Interframe Video Compression with Large Dictionaries of Tilings: Algorithms for Tiling Selection and Entropy Coding

Kai-Lung Hua, Rong Zhang, Mary Comer, and Ilya Pollak*

Abstract

H.264/AVC employs variable-block-size motion compensation that can significantly improve the coding performance compared with previous video coding standards. In this paper, we propose the use of large dictionaries of tilings for video compression. We construct a rate-distortion cost function that admits fast search algorithms to select the optimal tiling for the motion compensation stage of a video coder. The computation of the cost is enabled through novel algorithms to approximate the bit rate and the distortion. We propose efficient arithmetic coding algorithms to encode the selected tiling. We illustrate the effectiveness of our approach by showing that a video coder utilizing one of the proposed tiling selection methods results in up to 17% savings in bit rate for several standard video sequences as compared to H.264/AVC. This is accomplished with only a modest increase in the computation time at the encoder.

Index Terms

Large tree-structured dictionaries, motion compensation, H.264/AVC.

This work was supported in part by a National Science Foundation (NSF) CAREER award CCR-0093105, an NSF grant IIS-0329156, and a MediaTek fellowship. Preliminary versions of this work have been presented in part at the 2006 IEEE International Conference on Image Processing [1] and the 2006 Asilomar Conference on Signals, Systems, and Computers [2].

Kai-Lung Hua is with National Taiwan University of Science and Technology, Department of Computer Science and Information Engineering, #43, Sec. 4, Keelung Rd, Taipei 106, Taiwan, phone +886-2-2730-3664, fax +886-2-2730-1081, email hua@mail.ntust.edu.tw. Rong Zhang is with Qualcomm Incorporated, phone 858-651-6128, email zhangr@qualcomm.com. Mary Comer and Ilya Pollak are with School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Ave., West Lafayette, IN 47907, phone 765-494-3486 and 5916, fax 765-494-3358, e-mail comerm,ipollak@ecn.purdue.edu. Corresponding author's e-mail: ipollak@ecn.purdue.edu.
I. INTRODUCTION

Algorithms for best basis search in tree-structured dictionaries have been effectively used for many signal processing problems, including noise removal [3]–[8], curve and surface modeling and reconstruction [9], [10], image compression [11]–[22], audio compression [23], [24], and image segmentation [25]. An important class of best basis algorithms are methods that search for an optimal rectangular tiling of an image [3], [9], [11]–[14], [17]–[22], [25], [26]. Such methods can significantly improve the performance of image coding strategies based on block or lapped transforms through adapting the sizes and shapes of transform blocks to the structure of an image. The utility of using variable block sizes in video compression has long been recognized as well [20], [27]–[32], and has been incorporated into video compression standards. For example, the H.264/AVC video compression standard [33] incorporates variable block sizes in the motion compensation stage [34]. This allows better adaptation to the geometry of motion and leads to improved coding efficiency.

In the present paper, we show that video compression performance can be further improved by searching for the optimal tiling in the dyadic and multitree dictionaries introduced in [12] for 2D image compression. Both these dictionaries are much larger than the set of tilings allowed for motion compensation in H.264/AVC. In general, allowing the video coder to select a tiling from a larger dictionary implies that more overhead bits will be required to encode the selected tiling, possibly increasing the overall bit rate. Our algorithms overcome the larger overhead and produce overall bit rate reductions of up to 17% at typical PSNRs, compared to H.264/AVC, on several video sequences commonly used to evaluate the performance of video coders. This is accomplished through the following major contributions of the present paper.

1) In Section III we construct a rate-distortion cost that allows us to use the efficient tiling algorithms of [12] to select the optimal tiling. Since computing the actual rate and actual distortion associated with every tiling in the dictionary would be computationally infeasible, we use two approximations:
   • we develop an algorithm to approximate the number of overhead bits required to encode a tiling;
we develop a motion prediction algorithm to approximate the distortion and the non-overhead portion of the bit rate.

2) In Section IV, we develop a novel arithmetic coding algorithm to efficiently encode the selected tiling. This algorithm adaptively learns local probability models that exploit intra-frame spatial correlations. The basic idea of the algorithm is that a block is more likely to be finely tiled if its neighbor blocks are finely tiled.

Combined together, these two contributions insure that the increase in the number of overhead bits as compared to H.264/AVC is moderate, and is more than compensated for by the reduction in the number of bits required to encode the transform coefficients which stems from the greater adaptivity of our scheme.

We start in Section II with an overview of the H.264/AVC motion compensation scheme. After describing our new coder in Sections III and IV, we present experimental results in Section V.

II. OVERVIEW OF MOTION COMPENSATION IN H.264/AVC

There are two basic ways of encoding a frame of video: intra mode and inter mode. Intra-frame compression only uses the information contained within the current frame. Inter-frame mode predicts the current frame from one or several reference frames, and encodes the error between the predicted frame and the actual one. The ability to accurately predict the current frame is therefore crucial to the success of inter-frame compression. The inter-frame predictors are typically determined by motion estimation methods which partition a frame into blocks and match every block with a similar block in the reference frame. Conventional methods generally use blocks of fixed size. The choice of the block size is problematic for such methods: if the size is too small, too many bits are spent on encoding the motion vectors; whereas if it is too large, the prediction of complicated motion sequences is poor. Generally speaking, areas with no motion should use large block sizes and areas with complicated motion should use small block sizes. Therefore, H.264/AVC incorporates variable block sizes in the motion compensation stage.

