CACHE CONSCIOUS WALSH-HADAMARD TRANSFORM

Neungsoo Park and Viktor K. Prasanna

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562, USA
{neungsoo, prasanna}@usc.edu

ABSTRACT

The Walsh-Hadamard Transform (WHT) is an important algorithm in signal processing because of its simplicity. However, in computing large size WHT, non-unit stride access results in poor cache performance leading to severe degradation in performance. This poor cache performance is also a critical problem in achieving high performance in other large size signal transforms.

In this paper, we develop a *cache friendly* technique that improves the performance of large size WHT. In our approach, data reorganization is performed between computation stages to reduce cache pollution. Furthermore, we develop an efficient search algorithm to determine the optimal factorization tree based upon problem size and stride access in the decomposition. Experimental results show that our approach achieves upto 180% performance improvement over the state of the art package on Alpha 21264 and MIPS R10000. In addition, the proposed optimization is applicable to other signal transforms and is portable across various platforms.

1. INTRODUCTION

In this paper, we present a method for implementing and optimizing the Walsh-Hadamard transform. The WHT is an important tool in signal processing and coding theory [1, 2]. Fast algorithms for computing the WHT are similar to the Fast Fourier Transform (FFT) and its variants. Unlike the FFT, twiddle factors and bit-reversal operations are not needed. This simplicity allows us to concentrate on divide and conquer strategies for WHT.

The WHT package developed at Carnegie Mellon University (CMU) [3] is a flexible software architecture that can be configured to implement different algorithmic combinations. This package consists of a set of straight line unrolled codes. The combination of unrolled codes is represented in a tree structure similar to the planner of the FFTW package[4]. This tree-structured computation achieves good performance for small size WHT. However, as the size of WHT exceeds cache size, performance drastically degrades. This degradation is due to the memory hierarchy of state-of-the-art architectures.

In state-of-the-art architectures, processor-memory bandwidth is a bottleneck in achieving high performance. To improve this bandwidth, cache memory is used between memory and the processor. However, the cache in most of the state-of-the-art architectures is either direct-mapped or small set-associative. The distance between successively accessed data is called *stride*. Large stride data accesses during a computation 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 [5]. Such stride accesses occur in the previous approach, which computes a large WHT by decomposing it into a factorization tree. Such accesses result in more cache misses, thus degrading overall performance.

In this paper, we propose a method for implementing and optimizing large size WHT. Our approach is to dynamically reorganize the data layout in the memory between computation stages. Thus, we convert non-unit stride access to unit stride access, thereby reducing cache misses considerably. For small problem sizes, the data reorganization overhead reduces the overall performance of our approach. But as the problem size increases, our technique achieves better performance over the state-of-the-art approach. To achieve high performance for all problem sizes, a decision is to be made at each node of the factorization tree. We developed an efficient search algorithm using dynamic programming to make this decision. Our search algorithm picks the optimal tree and thus efficiently computes the WHT. Our approach achieved performance improvement upto 180% on Alpha 21264 machine and MIPS R10000.

We present a brief review of WHT in Section 2. In Section 3, we briefly describe the state-of-the-art WHT package developed at CMU. In Section 4, the cache behavior of factorized WHT trees is presented. In Section 5, we discuss our approach to optimize WHT and explain the search algorithm. Experimental results on two state-of-the-art plat-

Work supported by the DARPA/DSO OPAL Program, through the Carnegie Mellon University under subcontract number 1-541704-50296.

forms are shown in Section 6. Concluding remarks are made in Section 7.

2. THE WALSH-HADAMARD TRANSFORM

For the sake of completeness, we briefly review WHT. Additional details can be found in [3]. The Walsh-Hadamard transform of a signal x, of size $N = 2^n$, is the matrix vector product $\mathbf{WHT}_N \cdot x$, where

$$\mathbf{WHT}_N = \bigotimes_{i=1}^{n} \mathbf{DFT}_2 = \overbrace{\mathbf{DFT}_2 \otimes \cdots \otimes \mathbf{DFT}_2}^{n}.$$

The matrix

$$\mathbf{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

is the 2-point DFT matrix, and $\otimes$ denotes the tensor or Kronecker product. The tensor product of two matrices is computed by replacing each element of the first matrix with that element multiplied by the second matrix.

