A Prototypical Self-Optimizing Package for Parallel Implementation of Fast Signal Transforms

Kang Chen and J. R. Johnson
Mathematics and Computer Science
Drexel University, Philadelphia, PA 19104
kchen, jjohnson@mcs.drexel.edu

Abstract

This paper presents a self-adapting parallel package for computing the Walsh-Hadamard transform (WHT), a prototypical fast signal transform, similar to the fast Fourier transform. Using a search over a space of mathematical formulas representing different algorithms to compute the WHT, the package finds the best parallel implementation on a given shared-memory multiprocessor. The search automatically finds the best combination of sequential and parallel code leading to the effective granularity, load balance, and cache utilization. Experimental results are presented showing the optimizations required to obtain nearly linear speedup on a sample symmetric multiprocessor.

1 Introduction

In [6] a package for automatically implementing and optimizing the Walsh-Hadamard transform (WHT) was presented. The WHT is a prototypical digital signal processing (DSP) transform with applications to signal and image processing [1] and coding theory [8]. Fast algorithms for computing the WHT are similar to the fast Fourier transform (FFT) and its variants [7]. The only difference is that there are no twiddle factors and bit-reversal. The lack of these extra complications allows us to focus on the role of different divide and conquer strategies and data access patterns as they relate to performance.

The package provides a flexible software architecture that can be configured to implement many different algorithms, with potentially different performance, for computing the WHT. Algorithmic choices are represented by a simple grammar which provides mathematical formulas corresponding to different algorithms. Automatic optimization is performed by searching through the space of WHT formulas for the formula that leads to the best performance. This package and method of self-adaptation is similar to the approach used by FFTW [3], a well-known and efficient package for computing the FFT.

In this paper we extend the package so that it can generate and search for optimal parallel implementations. Currently the generated programs are expressed using OpenMP [10] and are applicable to shared-memory multiprocessors – in particular symmetric multiprocessors (SMPs). The automatic optimization techniques allow us to search for the appropriate combination of parallel and sequential code that produces the best combination of granularity, load balance, and cache performance. The optimal program found obtains nearly linear speedup on a 12 processor IBM S80 [5].

In Section 2 we review the WHT, the sequential WHT package, and the techniques of [11] for reducing cache misses. The following section summarizes the sequential performance obtained on a single node of the S80. In Section 4 we introduce the parallel extensions to the package and discuss the various performance optimizations that are available. The corresponding parallel performance and results of our automatic optimization are presented in Section 5. Conclusions and future work are summarized in Section 6.

2 Algorithms for Computing the WHT

The WHT applied to a signal x is the matrix-vector product $\mathrm{WHT}_N \cdot x$, where the signal x is represented by a vector of size $N = 2^n$ and the transform WHT_N is represented by an $N \times N$ matrix. The WHT is conveniently defined using the tensor (Kronecker) product. The tensor product of two matrices is the block matrix whose (i, j) block is equal to the (i, j) element of the first matrix multiplied by the second matrix.

$$\mathrm{WHT}_N = \bigotimes_{i=1}^{n} \mathrm{WHT}_2 = \overbrace{\mathrm{WHT}_2 \otimes \cdots \otimes \mathrm{WHT}_2}^{n},$$

where

$$\mathrm{WHT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

For example,

$$\mathrm{WHT}_4 = \mathrm{WHT}_2 \otimes \mathrm{WHT}_2$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}.$$

Algorithms for computing the WHT can be obtained by factorizations of the transform matrix WHT_N. The following factorization encompasses a wide range, $O(7^n)$, of possible algorithms. Let $n = n_1 + \cdots + n_t$ be a partition of the exponent n and let I_m denote the $m \times m$ identity matrix.

$$\mathrm{WHT}_N = \prod_{i=1}^{t} (\mathrm{I}_{2^{n_1+\cdots+n_{i-1}}} \otimes \mathrm{WHT}_{2^{n_i}} \otimes \mathrm{I}_{2^{n_{i+1}+\cdots+n_t}}). \tag{1}$$

Each of the recursive computations $WHT_{2^{n_i}}$ can be computed with a similar factorization. The entire computation can be denoted by a partition tree, where the root is labeled by n and the children by $n_1, \ldots, n_t$. The standard iterative algorithm is obtained by setting $t = n$ and $n_i = 1$ for $i = 1, \ldots, t$. The standard recursive program is obtained by setting $t = 2$ with $n_1 = 1$ and $n_2 = n - 1$ with the same substitution done recursively for $WHT_{2^{n-1}}$. The general factorization combines various amounts of recursion and iteration.