Specifically, H.264/AVC allows partitioning each $16 \times 16$ macroblock into rectangular blocks and performing motion estimation separately for each block. The specific tilings allowed by the standard
Fig. 1. Possible tilings of a 16 × 16 macroblock for the H.264/AVC motion estimation and compensation.

Fig. 2. Possible tilings of an 8 × 8 block for the H.264/AVC motion estimation and compensation.

are shown in Figs. 1 and 2. One of the four tilings in Fig. 1 may be used for a 16 × 16 macroblock. If the tiling of Fig. 1(d) is used, then each of the four 8 × 8 blocks can be further tiled using the four tilings of Fig. 2. Thus, this scheme selects from the dictionary of tilings that contains a total of $3 + 4^4 = 259$ tilings for each macroblock. These tilings contain a total of $9 + 8 \cdot 4 = 41$ distinct blocks. For each macroblock, the encoder in the H.264/AVC reference software [35] first finds the optimal motion vector for each of the 41 blocks, and then selects the optimal tiling from the 259 allowed tilings. During the motion vector selection stage, each motion vector $v$ is assigned the following rate-distortion cost:

$$d(v) + \lambda_m r(v).$$  \hfill (1)

Here, the distortion $d(v)$ is the sum, over all the pixels in the block, of the absolute differences between the pixel luminance and the luminance of the corresponding pixel in the reference frame, for integer pixel search; and the sum of absolute transformed differences for fractional pixel search. The rate $r(v)$ is defined as the number of bits required to encode the difference between the motion vector and the predicted\(^1\) motion vector. The estimates for this rate are tabulated in [35]. The Lagrange multiplier $\lambda_m$ is

$$\lambda_m = \sqrt{0.85 \cdot 2^{(QP-12)/3}},$$  \hfill (2)

where $QP$ is the quantization parameter.

\(^1\)A motion vector for a block is predicted based on the neighboring blocks, and then the difference between the prediction and the actual motion vector is encoded.
The optimal motion vectors are obtained by optimizing the cost (1) over a certain window [35]. After determining the optimal motion vectors for each block, the cost of each tiling of a macroblock into blocks is defined as a weighted sum of the tiling’s distortion $D$ and rate $R$: \( \text{COST} \equiv D + \lambda R \). Both the distortion and the rate are assumed to be additive over the blocks in the tiling, i.e., it is assumed that the cost of a tiling that consists of blocks $B_1, \ldots, B_d$ can be calculated as follows:

\[
\text{COST} = \sum_{i=1}^{d} (D_{B_i} + \lambda R_{B_i}),
\]

where $D_{B_i}$ and $R_{B_i}$ are the distortion and rate, respectively, for the block $B_i$, and the summation is taken over all blocks in the tiling. Specifically, the distortion $D_{B_i}$ is the sum of the squared differences between each original pixel in the block $B_i$ and its motion-compensated reconstruction\(^2\); and the rate $R_{B_i}$ is the sum of three terms: the number of bits $R_{B_i}^{(t)}$ to encode the selected tiling, the number of bits $R_{B_i}^{(e)}$ for the predicted motion vector error, and the number of bits $R_{B_i}^{(c)}$ for the quantized transform coefficients for block $B_i$:

\[
R_{B_i} = R_{B_i}^{(t)} + R_{B_i}^{(e)} + R_{B_i}^{(c)}.
\]

The Lagrange multiplier in Eq. (3) is defined as $\lambda \equiv \lambda_m^2$ where $\lambda_m$ is given by Eq. (2). The tiling selection procedure for a macroblock globally optimizes the cost (3) over the dictionary of 259 tilings.

### III. Optimal Tiling Algorithm

In this section, we propose a rate-distortion cost function that allows us to adapt the fast algorithm of [12] to solve the problem of selecting the optimal tilings from very large dictionaries. We start by discussing the algorithm’s adaptation to our problem and describing the dictionaries in Section III-A. We then describe our proposed rate-distortion cost function in Section III-B. Since the exact evaluation of rates and distortions during our optimization process would be computationally infeasible, we instead use rate

\(^2\)A motion-compensated reconstruction is the sum of the following two terms [34]: (1) the predicted pixel value and (2) the pixel value that results from transforming, quantizing, inverse-quantizing, and inverse-transforming the difference between the original frame and its prediction.
and distortion estimates in the definition of the cost function. The algorithm for estimating the overhead rate—i.e., the number of bits needed to encode a tiling—is introduced in Section III-C. Section III-D describes the motion prediction algorithm utilized in estimating the distortion and the non-overhead portion of the bit rate.

A. Fast Optimal Tiling Search

The algorithm of [12] constructs optimal tree-structured tilings of a rectangular image domain into smaller rectangles via recursive bipartitioning by vertical and horizontal splits. During this process, a rectangle may either be used as a single block or split further into two subblocks. For example, the tiling of Fig. 3(a) may be obtained through such a recursive bipartitioning process, as illustrated in Fig. 3(c,d). In Fig. 3(c,d), a vertical (horizontal) line through a tree node signifies a vertical (horizontal) split of the corresponding rectangle into two subrectangles. Note that in this case, two different trees correspond to the same tiling. On the other hand, the rectangular tiling of Fig. 3(b) cannot be obtained through such a recursive binary splitting process.

