Stage-based Frame-Partitioned Parallelization of H.264/AVC Decoding

Won-Jin Kim, Keol Cho and Ki-Seok Chung, *Member*, IEEE

Abstract — *Strong demands for high resolution video services lead to active studies on high speed video processing. Especially, widespread deployment of multi-core systems accelerates researches on high resolution video processing based on parallelization of multimedia software. In this paper, we propose a novel parallel H.264/AVC decoding scheme on a homogeneous multi-core platform. Parallelization of H.264/AVC decoding is challenging not only because parallelization may incur significant synchronization overhead but also because software may have complicated dependencies. To overcome such issues, we propose a novel approach called Stage-based Frame-Partitioned Parallelization (SFPP). In SFPP, we divide a frame into multiple partitions, and execute them in a pipelined fashion. To reduce synchronization overhead, a separate thread is allocated to each stage in the pipeline. In addition, an efficient memory reuse technique is used to reduce the memory requirement. To verify the effectiveness of the proposed approach, we parallelized FFmpeg H.264/AVC decoder with the proposed technique using OpenMP, and carried out experiments on an Intel Quad-Core platform. The proposed design performs better than FFmpeg H.264/AVC decoder before parallelization by 53%. We also reduced the amount of memory usage by 65% and 81% for a high-definition (HD) and a full high-definition (FHD) video, respectively compared with that of a popular existing method*[1].

Index Terms — H.264/AVC, decoding, parallel processing.

I. INTRODUCTION

Demands for high resolution video processing techniques are rapidly increasing as high-definition digital broadcasting services are widely provided. Therefore, standards and techniques for video compression and decoding are being actively developed. One of the most popular video codec standards is H.264/AVC. One main advantage of H.264/AVC is that it is capable of providing good video quality at substantially lower bit rates than previous standards. However, it is very challenging to achieve high performance by a software implementation because its complexity is pretty high. Therefore hardware implementations have been used commonly. Recently, enhancing performance through intelligent parallel processing with multiple cores integrated in a single chip is widely adopted. Since a multi-core system typically has better power efficiency than a single core system with a comparable processing power, it is expected that many high performance embedded systems will adopt multi-processor system-on-chip platforms soon. Even though multi-core systems will provide potentially ample computation power, it is not straightforward to achieve such high performance because efficient parallel programming for a multi-core system is difficult. Thus, it is crucial to re-write an existing software implementation into a new implementation which is more suitable for parallel processing. Also, it is very important to understand the target multi-core platform to achieve high performance. Understanding a target platform includes understanding of not only the processor itself but also the memory system and the on-chip interconnection system.

In this paper, we propose a novel parallel software implementation of H.264/AVC decoding. Parallel software implementation may be roughly classified into two categories: task parallelism or data parallelism. Task parallelism is to divide a task into multiple sub-tasks and to distribute each sub-task to multiple cores simultaneously. Data parallelism is to divide a block of data into multiple sub-blocks, and let them be processed by multiple cores in parallel. To parallelize H.264/AVC decoding, both approaches have been proposed. In case of task-level parallelism, it is not easy to divide a task into sub-tasks evenly. Therefore, the execution time of some sub-task is longer than that of others, which implies that task parallelism is not an efficient way to achieve high performance. On the other hand, data parallelism also has some limitation since complicated data dependencies exist in H.264/AVC decoding algorithm. Therefore it is not easy to partition a block of data into a set of independent sub-blocks. To overcome such issues, we propose a novel approach called Stage-based Frame-Partitioned Parallelization (SFPP). In this approach, first we divide a frame data into equally sized partitions. Partitioned data are processed in a pipelined fashion. A thread is created and assigned for each pipeline stage to reduce the overhead due to thread synchronization. In addition, an efficient memory reuse technique is used to reduce the memory requirement.

The rest of this paper is organized as follows. In Section II, we briefly present an overview of H.264/AVC decoding. Next, we introduce existing approaches for parallelization of H.264/AVC decoding. In Section III, we explain the proposed

[1] Won-Jin Kim is with the Department of Electronics, Computer & Communication Engineering, Hanyang University, Seoul, Korea, e-mail: kwonjin97@gmail.com).