Algorithm for computing the WHT can be derived using properties of the tensor product [6]. A recursive algorithm for the WHT is obtained from the factorization

$$\mathbf{WHT}_{2^n} = (\mathbf{WHT}_{2^k} \otimes \mathbf{I}_{2^{n-k}})(\mathbf{I}_{2^k} \otimes \mathbf{WHT}_{2^{n-k}}) \quad (1)$$

This algorithm is similar to the divide and conquer algorithm in a recursive FFT. An iterative algorithm for computing the WHT is obtained from the factorization

$$\mathbf{WHT}_{2^n} = \prod_{i=1}^{n} (\mathbf{I}_{2^{i-1}} \otimes \mathbf{WHT}_2 \otimes \mathbf{I}_{2^{n-i}}), \quad (2)$$

which corresponds to an iterative FFT.

3. PREVIOUS WORK

Fast algorithms to compute WHT are similar to those for the FFT. To achieve high performance, divide and conquer strategies can be applied for efficient implementation of FFT and WHT. Using a recursive algorithm, a highly optimized FFT package known as FFTW [4], was developed at MIT. It consists of a set of straight line unrolled codes of small size FFT, known as *codelets*. To compute a large size FFT, a set of these small size codelets are combined. The combination of codelets is represented as a binary tree. The search space comprising of all such trees is large. To choose the tree yielding the best performance among the available trees, dynamic programming was used.

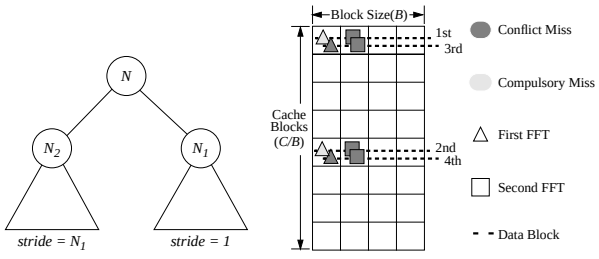
A similar technique was used to develop a package for WHT [3]. In this package, algorithmic choices are represented internally in a tree structure, similar to the plan data structure of the FFTW. However, the WHT package can support iterative and recursive data structures, as well as combinations of 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 for reading WHT expressions and translating them into a tree data structure. Optimizing the WHT becomes a search problem over a large space of available partition trees and dynamic programming was used to prune the search.

The divide and conquer method used in both the algorithms reduces the working set size during computation. However, the performance degrades drastically for large problem sizes. Even though the problem size is reduced by the divide and conquer method, leaf nodes require non-unit stride access during computation. This non-unit stride access of leaf nodes causes a lot of cache misses because of its poor locality in the memory hierarchy of the state-of-the-art machines. In the next section, we discuss the cache behavior of factorized WHT.

4. CACHE BEHAVIOR OF FACTORIZED WHT

To understand the cache behavior of WHT computation with factorization, we consider a two-level memory hierarchy consisting of a cache and main memory. Let C denote the size of the cache memory and B denote the size of the cache block. To simplify our analysis, we assume the cache is direct mapped since most of the state-of-the-art machines have direct mapped or small set associative caches. The number of cache blocks in a direct mapped cache is given by C/B.

A data access can result in a cache hit or a miss. Cache misses may be classified as compulsory or conflict. A *compulsory* miss occurs if the data was never accessed before and needs to be fetched for the first time. A *conflict* miss occurs when the data was previously fetched into the cache block but was replaced because another data access is mapped onto the same cache block. In general, a cache miss requests an expensive memory operation to fetch a data block from the memory into the cache. Therefore, several cache misses result in severe performance degradation.



(a) Factorization tree (b) Data mapped onto cache

Fig. 1. Cache behavior of two consecutive WHTs.

In order to explain the cache behavior of factorized WHT, a simple analysis is presented. Consider the factorization of an N-point WHT as $N_2 \times N_1$ as shown in Figure 1(a). In

the previous approach, N_2 computations of N_1-point WHT are first performed with unit stride. Following this, N_1 computations of N_2 point WHT are computed with stride $N_1 (> 1)$. These non-unit stride accesses in the N_1 computations of N_2-point WHT cause a large number of cache misses.

Consider a typical case where the stride is large such that $stride \times N_2 > C$. The data mapping onto cache for an N_2-point WHT is shown in Figure 1(b). In this example, we consider $N_2 = 4$. The $1^{st}(2^{nd})$ and the $3^{rd}(4^{th})$ data blocks are mapped onto the same cache block. As shown in the figure, conflict misses occur in addition to compulsory misses for such a WHT computation because of its stride access. There can be potentially $N_2 \times N_1$ misses in performing N_1 computations of N_2-point WHTs. Stride data accesses result in *cache pollution*: cache blocks are replaced before all data points in the cache block are fully utilized. Cache pollution and conflicts lead to considerable performance degradation particularly when a large size WHT is computed. In the next Section, we discuss a methodology to reduce these cache misses and thus improve the overall performance.