Referring back to Fig. 3(c,d), we point out that in some applications the important object is the tiling produced by the leaves of the tree, and thus there is no distinction between the different trees that may have produced the tiling. In our video compression algorithm, however, the various tilings will be encoded by encoding the corresponding trees and therefore we will select the tree that corresponds to the most efficient encoding. In other words, our tiling optimization algorithm will select the optimal tiling tree along with the optimal tiling. The following cost function for a tree with leaves $B_1, \ldots, B_d$ and internal nodes $Q_1, \ldots, Q_{d-1}$ is a simplified version of Eq. (8) in [12]:

$$C(\text{tree}) = \sum_{i=1}^{d} e(B_i) + \sum_{i=1}^{d-1} s(Q_i),$$

(5)

where $e$ and $s$ are the cost functions for leaf blocks and internal tree nodes, respectively. Figs. 6 and 7 in [12] give a general algorithm to find the optimal tree for such a cost function.

The smallest block size that we use is $4 \times 4$. We define two dictionaries of trees: the dyadic dictionary
Fig. 3. An illustration of tilings and tiling trees (reproduced from Fig. 1 of [12]). (a) An tree-structured tiling—i.e., a tiling that can be obtained via recursive binary splitting in horizontal and vertical directions. (b) A non-tree-structured inadmissible tiling. (c) A tree of splits that leads to the tiling in (a). (d) Another tree of splits that leads to the tiling in (a).

Fig. 4. Example of possible dyadic and multitree tilings.

and the multitree dictionary. The former contains all trees that produce tilings by only splitting rectangles in the middle, horizontally or vertically. The latter allows splits at arbitrary locations that are multiples of four. A tiling from the dyadic dictionary and a tiling from the multitree dictionary are illustrated in Fig. 4. Note that the dyadic dictionary is a subset of the multitree dictionary.

In order to find the globally optimal tree, the algorithm of [12] performs the following bottom-up optimization:

\[
C^*_B = \min \{ e(B), \min (C^*_{B'}, C^*_{B''}) + s(B) \},
\]

where \( C^*_B \) is the cost of the optimal tree with root \( B \), and the inner minimization is performed over all
pairs of blocks \((B', B'')\) which partition \(B\).

To avoid repetitive calculation, the optimal cost and its corresponding pair of subblocks \((B', B'')\) for each block \(B\) are recorded in a table. We consult the table before making a recursive call for any block \(B\) to make sure that \(B\) has not been visited before. With this table, we only need to make one recursive call per block. For the dyadic dictionary, each evaluation of the right-hand side of Eq. (6) involves comparing at most three possibilities: horizontal split vs vertical split vs no split. Therefore, for the dyadic dictionary, the complexity of the optimization algorithm\(^3\) applied to one macroblock is \(O(\text{number of possible blocks in a macroblock})\). [Note that for H.264/AVC, the computational complexity of the optimization is also \(O(\text{number of possible blocks in a macroblock})\).] For the multilayer dictionary, each evaluation of the right-hand side of Eq. (6) for a \(4M \times 4N\) block \(B\) involves comparing \(M + N - 1\) choices. Therefore, for the multilayer dictionary, the complexity of the optimization algorithm applied to a macroblock of size \(4N \times 4N\) is \(O(N \times \text{number of possible blocks in a macroblock})\).

The efficient search algorithm exploits the fact that although the number of possible trees and tilings is very large, the number of rectangular blocks is much smaller and manageable—-and it is the latter that determines the complexity of the algorithm. We compare the number of tilings and the number of blocks per macroblock for the H.264/AVC, dyadic, and multilayer dictionaries in Table I [36]. The numbers of blocks in the dyadic and multilayer dictionaries are only about 1.2 and 2.4 times larger than the number of H.264/AVC blocks, respectively. Yet, the respective numbers of tilings are approximately 26 and 264 times that of the H.264/AVC dictionary. Thus, with a modest increase in the computational burden, we are able to search over much sets of tilings.

\(^3\)The complexity bounds given here are for the best tiling search only, and exclude the computation of the costs. The complexity of computing the cost of a block is the same for H.264/ACV and for our algorithms. This is because, as explained in Section III-B, our cost function is similar to that of H.264/AVC.
TABLE I
NUMBER OF TILINGS AND BLOCKS IN A 16 × 16 MACROBLOCK

<table>
<thead>
<tr>
<th></th>
<th>H.264/AVC</th>
<th>Dyadic</th>
<th>Multitree</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of tilings</td>
<td>259</td>
<td>6857</td>
<td>68480</td>
</tr>
<tr>
<td>number of blocks</td>
<td>41</td>
<td>49</td>
<td>100</td>
</tr>
</tbody>
</table>

B. Rate-Distortion Cost

We now explain the specifics of the cost function in Eq. (5) that we have developed for our motion compensation application. The cost $s(Q_i)$ of an internal tree node $Q_i$ is defined as $\lambda R_i$, where $\lambda$ is a Lagrange multiplier, and $R_i$ is the estimate of the number of bits spent on indicating that $Q_i$ is not a leaf block and on encoding the split location of the block $Q_i$. Computing the exact value of this number of bits for every internal node of every tree is computationally infeasible, as this would require encoding every tree in the dictionary using our entropy coder of Section IV. Instead, we approximate it using the simple encoding scheme described in Section III-C.

