Efficient Breadth-First Implementation of the Wavelet Transform

Michael D. Adams
Dept. of Elec. and Comp. Engineering, University of Victoria
PO Box 3055 STN CSC, Victoria, BC, V8W 3P6, Canada
mdadams@ece.uvic.ca

***Abstract*— *Cache-efficient breadth-first lifting-based algorithms for the wavelet transform (WT) are considered. Two optimizations for improving the efficiency of the WT computation are proposed. These optimizations are then applied to two different baseline WT algorithms and shown to be highly effective, reducing the execution time by as much as one third. Moreover, the resulting optimized WT algorithms are shown to be quite competitive with another more sophisticated algorithm, in spite of having very substantially reduced memory requirements.*

Keywords*— *wavelet transform, lifting, cache-efficient algorithms, JPEG 2000.

I. Introduction

Separable two-dimensional (2-D) wavelet transform (WTs) are used in many applications (e.g., [1], [2]). Consequently, efficient implementation techniques for such transforms are of great interest. The class of breadth-first [3] lifting-based [4] algorithms is particularly intriguing, due to the suitability of such algorithms in a wide variety of applications. It is this class of algorithms that is the focus of the work herein.

In this paper, we propose two optimizations for improving the efficiency of WT algorithms. The goal of our work is to produce fast memory-efficient algorithms that are also reasonably portable. For this reason, we do not explicitly consider instruction-set specific optimizations.

The remainder of this paper is structured as follows. Section II briefly introduces some of the notation and conventions used herein. Section III briefly discusses the WT and issues associated with its computation. Then, several WT computation strategies are proposed in Section IV, and their performance is evaluated in Section V. Finally, Section VI concludes with a summary of our work and some closing remarks.

II. Notation and Conventions

Before proceeding further a brief digression is in order regarding the notation and conventions employed herein. For an integer n, the notation $\lfloor n \rfloor$ denotes the largest integer not more than n (i.e., the floor function). Also, since C is used as the language of implementation in our work, all arrays are tacitly assumed to be indexed from zero and stored in row-major order.

III. Wavelet Transform

The basic building block of the WT is the 1-D uniformly maximally decimated filter bank. Herein, we consider the lifting realization [4] of such a filter bank, which has the general form shown in Fig. 1. In the diagram, the $\{A_k\}$ are filters (called lifting filters), the $\{s_k\}$ are amplifiers, and the $\{Q_k\}$ are optional rounding operators. The forward transform, associated with the system shown in Fig. 1(a), consists of a split operation (i.e., polyphase decomposition) followed by a number of lifting/scaling operations, while the inverse transform, associated with the system shown in Fig 1(b), consists of a number of scaling/lifting operations followed by a join operation (i.e., polyphase recomposition). The split and join operations are effectively permutations, while the lifting/scaling operations correspond to filtering. The split operation rearranges the elements of an array such that the elements of the first-polyphase component appear first (in order), followed by the elements of the second-polyphase component (in order). The join operation is simply the inverse of the split operation. In other words, the split and join operations allow us to move between the two representations of a sequence shown in Fig. 2.

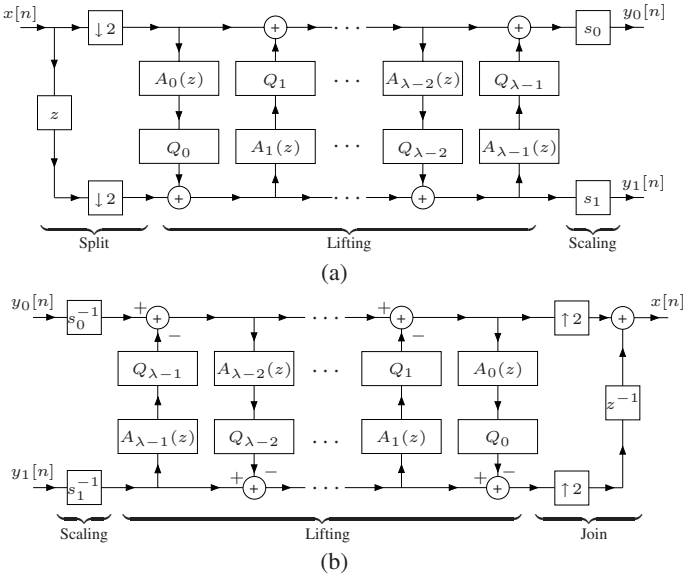

Fig. 1. Lifting realization of a 1-D filter bank. (a) Analysis side. (b) Synthesis side.

