Multimed Tools Appl (2017) 76:17891–17905
DOI 10.1007/s11042-015-3179-z

Image display time reduction scheme for mobile devices

Sung-Bong Jang[1] · **Young-Woong Ko**[2]

Received: 29 October 2015 / Revised: 18 November 2015 / Accepted: 16 December 2015 /
Published online: 23 December 2015

Abstract Image display time is defined as the time taken to load, decode, and render images on a screen. When users attempt to open images stored on a mobile phone, they can sometimes feel that it takes a lot of time to display the images. This paper analyzes the various causes of the delay and presents ideas to reduce the time and improve the customer's experience. To evaluate these ideas, an enhanced image viewer application was implemented and evaluated quantitatively and qualitatively by comparing it to the existing viewer. Experimental results show that the proposed solutions can reduce image display time by more than 10 % for high-pixel images. In addition, it can reduce thumbnail construction times effectively.

Keywords Image processing time · Image decoding · Mobile image viewer

1 Introduction

Recently, it has become very natural for users to take photos and view them instantly using a mobile phone. In particular, the rapid development of image processing technology has led to the wide use of a huge number of images easily. Now, convenient image viewer is an indispensible and important application in a phone. One of important issues in the viewer is how to speed up the image display time when a user wishes to search huge collections of files. Basically, image display processes can be divided into two categories according to cache usage, as illustrated in Fig. 1.

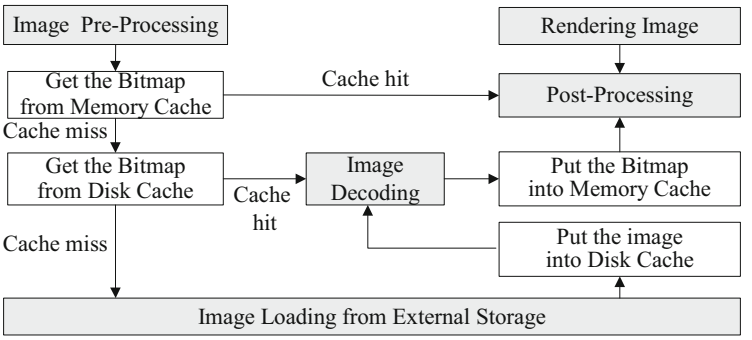
The first category is that the system displays an image with no cache. In this case, the display process consists of five steps, as shown in Fig. 1a. In the first step called image pre-processing, the application constructs thumbnails that are small icon images acquired by partially decoding each image. The users are able to select the target image to be seen on

✉ Young-Woong Ko
yuko@hallym.ac.kr

[1] Department of Computer Software Engineering, Kumoh National Institute of Technology, 61 Daehak-ro, Gumi, Kyoung-Buk 730-701, Republic of Korea

[2] Department of Computer Engineering, Hallym University, Chuncheon, Gangwon 200-702, Republic of Korea

(a) Image Display Processes in Mobile Image Viewer without Cache Operations



(b) Image Display Processes in Mobile Image Viewer with Cache Operations

Fig. 1 Basic processes for mobile image display

the screen by clicking one of thumbnail images. At the second phase, the system loads the chosen image on the memory by referring to the file metadata. In the third step, the loaded file needs to be decoded into YCbCr format by JPEG decoder. However, to understand the decoder, it is necessary to first understand the encoder. The JPEG encoder splits an image into blocks of 8×8 pixels. The blocks are placed in the correct order to constitute the original image. Then, the encoder performs well-known transformations that include a discrete cosine transform (DCT), zigzag scan, quantization and variable length encoding. After these operations are complete, the image is compressed. Meanwhile, the decoder reverses the operations done by the encoder. The decoder reads the compressed image data, and subsequently performs a variable length decoding, zigzag scan, de-quantization, and inverse DCT. As a result, we can obtain the decompressed image.

The host then post-processes the YCbCr data since most images do not fit on the mobile display. For example, if the captured image has 8 M pixels and the liquid crystal display (LCD) display supports only QVGA (320×240), and then it cannot be displayed on the LCD using its original size. In order to display the image, it must be resized using partial scanning. Finally, the host renders the image to the LCD. In this step, sometimes icons or user interface (UI) images are combined with the original images.

The second category for image display processes is that it utilizes cache, as illustrated in Fig. 2b. A cache is used to speed up processing time. There are two types of caches: memory cache and disk cache. If there is a request to display an image, the system checks whether the bitmap data exists in the memory cache or not If the bitmap exists, the image viewer application obtains it from memory cache, but, if it does not, the application checks the disk cache. Finally, if it does not exist even in the disk cache, the application retrieves the bitmap data from external storage.