Similar to H.264/AVC, we define the cost function $e(B_i)$ of Eq. (5) for a leaf block $B_i$ as a weighted sum of the distortion estimate $D_{B_i}$ and the rate estimate $R_{B_i}$:

$$e(B_i) = D_{B_i} + \lambda R_{B_i}.$$  

We define $D_{B_i}$ as an estimate of the sum of the squared differences between each pixel in the block $B_i$ and its motion-compensated reconstruction. This estimate is calculated using the motion prediction algorithm described in Section III-D. We borrow the value for the Lagrange multiplier $\lambda$ from H.264/AVC: $\lambda = \lambda_m^2$ with $\lambda_m$ given by Eq. (2). We define the rate $R_{B_i}$ as the sum of the estimates of three terms: the number of bits $R_{B_i}^{(e)}$ for the predicted motion vector error for block $B_i$, the number of bits $R_{B_i}^{(c)}$ for the quantized transform coefficients for block $B_i$, and the number of bits $R_{B_i}^{(s)}$ to indicate that $B_i$ is a leaf, i.e., that $B_i$ is not split further. The estimates $R_{B_i}^{(e)}$ and $R_{B_i}^{(c)}$ are calculated using the motion prediction algorithm described in Section III-D, whereas the estimated rate $R_{B_i}^{(s)}$ is, according to the approximation described
below in Section III-C, taken to be one bit per leaf:

\[
R_{B_i} = R_{B_i}^{(e)} + R_{B_i}^{(c)} + 1. \tag{7}
\]

Substituting our \(e(B_i) = D_{B_i} + \lambda R_{B_i} \) and \(s(Q_i) = \lambda R_i \) into the cost equation (5), we obtain the following overall cost of a tree:

\[
C(\text{tree}) = \sum_{i=1}^{d} \left[ D_{B_i} + \lambda (R_{B_i}^{(e)} + R_{B_i}^{(c)} + 1) \right] + \lambda \sum_{i=1}^{d-1} R_i. \tag{8}
\]

Fig. 5 illustrates different tiling selection methods for the motion compensation stage.

C. Approximate Computation of the Overhead Rate

We approximate the overhead bit rate using the following simple scheme that encodes the structure of a tree.
• One bit is used to encode whether \( Q_i \) is an internal node or a leaf.

• For an internal node, one more bit is used to encode its split direction (i.e., whether it is split horizontally or vertically).

• For the dyadic dictionary, there is nothing else to encode. For the multitree dictionary, if \( Q_i \) is a \( 4M \times 4N \) block, then there are \( M - 1 \) possible locations for a horizontal split. If \( M \geq 2 \), we assume that the horizontal split location can be encoded with \( \lceil \log_2 (M - 1) \rceil \) additional bits. Similarly, if \( N \geq 2 \), we assume that the vertical split location can be encoded with \( \lceil \log_2 (N - 1) \rceil \) bits.

Under this scheme, the encoding of an internal tree node in a dyadic dictionary would require two bits. The encoding of an internal tree node in a multitree dictionary would require \( 2 + \lceil \log_2 (M - 1) \rceil \) bits if it is split horizontally and \( 2 + \lceil \log_2 (N - 1) \rceil \) bits if it is split vertically. Our approximate rate \( R_i \) therefore looks like this:

\[
R_i = \begin{cases} 
2, & \text{dyadic dictionary} \\
2 + \lceil \log_2 (M - 1) \rceil, & \text{multitree dictionary, } Q_i \text{ is } 4M \times 4N \text{ and is split horizontally } (M \geq 2) \\
2 + \lceil \log_2 (N - 1) \rceil, & \text{multitree dictionary, } Q_i \text{ is } 4M \times 4N \text{ and is split vertically } (N \geq 2) 
\end{cases}
\]

Note that the simple encoding scheme that yields \( R_i \) is only used to approximate the number of bits when we search for the optimal tiling. During the actual encoding, a much more sophisticated scheme is used, as described in Section IV. This more sophisticated scheme yields a significantly better compression performance, as demonstrated in Section IV-C.

\[\text{D. Motion Prediction Algorithm Used during Optimal Tiling Search}\]

The motion vector for a block indicates the offset from a prediction reference in a previously encoded frame. Vectors for neighboring blocks are often correlated since object motion may extend across large areas of a frame. Compression of the motion vector field may be improved by predicting each motion vector from previously encoded vectors. The motion vector difference (the difference between the predicted and actual motion vector) is encoded and sent to the decoder. In H.264/AVC, the basic motion vector predictor for a block, \( B \), is the median of the motion vectors for the three following blocks:
Fig. 6. An example to illustrate motion vector prediction.

Fig. 7. Two possible upper-left neighbors for a block $B_1$ in dyadic tilings: $B_2$ and $B_3$.

- the block $L(B)$ which is the topmost block among the blocks bordering $B$ on the left;
- the block $T(B)$ which is the leftmost block among the blocks bordering $B$ from above;
- the block $D(B)$ which is the block immediately to the right of the rightmost block bordering $B$ from above.

This is illustrated in Fig. 6 using a macroblock which consists of four blocks, $B_1$, $B_2$, $B_3$, and $B_4$. In the figure, $L(B_1) = A_4$, $T(B_1) = A_1$, and $D(B_1) = A_2$. Therefore, the predicted motion vector for $B_1$ is the median of $MV_{A_1}$, $MV_{A_2}$, and $MV_{A_4}$, where $MV_i$ denotes the motion vector for block $i$. The predictor is modified accordingly if some of the neighboring blocks are not available.

