an algebraic framework. They use an approach similar to the copying compiler optimization. These approaches are based on optimizing the memory performance on vector and parallel computers. Though the techniques we present are analogous to those developed by Bailey, the problem we consider is at a different level in the memory hierarchy. In this paper, we develop techniques to optimize FFT performance on a uniprocessor rather than a vector or parallel processor. We develop an approach to *automatically* compute the factorization of a DFT based on the cache parameters to minimize the total execution time of the DFT including the data reorganization overheads.

3. Cache Behavior for Factorized DFT Computations

To study the performance characteristics of DFT factorizations we analyzed the cache behavior of DFT computations with factorization. We consider a processor with a two level memory hierarchy consisting of cache memory and main memory. The sizes of all parameters are measured in terms of the number of data points. In our FFT implementation, the size of each data point is either 16 or 8 bytes(double or single precision complex number). M denotes the size of the main memory, C denotes the size of the cache memory and B denotes the size of the cache block. The cache is assumed to implement a *write-back* policy on a write miss. We assume the cache is direct mapped but show that the analysis is similar for any k-way set associative cache. The number of cache blocks in a direct mapped cache is given by C/B. An N-point FFT with a data access stride of S is represented as $FFT(N, S)$. In this paper, we assume that all parameters are powers of 2. Figure 2 illustrates some of the parameters involved in the computation of the FFT.

A data access can result in a cache hit or a miss. A cache miss can occur due to various reasons. A *compulsory* miss occurs if the data was never accessed before and needs to be fetched for the first time. A *conflict* miss occurs when the data item was previously fetched into the cache block but was replaced because another data access was mapped to the same cache block. In FFT execution, once the data elements are read into the cache, the writes are into the same cache blocks. Hence, the cost incurred in write back caches due to write back to memory is very low compared to the cost of read miss memory accesses.

3.1. Cache Behavior Analysis

Factorization of DFT reduces the number of floating point operations to be performed. In addition, factorization reduces the size of each FFT so that the data required for one FFT fits in the cache. Consider the Cooley-Tukey factorization of an N-point DFT as $N_1 \times N_2$ where $N_1 < C$ and $N_2 < C$. First, N_2 N_1-point FFTs are performed and then N_1 N_2-point FFTs are performed. If the size of each FFT is less than the cache size, the required data can potentially fit in the cache and each FFT can have good cache performance. But, in the computation of the N_1-point FFT, the data is accessed at a stride greater than 1. The graph in Figure 1 shows the execution time on SUN UltraSPARC I for a single 32-point FFT with various data access strides. For strides larger than 2^{10}, the execution time is much higher than for unit stride.

Consider the computation of N_1-point FFT with stride S, $FFT(N_1, S)$. The cache behavior for performing two consecutive FFT computations is illustrated in Figure 2.

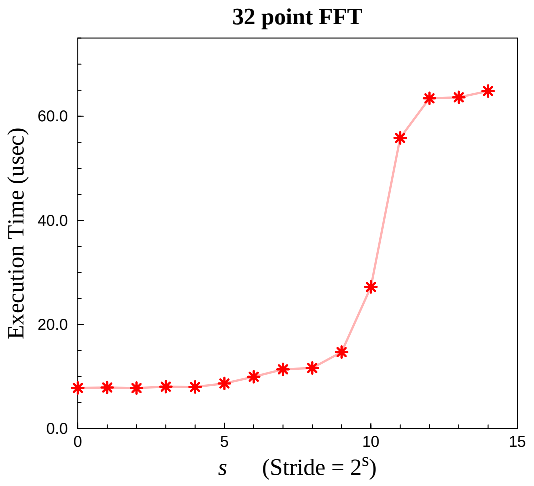


Figure 1. Effect of stride data access on execution time of FFT on UltraSPARC-I

Case I: $S = 1$
There is a cache miss every time a new cache block is fetched. The next $B-1$ accesses are cache hits. Since all the data maps to contiguous cache blocks and fits in the cache, there are no conflict misses. Computing multiple N_1-point FFTs results in N_1/B cache misses for each FFT.

