

# Dynamic Data Layouts for Cache-Conscious Implementation of a Class of Signal Transforms

Neungsoo Park, *Member, IEEE*, and Viktor K. Prasanna, *Fellow, IEEE*

**Abstract**—Effective utilization of cache memories is a key factor in achieving high performance for computing large signal transforms. Nonunit stride access in the computation of large signal transforms results in poor cache performance, leading to severe degradation in the overall performance. In this paper, we develop a *cache-conscious* technique, called a *dynamic data layout*, to improve the performance of large signal transforms. In our approach, data reorganization is performed between computation stages to reduce cache misses. We develop an efficient search algorithm to determine an optimal tree with the minimum execution time among possible factorization trees based on the size of the signal transform and the data access stride. Our approach is applied to compute the fast Fourier transform (FFT) and the Walsh–Hadamard transform (WHT). Experiments were performed on Alpha 21264, MIPS R10000, UltraSPARC III, and Pentium 4. Experimental results show that our FFT and WHT achieve performance improvement of up to 3.52 times over other state-of-the-art FFT and WHT packages. The proposed optimization is portable across various platforms.

**Index Terms**—Cache-conscious, cache miss, dynamic data layout, FFT, memory hierarchy, signal transform.

## I. INTRODUCTION

RECENTLY, the characteristics of signal transform algorithms and the memory hierarchy in architectural platforms have been exploited in concert to improve the computational performance of signal transforms [1]–[4]. Large signal transforms can be decomposed into smaller transforms by taking advantage of their inherent *divide-and-conquer* property. The working set size of signal transform computations can be reduced to fit into the cache. Hence, the cache performance can implicitly be optimized by considering only the working set size. In such cache-oblivious algorithms [5], the details of the cache organization are not considered.

Using the cache oblivious algorithm approach, a high-performance fast Fourier transform (FFT) package was developed at the Massachusetts Institute of Technology (MIT), known as the FFTW [1]. Using a similar approach, a Walsh–Hadamard transform (WHT) package was developed at Carnegie Mellon Uni-

versity (CMU) [3]. These are flexible software architectures that exploit the divide-and-conquer property of signal transforms. The divide-and-conquer rule is represented in a tree structure. These tree-structured computations in FFTW and WHT packages attempt to modify the computation in order to improve the spatial and temporal locality. In such approaches, the data layout in memory is *static*,<sup>1</sup> i.e., it does not change during the computation. These tree-structured computations in static data layouts achieve good performance for small signal transforms. However, their performance drastically degrades for large signal transforms. This degradation is due to the cache organization in the memory hierarchy of the state-of-the-art architectures.

The performance of caches is severely impacted by the data access pattern of a computation and the cache organization [6]. In state-of-the-art architectures, the cache is either direct-mapped or small set-associative. During a computation, data in memory are successively accessed with a distance called *stride*. Large stride data accesses do not possess spatial locality. Furthermore, several elements can compete to occupy the same location in the cache because of its small associativity, thereby increasing cache misses. In the FFTW and WHT packages [1], [3], it is assumed that the performance depends only on the node size in the factorization tree. However, the performance degrades as stride increases, even though the problem size is fixed. Such large stride accesses are usually incurred in computing large signal transforms, resulting in drastic performance degradation. This degradation is due to considerable cache misses because of the small set-associative caches. Thus, cache organization is a significant factor that should be considered to achieve high performance.

To improve cache utilization, we propose a method called the *dynamic data layout* (DDL) approach, where the data layout in memory is dynamically reorganized during computation. After reorganizations, nonunit stride accesses are converted to unit stride accesses, thereby reducing cache misses. These data reorganizations between the computation stages of signal transforms improve the effective processor-memory bandwidth. We show that the overhead of dynamic data reorganization is smaller than its performance gain. To achieve high performance in computing a large signal transform, it is necessary to determine an optimal factorization and a data layout for that signal transform. We develop a search algorithm to find such an optimal factorization that automatically includes data reorganizations. The complexity of our search algorithm for an  $N$ -point DFT is  $O(kn^2)$ , where  $n = \log N$  and  $k$  different data layouts are considered. Note that this search algorithm is performed off line.

Manuscript received June 6, 2002; revised August 7, 2003. This work was supported by the DARPA/DSO OPAL Program through Carnegie Mellon University under subcontract 1-541704-50296, the National Science Foundation under award ACI-0204046, and by the faculty research fund of Konkuk University in 2003. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Chaitali Chakrabarti.

N. Park is with the Department of Computer Science and Engineering, Konkuk University, Konkuk University, Seoul 143-701, Korea (e-mail: neungsoo@konkuk.ac.kr).

V. K. Prasanna is with the Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089-2562 USA (e-mail: prasanna@halcyon.usc.edu).

Digital Object Identifier 10.1109/TSP.2004.828946

<sup>1</sup>In this paper, we call such an approach the *static data layout* (SDL) approach.

To verify our DDL approach, we performed simulations and experiments. Simulation results show that the DDL approach achieves up to 25% improvement in cache miss rate compared with the SDL approach. For experiments on real platforms, we modified the FFT and WHT packages developed by CMU [3], [7] by applying our DDL approach. Experiments were performed on SUN UltraSPARC III, Compaq Alpha 21264, Intel Pentium 4, and SGI MIPS R10000. The modified FFT using the DDL approach achieves up to  $2.54 \times$  performance improvement over the the CMU FFT. Compared with FFTW, it shows up to  $2.97 \times$  performance improvement. Our WHT package using the DDL approach shows up to  $3.52 \times$  performance improvement compared with the CMU WHT package. FFTW is not always optimal, but it achieves high performance on all platforms, close to the best-known performance in many cases. We chose FFTW for performance comparisons because it is the state-of-the-art software architecture for FFT and can be ported to available platforms.

The contribution of our work is a high-level optimization based on data reorganization, which can be applied on platforms where the problem size exceeds the cache size. Our optimization is applicable to the class of signal transforms that can be factorized. Dynamic programming determines when and what reorganizations are to be performed, for a factorized computation of a signal transform, by finding an optimal cache conscious factorization. Moreover, our approach is not platform-specific and yields portable high performance without using any low-level optimizations. Our solution is not always optimal when compared with hand-tuned, vendor-optimized FFT implementations.

The rest of the paper is organized as follows. In Section II, we describe previous approaches that improve the performance of memory hierarchy and optimize the performance of FFT and WHT. In Section III, we discuss the factorization of signal transforms and provide the motivation for dynamic data layout based on an analytical model of the cache behavior of a factorized signal transform computation. We describe our dynamic data layout approach and a cache-conscious factorization in Section IV. The performance improvements obtained using our approach are illustrated in Section V. Section VI draws conclusions and gives an overview of the SPIRAL project [8] framework, which encapsulates our approach.

In the rest of the paper, for the  $N$ -point FFT and WHT, we have assumed  $N$  to be a power of 2 for the sake of illustration. The Cooley–Tukey factorization approach used in this paper is applicable for any general  $N$ . Similarly, our dynamic programming approach does not require  $N$  to be a power of 2.

## II. RELATED WORK

In this section, we briefly discuss general optimization techniques for improving memory hierarchy performance. We then summarize the optimization techniques for FFT and WHT.

### A. Memory Hierarchy Performance Optimizations

To improve memory hierarchy performance, various manual and automated optimization techniques have been developed for uniprocessors and parallel systems [9]–[11]. To exploit the spatial and temporal locality [12] properties, the data access order

is changed by reordering the computation. This is called *control transformation*. These optimization techniques, such as tiling, loop fusion, loop interleaving, and loop interchange, enhance the temporal locality of data accesses. However, the stride to access data is not changed since reordering a computation does not change the data layout. In state-of-the-art platforms, large strides to access data result in cache conflict misses since cache is direct-mapped or small set-associative cache. To further improve the performance, techniques such as copying [10] and padding [13], [14] are used in concert with control transformation techniques. Although cache conflict misses are reduced by the *copying* optimization, the performance improvement resulting from copying cannot alleviate the overhead of copying if data is repeatedly copied from (to) memory to (from) the buffer during the computation. In the *padding* optimization, during computation, the overhead of the index computation needed to access the array is high since data elements are not stored contiguously.

Some recent work [15]–[17] has proposed changing the data layout to match the data access pattern of dense linear algebra applications. This is called *data transformation*. This reduces cache misses. In [15], conventional (row or column-major) layout is changed to a recursive data layout before the computation in order to match the access pattern of recursive algorithms. The ATLAS project [18] automatically tunes several linear algebra implementations. It uses block data layout with tiling to exploit temporal and spatial locality.

In [15] and [18], the data layout is reorganized before computation begins. However, the data layout is not changed until computation finishes. In signal transforms, as the computation proceeds, strides between consecutively accessed data increase. Therefore, a static data layout might not necessarily be optimal for all the computation stages. Similarly, copying and padding is difficult to apply to signal transform applications. Therefore, static data layout results in a large number of cache misses because of the stride accesses. To optimize the performance, we propose that the data layout be reorganized to match the data access pattern as the computation proceeds.

Recently, both data and loop transformations were applied in concert to loop nests for optimizing cache locality [16], [19], [20]. In [19] and [20], the innermost loops are first reordered, and then, the data layouts are determined for all reordered loop nests. This dynamic loop level data transformation is done by dimension reindexing. In our approach, a large transform is decomposed into smaller ones, and then, data reorganization is applied for some of the smaller transforms after considering the reorganization costs. Thus, our approach takes into account the factorization of signal transforms and stride data access to achieve superior performance.

### B. FFT and WHT Performance Optimizations

Various optimization techniques have been developed to improve the performance of FFT on vector and parallel computers [1], [21]–[26] and are targeted toward optimizing the memory performance. Although our approach in this paper is analogous to Bailey’s six-step FFT [22] approach, it is focused on optimizing the performance of signal transforms on a uniprocessor rather than on a vector or parallel processor.

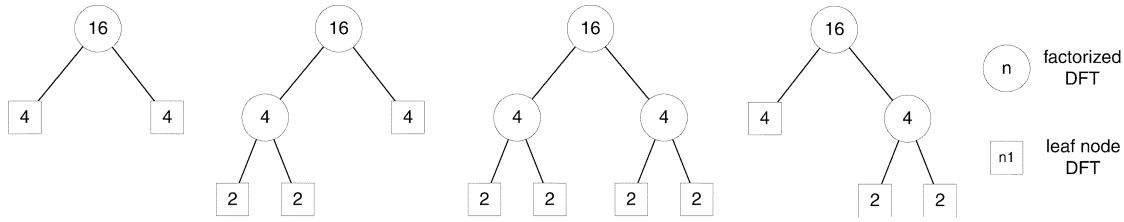


Fig. 1. Examples of factorization tree for 16-point DFT.