In addition to the actual encoding, the motion vector predictor is involved in the tiling selection procedure. This is because the motion vector predictor is one of the ingredients in the definitions of the distortion and the rate that form the cost functions both for an H.264/AVC tiling in Eq. (3) and for our tiling tree in Eq. (8). Thus, in order to compute the cost of a tiling during the tiling selection procedure, it
is necessary to know $L(B_i), T(B_i),$ and $D(B_i)$ for every potential block $B_i$ in the tiling. In H.264/AVC, the tiling selection procedure for a macroblock $MB$ is performed after the tilings for all the macroblocks that are above $MB$, as well as the tiling for the macroblock immediately to the left of $MB$, have all been determined. This order of computations, together with the special structure of the possible H.264/AVC tilings in Figs. 1 and 2, ensures that $L(B_i), T(B_i),$ and $D(B_i)$ are known for any block $B_i$ in any potential tiling of $MB$, enabling the computation of the cost for any candidate tiling during the tiling selection stage of H.264/AVC. For our tiling selection algorithm, however, $L(B_i), T(B_i),$ and $D(B_i)$ may not be uniquely defined for some blocks $B_i$, as illustrated through two dyadic tilings of a macroblock shown in Fig. 7.

For the tiling of Fig. 7(a), $L(B_1) = B_2$; whereas, for the tiling of Fig. 7(b), $L(B_1) = B_3$. Exhaustive search over all the neighbor possibilities for each block would make our algorithm too computationally expensive. We therefore introduce a different motion vector predictor for the tiling selection stage of our algorithm. For any block $B$ within a macroblock $MB$, our motion vector predictor is the median of the motion vectors for the blocks $L'(B), T'(B),$ and $D'(B)$, where

- $L'(B) = L(B)$ if $L(B)$ is uniquely defined—i.e., if $B$ is on the extreme left of the macroblock $MB$. Otherwise, $L'(B) = L(MB)$.
- $T'(B) = T(B)$ if $T(B)$ is uniquely defined—i.e., if $B$ is at the top of the macroblock $MB$. Otherwise, $T'(B) = T(MB)$.
- $D'(B) = D(B)$ if $D(B)$ is uniquely defined—i.e., if $B$ is at the top of the macroblock $MB$. Otherwise, $D'(B) = D(MB)$.

For example, in Fig. 6, our motion vector predictor for $B_1$ is the same as for H.264/AVC; our motion vector predictor for $B_2$ is the median of $MV_{A_2}, MV_{A_3},$ and $MV_{A_4}$; our motion vector predictor for $B_4$ is the median of $MV_{A_1}, MV_{A_3},$ and $MV_{A_4}$.

This motion prediction algorithm is used to compute the estimate $D_{B_i}$ of the distortion and the estimate $R_{B_i}^{(e)} + R_{B_i}^{(c)}$ of the non-overhead rate in the tiling tree cost equation (8).
Fig. 8. The rate-distortion curves the proposed motion prediction method used in the optimal tiling search step (black dashed line with circles) and for a simpler alternative in which the motion vector predictor for any block $B$ within a macroblock $MB$ is the median of the motion vectors for the blocks $L(MB)$, $T(MB)$, and $D(MB)$ (dotted line with triangles). For both methods, H.264/AVC motion prediction is applied during the actual encoding. The right panel shows bit rates as percentages of the bit rate for the alternative method. These curves are constructed for Stefan sequence with $QP = 22, 26, 30, 34$.

Note that once the optimal tiling is found, every block $B$ has unique $L(B)$, $T(B)$, and $D(B)$ which are available to use in the encoding stage. We therefore revert to the H.264/AVC motion vector prediction strategy when performing the actual encoding.

Fig. 8 compares our motion prediction algorithm for the dyadic dictionary with a simpler alternative which uses $L'(B) = L(MB)$, $T'(B) = T(MB)$, and $D'(B) = D(MB)$ for every block $B$ within a macroblock $MB$ during the optimal tiling search stage. For both methods, H.264/AVC motion prediction is applied during the actual encoding. Video sequence Stefan is used to compare the two methods, and four experiments are performed, using quality parameter settings $QP = 22, 26, 30, 34$. The rate-distortion curve for the proposed method is shown as a black dashed line with circles; the rate-distortion curve for the alternative method is shown as a dotted line with triangles. The right panel shows the rates as the percentages of the rate for the simpler method, indicating that our method results in up to 2.2% savings in bit rate as compared to the other method.

IV. Entropy Coders for the Optimal Tiling Tree

We use the standard H.264/AVC encoding methods [33], [34] to encode everything except the tiling structure. Since our tiling dictionaries are very different from the dictionary used in H.264/AVC, we
have no choice but to develop our own algorithms for encoding the tiling structure. We describe these algorithms in Sections IV-A and IV-B for the dyadic and multitree dictionaries, respectively.

To encode a tree, we first map it to a sequence of binary symbols which, following the terminology of [37], we call bins. We define several context models which are empirical conditional probability distributions of the bins. These models are dynamically updated during the encoding process. We use the arithmetic coder of [38] in our implementation. This combination of context modeling with arithmetic coding is a standard approach which has been used in many coders such as the Q-coder used in JPEG [39], the QM-coder used in JBIG [40], the MQ-coder used in JPEG2000 [41], and the CABAC coder used in H.264/AVC [37]. Our context models aim to exploit data dependencies in the trees constructed by our optimal tiling methods of Section III.