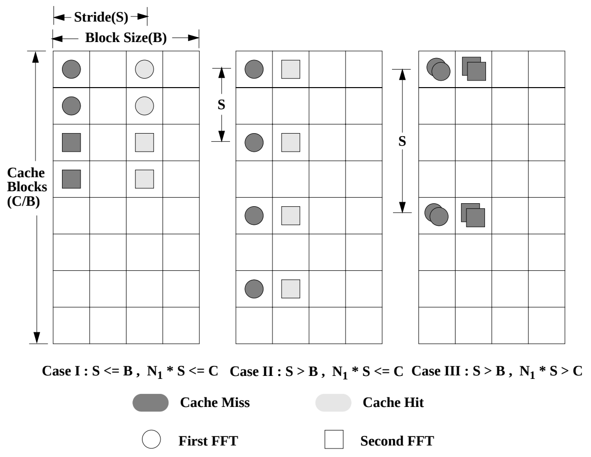

Stride(S)
Block Size(B)
Cache Blocks (C/B)
S
S
Case I : S <= B , N1 * S <= C
Case II : S > B , N1 * S <= C
Case III : S > B , N1 * S > C
Cache Miss
Cache Hit
First FFT
Second FFT

Figure 2. Cache behavior for two consecutive FFTs(N_1=4,B=4,C=32) (I)S=2 (II)S=8 (III)S=16

Case II: $S > 1$ and $N_1 \times S \leq C$
When the stride is greater than 1, only a fraction of the data elements fetched into each cache block are useful. The total number of compulsory cache misses is given by $min(N_1 \times S/B, N_1)$. As shown in Figure 2 there are no conflict misses since all the elements map to distinct cache blocks in the cache. When consecutive N_1-point FFTs are performed, the data elements accessed by successive FFTs are contiguous in cache. For stride $S > B$, only every B^{th} FFT results in cache misses. The next $B - 1$ FFTs can be computed with

cache hits for all required data points.
Case III: $S > 1$ and $N_1 \times S > C$
Even if $N_1 < C$, when the stride is large enough so that $N_1 \times S > C$, there will be conflict misses in addition to compulsory misses. These conflict misses have two significant effects on the performance. As shown in Figure 2, there can be conflict misses even in the computation of a single N_1-point FFT. There can be potentially $N_1 \log N_1$ cache misses in the $\log N_1$ computation stages of a single N_1-point FFT. The spatial reuse by subsequent FFTs as explained in *Case II* above is also lost due to the conflict misses. This results in each FFT having up to $N_1 \log N_1$ cache misses which is the total number of data accesses for this FFT stage.

A k-way set associative cache will have similar behavior. The number of cache blocks in a k-way set associative cache is $1/k$ times the number of blocks in a direct mapped cache. The effective block size is larger for the k-way set associative cache. Since the access pattern of FFT is regular and periodic, the number of conflict misses is the same for smaller number of cache blocks with a larger effective block size.

3.2. Effect of Stride on Performance

The behavior observed in Figure 1 can be explained based on the above analysis. The execution time increases significantly for stride greater than 2^{10}(e.g. $2^5 \times 2^{11}(N \times S) > 2^{15}(C)$). Since a small part of cache is occupied by instructions and other cached data, the effective cache size is slightly smaller than 2^{15} data points. We see this effect resulting in increase in execution time even for stride of 2^{10}. Non-unit stride memory access can potentially increase the cache misses by a factor of up to $\log N_1 \times B$. Such large strided access is very common in factorized FFT computation. In the above case of $N_1 \times N_2$ factorization, the stride in computing the N_1-point FFTs is N_2. Other optimization approaches such as FFTW develop high performance kernels for FFTs of various sizes. The performance of such routines is significantly higher than standard implementations. But, these optimization techniques can not avoid the performance degradation caused by non-unit stride accesses. High level cache conscious optimizations are required to improve the performance.