Without doubt, this scheme significantly speeds up image display time. However, the traditional approach based on cache usage to support fast image display is not sufficient as image sizes become even larger and cache size is very limited. To cope with these limitations, this paper analyzes the causes that degrade image display speed and presents ideas to address them. The remainder of the paper is organized as follows. Section 2 discusses related works,

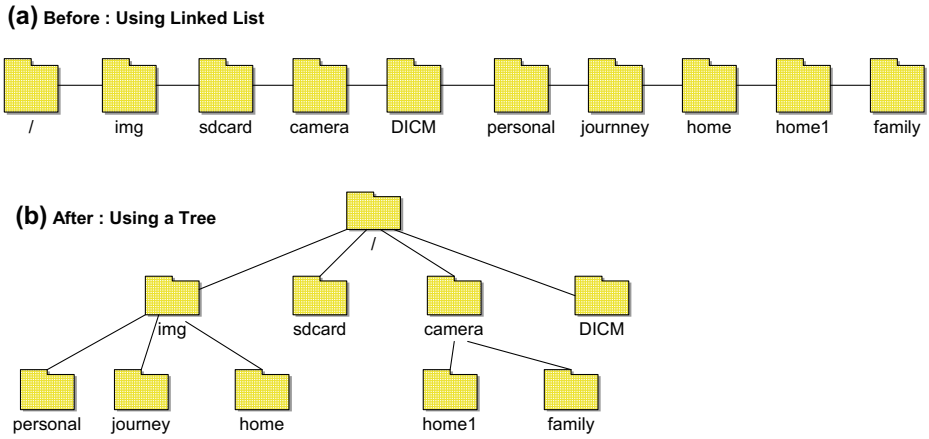


Fig. 2 Representation using linked-list and tree data structures

which includes the recent research efforts to reduce image processing time. Section 3 presents our proposed schemes. The performance evaluation and experimental results are described in Section 4. Finally, Section 5 concludes this paper.

2 Related work

Recently, active research has been conducted in the area of image processing time reduction. This research can be categorized into four approaches: JPEG decoding time reduction, hardware implementation, task-control, and cache approaches. The JPEG decoding time reduction approaches can be found in [4] and [1]. Ho and Lei [4] presented a faster JPEG decoding algorithm that is based on a multiple-bit search scheme during Huffman decoding in a Huffman tree search. The system is able to attain a faster speed using a modified Huffman table, in which special columns are added to contain the number of bits to be scanned next. By doing this, the scheme can significantly decrease the number of search iterations and instructions. Choi et al. tried to reduce the overall processing time by decreasing the inverse DCT (IDCT) processing time. The proposed solution removes computationally time-consuming multiplications, which are replaced by table look-up operations. Experiments show that the proposed scheme significantly reduces the computational time and outperform the other solutions because it efficiently skips the unnecessary operations. The easiest way to reduce mobile image display time is to implement all or part of the JPEG decoders using proprietary hardware [12]. Mody et al. [13] propose a method that is based on a baseline JPEG decoder hardware solution. The proposed scheme consists of two subsystems. The first subsystem upgrades the baseline JPEG decoding functions to improve the Huffman decoding speed. The next system reduces the frequency of DDR memory access by optimizing memory read/write for Progressive JPEG. Experiments show that the solution runs faster than a software JPEG decoder. Another approach based on a hardware implementation was presented by Papadonikolakis et al. [14]. The scheme implements a lossless JPEG-LS encoder in hardware. In addition, they used pipeline design skills, in which the low-complexity features of the special hardware can be used. Their experiments show that the solution is about five times faster than an application-specific integrated circuit (ASIC) implementation. The task-control

approach attempts to reduce processing time by parallelizing or distributing software related to JPEG image display. Klein and Wiseman [10] presented a parallel Huffman decoding approach in which the specific characteristics of the Huffman codes are used for fast resynchronization. In addition, the approach determines the block size efficiently and enforces alignment by adding additional bits to the end of each block. Hong et al. [5] presented a method that is based on the extension of the lib-jpeg JPEG decoder. The approach uses multiple CPUs during JPEG decompression. To enable parallel execution of image decoding on multicore architectures, they extended the image decoder to support thread-level parallelism. In their solution, each image is divided into smaller blocks and each block is decoded separately on the unused cores. In this solution, they combine this approach with a single instruction multiple decode (SIMD) decoder to enable an application to parallelize both data and task. To parallelize the task in the JPEG decoding job, they use a fork/join scheme. Heo [3], Karande and Maral [8], and Kasar and Chincholkar [9] adopted distributed processing to reduce processing time during image retrieval or decompression. Heo [3] presented distributed processing based on object activation technology to reduce the server-side bottle-neck that occurs when an image is being decoded. The server-side bottleneck happens in remote image processing systems where images are sent to a remote server that decodes the images in real-time. Karande and Maral [8] presented a distributed processing scheme based on Artificial Neural Networks (ANNs) to reduce image retrieval time. In their solution, customers transmit an image query to the system for retrieval, where they are interested in finding images similar to the querying image. The distributed system receives the requests through a controller. The controller plays the role of contact point to the customers and assigns various tasks to each node in the system. To evaluate the performance, they implemented the system using Java on ten computers. They used Corel Photo Gallery, which consists of 1000 images, as test images. The research on time reduction uses caches. Jiang et al. [6] presented a scheme to reduce disk access time using a buffer cache. Basically, data blocks, including images, have some spatial locality, which is very crucial to access speed. In general, it is known that the throughput may be an order of magnitude different when the blocks are placed sequentially. Hence, disk performance is degraded greatly when the tasks are not sequential accesses. In order to overcome this limitation, they presented a DUal LOcality (DULO) scheme. The scheme considers localities of the decoder temporally and spatially during disk cache operations. The scheme improves the processing performance by making the read/write requests more sequential and greatly increases the efficiency of scheduling and prefetching. They evaluated the scheme by implementing a prototype system. The experimental results showed that the proposed scheme can greatly enhance the processing speed and decrease the time for executing the decoding.