A. Entropy Coder for the Optimal Dyadic Tiling Tree

The tree structure is encoded starting with the root of a tree. The tree is traversed so that for every internal node that is split along a vertical line, the subtree rooted at the left child is fully traversed before the subtree rooted at the right child. For every internal node that is split along a horizontal line, the subtree rooted at the top child is fully traversed before the subtree rooted at the bottom child. Every internal node is traversed before any of its children.

We use zero, one, or two bins to encode the split status of each node in the tree. The first bin, if needed, indicates whether the node is a leaf: 0 means that it is a leaf, 1 means that it is an internal node. The second bin, if needed, encodes the orientation of the split boundary: 0 for horizontal, 1 for vertical. Thus, the string 0 denotes a leaf node in the tiling tree; the string 10 denotes an internal node which is split along a horizontal line; and the string 11 denotes an internal node which is split vertically.

The first bin is used for every node corresponding to a block larger than \(4 \times 4\). Since our smallest block size is \(4 \times 4\), all \(4 \times 4\) blocks are automatically leaves and cannot be split further. Thus, neither the first nor the second bin is required for such a block. No \(4 \times 8\) or \(4 \times 16\) block can be split along a horizontal line, since 4 is our smallest vertical extent for a block. Thus, the second bin is not needed for
Fig. 9. Binarization for the dyadic tiling encoder. For each block, bin one encodes the absence or presence of a split; if a split is present, then bin two encodes the orientation of the split boundary: 0 for horizontal, 1 for vertical. (a) Bin one is 0, and there is no bin two. (b) Bin one is 1, bin two is 0. (c) Bin one is 1, bin two is 1. (d) Bin one is 1, bin two is not needed since horizontal splitting is not allowed.

Fig. 10. (a) A dyadic tiling tree. (b) Its encoding.

such blocks. Similarly, $8 \times 4$ and $16 \times 4$ blocks cannot be split along a vertical line, and therefore each of them also only requires the first bin. Both bins are needed for $16 \times 16, 16 \times 8, 8 \times 16$, and $8 \times 8$ blocks. The various scenarios are illustrated in Fig. 9.

The encoding of a tree is illustrated in Fig. 10. The split status of a parent node is always encoded before its children nodes. So, for example, the subtree rooted at node B in Fig. 10(a) is encoded as 1100. Specifically, node B is encoded as 11 to indicate that it is split vertically, and each child node is encoded as 0 to indicate that it is a leaf. Similarly, the subtree rooted at node C is also encoded as 1100. The subtree rooted at A is encoded as 1101100: 11 to indicate that node A is split vertically, followed by 0 to indicate that its left child is a leaf, followed by the encoding of the subtree rooted at its right child, node C. The entire tree is therefore encoded as 101101100110: 10 to indicate that the root node is split horizontally, followed by the encoding of the top subtree rooted at A, followed by the encoding of the bottom subtree rooted at B.

For each block size $4M \times 4N$ except $4 \times 4$, we maintain two context models for the first bin, denoted
Fig. 11. Illustration of context models for a 16 × 16 block $B$ in a dyadic tiling. (a) Context model $1_{1,4,4}$: $B$ has a single left neighbor, $C$, and a single top neighbor, $A$. The remaining panels correspond to context model $1_{2,4,4}$ (more than two neighbors total). (b) Context model $2_{1,4,4}$: more left neighbors than top neighbors. (c) Context model $2_{2,4,4}$: more top neighbors than left neighbors. (d) Context model $2_{3,4,4}$: equal number of top neighbors and left neighbors.

For each block size $4M \times 4N$ with $M, N \geq 2$, we maintain three context models for the second bin, denoted $2_{1,M,N}$, $2_{2,M,N}$, and $2_{3,M,N}$. In each case, the context of a block $B$ is determined by its immediate neighbors from the left and from above.

Recall that the first bin is 0 if the block is a leaf and 1 if the block is an internal node. The first context model for the first bin, $1_{1,M,N}$, is for the case when a block has one neighbor above it and one neighbor to the left of it, as block $B$ does in Fig. 11(a). In other words, the first context model contains the conditional probabilities for the block to be a leaf or an internal node, given that it has one top neighbor and one left neighbor. The second context model for the first bin, $1_{2,M,N}$, is for the case when a block has more than one neighbor either to the left or above, as block $B$ does in Fig. 11(b). In other words, the second context model contains the conditional probabilities for the block to be a leaf or an internal node, given that it has more than one neighbor either to the left or above. The two models correspond to situations when $B$ is surrounded by coarse tilings (model $1_{1,M,N}$) and when $B$ is surrounded by fine tilings (model $1_{2,M,N}$). In the latter case, one expects the conditional probability that $B$ is an internal node to be larger than in the former case, as illustrated by the histograms in Fig. 12.