The large performance degradation caused by such non-unit stride accesses suggests that data reorganization to reduce the length of the stride can provide significant performance speed-ups. The analytical model of cache behavior analysis provides a framework for predicting the performance of a given FFT computation based on the factorization. The data access stride is based on the factorization and independent of implementation. It is possible to predict the memory access performance of an FFT with a given size and factorization. The above analysis also indicates the conditions when dynamic data reorganization is beneficial. This can be utilized for pruning the search space in determining the optimal factorization for an FFT of given size.

4. Dynamic Data Layouts

Factorization such as Cooley-Tukey can be applied recursively to perform a given DFT. The resulting factorization can be represented by a factorization tree. Previous approaches assume that the cost of executing any node in the tree is dependent on the size of the node and is independent of any data access pattern(i.e. state of the memory). As illustrated in Section 3, data access pattern affects the performance. To improve performance, data reorganization can be performed at any node in the factorization tree. In this section we focus on optimal factorization of a DFT of a given size N. An optimal factorization has minimal total execution time including data access cost. Throughout this paper, we assume that N is a power of 2.

In *dynamic data layout* data reorganization is performed between computation stages to reduce the number of cache misses incurred during the computation. Data reorganization itself involves memory accesses which is an overhead. For dynamic data reorganization to be beneficial, this overhead should be lower than the performance gains that can be obtained.

4.1. Cache-conscious factorization of DFT

Performing dynamic data reorganization involves two data reorganization steps for each factorization step. Consider the factorization of an N-point FFT into $2^k \times \frac{N}{2^k}$. The reorganization steps are performed before the 2^k-point FFTs and before the $N/2^k$-point FFTs. The computation can be factorized recursively to yield a factorization tree. The data access strides resulting from the factorization at a node of size N, with a data access of stride S are shown in Figure 3. The cost of performing an optimal N-point FFT can be computed by the recursive equation:

$$\begin{aligned} FFT(N,S) \quad = \quad & \min_{k,s_1}[D(S \times \frac{N}{2^k}, s_1) + FFT(2^k, s_1) \times \frac{N}{2^k} + \\ & D(2^k * s_1, 1) + FFT(\frac{N}{2^k}, 1)] \\ & 1 \le k \le \log N \;\; and \;\; 1 \le s_1 \le \frac{N}{2^k} \end{aligned}$$

$D(s_i, s_j)$ denotes the cost of performing the data reorganization to convert a stride s_i access to stride s_j access. When $s_i = s_j$, no data reorganization is performed. When $s_i > 1$ and $s_j = 1$, a non-unit stride access is converted to a unit-stride access. We perform data reorganization using blocked data movement. The reorganization overhead is $O(\frac{N}{B})$ memory accesses for DFT data of size N, where B is the cache block size.

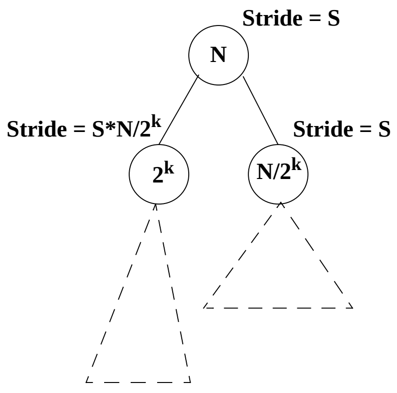


Figure 3. Strides resulting from Cooley-Tukey factorization

The solution space for finding the optimal factorization of an DFT computation is large. The search space includes all possible factorization trees and all possible strided accesses. Even with the assumption of all factorizations and strides being powers of two, the solution space is still large. The search space for optimal factorization including dynamic data reorganization can be reduced based on the following observations: (1) The factorization tree contains many nodes which are duplicated at several parts of the tree. These costs need not be computed every time but can be stored and utilized. (2) As shown in Figure 3, the stride of a node in the tree is dependent on the stride of its parent node and the size of the factor. This constraints the strides at various parts of the tree and reduces the search space. (3) The analysis in Section 3 illustrates that data reorganization is beneficial only when the product of the size of the FFT and the stride is greater than the cache size($N \times S > C$).