Let $\mathrm{N} = \mathrm{N}_1 \cdots \mathrm{N}_t$, where $\mathrm{N}_i = 2^{n_i}$, and let $x^M_{b,s}$denote the vector $(x(b), x(b+s), \cdots, x(b+(M-1)s))$. A sequential implementation of the factorization in Equation 1 is as follows. This scheme assumes that the algorithm works in-place and is able to accept stride parameters.

```
R = N; S = 1;
for i = 1, ..., t
    R = R / $N_i$;
    for j = 0, ..., R - 1
        for k = 0, ..., S - 1
            $x^{N_i}_{jN_iS+k,S} = \mathrm{WHT}_{N_i} * x^{N_i}_{jN_iS+k,S}$;
    S = S * $N_i$;
```

While all factorizations have exactly the same arithmetic ($N \log(N)$ operations), different factorizations lead to algorithms with different data access patterns and consequently can have vastly different performance.

In [6] a package for computing the WHT based on these ideas was presented. The package is available from http://www.ece.cmu.edu/~spiral, and was developed as part of the SPIRAL project [9]. Particular algorithms, corresponding to instances of Equation 1, are represented by a tree data structure corresponding to the associated partition tree. Leaf nodes correspond to straight-line code (used to reduce recursion and iteration overhead). Internal nodes, called *split* nodes, correspond to applications of Equation 1. WHT trees can be described using a grammar with the keyword **split** for internal nodes and **small** for leaf nodes.

The optimal tree for a given size, corresponding to the fastest implementation, is a combination of recursion, iteration and straight-line code. The optimal tree is architecture specific and is determined automatically using a search based on dynamic programming (DP). DP does not necessarily return the optimal tree since the dynamic programming assumption may be violated. Nonetheless, experience shows that it usually returns a tree with very good performance. Dynamic programming, usually restricted to binary trees, is used since exhaustive search is too costly. Alternative search methods, that do not require the dynamic programming assumption, have been explored in [12].

The factorization in Equation 1 is a product of matrices of the form $I \otimes A \otimes I$. In the special case of binary trees all factors are of the form $I \otimes A$ (parallel form) and $A \otimes I$ (vector form) [7]. The vector form accesses the data at stride and consequently can introduce conflict misses when the stride is large [11].

It is possible to convert the vector form to a parallel form by dynamically permuting the data. If A is an $n \times n$ matrix, then $A \otimes I_m = L^{mn}_n (I_m \otimes A) L^{mn}_m$, where L^{mn}_m is a permutation called a stride permutation since it gathers the elements of a vector at stride m [7]. Since L^{mn}_n is the inverse of L^{mn}_m this transformation corresponds to relabeling the input and output data. Since the relabeling is performed at runtime, it has been called Dynamic Data Layout (DDL) [11]. Introducing DDL may reduce the runtime since the parallel form of the tensor product accesses data consecutively and consequently reduces cache misses. Whether or not the runtime is reduced depends on the cache miss penalty as compared to the overhead of performing the runtime permutations.

The option of using DDL in the WHT package was introduced in [11] by introducing an additional internal node called a *splitddl* node. In order to conform to the in-place computation used in the WHT package, an alternative permutation rather than a stride permutation was used (the permutation that converts the vector form to the parallel form is not unique). A permutation of order two, which can be performed in-place was selected. Assume $N = R * S$ and $R \leq S$ (since $N = 2^n$, R divides S). The $R * S$ data vector viewed as an $R \times S$ matrix is divided into S/R square matrices in size of $R \times R$. After division, transpose is performed individually on each square matrix. This procedure is called in-place pseudo-transpose. Since it is of order two, the same process is used to perform the inverse computation.