The FFTW [1] software architecture consists of a set of straight line unrolled codes, known as *codelets*. These codelets compute small DFTs obtained from a factorization. To compute a large size DFT, a set of these codelets is combined using divide-and-conquer. The method of combining these codelets is represented in a binary tree, and dynamic programming was used to efficiently find an optimal tree. *In this approach, it is assumed that all FFTs of the same size have the same performance.* It is based on a *cache oblivious* algorithm [5], where the cache is assumed to be fully set-associative.

In a factorized DFT such as FFTW, codelets require stride access for input (or output) data. As these strides increase, the performance of codelets degrades [4]. Even if DFTs are of the same data size, DFT performance depends on the data access pattern. To reduce cache conflict misses, an interleaving optimization is proposed in the architecture-cognizant FFT (ACFFT) [2]. However, this approach increases the codelet size, resulting in register explosion.

A technique similar to FFTW was used to implement WHT [3]. In this implementation, algorithmic choices are represented internally in a tree structure. This implementation can support iterative and recursive data structures, as well as combinations of both. Externally algorithmic choices are described by a simple grammar, which can be parsed to create different algorithms that can be executed and timed. A parser is provided to read WHT expressions and translate them into a tree data structure.

In our approach, we determine when and how many times data layout should be reorganized for a factorized computation of a signal transform on a target platform. We find an optimal factorization that includes these reorganizations. Based on these, we develop an approach to *automatically* compute cache-conscious factorized signal transforms to minimize the total execution time, including the data reorganization overheads.

### III. FACTORIZED SIGNAL TRANSFORMS

#### A. Factorization of Signal Transforms

A linear discrete signal processing transform is usually expressed as a matrix-vector product  $x \rightarrow \mathbf{M}x$ , where  $x$  is the (sampled) signal, and  $\mathbf{M}$  is a transform matrix. Examples for discrete signal transforms are the discrete Fourier transform (DFT), discrete cosine transform (DCT), Walsh–Hadamard transform (WHT), etc. Using the inherent divide-and-conquer property of signal transforms, a transform matrix can be factorized into a product of sparse matrices. These sparse matrices are highly structured. They can be represented in concise mathematical form as a tensor product of a smaller

transform and an  $N \times N$  identity matrix ( $\mathbf{I}(N)$ ). For example, the general Cooley–Tukey DFT [27] can be expressed as

$$\mathbf{DFT}(N) = \mathbf{L}(N, N_2) \cdot (\mathbf{I}(N_2) \otimes \mathbf{DFT}(N_1)) \cdot \mathbf{T}(N, N_1) \cdot (\mathbf{DFT}(N_2) \otimes \mathbf{I}(N_1)) \quad (1)$$

where  $N = N_1 \times N_2$ ,  $\mathbf{DFT}(N)$  is the DFT of size  $N$ ,  $\otimes$  is a tensor product operator,  $\mathbf{L}(N, N_1)$  is the stride permutation matrix, and  $\mathbf{T}(N, N_1)$  is the twiddle matrix [28]. Another example of a factorization is for a WHT of size  $2^n$ :

$$\mathbf{WHT}(2^n) = \prod_{i=1}^t (\mathbf{I}(2^{n_1+\dots+n_{i-1}}) \otimes \mathbf{WHT}(2^{n_i}) \otimes \mathbf{I}(2^{n_{i+1}+\dots+n_t}))$$

where  $n = n_1 + \dots + n_t$ . The above factorization method can be applied repeatedly to the smaller signal transforms. This chain of factorization is represented in a tree [3] called a *factorization tree*. For a given transform, alternative trees can be obtained by applying different factorizations at each node, as shown in Fig. 1. In a tree, a node represents a transform of a specific size. In this paper, we call such nodes factorized nodes. Leaf nodes (nodes without children) in a tree represent unfactorized transforms, which are denoted leaf node transforms. A transform (root node in a tree) is computed by executing the leaf node transforms and combining all such child node transforms to realize the root transform. Even though we have used a tree representation to illustrate possible decompositions, an alternate representation of the Cooley–Tukey algorithm can be found in [29]. This alternate representation illustrates the strides.

The divide-and-conquer method reduces the size of a leaf node transform in the factorization tree, thereby reducing the working set size of the computation. Thus, the required data can potentially fit in the cache, and each leaf node can have good cache performance. FFTW [1] and the WHT package [3] exploit the divide-and-conquer property. These packages are developed based on the cache-oblivious algorithm, where the cache is assumed to be fully set-associative, and the replacement policy is ideal. The performance of a factorized transform is assumed to be dependent only on its size. The performance of DFT (assuming a fully set-associative cache) can be analyzed using the input–output complexity analysis of the “red-blue pebbling game” [30]. The lower bound on cache misses for an  $N$ -point FFT is  $O(N \log_2 N / \log_2 C)$ , where  $C$  is the cache size.

#### B. Cache Performance of Factorized Signal Transforms

In this subsection, we discuss the cache performance of a leaf node with stride data access, which is involved in the computation of a large signal transform factorized by the divide-and-con-

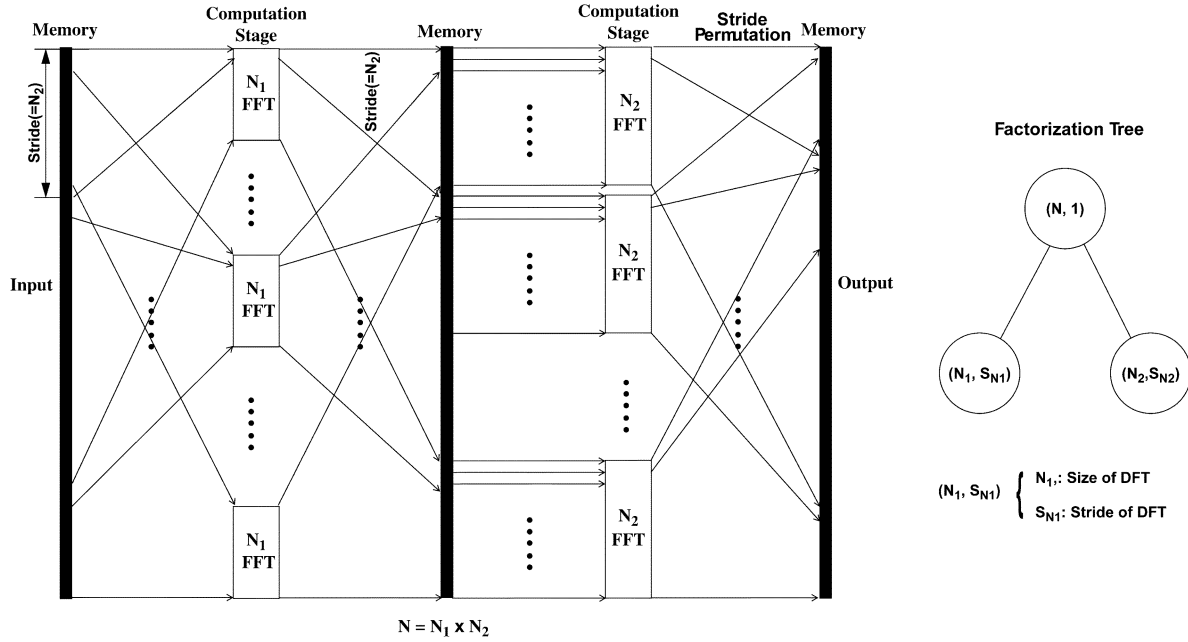


Fig. 2. Computation of  $N$ -point DFT using the Cooley–Tukey algorithm.

quer method. TLB misses and page faults also occur along with cache misses, but they are not critical to the performance for the small sized transforms obtained after factorization. For the sake of illustration, we consider DFT to be an example signal transform throughout this paper.

We consider an  $N$ -point DFT, which is factorized as  $N_1 \times N_2$  using the Cooley–Tukey algorithm [see (1)]. A tensor product of a small DFT and an identity matrix can be implemented as a small DFT with a stride access. First,  $N_2$   $N_1$ -point DFTs are performed with stride  $N_2$  followed by a twiddle multiplication. Following these computations,  $N_1$   $N_2$ -point DFTs are performed with unit stride. A stride permutation follows to correct the order of the transformed data. Fig. 2 illustrates the computation of the Cooley–Tukey algorithm and the new tree representation, where each node is represented with its size and stride. The divide-and-conquer method is recursively applied to the child nodes until the size of leaf nodes in a tree becomes smaller than the cache size. In addition, stride accesses are involved in leaf node computations.

To understand the cache behavior, we consider a two-level memory hierarchy consisting of cache and main memory. The size of all parameters is represented in terms of the number of data points. Let  $C$  denote the size of the cache and  $B$  denote the size of the cache line. To simplify our analysis, we assume the cache to be direct mapped. The number of cache lines in a direct mapped cache is given by  $C/B$ . Note that most of the state-of-the-art platforms have either direct-mapped or small set associative caches. The analysis for a  $k$ -way set-associative cache is similar to that for a direct-mapped cache.

To show the impact of a data access stride on cache performance, we consider a leaf node DFT with stride access in a factorization tree. A leaf node is an  $N_l$ -point DFT with stride  $S$ , where  $N_l < C$ . The cache behavior for performing two successive DFTs is illustrated in Fig. 3, where  $N_l$  is assumed to be 4.

- *Case I:  $S = 1$  and Case II:  $S > 1$  and  $N_l \times S \leq C$*

The number of compulsory cache misses caused by an  $N_l$ -point DFT with stride  $S$  is  $\min(N_l \times S/B, N_l)$ . As shown in Fig. 3, all of the data points of an  $N_l$ -point DFT are mapped onto the cache without any conflict. When a successive  $N_l$ -point DFT needs to be performed, its data points already exist in cache lines since the data in these cache lines were fetched by the previous DFT. There is spatial reuse of data.