The well-known 5/3 WT from the JPEG-2000 standard [1] has an underlying 1-D filter bank with the lifting parameters:

$$\lambda = 2, \quad A_0(z) = -\tfrac{1}{2}(z+1), \quad A_1(z) = \tfrac{1}{4}(1+z^{-1}),$$
$$Q_0(x) = -\lfloor -x \rfloor, \quad Q_1(x) = \left\lfloor x + \tfrac{1}{2} \right\rfloor, \quad s_0 = s_1 = 1. \tag{1}$$

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

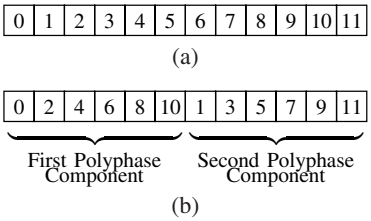

Fig. 2. Two different representations of sequence of length $N = 12$. (a) Original sequence. (b) Polyphase decomposition of sequence (obtained via split operation).

Although, in some parts of this paper, we specifically consider the implementation of the 5/3 WT, our ideas are equally applicable to other WTs as well (such as the 9/7 WT from [1]).

To form a 2-D WT, we simply apply the 1-D WT in each of the horizontal and vertical directions. This process is repeated L times recursively to the lowpass signal in order to form an L-level WT. Unfortunately, the 2-D nature of the WT poses some problems for efficient implementation. Performing operations in the horizontal direction can be done very efficiently, as neighbouring elements in a row are contiguous in memory. Processing in the vertical direction, however, can be extremely inefficient, due to the large stride in memory between neighbouring elements in a column, which typically reduces the effectiveness of memory caches. In the extreme case, every element in a single column maps to the same cache set, resulting in an extremely large number of cache misses during vertical processing. In practice, this situation usually occurs when the array width is an integer multiple of some sufficiently large power of two. This leads to a periodic spike in the WT execution time as one increases the array width. For example, this behavior is evident in some of the later plots, such as Figs. 7(a) and 8(a). To mitigate this problem, we normally perform vertical processing on groups of columns together in a process known as stripmining [5] (or loop tiling). Although this does not, by any means, eliminate the problem, it does help to reduce the problem's severity.

IV. Proposed Methods

In our work, we consider two baseline algorithms for the computation of the WT. Both of these algorithms are stripmined, due to the effectiveness of this technique, especially in pathological cases. In what follows, we only discuss the WT algorithms in a 1-D context, as the 2-D WT algorithms are directly constructed from the 1-D algorithms in a straightforward manner.

The first of the two baseline algorithms, which we refer to as the **non-interleaved filtering (NIF)** scheme, is the one that follows most naturally from the block diagrams shown in Fig. 1. The forward WT is implemented by first applying a split (i.e., permutation) operation, followed by lifting/scaling (i.e., filtering) operations, while the inverse transform is realized by first applying lifting/scaling (i.e., filtering) operations, followed by a join (i.e., permutation) operation.

The second of the two baseline algorithms, which we refer to as the **interleaved filtering (IF)** scheme, represents a slight departure from the algorithm described above. In this scheme, the order of the permutation and filtering operations are reversed in each of the forward and inverse WT algorithms. That is, the forward transform first performs lifting/scaling operations (i.e., filtering), followed by a split (i.e., permutation) operation, while the inverse transform first applies a join (i.e., permutation) operation, followed by scaling/lifting (i.e., filtering) operations. Due to the re-ordering of the permutation and filtering operations, the two polyphase components are interleaved at the time of filtering (hence, the name of this algorithm). This interleaved structure, however, results in substantially different memory access patterns as compared to the NIF case (from above). Thus, the effects of optimization on the NIF and IF baseline algorithms may be different. For this reason, we choose to explicitly consider both baseline algorithms herein.