Keol Cho is with the Department of Electronics, Computer & Communication Engineering, Hanyang University, Seoul, Korea, e-mail: keolman2@gmail.com).

Ki-Seok Chung is the corresponding author, and with the Department of Electronics, Computer & Communication Engineering, Hanyang University, Seoul, Korea, e-mail: kchung@hanyang.ac.kr).

Contributed Paper
Manuscript received April 15, 2010
Current version published 06 29 2010;
Electronic version published 07 06 2010.

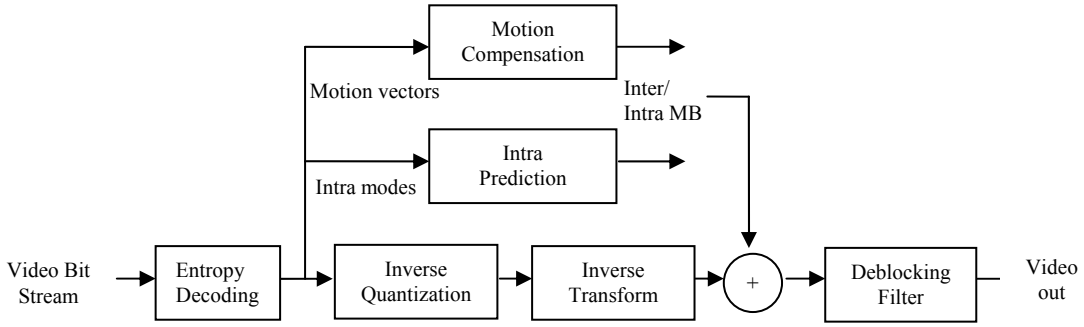

Fig. 1. Block diagram of H.264/AVC decoding.

SFPP method. In Section IV, experimental environments and experimental results will be addressed. Conclusion and future works will follow.

II. Related Work

A. H. 264/AVC decoder

H.264 or MPEG-4 AVC is a video compression standard proposed by ISO/IEC and ITU. It is popular since its compression ratio is much better than other existing standards and it is adequate for video streaming through networks. Detailed specification of H.264/AVC standard and key algorithms are found in ITU H.264 standard [1] and ISO MPEG-4/AVC standard [2]. Also, [3] and [4] explain the overview of the H.264 standard. H.264/AVC decoding consists of entropy decoding, inverse discrete cosine transformation, inverse quantization, intra prediction, motion compensation, and deblocking filter. A brief explanation of these steps is given as follows.

1) Entropy Decoding (ED)

A bit stream in H.264/AVC is received as a unit of Network Adaptation Layer (NAL) and a set of coefficients is generated by an entropy decoder. In H.264/AVC, either Context-Adaptive Variable Length Coding (CAVLC) or Context-Adaptive Binary Arithmetic Coding (CABAC) is employed as an entropy decoding technique. In a baseline profile, CAVLC is often employed, while CABAC is more commonly used in the main and the high profile.

2) Inverse Transformation (IT)/Inverse Quantization (IQ)

Inverse transformation (IT) and inverse quantization (IQ) steps process the set of coefficients generated by the entropy decoding to generate a set of residual data.

3) Intra Prediction (IP) and Motion Compensation (MC)

According to the type of a macroblock, either intra-prediction (IP) or motion compensation (MC) is applied. IP explores spatial redundancy between blocks within a frame while MC explores temporal redundancy between successive frames. After IP and MC steps, the generated block is combined with the residual data from IT and IQ.

4) Deblocking Filter (DF)

The resulting decoded video may suffer from blocking effects which reveals the boundary between blocks. To eliminate the boundary, deblocking filtering is necessary. Deblocking filter (DF) adaptively controls weights to avoid blocking effects

B. Parallelization of H.264/AVC decoding

Parallelization of H.264/AVC decoding has been studied actively. A typical structure for encoded data structure is shown in Fig. 2. A sequence of encoded video data consists of Group of Pictures (GOP). A GOP consists of multiple frames (or pictures), and each frame is composed of one or multiple slices where each slice is a group of macro blocks. Various parallelization techniques of H.264/AVC encoding have been proposed [5]-[9]. For parallelizing H.264/AVC decoding, either task-level parallelism or data-level parallelism has been proposed [10]-[17]. For data-level parallelism, depending on the size of data, slice-level parallelism [12] and macro-block parallelism [13]-[17] have been studied. In [14], data scheduling techniques for parallel processing were proposed. In [15] and [16], entropy decoding and deblocking filtering of previously entropy-decoded block are processed in parallel. In the following, we briefly explain the pros and cons of the task-level parallelism and the data-level parallelism.

