IEICE TRANS. INF. & SYST., VOL.E94–D, NO.4 APRIL 2011

PAPER

An H.264/AVC Decoder with Reduced External Memory Access for Motion Compensation

Jaesun KIM[†], **Younghoon KIM**[†], *Nonmembers*, *and* **Hyuk-Jae LEE**[†a)], *Member*

SUMMARY The excessive memory access required to perform motion compensation when decoding compressed video is one of the main limitations in improving the performance of an H.264/AVC decoder. This paper proposes an H.264/AVC decoder that employs three techniques to reduce external memory access events: efficient distribution of reference frame data, on-chip cache memory, and frame memory recompression. The distribution of reference frame data is optimized to reduce the number of row activations during SDRAM access. The novel cache organization is proposed to simplify tag comparisons and ease the access to consecutive 4x4 blocks. A recompression algorithm is modified to improve compression efficiency by using unused storage space in neighboring blocks as well as the correlation with the neighboring pixels stored in the cache. Experimental results show that the three techniques together reduce external memory access time by an average of 90%, which is 16% better than the improvements achieved by previous work. Efficiency of the frame memory recompression algorithm is improved with a 32x32 cache, resulting in a PSNR improvement of 0.371 dB. The H.264/AVC decoder with the three techniques is fabricated and implemented as an ASIC using 0.18 μm technology.

key words: *H.264/AVC decoder, motion compensation, external memory bandwidth, frame recompression*

1. Introduction

H.264/AVC is a video compression standard that employs strong compression techniques such as variable block size and quarter-sample accurate motion compensation [1], [2]. In order to support H.264/AVC motion compensation, an H.264/AVC decoder must access large amounts of data from a reference frame memory. As a result, motion compensation accounts for about 75% of the reference memory bandwidth required by an H.264/AVC decoder. This excessive memory access requirement for motion compensation is one of the main limitations in improving the performance of an H.264/AVC decoder [3].

A number of efficient techniques have been proposed to reduce the memory bandwidth required when accessing reference frames during motion compensation. A popular technique is to store data read from an external memory in on-chip buffers and reuse the data multiple times [3]–[6]. This data-reuse scheme is effective because common data are used multiple times by motion compensation of different blocks. In [3]–[5], adjacent reference blocks are fetched together when they have the same motion vectors. These techniques achieve a significant reduction in memory access by avoiding the repetitive access of data shared by adjacent blocks. In [6], a circular line cache is used to store the reference frame, achieving an improvement even for relatively large motion vectors.

Another approach to reducing memory access requirement is to adjust the order of data access events and thus minimize the overhead for memory access. Every time when data is accessed from a different row within an SDRAM, a row activation time is required. Therefore, a burst memory access with a large number of data is required to minimize the overhead from row activation. This can be achieved by adjusting the memory access order to increase burst data transfers [7]–[9]. The storage pattern of reference frame data can also be modified to allow a single burst to transfer the necessary data without unnecessary row activation overhead.

Another approach is known as frame memory recompression (also called embedded compression), which reduces the amount of data stored in memory at the expense of sacrificing image quality [10]–[16]. In this approach, the reference frame is slightly compressed before it is stored in an external memory. A number of frame memory recompression techniques have been proposed, and a popular one employs a transform-based approach in which a frame is decomposed into small blocks that are transformed into a frequency domain using a simple transform such as the discrete cosine transform (DCT), Hadamard transform or one of its variations [11]. The frequency domain coefficients are then compressed by quantization followed by variable length encoding, such as Golomb-Rice coding. Downsampling-based recompression requires a relatively small amount of computation [12]; however, image quality may be degraded due to edge pattern loss during downsampling for compression. Spatial domain compression based on differential pulse code modulation (DPCM) has been proposed in [13], [16]. An adaptive vector quantization scheme [14] and a differential Huffman coding based compression scheme [15] have been proposed for the compression of display systems.

This paper presents an H.264/AVC decoder that employs all three of the approaches discussed above in order to reduce the burden of motion compensation on external memory bandwidth. To this end, an efficient method for data mapping during motion compensation with the use of cache memory is investigated. In addition, an efficient organization of cache memory is proposed for motion compensation, and the frame memory recompression method in [16] is modified for use in an H.264/AVC decoder with cache mem-

Manuscript received March 4, 2010.
Manuscript revised November 21, 2010.

[†]The authors are with the Inter-university Semiconductor Research Center, Department of Electrical Engineering, Seoul National University, Seoul, Korea.
a) E-mail: hyuk_jae_lee@capp.snu.ac.kr

DOI: 10.1587/transinf.E94.D.798

ory. Use of all three techniques is found to reduce memory access time by 90%, which is 16% better than the reduction achieved by previous work. The H.264/AVC decoder employing the three techniques is fabricated using the Dongbu 1P6M 0.18 μm CMOS technology and consists of 168,000 hardware logic gates and 1.6 kilobytes of internal SRAM buffer.

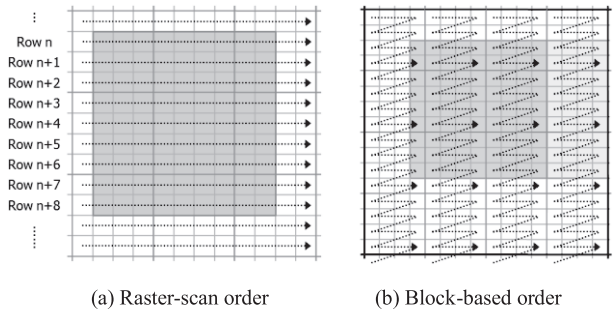
The rest of this paper is organized as follows. Section 2 presents the data mapping scheme used to reduce the overhead involved in accessing an SDRAM during H.264/AVC motion compensation. Section 3 proposes a novel cache organization that improves the efficiency of motion compensation, and Sect. 4 presents modifications to an existing recompression technique optimized for H.264/AVC motion compensation. Section 5 evaluates the proposed techniques and Sect. 6 presents the implementation of the proposed H.264/AVC decoder. Conclusions are presented in Sect. 7.

2. Reference Frame Storage Pattern

An SDRAM is a widely used type of memory to store the reference frames used by an H.264/AVC decoder. Access to reference frames from an SDRAM requires activation of the addressed row. If two consecutive pieces of data are stored in the same row, only the action of accessing the first piece involves row activation latency. Therefore, memory access time can be reduced if consecutive memory accesses occur in the same row. An efficient distribution scheme to reduce row activation time has been proposed in [7]. This section briefly explains the background of a previous frame distribution scheme and proposes a modification that makes the distribution suitable for use in the cached architecture proposed in Sect. 3 as well as the data recompression method proposed in Sect. 4.