In what follows, we will now consider two different optimizations that can be applied to each of the above two baseline algorithms. Ideally, through these optimizations, we seek to improve the WT algorithm performance in pathological cases without having an overly adverse effect in the other remaining cases.

A. Modified Split/Join (MSJ)

The first of our proposed optimizations affects the way in which split/join operations are performed. In what follows, let us consider a split/join operation applied to a data array `orig` of length N.

Traditionally, the split operation is accomplished via the following three-step algorithm (e.g., as in [3]): 1) copy the elements of the second-polyphase component in the data array `orig` to a temporary array `tmp`; 2) copy the elements of the first-polyphase component to their correct final positions in the data array `orig`; 3) copy the elements of the temporary array `tmp` to their correct final positions in the data array `orig`. This algorithm is illustrated in Fig. 3. In the diagrams, copy operations are denoted by arrows, and in instances where the ordering of such operations is important, the arrows are numbered to indicate the correct order.

Traditionally, the join operation is implemented in an analogous way to above. The algorithm consists of the following three steps: 1) copy the elements of the second-polyphase component to the temporary array `tmp`; 2) copy the elements of the first-polyphase component to their correct final positions in the array `orig`; 3) copy the elements from the temporary array `tmp` to their correct final positions in the array `orig`. This process is illustrated in Fig. 4.

Now, we make a few observations concerning the traditional split and join algorithms introduced above. First, the temporary array `tmp` needs to be of size $\frac{N}{2}$ (approximately). Second, each of the three steps in these algorithms requires (about) $\frac{N}{2}$ reads and $\frac{N}{2}$ writes. Thus, the split and join algorithms each need (approximately) $\frac{3N}{2}$ reads and $\frac{3N}{2}$ writes in total.

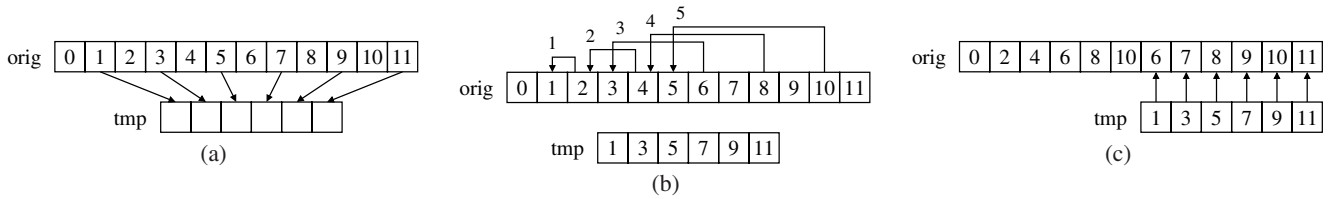

Fig. 3. Traditional split algorithm for an array of length $N = 12$. (a) First, (b) second, and (c) third steps.

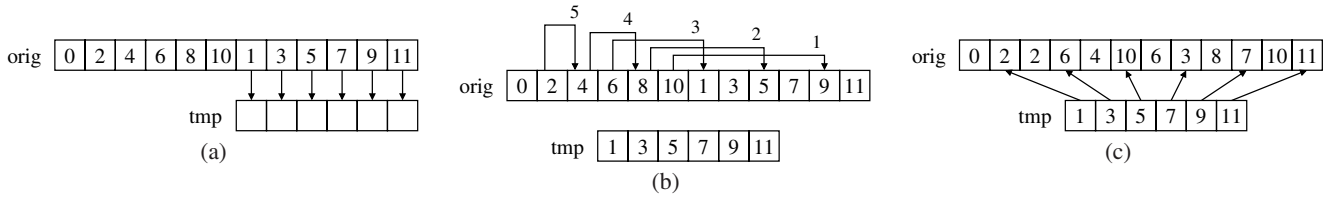

Fig. 4. Traditional join algorithm for an array of length $N = 12$. (a) First, (b) second, and (c) third steps.