3 System design and implementation

3.1 Analysis

To obtain possible areas for processing time reduction, we analyzed the image gallery on Android smartphones. First, we analyzed the directory search scheme used in the application and found that a singly linked list is being used to store directory information. In a singly linked list, every node contains some data and a link to the next node. The advantage of this data structure compared to an array is that data elements in lists can easily be removed or

added because it is not necessary for the entire structure to be reallocated or reorganized. Furthermore, it does not degrade the performance when the data elements are not saved continuously on flash memory. In contrast, an array must be declared in the program code before compiling because it is a static memory allocation. When using lists, we can add and remove nodes at any position with constant computational time if the node located in front of the node being removed or inserted is still alive during list search. However, a linked list does not support random access to the data elements in the nodes. In other words, if we try to access the last element (node) in the list, find a node containing a specific data value, or try to locate the position of a node to be inserted, we have to sequentially search all or part of the list nodes. Because the list data are stored in flash memory, unnecessary access to this disk leads to long image loading times. In this case, the data structure must be changed to reduce frequent directory accesses.

Second, we analyzed the JPEG decompression delay. The JPEG decompression delay refers to the time needed to decompress a JPEG image. Decompressing operations are performed in the reverse order of the encoding processes. The first step in decompression is to extract the coding and quantization tables included in the coded bit stream. We then determine the symbols by applying entropy decoding to the coded bit data. Using the symbols, coded data are de-quantized by multiplying the DCT coefficient with the quantization table entries. Finally, we obtain the decoded image by applying the inverse DCT operations to the de-quantized data. The minimum requirement for a JPEG decoder is that it should support a baseline implementation, which includes Huffman encoding, image scans using one to four components, eight bits per pixel (8 bpp), and both interleaved and non-interleaved scans. To reduce the delay, we need to analyze which is the most time-consuming of the decompression processes. An existing study [2] shows that an entropy decoding process spends the greatest amount of time on IDCT and color space conversion, as illustrated in Table 1.

These results show that we should focus on the reduction of variable length decoding time, so-called Huffman decoding. Huffman decoding is a very popular text compression scheme. Huffman's original scheme is not flexible, and so, only two scans for the text are allowed to compress data.

3.2 Proposed scheme

This section describes solutions to speed up the display time. First, in order to reduce the time taken to search a directory containing image file information, as pointed out in the previous section, we change the data structure used to save directory information from singly linked list to a tree. Let T_d represent the time taken to find a directory containing the target image file and let N_i represent the total number of image files to be accessed in the directory. The total

Table 1 Load distribution of JPEG decoder

Sub-process	Percentage of the system load
VLD	36 %
Zig-zag scanning	4 %
Reverse quantization	9 %
IDCT	21 %
Color conversion	14 %
Re-ordering	16 %

computational time taken to open all the image files in the same directory can be computed using the following equation.

$$T_{total} = (N_i * T_d) + N_i(N_i + 1)/2$$

This equation shows that the required time linearly increases proportionally to the number of image files. Therefore, in order to reduce the time, T_d should be minimized and should be performed once. In the original scheme, when users open and click on a specific directory, the mobile image viewer searches the directory entries using a sequential search. It is well known that sequential search takes long time and is not appropriate for real-time systems. A tree is a widely used data structure for storing a set of nodes connected by links (or edges). The elements in each node are organized non-linearly, in contrast to linear data structures such as arrays, linked lists, and queues. Hence, in the best case, the computational time of a directory access can be decreased by O(log(n)). In a tree, there exists a root node, and the remaining nodes are composed of discrete data. Figure 2 illustrates an example of the usage of the tree data structure.

Suppose that a user wants to find the "/camera/home1/" directory. In singly-linked list, the system has to search all previous folders to reach the target folder; however, in a tree, if the elements are sorted, the "home1" directory can be found within two steps. We next attempt to eliminate frequent directory accesses when a target image file is located within the same directory as previously accessed.