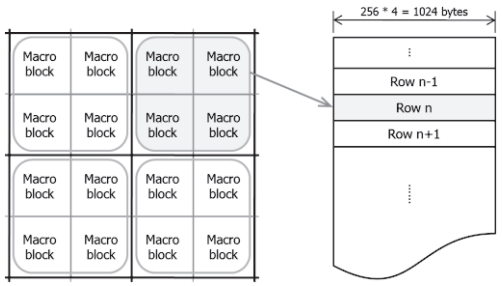
For the generation of the pixel at the half-pel position, motion compensation requires 6-tap filtering operation which needs 2 pixels in the left and 3 pixels in the right of the current pixel at an integer position. As a result, motion compensation of a 4x4 block in an H.264/AVC decoder may need to access at most a 9x9 block from a reference frame [17]. Figure 1 (a) shows the conventional data distribution for a single 16x16 macroblock. In the conventional data distribution, image pixels are stored along the raster-scan order. Each small square in the figure represents a single image pixel and arrows in the figure show the direction along which addresses of the pixels increase. This raster-scan order distribution is not an efficient method for an SDRAM access because numerous row changes are required.

In order to reduce the number of row activations, the memory controller in [7] stores the reference frame in an SDRAM in such a way that the entire macroblock is stored in the same row. This data distribution method is presented in Fig. 2. For a 32-bit SDRAM where each row can store 256 words, the storage capacity of each row is 1024 bytes. Thus, four adjacent 2x2 macroblocks can be stored in the same row, as shown in Fig. 2, so that all the data in these macroblocks can be accessed with only a single row activation event.



(a) Raster-scan order (b) Block-based order

Fig. 1 Reference frame.



(a) Reference frame memory map (b) SDRAM memory space

Fig. 2 Memory distribution for a reference frame with 4 adjacent macroblocks stored in the same row of an SDRAM.

If, however, a 9x9 block overlaps with the boundary of the 2x2 macroblocks stored in the same row, then part of the 9x9 block are stored on a different row and multiple rows have to be activated to access this 9x9 block. A bank interleaving scheme can be adopted to avoid the additional row activations caused by such a situation [7]. The bank interleaving scheme stores the adjacent 2x2 macroblocks in the same row but in different banks. In an SDRAM, data stored in different banks but in the same row of each bank can be accessed without incurring the penalties for activating additional rows. Thus, all the data in 4x4 macroblocks can be stored in the same row and accessed without additional row activation.

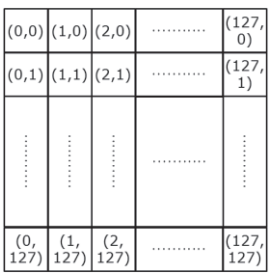
This paper adopts the frame memory distribution with bank interleaving scheme presented in [7]. In addition, a new data distribution scheme for data within a single macroblock is proposed as indicated by the arrows, which show the direction along which column addresses increase, in Fig. 1 (b). This method of data distribution simplifies the process of addressing a 4x4 block because the address is simply increased incrementally with each successive access to subsequent data within the block. This is an efficient addressing scheme for use in an H.264/AVC decoder that includes many operations using a 4x4 block as the basic processing unit. Examples of such operations include entropy coding, transform, deblocking filtering, and intra prediction. More importantly, the caching and recompression schemes

proposed in the later sections of this paper also make use of 4x4 blocks as the basic data transfer unit, so the proposed addressing scheme is well suited for these schemes.

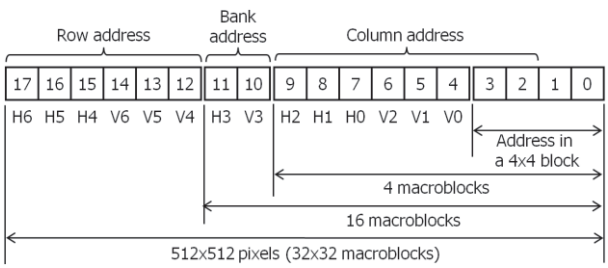
Figure 3 shows how addresses are identified when a reference frame is accessed using 4x4 blocks as shown in Fig. 1 (b). In Fig. 3 (a), each square represents a 4x4 block. Assuming that the image size is 512x512, the number of 4x4 blocks in the image is 128x128. The pair of numbers in the parenthesis in each square represents the vertical and horizontal indices of each block, respectively. Figure 3 (b) shows the partitioning of the address bits for the proposed data distribution. Assigning an address to a single frame for this image requires 18 address bits. Let A[m:n] denote the portion of the address from the mth to nth bits and A[k] denote the kth address bit. Then, A[3:0] represents the address of a piece of data in a single 4x4 block and A[17:4] represents the index of the 4x4 block.

Both the horizontal and vertical indices of a 4x4 block require 7 bits each because a frame consists of 128x128 blocks. A[17:15], A[11] and A[9:7] correspond to the horizontal index, which is denoted by H7, H6, ..., H0 in Fig. 3 (b), whereas A[14:12], A[10], and A[6:4] correspond to the vertical index, which is denoted by V7, V6, ..., V0. A single row within an SDRAM chip stores 4 macroblocks (64 4x4 blocks) consisting of 256 words (assuming 1 word = 4 bytes). Thus, A[9:2] represents the column address within a single row and A[1:0] represents the byte address within a word. A[11:10] represents a bank address and A[17:12] represents a row address. With 4 banks, 16 macroblocks can be stored in the same row. For Chroma (chrominance) components, the amount of data required is one quarter of that required for Luma (luminance) components. Thus, the number of address bits is reduced by two bits as shown in Fig. 3 (c).

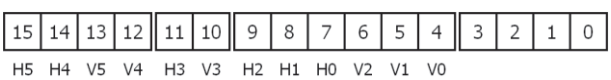
The address format shown in Fig. 3 (b) is designed to access a single frame of an image. In general, an H.264/AVC decoder requires multiple reference frames. Thus, an additional field is necessary to indicate the reference frame number. In general, the position of the additional field is at the most significant bits, and the number of required bits depends on the number of reference frames.



(a) Reference frame decomposed into 4x4 blocks



(b) Luma address format



(c) Chroma address format

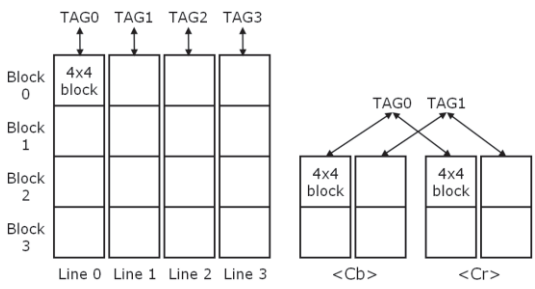
Fig. 3 Address format of a reference frame.