- *Case III:  $S > 1$  and  $N_l \times S > C$*

When the stride is large such that  $N_l \times S > C$ , there are conflict misses in addition to compulsory misses, even though  $N_l < C$ . These conflict misses have significant impact on the performance. As shown in Fig. 3, the block containing the first (second) data point and the block containing the third (fourth) data point will be mapped onto the same cache line. Therefore, conflict misses occur in the computation of a single  $N_l$ -point DFT. Furthermore, the spatial reuse in the computation of successive DFTs is also lost. For a successive DFT computation, the first and second blocks are accessed again. However, the first (second) block was replaced by the third (fourth) block in the previous DFT computation due to conflict misses. Therefore, these stride data accesses result in *cache pollution*: Cache lines are replaced before all data points in the cache line are fully utilized [31]. Cache pollution and conflicts lead to considerable performance degradation in the computation of successive leaf node DFTs. In our analysis above, we assume that  $N$  is a power of two. In general, cache pollution will occur for any  $N$ , but the amount of conflict misses when  $N$  is not a power of two will be less than the amount of conflict misses when  $N$  is a power of two.

To improve the performance, we discuss a high level optimization in Section IV.

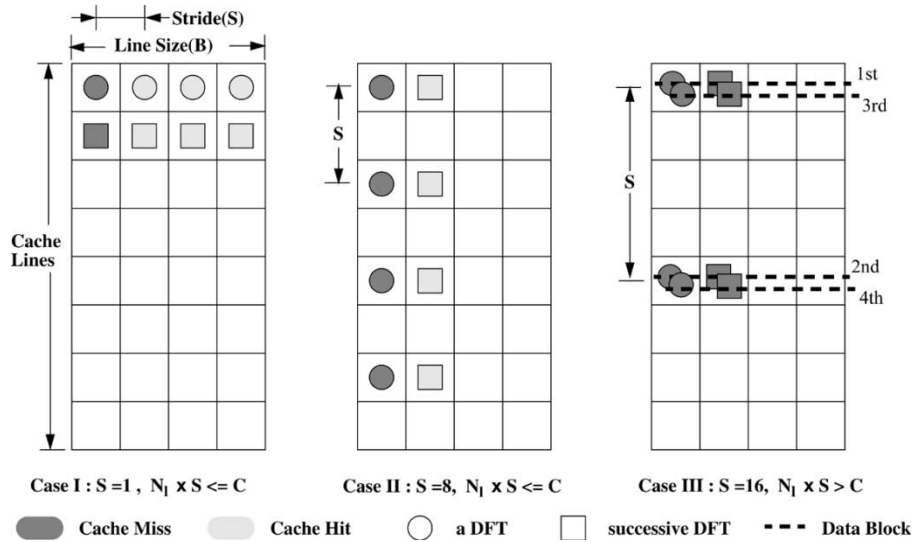


Fig. 3. Cache behavior for two successive DFTs ( $N_1 = 4, B = 4, C = 32$ ).

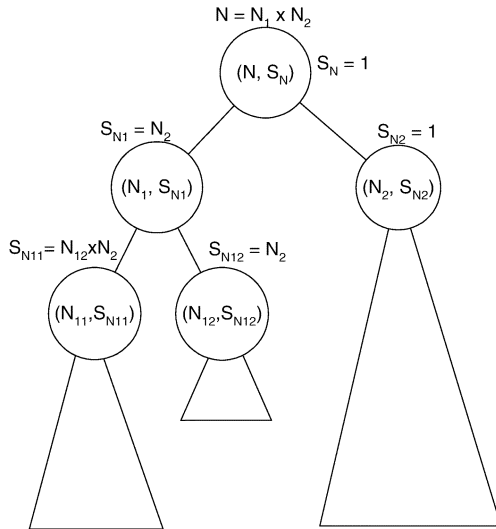


Fig. 4. Strides resulting from Cooley–Tukey factorization.

#### IV. CACHE-CONSCIOUS FACTORIZATION OF SIGNAL TRANSFORMS

In Section III, we explained the effect of large stride on cache performance. To improve performance, we propose an approach where the data layout in the memory is dynamically reorganized between the computation stages of leaf node DFTs in a tree. In addition, we propose a dynamic programming-based search algorithm to find an optimal factorization tree that has the minimum overall execution time. This approach considers the stride of the data access as well as the size of the DFTs to be computed.

##### A. Dynamic Data Layout

As discussed in Section III-B, each node in a tree is represented as a DFT with stride access (Fig. 2). Fig. 4 shows the factorization tree where a node represents a DFT with its size and stride. As explained in Section III-B, these stride accesses can cause cache conflict and pollution in leaf node computations, thereby resulting in performance degradation. This is due to the mismatch between the data access pattern and the data layout in memory. To

improve the performance of a factorized DFT, data layout should be *dynamically* reorganized to match the data access pattern of the computation. Note that data refers to the input and output data as well as intermediate results. As shown in Fig. 5, data layout in the memory is reorganized between computation stages, thereby reducing the number of cache misses incurred during the computation. We call this the *dynamic data layout (DDL)* approach. After performing computations, reverse reorganizations are performed to restore the data in the natural order.

To show the effectiveness of the DDL approach, we present a simple example of a 256-point DFT, decomposed as  $16 \times 16$  using the Cooley–Tukey algorithm. In Fig. 6,  $16 \times 16$  data points are arranged in row-major order. Cache is assumed to be direct mapped. For the sake of illustration, the cache size is assumed to be 64 points, and the line size is assumed to be 4. In Cooley–Tukey computation, the first sixteen 16-point DFTs have stride 16. Fig. 6(a) shows the data points accessed by a 16-point DFT with stride 16. Every fourth data point in the computation of the 16-point DFT is mapped onto the same cache line. Therefore, 16 data points are mapped onto only four cache lines, resulting in conflict misses. However, after reorganizing the data layout [see Fig. 6(b)], all of the 16 data points for a DFT are mapped onto contiguous locations in the cache, resulting in no conflict. Thus, the number of cache misses is reduced, and each cache line is fully utilized, thereby improving the overall performance.

In a factorization tree, our approach can be applied at any node. However, data reorganization is an overhead on performance since it requires memory accesses. In order to improve the overall performance, the number of nodes where reorganizations are applied should be minimized. We also need to determine the nodes where the DDL approach has to be applied in a factorization tree.

In our DDL approach, a tree called a *DDL factorization tree* describes the factorization for nodes with their size and stride, as shown in Fig. 5. Changing the data layout causes a change in the data access stride in a node. For a particular factorization, there are several different DDL factorization trees since various layouts can be applied to the nodes in the tree. Among all the possible trees for a factorization, we should find a factorization

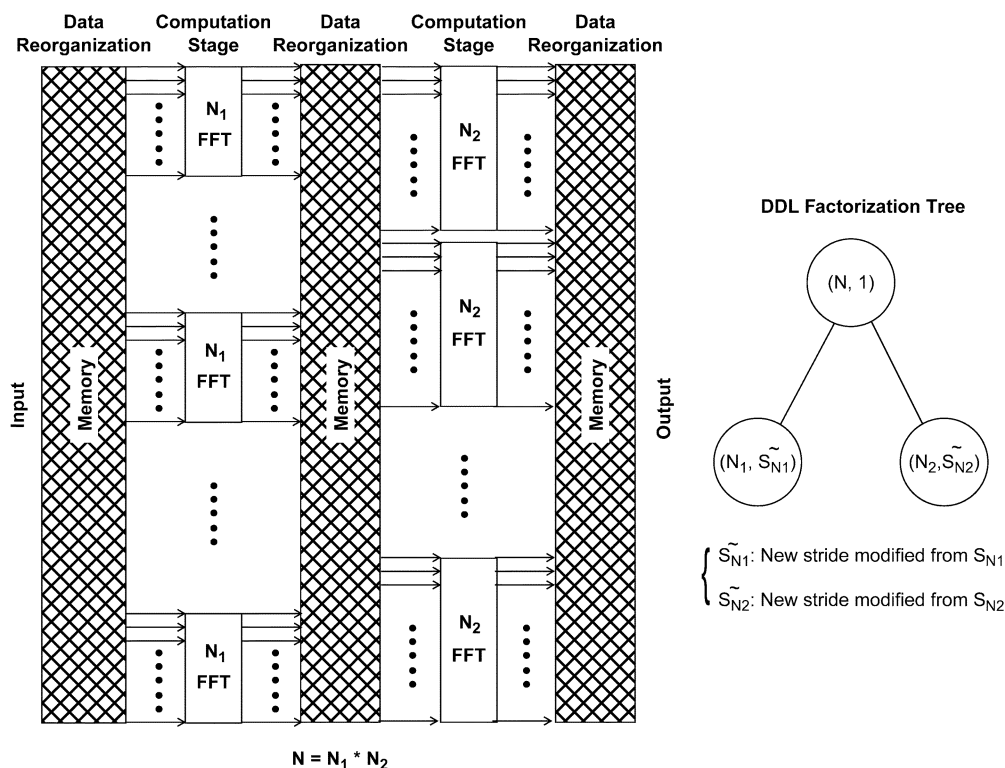
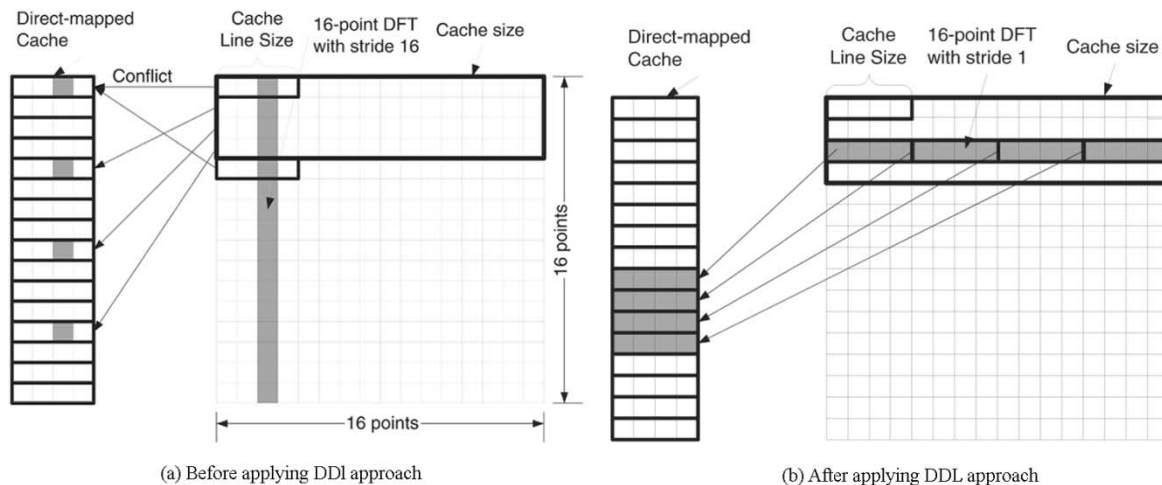


Fig. 5. Dynamic data layout approach for the Cooley–Tukey algorithm.