To achieve this, a directory cache (buffer) is used to store information about the previously accessed directory. This approach is a very effective way to reduce access time because most users choose images from the currently accessed directory. In this approach, when users want to access a particular directory, the system first searches for it in the directory cache. If it exists there, the system takes it from the cache and delivers it to the user. Otherwise, the system accesses it from the disk. All the elements in a directory cache are connected to each other using a double-linked list. In addition, in order to further reduce directory cache accesses, a least recently used (LRU) policy is used. In LRU, the block to be accessed is typically put into a linked list. When the block is accessed, it is shifted to the end of the list. Therefore, the end of the list always contains the most recently used blocks. Because used items are continuously moved to the back of the list, the least-used ones end up naturally at the front of the list. When there is not enough room in the list, blocks (called victims) are removed from the front in the list. The function call sequence for LRU processing is illustrated in Table 2.

In order to make the cache more effective, it is necessary to enhance data locality and thus improve the cache hit ratio. In other words, most blocks should be stored continuously on the

Table 2 Functions for LRU Operations in the Proposed System

Descriptions	Function name
Initialize a directory cache	ImgDirCache_Init()
Create and allocate memory blocks for storing images	Cache_Create()
Destroy and deallocate an image directory cache	DirectoryCache_Destroy()
Insert an element into a directory cache	DirectoryCache_InsertElement()
Delete an element from a directory cache	DirectoryCache_DelElement()
Search for an element in a directory cache	DirectoryCache_SearchElement()
Compute the number of elements in a directory cache	DirectoryCache_GetElementNum()

disk. The ImgDirCache_Init() function initializes the directory cache and sets the basic parameters to operate the cache. The Cache_Create() function is used to create and allocate memory blocks for image storage. The DirectoryCache_Destory() function is used to deallocate and return used memory. The DirectoryCache_InsertElement(), DirectoryCache_DelElement(), and DirectoryCache_SearchElement() functions are used to insert, delete, and search for an element in a directory cache, respectively.

The second reduction approach is to reduce image file decoding time. The decoding time refers to the time necessary to decompress and the convert JPEG image to bitmap data. The JPEG image file starts with the hex value FFD8, which indicates that the file is compressed as JPEG format. The metadata information starts with the value FFE0, which includes the image creation date, file length, creator, and so on. The value FFC0 represents the frame start marker, after which frame information such as image length, width, bits per pixel, and color or monochrome are included.

The main factors affecting image decoding time can be divided into language-dependent delays and intrinsic decompression delays. A language-dependent delay is a delay specific to a particular language. For example, if the decoder is implemented using Java, it takes longer to decode image than using C or C++. One experiment showed that a Java-based application is two times slower than a C-based application [11]. In an Android-based mobile phone, most user-level applications are implemented using Java. In our work, to reduce this intrinsic Java-related delay, the decoder is re-implemented using C, and its position has been moved down to the middle layer of the Android framework, as shown in Fig. 3.

Third, we attempted to reduce the Huffman decoding time. The typical method to decode a Huffman code uses a binary tree in which the codes are inserted in the tree so that one bit in the code is mapped to either to the left edge (represented by 0) or right edge (represented by 1). Finally, the decoded bytes are put in the leaf nodes in the tree. The large delay incurred during these processes is caused by binary tree operations. One approach to

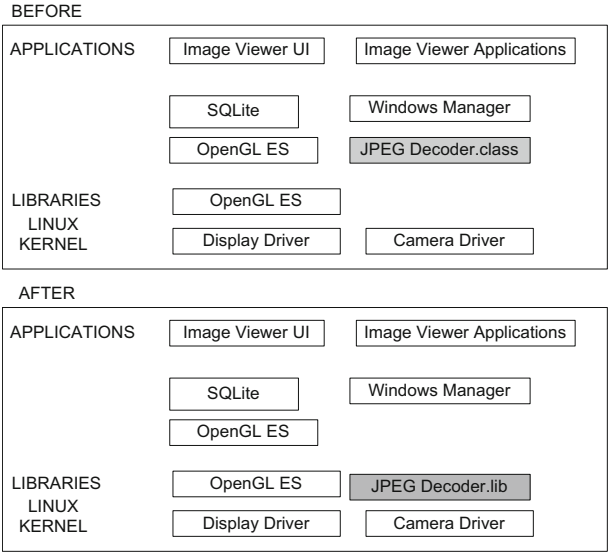


Fig. 3 Changing the position of the JPEG decoder on a mobile framework

speeding up the binary tree operations is to replace runtime computation by a simpler lookup table. A lookup table refers to a data structure based on an array or associative array that can save a significant amount of processing time because memory scanning for data retrieval is much faster than the usual I/O operations. In the scheme, the table must be arranged to make it possible to determine a particular bit-pattern directly, supporting the maximum possible bit-length of code. In reality, because most encoded codes do not have to use the maximum width, they can be put at multiple locations within the table. In addition, the data in the table is used to represent how many bits to truncate from the input and decoded outputs. However, if the largest code is too lengthy, it is impossible to use the table. To solve this problem, smaller lookups with fixed-width subscripts can be used. For example, suppose that a look up table that can store 256 numbers of elements is used to decode a byte. If the input code has a size of more than 8 bits, it is impossible to decode the code using only one table entry, and therefore, the next 8 bits must be inevitably used to handle them. In fact, if the look-up table is not used, it takes several hours to decode a 3 M JPEG image. Therefore, it must be set as the default approach. In addition to the look-up table scheme, our work uses additional schemes to further reduce the Huffman decoding time as follows.