Fortunately, we can improve upon the above situation. As for how to accomplish this, the key insight can be obtained by considering what happens if we do not perform step 1 in the split algorithm above. A careful analysis shows that only (about) half of the samples are lost in so doing. Thus, we can implement the split operation by saving only (approximately) half of the second-polyphase-component (i.e., odd-indexed) samples, in particular, those that will subsequently be overwritten. The resulting two-step algorithm is illustrated in Fig. 5. Observe that the temporary array `tmp` in this case is (approximately) half of the size of the one used in the traditional split algorithm. Furthermore, the number of memory accesses is reduced. The first step requires (approximately) $\frac{3N}{4}$ reads and $\frac{3N}{4}$ writes, while the second step requires (approximately) $\frac{N}{2}$ reads and $\frac{N}{2}$ writes. Thus, a total of (about) $\frac{5}{4}N$ reads and $\frac{5}{4}N$ writes are necessary, a 16.7% reduction in total memory accesses from the traditional method.

We can also modify the join algorithm in a similar way as above. This leads to an algorithm like that shown in Fig. 6. As in the case of the split algorithm, this modification to the join algorithm reduces the total number of memory accesses by 16.7% and reduces the size of the temporary array by one half. In the remainder of this paper, we will refer to the above enhancements to the split/join algorithms as the **modified split/join (MSJ)** optimization.

In some applications, the above memory savings (due to a reduced-size temporary array) could be quite desirable, especially with stripmined vertical filtering (which requires a larger-sized temporary array). More importantly, as will be demonstrated later, the reduced number of memory accesses can also lead to significant reductions in WT execution time, especially in the pathological cases. Lastly, in passing, we note that the above optimizations (with minor modifications) can also be applied if one desires to follow the convention of placing the odd-indexed samples first in the data array during a split operation (as is sometimes done, for example, in JPEG 2000 [1]).

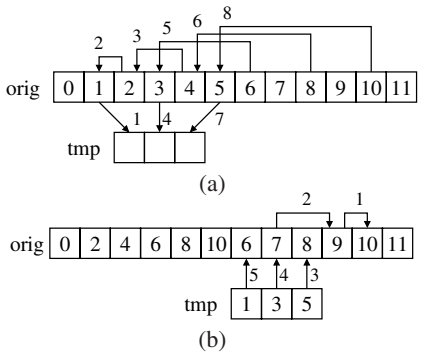

Fig. 5. New split algorithm for an array of length $N = 12$. (a) First and (b) second steps.

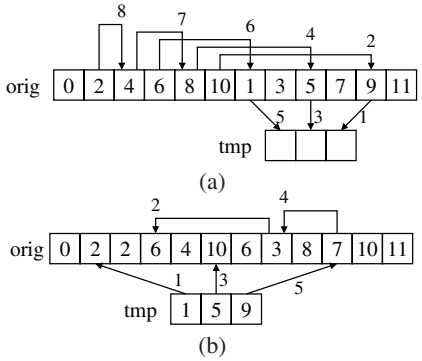

Fig. 6. New join algorithm for an array of length $N = 12$. (a) First and (b) second steps.

B. Pipelined Filtering (PF)

The second of our proposed optimizations affects the way in which filtering is performed. In the classic WT implementation, each lifting operation is applied in totality before commencing any computation for subsequent lifting operations. With our proposed optimization, the computation for all lifting operations are performed simultaneously with the computation of quantities beginning as soon as the necessary data dependencies are satisfied. Drawing on an analogy with hardware pipelining, we shall refer to this optimization as

Table 1. System Specifications

Processor	Intel Pentium M 733 (Dothan) 1.10 GHz
L1 Data Cache	32 KB, 8-way, 64-byte lines
L2 Unified Cache	2 MB, 8-way, sectored, 64-byte lines

pipelined filtering (PF). In passing, we note that, for the case of $\lambda \leq 2$ (in Fig. 1), the PF optimization is similar to ideas proposed in [3], [6], except that stripmining is not considered in these other works.