 Fig. 6. Data access pattern for a  $16 \times 16$  factorization.

tree with the minimum execution time. Such a tree is called a *cache-conscious factorization tree*. To find such a tree, we make the following assumptions.

- All DFTs of the same size and stride have the same computation cost.
- At each node,  $k$  different data layouts (strides) are considered, where  $k > 1$ . No other layout is considered.

We need to define a cost model for an  $N$ -point DFT based on the execution time using all possible DDL factorization trees. Consider an  $N$ -point DFT with stride  $S_N$ , which is a node labeled  $(N, S_N)$  in Fig. 4. Its computation cost is represented as  $\text{DFT}(N, S_N)$ . For a given  $N_1 \times N_2$  factorization of  $N$ , consider that an  $N_1$ -point DFT is computed with stride  $S_{N_1}$ , and an  $N_2$ -point DFT is computed with stride  $S_{N_2}$ .  $L_{S_{N_1}}$  ( $L_{S_{N_2}}$ ) de-

notes the original data layout for the  $N_1$  ( $N_2$ )-point DFT with stride  $S_{N_1}$  ( $S_{N_2}$ ). The minimum cost including data reorganizations for a given factorization is as follows:

$$\min_{S_{N_1}, S_{N_2}} \left[ \text{Dr}(N, L_{S_{N_1}}, L_{S_{N_1}^{\sim}}) + N_2 \times \text{DFT}(N_1, S_{N_1}^{\sim}) + \text{Dr}(N, L_{S_{N_1}^{\sim}}, L_{S_{N_2}^{\sim}}) + T(N, N_1) + N_1 \times \text{DFT}(N_2, S_{N_2}^{\sim}) + \text{Dr}(N, L_{S_{N_2}^{\sim}}, L_{S_{N_2}}) \right] \quad (2)$$

where  $S_{N_1}^{\sim}$  ( $S_{N_2}^{\sim}$ ) denotes a new stride for the  $N_1$  ( $N_2$ )-point DFT using the reorganized data layout  $L_{S_{N_1}^{\sim}}$  ( $L_{S_{N_2}^{\sim}}$ ).

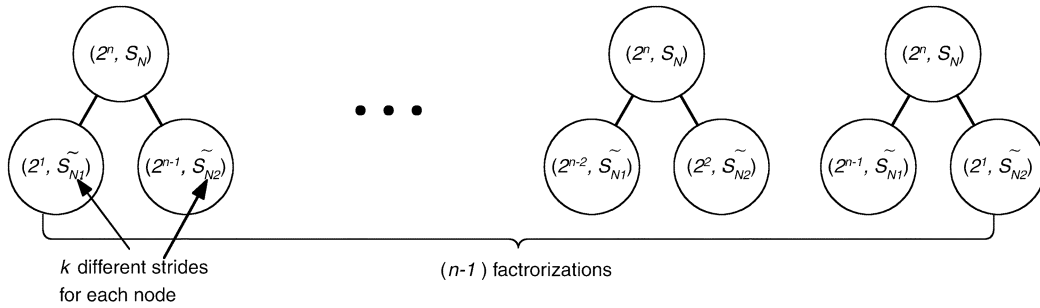


Fig. 7. Different DDL factorization trees for  $2^n$ -point DFT.

$T(N, N_1)$  is the cost incurred in twiddle factor multiplications.  $Dr(N, L_{S_{N_i}}, L_{\tilde{S}_{N_i}})$  is the cost of reorganizing the  $N$  data points used for  $N_i$ -point DFTs in the layout  $L_{S_{N_i}}$  to the layout  $L_{\tilde{S}_{N_i}}$ , where  $i = 1$  or  $2$ . The cost of reorganizing data of size  $N$  involves  $O(N/B)$  memory accesses, where  $B$  is the cache line size. When  $L_{S_{N_i}} = L_{\tilde{S}_{N_i}}$  in  $Dr(N, L_{S_{N_i}}, L_{\tilde{S}_{N_i}})$ , no data reorganization is performed. If  $k$  different strides are considered for both  $\tilde{S}_{N_1}$  and  $\tilde{S}_{N_2}$ , the complexity of evaluating (2) (for a given factorization) is  $O(k^2)$ . In Section IV-B, we will develop a search algorithm to find an optimal factorization tree with the minimum execution time for an  $N$ -point DFT.

### B. Search for an Optimal Factorization

An  $N$ -point DFT can be represented using many possible factorization trees, where  $N = 2^n$ . Optimizing the performance of a DFT becomes a search problem of finding an optimal tree with the minimum execution time over a large space of possible factorization trees. The search algorithm of the previous approaches [1], [3] considered only the size of the DFTs. The search space is  $\Theta(5^n/n^{3/2})$  [3]. Since we consider the stride as well as the size of DFTs, the search space in our approach is much larger than that in the previous approaches. Exhaustive search for an optimal factorization tree is impractical. To efficiently search for an optimal tree, we use dynamic programming. In dynamic programming, a DDL factorization tree is built bottom-up. Optimal factorizations for DFTs smaller than the DFT under consideration are already determined by dynamic programming. The root node of the  $N$ -point DFT is factorized into two children, as shown in Fig. 7. If the factorization tree is built bottom-up considering only the size of DFTs, the complexity is  $O(n^2)$ . As discussed in Section IV-A, if  $k$  different strides are considered at each node, the complexity to find a cache-conscious factorization tree for a given factorization would be  $O(k^2)$ . Therefore, the overall search complexity using the DDL approach is  $O(k^2 n^2)$ . In the following, the size of a node refers to the size of the DFT to be performed at the node (see Fig. 2). The search space for an optimal factorization tree can be further reduced based on the following property.

*Property 1:* The stride of a (child) node in the tree depends on the stride of its parent node, its position (left or right), and the size of its sibling.

Based on this property, the strides of the left and the right nodes are as follows:

$$\begin{aligned} S_{N_1} &= S_p \times N_2 \\ S_{N_2} &= S_p \end{aligned}$$

where  $S_{N_1}$  ( $S_{N_2}$ ) is the stride of the left (right) node,  $N_1$  ( $N_2$ ) is the size of the left (right) node, and  $S_p$  is the stride of the parent. This property helps to reduce our search space. As shown in Fig. 4, node  $(N_1, S_{N_1})$  is recursively decomposed to node  $(N_{11}, S_{N_{11}})$  and node  $(N_{12}, S_{N_{12}})$ . The stride of a child node is related to that of its parent node, the size of its sibling, and its location in the tree. As shown in Fig. 4, the root in a tree always has unit stride. Thus, optimal trees determined by our dynamic programming have unit stride. Based on Property 1, the right subtree also has unit stride. In dynamic programming, the optimal cost of performing an  $N_2$ -point DFT with unit stride has already been determined before searching for an optimal tree for the  $N$ -point DFT. Thus, the right subtree is an optimal tree. However, the left subtree, which is an  $N_1$ -point DFT, has stride accesses. Hence, we apply the DDL approach only to the left subtree. Based on this, we can simplify our cost model for dynamic programming. Consider a DFT of size  $N (= 2^n)$ . This DFT can be factorized as  $N_1 (= 2^{n_1}) \times N_2 (= 2^{n-n_1})$ , where  $n_1 = 0, \dots, n-1$ . The cost of the  $2^n$ -point DFT using an optimal factorization tree can be computed as follows:

$$\begin{aligned} \text{DFT}(2^n, 1) &= \min_{n_1, S_{2^{n_1}}} \left[ \text{Dr}(2^n, L_{S_{2^{n_1}}}, L_{\tilde{S}_{2^{n_1}}}) \right. \\ &\quad + 2^{n-n_1} \times \text{DFT}(2^{n_1}, \tilde{S}_{2^{n_1}}) \\ &\quad + \text{Dr}(2^n, L_{\tilde{S}_{2^{n_1}}}, L_{S_{2^{n_1}}}) \\ &\quad + T(2^n, 2^{n_1}) \\ &\quad \left. + 2^{n_1} \times \text{DFT}(2^{n-n_1}, 1) \right] \quad (3) \end{aligned}$$

where  $N = 2^{n_1} \times 2^{n-n_1}$ . The initial values are the costs for computing small DFTs with different strides and the costs for performing data reorganizations between various strides. The initial values are determined by executing the codes for these operations.  $\text{DFT}(N, S)$  represents the optimal cost to compute an FFT of size  $N$  with stride  $S$ . Furthermore, according to the analysis in Section III-B, the DDL approach can be beneficial only when the product of the size of the DFT and the stride is greater than the cache size ( $N \times S > C$ ). Thus, for the size of a DFT smaller than the cache size, a factorization tree with data reorganization may not be an optimal solution because of its reorganization overhead. For the size of a DFT larger than the cache size, factorization trees with data reorganizations can lead to faster DFT implementations. We apply the DDL approach only to transforms, whose sizes are equal to or larger than the cache size.

```

/* To find an optimal factorization tree for a 2i-point DFT */
FindOptTree (int i, tree *OptTree)
{
    minTime = Infinity
    for j = 0 to i-1 /* i different factorizations */
        for l = 0 to k-1 /* k different strides */
            Sl /* lth different stride of the left subtree*/
            Tree = GenerateTree(OptTree[j], OptTree[i-j], Sl)
            Time = FFT_Measure(Tree)
            if (Time < minTime)
                OptTree[i] = Tree
                minTime = Time
            end if
        end for
    end for
}

```

Fig. 8. Search algorithm considering dynamic data layout.