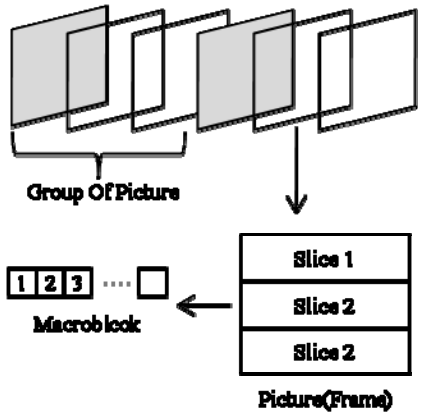

Fig. 2. H.264/AVC video frames.

1) Task-level parallelization techniques

Task parallelism is carried out by dividing a task into multiple sub-tasks and distributing each sub-task to a core

simultaneously [10][13]. It is similar to a typical pipelined execution. A major problem in using task-level parallelism for H.264/AVC decoding is that it is not easy to partition the task in such a way that the execution time of each sub-task should be evenly balanced. Fig. 3 shows an example when a task-level parallelization technique is applied.

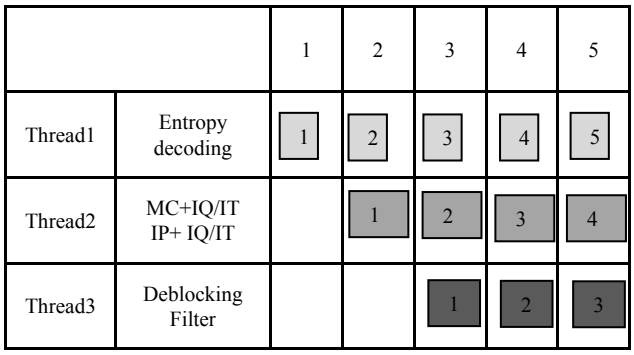

Fig. 3. Pipelined parallel processing.

You can see that execution times for entropy decoding, MC+IQ/IT, IP+IQ/IT, and deblocking filter are all quite different. Also, each thread processing time will differ depending on the type of macro-blocks. There are three types of macroblocks in H.264/AVC: intra-coded, inter-coded, and skipped macroblocks. Intra-coded macroblocks are encoded by utilizing spatial redundancies while inter-coded blocks are encoded by utilizing temporal redundancies. A skipped block is basically the reference block since the skipped macroblock and the reference block are almost identical. Therefore, the reference block is used instead of processing the skipped block.

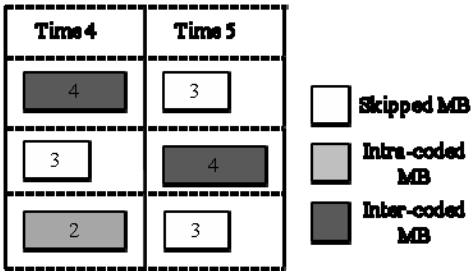

Fig. 4. Processing time of macroblocks.

Fig. 4 shows that processing skipped blocks takes much shorter time than processing inter-coded or intra-coded blocks. Thus, each step may take a variable amount of time depending on the block type. Since the pipeline step time should be greater than equal to the slowest step processing time, this variable step processing time may cause significant wastes in processing time.