These properties help us search the space by using dynamic programming. The number of different solution points for sub-problems to be computed is given by the product of the number of possible sizes for the factors and the number of strides. The initial values to be computed are the costs for performing radix-2 FFTs of different sizes with different strides and the costs for performing data reorganizations between various strides. The iterative equation for computing the optimal factorization can be defined as:

$$\begin{aligned} F[i,j] &= \min_{p,q}\{D(2^j * \frac{2^i}{2^p}, 2^q) + F[p,q] + D(2^{p+q},1) \\ &\quad +F[i-p,1]\} \\ &\quad 1 \leq p \leq \log N \;\; and \;\; 0 \leq q < \log N \end{aligned}$$

$F[i,j]$ represents the optimal cost to compute an FFT of size 2^i with an input data stride of 2^j. The initial values of $F[i,j]$ represent computation of these FFTs by directly using radix-2 computation. It has to be noted however that the initial values can represent any optimized implementation of FFT(such as that in FFTW). This facilitates integration of low level optimized FFT libraries into our high level optimization technique.

In the i^{th} iteration, the cost of computing FFT of size 2^i with various factorizations and dynamic data reorganization options are considered. If we assume that optimal costs for all FFTs of sizes smaller than 2^i have been determined in iterations $1 \ldots i-1$, then the i^{th} iteration will determine the optimal cost of FFT of size 2^i for all strides 2^j, $0 \leq j \leq \log N$. When N is a power of 2, the various possible factorizations and strides are also powers of 2. The optimal cost includes the various possible Cooley-Tukey factorizations of different sizes for each step and the possible data reorganization options for each factorization step. The time complexity of the algorithm is $O(\log^3 N)$. The factorization computed using our technique provides a factorization with optimal execution time which includes the data reorganization cost.

4.2. Alternate Factorizations with Dynamic Data Layouts

Factorization can also be explored using only *static* data layout where the data layout does not vary during the computation. We can search the solution space with a fixed data layout without no dynamic data reorganization. After determining an optimal factorization with *static* data layout, we can perform data reorganization between the computation to see if any speed-up can be obtained. But, this solution might be sub-optimal since a different factorization integrating data reorganization in the search phase might have a lower execution cost. As shown in Table 1 we found experimentally that an alternate factorization for *dynamic data layout* can have a better performance than the best factorization for the *static* approach. The experiments were performed on an UltraSPARC-I with a clock speed of 143 MHz and cache size of 1MB(2^7 data points). The factorization for the 2^{19} size DFT illustrates that the order of factors for the same factorization also affects the execution time since it affects the data access stride during the computation.

Table 1. Alternate factorizations using dynamic data layouts

DFT Size	Factorization	Static Layout Exec. Time (ms)	Dynamic Layout Exec. Time (ms)
2^{17}	$2^9 \times 2^8$	472.543	397.167
	$2^7 \times 2^{10}$	498.440	381.202
2^{19}	$2^{10} \times 2^9$	2068.122	1703.501
	$2^9 \times 2^{10}$	2078.551	1692.592

5. Performance Results

In this section, we present simulation results as well as experimental results of the cache performance on various platforms. The memory hierarchy in typical state-of-the-art processors consists of split L1 caches (instruction and data cache), a unified L2 cache and a main memory. In our simulations and experiments, we focus on the L2 cache behavior. Three different approaches were evaluated on various platforms - *dynamic data layout* (DDL), static data layout with *copying* optimization (SDL) and the naive radix-2 FFT (Naive).