3. Cache Organization for Reference Frame Access

One way to reduce the amount of memory access required for motion compensation is to use a small on-chip cache that stores data for one block temporarily and reuses the data for adjacent blocks. The cache proposed in previous work such as [6] employs a line-based cache organization in which image data in the same row are stored in the same cache line. As an H.264 decoder accesses block-by-block, the line-based organization often requires excessive row activations in SDRAM access, thereby, increasing memory access time. Furthermore, the line-based cache organization is not suitable for the block-based data distribution presented in the previous section. This section proposes a new cache organization that avoids abundant SDRAM access and is suitable for the data distribution proposed in Sect. 2.

3.1 Cache Organization

Figure 4 shows the cache organization proposed in this section. For Luma data, the cache size is 16x16 bytes, which can store a single macroblock. For Chroma data, the cache size is 8x8 bytes for each of the Cb and Cr components. The Luma cache consists of four lines, each of which stores four 4x4 blocks. A 4x4 block is the minimum transfer unit for each data access to this cache. It is not necessary to read (or write) the entire four 4x4 blocks in a single cache line. A single tag is associated with each cache line, indicating the address of the data stored in that line. All four 4x4 blocks in a single line are not always required to store valid data;



(a) Luma component (b) Chroma components

Fig. 4 Cache memory organization with associated tags.

therefore, additional information noting the valid block in a cache line is also given in the tag. This information consists of the starting index and valid length. The details of this information are explained in the next paragraph. The Chroma cache consists of two cache lines, each of which stores two 4x4 blocks. As both Cb and Cr components always have a common motion vector, tags are shared by both components.

Figure 5 shows how data are mapped from external memory to the cache. Only Luma data are considered in this figure in order to simplify the example. Each square represents a 4x4 block that is mapped to a single 4x4 block in the cache. The numbers in each block denote the horizontal and vertical indices, respectively. All blocks in the first column (i.e. data for the four leftmost pixels) are mapped into cache line 0. Blocks in the next column to the right are mapped into cache line 1 whereas blocks from the third and fourth columns are mapped into cache lines 2 and 3, respectively. Blocks in the fifth column are once again mapped into cache line 0. Figure 5 also shows the position of a given 4x4 block to be stored within a cache line. All blocks in the first row are stored in the first block (block 0) of a cache line while the blocks in the second row are stored next (block 1), and so on. Note that each 4x4 block from an image has a fixed position within the cache.

Figure 6 shows the format for a cache tag. An image size of 512x512 is used once again for this example although

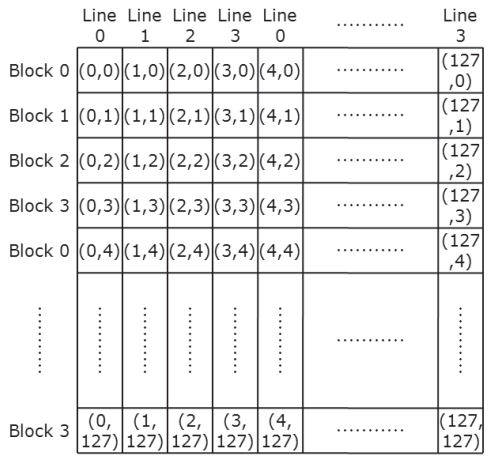
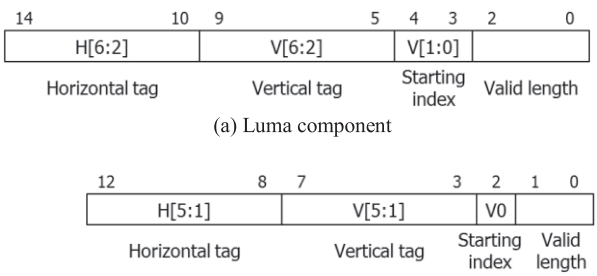


Fig. 5 Mapping of a frame into a cache line.



(a) Luma component

(b) Chroma component

Fig. 6 Cache tag format.

the proposed scheme is applicable to an image of an arbitrary size. Recall that H[6:0] and V[6:0] in Fig. 3 (b) represent the horizontal and vertical indices, respectively, of a 4x4 block. Among the seven bits of the horizontal index, the least significant two bits, H[1:0], correspond to the cache line index. Thus, the remaining bits, H[6:2], form part of the cache tag (horizontal tag) that is used to detect a cache hit. A single cache line contains four 4x4 blocks. Thus, V[1:0] indicates the position of a 4x4 block in a single cache line and the remaining bits for the vertical index, V[6:2], are used as a tag (vertical tag). As shown in Fig. 6, the horizontal tag, H[6:2], corresponds to bits 14 down to 10 of the tag whereas the vertical tag corresponds to bits 9 down to 5 of the tag.

A conventional direct-mapped cache fetches all data in the same cache block together even though only a part of the data is needed. This often results in the access of unnecessary data and wastes external memory bandwidth. To avoid such bandwidth waste, this paper proposes a cache organization designed to avoid the storage of unnecessary data in a cache line. To achieve this, the cache tag stores additional information to indicate the valid part of the cache line - the starting index and the length of the valid block in a cache line. The starting index represents the address of the first valid block in a cache line whereas the valid length represents the number of valid blocks in the cache line. All valid blocks are stored contiguously in a cache line from the starting index. If the sum of the starting index and the valid length is larger than the cache line size, blocks are stored contiguously from the starting index to the bottom of the cache line and also stored from the top of the same cache line. Bits 4 and 3 in the tag indicate the starting index while the least significant three bits indicate the valid length.

For Chroma cache, each cache line stores only two 4x4 blocks. Thus, only one bit is necessary for the starting index and two bits for the valid length. The tag format for Chroma cache is shown in Fig. 6 (b). If multiple reference frames are required, the additional address bits indicating the reference number need to be included in the most significant bits of the cache tag.

Using the above cache organization, the number of tag matching operations is minimized. The mapping of a 4x4 block into a single cache line allows just a single tag matching operation to be performed to access the entire block. Thus, this block-oriented cache line organization is effective for H.264/AVC motion compensation, which often uses a 4x4 block as the basic transfer unit. The sharing of a tag with four 4x4 blocks further reduces tag matching operations because four 4x4 blocks can be accessed with a single tag matching operation. This sharing is possible because H.264/AVC decoding involves a high probability of accessing adjacent 4x4 blocks together. A drawback of this block-based cache organization is that more data are often accessed than necessary. For example, if a 9x9 block of data is necessary, a 12x12 block of data is accessed. However, the increase in data access is not significant because a 32-bit wide SDRAM requires at least 4 bytes of data to be transmitted. Therefore, it is impossible to access only 9 bytes

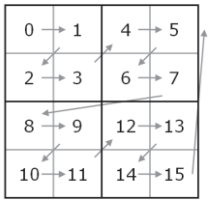


Fig. 7 The processing order for motion compensation of a 16x16 macroblock.

of horizontally-adjacent data. In the vertical direction, the memory access time for the additional data is not significantly increased because the additional data accesses are in the same row when using the data distribution described in Fig. 1 (b). Furthermore, the additional fetched data are often used by the next block.