To make the PF optimization more concrete, we explain how it can be applied to the 5/3 WT as defined by (1). Let l and h denote the arrays associated with the first- and second-polyphase components (which may or may not be interleaved with one another in memory, depending on whether the optimization is being applied to the NIF or IF baseline WT algorithm). Neglecting the initialization and termination conditions (which can be derived in a straightforward manner), the body of the main loop in the forward WT has the form:

1) $h[k] := h[k] - \left\lfloor \frac{1}{2}(l[k] + l[k+1]) \right\rfloor$
2) $l[k] := l[k] + \left\lfloor \frac{1}{4}(h[k-1] + h[k] + 2) \right\rfloor$
3) $k := k + 1$

(Here, the symbol ":=" denotes assignment.) Similarly, the body of the main loop in the inverse WT has the form:

1) $l[k] := l[k] - \left\lfloor \frac{1}{4}(h[k-1] + h[k] + 2) \right\rfloor$
2) $h[k] := h[k] + \left\lfloor \frac{1}{2}(l[k] + l[k+1]) \right\rfloor$
3) $k := k - 1$

In order to avoid unnecessarily complicating the above example, stripmining has been ignored in the preceding pseudocode.

The PF optimization described above is advantageous as it reduces the number of passes that need to be made over the WT data, resulting in greater locality for memory references and improved cache efficiency.

V. Experimental Results

Having introduced our proposed (MSJ and PF) WT optimizations, we will now proceed to evaluate their performance. To do this, for the case of the 5/3 WT (as introduced earlier), we implemented the proposed baseline (NIF and IF) WT algorithms as well as the proposed optimizations. In our implementation, we elected to stripmine using groups of 16 columns, as this provides a reasonable tradeoff between performance and memory requirements (which increase with column group size). For further comparison purposes, we also implemented the nonrecursive-layout WT algorithm described in [7]. We shall henceforth refer to this as the **Chatterjee-Brooks (CB)** algorithm. All of the code was written in C and executed on a system with the processor and cache specifications shown in Table 1. Also, 32-bit integers were used for the sample data (to be transformed).

As suggested earlier, the width of the 2-D data array to be transformed has, by far, the most dominant effect on execution time. Thus, in order to better understand how algorithm performance varies with array size, we chose to conduct timing experiments in which the data-array height was fixed at 1024 while the width was varied. In all of our experiments, a 5-level wavelet decomposition was employed, typical of that used in image coding applications. In what follows, we begin by presenting the results obtained for each of the NIF and IF baseline WT algorithms.

A. Non-Interleaved Filtering (NIF) Case

For each of the forward and inverse WTs, we measured the execution time of the baseline NIF algorithm as well as the relative change in execution time when each of the MSJ, PF, and combined MSJ-PF optimizations were employed. The results are shown in Figs. 7 and 8. Since, in the figures, the relative time differences use the baseline as a reference, negative values correspond to a reduction in execution time as compared to the baseline. Evidently, from Figs. 7(a) and 8(a), the pathological cases (i.e., the cases where the execution time spikes) occur at widths that are multiples of 512 (which correspond to either every element or every other element in a column mapping to the same cache set). To more clearly see from the graphs the behavior of the various schemes in these pathological cases, the points associated with multiple-of-256 widths have been marked with the symbol "$\times$".

Let us now examine the forward transform results in Fig. 7. From Figs. 7(b) and (c), it is clear that the MSJ and PF optimizations significantly reduce execution time for all array widths. The MSJ optimization yields a reduction of 6.8 to 15.9% (with a median of 10.2%), while the PF optimization yields a reduction of 4.7 to 17.9% (with a median of 7.2%). Moreover, from Fig. 7(d), it is evident that the MSJ and PF optimizations can be combined to good effect. More specifically, the combined MSJ-PF optimization produces a time savings of 12.9 to 31.1% (with a median of 16.8%). Often, the largest time reductions come at multiple-of-512 widths, where the original baseline algorithm is slowest, and therefore, stands to benefit most.

Consider now, the inverse transform results in Fig. 8. From Figs. 8(b) and (c), we can observe that the MSJ optimization yields a time savings of up to 13.1% (with a median of 5.1%). In particular, it is highly effective for larger widths and multiple-of-512 widths. The PF optimization is extremely effective, yielding a time reduction of 5.7 to 24.4% (with a median of 14.5%), and a very signficiant reduction for multiple-of-512 widths. From Fig. 8(d), it is evident that the MSJ and PF optimizations can be used together to good effect. The combined MSJ-PF optimization produces a time savings of up to 26.9% (with a median of 18.0%). Furthermore, the improvement is extremely significant for multiple-of-512 widths, namely about 15 to 27%.