2) Data-level parallelization techniques

Data parallelism is to divide a block of data into multiple sub-blocks, and let them be processed by multiple cores in parallel. The unit of the divided data block may be a frame, a slice or a macroblock. There are three types of frames in H.264/AVC: I-frames, P-frames and B-frames. I-frames contain only intra-coded macroblocks while P-frames contain either intra-coded macroblocks or inter-coded macroblocks. B-frames contain bidirectional inter-coded macroblocks in addition to intra-coded and inter-coded macroblocks. Frame-level parallelization may have inter-frame dependencies for P-frames and B-frames. Slice-level parallelization does not have inter-slice dependencies but encoding should have been done for each slice after dividing each frame into slices. Therefore, macroblock-based parallelization approaches have actively been attempted [13]-[17]. Macroblock-level parallelization is typically carried out by allocating a thread to process a macroblock as shown in Fig. 6. However, data dependencies exist in H.264/AVC as shown in Fig. 5. For example, before we conduct an intra-prediction step for Current MB, intra-predictions for macroblocks 1, 2, 3 and 4 must be done first.

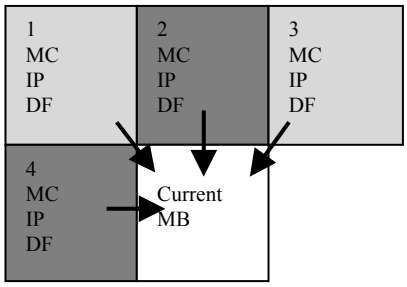

Fig. 5. Spatial data dependencies for a macroblock.

Fig. 6 shows an example of data-level parallel processing observing data dependencies. Macroblocks MB(4,0), MB(2,1) and MB(0,2) can be processed in parallel at the 5th time step(T5).

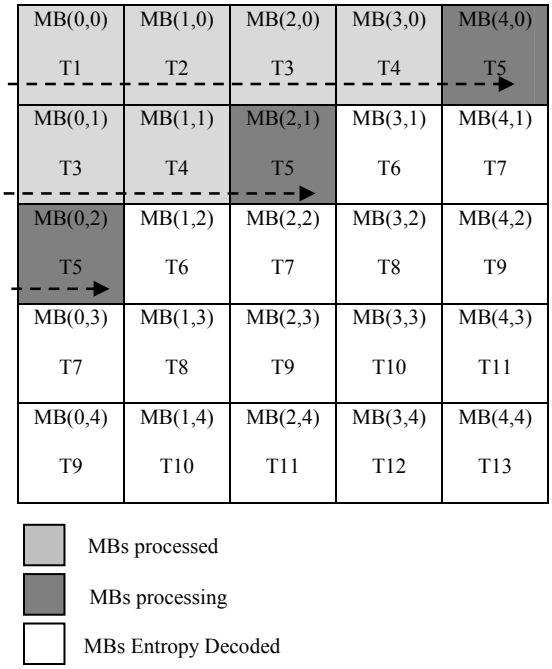

Fig. 6. Example of data-level parallelization.

Yet, since the processing time of each macroblock will be different, synchronization should be taken care of. One of the popular data-level parallelization methods is 2D-Wave[10]. In 2D-Wave, threads are allocated to macroblocks and processed in the order that arrows indicate in Fig. 6. 3D-Wave [17] is an extension to 2D-Wave.

Fig. 7 shows how threads are allocated in data-level parallel processing. Entropy decoding should be done before data-level parallel processing starts since entropy decoding cannot process partitioned data in parallel. Macroblocks generated as a result from entropy decoding should be saved in memory. In case of FHD, 1920x1080, 8160 macroblocks are created, which will require a significant amount of memory space. This memory requirement imposes a heavy burden on the system when we increase the level of data parallelism.

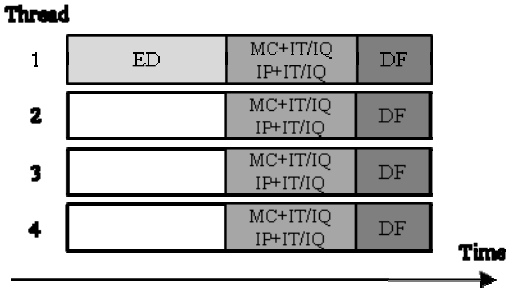

Fig. 7. Thread allocation in data-level parallelization.