Figure 7 shows the processing order of data for motion compensation. This figure shows that the four 4x4 blocks in the upper left are grouped and processed first, then the four 4x4 blocks in the upper right are processed, and so on. Therefore, it is desirable to have the cache store all the necessary data for processing the four 4x4 blocks in the same group. As a 4x4 block requires at most 9x9 pixels, four 4x4 blocks may require 13x13 pixels if they have identical sub-pel motion vectors. Thus, a cache size of 16x16 is chosen, providing a slightly larger size than the 13x13 requirement. Experimental results show that this size achieves a reasonably large hit ratio. The Chroma cache is designed to be one fourth the size of the Luma cache as the Chroma data is one fourth as many as Luma data.

Figure 8 shows an example that illustrates how Luma data blocks are mapped from external memory to cache memory with the proposed organization. Figure 8 (a) shows a reference image with two shaded squares in the figure, which are to be stored in the cache. The number at the top of the image represents the horizontal index of a 4x4 block whereas the number at the left shows the vertical index. For the upper shaded square, which is 12x12 in pixels in size, the address of the top left pixel is 0x00240 while the lower shaded square, also 12x12 pixels in size, has an address for its top left pixel of 0x002e0. In the figure, the two pixels are indicated by arrows. Suppose that the upper shaded square is fetched into the cache first. Figure 8 (b) shows the cache contents when this shaded square is stored in the cache. The first three cache lines are filled with three 4x4 blocks each. The numbers in each block represent the corresponding indices for each block. Figure 8 (c) shows the contents of the cache tags. For cache tag 0, H[6:2] and V[6:2] are stored as the horizontal and vertical tags, respectively. The starting index is 0 because the valid block begins with the first block. The valid length is 3 because three blocks are valid. The second and third line tags store the same contents as the first line while the fourth cache line remains in the empty state.

Suppose that the lower shaded square is fetched next. This square overlaps partly with the first square, specifically the block indexed (5,6) and (6,6), and it is not necessary

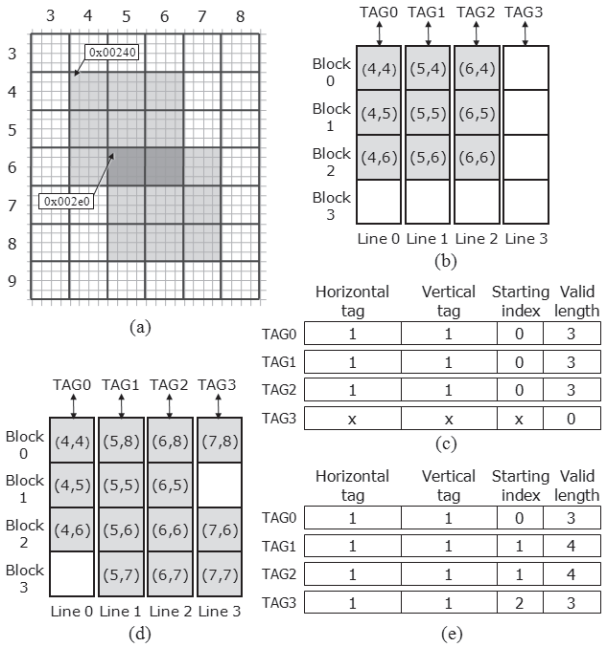


Fig. 8 Cache mapping example; (a) reference image; (b) cache contents after the first data fetch; (c) cache tag contents after the first data fetch; (d) cache contents after the second data fetch; (e) cache tag contents after the second data fetch.

to fetch this overlapped data again. Figure 8 (d) shows the contents of the cache after the second square is fetched. For cache line 1, two new blocks, (5,7) and (5,8), are loaded into the last and first block positions, respectively. Thus, all four blocks in this cache line are valid, and the valid length is 4. Note that the four 4x4 blocks stored in this cache line are (5,5), (5,6), (5,7) and (5,8) and that the first block, (5,5), is stored in the second block position of this cache line. Therefore, the starting index is 1. Cache line 2 also stores two new blocks and the cache tag information is the same as that of cache line 1. For cache line 3, three blocks (7,6), (7,7) and (7,8) are stored as shown in Fig. 8 (d) and the corresponding cache tag is shown in Fig. 8 (e).

The horizontal and vertical tags point to the valid data with the lowest address in a cache line and data in the same cache line may have different values for the vertical tag. For example, blocks indexed (7,6), (7,7) and (7,8) in Fig. 8 (d) are stored in the same cache line. Note that the vertical tag for these three blocks is 1. On the other hand, if only block (7,8) is stored, then its vertical tag would be 2. This implies that the same block may be associated with two different tags. Therefore, to check a cache hit, it is necessary to compare not only the cache tag but also the starting index with the valid length.

3.2 Cache Prefetching Scheme

This subsection presents the cache prefetching scheme efficient for data access in motion compensation. When a motion vector points to a sub-pel position, 9x9 data are necessary from a reference frame memory. In the proposed cache,

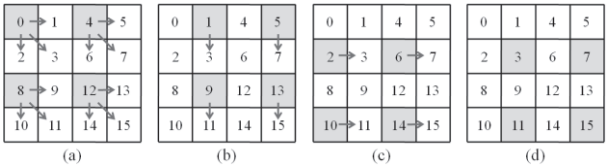


Fig. 9 Prefetching direction represented by arrows; (a) current block: 0, 4, 8, and 12; (b) current block: 1, 5, 9, and 13; (c) current block: 2, 6, 10, and 14; (c) current block: 3, 7, 11, and 15.

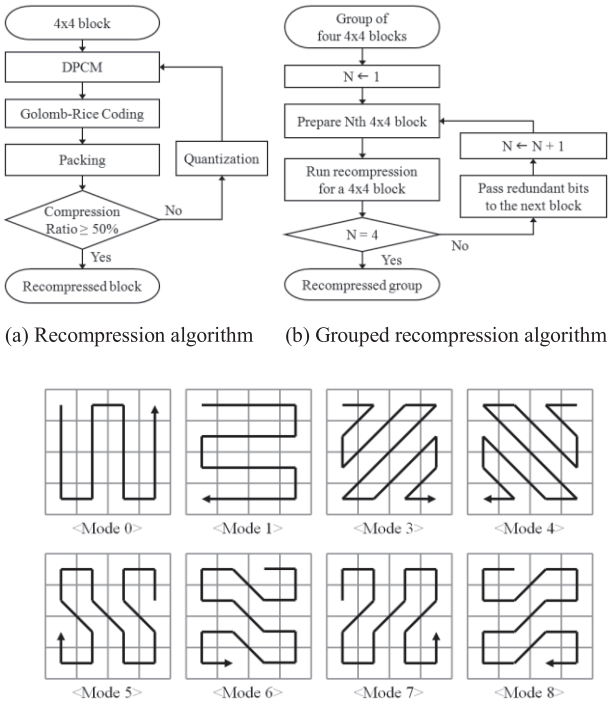
a 12x12 block of data is fetched from the reference memory as this cache always fetches a 4x4 block at a time. If the prefetching option is activated, additional 4x4 blocks to the right and/or lower side of the current block are also fetched.