a) The Junior [7] multiple bits read scheme is applied. The scheme combines several bits, detects the specific pattern from the table, and continues reading from that point in the tree. Existing algorithms for reading multiple bits include Junior et al. [7] and Ho and Lei [4] algorithms. The Junior scheme modifies an existing look-up table in which an extra column is added to keep the number of bits to be read next, as illustrated in Table 3.

   This can reduce the number of iterations necessary for decoding because it is possible to read multiple bits at a time. Experimental results show that the scheme can reduce the time spent in decoding by more than 11.59 % when it is applied to an advanced audio coding (AAC) decoder. The disadvantage is that it cannot further reduce the number of search iterations because the code word length increment is 1 or 2 bits. The Ho algorithm is able to process up to 5 bits with a single search. The

Table 3 Huffman table used in the junior approach

Codeword	Offset	Value	Number of bits
00	0	8	2
01	0	9	–
10	2	–	–
11	5	–	–
1000	0	10	2
1001	0	11	–
1010	0	12	–
1011	0	13	–
110	0	14	1
111	1	–	–
1110	0	15	1
1111	0	16	–

advantage of this approach is that it needs only one search to decode the symbol, and thus, it can greatly improve the computational load. However, it can only be applied to code words whose increment values range from one to five. Because of this problem, our work adopts the Junior approach for reading multiple bits.

b) The bit read operation is implemented using assembly language. This implementation takes less time to read a bit from memory than when it is implemented using C. The bit read assembly code is presented in Fig. 4.

In Fig. 4, the read_bit() function has two input parameters: "data" and "bit_position." Here, "data" points to the data to be read, and "bit_postion" is the bit number to be read. The code first moves one byte to the eax register. Second, it adds "length" to the eax register and moves the value of "offset" to the ecx register. Next, it masks the ah register with one and tests if ah is zero. If ah is zero, the bit is zero. It then returns zero. Otherwise, if ah is not zero, the bit read is one.

c) In our work, we replaced arithmetic operators by bitwise operators because it is known that bitwise operations are faster than arithmetic operations [15,16]. Bitwise operators are special purpose operators that are usually used to make a program on the CPU processor. In low level processors, arithmetic operations such as addition, subtraction, multiplication, and division are converted into bitwise operations because they make processing faster. In addition, in order to maximize cache hit ratio, we use data alignment to optimize cache usage. All data types in a language have an associated alignment, which is enforced by the central processing unit (CPU). Data alignment improves run-time performance because the CPU is able to fetch data from storage in an efficient manner. The compiler provides various mechanisms such as "padding" for this data aligning, in which unused memory is inserted between data elements. Sometimes, the compiler expands the size of an element to make it double the alignment. Using this data alignment, data cache usage can be optimized. For example, in C, small data elements can be clustered into a "struct" data type and enforce the data to be allocated at the starting position of a cache line. This results in a large

Fig. 4 Assembly source code for reading a bit from a look-up table

```
inline int read_bit(unsigned char *data, unsigned int bitposition) {
            unsigned int length = bit_position / 8;
            unsigned int offset = bit_position % 8;
            int return_bit;

            __asm {
                  mov eax, data;        /* move one byte to eax register */
                  add eax, length ;     /* add length to eax register */
                  mov ecx, offset;   /* move offset value to ecx register */
                  bt[eax], ecx;
                 lahf;
                 and ah, 1;   /* mask ah with 1 */
                   test ah, ah;   /* test if ah is zero */
                   jz zero;          /* If ah is zero, go to labled zero */
                  mov bit, 1;      /* The read bit is one  */
                  jmp end;         /* The read bit is zero */
zero:
                  mov bit, 0;
end:

            }
             return return_bit;
}
```

performance improvement because it guarantees that each element is effectively loaded into the cache whenever any one element is accessed. Suppose that two variables j and k are very frequently accessed together but always allocated on different cache lines. They can be declared using a cache line enforcement keyword, as follows.

$$\_\_\text{equal\_cacheline}\ (\text{align}(16))\ \text{struct}\ \{\text{int}\ j, k\}\ \text{sub};$$

This declaration ensures that the two variables are always allocated on the same cache line.