Even though various approaches have been proposed for both task-level and data-level parallelization, most techniques may not fully overcome major problems such as synchronization overhead, complex data dependency, and heavy memory requirement. Therefore, we propose a novel method to improve the decoding performance by resolving these issues. The proposed method is called Stage-based Frame-Partitioned Parallelization (SFPP). In SFPP, each frame is partitioned horizontally, and each partition is processed in a pipelined parallel fashion. By allocating independent threads to each stage of the pipelining, synchronization overheads are considerably reduced. Also, by utilizing a memory reuse method, the memory requirement has been significantly reduced. Our experimental results show that by applying SFPP, the decoding performance is improved by up to 53% over the implementation before we apply SFPP. The memory requirement is reduced by 65% for HD and by 81% for FHD when compared with 2D-Wave.

3) Parallelization using OpenMP

OpenMP[20][21] is an application program interface standard for a shared-memory multi-processor. Many commercial compilers support OpenMP standards. Application programmers can parallelize a C or Fortran program by inserting OpenMP pragmas properly. Since OpenMP is a pragma-based parallelization mechanism, you don't have to change the application code itself. Therefore, it's quite a simple way to parallelize a code which was originally written for a sequential processing. Inserted OpenMP pragmas are pre-processed by an OpenMP-compliant compiler, and as a result, multiple threads for parallel processing are generated. We insert OpenMP pragmas to parallelize the decoding process.

III. STAGE-BASED FRAME-PARTITIONED PARALLELIZATION

In this section, we will explain our proposed SFPP in detail. The main goal of this method is to achieve an excellent speed-up for high resolution video processing by parallelization of the H.264/AVC decoding process. The H.264/AVC decoding consists of Entropy Decoding (ED), Inverse Quantization (IQ), Inverse Transformation (IT), Intra-Prediction (IP), Motion Compensation (MC), and Deblocking Filter (DF) steps. Depending on the type of a macroblock, either MC or IP step is carried out. Typically IQ and IT step are processed combined with MC or IP in a single step since the processing time of IQ and IT is too small to be processed as a separate step. Both JM which is a popular reference software for H.264/AVC decoding and FFmpeg which is being developed as an open source H.264/AVC decoder project execute IQ/IT combined with MC or IP in a single step. Therefore, in SPFF, we also divide a decoding process into three pipeline stages: ED, either MC+IQ/IT or IP+IQ/IT, and DF. Decoding is processed in a pipelined fashion after allocating threads to these three stages. SFPP is applied in three steps: frame partitioning, thread creation, and memory utilization.

A. Frame Partitioning

Conventional pipelined processing decodes one macroblock at a time, and checks synchronization of each thread at the end of every step. Thread synchronization often becomes a serious performance bottleneck. As the video resolution becomes higher, the number of macroblocks in a fame increases proportionally, and so does the synchronization overhead. To avoid such overhead, SFPP partitions frames and processes each partition in a pipelined fashion. Fig. 8 shows an example of frame partitioning. This method is similar to what a superscalar [18] or a VLIW [19] architecture does. By processing a partitioned frame which is a group of macroblocks, we can reduce synchronization overhead considerably compared with a macroblock-level pipelined processing.

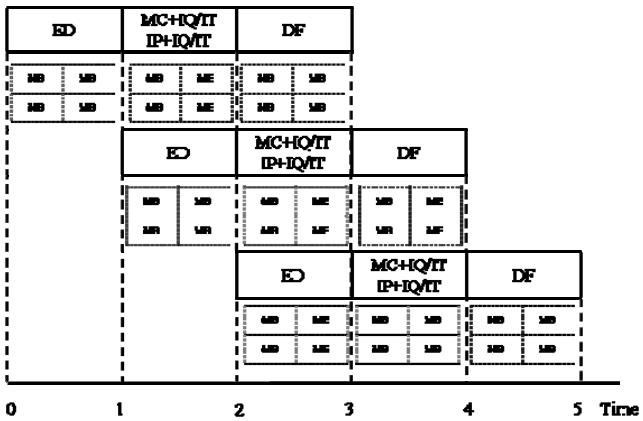

Fig. 8. Pipelined execution of partitioned frames.