These prefetched blocks are selected according to the position of the current 4x4 block in a macroblock, as shown in Fig. 9. In this figure, a large square represents a 16x16 macroblock and small squares represent 4x4 blocks. The number inside each small square shows the processing order of the 4x4 blocks. Figure 9 (a) shows the case when one of the blocks numbered 0, 4, 8 and 12 are accessed. In this case, the blocks to the right (1, 5, 9, 13) are to be processed next. The blocks to the lower side (2, 6, 10, 14) are to be processed next followed by the blocks to the lower right (3, 7, 11, 15). Therefore, all the blocks to the right, lower and lower right sides are prefetched because these blocks are likely to be used in the near future. This prefetching is shown by the arrows in the figure. Figure 9 (b) shows the case when blocks 1, 5, 9 and 13 are currently fetched. In this case, the blocks to the lower side (3, 7, 11, 15) are prefetched because they are likely to be processed next. Likewise, blocks 3, 7, 11, and 15 are prefetched with blocks 2, 6, 10 and 14 being currently processed (Fig. 9 (c)). For blocks 3, 7, 11 and 15, no data are prefetched (Fig. 9 (d)).

For the Chroma component, the maximum size of the block of data to be fetched is 3x3. Note that the data transfer unit (i.e., cache block size) is 4x4 so each cache block access transfers a larger block than necessary, resulting in the same effect as prefetching. Additional prefetching leads to excessive cache replacement. Thus, the prefetching scheme is not applied to the Chroma cache.

4. Frame Memory Recompression

Frame memory recompression is a technique used to reduce frame memory size and bandwidth requirements by compressing the data to be stored in frame memory. When a reference frame is transferred from the video processor to off-chip memory the recompression encoder compresses the data. When the video processor requires access to the stored reference frame, the recompression decoder decompresses and restores the original data as it is fetched from off-chip memory. Several recompression algorithms have been developed to reduce the size and bandwidth requirements of frame memory [10]–[16]. Among these, Lee in [16] proposes an algorithm in which an input image is decomposed into 4x4 blocks. As a 4x4 block is used as the basic transfer unit for data distribution and for the cache organization proposed in this paper, this algorithm is most suitable for the H.264/AVC decoder presented here. This section briefly introduces the recompression algorithm proposed in [16] and then proposes an efficient integration of the recompression algorithm into the cache architecture presented in the previous section.



(a) Recompression algorithm (b) Grouped recompression algorithm

(c) The eight scan modes used by DPCM in the recompression algorithm

Fig. 10 Recompression algorithm integrated into the H.264/AVC decoder.

4.1 Recompression Algorithm

The recompression algorithm in [16] decomposes an image frame into 4x4 blocks and then compresses each block independently, achieving a 50% compression ratio. The flowchart in Fig. 10 (a) gives an outline of the recompression algorithm. DPCM is performed along a predefined scan order for each 4x4 block. Eight DPCM scans, with different scan orders as shown in Fig. 10 (c), are performed and the best scan order is selected by comparing the results from all eight scan orders. Then, the DPCM results are further compressed using Golomb-Rice coding. If the compression ratio does not reach the target of 50%, the 4x4 block pixel data are quantized by a 1-bit right shift, processed by DPCM and entropy coding again. This process of quantization, DPCM and entropy coding are repeated until the target compression ratio is achieved.

4.2 Integration of the Recompression Algorithm

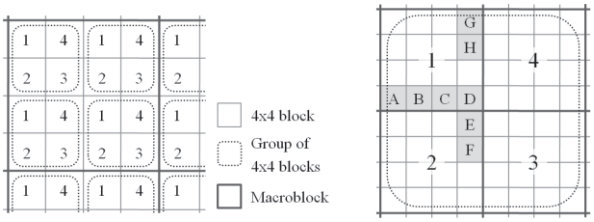
In encoded stream by DPCM, the number of the bits en-

coding the first pixel is often much larger than that of any other pixel because the first pixel is the only pixel of which its value itself is encoded. For all the other pixels, the difference of adjacent pixels along the scan order shown in Fig. 10 (c) is encoded. Note that the difference of pixels is often much smaller than the value itself of a pixel. One way to improve the compression efficiency is that the first pixel is subtracted from another pixel in and the difference is encoded for the first pixel, too. As the first pixel cannot be subtracted from any pixel in the same block, it should be subtracted from a pixel in another block. Therefore, this differential coding of the first pixel requires multiple blocks to be compressed together so that a pixel from another block is available for an encoder as well as a decoder. The simultaneous compression of multiple blocks gives another advantage for improving compression efficiency. If a relatively simple block among multiple blocks does not require all the assigned bits, then the surplus bits for this block can be used by another block which may require bits larger than the preassigned bits. Although the compression of multiple blocks improves compression efficiency, the data transfer unit increases, thereby, also increasing the memory access time.

The use of cache memory can avoid the increase of memory access time as it holds multiple blocks to be encoded together in the cache. This paper proposes the recompression algorithm that compresses four adjacent 4x4 blocks together. As the cache in this research can store multiple 4x4 blocks, it can be used to recompress multiple 4x4 blocks together. Without the presence of the cache, additional storage space may be required. The modified recompression that simultaneously recompresses multiple 4x4 blocks is referred to as grouped recompression hereafter whereas the original recompression algorithm is called non-grouped recompression. One side-effect of the grouped recompression algorithm is that it requires a 8x8 block to be stored in cache simultaneously, and thereby, increasing the cache size.

Figure 11 (a) shows how the adjacent 4x4 blocks are used to improve the compression efficiency. In this figure, the small squares represent 4x4 blocks and the thick and large square represents a macroblock. The number inside each small square shows the processing order when four 4x4 blocks are compressed by the recompression encoder. Figure 10 (b) shows a flowchart for the grouped recompression algorithm. The upper-left 4x4 block (numbered 1) is compressed first in exactly the same manner as the original recompression algorithm. If the length of the compressed block is less than 64 bits, the remaining bits are used for the compression of the next block (numbered 2). In such cases the second block can achieve a compression ratio of less than 50%, resulting in improved quality of the compressed image. Bits remaining after compression of the second block can also be used for the third block and, likewise, the last block can use bits remaining after compression of the third block.