Note that blocks along the top of a frame do not have any neighbors above, and blocks along the left side of a frame do not have any neighbors on the left. When constructing context models, all these blocks are handled as special cases. Specifically, each block that does not have any neighbors above is handled as though it has exactly one neighbor above; and each block that does not have any neighbors to the left is handled as though it has exactly one neighbor to the left.
Fig. 12. Two context models for the first bin of a 16 × 16 block. In both panels, ’0’ and ’1’ on the horizontal axis represent 16 × 16 blocks that are leaves and internal nodes, respectively. The left panel is the conditional histogram of 16 × 16 leaves and internal nodes given one neighbor above and one neighbor to the left. The right panel is the conditional histogram of 16 × 16 leaves and internal nodes given that there are at least two neighbors on the left or at least two neighbors above.

Fig. 13. Three context models 2_1,4,4, 2_2,4,4, and 2_3,4,4 for the second bin of a 16 × 16 block. In all three panels, ’0’ and ’1’ on the horizontal axis represent horizontal split and vertical split, respectively. The three models correspond to the situations when the number of neighbors on the left is, respectively, greater than (left panel), less than (center panel), and equal to (right panel) the number of neighbors above.

Recall that the second bin is only used for internal nodes, and encodes the orientation of the split: 0 for horizontal, and 1 for vertical. The three context models for the second bin, 2_1,M,N, 2_2,M,N, and 2_3,M,N, correspond to the situations when the number of neighbors on the left is, respectively, greater than, less than, and equal to, the number of neighbors above. These three scenarios are illustrated, respectively, in Figs. 11(b), 11(c), and 11(d), for block B. In the first scenario, we would typically expect that B is more likely to be split horizontally than vertically, as illustrated in the left panel of Fig. 13. In the second scenario, we would expect the vertical split to be more likely, as in the center panel of Fig. 13. In the third scenario, we would expect the conditional probabilities of B to be split horizontally and vertically to both be approximately 0.5. This is not quite the case for the right panel of Fig. 13, as horizontal splits are prevalent in the particular sequence used for generating Fig. 13.

The selected context model supplies two probability estimates: the probability that the bin contains
TABLE II

Bins 3 and 4 for the Multitree Entropy Coder

<table>
<thead>
<tr>
<th>Block Description</th>
<th>Bin 3</th>
<th>Bin 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4M \times 12$ block split into $4M \times 4$ and $4M \times 8$</td>
<td>0</td>
<td>not used</td>
</tr>
<tr>
<td>$4M \times 12$ block split into $4M \times 8$ and $4M \times 4$</td>
<td>1</td>
<td>not used</td>
</tr>
<tr>
<td>$4M \times 16$ block split into $4M \times 4$ and $4M \times 12$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$4M \times 16$ block split into $4M \times 8$ and $4M \times 8$</td>
<td>1</td>
<td>not used</td>
</tr>
<tr>
<td>$4M \times 16$ block split into $4M \times 12$ and $4M \times 4$</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

one and the probability that the bin contains zero. These estimates determine the two sub-ranges that the arithmetic coder [38] uses to encode the bin.

B. Entropy Coder for the Optimal Multitree Tiling Tree

We use the same tree traversal strategy as in our dyadic tree coder. As described in Section III, the difference between the dyadic and multitree dictionaries is that some blocks have more possible split locations under the multitree scheme. To accommodate this, we use up to four bins to encode the split status of each node.

No bins are used for $4 \times 4$ blocks since these blocks cannot be split. For all the other blocks, the first bin is the same as in the dyadic scheme: 0 for a leaf, 1 for an internal node. The second bin is also the same as in the dyadic scheme and indicates the orientation of the split: 0 for horizontal and 1 for vertical. This bin is not needed for $4 \times 4N$ and $4M \times 4$ blocks, as these blocks may only be split along one direction. The third bin is used for $4M \times 4N$ blocks for which $M \geq 3$ or $N \geq 3$. The fourth bin is used for $4M \times 4N$ blocks for which $M = 4$ or $N = 4$. The third and fourth bin are described for the case of a vertical split in Table II. The horizontal split is handled analogously.

For each block size $4M \times 4N$, the context models of multitree method for the first and second bins are the same as the ones of dyadic method. For the third and fourth bins, we simply use arithmetic coding without context models.
C. Performance of Tiling Tree Coders: An Example

A fixed-length encoder for an alphabet with $n$ different symbols must have codewords of length $\lceil \log_2 n \rceil$ bits. As shown in Table I, the dyadic dictionary has 6857 tilings for every $16 \times 16$ macroblock, and the multitree dictionary has 68480 tilings. Therefore, if we wanted to use fixed-length encoders to encode the dyadic and multitree tilings, we would need, respectively, $\lceil \log_2 6857 \rceil = 13$ bits and $\lceil \log_2 68480 \rceil = 17$ bits per macroblock. In Fig. 14, we compare the overhead bit rates for fixed-length encoders, for the simple encoding scheme of Section III-C, and for the entropy coders described in Sections IV-B and IV-A. The experiments are done using Stefan sequence encoded with $QP = 22, 26, 30, 34$.

Our multitree entropy coder outperforms fixed-length encoding by 45–70% and the scheme of Section III-C by 45–55%. Our dyadic entropy coder outperforms fixed-length encoding by 50-70% and the scheme of Section III-C by about 50%. Note that both the entropy coders and the scheme of Section III-C suffer from significant redundancy: they encode trees instead of tilings. It can be shown that the total number of trees per macroblock is 22899 and 2,590,351, for the dyadic and multitree dictionaries, respectively. Thus, instead of constructing codewords for the alphabet of 6857 tilings, our dyadic coders construct codewords for the alphabet of 22899 trees. Our multitree coders deal with the alphabet of

---

5This is a straightforward but tedious counting exercise, and hence we omit it.
2,590,351 trees instead of 68480 tilings. Despite this, our entropy coding algorithms are so effective that the resulting bit rates are much smaller than those of both the fixed-length encoders and the scheme of Section III-C. The simple scheme of Section III-C is quite naive in that it spends more bits on deeper trees. Its performance therefore deteriorates at higher PSNRs where typically finer tilings are selected, corresponding to deeper tiling trees.