5. A PORTABLE OPTIMIZED WHT PACKAGE

As illustrated in Section 4, stride data access affects performance. The N_1 computations of an N_2-point WHT, performed in the previous approach with non-unit stride, causes a large number of cache misses.

In our approach, data layout in the memory is reorganized to convert the non-unit stride access of N_2-point WHT to unity. Thus, we can reduce cache conflicts that occur during the N_1 computations. After these computations, reverse reorganizations are performed to correct the order of the data. To achieve improved performance, the number of reorganizations should be minimized. We also have to determine the nodes at which reorganization should be applied.

To develop the search algorithm, we need to define a cost model for the computation of WHT. In the previous approach, size was the only parameter considered in finding the optimal factorization tree. However, as discussed above, stride is also a critical parameter affecting the performance of a large size WHT. For a given $N = N_1 \times N_2$, the minimum cost of performing an N-point WHT is given by :

$$\min_{S_i, S_j} [N_2 \times WHT(N_1, S_i) + Dr(N, L_{S_i}, L_{S_j}) + N_1 \times WHT(N_2, S_j) + Dr(N, L_{S_j}, L_{S_i})], \quad (3)$$

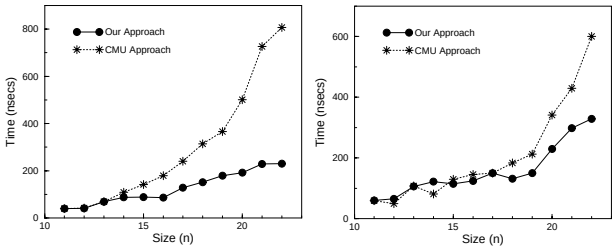
where S_i and $S_j = 1, 2, \cdots$. L_{S_i} (L_{S_j}) denotes the data layout for the computation of an $N_1(N_2)$-point WHT with stride $S_i(S_j)$. $Dr(N, L_{S_i}, L_{S_j})$ is the cost of reorganizing layout L_{S_i} to layout L_{S_j}. When $L_{S_i} = L_{S_j}$, no data reorganization is performed. The cost of performing reorganization is $O(\frac{N}{B})$ memory accesses for WHT of size N. If k different strides are considered for $S_i(S_j)$, the cost of evaluating Eq. (3) is $O(k^2)$.

Optimizing the WHT becomes a search problem over a large space of possible factorization trees. The search space considering only size as the parameter is $O(4^n)$, where $n = \log_2 N$. It is thus impractical to find an optimal tree by exhaustive search, considering both stride and size. We use dynamic programming to reduce the complexity. Restricting our search to a small set of strides reduces the search space. At each node, the search complexity is $O(k^2)$ as explained above. The factorization tree is built bottom-up in dynamic programming. Its complexity is $O(n^2)$. Therefore, the complexity of our algorithm is $O(k^2 n^2)$. The decomposition computed using our technique provides a factorization with optimal execution time including various layouts (having different stride access costs). For a small size WHT, the factorization tree with data reorganization may not be an optimal solution because of the overhead involved. However, for large size WHTs, trees including data reorganization lead to faster WHT implementations.

Our approach is a high level optimization that exploits the characteristics of the memory hierarchy by analyzing the data access pattern, therby making our approach portable. Therefore, this high-level optimization can be applied on top of low-level optimizations such as those employed in FFTW [4] or the CMU package [3].

6. EXPERIMENTAL RESULTS

In this section, we report experimental results conducted on two state-of-the-art platforms. Table 1 summarizes the relevant architectural parameters, compilers, and optimization options for various platforms used in our experiments. We measured the wall clock time using `clock()` function. To obtain the accurate execution time, computations were repeated until the overall execution time is larger than 1 second. The total execution time was obtained by deducting the loop overhead from that time. The average execution time is reported.



(a) Alpha 21264 (b) MIPS R10000

Fig. 2. Execution time per point.