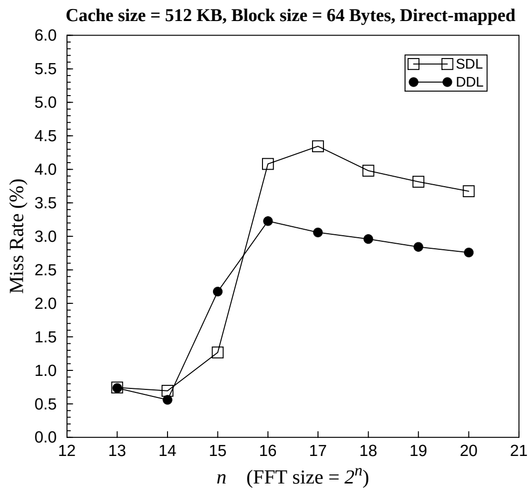


Figure 4. Miss rates for a fixed cache size with various FFT sizes

5.1. Cache Simulation Results

To validate our analysis of the cache performance in DDL and SDL approaches, we performed simulations using the cache simulator in the SUN Shade simulator [13]. We focus on the data cache misses since the number of instruction cache misses in L2 cache is very low(0.008%). The L2 cache is assumed to be direct mapped in our simulations. As shown in Section 3, the FFT performance with a direct-mapped cache is similar to the performance with a *k*-way set associative cache of the same total cache size(C). We studied two different aspects of the performance of the cache. First, we studied the cache miss rate for FFTs of different sizes with a fixed cache block size. Second, we performed simulations by varying the cache block size(B) for a fixed size FFT(N).

For FFT size larger than the cache size we observed that the proposed DDL approach has significantly better cache performance than SDL approach. The graphs in Figure 4 show the cache miss rate as the size of the FFT is varied. For small size FFTs, DDL has no significant benefit compared with SDL. When the size of the FFT is larger than the cache size, the miss rate of DDL is much lower than that of SDL. For a 512 KB cache, 2^{15} points can fit in the cache. As the number of FFT points increases from 2^{14} to 2^{16}, the

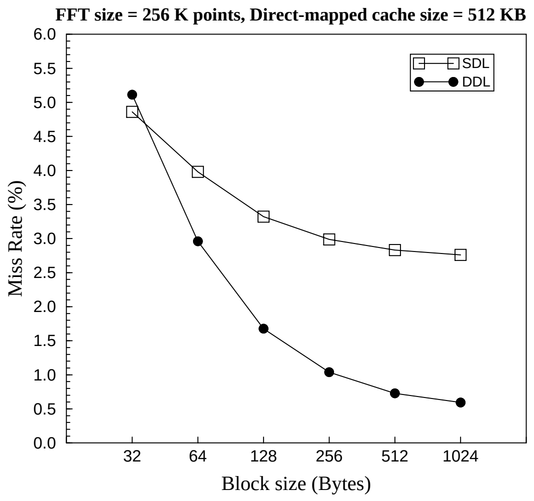


Figure 5. Miss rates for a single precision 256K-point FFT with various cache block sizes

miss rate increases significantly from 0.695% to 4.081% in SDL. However, the miss rate in DDL increases from 0.560% to 3.228% only. This corroborates our analysis in Section 3 that for FFTs of size larger than the cache size, DDL has significantly lower number of cache misses compared to SDL.

Figure 5 and Figure 6 show the cache miss rate as the cache block size is varied. When the cache block size is 64 bytes(same as the cache block size in SUN UltraSPARC-II), the cache miss rate is 3.98% in SDL and 2.96% in DDL. DDL has a 25% lower miss rate compared to SDL. As we expect from our analysis in Section 3, DDL approach utilizes larger block sizes more efficiently than SDL approach.

5.2. Experimental Results

We studied the benefits of the proposed DDL approach on three different platforms. Table 2 summarizes the relevant architectural parameters of various platforms used in our implementations. The graphs in Figures 7 to 12 show the performance achieved on different platforms in MFlops for FFTs of various sizes. The dotted vertical line in each figure indicates the point where the size of the FFT is same as the cache size.

The graphs in Figure 7 and Figure 8 demonstrate the performance on SUN UltraSPARC-II using double and single precision, respectively. When FFT size is larger than the cache size, DDL has significantly higher performance than SDL and Naive approach. For large FFTs, DDL is up to 33% faster than SDL when using single precision and up to 25% faster using double precision. Figure 9 and Figure 10 show the performance achieved on DEC Alpha using double and single precision, respectively. When the number of FFT points is 2^{20}, DDL is 38.9% faster than SDL using sin-

Table 2. Parameters of the platforms

Processor	SUN Ultra-II	DEC Alpha 21164	Intel Pentium III
Clock (MHz)	400	300	500
L2 Cache size (KB)	2048	96	512
L2 Cache block size (Bytes)	64	64	32
OS	Solaris 2.7.x	UNICOS/mk 2.0.4.48	Linux 2.2.5
Compiler	gcc 2.8.1	Cray Standard C Ver. 6.1.0.1	gcc egcs-2.91.66
Optimize option	-O3 -Wall	-O3	speed, pentium pro