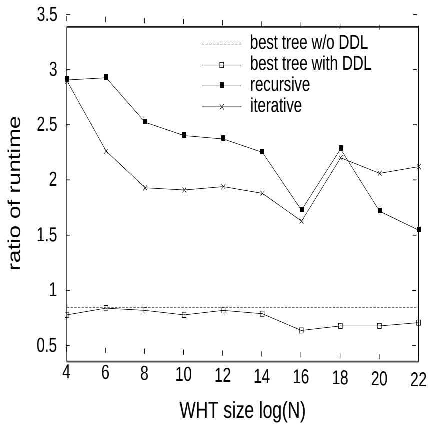

Figure 1. Ratio of runtime to the best WHT without DDL

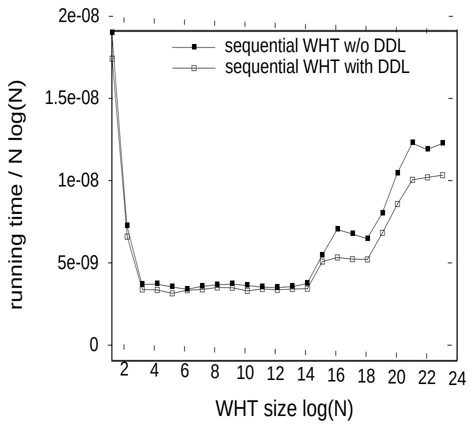

Figure 2. Normalized runtime at different sizes

3 Sequential Performance of the WHT

To prepare for our parallel study we analyzed the performance of the sequential package on a single node of our SMP. We used an IBM S80 RS64 III with twelve 450 MHz CPUs. Each CPU has separate 128 KB L1 data and instruction caches and an 8 MB L2 cache. There is a total of 8 GB of memory. The operating system was AIX 4.3.3, and we used version 5.0.5 of the IBM C compiler with flags set to -O5 -q64.

Figure 1 shows the relative running times for four WHT algorithms: recursive, iterative, and the best algorithms found by DP with and without DDL. Both the recursive and iterative algorithms use WHT$_4$ instead of WHT$_2$ for leaf nodes. All times are presented as ratios compared to the best tree without DDL. Several key observations should be made. First, the iterative algorithm is initially faster than the recursive algorithm, but the recursive algorithm becomes faster at $N = 2^{18}$. The iterative algorithm has less control overhead than the recursive algorithm, but the recursive algorithm exhibits better locality. Hence as the cache boundaries are crossed the influence of the cache becomes more important and at size 2^{18} the benefit of better cache utilization outweighs the additional control overhead. Second, Figure 1, shows that the trees found by DP are at least two times faster than either the recursive or iterative algorithms. Finally DDL improves performance by about 15% when $N \geq 2^{16}$.

Fig. 4(a) and Fig. 4(b) displays the best WHT tree with and without DDL for size 2^{23}. The tree with DDL is more balanced than the tree without DDL. This improves performance since nodes of size 2^{11} and 2^{12} used by DDL fit in the L1 cache, whereas the node of size 2^{19} used by the non-DDL version exceeds the L1 cache.

The influence of the cache on performance is better seen in a plot of runtimes normalized by $n\lg(n)$, the number of arithmetic operations. Normalized runtime is independent of data size, and it reflects the cost of different data access patterns with respect to the underlying architecture. Figure 2 shows the normalized runtime of the WHT program. The three plateaus in the figure are related to the L1 and L2 caches (the L1 data cache is 128 KB or 2^{14} doubles, and the L2 combined cache is 8 MB or 2^{20} doubles). When the data size exceeds the cache boundary, additional costs due to cache misses account for the jumps in runtime. Fig. 2 indicates that an uneven partition of a large WHT node may result in a node in the first plateau and another node in the second plateau, which implies more cache misses and a greater runtime. A better binary partition would be built from nodes within the first plateau. This favors balanced trees.

In order to minimize the overhead of performing the pseudo-transpose required by DDL blocking is used. When blocking is used, the input, viewed as a matrix, is organized into blocks and all of the elements within a block are transposed prior to moving to the next block. Prefetching can take advantage of the localized data access pattern provided by blocking to reduce the number of cache misses. Blocking introduces the control overhead of two additional loops, but it may improve cache efficiency and thus overall performance.

Most modern caches are organized into blocks called cache lines. When a single data element in the cache line is accessed for the first time, the entire cache line is brought into cache. Thus additional accesses to the same cache line will not cause cache misses. When data is accessed with a large stride, the cache line may be replaced before adjacent elements are accessed. When blocking is used this problem can be avoided. The optimal block size depends on the cache line size and the problem size and stride. For some sizes and strides, it is beneficial to have a block size smaller than the cache line size. The optimal block size can be determined empirically using search. For the S80 the optimal block size ranged from 64 to 256 bytes.