The modified recompression algorithm takes advantage of another aspect of having neighboring blocks in the cache. In the original recompression algorithm, the first pixel requires (8-QP) bits for a given quantization parameter, QP. This is a large number of bits for the first pixel compared to the number of bits assigned for the remaining pixels that have already been compressed by DPCM. The modified recompression algorithm makes the neighboring pixels available for the second, third and fourth 4x4 blocks so the first pixels of these blocks can be compressed by DPCM using adjacent pixels from the previous blocks. Figure 11 (b) shows the pixels used for the DPCM operations on the second, third and fourth blocks. In this figure, each small square represents a pixel. Suppose that the block numbered 2 is to be processed and that the best scan mode for this block is mode 1, which begins data scanning from the upper-leftmost pixel of the block (see Fig. 10). Then, the adjacent pixel, labeled A in Fig. 11 (b), is used to perform DPCM and obtain the first codeword for the second block. If other scan modes are selected, the first pixel of the scan mode is subtracted from the closest pixel in the first block. Table 1 shows the closest pixel from which the first pixel is subtracted to obtain the codeword. For example, when the best scan mode of block 2 is found to be mode 8, the first pixel of block 2 is subtracted from pixel B for DPCM.



(a) Processing order (b) Pixel selection for DPCM

Fig. 11 Processing order and pixel selection for DPCM in the grouped recompression algorithm.

Table 1 The pixel used for DPCM of the first pixel of a 4x4 block.

	Block 2				Block 3		Block 4	
Best scan mode	0, 1, 3, 7	8	6	4, 5	0, 1, 3, 4, 6, 8	5, 7	0, 1, 3, 4, 6, 8	5, 7
Pixel for DPCM	A	B	C	D	E	F	G	H

5. Experimental Results

Table 2 shows the memory access time for motion compensation with various data access schemes. A CIF-sized Stefan sequence with 300 frames is encoded as the baseline profile and used as the test video. The number of reference frames is 3, QP is 28, I frame period is 10, and the full search algorithm with the search range of ±16 is used for motion estimation. The specification of the used SDRAM is given in Table 3. The "raster-scan" column shows the results using the conventional data distribution with a raster scan order. The "block-based" column shows the results from using the proposed data distribution presented in Sect. 2, and the "cache" column shows the results from using a cache proposed in Sect. 3. The cache size is 16x16 for Luma data and

Table 2 Comparison of memory access time.

	Raster-scan	Block-based	Cache	Prefetch	Recompress
Amount of data	1.00	1.33	0.45	0.46	0.23
Number of row activations	1.00	0.17	0.11	0.06	0.05
Time	1.00	0.45	0.19	0.16	0.10

Table 3 SDRAM parameters.

CAS delay	3 cycles
Page size	1,204 bytes
Data bus width	32 bits
Number of banks	4
Minimum RAS-to-CAS delay (t_{RCD})	18 ns
Minimum row active time (t_{RAS})	42 ns
Minimum precharge time (t_{RP})	18 ns
Refresh period	8k / 64 ms

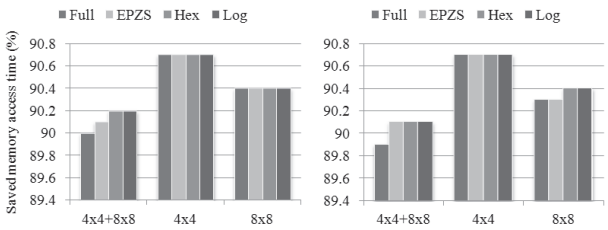
8x8 for Chroma data. The "prefetch" column shows the results from using the prefetching scheme and the last column denoted by "recompress" shows the results from using the non-grouped recompression scheme presented in Sect. 4.1. The "amount of data" row shows the total number of bytes transferred from external memory. The "number of row activations" shows the number of access events that require a row change whereas the "time" shows the total memory access time, taking into consideration the initial latency from row activation and time required for column addressing together with the actual data transfer time. The latency for an addressing operation is modeled to be six cycles which are normal in SDRAM access. Results using the raster-scan order for data distribution are used as a reference, and the results for the other scenarios are represented as their ratio relative to the reference result in this table.

The block-based data distribution increases the amount of data accessed from memory because this method may access unnecessary data. However, the number of row activations significantly decreases using the block distribution, resulting in memory access time of less than a half of that required by the raster-scan distribution. With the presence of cache, both the amount of data and the number of row activations are significantly reduced. As a result, the memory access time is reduced only to 19% of the conventional raster-scan distribution. With the prefetching scheme, the cache hit ratio increases up to 80% and the memory access time is further reduced to 16%. Recompression further contributes to an additional decrease in memory access time. When using all the proposed schemes combined, memory access time is reduced to only 10% of the raster scan method access time.

Table 4 shows experimental results using three additional test sequences: Akiyo, Foreman and Mobile. These sequences are also encoded as the baseline profile. The numbers given in the table show the percent reduction of the proposed scheme compared to the conventional raster-scan order of data distribution. On average, the amount of data accessed is reduced by 77% while the number of row activations is reduced by 95%. The overall result is a reduction in memory access time of 90%.

Table 4 Reduction in memory access by combining all three techniques.

	Cache hit ratio	Amount of data (saved)	Number of row activations (saved)	Time (saved)
Akiyo	78%	77%	95%	92%
Foreman	83%	77%	94%	90%
Mobile	81%	75%	94%	89%
Stefan	82%	77%	95%	90%
Average	81%	77%	95%	90%



(a) 3 reference frames (b) 5 reference frames

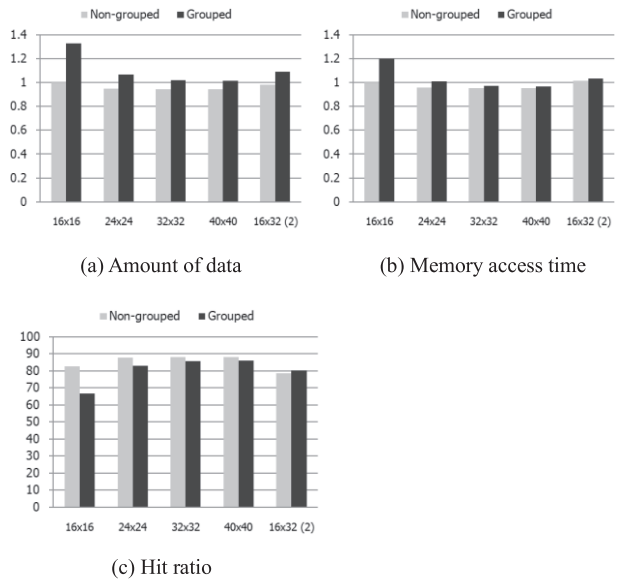
Fig. 12 Comparison of various motion estimation algorithms.