Fig. 9 shows an example of pipelining with a partitioned frame with a straightforward thread allocation. Thread1 is allocated to ED, and Thread2 is allocated to either MC+IQ/IT or IP+IQ/IT. Thread3 is allocated to DF. When we allocate threads in this way, thread synchronization is carried out at the end of every stage. For example, in Fig. 9 before we start the fourth time step in the pipelining, thread processing for the third step should be complete. For this reason, synchronization overhead may still be significant.

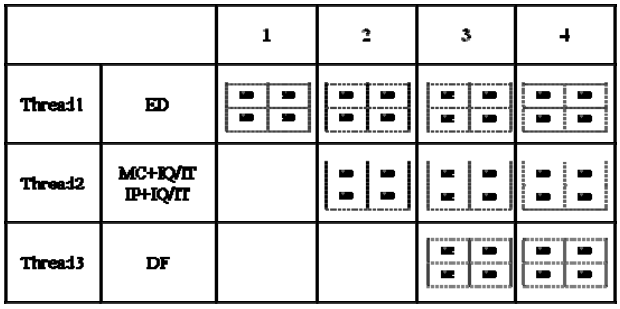

Fig. 9. Thread assignment for pipelined execution .

B. Thread Creation for Stage

To reduce thread synchronization overhead, we allocate threads to each pipeline step independently. Each thread created in each pipelined stage is terminated after processing one group of macroblocks. Fig. 10 shows how threads are allocated in SFPP. For example, for the third time step, Thread3-1, Thread3-2 and Thread3-3 are created and they are terminated after processing a partition of a frame. Similarly, for the fourth stage, Thread4-1, Thread4-2, and Thread4-3 are created. And these are terminated upon the completion of the stage processing. Since threads are independently created and terminated, the thread synchronization overhead is significantly reduced.

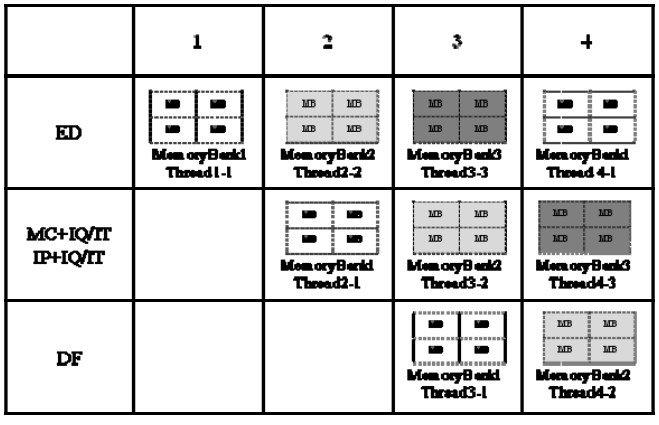

Fig. 10. Thread assignment in SFPP.

However, SFPP may suffer from overhead due to frequent thread creation. There is a trade-off between overhead due to thread creation and the size of each partition. If we have a smaller partition, we have more groups of macroblocks and more frequent thread creations. On the contrary, bigger partitions may suffer from other overhead in handling bigger partitions such as memory requirement. Therefore, we need to determine the best partition size for each resolution based on experiments on performance evaluations. Performance evaluation results of SFPP with respect to various partition sizes for two different resolutions are summarized in Table I. N_{MB}/G denotes the number of macroblocks per group and Nps denotes the number of total pipeline steps. For FHD, 1920x1080 videos, we have tried 240 macroblocks per group with 34 total pipeline steps (240x34), 480 with 17 (480x17) and 960 with 8 (960x8). Performance evaluation results show that (480x17) is the best among three. For HD, 1280x720 videos, we have evaluated with respect to three different partitions: (240x15), (400x9) and (600x6). As it turns out that (400x9) is the best in case of HD videos. More detailed explanation on experimental environment and results will be presented in Section IV.