Figure 2 shows the computation time for each point of WHT on both platforms. Figure 2 (a) shows the performance of WHT on Alpha 21264 platform. As all the points required for the computation of WHT reside in the cache for smaller

Table 1. Parameters of the platforms

Processor	Alpha 21264	MIPS R10000
Clock (MHz)	500	195
L1 Cache(KB) / Block (Bytes)	64 / 64	32 / 32
L2 Cache(MB) / Block (Bytes)	4 / 64	4 / 64
OS	Linux	IRIX64 6.5
Compiler	gcc ver. egcs-1.91.66	MIPSpro Compilers: Ver. 7.2.1
Optimization options	-O6 -fomit-frame-pointer -pedantic	-r10000 -O3 -Ofast -mips4

size problems (N less than 2^{14}), the performance of our approach is the same as that of the CMU approach. For sizes between 2^{14} and 2^{19}, the required data can reside in the $L2$ cache. In this range, our approach produces a marginal gain as it reduces the $L1$ misses. But as the problem size exceeds the size of the $L2$ cache, a 3.52 times speed up is obtained since both $L1$ and $L2$ caches are efficiently utilized in our approach. Figure 2 (b) shows the performance improvement on the MIPS R10000 platform. On other platforms such as AMD Athlon and SUN UltraSPARC IIs, benchmark results using our WHT package are available at our SPIRAL (Signal Processing Algorithms Implementation Research for Adaptive Library) [7] web site.

Table 2 shows the optimal trees chosen by pruning the search space in both approaches. In Table 2, "[n]" is a straight line unrolled code for 2^n-point WHT. "[[n1], [n2]]" represents a tree computation using CMU approach and "d[[n1],[n2]]", a tree computation using our approach. For WHT cases smaller than 2^{14}, all points of WHT reside in the cache. So cache misses do not occur during the computation. Hence our search algorithm selects the same tree as that of the CMU approach. For problem sizes larger than 2^{14}, the tree based on our approach achieves better performance. The optimal tree for any size is selected automatically by our dynamic programming algorithm.

Table 2. Optimal decompositions on Alpha 21264

Size	CMU Approach [3]	Our Approach
11	[[3],[[4],[4]]]	[[3],[[4],[4]]]
12	[[4],[[4],[4]]]	[[4],[[4],[4]]]
13	[[5],[[4],[4]]]	[[5],[[4],[4]]]
14	[[1],[[5],[[4],[4]]]]	d[[[3],[4]],[[3],[4]]]
15	[[5],[[2],[[4],[4]]]]	d[[[4],[3]],[[4],[4]]]
16	[[5],[[3],[[4],[4]]]]	d[[[4],[4]],[[4],[4]]]
17	[[1],[[5],[[3],[[4],[4]]]]]	d[[[4],[4]],[[4],[5]]]
18	[[1],[[1],[[5],[[3],[[4],[4]]]]]]	d[[[4],[5]],[[4],[5]]]
19	[[4],[[5],[[2],[[4],[4]]]]]	d[[[4],[5]],[[2],[[4],[4]]]]
20	[[4],[[5],[[3],[[4],[4]]]]]	d[[[2],[[4],[4]]],[[2],[[4],[4]]]]
21	[[1],[[4],[[5],[[3],[[4],[4]]]]]]	d[[[2],[[4],[4]]],[[3],[[4],[4]]]]
22	[[3],[[4],[[5],[[2],[[4],[4]]]]]]	d[[[3],[[4],[4]]],[[3],[[4],[4]]]]

7. CONCLUSION

In this paper, an efficient package to implement the WHT was presented. Though the working set is divided into smaller sets by breakdown strategies, cache pollution causes performance degradation in the computation of a large size WHT. In our package, the data is reorganized and thus accessed efficiently. Our approach is a high level optimization method that exploits the characteristics of the memory hierarchy based on cache behavior analysis of the data access patterns. It is portable across platforms. Furthermore, it is applicable to various signal transforms including FFT and others. The work reported here is part of the SPIRAL [7] project. It is an ongoing collaborative project involving CMU, Drexel University, MathStar Inc., 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. A WHT package using our approach is available at the SPIRAL [7] web site.

8. ACKNOWLEDGMENT

We would like to thank Dr. Markus Püschel for his assistance with the WHT package in [3]. We also would like to thank Bhargava Gundala for his editorial assistance.

9. REFERENCES

[1] K. G. Beauchamp, *Applications of Walsh and Related Functions*, Academic Press, 1984.

[2] F. J. MacWilliams and N. J. Sloane, *The Theory of Error-Correcting Codes*, North-Holl. Publ. Co., 1992.

[3] J. Johnson, M. Püschel, "In Search of the Optimal Walsh-Hadamard Transform," *ICASSP*, 2000.

[4] M. Frigo and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *ICASSP '98*, 1998.

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

[6] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, vol. 10 of *Frontiers in Applied Mathmetics*, SIAM, 1992.

[7] SPIRAL Project, "http://www.ece.cmu.edu/∼spiral" .