In order to evaluate effect of motion estimation algorithm, the test video is encoded with three additional algorithms in addition to the full search algorithm used in the previous experiment. Figure 12 shows the results of various motion estimation algorithms. The graph labeled "Full" represent the full search algorithm whereas the graphs labeled "EPZS", "Hex" and "Log" represent the enhanced predictive zonal search, hexagon-based search, and logarithmic search algorithms, respectively. To evaluate the effect of the block size, the experiments enforce constraints on the block size of motion estimation. The graph labeled "4x4" in Fig. 12 represents the case when motion estimation is performed for only the 4x4 block size. On the other hand, "8x8" represents the motion estimation with its block size larger than or equal to 8x8 (i.e., 8x8, 8x16, 16x8 and 16x16). The graph "4x4+8x8" represents the motion estimation of the block size, 4x4, 8x8, 16x8, 8x16, and 16x16. As shown in these results, the effect of the motion estimation algorithm on the result is ignorable. The search range and I-frame period are also adjusted to find their effect, but experimental results also show that their effects are negligible.

In Table 5, the proposed scheme with the original recompression is compared with two other approaches, [5] and [6], which also reduce memory access time for H.264/AVC motion compensation. Results from [6] are given in Table 5. However, [5] only presents results on the amount of accessed data and does not present results on memory access time. The memory access time for [5] is derived with the assumption that the number of row activations decreases proportionally with the reduction in the amount of

Table 5 Comparison of memory access time with previous work.

	Register file [5]	Circular cache [6]	Proposed	PSNR drop	Proposed w/o recompression
Akiyo	79%	77%	92%	1.47 dB	87%
Foreman	65%	74%	90%	1.49 dB	83%
Mobile	63%	74%	89%	1.98 dB	82%
Stefan	62%	74%	90%	1.88 dB	84%
Average	67%	74%	90%	1.71 dB	84%



(a) Amount of data

(b) Memory access time

(c) Hit ratio

Fig. 13 Comparison of various cache organizations for the grouped recompression algorithm.

accessed data. The reduction in memory access time for the proposed method is, on average, about 16% better than that for [6] and 23% better than that for [5]. The next column shows the PSNR drop caused by recompression. The average PSNR drop is 1.71 dB. If this PSNR drop is too large, the recompression algorithm can be turned off and the memory access time without recompression is shown in the last column. On average, the proposed scheme without recompression still achieves 17% and 10% better results than those obtained in [5] and [6], respectively.

In order find the appropriate cache size, experiments with various cache sizes are performed. Furthermore, the performance is also compared with two cache organizations: the direct-mapped cache that allows a data to be mapped into only a predefined location in a cache and 2-way set associative cache that allows a data to be mapped to two predefined locations [18]. Figure 13 (a), (b) and (c) show the amount of transferred data, the memory access time, and the cache hit ratio, respectively, with various cache organizations. The Stefan sequence is used for this experiment. The horizontal axis represents cache sizes and organizations. Direct-mapped caches with four different sizes (from 16x16 to 40x40) are used to obtain the left four graphs while 2-way set-associative cache is used for the rightmost graph (denoted by 16x32(2)). Results from the original non-grouped recompression with a 16x16 direct-mapped cache are used as the reference and the results in this figure are shown as relative values. With the 16x16 direct-mapped cache, the grouped recompression algorithm increases the amount of transferred data by 32.9% and the memory access time by 20.0%. This shows that a cache size of 16x16 is not large enough for the grouped recompression algorithm. With the 24x24 and 32x32 direct-mapped caches, both the increases in the amount of data and memory access time by the grouped recompression algorithm are significantly reduced. The difference between 32x32 and 40x40 direct-mapped caches is very small. A 16x32 set-associative cache requires longer SDRAM access time than 24x24 or 32x32 direct-mapped caches. Figure 13 (c) shows that the 32x32 direct-mapped cache gives the second best hit ratio with only a marginal difference from the best 40x40 direct-mapped cache. Based on these observations, the 32x32 direct-mapped cache is chosen as the final cache organization for the grouped recompression algorithm.

Table 6 Performance of the grouped recompression algorithm.

	Amount of data (%)	Memory access time (%)	Hit ratio (%)	PSNR improvement (dB)
Akiyo	1.89	-2.21	-1.78	0.272
Foreman	8.04	2.38	-2.48	0.273
Mobile	10.54	3.69	-3.77	0.568
Stefan	7.82	1.96	-2.41	0.370
Average	7.07	1.46	-2.61	0.371

With 32x32 cache, the grouped recompression reduces the number of row activations from 1,154,211 (non-grouped recompression) to 1,109,302 averaged over the four test sequences. The amount of data is reduced from 6,574,071 words to 6,099,378 words on average and the memory access time is reduced from an average of 13,229,887 cycles to 13,024,644 cycles. The PSNRs of the decoded video is improved from 36.600 to 36.971 on average. Table 6 shows the ratios. As shown in the table, the amount of data and the memory access time increase by averages of 7.07% and 1.46%, respectively, whereas the cache hit ratio decreases by an average of 2.61%. PSNR increases by an average of 0.371 dB, which is a significant improvement. This result shows that grouping significantly improves image quality while causing only a slight increase in memory access time and 768 bytes of additional cache buffer.

This paper employs the DPCM algorithm for frame memory recompression. Other algorithm can also be integrated into the H.264 decoder with the cache presented in this paper. Although the grouped recompression algorithm proposed in 4.2 is aimed for the DPCM algorithm, the cache can also be used for other recompression algorithm because, in general, the use of cache can increase the block size of a recompression algorithm, which often improves the compression efficiency.

6. H.264/AVC Decoder Implementation

The proposed memory access scheme is integrated into an H.264/AVC decoder, which is represented by the block diagram in Fig. 14. The motion compensation (MC), deblocking filter (DB), intra prediction (IP), inverse quantization/inverse transform (IQ/IT) and variable length decoder (VLD) are implemented in hardware and the remaining computation is processed by the ARM7TDMI processor. The recompression module is inserted between the DMA controller (DMAC) and SDRAM controller to perform compression/decompression on data moving to and from the external memory.

Figure 15 shows the layout and the photograph of the H.264/AVC decoder chip from Fig. 14. This decoder implements all the schemes except for the grouped recompression algorithm presented in Sect. 4.2. The die area of the H.264/AVC decoder chip is 4.5 mm x 4.5 mm and is fabricated using the Dongbu 1P6M 0.18 μm CMOS process. The chip is composed of 168,000 hardware logic gates and 1.6 kilobytes of internal SRAM buffer. Table 7 shows the gate count and SRAM buffer size of each hardware module. About 10 K gates of logic circuits and 384 bytes of SRAM are added for cache implementation whereas 11 K gates of logic circuits and 128 bytes of SRAM are added for the implementation of the recompression algorithm. Although the grouped recompression algorithm is not integrated into the chip, it is implemented and verified with a Verilog program model. The size of logic circuits added for the grouped recompression is 3.6 K gates.

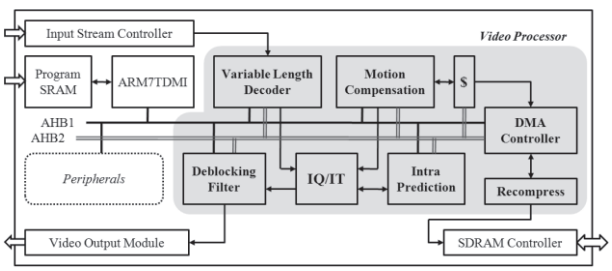


Fig. 14 Block diagram of the H.264/AVC decoder.

