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

Kang Chen
and
Jeremy R. Johnson

Technical Report DU-CS-02-02
Department of Computer Science
Drexel University
Philadelphia, PA 19104
May 2002

© Copyright 2002

Kang Chen. All Rights Reserved.

Dedication

This thesis is dedicated to Annie.

Acknowledgements

I sincerely thank my advisor, Dr. Jeremy Johnson, for all the guidance, help and support throughout my graduate study at Drexel. Last winter Dr. Johnson gave me the opportunity to join his team and to work on the SPIRAL project. This one and half year turns out to be one of the most wonderful and rewarding experiences I have ever had. I have learned so much from him and other team members, and the research greatly broadened my view and enhanced my knowledge. With his guidance, encouragement and challenge my research went very smooth and fruitful. I experienced the excitement of contributing to the project and having paper accepted. Last month Dr. Johnson brought me to IPDPS'02 conference at Ft. Lauderdale to present our research. He spent a lot of time to help me organize the presentation and prepare the speech, which was one of my weakness. During the meeting I was exposed to various latest research and discussions among international parallel computing community. I had chances to meet many talented people from different background and to learn from them. I am really grateful for this wonderful week.

Table of Contents

List of Tables vi

List of Figures vii

Abstract ix

1 Introduction 1

2 Algorithms for Computing the WHT 3

3 Sequential Performance of the WHT 10

3.1 Self-Optimization to Architecture 10

3.2 Normalized Runtime 13

3.3 Effect of DDL and IL 16

4 Parallel WHT Package using OpenMP 20

4.1 Parallel Split 20

4.2 Parallel DDL 22

4.3 Data Contention 23

5 Parallel Performance of the WHT 25

5.1 Platform-Adapted Parallel Algorithm 25

5.2 Parallel Scheduling of WHT Transform 26

5.3 Parallel Scheduling of DDL Pseudo-Transpose 35

6 Conclusion 39

Bibliography 40

A Source Code of Parallel Scheduling 42
A.1 Scheduling of WHT transform 42
A.1.1 Small Granularity 42
A.1.2 Large Granularity 43
A.1.3 Small Granularity with IL-enabled 44
A.1.4 Large Granularity with IL-enabled 45
A.2 Scheduling of DDL Pseudo-Transpose 48
A.2.1 Coarse-grained Scheduling 48
A.2.2 Fine-grained Scheduling 49
A.2.3 Fine-grained Scheduling with ID shift 52

List of Tables

2.1 Cost of search 7

3.1 Experiment environment 11

3.2 Best sequential WHT algorithms on PowerPC RS64 III 12

5.1 Compiler and flags information 26

5.2 Best WHT algorithms on PowerPC RS64 III with 10 threads 29

5.3 Best WHT algorithms on POWER3 II with 3 threads 30

5.4 Best WHT algorithms on UltraSPARC II with 3 threads 31

5.5 Best WHT algorithms on UltraSPARC with 3 threads 32

5.6 Performance of parallel WHT with DDL at size 2^{26} 36

List of Figures

2.1 Partition trees of iterative and recursive WHT algorithm 5

2.2 Data access patterns of iterative and recursive WHT algorithm 5

2.3 Pseudo-code of the generic fast WHT algorithm 5

2.4 Illustration of the in-place pseudo-transpose of DDL 8

2.5 Sample of IL-implemented straight-line code 9

3.1 Ratio of runtimes to the best WHT algorithms 11

3.2 Best WHT trees on different architectures at data size 2^{22} 13

3.3 Normalized runtime of PowerPC RS64 III 14

3.4 Normalized runtime of POWER3 II 15

3.5 Normalized runtime of UltraSPARC II 15

3.6 Normalized runtime of UltraSPARC 16

3.7 Usage of WHT subtrees 17

3.8 Effects of DDL and IL on WHT performance 18

3.9 Effects of DDL and IL on WHT partition trees 19

4.1 Pseudo-code of a simple parallel WHT algorithm 21

4.2 Three scheduling schemes for parallel pseudo-transpose 22

5.1 Parallel speedup on PowerPC RS64 III 27

5.2 Parallel speedup on POWER3 II 27

5.3 Parallel speedup on UltraSPARC II 28

5.4 Parallel speedup on UltraSPARC 28

5.5 Best parallel WHT trees on four SMPs at data size 2^{22} and 2^{23} 33

5.6 Effect of granularity in scheduling WHT transform 34

5.7 Effect of transform scheduling scheme on parallel WHT trees 34

5.8 Effect of granularity in scheduling DDL pseudo-transpose 35

5.9 Best partition tree of different DDL scheduling schemes 37

5.10 Effect of DDL block size on parallel performance 38

Abstract

A Prototypical Self-Optimizing Package for Parallel
Implementation of Fast Signal Transforms
Kang Chen
Jeremy R. Johnson

This thesis 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 effective granularity, load balance, and cache utilization. Experimental data are presented in the thesis showing the performance of the package on four different architectures. Results are also presented showing the optimizations required to obtain nearly linear speedup on these sample symmetric multiprocessors.

1 Introduction

The Walsh-Hadamard transform (WHT) is a prototypical digital signal processing (DSP) transform with applications to signal and image processing [1] and coding theory [11]. Fast algorithms for computing the WHT are similar to the fast Fourier transform (FFT) and its variants [8]. 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.

Johnson and Püschel [9] presented a package for automatically implementing and optimizing the Walsh-Hadamard transform. 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. However, in the FFTW package automatic search is conducted by manipulating codelets, and the search is limited to a smaller set of algorithms.

In this thesis we extend the WHT package so that it can generate parallel algorithms and search for optimal parallel implementations. Currently the generated programs are expressed using OpenMP [15] 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 PowerPC RS64 III machine [6]. The parallel package was also tested on three 4-processor SMPs with UltraSPARC, UltraSPARC II, and POWER3 II processors.

In Chapter 2 we review the WHT, the sequential WHT package, and the techniques of [4, 16] for reducing cache misses. The following chapter summarizes the sequential performance obtained on a single node of SMP systems that we used. In Chapter 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 Chapter 5. Conclusions and future work are summarized in Chapter 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.

$$\mathrm{WHT}_N = \bigotimes_{i=1}^{n} \mathrm{WHT}_2 = \overbrace{\mathrm{WHT}_2 \otimes \cdots \otimes \mathrm{WHT}_2}^{n}, \tag{2.1}$$

where

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

For example,

$$\begin{aligned} \mathrm{WHT}_4 &= \mathrm{WHT}_2 \otimes \mathrm{WHT}_2 \\ &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}. \end{aligned}$$

The WHT can be computed with N^2 arithmetic operations using a general matrix-vector product. Actual runtimes are heavily influenced by other factors than operation cost such as cache utilization. Fast algorithms for computing the WHT can be obtained by factorizations of the transform matrix WHT_N. The factorization is based on the multiplication property of the tensor product:

$$(\mathrm{A} \otimes \mathrm{B})(\mathrm{C} \otimes \mathrm{D}) = \mathrm{AC} \otimes \mathrm{BD} \tag{2.2}$$

Let I_m denote the $m \times m$ identity matrix. Using Equation 2.2 with $\mathrm{A} = \mathrm{WHT}_2$, $\mathrm{B} = \mathrm{I}_2$, $\mathrm{C} = \mathrm{I}_2$, and $\mathrm{D} = \mathrm{WHT}_2$, then

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

$$= (\mathrm{WHT}_2 \otimes \mathrm{I}_2)(\mathrm{I}_2 \otimes)\mathrm{WHT}_2)$$

$$= \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix}.$$

Applying this construction inductively the following factorization is obtained. Let $n = n_1 + \cdots + n_t$ be a partition of the exponent n, then

$$\mathrm{WHT}_N = \prod_{i=1}^{t} (\mathrm{I}_{2^{n_1+\cdots+n_{i-1}}} \otimes \mathrm{WHT}_{2^{n_i}} \otimes \mathrm{I}_{2^{n_{i+1}+\cdots+n_t}}). \tag{2.3}$$

The matrix factorization in Equation 2.3 can be interpreted as an algorithm for computing $y = \mathrm{WHT}_{2^{n_i}} \cdot x$. Let $y_0 = x$. The following process applies each WHT factor on data iteratively and computes y.

for $i = 1, \ldots, t$

$\quad y_i = \mathrm{WHT}_{2^{n_i}} \cdot y_{i-1}$

$y = y_t$

Each of the recursive computations $\mathrm{WHT}_{2^{n_i}}$ 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 $\mathrm{WHT}_{2^{n-1}}$. Figure 2.1 illustrates the algorithmic structure for iterative and recursive algorithms for WHT_{2^4}. For a data vector $x_0 - x_{15}$ (x-axis) Fig. 2.2 shows the data access pattern for iterative and recursive algorithms as time passes (y-axis). The general factorization combines various amounts of recursion and iteration. So its data access pattern depends the combination of the two components.

Let $\mathrm{N} = \mathrm{N}_1 \cdots \mathrm{N}_t$, where $\mathrm{N}_i = 2^{n_i}$, and let $x_{b,s}^M$ denote the vector $(x(b), x(b + s), \cdots, x(b + (M - 1)s))$. A sequential implementation of the factorization in Equa-

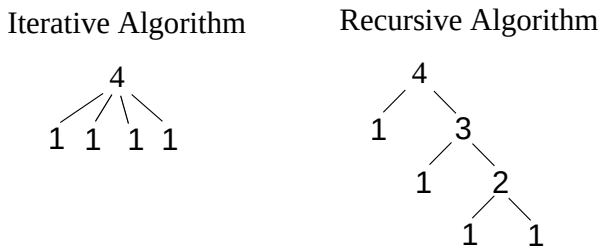

Figure 2.1: Partition trees of iterative and recursive WHT algorithm

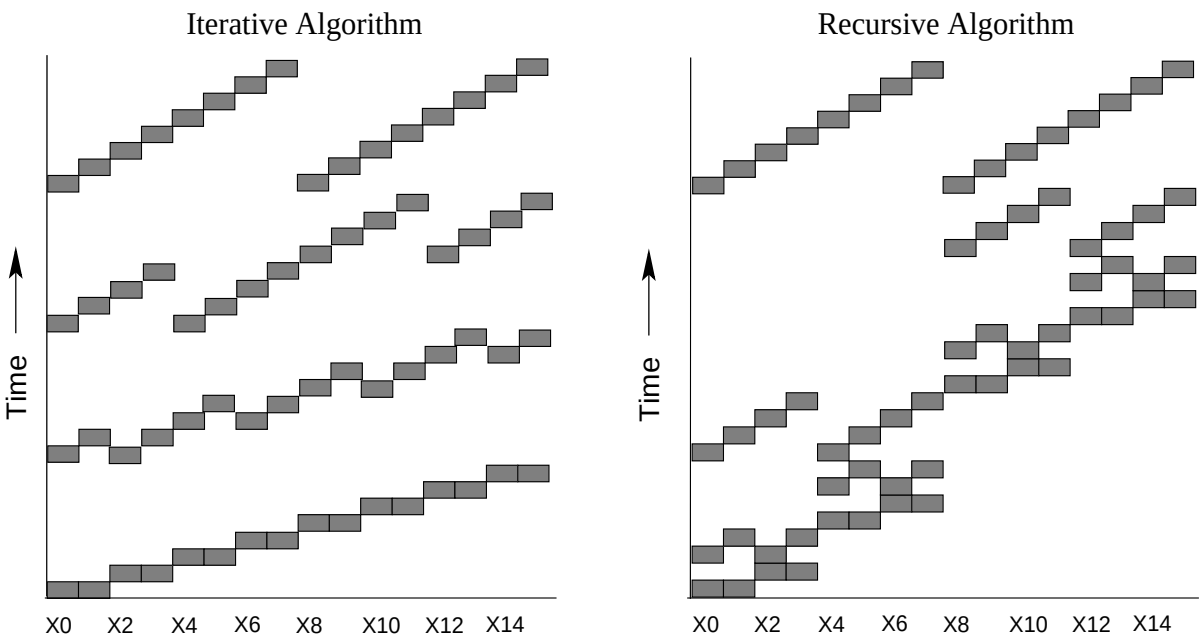

Figure 2.2: Data access patterns of iterative and recursive WHT algorithm

```
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} = WHT_{N_i} * x^{N_i}_{jN_iS+k,S};
    S = S * N_i;
```

Figure 2.3: Pseudo-code of the generic fast WHT algorithm

tion 2.3 is shown in Figure 2.3. This scheme assumes that the algorithm works in-place and is able to accept stride parameters. While all factorizations have exactly the same arithmetic ($N \log(N)$ operations [1]), different factorizations lead to algorithms with different data access patterns and consequently can have vastly different performance (Fig. 2.2).