4 Parallel WHT Package using OpenMP

The WHT package was extended to support parallel computation on shared-memory SMPs. The package obtains parallelism through the use of parallel *split* and *splitddl* nodes, and is optimized by searching for the best use of these nodes. Additional optimization is obtained by tuning the implementation of these nodes. The tuning process can be automated by searching over a set of implementation parameters. The search process automatically optimizes granularity, load balance, cache utilization, and the selection of appropriately optimized sequential code.

Parallel code was obtained using OpenMP [10], a parallel programming model for shared-memory multiprocessors. OpenMP is comprised of a set of compiler directives and a small supporting library of subroutines. The directives describe the desired parallelism of the source code (C/C++ or Fortran) to any OpenMP supporting compiler. The book [2] provides a good introduction to the design of parallel programs using OpenMP.

The *parallel split* node is similar to the *split* node except that the work is distributed over a collection of parallel threads. Additional code is required to create, manage, and synchronize the threads.

```
#begin parallel region
R = N; S = 1; id = get_thread_id();
num = get_total_thread();
for i = 1, ..., t
    R = R / $N_i$;
    for id = id, ..., R * S - 1, step = num
        j = id / S;
        k = id mod S;
        $x^{N_i}_{jN_iS+k,S} = \mathrm{WHT}_{N_i} * x^{N_i}_{jN_iS+k,S}$;
    S = S * $N_i$;
    #parallel barrier
#end parallel region
```

The inner loop allocates the work (recursive WHT applications) for each stage in the factorization in Equation 1. Since the input from each stage depends on the output from the previous stage, a barrier synchronization is inserted between stages. Alternatively, new threads could be created and joined each iteration of the outer loop with the use of a parallel region. This would simplify the code, but would add substantial overhead.

The *parallel splitddl* node is comprised of four stages corresponding to the factorization $L(I_S \otimes \mathrm{WHT}_R)L(I_R \otimes \mathrm{WHT}_S)$, where L is the pseudo-transpose operation. In order to obtain good efficiency all four stages must be parallelized individually with barriers inserted between the stages. Since the $R \times R$ block transpose operations in the pseudo-transpose are independent, they can be performed in parallel. This coarse-grained DDL works well if there are sufficiently many square matrices for the threads to work on. However, this is not always the case. In a balanced split with $R \approx S$, the number of square matrices S/R is small, and the size of each square matrix is very big. This leads to a poorly balanced workload and degrades parallel performance. In such cases, a fine grained version can be obtained by parallelizing the blocking technique used during the transpose of the square matrices in the sequential implementation.

In the sequential DDL, data are transposed in small blocks except those on the diagonal. So it is possible to distribute the transpose tasks at the block level. The small granularity ensures good workload balance among the threads. The fine-grained parallel DDL incurs additional overhead, but the cost is negligible compared with the benefit of better load balance. Since the square matrices are independent, barrier synchronization is not required. As soon as a thread finishes its work on one square matrix it continues in the same row working on the next matrix until all of the blocks in the row are complete. Since the bottom rows contain fewer tasks than the top rows, it is necessary to rotate the rows assigned to a thread as the threads move from one $R \times R$ matrix to the next to completely balance the work.

The tradeoff between parallel versions of *split* and *splitddl* is subject to the same compromises as the sequential versions. In a parallel *split* node, each thread applies WHT_{N_i} on different blocks of data, $x^{N_i}_{jN_iS+k,S}$, of size is N_i at stride S. If the size or the stride exceeds the cache boundary, there will be additional cost due to cache misses as is the case in the sequential program. The penalty is even greater in the parallel case due to interactions amongst the caches of the separate processors. The IBM S80 uses a bus snooping policy to enforce cache coherence [5]. When $S \neq 1$ independent WHT computations access data that is interleaved. When the interleaving is in the same cache line the need to maintain cache coherency introduces extra cache misses and synchronization overhead. This interaction between processors severely degrades performance. Multiple copies of the same cache line will be accessed at the same time by different processors. This is the case even though the data accessed is different (only the cache lines are shared). This situation can be avoided by using a parallel version of a *splitddl* node, where access is at stride one. However, there is additional overhead due to the parallel pseudo-transpose.