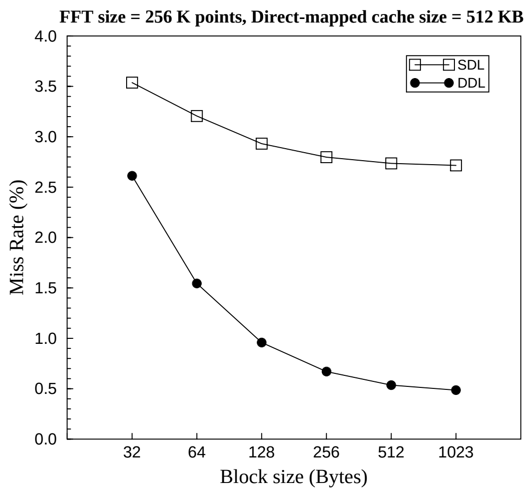


Figure 6. Miss rates for a double precision 256K-point FFT with various cache block sizes

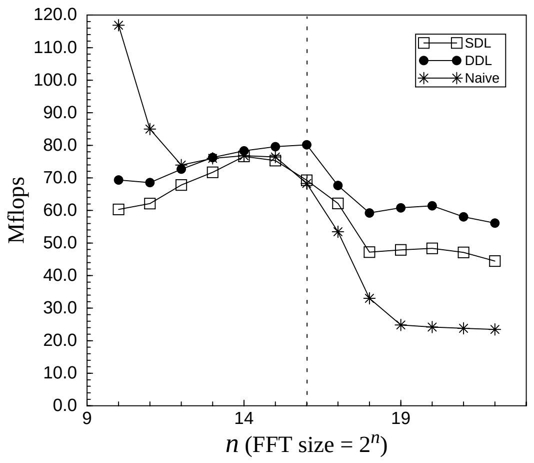


Figure 7. Sustained performance of SUN UltraSPARC-II(double precision)

gle precision and is 28.9% faster than SDL using double precision.

On the Pentium III, the performance improvement of DDL with single precision is up to 23% compared with SDL as shown in Figure 11. Using double precision, the performance of DDL is up to 10% higher than SDL(Figure 12). The performance improvement on the Pentium III is lower than other platforms. This lower performance benefit is as expected from our cache performance analysis, since Pentium III has a smaller cache block size(32 bytes) compared to the cache block size of UltraSPARC-II and Alpha(64 bytes).

In summary, the proposed DDL approach provides significant performance benefits compared to SDL and the Naive approaches on several different classes of architectures. DFT computation with *dynamic data layout* integrated into the computation runs around 25% faster on various platforms. Significantly, all the experiments were performed with the same source code in C with no platform specific optimizations. The only modification performed was in the time measurement function on different platforms. The SDL approach includes the *copying* software optimization technique. This substantiates our claim that DDL is a high level optimization which can benefit DFT computation on various platforms. We believe it is also possible to obtain performance benefits by using the proposed *dynamic data layout* approach even for DFT computation which has been highly optimized for a specific platform.

6. Conclusions

We proposed a new high level optimization technique, *dynamic data layout*, which performs dynamic data reorganization to improve cache performance. We developed efficient techniques to compute the data reorganization and FFT factorization based on cache-conscious algorithmic techniques. Our simulation and implementation results illustrate that our cache-conscious algorithmic techniques can achieve significant performance improvements. We showed a factor of 4 improvement over standard implementations of FFT and up to 33% improvement over other optimizations such as *copying*. On all the experimental platforms, DDL