In [9] 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 [14]. Particular algorithms, corresponding to instances of Equation 2.3, are represented by a tree data structure corresponding to the associated partition tree. Leaf nodes, called *small* nodes, correspond to straight-line code. It is used to reduce recursion and iteration overhead. Internal nodes, called *split* nodes, correspond to applications of Equation 2.3. WHT trees can be described using a grammar with the keyword **split** for internal nodes and **small** for leaf nodes. For example, the iterative WHT algorithm at size 2^4 can be represented by

split[**small**[1], [**small**[1], [**small**[1], [**small**[1]]

The recursive WHT algorithm can be represented by

split[**small**[1], **split**[**small**[1], **split**[**small**[1], **small**[1]]]]

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 only explores the algorithms generated from previously determined best algorithms. In the WHT package, DP builds partition trees using other smaller best trees as subtrees and searches the best combination. Dynamic programming, usually restricted to binary trees, is used since exhaustive search is too costly ($\Theta(4^n/n^{3/2})$). In contrast, the cost of DP is $\Theta(n^2)$ (Table 2.1).

---

[1] This can be easily proved by induction.

Table 2.1: Cost of search

Size	B_n(DP)	T_n(DP)	B_n(Exhaustive)	T_n(Exhaustive)
1	1	1	1	1
2	2	2	2	2
3	3	4	5	6
4	4	7	15	20
5	5	11	51	68
n	n^2	2^n	$4^n/n^{3/2}$	$5^n/n^{3/2}$

B_n: number of binary trees, T_n: number of full trees

Dynamic programming does not necessarily return the optimal tree since its assumption may be violated. A best algorithm is determined based on a consecutive block of data in certain size. However, the same algorithm may not be the optimal algorithm if the data is distributed. This is unfortunately the case happens in the WHT package. The factorization in Equation 2.3 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) [8]. In the parallel form, data are processed in consecutive block, but the vector form accesses the data at stride. Therefore, a best algorithm used in the vector form may not yield the same best performance as it is used in the parallel form. Nonetheless, experience shows that dynamic programming usually returns a tree with very good performance. So DP is still used in the package. Alternative search methods, that do not require the dynamic programming assumption, have been explored in [17].

The stride in vector form can introduce conflict misses when the stride is large [16]. Too many cache misses deteriorate the performance of the fast WHT algorithm. However, 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 $A \otimes I_m = L_n^{mn}(I_m \otimes A)L_m^{mn}$, where L_m^{mn} is a permutation called a stride permutation since it gathers the elements of a vector at stride m [8]. Since L_n^{mn} is the inverse of L_m^{mn} this transformation corresponds

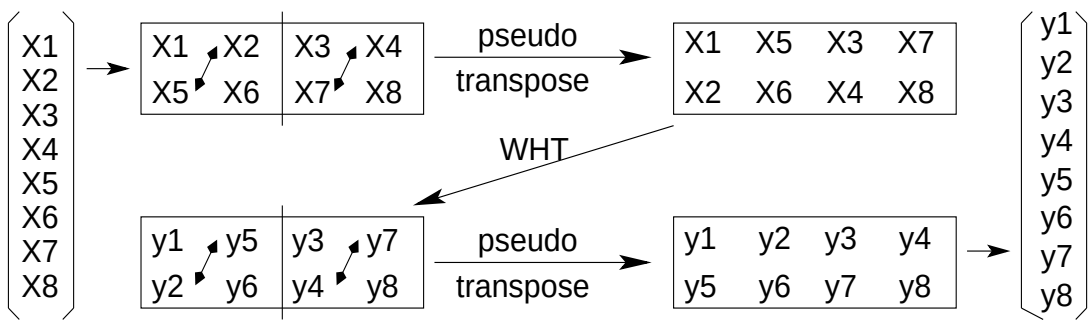

Figure 2.4: Illustration of the in-place pseudo-transpose of DDL

to relabeling the input and output data. Since the relabeling is performed at runtime, it has been called Dynamic Data Layout (DDL) [16]. 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 discussed in [16] 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 (Fig. 2.4).

Recently a new technique was implemented in the WHT package to reduce the cache misses caused by the vector form of the tensor product. The technique, called Loop Interleaving (IL), interleaves the unrolled operations of a certain number of

```
IL level 0          IL level 1                  IL level 2
                    x1 = x + 1;                 x1 = x + 1;
                                                x2 = x + 2;
                                                x3 = x + 3;

t0 = x[0] + x[1];   t0 = x[0] + x[1];           t0 = x[0] + x[1];
                    tu1_0 = x1[0] + x1[1];      tu1_0 = x1[0] + x1[1];
                                                tu2_0 = x2[0] + x2[1];
                                                tu3_0 = x3[0] + x3[1];

t1 = x[0] - x[1];   t1 = x[0] - x[1];           t1 = x[0] - x[1];
                    tu1_1 = x1[0] - x1[1];      tu1_1 = x1[0] - x1[1];
                                                tu2_1 = x2[0] - x2[1];
                                                tu3_1 = x3[0] - x3[1];

x[0] = t0;          x[0] = t0;                  x[0] = t0;
                    x1[0] = tu1_0;              x1[0] = tu1_0;
                                                x2[0] = tu2_0;
                                                x3[0] = tu3_0;

x[1] = t1;          x[1] = t1;                  x[1] = t1;
                    x1[1] = tu1_1;              x1[1] = tu1_1;
                                                x2[1] = tu2_1;
                                                x3[1] = tu3_1;
```

* code from **small**[**1**] with stride 1

Figure 2.5: Sample of IL-implemented straight-line code

WHT transforms into one transform. As Fig. 2.5 shows, **smallil1**[**1**] has two copies of **small**[**1**] with an offset of 1, and IL level 2 has four copies of the same transform interleaved into one transform. IL exploits the data prefetching behavior of the cache, and the performance improvement depends on the cache line size. Thus IL was implemented with different level of interleaving. The new straight-line code is called *smallilw* nodes, where $0 \leq w \leq 5$ is the interleaving factor. *smallil*0 node is the same as an ordinary *small* node. Since the parallel form of the tensor product accesses data in consecutively loops, interleaving is not necessary and consequently is not used in this case. Because a *splitddl* node changes vector form into a parallel form through in-place pseudo-transpose, IL-implemented *smallilw* nodes are not used as child node of *splitddl* node. In the same way as DDL, IL was provided as an option to dynamical programming search, which decides whether to use it and the interleaving level.

3 Sequential Performance of the WHT

To prepare for our parallel study we investigated the performance of the sequential package on a single node of SMP system. We used four SMP systems as listed in Table 3.1. More detailed information about each architecture is available at [6], [7], [12], and [13]. We analyzed the best algorithms, the partition trees, and the normalized runtimes of different architectures. We also discussed the effect of DDL and IL on the algorithms and performance.

3.1 Self-Optimization to Architecture

We compared the relative running times for three WHT algorithms: recursive, iterative, and the best algorithms found by DP (Fig. 3.1). Both the recursive and iterative algorithm use WHT_4 instead of WHT_2 as leaf node. All times are presented as ratios compared to the best tree without DDL and IL. Two 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 3.1, shows that the trees found by DP are at least two times faster than either the recursive or iterative algorithms. The best algorithms are the optimal combination of iteration and recursion changing at different data size (See Table 3.2). The combination is the tradeoff between the control overhead and data locality.

The best algorithms found by dynamic programming are highly architecture dependent. Figure 3.2 shows the best algorithm found by DP at data size 2^{22} for the

Table 3.1: Experiment environment

Machine	IBM RS/6000 S80	IBM RS/6000 44P	Sun Enterprise 450	Sun Enterprise 3000
CPU	12×450MHz	4×375MHz	4×400MHz	4×168MHz
CPU Type	PowerPC RS64 III	POWER3 II	UltraSPARC II v9	UltraSPARC v8plus
L1 Data/Instr. Cache	128KB/128KB	64KB/32KB	16KB/16KB	16KB/16KB
L1 Cache Line	128 Byte	128 Byte	2×16 Byte	2×16 Byte
L1 Associativity	2-way set	128-way set	direct-mapped	direct-mapped
L2 Cache	8MB	4MB	-	-
Memory	8GB	2GB	1GB	256MB
Compiler/Version	cc/IBM Visual Age 5		cc/Sun WorkShop 6	

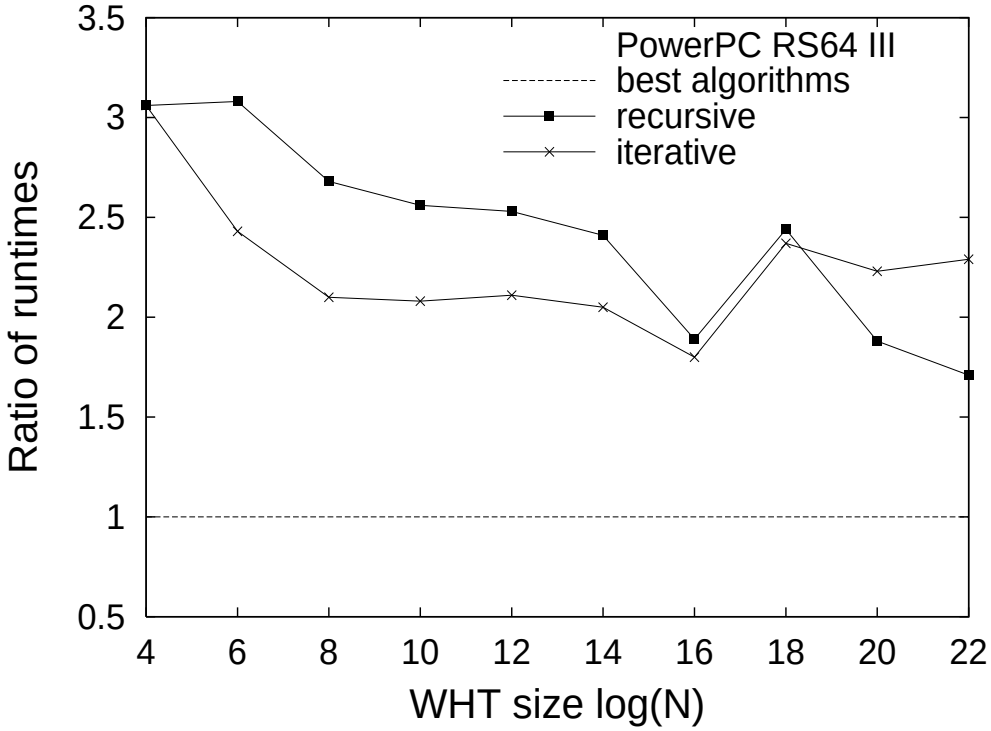

Figure 3.1: Ratio of runtimes to the best WHT algorithms

Table 3.2: Best sequential WHT algorithms on PowerPC RS64 III

Size	Parallel WHT algorithm
1	small[1]
2	small[2]
3	small[3]
4	small[4]
5	small[5]
6	small[6]
7	small[7]
8	small[8]
9	split[small[3],small[6]]
10	split[small[4],small[6]]
11	split[small[5],small[6]]
12	split[small[6],small[6]]
13	split[small[6],small[7]]
14	split[small[4],split[small[4],small[6]]]
15	split[small[1],split[small[4],split[small[4],small[6]]]]
16	split[small[1],split[small[1],split[small[4],split[small[4],small[6]]]]]
17	split[small[6],split[small[5],small[6]]]
18	split[small[6],split[small[6],small[6]]]
19	split[small[1],split[small[6],split[small[6],small[6]]]]
20	split[small[1],split[small[1],split[small[6],split[small[6],small[6]]]]]
21	split[small[8],split[small[6],small[7]]]
22	split[small[2],split[small[1],split[small[1],split[small[6],split[small[6],small[6]]]]]]
23	split[small[4],split[small[1],split[small[6],split[small[6],small[6]]]]]
24	split[small[5],split[small[1],split[small[6],split[small[6],small[6]]]]]

* DDL and IL were not enabled.

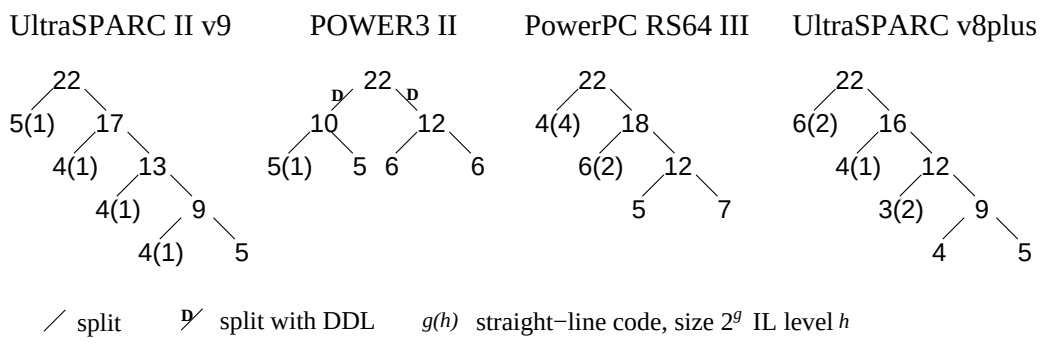

Figure 3.2: Best WHT trees on different architectures at data size 2^{22}

four architectures. It is apparent that the best algorithms differ significantly. Our experiment indicates that, if the UltraSPARC II system uses the best algorithm for the POWER3 II, its performance will decrease by 75%, and similarly, performance of the POWER3 II system will decrease by 50% if it uses the best algorithm for the UltraSPARC II. There are many system factors affecting the WHT performance, such as memory hierarchy, cache size, cache structure (associativity and cache line size), write policy, and cache miss penalty. While it is possible to qualitatively relate algorithm performance with the underlying hardware [5], it is difficult to obtain an accurate correlation that can be used to *a priori* predict the best algorithm. Therefore, the package uses automatic optimization with empirical to determine the best algorithm on a given architecture.