To find an optimal factorization tree for  $N(= 2^n)$ -point DFT, dynamic programming requires  $n$  iterations. Consider the  $i$ th iteration to build an optimal factorization tree for a  $2^i$ -point DFT, where  $i = 1, \dots, n$ . The optimal factorization trees for all DFTs smaller than a  $2^i$ -point DFT have already been determined in earlier iterations. During the  $i$ th iteration, we search all possible DDL factorizations. The algorithm (FindOptTree) to find the optimal tree for a  $2^i$ -point DFT is described in Fig. 8. In the algorithm, GenerateTree generates a new tree considering the size of both child nodes and the stride of the left child node. The function FFT\_Measure measures the FFT execution time for a given tree including data reorganization. If the execution time is less than minTime, the OptTree and minTime are updated with current Tree and Time. When  $N$  is a power of 2, sizes and strides in various possible factorization trees are also powers of 2. As shown in Fig. 8,  $(i - 1)$  different factorizations and  $k$  different strides for the left subtree of the root node are considered. The overall complexity of our search algorithm to find an optimal factorization tree of a  $2^n$ -point DFT using dynamic programming is thus  $O(kn^2)$ . The search algorithm for finding the optimal factorization trees in implementing WHT is also similar. Note that by performing dynamic data layout, the complexity in terms of the number of arithmetic operations is not changed.

In dynamic programming, while searching for an optimal tree for a  $N$ -point DFT decomposed as  $N_1 \times N_2$ , the optimal trees for the  $N_1$  and the  $N_2$ -point DFTs with unit stride have already been determined. Therefore, we have to consider only the  $N_1$ -point DFT with stride  $N_2$ . We can consider  $k$  different strides for the  $N_1$ -point DFT, but exploring all  $k$  strides will increase the search time if  $N$  is large. Since unit strides reduce cache pollution significantly, we considered changing the nonunit stride to unit stride only. After reorganization to unit stride access, the data layout is compacted for the  $N_1$ -point DFT, and the cache misses will be reduced. Hence, we intuitively select  $k = 2$ , the two strides being the original stride and unit stride, for our implementation in Section V. Since we use  $k = 2$ , the complexity is  $O(n^2)$ . The implementation for WHT is also similar.

Our approach is a high-level optimization that exploits the characteristics of the memory hierarchy by analyzing the data access pattern. Our approach can be applied on top of low-level optimized codes such as those employed in FFTW [1] or the CMU package [3], thus making our approach portable.

### C. Alternate Factorizations Using Dynamic Data Layout

Table I shows the performance of some factorization trees using dynamic data layout and static data layout. By “static data layout,” we mean that the data layout does not vary during the computation. This SDL approach is used in FFTW and the CMU package. These packages are continuously updated, and improvements are made. The comparisons are based on the available packages at the time of submission of this manuscript. Some versions of these packages may incorporate the layout optimization approach discussed in this paper. The primary purpose of our experiments is to understand the impact of our dynamic data layout approach.

The experiments were performed on Alpha 21 264 (500 MHz clock speed, 2 MB L2 cache). The compiler and its optimization options used in the experiments are shown in Table IV. The execution time of the optimal DFTs for both data layouts is shown with a bold font. In Table I, “ $n$ ” denotes a  $2^n$ -point unfactorized DFT. Cooley–Tukey algorithms using static and dynamic data layouts, which are factorized into  $2^{n_1} \times 2^{n_2}$ , are represented as “ct[ $n_1, n_2$ ]” and “ctddl[ $n_1, n_2$ ],” respectively. Some trees in Table I include the term “ctddl” twice, which means that data reorganization is applied at two nodes in the trees.

As can be expected, the experimental results in Table I show that the optimal factorization tree using dynamic data layout has better performance than the optimal factorization tree using static data layout. We noticed that the optimal factorization trees using static data layout were close to a right-most tree. (Right-most trees consist of a leaf node on the left side and a right subtree, which is also a right-most tree.) The optimal factorization trees using dynamic data layout were close to balanced trees.

In order to validate the cost estimation equation [see (3)], in Table I, we also list the estimated execution time for the factorization trees using the DDL approach. In (3), the estimated execution time is calculated as  $T_1 + T_2 + T_3$ , where  $T_1$  is the execution time of the left and right subtrees  $2^{n-n_1} \times \text{DFT}(2^{n_1}, S_{2^{n_1}}) + 2^{n_1} \times \text{DFT}(2^{n-n_1}, 1)$ ,  $T_2$  is the time spent on data reorganization  $\text{Dr}(2^n, L_{S_{2^{n_1}}}, L_{S_{2^{n_1}}}) + \text{Dr}(2^n, L_{S_{2^{n_1}}}, L_{S_{2^{n_1}}})$ , and  $T_3$  is the time spent on twiddle factor multiplication  $T(2^n, 2^{n_1})$ . In Table I, the values for  $T_1$  were calculated from  $\text{DFT}(2^1, 1) \dots \text{DFT}(2^{19}, 1)$ . The values for  $\text{DFT}(2^1, 1) \dots \text{DFT}(2^{19}, 1)$  were obtained by using the proposed dynamic programming search. The values for  $T_2$  and  $T_3$  were obtained offline (before the dynamic programming search was performed) by actually executing the data reorganizations and the twiddle factor multiplications. As can be seen from Table I, (3) provides a close estimation to the execution time of the factorization trees. The optimal factorization tree obtained through the estimated execution time was the same as the optimal factorization tree determined after executing all possible factorization trees by considering the data layouts discussed in Section IV-A.

## V. EXPERIMENTAL RESULTS

To show the cache performance improvements using our technique, we conducted several cache simulations. We compare the cache miss rates for FFT implemented using the DDL and SDL

TABLE I  
SOME ALTERNATE FACTORIZATION TREES ON ALPHA 21 264. THE SIZE OF THE FFT IS  $2^{20}$

Using the DDL approach					
Factorization Tree	Exec. Time (ms)	Estimated Exec. Time(ms)	$T_1$ (ms)	$T_2$ (ms)	$T_3$ (ms)
ctddl[4,ctddl[ct[[2],ct[[3],[3]]],ct[[2],ct[[3],[3]]]]	1190.72	917.27	460.97	360.36	95.93
ctddl[ct[[2],[3]],ctddl[ct[[3],[4]],ct[[2],ct[[3],[3]]]]	1139.96	886.69	443.98	348.49	94.20
ctddl[ct[[3],[3]],ctddl[ct[[3],[4]],ct[[3],[4]]]	1102.88	851.16	358.25	397.31	95.59
ctddl[ct[[3],[4]],ctddl[ct[[3],[3]],ct[[3],[4]]]	1116.54	845.91	377.10	373.08	95.72
ctddl[ct[[2],ct[[3],[3]]],ct[[3],ct[[3],ct[[3],[3]]]]	958.43	715.82	296.58	323.52	95.71
ctddl[ct[[3],ct[[3],[3]]],ct[[2],ct[[3],ct[[3],[3]]]]	898.89	694.39	282.18	315.66	96.54
ctddl[ct[[3],ct[[3],[4]]],ct[[3],ct[[3],[4]]]	<b>620.24</b>	<b>480.74</b>	216.67	166.91	97.16
ctddl[ct[[2],ct[[3],ct[[3],[3]]]],ct[[3],ct[[3],[3]]]	894.99	694.72	282.18	315.66	96.87
ctddl[ct[[3],ct[[3],ct[[3],[3]]]],ct[[2],ct[[3],[3]]]	909.63	716.74	296.58	323.52	96.63
ctddl[ctddl[ct[[3],[3]],ct[[3],[4]]],ct[[3],[4]]]	1097.02	846.68	377.10	373.08	96.49
ctddl[ctddl[ct[[3],[4]],ct[[3],[4]]],ct[[3],[3]]]	1074.57	852.37	358.25	397.31	96.80
ctddl[ctddl[ct[[3],[4]],ct[[2],ct[[3],[3]]]],ct[[2],[3]]]	1147.77	887.50	443.98	348.49	95.02
ctddl[ctddl[ct[[2],ct[[3],[3]]],ct[[2],ct[[3],[3]]]],{4}	1159.48	915.39	460.97	360.36	94.05

Using the SDL approach	
Factorization Tree	Exec. Time (ms)
ct[2,ct[3,ct[3,ct[3,ct[3,ct[3,3]]]]]	1899.29
ct[3,ct[4,ct[1,ct[3,ct[3,ct[3,3]]]]]	1826.09
ct[4,ct[4,ct[3,ct[3,ct[3,3]]]]]	<b>1577.21</b>
ct[ct[3,ct[3,3]],ct[2,ct[3,ct[3,3]]]]	2819.66
ct[ct[4,ct[3,3]],ct[4,ct[3,3]]]	2652.76
ct[ct[2,ct[3,ct[3,3]]],ct[3,ct[3,3]]]	3213.96

approaches. Further, we apply our approach to recent FFT and WHT packages developed at CMU. We perform experiments on several platforms and present a performance comparison of our implementations<sup>2</sup> with the previous packages [3], [7]. In addition, we show that the optimal factorization trees from the DDL approach are different from those in the SDL approach.

#### A. Simulation Results

We performed simulations using the cache simulator in the SUN Shade simulator [32]. In our simulations, we used a direct mapped cache of size 512 KB. Each data point is a double-precision complex number (16 Bytes). Thus, the cache can hold up to  $2^{15}$  data points. We studied two different cases to illustrate cache performance. First, the cache miss rates for different sized FFTs ( $N$ ) were studied. Second, we performed simulations by varying the cache line size ( $B$ ) for a fixed-size FFT ( $N$ ).

Fig. 9 plots the cache miss rate. Table II shows the number of cache accesses and misses for DDL and SDL. As can be seen

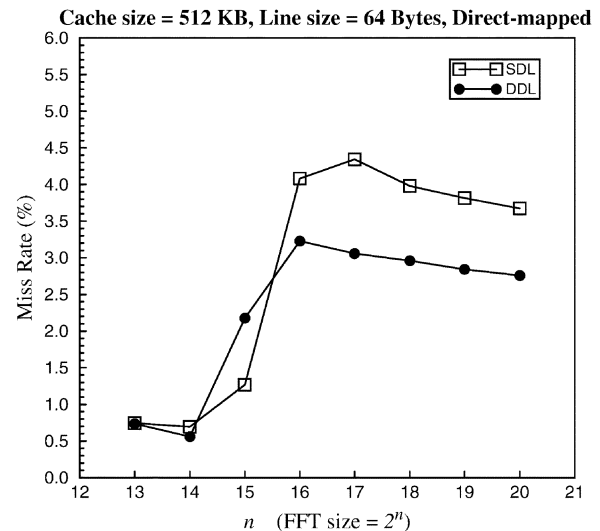


Fig. 9. Miss rates for various FFT sizes.