V. EXPERIMENTAL RESULTS

The coding algorithms are evaluated using bus, Stefan, garden, and harbour video sequences. Bus, Stefan, and harbour are in common intermediate format (CIF, 352 × 288 pixels), and garden is in source input format (SIF, 352 × 240 pixels). The search range is ±16. Since our implementation works with single-channel frames, and since our compression strategy only differs from H.264/AVC for inter-frames, all our rate and distortion calculations are done with the luminance component of the inter-frames only.

We use a sixteen-frame group of pictures (GOP) with coding pattern IPPPPPPPPPPPPPPP, where I and P represent intra-frames and forward predicted inter-frames, respectively. In Fig. 15, we compare the peak signal-to-noise ratio (PSNR) of individual predicted inter-frames for the multitree algorithm (black dashed line with circles), dyadic algorithm (red solid line with stars), and H.264/AVC reference software [35] (dotted line with triangles). These curves are constructed for Stefan sequence, with $QP = 34$. We can see from Fig. 15 that in general, both multitree and dyadic algorithms have better PSNR performance as compared to the H.264/AVC reference software. The three solid straight lines in the figure show the average PSNR for each algorithm: 30.12dB, 30.11dB, and 29.75dB for multitree, dyadic, and H.264/AVC, respectively. Hence, on average, the multitree and dyadic algorithms more accurately predict the original frame than H.264/AVC reference software by 0.37dB and 0.36dB, respectively. Since our prediction errors have smaller energy, this suggests that we can reduce the number of bits required to encode the predicted

---

6We believe that the amount of improvement our algorithms would produce for the bidirectionally predicted B-frames would be similar to what we have for P-frames. We therefore opt not to have B-frames, as their implementation is significantly more complicated than that of P-frames.
error (the differences between the original and predicted frames).

For the second experiment, we use four quantization parameter settings ($QPP^7 = 22, 26, 30, 34$, and $QPI^8 = QPP - 2$). In this experiment, we use H.264/AVC reference software [35] as a baseline, and evaluate our new compression strategies which incorporate our proposed adaptive tiling algorithms of Section III in the motion compensation stage as well as our proposed entropy coding algorithms of Section IV in the encoding stage. Fig. 16 shows the rate-distortion curves for four test sequences. The right column of the figure shows the rate-distortion curves for the three schemes with the bit rates displayed as percentages of the baseline bit rate.

It can be seen from Fig. 16 that our proposed algorithms result in up to 17% savings in bit rate as compared to H.264/AVC. The improved performance is accompanied by a modest increase in computation time for the encoder. Note that the decoding complexity is the same for all three algorithms, and therefore the computation times of the three decoders are similar. For the H.264/AVC reference software encoder, the average running time per P-frame is approximately 1.5 seconds on a Pentium IV 3.2 GHz desktop. The average running time per P-frame on the same machine for the proposed dyadic and multitree encoders are 3.2 and 4.0 seconds, respectively, which is about 2.1 and 2.7 times slower than the H.264/AVC reference software.

$^7$QPP indicates the quantization parameter for P frames.

$^8$QPI indicates the quantization parameter for I frames.
Fig. 16. The rate-distortion curves of “CIF bus” (top row), “CIF Stefan” (second row), “SIF garden” (third row), and “CIF harbour” (bottom row). The right column shows bit rates as percentages of the bit rate for the H.264/AVC reference software.
The more complex multitree algorithm performs slightly worse than the dyadic algorithm in most cases. This is due to a significantly larger overhead bit rate incurred by the multitree algorithm.

Note that in two out of four panels in the right column of Fig. 16, multitree and dyadic rate-distortion curves follow a U-shaped pattern: our algorithms outperform H.264/AVC most significantly in the medium PSNR range. (In fact, the same pattern would be observed for the garden and harbour sequence as well if we extended the experiments to lower PSNRs.) The reasons for this pattern are mainly as follows. At very low PSNR, the motion vector differences and tiling structure dominate the bit rate. Since our proposed schemes require more overhead bits for our larger dictionaries than H.264/AVC, the gain from using our schemes is reduced in this regime. At very high PSNR, since all three algorithms choose very fine tilings, our schemes’ gains from using larger dictionaries are small.

VI. CONCLUSION

We have proposed algorithms for the selection of the optimal tiling for the motion compensation stage of a video coder and for the efficient encoding of the optimal tiling. Our algorithms lead to significant gains (up to 17% savings in bit rate on several typical sequences) as compared to H.264/AVC. This is accomplished with only a modest increase in the computation time at the encoder.

VII. ACKNOWLEDGMENT

The authors would like to thank Dr. Yan Huang, Prof. Minh Do, Prof. Charles Bouman, Prof. Edward Delp, and Prof. Fernando Pereira for stimulating discussions.

REFERENCES


[38] A. Said, “Fastac: Arithmetic coding implementation.”