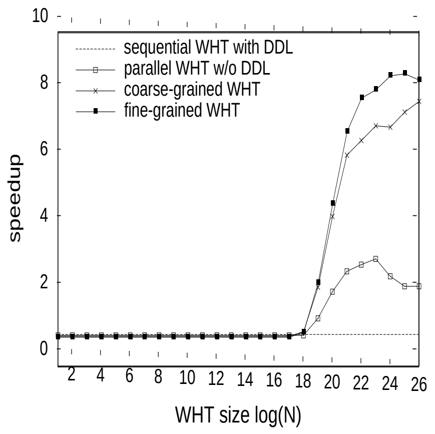

Figure 3. Speedup of parallel WHT using 10 threads

5 Parallel Performance of the WHT

Dynamic programming was used to find the best parallel algorithms, built using parallel split (p_split) and splitddl (p_splitddl) nodes, for computing the WHT using ten threads on the IBM S80 (the number of threads can be varied during the search). Programs were compiled with IBM's OpenMP compiler, cc_r, with flags set to -qsmp=omp -O5 -q64. The best trees and running times for data size up to 2^{26} were recorded. To compare the performance of the different parallel implementations, p_split nodes, p_splitddl nodes with coarse-grained pseudo-transpose, and the p_splitddl nodes with fine-grained pseudo-transpose were incorporated separately into the sequential WHT package and tested individually.

Figure 3 shows the speedup of the parallel WHTs for sizes from 2 to 2^{26}. Speedup is computed as the ratio of the running time of the best sequential WHT to the running time of the parallel WHT. The resulting best WHT partition trees show that the parallel split nodes replaces the sequential split node as the root node when the data size is larger than 2^{18}. That means, after this point, the benefit of sharing work among multiple threads offsets the parallel overhead. Parallelism becomes more and more beneficial as the data size increases. The best efficiency obtained (speedup/number of threads) was approximately 90%. Observe that the speedup obtained by the parallel WHT without DDL is less than three. That means DDL greatly improves the performance of the parallel WHT. The improvement, which is as high as four at 2^{26}, is substantially more significant than in the sequential case.

Table 1 compares the performance of the parallel WHT with coarse-grained DDL and with fine-grained DDL. The speedup is calculated based on the best runtime for sequential WHT at size 2^{26}, which is 20.89 sec. The efficiency is the speedup divided by the number of threads involved in the computation.

Table 1. Performance of parallel WHT with DDL at size 2^{26}

Thread	Coarse-grained DDL			Fine-grained DDL		
	Time(sec)	Speedup	Eff. (%)	Time(sec)	Speedup	Eff. (%)
1	24.34	0.86	86	20.35	1.03	103
2	12.36	1.69	85	10.33	2.02	101
3	8.38	2.49	83	6.97	3.00	100
4	6.24	3.35	84	5.22	4.00	100
5	5.10	4.10	82	4.24	4.93	99
6	4.27	4.89	82	3.58	5.84	97
7	3.69	5.66	81	3.11	6.72	96
8	3.25	6.43	80	2.75	7.60	95
9	2.97	7.03	78	2.5	8.36	93
10	2.78	7.51	75	2.27	9.20	92

To understand the improvement due to fine-grained DDL it is helpful to look at the trees that were selected. The best tree of size 2^{26} with coarse-grained DDL is p_splitdll[split[small[4], small[5]], splitddl[small[8], split[small[4], small[5]]]], while the best tree with fine-grained DDL is p_splitdll[split[small[6], small[7]], split[small[6], small[7]]].

Recall that our analysis of the sequential performance (see Figure 2) indicated that balanced partitions are preferred to unbalanced partitions. However, for the parallel WHT with coarse-grained DDL, balanced partitions lead to suboptimal load balance. Since profiling data indicates the pseudo-transpose takes more than 20% of the WHT runtime, too many threads being idle at this step is an efficiency bottleneck. Therefore, even though uneven partitions might lead to poor performance, they are still better off than the close-to-even partitions. As a result of this trade-off, the best trees found by DP search tend to be uneven but not extremely uneven (see Fig. 4(d)). This result indicates that coarse-grained parallelism has a strong preference to uneven partitions, and this preference becomes a restriction imposed on the DP search and consequently leads it to find suboptimal trees.

This problem does not occur for the fine-grained implementation of DDL. Since load balancing of the pseudo-transpose is no longer a problem, DP is free to select a parallel tree built from the best sequential trees. Consequently, the best trees found have a similar structure to the best sequential trees as is seen in Figure 4.