3.2 Normalized Runtime

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 3.3 shows the normalized runtime of the WHT program. The three plateaus in the Figure 3.3 are related to the L1 and L2 caches (the L1 data cache is 128 KB or 2^{14} doubles, and the L2 combined cache

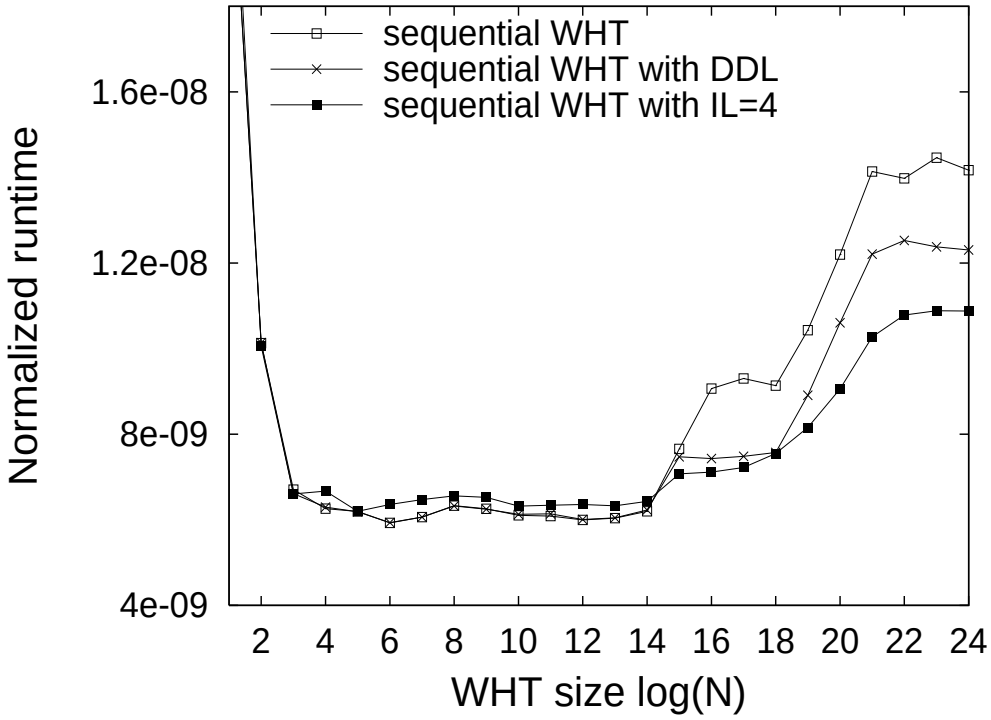

Figure 3.3: Normalized runtime of PowerPC RS64 III

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. For the POWER3 II system in Figure 3.4 the L1 cache design has a large set associativity that reduces conflict misses. So it doesn't show any significant miss penalty as long as the data size is within the L1 cache. However, the penalty soars when the data size increases close to the L2 cache size.

During the search for optimal WHT algorithms, DP tests different combinations of pre-determined best algorithms and chooses the fastest one. The figures of normalized runtime illustrate that WHT partition trees at certain sizes, such as at size 2^5, 2^{10}, and 2^{15} on UltraSPARC II, are much faster than others. Therefore, it is reasonable to conjecture that the potentially fastest WHT tree is more likely to be constructed by combining subtrees at the favorable sizes. This notion is supported by experimental data at UltraSPARC II. As it is shown in Fig. 3.7 some subtrees are used statistically

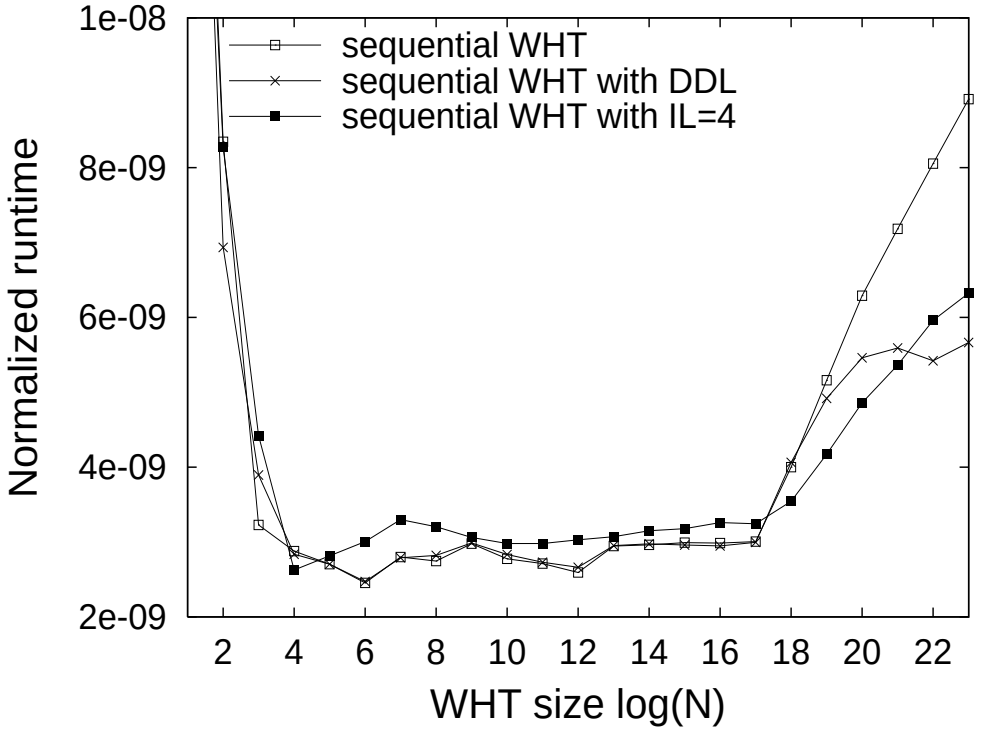

Figure 3.4: Normalized runtime of POWER3 II

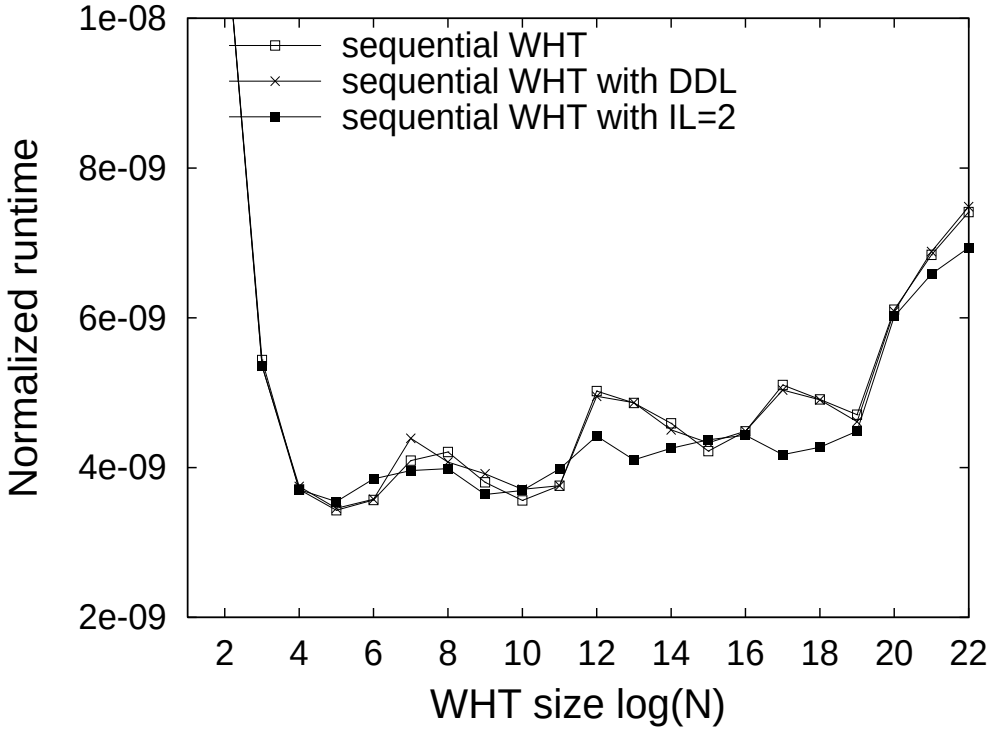

Figure 3.5: Normalized runtime of UltraSPARC II

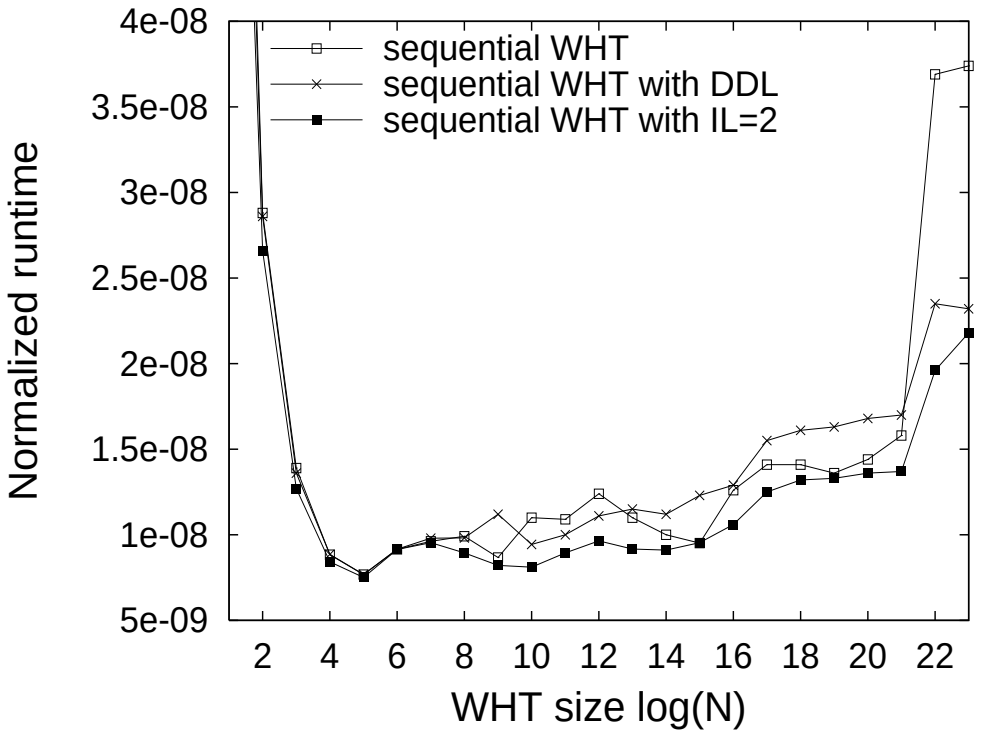

Figure 3.6: Normalized runtime of UltraSPARC

more often than others because of their low computing cost, which matches well with the pattern in Fig. 3.5. Figure 3.3 for the PowerPC RS64 III indicates that an uneven partition of a large WHT node may result in a small node in the first plateau and another large 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.

3.3 Effect of DDL and IL

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

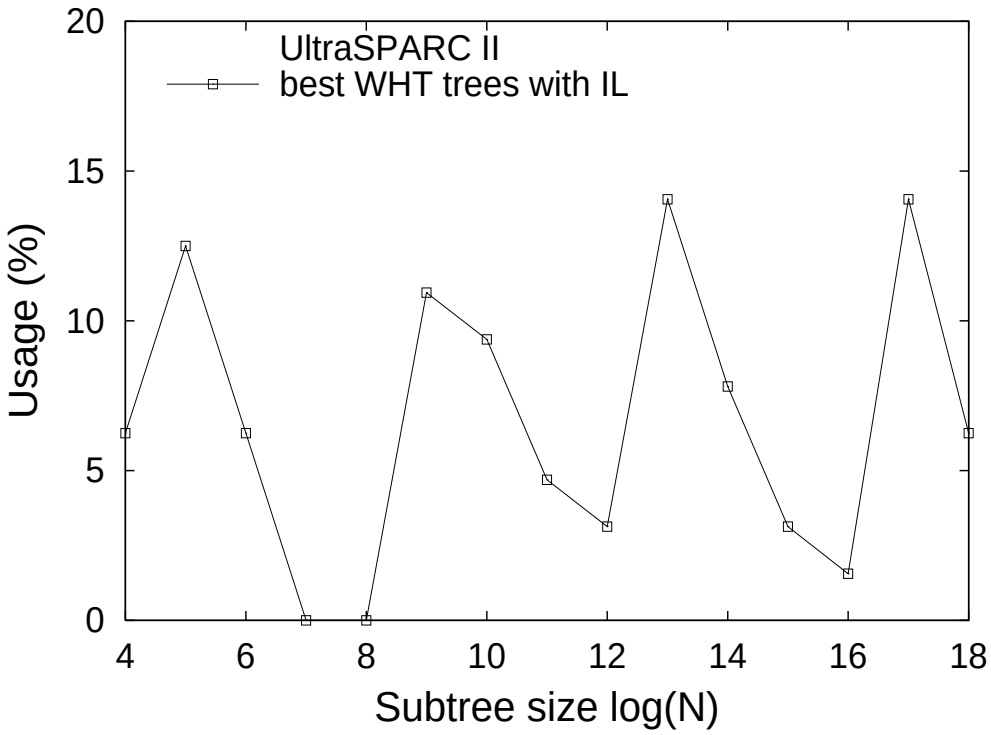

Figure 3.7: Usage of WHT subtrees

before adjacent elements are accessed. This type of cache miss is called conflict miss. Some cache design uses large set associativity to reduce conflict cache misses. However, increasing the associativity decreases the number of available cache lines, and consequently may increase capacity misses.