<sup>2</sup>Sample optimized codes can be accessed at <http://www.ece.cmu.edu/~spiral/wht.html#DOWNLOAD> or <http://lotus1000.usc.edu/prasanna/reports>.

from these results, when the size of the FFT becomes larger than the cache size, the number of cache misses using the DDL

TABLE II  
NUMBER OF CACHE ACCESSES AND MISSES FOR VARIOUS FFT SIZES

FFT Size ( $\log_2 N$ )		13	14	15	16	17	18	19	20
DDL	Misses	4261	8801	71434	234477	465062	939010	1882192	3801567
	Accesses	733579	1548619	3268555	6881095	14416967	30145351	62977351	131330887
SDL	Misses	4341	9497	40323	298294	596750	1191287	2382261	4767075
	Accesses	722238	1522558	3196670	6697850	14043770	29385594	61313402	127714170
DDL/SDL	Misses	98.15%	92.67%	177.15%	78.61%	77.93%	78.82%	79.01%	79.74%
	Accesses	101.57%	101.71%	102.25%	102.74%	102.66%	102.59%	102.71%	102.83%

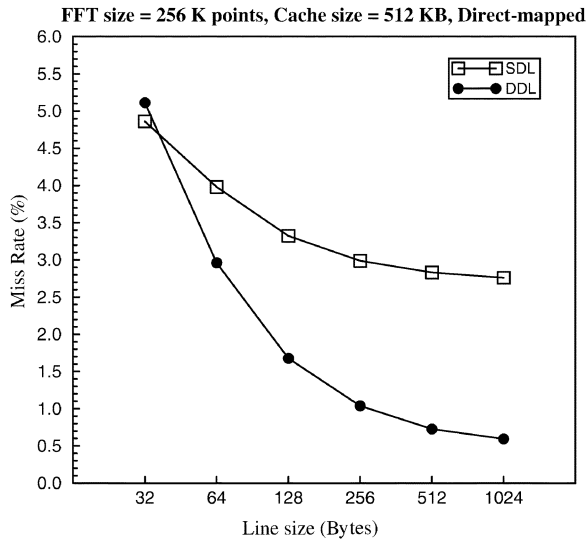


Fig. 10. Miss rates for various cache line sizes.

approach is much smaller than that using the SDL approach. The improvement in the overall cache performance results from the reduction (up to 22.07%) in cache misses in spite of the slight increase (less than 3%) in the cache accesses caused by data reorganization.

Fig. 10 shows the cache miss rate as the cache line size is varied. As cache line size increases, the cache miss rate decreases. Since the DDL approach changes the stride in the FFT computation to unity, it utilizes the cache line more efficiently than the SDL approach, resulting in lower cache miss rate. When the cache line size is 64 bytes, which is the cache line size in most state-of-the-art platforms, the cache miss rate is 3.98% using the SDL approach and 2.96% using the DDL approach. The DDL approach has a 25% lower miss rate compared with the SDL approach.

### B. Experimental Results

In this subsection, we report experimental results conducted on four state-of-the-art platforms. Tables III and IV summarize the relevant architectural parameters, compilers, and optimization options for various platforms used in our experiments. To obtain accurate execution times, computations were repeated until the overall execution time was larger than 1 s. The measured time is the wall clock time using the `clock()` function.

TABLE III  
PARAMETERS OF THE PLATFORMS

Processor		Alpha 21264	MIPS R10000	Pentium 4	UltraSPARC III
Clock (MHz)		500	195	1700	750
L1 Cache	Size (KB)	64	32	8	64
	Line (Bytes)	64	32	64	32
L2 Cache	Size (KB)	4096	4096	256	4096
	Line (Bytes)	64	64	64	64

The total execution time was obtained by deducting the loop overhead from this time. The average execution time is reported. Throughout this subsection,  $N$  denotes the size of FFT or WHT  $N = 2^n$  for some positive integer  $n$ .

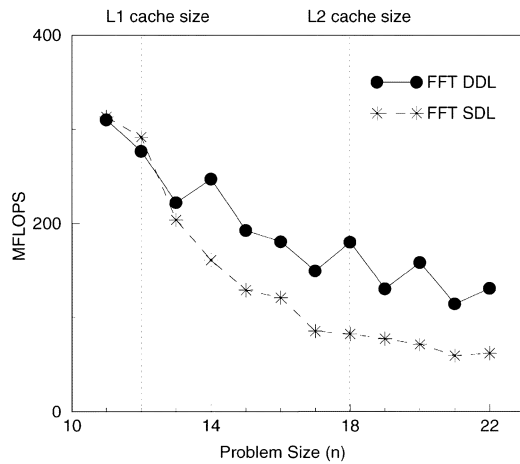
In experiments, we used the FFT package [7] provided by CMU, which is called FFT SDL in this paper. FFT SDL uses the optimized code modules (codelets) from FFTW as leaf node FFTs. Its design is based on an FFT package presented in [33], which is different from FFTW. We applied our DDL approach as a high-level optimization over the FFT SDL. We call our modified FFT package as FFT DDL. For experiments, all data points in a DFT are double-precision complex numbers (16 bytes). To show performance improvement using the DDL approach, we compare the performance of FFT DDL with that of FFT SDL. In addition, we present the relative performance improvement of FFT DDL compared with FFTW, using the following formula:

$$\frac{\text{MFLOPS of FFT DDL} - \text{MFLOPS of FFTW}}{\text{MFLOPS of FFTW}} \times 100\%.$$

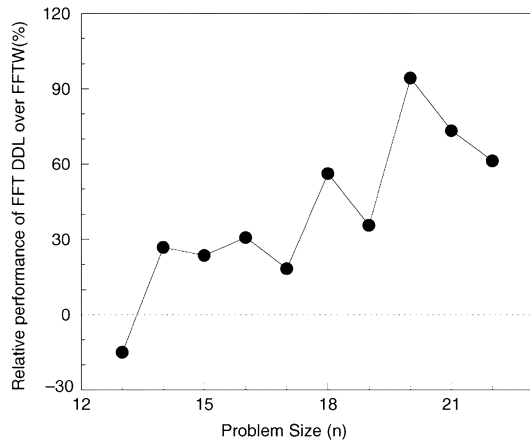
For these experiments, we used the recent FFTW (version 2.1.3). Note that different factorization trees for a FFT of size  $N$  are of the same order of complexity  $O(N \log N)$ . However, the exact number of floating-point operations for various factorization trees may be different. The difference can be due to different constant factors or different lower order terms. We therefore use a normalized measure of MFLOPS as the performance metric, which is calculated as  $5N \log N/t$ , where  $t$  is the execution time in micrometers. Throughout this section, this metric is used. The same performance metric is used in FFTW [1].

TABLE IV  
COMPILER AND OPTIMIZATION OPTIONS USED IN THE EXPERIMENTS

Processor	Compiler	Optimization Flags
Alpha 21264	gcc version egcs-2.91.66	-O6 -fomit-frame-pointer -pedantic
MIPS R10000	MIPSpro Compilers: Ver. 7.2.1	-r10000 -O3 -Ofast -mips4
Pentium 4	gcc version 2.96	-O6 -fomit-frame-pointer -pedantic -malign-double
UltraSPARC III	gcc version 2.8.1	-O6 -fomit-frame-pointer -pedantic



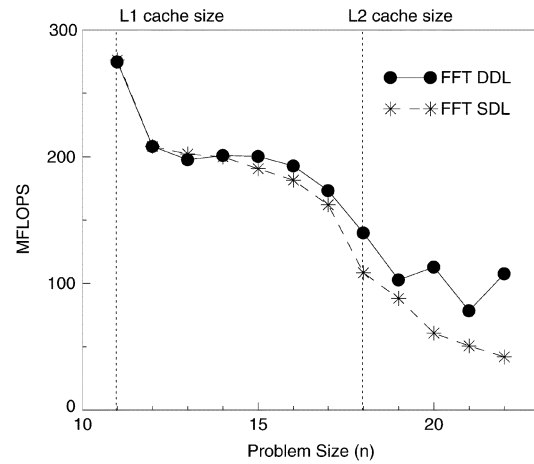
(a) Comparison with FFT SDL



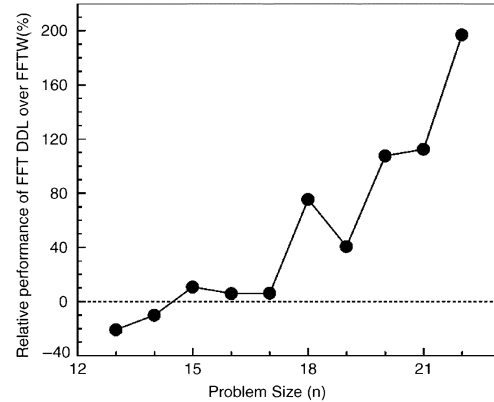
(b) Comparison with FFTW

Fig. 11. Performance of FFT DDL on Alpha 21264.

Figs. 11–14 show the performance of FFT DDL in comparison with FFT SDL and FFTW on four different platforms. Fig. 11(a) shows the performance of FFT SDL and FFT DDL on the Alpha 21 264 platform. For small problems ( $N$  less than  $2^{13}$ ), all the data points required for the FFT can reside in the cache. It is not necessary to reorganize the data layout. Our search algorithm selects the same tree as the tree used in the SDL approach. Thus, FFT DDL showed the same performance as FFT SDL. For sizes between  $2^{13}$  and  $2^{18}$ , the required data can reside in the  $L2$  cache. In this range, our approach produces a marginal gain as it reduces the  $L1$  cache misses, but as the problem size exceeds the size of the  $L2$  cache, the FFT DDL achieves a speed up of up to  $2.23 \times$ . Fig. 11(b) shows the relative performance of FFT DDL compared with FFTW



(a) Comparison with FFT SDL



(b) Comparison with FFTW

Fig. 12. Performance of FFT DDL on MIPS R10000.

on Alpha 21 264. If the size of the FFT is less than  $2^{13}$ , the performance of FFT DDL was worse than FFTW. However, for sizes between  $2^{13}$  and  $2^{18}$ , FFT DDL was superior by approximately 30%. If the size is larger than the  $L2$  cache size, the relative performance of FFT DDL is up to 98% better compared with FFTW.