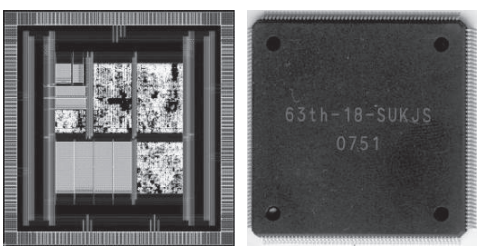


Fig. 15 H.264/AVC decoder chip layout and photograph.

Table 7 Gate count and internal buffer size of H.264/AVC decoder modules.

Module	VLD	IP	IQ/IT	MC	Cache	DB	DMAC	Recomp
Gate	26K	19K	31K	41K	10K	22K	8K	11K
SRAM (bytes)	-	-	512	128	384	512	-	128

7. Conclusion

This paper proposes an H.264/AVC decoder that reduces the external memory access time incurred during motion compensation by combining three techniques: efficient distribution of reference frame data, on-chip cache memory, and frame memory recompression. The proposed decoder is successful in reducing the access time to external memory with a 16x16 cache by 90%, which is 16% better than the results seen in previous work. If the frame memory recompression algorithm is excluded from the decoder design, the reduction in external memory access time is 84%, which is 10% better than the results from previous work. The recompression algorithm may decrease the image quality and should only be used when slight degradation to image quality is acceptable. The grouped recompression algorithm can be used to reduce the image quality degradation caused by recompression, as it improves the PSNR value by 0.371 dB.

Acknowledgement

This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST).

References

[1] Joint Video Team, Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification, ITU-T Rec. H.264 and ISO/IEC 14496-10 AVC, May 2003.
[2] I.E.G. Richardson, H.264 and MPEG-4 Video Compression, John Wiley & Sons, 2003.
[3] R.G. Wang, J.T. Li, and C. Huang, "Motion compensation memory access optimization strategies for H.264/AVC decoder," Proc. Int. Conf. on Acoustics, Speech, and Signal Processing, vol.5, pp.97–100, March 2005.
[4] T.-C. Chen, Y.-W. Huang, and L.-G. Chen, "Fully utilized and reusable architecture for fractional motion estimation of H.264/AVC," Proc. Int. Conf. on Acoustics, Speech, and Signal Processing, vol.5, pp.9–12, May 2004.
[5] C.-Y. Tsai, T.-C. Chen, T.-W. Chen, and L.-G. Chen, "Bandwidth optimized motion compensation hardware design for H.264/AVC HDTV decoder," Proc. Int. Symp. on Circuits and Systems, vol.2, pp.273–276, Aug. 2005.
[6] J.-H. Kim, G.-H. Hyun, and H.-J. Lee, "Cache organizations for H.264/AVC motion compensation," Proc. Int. Conf. on Embedded and Real-Time Computing Systems and Applications, pp.534–541, Aug. 2007.
[7] H. Kim and I.-C. Park, "High-performance and low-power memory-interface architecture for video processing applications," IEEE Trans. Circuits Syst. Video Technol., vol.11, no.11, pp.1160–1170, Nov. 2001.
[8] J. Zhu, L. Hou, W. Wu, R. Wang, C. Huang, and J.T. Li, "High performance synchronous DRAMs controller in H.264 HDTV decoder," Proc. Int. Conf. on Solid-State and Integrated Circuits Technology, vol.3, pp.1621–1624, Oct. 2004.

[9] G.-S. Yu and T.S. Chang, "Optimal data mapping for motion compensation in H.264 video decoding," Proc. IEEE Workshop on Signal Processing Systems, pp.505–508, Oct. 2007.
[10] V.G. Moshnyaga, "Reduction of memory accesses in motion estimation by block-data reuse," Proc. Int. Conf. on Acoustics, Speech, and Signal Processing, vol.3, pp.3128–3131, May 2002.
[11] T.Y. Lee, "A new frame recompression algorithm and its hardware design for MPEG-2 video decoders," IEEE Trans. Circuits Syst. Video Technol., vol.13, no.6, pp.529–534, June 2003.
[12] R. Dugad and N. Ahuja, "A fast scheme for image size change in the compressed domain," IEEE Trans. Circuits Syst. Video Technol., vol.11, no.4, pp.461–474, April 2001.
[13] D. Pau and R. Sannino, "MPEG-2 decoding with a reduced RAM requisite by ADPCM recompression before storing MPEG decompressed data," U.S. patent 5838597, Nov. 1998.
[14] R. Bruni, A. Chimienti, M. Lucenteforte, D. Pau, and R. Sannino, "A novel adaptive vector quantization method for memory reduction in MPEG-2 HDTV decoders," Proc. Int. Conf. on Consumer Electronics, pp.58–59, June 1998.
[15] H. Shim, Y. Cho, and N. Chang, "Frame buffer compression using a limited-size code book for low-power display systems," Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia, pp.7–12, Sept. 2005.
[16] Y. Lee, C.-E. Rhee, and H.-J. Lee, "A new frame recompression algorithm integrated with H.264 video compression," Proc. Int. Symp. on Circuits and Systems, pp.1621–1624, May 2007.
[17] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," IEEE Trans. Circuits Syst. Video Technol., vol.13, no.7, pp.560–576, July 2003.
[18] D. Patterson and J. Hennessy, Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufmann, 2003.

Jaesun Kim received the B.S. degree in electrical engineering from Seoul National University, Korea, in 2003. He is currently working toward the Ph.D. degree in electrical engineering and computer science at Seoul National University, Korea. His research interests are in the areas of SoC architecture, multimedia processor and low power design.

Younghoon Kim received the B.S. degree in electrical engineering from Seoul National University, Korea, in 2008. He is currently working toward the M.S. degree in electrical engineering and computer science at Seoul National University, Korea. His research interests are in the areas of mobile multimedia applications and low power design.

Hyuk-Jae Lee received the B.S. and M.S. degrees in Electronics Engineering from Seoul National University, Korea, in 1987 and 1989, respectively, and the Ph.D. degree in Electrical and Computer Engineering from Purdue University at West Lafayette, Indiana, in 1996. From 1998 to 2001, he worked at the Sever and Workstation Chipset Division of Intel Corporation in Hillsboro, Oregon as a senior component design engineer. From 1996 to 1998, he was on the faculty of the Department of Computer Science of Louisiana Tech University at Ruston, Louisiana. In 2001, he joined the School of Electrical Engineering and Computer Science at Seoul National University, Korea, where he is currently working as a professor. He is a founder of Mamurian Design, Inc., a fabless SoC design house for mobile multimedia applications. His research interests are in the areas of computer architecture and SoC design for multimedia applications.