DDL reduces stride access by dynamically rearranging data. Figure 3.3 indicates that on PowerPC RS64 III when the stride is larger than the L1 cache, the benefit from the improved data access pattern outweighs the additional computations of pseudo-transpose. When DDL is enabled in the sequential WHT algorithms, the improved data access pattern reduces the L1 and L2 cache misses caused by large stride (Fig. 3.3). Figure 3.8 shows DDL improves WHT performance by about 15% on average. The cache on the POWER3 II has a very large associativity so the stride access pattern of the vector tensor form does not cost more than the parallel tensor form. Therefore, DDL cannot improve the performance until the data size is larger

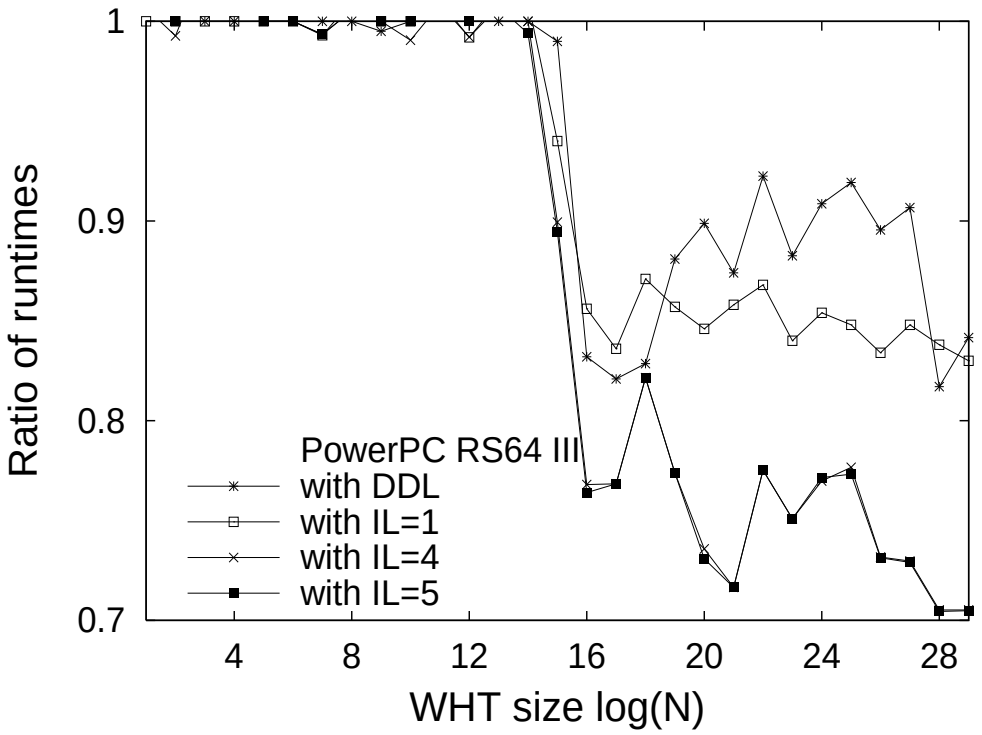

Figure 3.8: Effects of DDL and IL on WHT performance

than the L2 cache. Figures 3.5 and 3.6 show that DDL fails to improve the WHT performance on the UltraSPARC architecture family. Figure 3.9 displays the best WHT partition tree with and without DDL for size 2^{22} on the PowerPC RS64 III. The tree with DDL is more balanced than the tree without DDL. This improves performance since nodes of size 2^{10} and 2^{12} used by DDL fit in the L1 cache, whereas the node of size 2^{20} used by the non-DDL tree exceeds the L1 cache.

As discussed in Chapter 2, IL does not eliminate stride access, instead it interleaves the unrolled operations of adjacent transforms into one transform. Thus it improves cache line reuse. Unlike DDL, IL does not incur any additional computation, and it even removes part of the control overhead. Figure 3.8 indicates that the data access pattern improved by IL has even less L1 and L2 cache misses compared to DDL. Apparently, the improvement of IL is largely based on cache line size. Figure 3.8 shows that as the IL level increases, the performance of WHT algorithm improves.

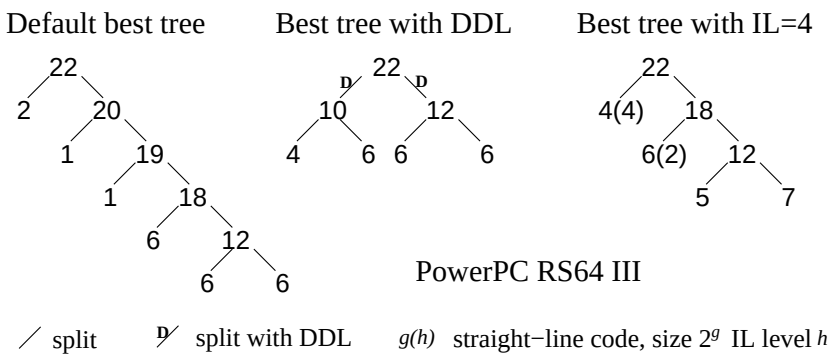

Figure 3.9: Effects of DDL and IL on WHT partition trees

At level 4, IL matches the cache line; it thus reaches the maximal improvement. Another observation from the figure is that there is no need for DDL and IL to alter the data access pattern when the data size is less than the L1 cache size. On UltraSPARC systems each cache line is divided into two sub-blocks with a size of 16 bytes or 2 doubles. Therefore, the benefit of IL is limited to two levels, and it does not lead to a significant improvement on these systems, however, IL still reduces L1 cache misses for WHT trees at certain sizes as shown in Figures 3.5 and 3.6.

These experiments indicate that IL improves the data access pattern for WHT algorithm without additional operations required by DDL. Hence it is more efficient than DDL on most systems. However, IL may impose an implicit constraint on WHT partition trees. For example, in the IL-enabled WHT tree (Fig. 3.9) the size of the straight-line code **smallil4**[4] is as large as that of **small**[8], which has reached the maximal size of straight-line code. Therefore, further increasing the straight-line code size is impossible. This results in a very uneven partition with a node of size 2^{18} as subtree that incurs large L1 cache misses. This explains why DDL becomes faster than IL at larger data size on some system such as POWER3 II (Fig. 3.4).

4 Parallel WHT Package using OpenMP

We extended the WHT package to support parallel computation on shared-memory multiprocessors (SMPs). Parallel code was obtained using OpenMP [15], a parallel programming model for SMPs. 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 package obtains parallelism through the use of *parallel split*, *parallel split* with IL, and *parallel splitddl* nodes. When the data size is too small, the cost of initializing and synchronizing threads may exceed the speedup gained by parallelism. Therefore, parallelization is limited to the root level to ensure the largest data size and to reduce parallel overhead. Self-optimization to different platforms is achieved by searching for the best parallel nodes and by searching for the best partition tree selecting the optimized sequential trees as subtrees. Additional optimization is obtained by tuning the implementation of these nodes and optimizing granularity, load balance, and cache utilization. The package is available for download at SPIRAL website http://www.ece.cmu.edu/~spiral. Appendix A also lists the souce code of different scheduling schemes we designed in this thesis.

4.1 Parallel Split

OpenMP provides work-sharing directives, which parallelize sequential iterations into parallel iterations automatically and seamlessly. This is the easiest way to convert a sequential program into a parallel program. However, the resulting parallel program may not yield premium parallel performance. In this research, we designed the

```
#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} = WHT_{N_i} * x^{N_i}_{jN_iS+k,S};
    S = S * N_i;
    #parallel barrier
#end parallel region
```

Figure 4.1: Pseudo-code of a simple parallel WHT algorithm

scheduling scheme according to the data access pattern of WHT. 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.

Figure 4.1 lists the pseudo-code of a simple parallel WHT algorithm. The inner loop allocates the work (recursive WHT applications) for each stage in the factorization in Equation 2.3. 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 due to the repeated initialization of threads.

In Fig. 4.1 tasks involving the computation of each single WHT_{N_i} are scheduled in a round robin fashion to threads. The task size is therefore the size of the transform. This is the smallest granularity for static scheduling. Alternatively, a collection of WHT_{N_i} computation can be grouped together to form larger tasks. The largest granularity for scheduling is to divide the whole computation evenly into as many chunks

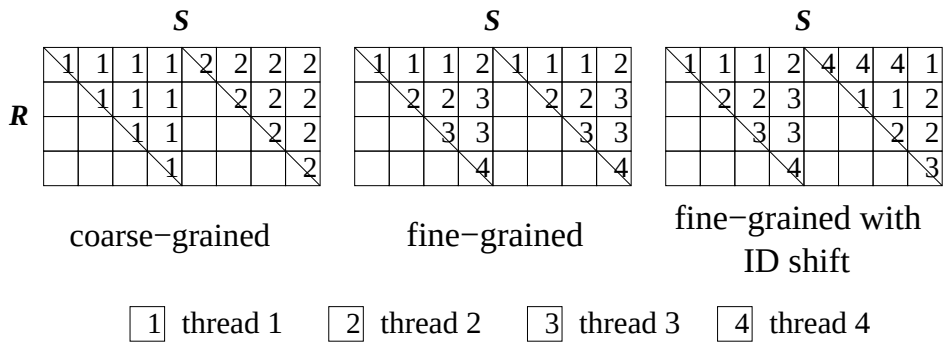

Figure 4.2: Three scheduling schemes for parallel pseudo-transpose

as thread number. This scheduling scheme requires additional control overhead but it may offer better data locality and less data sharing among threads. Both the parallel tensor form and the vector tensor form can be parallelized using the same scheduling. With some adjustment, this scheme is also applicable to the IL-enabled *split* node.

4.2 Parallel DDL

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. While the WHT factors can use the same parallelism mentioned above, the pseudo-transpose of DDL must be parallelized differently. Since the S/R square matrices in size of $R \times R$ are independent in the pseudo-transpose, it can be performed in parallel. This coarse-grained parallel DDL is the most natural scheduling and has minimal control overhead. It 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 the sequential DDL, blocking technique is used in order to minimize the cache

misses while performing the pseudo-transpose [16]. 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. Cache 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. A fine-grained parallel DDL can be obtained by parallelizing this blocking technique. Distributing tasks at the block level 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 with the blocks at the same position in the next matrix (Fig. 4.2). The problem of this scheduling is the last thread has fewer tasks than the others do in some cases. Since the scheduling is based on thread ID, an easy solution is to use a virtual thread ID and rotate the ID as the threads move from one $R \times R$ matrix to the next. This method can completely balance the load (Fig. 4.2).

4.3 Data Contention

The tradeoff between parallel versions of *split*, *split* with IL, 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 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. In the parallel case there is a new cache access cost no matter whether the data size is larger than the cache or not. This is due to data contention caused by interactions amongst the caches of the separate processors. In the shared-memory system, the cache of each processor

tries to store a valid copy of recent accessed data. Since multiple copies of the same data may exist across processors, To avoid race condition many policies are designed to maintain the coherence. The IBM S80 uses a bus snooping policy to enforce cache coherence [6]. 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. This is the case even though the data accessed is different (only the cache lines are shared).

Data contention can be avoided by using a parallel version of a *splitddl* node, where data is accessed is at stride one. However, there is additional overhead due to the parallel pseudo-transpose. The parallel version of IL also solves the problem. If the IL level exactly matches the cache line size, a processor can have an exclusive copy of the entire cache line without coherence problem, therefore, it is free of data contention. However, locking the IL level at some value limits the choice of straight-line code for DP search, and results in very uneven partitions of the WHT trees. If the data size is smaller than the L1 cache, the *parallel split* with scheduling granularity larger than the cache line size can be used. It provides another way to avoid data contention.