Fig. 12(a) shows the performance of FFT SDL and FFT DDL on MIPS R10000. Fig. 12(b) shows the relative performance of FFT DDL over FFTW. For FFT size between  $2^{12}$  and  $2^{18}$  (i.e., for sizes larger than the  $L1$  cache size and smaller than the  $L2$  cache size), the relative performance gain was around 10%. It is smaller than the performance improvement on Alpha 21 264 since the cache line size of MIPS R10000 is smaller than that of Alpha 21 264. As discussed in Section V-A, our approach is less effective for smaller cache lines. On Pentium 4 and UltraSPARC

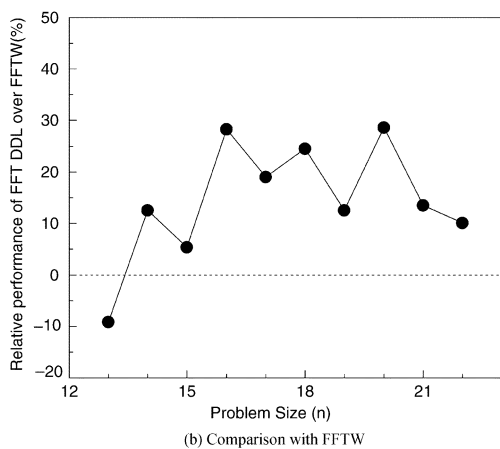
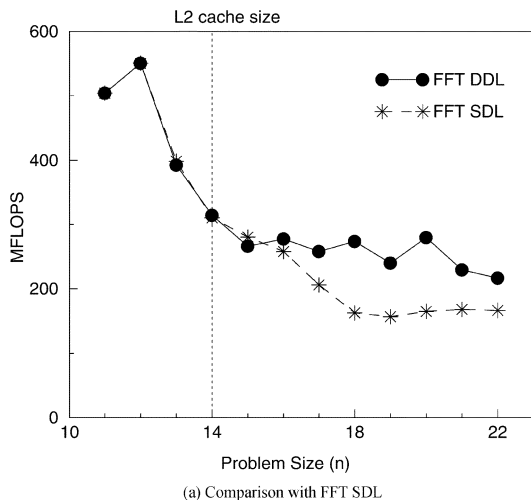


Fig. 13. Performance of FFT DDL on Pentium 4.

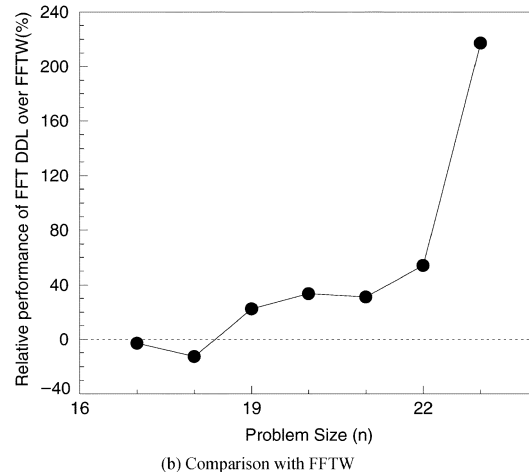
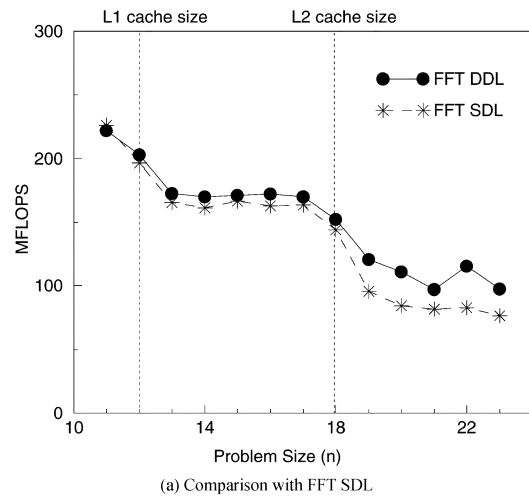


Fig. 14. Performance of FFT DDL on UltraSPARC III.

III, FFT DDL shows improved performance when the problem size is larger than the  $L2$  cache size, as shown in Figs. 13 and 14. These performance improvements are obtained due to a reduction in the number of cache misses as a result of data reorganization. Note that after data reorganization, the number of floating-point operations for computing a FFT of given size does not change.

We also applied our approach to the WHT package developed at CMU [3], which is called WHT SDL in this paper. WHT SDL consists of a set of straight unrolled codes for small WHT, which are similar to the codelets in FFTW. Our WHT package is called WHT DDL. In all our experiments, the data points are double precision (8 Bytes). Fig. 15 shows the computation time per point on four different platforms. We achieved performance gains in WHT similar to those in FFT.

Table V shows the optimal trees chosen by dynamic programming for WHT DDL and WHT SDL on Alpha 21264. In Table V, “ $n$ ” is a leaf node in a tree. It represents a straight line unrolled code for a  $2^n$ -point WHT. “wht[ $n1, n2$ ]” represents a tree of the WHT, factorized as  $2^{n1} \times 2^{n2}$  using the SDL approach. “whtddl[ $n1, n2$ ]” represents the (factorized) WHT tree using the DDL approach. For a WHT smaller than  $2^{14}$ , all data points reside in the cache. Therefore, the WHT is efficiently computed without any cache conflict misses. Hence, the search

algorithm of WHT DDL selects the same tree as that selected by WHT SDL. For problems larger than  $2^{14}$ , WHT DDL achieves better performance than WHT SDL. Table VI also shows the comparison of optimal trees of FFT SDL with that of FFT DDL on MIPS R10000. In the table, “ $n$ ” is denoted as a  $2^n$ -point unfactorized DFT. Cooley–Tukey algorithms using static and dynamic data layouts, which are factorized into  $2^{n1} \times 2^{n2}$ , are represented as “ct[ $n1, n2$ ]” and “ctddl[ $n1, n2$ ]”, respectively.

## VI. CONCLUDING REMARKS

In this paper, we developed an efficient approach based on cache-conscious algorithm design techniques to implement factorized signal transforms. The proposed approach, which dynamically reorganizes the data layout to improve cache performance, is a high-level optimization technique. Our approach exploits the characteristics of the memory hierarchy based on cache behavior analysis for the data access patterns of factorized signal transforms. In addition, our approach can be applied as a high-level optimization on the top of signal transform packages. In this paper, we applied our approach to FFT and WHT packages. Our simulation and execution results illustrate that our cache-conscious dynamic data layout approach achieved significant performance improvements.

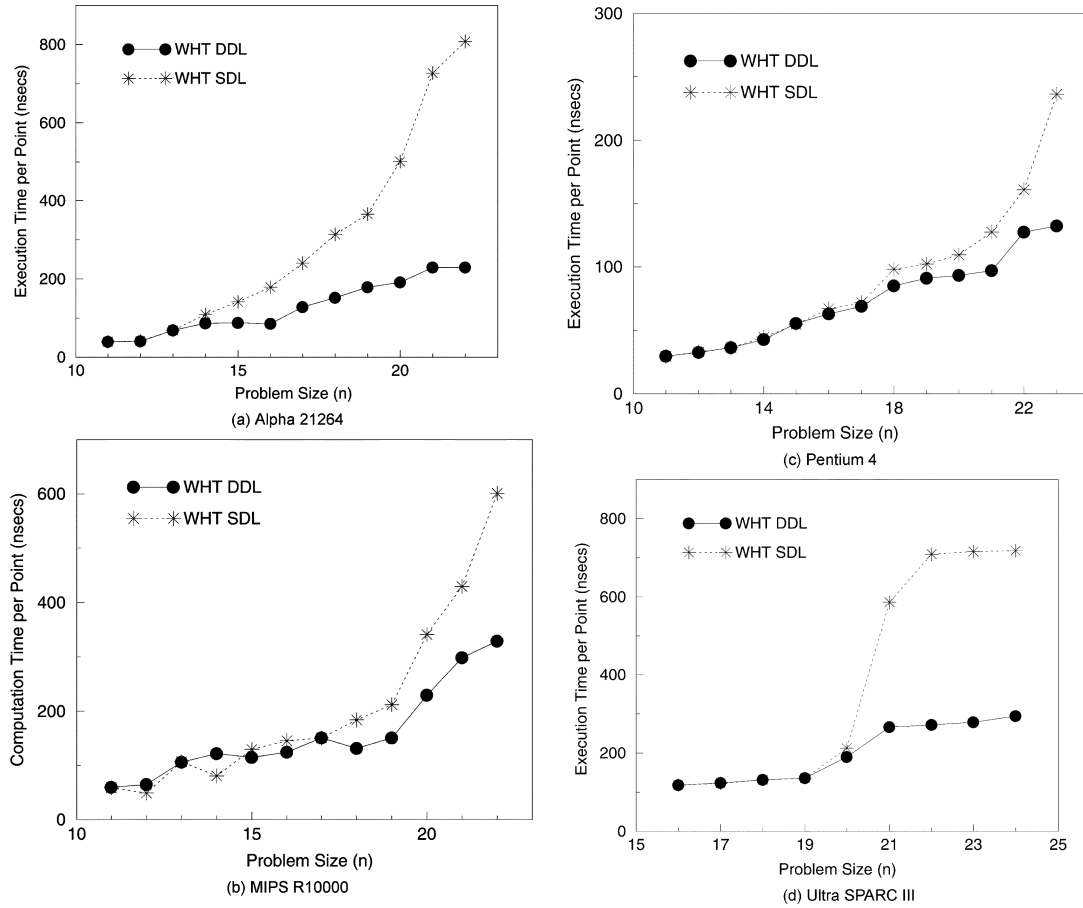


Fig. 15. Performance comparison of WHT implementations.