B. Interleaved Filtering (IF) Case

Next, we evaluated our two proposed optimizations in the IF case. In particular, for each of the forward and inverse WTs, we measured the execution time for baseline IF algorithm as well as the relative difference in execution time obtained when

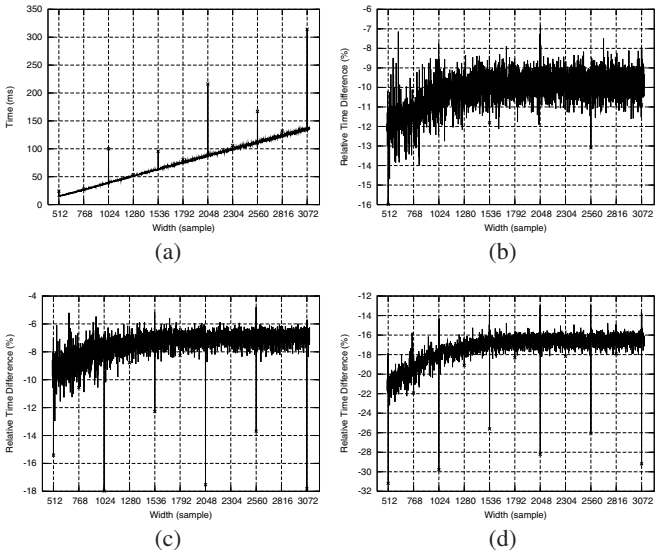

Fig. 7. Forward WT performance comparison for the NIF case. (a) Execution time for the baseline algorithm. Relative execution times with the addition of the (b) MSJ, (c) PF, and (d) combined MSJ-PF optimizations.

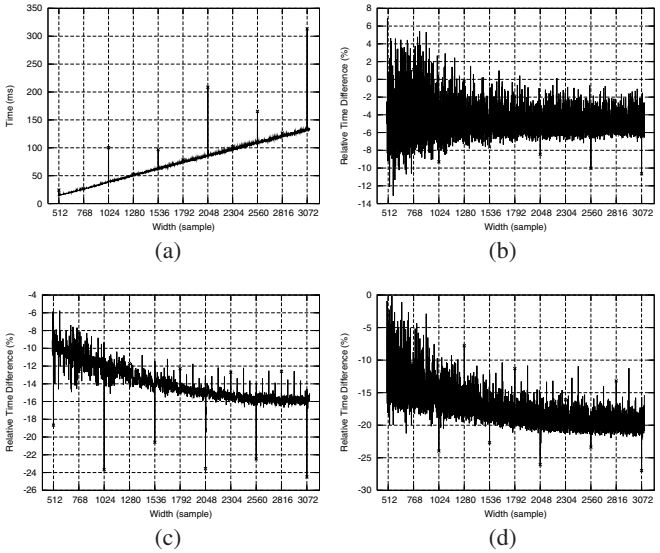

Fig. 8. Inverse WT performance comparison for the NIF case. (a) Execution time for the baseline algorithm. Relative execution times with the addition of the (b) MSJ, (c) PF, and (d) combined MSJ-PF optimizations.

each of the MSJ, PF, and combined MSJ-PF optimizations were employed. The results are shown in Figs. 9 and 10.

Let us first examine the results for the forward WT in Fig. 9. From Figs. 9(b) and (c), we can see that both the MSJ and PF optimizations significantly improve performance for all array widths. More specifically, the MSJ optimization yields an improvement of 4.9 to 14.1% (with a median of 6.7%), while the PF optimization offers an improvement of 6.4 to 19.2% (with a median of 14.8%). From Fig. 9(d), it is clear that the best performance is obtained from the combined MSJ-PF optimization. In this case, an improvement of 15.1 to 27.7% (with a median of 21.5%) is obtained. For each of the optimizations considered, the most significant improvements occur at multiple-of-512 widths, where the baseline algorithm can benefit most.