5 Parallel Performance of the WHT

The parallel performance is evaluated with speedup, which is computed as the ratio of the best sequential runtime to the parallel runtime. In Chapter 3 runtimes were measured using CPU timer, and it is independent of system load during the experiment. However, the parallel runtime has to be measured with a wall clock timer, which depends on the system load. Therefore, to make the two times comparable all the sequential runtimes in this chapter were measured with wall clock timer. The compiler and flags used in our experiment on the four SMP systems are listed in Table 5.1.

5.1 Platform-Adapted Parallel Algorithm

Dynamic programming was used on the four SMP systems to find the best parallel algorithms, built using *parallel split* with and without IL (p_split), and *parallel splitddl* (p_splitddl) nodes, for computing the WHT using different number of threads. The thread number can be varied during the search. The best parallel trees and runtimes were recorded.

Figures 5.1, 5.2, 5.3, and 5.4 show the speedup of the parallel WHT package on the four SMP systems with varying thread numbers. As the graphs indicate, the parallel split nodes start to replace the sequential split node as the root node when the data size is larger than 2^{11}. 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 or thread number increases. The package has close to linear speedup on the PowerPC RS64 (up to 10 threads), the POWER3 (up to 3 threads), and the UltraSPARC (up to 3 threads). The best efficiency obtained (speedup/number of threads) was approximately 90%. Our experiment shows

Table 5.1: Compiler and flags information

Architecture	Compiler	Flags	OpenMP flags
POWER3 II	cc	-O3 -qarch=pwr3 -qtune=pwr3 -qansialias -w -q64	-qsmp=omp
PowerPC RS64 III	cc	-O3 -qarch=rs64c -qtune=rs64c -qansialias -w -q64	-qsmp=omp
UltrsSPARC	cc	-fast -xO5 -dalign -xarch=v8plus	-xopenmp
UltraSPARC II	cc	-fast -xO5 -dalign -xarch=v9	-xopenmp

that, however, the parallel performance deteriorated fast on the UltraSPARC II SMP machine. It is possibly due to the machine as an administrative server is consistently with high system load.

Fig. 5.5 illustrates some of the parallel WHT algorithms found by dynamic programming on the four systems. Full lists of the best parallel algorithms are given in Tables 5.2, 5.3, 5.4, and 5.5. As the partition trees show, DP found the best parallel root node and the best breakdown strategy for each data size. The best parallel algorithms are SMP dependent.

5.2 Parallel Scheduling of WHT Transform

Both the vector tensor form and the parallel tensor form of the WHT transform can be readily parallelized with static scheduling. To investigate the effect of granularity on static scheduling, two scheduling schemes were implemented separately and tested individually. One scheme used tasks of the smallest granularity, i.e. the size of the transform, and the other scheme used tasks of the largest possible granularity. Figure 5.6 shows that the large granularity scheduling not only improves the parallel speedup, it also reduces parallel overhead so that smaller data size can benefit from parallel speedup with multiple threads. This is because with large granularity the scheduling helps to avoid data contention and enhance data locality. This is particularly evident for *parallel split* without IL and DDL. So most parallel trees for data

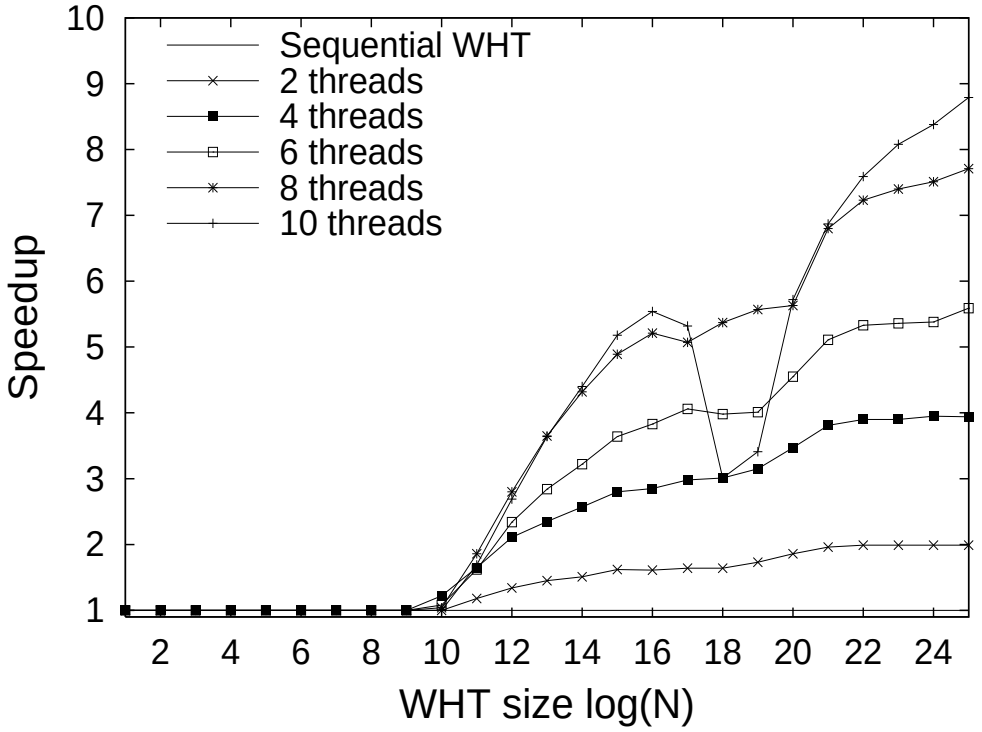

Figure 5.1: Parallel speedup on PowerPC RS64 III

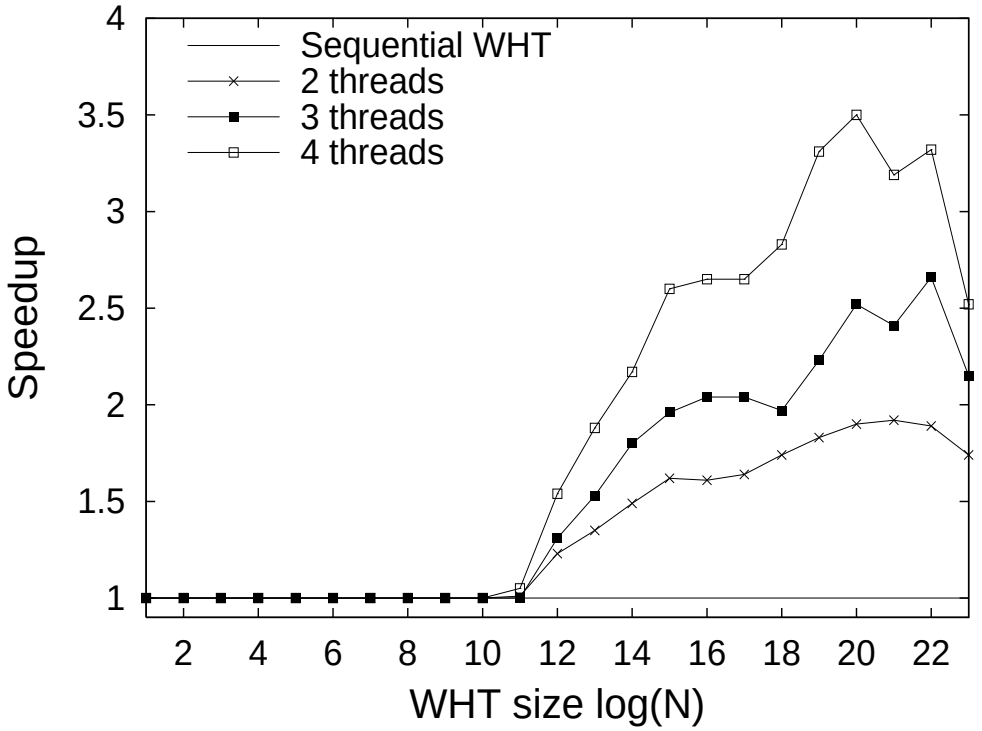

Figure 5.2: Parallel speedup on POWER3 II

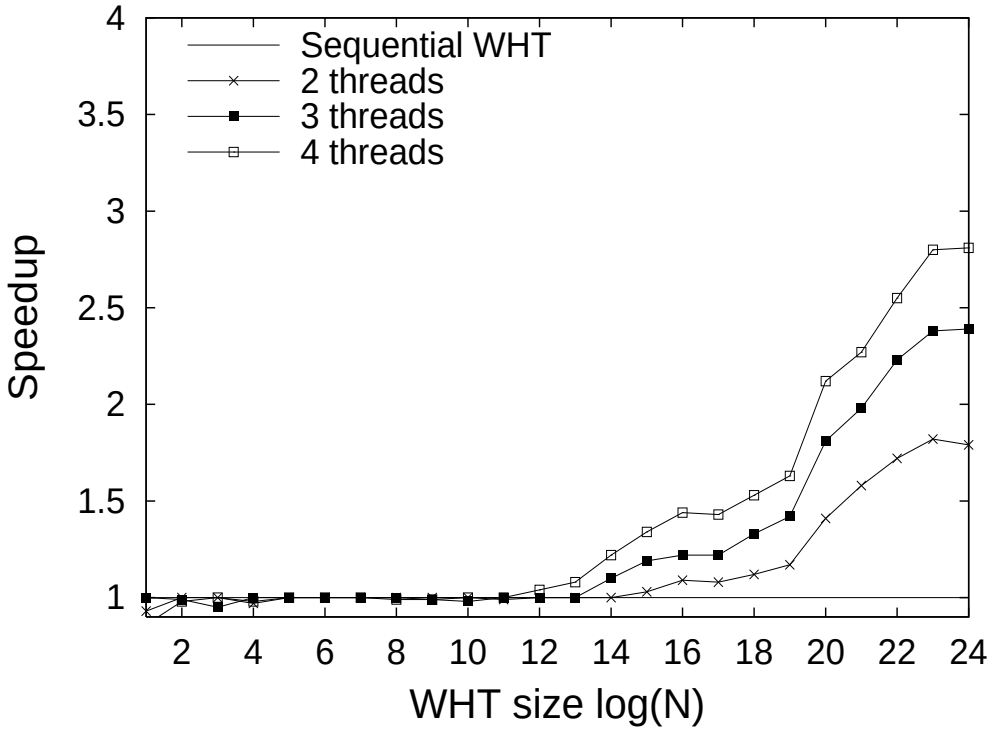

Figure 5.3: Parallel speedup on UltraSPARC II

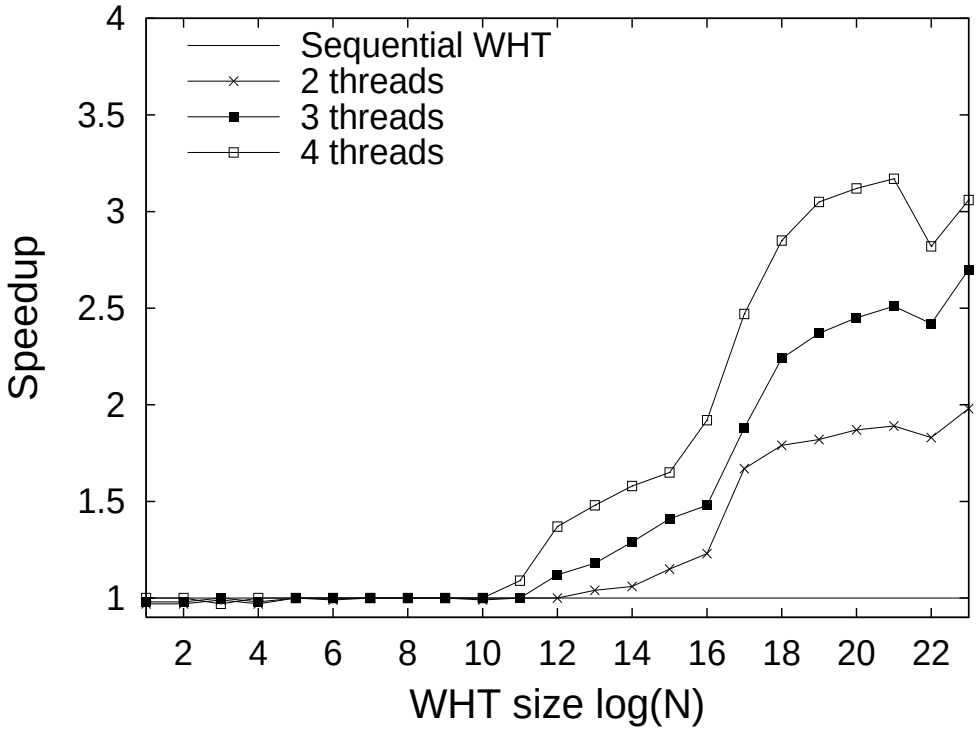

Figure 5.4: Parallel speedup on UltraSPARC

Table 5.2: Best WHT algorithms on PowerPC RS64 III with 10 threads

Size	Parallel WHT algorithm
1	small[1]
2	small[2]
3	small[3]
4	small[4]
5	small[5]
6	small[6]
7	small[7]
8	split[small[3],small[5]]
9	split[small[3],small[6]]
10	split[small[5],small[5]]
11	p_split[smallil3[3],split[small[3],small[5]]]
12	p_split[small[3],split[small[4],small[5]]]
13	p_split[small[3],split[small[5],small[5]]]
14	p_split[small[3],split[small[5],small[6]]]
15	p_split[smallil3[4],split[small[5],small[6]]]
16	p_split[smallil3[5],split[small[5],small[6]]]
17	p_split[smallil4[4],split[small[3],split[small[5],small[5]]]]
18	p_split[small[7],split[small[5],small[6]]]
19	p_split[smallil1[6],split[small[3],split[small[5],small[5]]]]
20	p_split[smallil2[3],split[smallil3[5],split[small[5],small[7]]]]
21	p_split[smallil3[5],split[smallil3[5],split[small[5],small[6]]]]
22	p_split[smallil2[6],split[smallil3[5],split[small[5],small[6]]]]
23	p_split[smallil2[6],split[smallil3[5],split[small[5],small[7]]]]
24	p_split[smallil2[6],split[smallil2[6],split[small[5],small[7]]]]
25	p_split[smallil2[6],split[smallil2[1],split[smallil2[6],split[small[5],small[7]]]]]