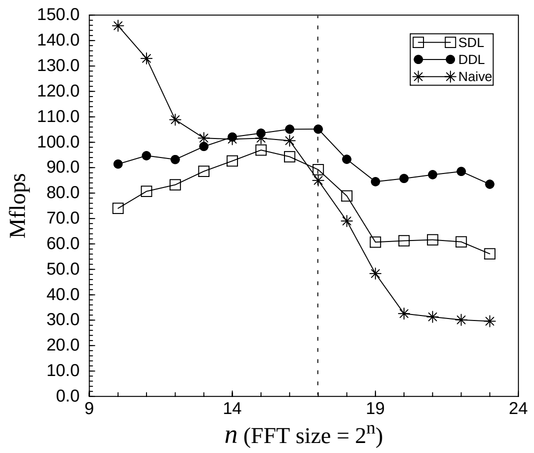


Figure 8. Sustained performance of SUN UltraSPARC-II(single precision)

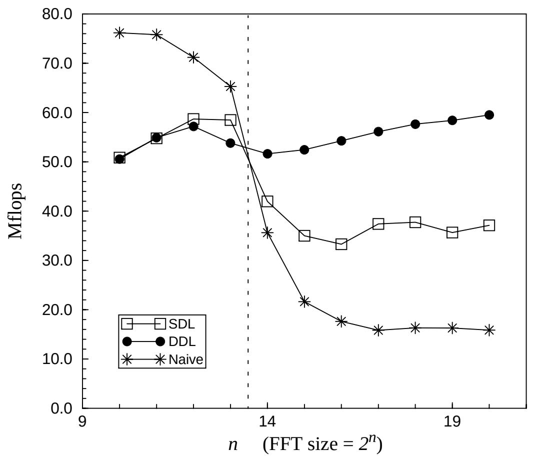


Figure 10. Sustained performance of Alpha 21164(single precision)

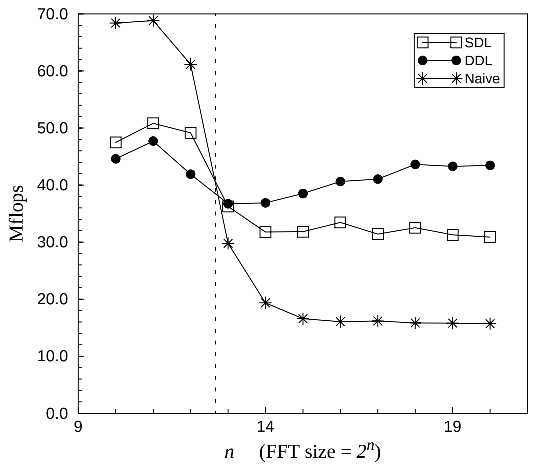


Figure 9. Sustained performance of Alpha 21164(double precision)

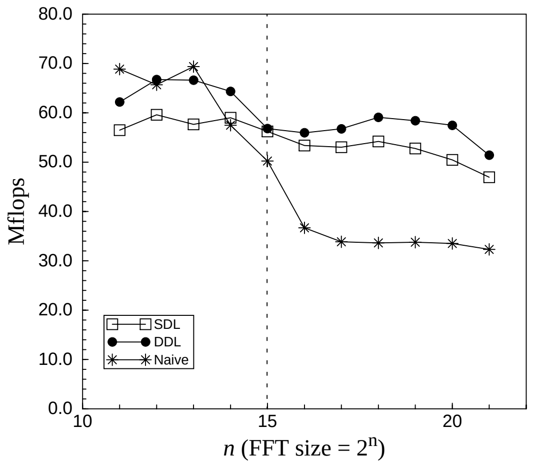


Figure 11. Sustained performance of Pentium III(double precision)

consistently achieves higher performance than SDL when the size of the FFT is larger than the cache size.

The work reported here is a component of the Signal Processing algorithms Implementation Research for Adaptable Libraries(SPIRAL) project [12]. An overview of the SPIRAL framework is given in Figure 13. The SPIRAL framework is being developed collaboratively by Carnegie Mellon University, Drexel, MathStar Inc., University of Illinois at Urbana Champaign and University of Southern California. The SPIRAL project is developing a unified framework for realization of portable high performance implementations of signal processing algorithms from a uniform representation of the algorithms. The cache performance models form a component of the high level models in the framework. Multi-level performance models and benchmarking data will be utilized to explore the algorithm and implementation search space using machine learning techniques.