Consider now, the inverse transform results in Fig. 10. From Fig. 10(b), we see that MSJ optimization leads to improved performance at multiple-of-512 widths (about 2 to 8%). Also, the median behavior is clearly better for larger widths. From Fig. 10(c), it is evident that the PF optimization leads to improved performance for all array widths, offering time savings of 2.6 to 28.8% (with a median of 8.5%), and very large improvements at multiple-of-512 widths (of about 23 to 29%). Lastly, from Fig. 10(d), we see that although the combined MSJ-PF optimization does perform well, it does not offer a substantial improvement over the PF optimization alone. This is due to the relatively poorer performance of the MSJ optimization in this case. This, however, is the only instance where the combined MSJ-PF optimization does not yield significantly better performance than that obtained with each of the MSJ and PF optimizations alone.

C. Additional Comparison

To further illustrate the benefits of our proposed methods, we compare our combined MSJ-PF scheme to the CB method (introduced earlier). Although the CB approach is highly efficient, it has one extremely important drawback, namely the size of the temporary array required.

Let W and H denote the width and height of the data array to be transformed (in samples), and let c denote the number of columns grouped together in stripmining (where, as indicated earlier, $c = 16$ in our experiments). With our approach, the number of entries required in the temporary array is $\frac{1}{2}c\max(W,H) = 8\max(W,H)$. On the other hand, with the CB approach the number of entries required in the temporary array is WH. Clearly, in practical situations, the difference in memory requirements is extremely large. For example, in the case of a 1024×1024 data array, the sizes of temporary arrays required by our approach and the CB approach are 8192 and 1048576, respectively. In other words, our approach needs less than 1% of the auxiliary memory required by the competing scheme.

Of course, we must also consider how the different approaches compare in terms of execution time. To do this, we compared the relative difference in execution times between the CB approach and our approach for the forward and inverse WTs. The results are shown in Fig. 11 (where negative values correspond to the CB method being faster). On each graph, we include two sets of numbers, labelled "dynamic" and "static", the only difference between them being in how the memory for the temporary array is allocated in the CB approach, namely either dynamically or statically. Static allocation is generally faster than dynamic allocation, especially for large-size memory requests. With the CB approach, however, it would be infeasible in many applications to statically allocate the temporary array, due to its extremely large size. Thus, in a practical sense, the dynamically-allocated case is most

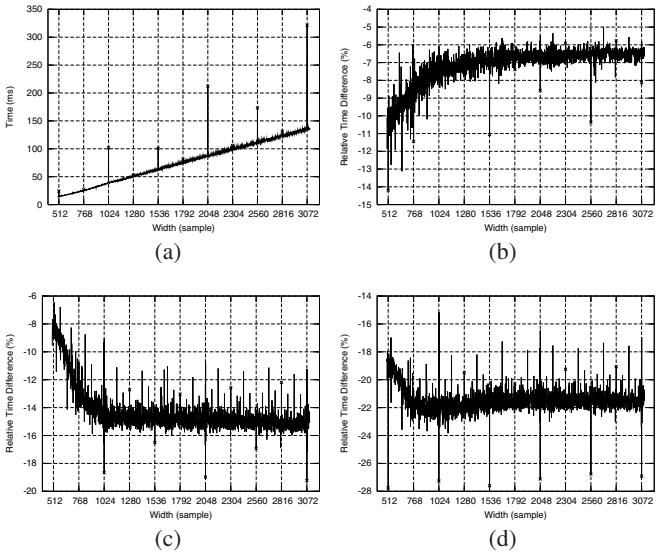

Fig. 9. Forward WT performance comparison for the IF case. (a) Execution time for the baseline algorithm. Relative execution times with the addition of the (b) MSJ, (c) PF, and (d) combined MSJ-PF optimizations.