Table 5.3: Best WHT algorithms on POWER3 II with 3 threads

Size	Parallel WHT algorithm
1	small[1]
2	small[2]
3	small[3]
4	small[4]
5	small[5]
6	small[6]
7	small[7]
8	small[8]
9	split[smallil1[5],small[4]]
10	split[smallil1[5],small[5]]
11	split[smallil1[5],small[6]]
12	p_split[small[3],split[smallil1[5],small[4]]]
13	p_split[small[4],split[smallil1[5],small[4]]]
14	p_split[smallil1[5],split[smallil1[5],small[4]]]
15	p_split[smallil1[5],split[smallil1[5],small[5]]]
16	p_split[smallil1[5],split[smallil1[5],small[6]]]
17	p_split[smallil3[4],split[small[4],split[smallil1[5],small[4]]]]
18	p_split[smallil3[5],split[small[4],split[smallil1[5],small[4]]]]
19	p_split[smallil3[5],split[smallil2[4],split[smallil1[5],small[5]]]]
20	p_split[smallil3[5],split[smallil1[5],split[smallil1[5],small[5]]]]
21	p_split[smallil4[4],split[smallil2[4],split[small[4],split[smallil1[5],small[4]]]]]
22	p_split[smallil3[5],split[smallil2[4],split[small[4],split[smallil1[5],small[4]]]]]
23	p_splitddl[split[smallil1[5],small[6]],split[small[6],small[6]]]

Table 5.4: Best WHT algorithms on UltraSPARC II with 3 threads

Size	Parallel WHT algorithm
1	small[1]
2	small[2]
3	small[3]
4	small[4]
5	small[5]
6	small[6]
7	small[7]
8	split[smallil1[4],small[4]]
9	split[smallil1[4],small[5]]
10	split[small[5],small[5]]
11	split[small[5],small[6]]
12	split[smallil2[3],split[smallil1[4],small[5]]]
13	split[smallil1[4],split[smallil1[4],small[5]]]
14	p_split[smallil1[5],split[smallil1[4],small[5]]]
15	p_split[smallil1[5],split[small[5],small[5]]]
16	p_split[smallil1[5],split[small[5],small[6]]]
17	p_split[smallil1[5],split[smallil2[3],split[smallil1[4],small[5]]]]
18	p_split[smallil1[5],split[smallil1[4],split[smallil1[4],small[5]]]]
19	p_split[smallil1[5],split[smallil1[4],split[small[5],small[5]]]]
20	p_split[small[5],split[smallil1[5],split[small[5],small[5]]]]
21	p_split[smallil1[5],split[smallil2[3],split[smallil1[4],split[smallil1[4],small[5]]]]]
22	p_split[smallil2[5],split[smallil1[4],split[smallil1[4],split[smallil1[4],small[5]]]]]
23	p_split[smallil1[5],split[smallil1[4],split[smallil1[4],split[small[5],small[5]]]]]
24	p_split[smallil2[5],split[smallil1[4],split[smallil1[5],split[small[5],small[5]]]]]

Table 5.5: Best WHT algorithms on UltraSPARC with 3 threads

Size	Parallel WHT algorithm
1	small[1]
2	small[2]
3	small[3]
4	small[4]
5	small[5]
6	small[6]
7	small[7]
8	split[smallil2[3],small[5]]
9	split[smallil1[4],small[5]]
10	split[small[5],small[5]]
11	split[small[5],small[6]]
12	p_split[smallil2[3],split[smallil1[4],small[5]]]
13	p_split[smallil1[4],split[smallil1[4],small[5]]]
14	p_split[small[5],split[smallil1[4],small[5]]]
15	p_split[small[5],split[small[5],small[5]]]
16	p_split[smallil1[5],split[small[5],small[6]]]
17	p_split[smallil1[4],split[smallil1[4],split[smallil1[4],small[5]]]]
18	p_split[smallil1[4],split[smallil1[4],split[small[5],small[5]]]]
19	p_split[smallil1[4],split[small[5],split[small[5],small[5]]]]
20	p_split[smallil1[5],split[small[5],split[small[5],small[5]]]]
21	p_split[smallil1[5],split[smallil1[4],split[smallil2[3],split[smallil1[4],small[5]]]]]
22	p_split[smallil2[6],split[smallil1[4],split[smallil2[3],split[smallil1[4],small[5]]]]]
23	p_splitddl[small[7],split[smallil2[3],split[smallil1[4],split[smallil1[4],small[5]]]]]

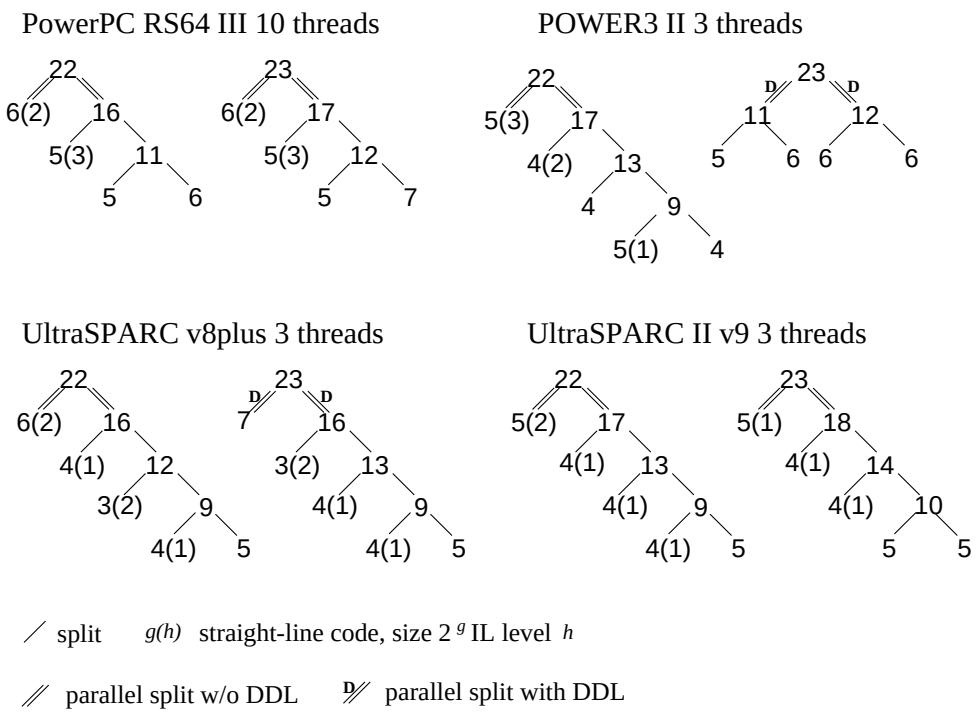

Figure 5.5: Best parallel WHT trees on four SMPs at data size 2^{22} and 2^{23}

size less then L1 cache uses *parallel split*.

Figure 5.7 shows the best parallel WHT algorithms when different scheduling schemes were used. As we can see, the tree with IL-enabled and small granularity scheduling of WHT transform has to use IL level 4, *smallil*4, to avoid data contention on the cache line. At this IL level, the maximal straight-line code size is 2^4. As a result, the node size of the right subtree is 2^{18}, which has a large L1 cache miss penalty (see Fig. 3.3). In contrast, with large granularity IL can use any level without incurring data contention. So, the best tree found by DP uses *smallil*2(6) as the left subtree, which is the best compromise between the IL level and the node size.

Large granularity scheduling also improves the performance of *parallel splitddl* even though it only has the parallel tensor form. Our experiment results indicate that the improvement is consistent on other SMP systems and with different number of threads.