References

[1] D. H. Bailey. FFTs in External or Hierarchical Memory. *Journal of Supercomputing*, 4, March 1990.

[2] D. H. Bailey. Unfavorable Strides in Cache Memory Systems. *Scientific Programming*, 4, 1995.

[3] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. *ACM International Conference on Supercomputing*, 1999.

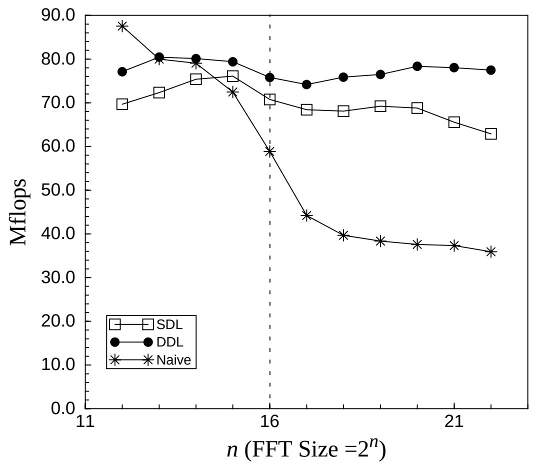
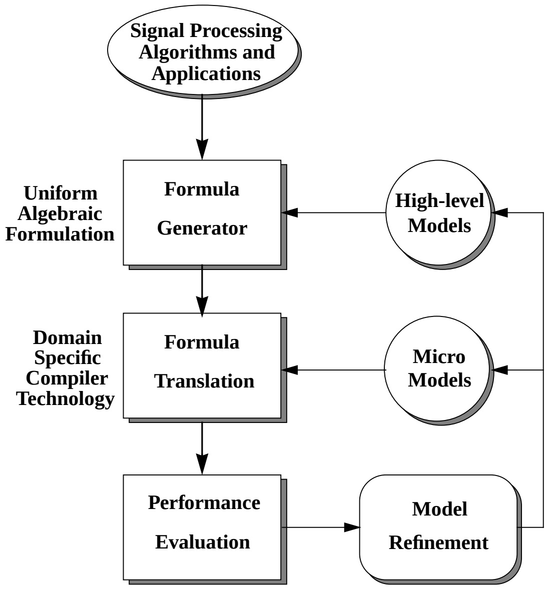


Figure 12. Sustained performance of Pentium III(single precision)



Signal Processing Algorithms and Applications
Uniform Algebraic Formulation
Formula Generator
High-level Models
Domain Specific Compiler Technology
Formula Translation
Micro Models
Performance Evaluation
Model Refinement

Figure 13. SPIRAL project overview

[4] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.*, 19, 1965.

[5] A. Dandalis and V. K. Prasanna. Fast parallel implementation of DFT using configurable devices. *International Workshop on Field Programmable Logic and Applications*, September 1997.

[6] M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. *ICASSP*, 3, 1998.

[7] D. Gannon and W. Jalby. The Influence of Memory Hierarchy on Algorithm Organization: Programming FFTs on a Vector Multiprocessor. *The Characteristics of Parallel Algorithms, The MIT Press*, 1987.

[8] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, December 1996.

[9] Embeddable Systems Homepage. www.ito.arpa.mil/ResearchAreas/Embeddable.html.

[10] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *ASPLOS IV*, April 1991.

[11] W. Liu and V. K. Prasanna. Utilizing the power of high-performance computing. *IEEE Signal Processing*, September 1998.

[12] SPIRAL Project. www.ece.cmu.edu/˜spiral/.

[13] SHADE Simulator. http://sw.sun.com/shade/.

[14] J. Suh and V. K. Prasanna. Parallel Implementations of Synthetic Aperture Radar on High Performance Computing PLatforms. *International Conference on Algorithms and Architectures for Parallel Processing*, December 1997.

[15] E. Torrie, M. Martonosi, C.-W. Tseng, and M. W. Hall. Characterizing the Memory Behavior of Compiler-Parallelized Applications. *IEEE Transactions on Parallel and Distributed Systems*, December 1996.

[16] K. R. Wadleigh. High Performance FFT Algorithms for Cache Coherent Multiprocessors.