TABLE V  
OPTIMAL FACTORIZATIONS OF WHT ON ALPHA 21264

WHT Size ( $n$ )	WHT SDL [3]	WHT DDL
11	wht[3,wht[4,4]]	wht[3,wht[4,4]]
12	wht[4,wht[4,4]]	wht[4,wht[4,4]]
13	wht[5,wht[4,4]]	wht[5,wht[4,4]]
14	wht[1,wht[5,wht[4,4]]]	whtddl[wht[3,4],wht[3,4]]
15	wht[5,wht[2,wht[4,4]]]	whtddl[wht[4,3],wht[4,4]]
16	wht[5,wht[3,wht[4,4]]]	whtddl[wht[4,4],wht[4,4]]
17	wht[1,wht[5,wht[3,wht[4,4]]]]	whtddl[wht[4,4],wht[4,5]]
18	wht[1,wht[1,wht[5,wht[3,wht[4,4]]]]]	whtddl[wht[4,5],wht[4,5]]
19	wht[4,wht[5,wht[2,wht[4,4]]]]]	whtddl[wht[4,5],wht[2,wht[4,4]]]
20	wht[4,wht[5,wht[3,wht[4,4]]]]]	whtddl[wht[2,wht[4,4]],wht[2,wht[4,4]]]
21	wht[1,wht[4,wht[5,wht[3,wht[4,4]]]]]]]	whtddl[wht[2,wht[4,4]],wht[3,wht[4,4]]]
22	wht[3,wht[4,wht[5,wht[2,wht[4,4]]]]]]]	whtddl[wht[3,wht[4,4]],wht[3,wht[4,4]]]

TABLE VI  
OPTIMAL FACTORIZATIONS OF FFT ON MIPS R10000

FFT Size ( $n$ )	FFT SDL	FFT DDL
12	ct[4,ct[4,4]]	ct[4,ct[4,4]]
13	ct[5,ct[4,4]]	ct[5,ct[4,5]]
14	ct[4,ct[5,5]]	ctddl[ct[3,4],ct[3,4]]
15	ct[5,ct[5,5]]	ctddl[ct[3,4],ct[4,4]]
16	ct[5,ct[3,ct[4,4]]]	ct[5,ct[4,ct[3,4]]]
17	ct[5,ct[4,ct[4,4]]]	ctddl[ct[4,3],ct[5,5]]
18	ct[2,ct[6,ct[5,5]]]	ctddl[6,ctddl[6,6]]
19	ct[4,ct[1,ct[4,ct[4,6]]]]	ctddl[ct[4,5],ct[5,5]]
20	ct[3,ct[5,ct[4,ct[4,4]]]]	ctddl[ct[5,5],ct[5,5]]
21	ct[4,ct[3,ct[4,ct[4,6]]]]	ctddl[ct[3,4],ctddl[ct[3,4],ct[3,4]]]
22	ct[4,ct[ct[3,ct[3,6]],6]]	ctddl[ct[2,ct[4,5]],ct[2,ct[4,5]]]

The work in this paper is part of the Signal Processing Algorithms Implementation Research for Adaptable Libraries (SPIRAL) project [8], [34]. It is a collaborative project involving Carnegie Mellon University, Drexel University, MathStar Inc., the University of Illinois at Urbana Champaign, and USC. The SPIRAL project is developing a unified framework for the 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. Multilevel performance models and benchmarking data are utilized to explore the algorithm and implementation search space using machine learning techniques.

#### ACKNOWLEDGMENT

The authors would like to thank the SPIRAL Project members for many helpful discussions. In particular, they thank M. Püschel, J. Johnson, D. Padua, B. Singer, J. Moura, and M. Veloso. They also thank G. Govindu, B. Hong, and B. Gundala for their editorial assistance.

#### REFERENCES

- [1] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, vol. 3, 1998.
- [2] K. S. Gatin and L. Carter, "Faster FFT's via architecture cognizance," in *Proc. Int. Conf. Parallel Architectures Compilation Techniques*, Oct. 2000.
- [3] J. Johnson and M. Püschel, "In search of the optimal Walsh-Hadamard transform," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, June 2000.
- [4] D. Mirković, R. Mahassom, and L. Johnsson, "An adaptive software library for fast fourier transforms," in *Proc. Int. Conf. Supercomputing*, May 2000.
- [5] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. 40th Symp. Foundations Comput. Sci.*, 1999.
- [6] D. H. Bailey, "Unfavorable strides in cache memory systems," *Sci. Program.*, vol. 4, 1995.

- [7] G. Haentjens, "An Investigation of Recursive FFT Implementations," Master's, Dept. Elect. Comput. Eng., Carnegie Mellon Univ., Pittsburgh, PA, 2000.
- [8] SPIRAL Project [Online]. Available: <http://www.ece.cmu.edu/~spiral>
- [9] 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 Comput.*, Dec. 1996.
- [10] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. ASPLOS IV*, Apr. 1991.
- [11] E. Torrie, M. Martonosi, C.-W. Tseng, and M. W. Hall, "Characterizing the memory behavior of compiler-parallelized applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, pp. 1224–1237, Dec. 1996.
- [12] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Second ed. San Francisco, CA: Morgan Kaufmann, 1996.
- [13] P. R. Panda, H. Nakamura, N. Dutt, and A. Nicolau, "Augmenting loop tiling with data alignment for improved cache performance," *IEEE Trans. Comput.*, vol. 48, pp. 142–149, Feb. 1999.
- [14] G. Rivera and C.-W. Tseng, "Data transformations for eliminating conflict misses," in *Proc. ACM SIGPLAN Conf. Programming Language Des. Implementation*, June 1998.
- [15] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear array layouts for hierarchical memory systems," in *Proc. 13th ACM Int. Conf. Supercomput.*, June 1999.
- [16] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "Improving locality using loop and data transformations in an integrated framework," in *Proc. 31st IEEE/ACM Int. Symp. Microarchitecture*, Nov. 1998.
- [17] O. Temam, E. D. Granston, and W. Jalby, "To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts," in *Proc. IEEE Supercomput.*, Nov. 1993.
- [18] R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software (ATLAS)," in *Proc. IEEE Supercomput.*, Nov. 1998.
- [19] M. Kandemir and I. Kadayif, "Compiler-dependent selection of dynamic memory layouts," in *Proc. ACM SIGDA/SIGSOFT, Ninth Int. Symp. Hardware/Software Codesign*, Copenhagen, Denmark, Apr. 2001.
- [20] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and E. Ayguadé, "Static and dynamic locality optimizations using integer linear programming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, pp. 922–941, Sept. 2001.
- [21] R. C. Agarwal, F. G. Gustavson, and M. Zubair, "A high performance parallel algorithm for 1-D FFT," in *Proc. Supercomputing*, 1994, pp. 34–40.
- [22] D. H. Bailey, "FFT's in external or hierarchical memory," *J. Supercomput.*, vol. 4, Mar. 1990.
- [23] D. Gannon and W. Jalby, "The influence of memory hierarchy on algorithm organization: Programming FFT's on a vector multiprocessor," in *The Characteristics of Parallel Algorithms*. Cambridge, MA: MIT Press, 1987.
- [24] S. K. S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson, "Implementing fast Fourier transforms on distributed-memory multiprocessors using data redistributions," *Parallel Processing Lett.*, vol. 4, no. 4, pp. 477–488, 1994.
- [25] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*. London, U.K.: Benjamin Cummings, 1993.
- [26] K. R. Wadleigh, "High performance FFT algorithms for cache-coherent multiprocessors," *Int. J. High-Performance Comput. Applicat.*, vol. 13, no. 2, pp. 163–171, Summer 1999.
- [27] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, ser. Frontiers in Applied Mathematics. Philadelphia, PA: SIAM, 1992, vol. 10.
- [28] R. Tolimieri, M. An, and C. Lu, *Algorithms for Discrete Fourier Transforms and Convolution*. New York: Springer, 1997.
- [29] A. V. Oppenheim and R. W. Schaffer, *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
- [30] J.-W. Hong and H. T. Kung, "I/O complexity: The red-blue pebbling game," in *Proc. 13th Ann. ACM Symp. Theory Comput.*, 1981, pp. 326–333.
- [31] A. Ailamaki, D. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *Proc. 27th Int. Conf. Very Large Data Base*, 2001.
- [32] [Online]. Available: <http://www.sun.com/microelectronics/shade/>

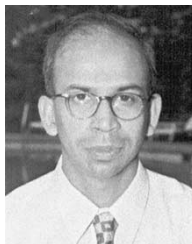
- [33] S. Egner, "Zur Algorithmischen Zerlegungstheorie Linearer Transformationen Mit Symmetrie," Ph.D. dissertation, Univ. Karlsruhe, Karlsruhe, Germany, 1997.
- [34] J. Moura, J. Johnson, R. W. Johnson, D. Padua, V. Prasanna, M. Püschel, and M. Veloso, "SPIRAL: Automatic implementation of signal processing algorithms," in *Proc. High-Performance Embedded Comput.* Cambridge, MA: Lincoln Lab., Mass. Inst. Technol., 2000.



**Neungsoo Park** (M'03) received the B.S. and M.S. degrees in electrical engineering from Yonsei University, Seoul, Korea, in 1991 and 1993, respectively, and the M.S. degree in computer engineering and the Ph.D. degree in electrical engineering from the University of Southern California, Los Angeles, in 2002.

Currently, he is an assistant professor with the Department of Computer Science and Engineering, Konkuk University, Seoul. Prior to joining Konkuk University, he was a senior engineer with Samsung Electronics Corporation, Seoul. His research interests

include parallel computation, computer architecture, high-performance computing for signal processing, VLSI computations, and multimedia systems.



**Viktor K. Prasanna** (F'96) received the B.S. degree in electronics engineering from the Bangalore University, Bangalore, India, the M.S. degree from the School of Automation, Indian Institute of Science, Bangalore, and the Ph.D. degree in computer science from the Pennsylvania State University, University Park, in 1983.

Currently, he is a Professor with the Department of Electrical Engineering as well as in the Department of Computer Science, the University of Southern California (USC), Los Angeles. He is also an associate member of the Center for Applied Mathematical Sciences (CAMS) at USC. He served as the Division Director for the Computer Engineering Division from 1994 to 1998. His research interests include parallel and distributed systems, embedded systems, configurable architectures, and high-performance computing. He has published extensively and consulted for industries in the above areas.

Dr. Prasanna has served on the organizing committees of several international meetings on VLSI computations, parallel computation, and high-performance computing. He is the Steering Co-chair of the International Parallel and Distributed Processing Symposium [formerly the IEEE International Parallel Processing Symposium (IPPS) and the Symposium on Parallel and Distributed Processing (SPDP)] and is the Steering Chair of the International Conference on High-Performance Computing (HiPC). He serves on the editorial boards of the *Journal of Parallel and Distributed Computing* and the *PROCEEDINGS OF THE IEEE*. He is the Editor-in-Chief of the *IEEE TRANSACTIONS ON COMPUTERS*. He was the founding Chair of the IEEE Computer Society Technical Committee on Parallel Processing.