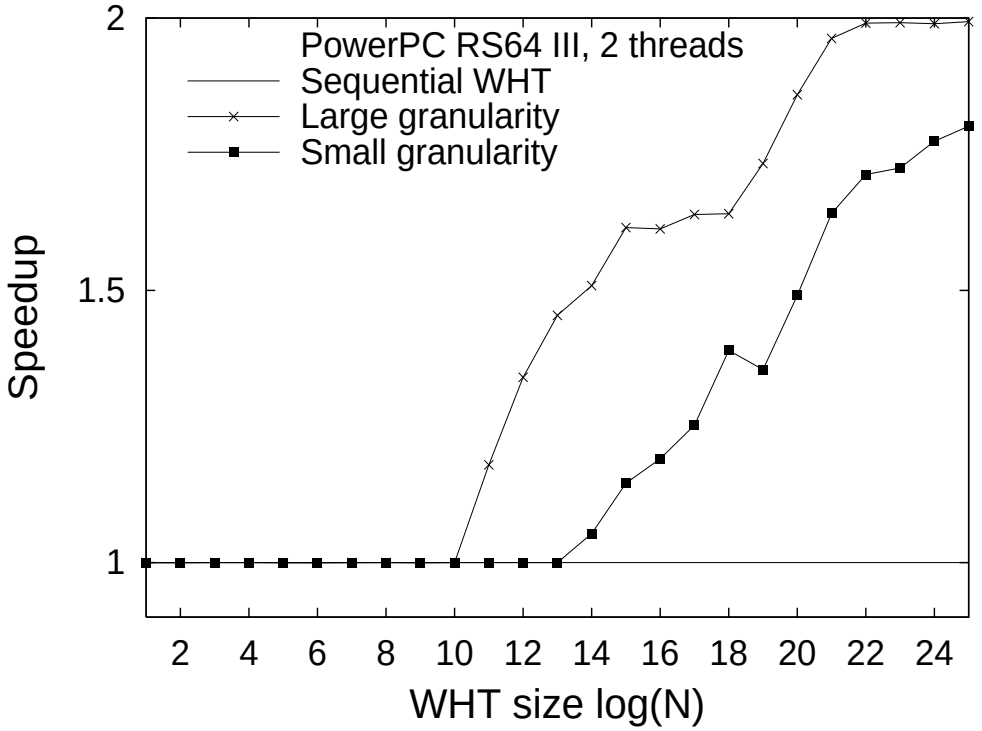

Figure 5.6: Effect of granularity in scheduling WHT transform

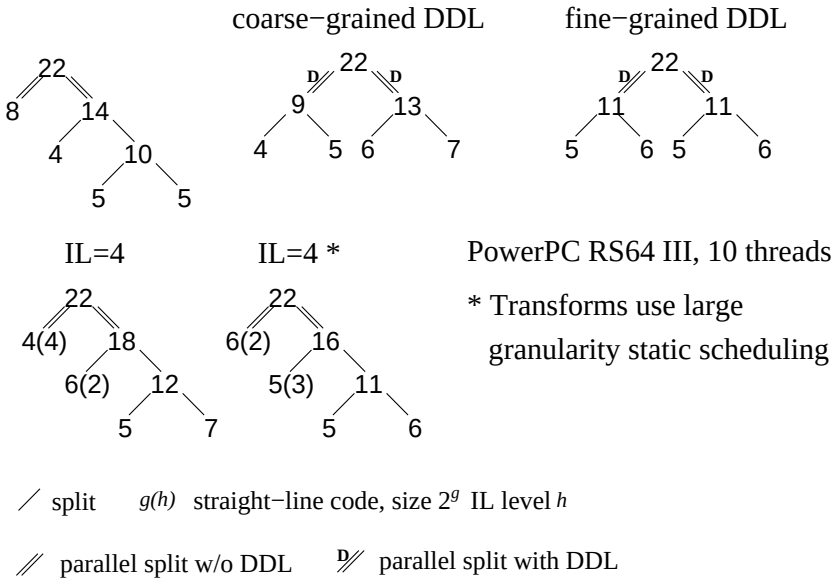

Figure 5.7: Effect of transform scheduling scheme on parallel WHT trees

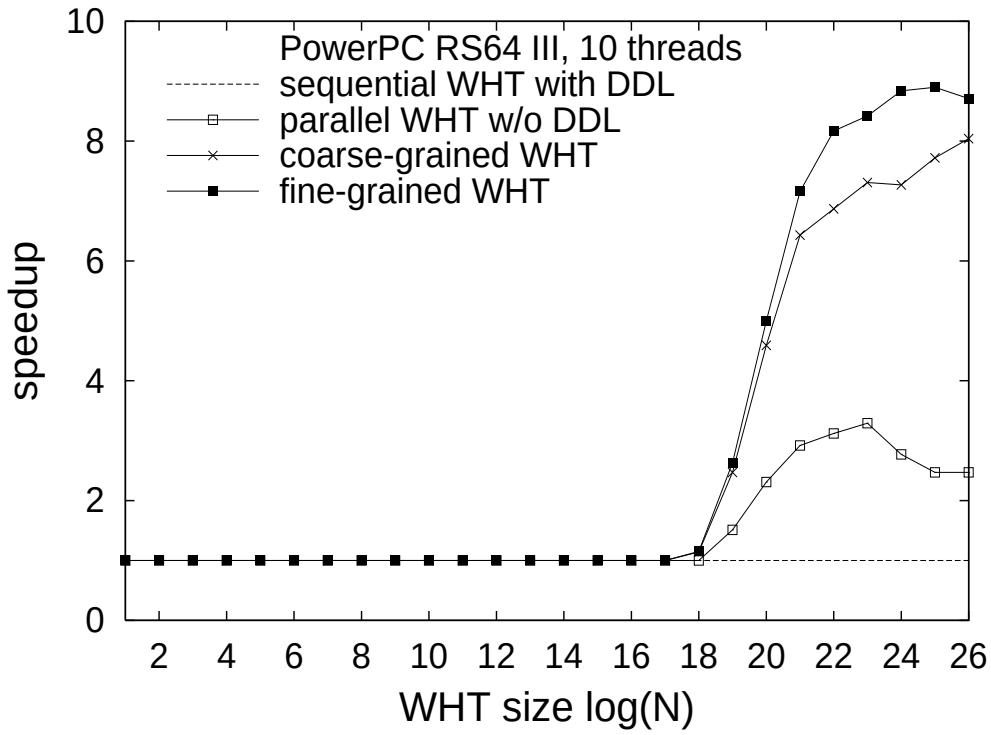

Figure 5.8: Effect of granularity in scheduling DDL pseudo-transpose

5.3 Parallel Scheduling of DDL Pseudo-Transpose

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. In this experiment, the transforms used small granularity static scheduling.

Figure 5.8 shows the speedup of the parallel WHTs for sizes from 2 to 2^{26}. Observe that the speedup obtained by the parallel WHT without DDL and IL is less than three. That means DDL greatly improves the performance of the parallel WHT on this architecture. The improvement, which is as high as four at 2^{26}, is substantially more significant than in the sequential case.

Table 5.6 compares the performance of the parallel WHT with coarse-grained DDL

Table 5.6: Performance of parallel WHT with DDL at size 2^{26}

Thread	Coarse-grained			Fine-grained w/ ID shift		
	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

* The WHT transforms use small grain static scheduling + PowerPC RS64 III

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.

To understand the improvement due to fine-grained DDL it is helpful to look at the trees that were selected (Fig. 5.9). Recall that our analysis of the sequential performance (see Figure 3.3) indicated that balanced partitions are preferred to unbalanced partitions. However, for the parallel WHT with coarse-grained DDL, balanced partitions lead to bad 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 also Fig. 5.7). 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.

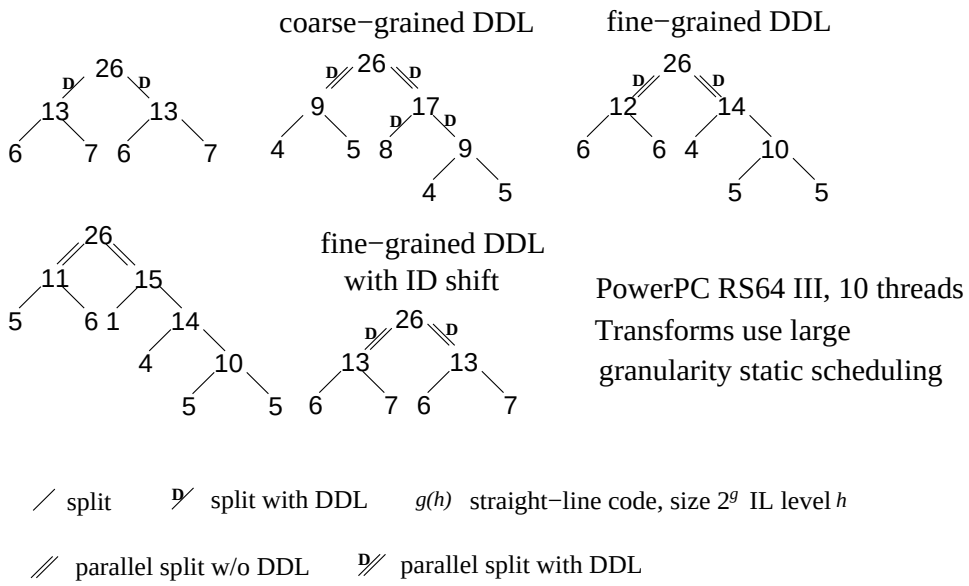

Figure 5.9: Best partition tree of different DDL scheduling schemes

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 Fig. 5.7 and 3.9.

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 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. One processor obtaining the permission puts the other processor in idle waiting state. Moreover, if the processor modifies the data, it automatically invalidates the whole cache line stored in the other processor and consequently causes cache misses in that processor. Once the other processor obtains the permission, it repeats the whole process on this processor. This phenomenon is called "ping pong"—a performance hazard in SMP systems [10].

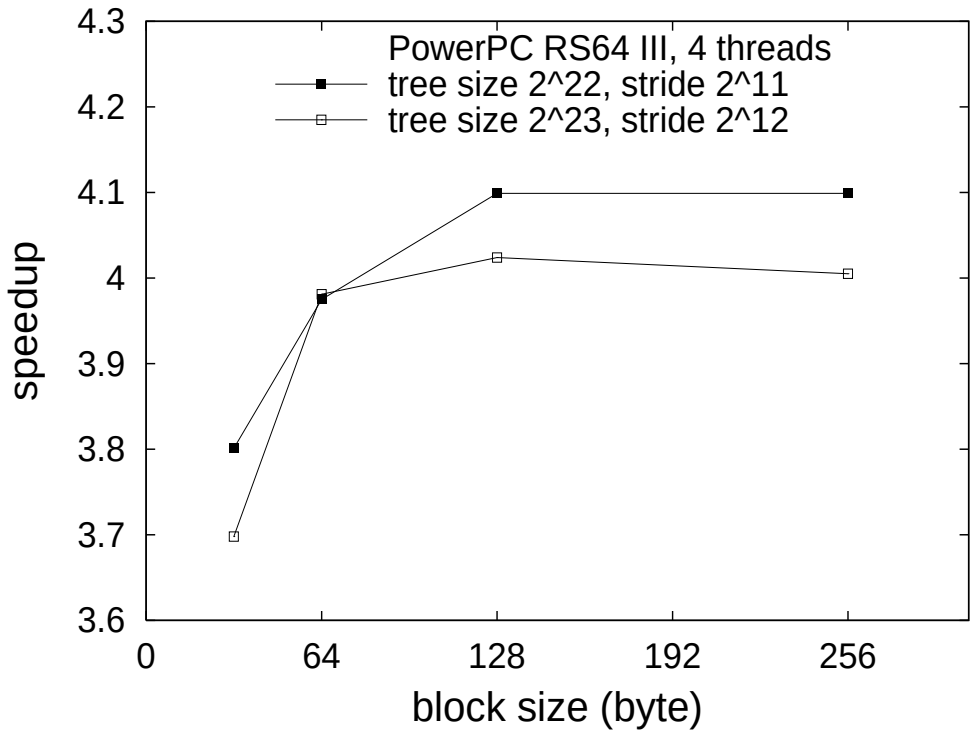

Figure 5.10: Effect of DDL block size on parallel performance

Figure 5.10 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.

6 Conclusion

In this thesis, 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 find the best parallel node and the partition of the WHT tree. Further optimization is obtained by adjusting granularity, load balance and cache utilization. Empirical performance data on sample SMP architectures (PowerPC RS64 III and UltraSPARC) 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 extend our package to work on distributed memory multiprocessors. The package and updated performance data can be obtained from http://www.ece.cmu.edu/~spiral.

Bibliography

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

[2] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP.* Morgan Kaufman Publishers, 2000.

[3] Matteo Frigo and Steven 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*, October 2000.

[5] Hung-Jen Huang. *Performance Analysis of an Adaptive Algorithm for the Walsh-Hadamard Transform.* Master Thesis at Drexel University, 2002.

[6] IBM. The RS/6000 enterprise server model S80, technology and architecture. Technical report. `http://www.rs6000.ibm.com/resource/technology/s80techarch.html`.

[7] IBM. `POWER3: The next generation of PowerPC processors`. Technical report, 2000. `http://researchweb.watson.ibm.com/journal/rd/446/oconnell.html`.

[8] 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.

[9] Jeremy Johnson and Markus Püschel. In search of the optimal Walsh-Hadamard transform. *ICASSP*, 2000.

[10] D. E. Lenoski and W. Weber. *Scalable Shared-Memory Multiprocessing.* Morgan Kaufmann Publishers, 1995.

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

[12] Sun Microsystems. `UltraSPARC-II Processor`. Technical report. `http://www.sun.com/microelectronics/UltraSPARC-II/`.

[13] Sun Microsystems. `UltraSPARC Processor`. Technical report. `http://www.sun.com/microelectronics/UltraSPARC/`.

[14] 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.

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

[16] Neungsoo Park and Viktor K. Prasanna. Cache conscious Walsh-Hadamard transform. *ICASSP*, 2001.

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

A Source Code of Parallel Scheduling

Our source code at SPIRAL website only includes the two best scheduling schemes, the large granularity scheduling of WHT transform and the fine-grained, ID shift scheduling of DDL pseudo-transpose. In this appendix, we list all schemes that we experimented and discussed in this thesis. There are two schemes for scheduling WHT transform and three schemes for scheduling DDL pseudo-transpose.

A.1 Scheduling of WHT transform

The following scheduling schemes only support a maximum of two factors in the WHT factorization in Equation 2.3 because the implementation of dynamic programming only searches binary WHT trees. However, the schemes can be extended to support nodes with more than two children.

A.1.1 Small Granularity

```
/* Small granularity static scheduling distribute each single transform
 * to threads in round robin.
 *
 * p_split in spiral_wht.c
 */
#pragma omp parallel private (xpt, id)
    {
      N  = W->N;
      Ni = W->priv.split.ns[1];
      R = W->priv.split.ns[0];
      total = omp_get_num_threads();
      id = omp_get_thread_num();

      block = Ni * S;
      while (id < R) {
        xpt = x + id * block;
        wht_apply(W->priv.split.Ws[1], S, xpt);
        id += total;
      }
```

```
#pragma omp barrier
      id = omp_get_thread_num();
      while (id < Ni) {
        xpt = x + id * S;
        wht_apply(W->priv.split.Ws[0], block, xpt);
        id += total;
      }
    }
```

A.1.2 Large Granularity

```
/* Large granularity static scheduling distribute large chunks
 * to threads.
 *
 * p_split in spiral_wht.c
 */
#pragma omp parallel private (xpt, id, j, chunk)
    {
      N  = W->N;
      Ni = W->priv.split.ns[1];
      R = W->priv.split.ns[0];
      total = omp_get_num_threads();
      block = Ni * S;

      chunk = R / total;
      if (chunk * total == R) {
        /* each thread has an equal share of work */
        id = omp_get_thread_num() * chunk;
        for (j = 0; j < chunk; ++ j, ++ id) {
          xpt = x + id * block;
          wht_apply(W->priv.split.Ws[1], S, xpt);
        }
      } else {
        /* some threads have more work to do */
        ++ chunk;
        id = omp_get_thread_num() * chunk;
        if (id <= R - chunk) {
          for (j = 0; j < chunk; ++ j, ++ id) {
            xpt = x + id * block;
            wht_apply(W->priv.split.Ws[1], S, xpt);
          }
        } else if (id < R) {
          for (; id < R; ++ id) {
            xpt = x + id * block;
            wht_apply(W->priv.split.Ws[1], S, xpt);
```

```
          }
        }
      }

#pragma omp barrier
      chunk = Ni / total;
      if (chunk * total == Ni) {
        /* each thread has an equal share of work */
        id = omp_get_thread_num() * chunk;
        for (j = 0; j < chunk; ++ j, ++ id) {
          xpt = x + id * S;
          wht_apply(W->priv.split.Ws[0], block, xpt);
        }
      } else {
        /* some threads have more work to do */
        ++ chunk;
        id = omp_get_thread_num() * chunk;
        if (id <= Ni - chunk) {
          for (j = 0; j < chunk; ++ j, ++ id) {
            xpt = x + id * S;
            wht_apply(W->priv.split.Ws[0], block, xpt);
          }
        } else if (id < Ni) {
          for (; id < Ni; ++ id) {
            xpt = x + id * S;
            wht_apply(W->priv.split.Ws[0], block, xpt);
          }
        }
      }
    }
```