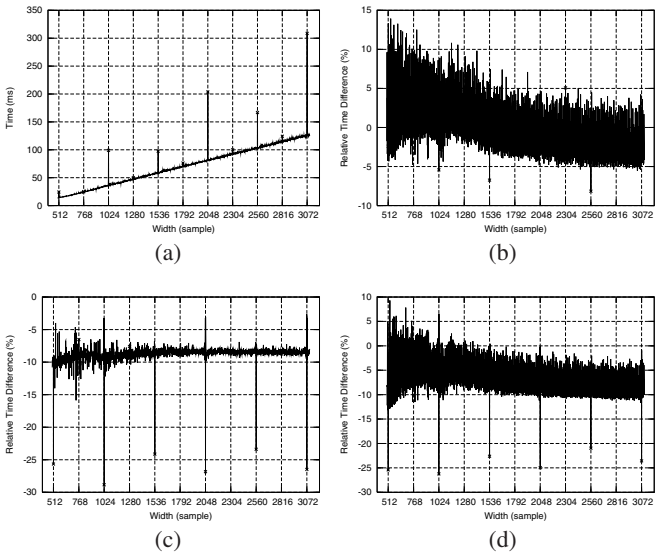

Fig. 10. Inverse WT performance comparison for the IF case. (a) Execution time for the baseline algorithm. Relative execution times with the addition of the (b) MSJ, (c) PF, and (d) combined MSJ-PF optimizations.

important. Nevertheless, it is still interesting to consider the statically-allocated case. (In the case of our approach, the above issue does not arise, since the temporary array is always relatively small and can be dynamically allocated without excessive time penalties.)

Examining the results shown in Fig. 11, we can make the following observations. For both the forward and inverse WT, our approach is faster than the CB approach with dynamic allocation, often by more than 10% and sometimes by as much as 93%. This is rather impressive, as the CB approach uses orders of magnitude more memory. Even in the case of static allocation, our approach compares quite favorably

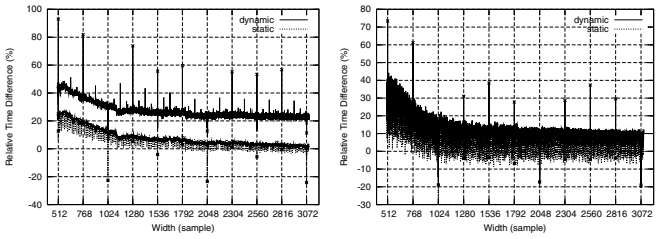

Fig. 11. Relative difference in execution time between the CB approach and our approach for the (a) forward and (b) inverse transforms.

with the CB approach, considering the significant difference in memory requirements between the two approaches. This further demonstrates the power of our methods proposed herein.

VI. Conclusions

In this paper, we proposed two optimizations for improving the efficiency of the WT computation. These optimizations were employed in two different baseline WT algorithms, and shown to lead to improved efficiency, sometimes reducing the execution time by one third. Furthermore, the optimizations were also shown to lead to WT algorithms that compete quite favorably with more sophisticated methods while requiring only a small fraction of the memory cost. Clearly, our methods can benefit the many applications in which the WT is employed.

Acknowledgments

The author would like to thank Dr. Siddhartha Chatterjee and Christopher Brooks for providing additional details concerning the forward WT algorithm proposed in [7], including a sample implementation. This information allowed a more faithful reproduction of their algorithm to be used for comparison purposes herein.

References

[1] *ISO/IEC 15444-1: Information technology—JPEG 2000 image coding system—Part 1: Core coding system*, 2000.

[2] M. D. Adams and R. K. Ward, "JasPer: A portable flexible open-source software toolkit for image coding/processing," in *Proc. of IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, Montreal, PQ, Canada, May 2004, pp. 241–244.

[3] Y. Andreopoulos, P. Schelkens, G. Lafruit, K. Masselos, and J. Cornelis, "High-level cache modeling for 2-D discrete wavelet transform implementations," *Journal of VLSI Signal Processing*, vol. 34, no. 3, pp. 209–226, July 2003.

[4] W. Sweldens, "The lifting scheme: A new philosophy in biorthogonal wavelet constructions," in *Proc. of SPIE*, vol. 2569, San Diego, CA, USA, Sept. 1995, pp. 68–79.

[5] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann, 1997.

[6] Y. Andreopoulos, P. Schelkens, and J. Cornelis, "Analysis of wavelet transform implementations for image and texture coding applications in programmable platforms," in *IEEE Workshop on Signal Processing Systems*, Antwerp, Belgium, Sept. 2001, pp. 273–284.

[7] S. Chatterjee and C. D. Brooks, "Cache-efficient wavelet lifting in JPEG 2000," in *IEEE International Conference on Multimedia and Expo*, vol. 1, Lausanne, Switzerland, Aug. 2002, pp. 797–800.