Similar to the case with the sequential implementation of DDL, it is important to adjust the block size for the fine-grained parallel DDL. First, the block size should not be too big, otherwise there won't be enough blocks to be assigned to the threads. Second, the block size should not be smaller than the cache line size. A small block size may severely

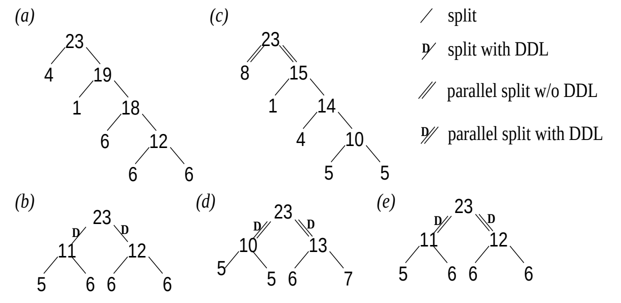

Figure 4. Best WHT trees at data size 2^{23}. (a)sequential WHT without DDL, (b)sequential WHT with DDL, (c)parallel WHT without DDL, (d)parallel WHT with coarse-grained DDL, (e)parallel WHT with fine-grained DDL

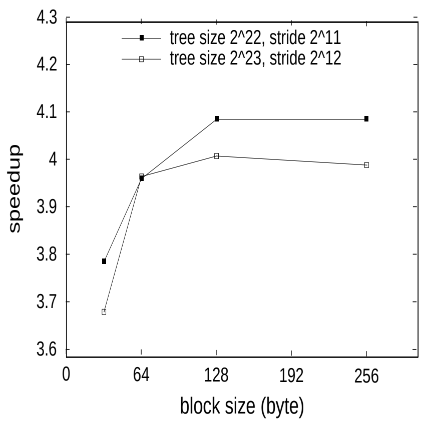

Figure 5. The effect of block size on parallel performance

reduce parallel performance. If two threads are working on two different blocks sharing the same cache line, the two processors will compete for the "exclusive read/write" status of the cache line, which results in "ping pong"—a performance hazard in SMP systems. Figure 5 shows the effect of block size on parallel performance with four threads. As the size decreases, the parallel performance deteriorates quickly. We can expect the situation to become even worse if more threads are involved.

The parallel WHT package was also installed on a 4 processor Sun Enterprise 450. On this machine the benefits of DDL (see http://www.ece.cmu.edu/˜spiral) did not compensate for the extra overhead, and consequently parallelism based on regular split nodes was selected. To obtain better performance, the techniques of [4] might be used to improve cache performance without the overhead of DDL.

6 Conclusion

In this paper, we presented a shared-memory parallel version of a package for implementing efficient Walsh-Hadamard transforms (a prototypical fast signal transform). The package uses search to automatically optimize the implementation. The search process is used to adjust granularity, load balance and cache utilization. Empirical performance data on the IBM S80 showed nearly linear speedup for the optimal implementations discovered by the package. It is important to note that if the search space does not allow enough algorithmic choices (e.g. DDL with fine-grained parallel pseudo-transpose) then suboptimal code will be found. In the future we will obtain additional data for other SMPs and extend our package to work on distributed memory multiprocessors. The package and updated performance data can be obtained at http://www.ece.cmu.edu/˜spiral.

References

[1] K. Beauchamp. *Applications of Walsh and related functions*. Academic Press, 1984.

[2] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufman Publishers, 2000.

[3] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *ICASSP '98*, volume 3, pages 1381–1384, 1998. `http://www.fftw.org`.

[4] K.-S. Gatlin and L. Carter. Faster FFTs via architecture-cognizance. In *Proceedings of PACT 2000*, Oct. 2000.

[5] IBM. The RS/6000 enterprise server model s80, technology and architecture. Technical report. `http://www.rs6000.ibm.com/resource/technology/s80techarch.html`.

[6] J. Johnson and M. Püschel. In search of the optimal Walsh-Hadamard transform. *ICASSP*, 2000.

[7] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circuits, Systems, and Signal Processing*, 9(4):449–500, 1990.

[8] F. MacWilliams and N. Sloane. *The theory of error-correcting codes*. North-Holland Publ.Comp., 1992.

[9] J. M. F. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, and M. M. Veloso. SPIRAL: Portable Library of Optimized Signal Processing Algorithms, 1998. http://www.ece.cmu.edu/˜spiral.

[10] OpenMP. *OpenMP C and C++ Application Pragram Interface, Version 1.0*, 1998. `http://www.openmp.org`.

[11] N. Park and V. K. Prasanna. Cache conscious Walsh-Hadamard transform. *ICASSP*, 2001.

[12] B. Singer and M. Veloso. Stochastic search for signal processing algorithm optimization. In *Proc. Supercomputing*, Nov. 2001.