A.1.3 Small Granularity with IL-enabled

```
/* Small granularity static scheduling distribute each single transform
 * to threads in round robin.
 *
 * p_split in spiral_wht.c with IL-enabled
 */
#pragma omp parallel private (xpt, id)
    {
      N  = W->N;
      Ni = W->priv.split.ns[1];
      R = W->priv.split.ns[0];
      total = omp_get_num_threads();
      id = omp_get_thread_num();
```

```
      block = Ni * S;
      while (id < R) {
        xpt = x + id * block;
        wht_apply(W->priv.split.Ws[1], S, xpt);
        id += total;
      }

#pragma omp barrier
      if((W->priv.split.Ws[0])->type != wht_small_il) {
        id = omp_get_thread_num();
        while (id < Ni) {
          xpt = x + (int) (id * S);
          wht_apply(W->priv.split.Ws[0], block, xpt);
          id += total;
        }
      } else {
        nIL = (W->priv.split.Ws[0])->nILNumber;
        id = omp_get_thread_num() * nIL;
        while (id < Ni) {
          xpt = x + id * S;
          wht_apply_4_para(W->priv.split.Ws[0], block, S, xpt);
          id += total * nIL;
        }
      }
    }
```

A.1.4 Large Granularity with IL-enabled

```
/* Small granularity static scheduling distribute large chunks
 * to threads.
 *
 * p_split in spiral_wht.c with IL-enabled
 */
#pragma omp parallel private (xpt, id, j, chunk)
    {
      N  = W->N;
      Ni = W->priv.split.ns[1];
      R = W->priv.split.ns[0];
      total = omp_get_num_threads();
      id = omp_get_thread_num();
      block = Ni * S;

      chunk = R / total;
      if (chunk * total == R) {
```

```
        /* each thread has an equal share of work */
        id = omp_get_thread_num() * chunk;
        for (j = 0; j < chunk; ++ j, ++ id) {
          xpt = x + id * block;
          wht_apply(W->priv.split.Ws[1], S, xpt);
        }
      } else {
        /* some threads have more work to do */
        ++ chunk;
        id = omp_get_thread_num() * chunk;
        if (id <= R - chunk) {
          for (j = 0; j < chunk; ++ j, ++ id) {
            xpt = x + id * block;
            wht_apply(W->priv.split.Ws[1], S, xpt);
          }
        } else if (id < R) {
          for (; id < R; ++ id) {
            xpt = x + id * block;
            wht_apply(W->priv.split.Ws[1], S, xpt);
          }
        }
      }

#pragma omp barrier
      if((W->priv.split.Ws[0])->type != wht_small_il) {
        chunk = Ni / total;
        if (chunk * total == Ni) {
          /* each thread has an equal share of work */
          id = omp_get_thread_num() * chunk;
          for (j = 0; j < chunk; ++ j, ++ id) {
            xpt = x + id * S;
            wht_apply(W->priv.split.Ws[0], block, xpt);
          }
        } else {
          /* some threads have more work to do */
          ++ chunk;
          id = omp_get_thread_num() * chunk;
          if (id <= Ni - chunk) {
            for (j = 0; j < chunk; ++ j, ++ id) {
              xpt = x + id * S;
              wht_apply(W->priv.split.Ws[0], block, xpt);
            }
          } else if (id < Ni) {
            for (; id < Ni; ++ id) {
              xpt = x + id * S;
```

```
            wht_apply(W->priv.split.Ws[0], block, xpt);
          }
        }
      }

    } else {
      nIL = (W->priv.split.Ws[0])->nILNumber;
      if (Ni < total * nIL) {
        /* Use small granularity at some unusual cases when there
         * isn't enough IL-blocks to parallelize with some many threads.
         */
        id = omp_get_thread_num() * nIL;
        while (id < Ni) {
          xpt = x + id * S;
          wht_apply_4_para(W->priv.split.Ws[0], block, S, xpt);
          id += total * nIL;
        }

      } else {
        chunk = Ni / total;
        if (chunk * total == Ni) {
          id = omp_get_thread_num() * chunk;
          for (j = 0; j < chunk; j += nIL, id += nIL) {
            xpt = x + id * S;
            wht_apply_4_para(W->priv.split.Ws[0], block, S, xpt);
          }
        } else {
          chunk = (Ni / (total * nIL)) * nIL;
          chunk += nIL;
          id = omp_get_thread_num() * chunk;
          if (id <= Ni - chunk) {
            for (j = 0; j < chunk; j += nIL, id += nIL) {
              xpt = x + id * S;
              wht_apply_4_para(W->priv.split.Ws[0], block, S, xpt);
            }
          } else if (id < Ni) {
            for (; id < Ni; id += nIL) {
              xpt = x + id * S;
              wht_apply_4_para(W->priv.split.Ws[0], block, S, xpt);
            }
          }
        }
      }
    }
  }
```

A.2 Scheduling of DDL Pseudo-Transpose

Like *splitddl*, the parallel *splitddl* supports only binary split of the WHT root node. The parallel scheduling for DDL pseudo-transpose is completely independent of the scheduling used for WHT transform, which may be either large granularity or small granularity as listed in the above. The following code for pseudo-transpose doesn't have OpenMP directives because the code is still implicitely within the parallel region defined in the parallel *splitddl*.

A.2.1 Coarse-grained Scheduling

```
/* in p_transpose.c */
void p_transpose(wht_value *xx, int n, int n1, int pll)
{
  wht_value **x;
  int B = BlockSize;
  int i, j, xbi, xbj;

  wht_value temp0, temp1, temp2, temp3;
  int n2 = n / n1;
  int id, total;

  x = (wht_value **) malloc(n1 * sizeof(wht_value *));

  if (pll) {
#ifdef _OPENMP
    total = omp_get_num_threads() * n1;
    id = omp_get_thread_num() * n1;
#else
    total = n1;
    id = 0;
#endif
  } else {
    total = n1;
    id = 0;
  }

  if (n1 < BlockSize)
    B = n1;

  while (id < n2) {
```

```
    x[0] = &(xx[0]) + id;
    id += total;

    for (i = 1; i < n1; ++ i)
      x[i] = x[i-1] + n2;

    for(xbj = 0; xbj < n1; xbj += B) {
      xbi = xbj;
      for(i = xbi; i< B + xbi; ++ i) {
        for (j = i + 1; j < B + xbj; ++ j) {
          temp0 = x[i][j];
          x[i][j] =  x[j][i];
          x[j][i] = temp0;
        }
      }
      xbi += B;
      for( ; xbi < n1; xbi += B) {
        for(j = xbj; j < B + xbj; ++ j) {
          for(i = xbi; i < B + xbi; i += NoUnroll) {
            temp0 = x[i][j];
            temp1 = x[i+1][j];
            temp2 = x[i+2][j];
            temp3 = x[i+3][j];

            x[i][j]   = x[j][i];
            x[i+1][j] = x[j][i+1];
            x[i+2][j] = x[j][i+2];
            x[i+3][j] = x[j][i+3];

            x[j][i]  = temp0;
            x[j][i+1]= temp1;
            x[j][i+2]= temp2;
            x[j][i+3]= temp3;
          }
        }
      }
    }
  }
  free(x);
}
```

A.2.2 Fine-grained Scheduling

```
/* in p_transpose.c */
void p_transpose(wht_value *xx, int n, int n1, int pll)
```

```
{
  wht_value **x;
  int B = BlockSize, Bi, Bj;
  int i, j, k, xbi, xbj;
  int band_id = 0, load, beta, shift, totalload, row = 0, col = 0;
  wht_value temp0, temp1, temp2, temp3;
  int n2 = n / n1;
  int id, total;

  if (n1 < BlockSize) {
    B = n1;
    beta = 1;
  } else {
    beta = n1 / BlockSize;
  }

  if (pll) {
#ifdef _OPENMP
    total = omp_get_num_threads() * n1;
    id = omp_get_thread_num() * n1;
#else
    total = n1;
    id = 0;
#endif
  } else {
    total = 1;
    id = 0;
  }

  x = (wht_value **) malloc(n1 * sizeof(wht_value *));

  /* determine total blocks (work) for each thread */
  totalload = beta * (beta + 1) / 2;
  shift = totalload % total;
  (shift) ? (load = totalload / total + 1) : (load = totalload / total);

  while (band_id < n2) {
    x[0] = &(xx[0]) + band_id;
    band_id += n1;

    for (i = 1; i < n1; ++ i)
      x[i] = x[i-1] + n2;

    col = load * id;
    if (col >= totalload) {
```

```
  id = (id + total - shift) % total;
  continue;
}
row = 0;
temp0 = beta;
while (col >= temp0) {
  col -= temp0;
  temp0 --;
  row ++;
}
xbj = row * B;
xbi = (col + row) * B;

for (k = 0; k < load; ++ k) {
  if (xbj >= n1) break;
  Bi = B + xbi;
  Bj = B + xbj;
  if (xbi == xbj) {
    for(i = xbi; i< Bi; ++ i) {
      for (j = i + 1; j < Bj; ++ j) {
        temp0 = x[i][j];
        x[i][j] =  x[j][i];
        x[j][i] = temp0;
      }
    }
  } else {
    for(j = xbj; j < Bj; ++ j) {
      for(i = xbi; i < Bi; i += NoUnroll) {
        temp0 = x[i][j];
        temp1 = x[i+1][j];
        temp2 = x[i+2][j];
        temp3 = x[i+3][j];

        x[i][j]   = x[j][i];
        x[i+1][j] = x[j][i+1];
        x[i+2][j] = x[j][i+2];
        x[i+3][j] = x[j][i+3];

        x[j][i]  = temp0;
        x[j][i+1]= temp1;
        x[j][i+2]= temp2;
        x[j][i+3]= temp3;
      }
    }
  }
```

```
      xbi += B;
      if (xbi >= n1) {
        xbj += B;
        xbi = xbj;
      }
    }
  }
  free(x);
}
```

A.2.3 Fine-grained Scheduling with ID shift

```
/* in p_transpose.c */
void transpose_pll(wht_value *xx, int n, int n1, int pll)
{
  wht_value **x;
  int B = BlockSize, Bi, Bj;
  int i, j, k, xbi, xbj;
  int band_id = 0, load, beta, shift, totalload, row = 0, col = 0;
  wht_value temp0, temp1, temp2, temp3;
  int n2 = n / n1;
  int id, total;

  if (n1 < BlockSize) {
    B = n1;
    beta = 1;
  } else {
    beta = n1 / BlockSize;
  }

  if (pll) {
#ifdef _OPENMP
    total = omp_get_num_threads() * n1;
    id = omp_get_thread_num() * n1;
#else
    total = n1;
    id = 0;
#endif
  } else {
    total = 1;
    id = 0;
  }

  x = (wht_value **) malloc(n1 * sizeof(wht_value *));
```

```
/* determine total blocks (work) for each thread */
totalload = beta * (beta + 1) / 2;
shift = totalload % total;
(shift) ? (load = totalload / total + 1) : (load = totalload / total);

while (band_id < n2) {
  x[0] = &(xx[0]) + band_id;
  band_id += n1;

  for (i = 1; i < n1; ++ i)
    x[i] = x[i-1] + n2;

  col = load * id;
  if (col >= totalload) {
    id = (id + total - shift) % total;
    continue;
  }
  row = 0;
  temp0 = beta;
  while (col >= temp0) {
    col -= temp0;
    temp0 --;
    row ++;
  }
  xbj = row * B;
  xbi = (col + row) * B;

  for (k = 0; k < load; ++ k) {
    if (xbj >= n1) break;
    Bi = B + xbi;
    Bj = B + xbj;
    if (xbi == xbj) {
      for(i = xbi; i< Bi; ++ i) {
        for (j = i + 1; j < Bj; ++ j) {
          temp0 = x[i][j];
          x[i][j] =  x[j][i];
          x[j][i] = temp0;
        }
      }
    } else {
      for(j = xbj; j < Bj; ++ j) {
        for(i = xbi; i < Bi; i += NoUnroll) {
          temp0 = x[i][j];
          temp1 = x[i+1][j];
          temp2 = x[i+2][j];
```

```
            temp3 = x[i+3][j];

            x[i][j]   = x[j][i];
            x[i+1][j] = x[j][i+1];
            x[i+2][j] = x[j][i+2];
            x[i+3][j] = x[j][i+3];

            x[j][i]  = temp0;
            x[j][i+1]= temp1;
            x[j][i+2]= temp2;
            x[j][i+3]= temp3;
          }
        }
      }
      xbi += B;
      if (xbi >= n1) {
        xbj += B;
        xbi = xbj;
      }
    }
    id = (id + total - shift) % total;
  }
  free(x);
}
```