d) We execute Huffman decoding in parallel. If the CPU has a multicore architecture, the compressed image can be divided into separate blocks (for example, 4×4 blocks), and each block is assigned to each processor. When applying this scheme, the boundary of a block cannot be decided clearly. The reason for this is that codewords in the Huffman method have variable lengths. The simplest way to solve the problem is to determine the size of the blocks in advance and add padding bits to the end of each block. The most common approach to implementing a codeword with a fixed block b is as follows. First, we traverse a code string S and continue until S is null. Second, the prefix of block b from S is removed. The prefix indicates a new block c. Fourth, the last codeword is checked to determine whether it fits in the block c. The basic approach to guarantee parallelism is to assign one processor to a task for decoding block b and keep decoding $(b+1)$. Figure 5 shows an example of parallel execution.

```
Get index_of_block_list and save it into i;
If (Block(i) != LAST_BLOCK) {  /* if statement */
        increase(i);
        Get head data and save it into temp_block_index;
        while    {       /* decode character code */
          Read a code and save it into temp_code ;
              if  (temp_code != End_of_Code) {
                    if  (temp_block is skipped)          {
                  increase(i);
                  Get head data and save it into temp_block;
                } else {            /* Start decoding */
                  k=Get index of Code block in Block(i);
                        while  Block[temp_block_index].index < k  {
                      Move the pointer to the next in the block list;
                      Delete old item in the block list;
                  }  /* end of whilw */
                    }
            }
              if   block_list[i].index == k  break;
              else  add (k,temp_code) at the front of the list;
        }  /* End of out while loop*/
}
```

Fig. 5 Parallel execution of huffman decoding

4 Experiment results

This section describes the experiments conducted for this study. To evaluate the feasibility of the proposed idea, the existing mobile image viewer was changed and the solutions were implemented on an Android-based mobile system. The evaluation system specifications are shown in Table 4. A Cortex-A15 CPU was used in the evaluation system: its maximum speed is 1.6 GHz and it has 2 GByte of DD3 memory. In the system, the graphics processing unit (GPU) was not activated because if it is used, the decoding time cannot be precisely calculated.

The basic software architecture is illustrated in Fig. 3. The top layer includes the Java image applications. For example, Image Viewer UI is an application that provides an interface to users. Image viewer applications are applications related to image processing. The JPEG decoder library is located at the middle layer. The decoded image is converted into RGB data and displayed on the LCD. In reality, to reduce the display time, we need to find a method to reduce the LCD processing overhead. To do this, we have to change the kernel code. For quantitative evaluation, we measured image decoding time for sample images saved in flash memory in the phone and compared the decoding time on existing image gallery with that on an enhanced image viewer. The decoding time is defined as follows. When a user starts image gallery, the thumbnail images appears on the screen. When a user clicks on one of these images, the system records the before-decoding timestamp, and when the image decoding finishes, the system records the after-decoding timestamp. The total amount of time taken to display a image can be computed by subtracting the before-decoding time from the after-decoding time. The time measurements were repeated 10 times per image and averaged.

The total decoding time is saved in an output file. In the experiment, if an image that is the same as a previous one is selected when the thumbnails are open, there exists the possibility that the decoding time calculation is wrong. Sizes of the sample images are 5 and 8 M (3280×2447) pixels. Figure 6 shows the experimental results, which shows that both loading time increases linearly proportionally to the image pixels size.

From this figure, it can be seen that when the target image size increases, the effect also increases. In fact, there is almost no reduction in time for images smaller than VGA. However, for large images of 8 M, the reduction in time can reached 10 %.

We next measured and compared the times to construct thumbnail images in a viewer. In the experiments, we created various numbers of image files up to 800. The experimental results are illustrated in Fig. 7. From this figure, we can see that the construction time

Table 4 Evaluation system specifications

Items	Specifications
Processor	Cortex-A15 1.6 GHz
Memory	2 GByte LPDDR3 PoP (800 MHz)
GPU	PowerVR GPU (OpenGL ES 2.0 and 1.1)
Display	7 in. WXGA (800 × 1280) IPS LCD
Touch key	4EA capacitive touch key
Storage	eMMC 16 GByte (eMMC 4.5 Support)
PMIC	Power management IC
Digital video	HDMI 1.5 video out (1080 p)
Camera	3 M CMOS sensor

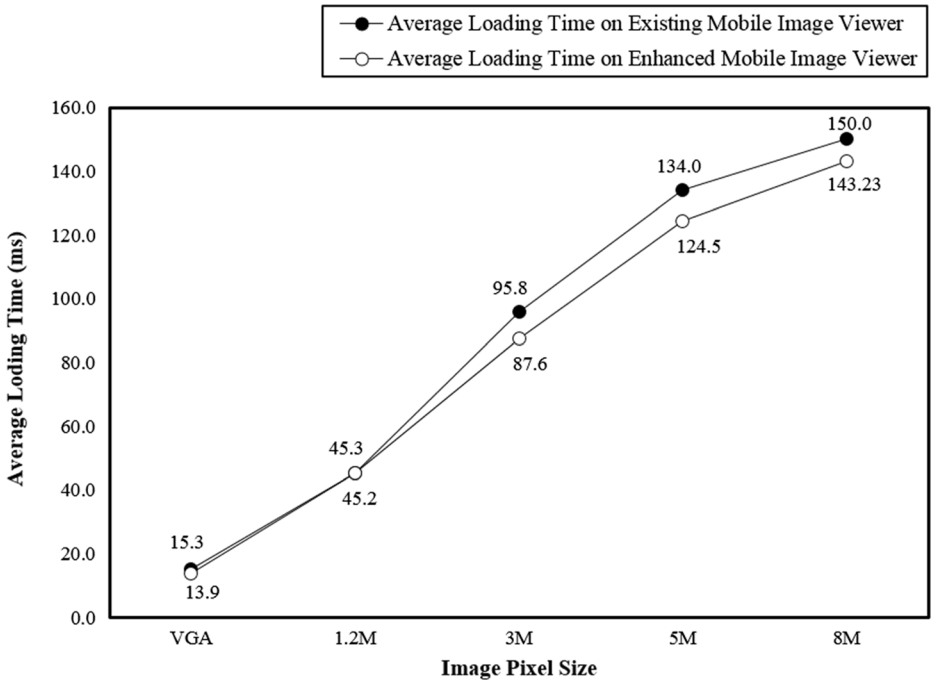


Fig. 6 Comparison of average loading time for sample images

exponentially increases with the number of image files in the existing solution. However, in the proposed scheme, the time is much smaller. This is very natural because we have changed the data structure for storing file directories into a tree structure.

In addition, to evaluate the performance, we conducted a subjective image display test. Even though it is not easy to get reliable measurements from this test, it is necessary for real users to feel the difference. In the test, participants do not know which mobile phone has an enhanced image viewer. In the test, a number of students were chosen ranging from 20 to 24 years. All students were given two mobile phones: one adopting the conventional image viewer, the other with an enhance image viewer. A total 60 small, medium, and large sizes images were installed on each phone. The small sized images were QVGA and QCIF images; medium sized images were VGA, 1.2, and 3 M images; and large sized images were SVGA images. The tester opened the image viewer, clicked on the image, and flicked through the images for each size. The tester was then asked to give a mean opinion score (MOS) from one to five for each image size. The same test was repeated on the other phone by the tester. To remove the variability among testers, a two-way analysis of variance was done on the original scores. The MOSs were calculated with the 95 % confidence intervals for each image size after normalization. The MOS results were also averaged for each image size. The final results are shown in Table 5.

From Table 5, we can see that for small and medium sized images, the proposed scheme does not affect the MOS results, and there is almost no difference between the schemes. We can guess that most participants could not feel any difference between the existing and proposed schemes because the change is too small. However, for large images, the MOS value for the proposed scheme is higher than that of the conventional scheme. This is as

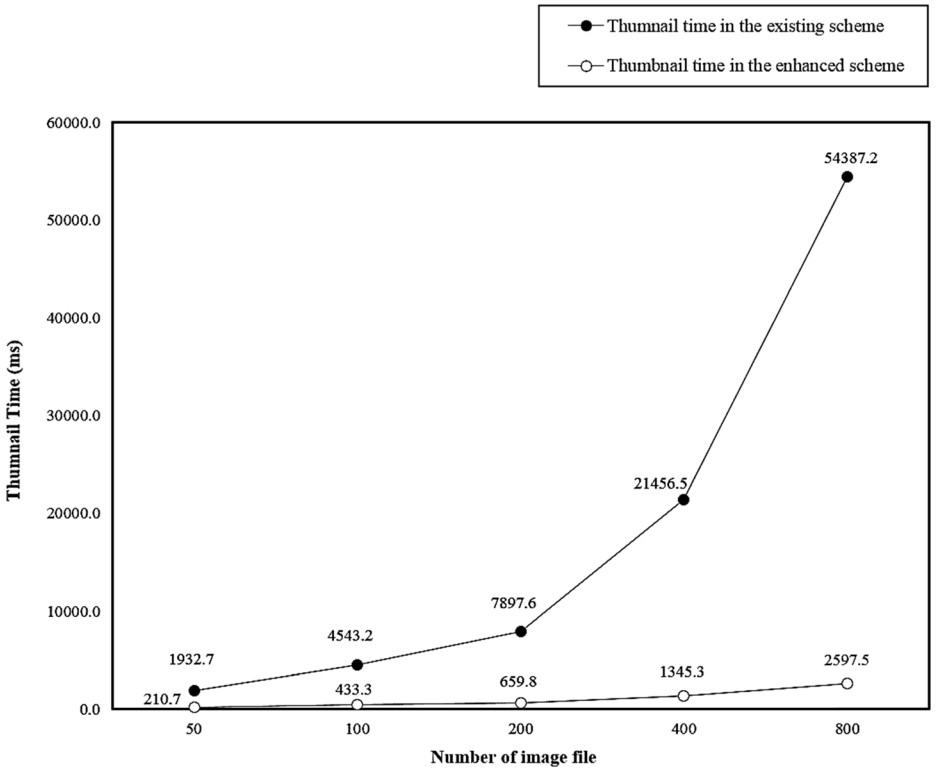


Fig. 7 Comparison of thumbnail construction time

expected and almost similar to the quantitative results because the proposed scheme can reduce the display time by more than 10 %.

5 Discussion

In future, it is predicted that the size of images to be displayed on a mobile phone will become much larger. Thus, it will take more time to display an image because of the limited capacity of mobile CPUs. This paper analyzes some causes of the long delay in a mobile image viewer, and presents ideas to reduce this delay. The proposed ideas can be used as hints when a delay problem happens in a mobile image viewer. To evaluate the performance, a new image viewer application was implemented and evaluated quantitatively and qualitatively by comparing it to the existing scheme. Experimental results show that proposed solutions can reduce the image

Table 5 Qualitative evaluation results

Schemes	Image size		
	Small	Medium	Large
Conventional scheme	4.67	4.23	3.34
Proposed scheme	4.64	4.21	3.89

display time by more than 10 % for high-pixel images. In addition, it can reduce thumbnail construction times effectively. The most efficient way is to implement a JPEG decoder using a proprietary hardware. However, this approach is too expensive because it requires system changes. Future work is to integrate the cache scheme.

Acknowledgments This paper was supported by Research Fund, Kumoh National Institute of Technology and this research was also supported by Hallym University Research Fund (HRF-201412-010).

References

1. Choi K, Lee S, Jang ES (2010) Zero coefficient-aware IDCT algorithm for fast video decoding. IEEE Trans Consum Electron 56(3):1822–1829
2. Cucchiara R, Piccardi M, Prati A (2014) Neighbor cache prefetching for multimedia image and video processing. IEEE Trans Multimedia 6(4):539–552
3. Heo JK (2011) Distributed image preprocessing using object activation. J Inst Webcasting Internet Telecommun 11(1):87–92
4. Ho H-C, Lei S-F (2010) Fast huffman decoding algorithm by multiple-bit length search scheme for MPEG-2/4 AAC. IEEE ISCAS 2844–2847
5. Hong J, Sodsong W, Chung S, Kim CG, Lim Y, Kim SD, Burgstaller B (2012) Task-parallel JPEG decoding with the Libjpeg-turbo library. ICIST
6. Jiang S, Ding X, Chen F, Tan E, Zhang X (2005) DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. 4th USENIX Conf FAST 101–114
7. Junior GGS, Lima MAM, Filho WOG, Perkusich A, Morais MRA, Lima AMN (2008) A fast and memory efficient huffman decoding method for the MPEG-4 AAC standard. ICCE 1–2
8. Karande SJ, Maral V (2014) ANN based semantic content based image retrieval with distributed processing. 2014 IC3I 513–516
9. Kasar M, Chincholkar YD (2013) JPEG algorithm implementation using ARM processor. Int Adv Res Comput Sci Softw Eng 3(7):211–215
10. Klein ST, Wiseman Y (2003) Parallel Huffman decoding with applications to JPEG files. Comput J 46(5): 487–497
11. Lee J, Ahn J, Lee Y, Kim C (2013) Implementation of semantic directory service on image gallery application of mobile devices. ICISA 2013:1–2
12. Milinkovic SA (2012) Software JPEG decoder for embedded systems. 2012 20th TELFOR 1284–1287
13. Mody M, Paladiya V, Ahuja K (2013) Efficient progressive JPEG decoder using JPEG baseline hardware. ICIIP 369–372
14. Papadonikolakis M, Pantazis V, Kakarountas AP (2007) Efficient high-performance ASIC implementation of JPEG-LS encoder. In Proc. of the DATECE 1–6
15. Roy S, Mitra A, Setua SK (2015) Color & grayscale image representation using multivector. Third Intl Conf C3I 1–6
16. Wang X, Jia J, Yin J, Cai L (2013) Interpretable aesthetic features for affective image classification. 20th IEEE ICIP 3230–3234

Sung Bong Jang Sung-Bong Jang received his B.S., M.S., and Ph.D. degrees from Korea University, Seoul, Korea in 1997, 1999, and 2010, respectively. He worked at the Mobile Handset R&D Center, LG Electronics from 1999 to 2012. Currently, he is an assistant professor in the Department of Computer Software Engineering, Kumoh National Institute of Technology in Korea. His interests include Software Documentation Model, Multimedia Security, and Big Data Privacy Protection.

Young Woong Ko Young Woong Ko received both a M.S. and Ph.D. in computer science from Korea University, Seoul, Korea, in 1999 and 2003, respectively. He is now a professor in Department of Computer engineering, Hallym University in Korea. His research interests include operating systems, embedded systems and multimedia systems.