TABLE I
PERFORMANCE EVALUATION FOR VARIOUS PARTITION SIZES

N_{MB}/G x N_{ps}	FHD,1920X1088, 8160 MB (frame/μs)			
	rush_hour	blue_sky	pedestrian_area	sunflower
240 x 34	10524	10025	11947	9129
480 x 17	9236	8477	10875	8822
960 x 8	11443	11965	12891	9979

N_{MB}/G x N_{ps}	HD,1280X720, 3600 MB (frame/μs)			
	rush_hour	blue_sky	pedestrian_area	sunflower
240 x 15	7665	6016	5134	5409
400 x 9	6994	5458	4802	4950
600 x 6	7915	6272	5148	5645

C. Memory Utilization Strategy

As the video resolution becomes higher, the number of macroblocks per frame increases rapidly. In parallelization of H.264/AVC decoding, macroblocks are generated after entropy decoding and the generated macroblocks are stored in memory. For a high resolution video decoding, a large amount of memory is required to store the generated macroblocks. A large amount of required memory will cause not only design cost to go up, but also system performance to go down. To reduce the memory requirement, we employ a memory reuse method. We define memory bank as the memory block required to store one partition of macroblocks. In SFPP, an individual thread is accessing a separate memory bank. Fig. 11 shows how a thread accesses a memory bank. In Time1, Thread1-1 stores macroblocks generated by an ED process into Memory Bank1. Then in Time2, Thread2-1 obtains data from Memory Bank1 for either MC+IQ/IT or IP+IQ/IT. Then in Time3, Thread3-1 uses Memory Bank1 for DF. Since Thread3-3 is the last process before we get a fully decoded video, in Time4, Thread4-1 can use Memory Bank1. By

reusing memory banks in this way, we can reduce the memory requirement. These steps are repeated until processing the entire frame is completed.

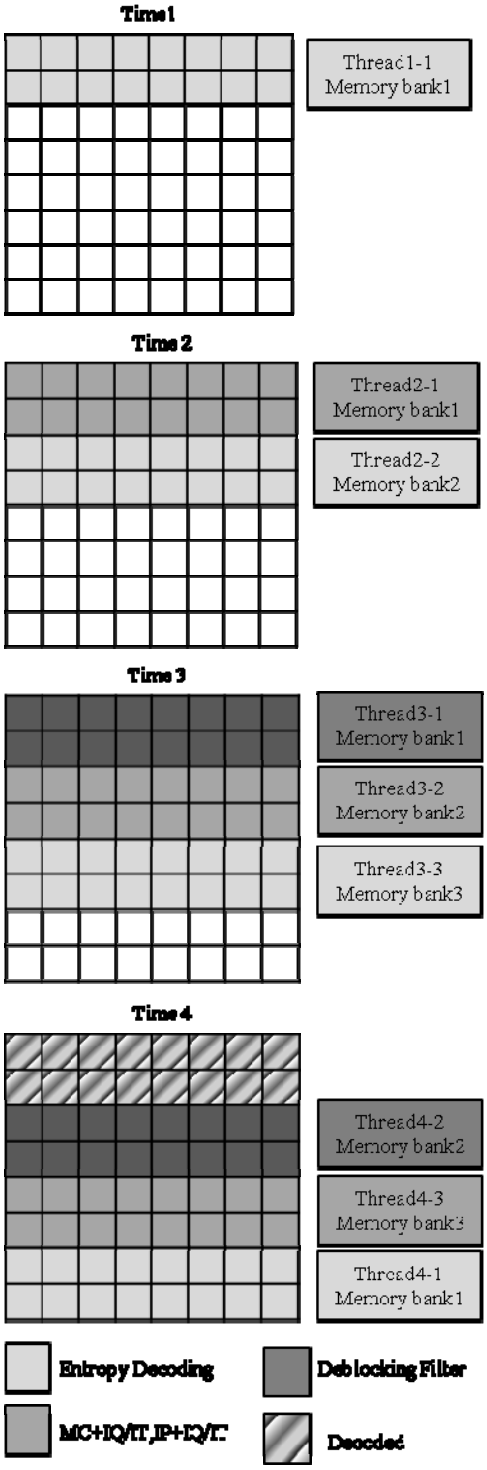

Fig. 11. Utilization of memory in SFPP.

Table II summarizes comparison results of memory requirement for a serial macroblock processing, 2D-Wave pipelining and SFPP. The amount of memory requirement is measured by the memory used by H264Context data structure in FFmpeg H.264/AVC decoder. H264Context contains Sequence Parameter Set (SPS) which holds various video-related information and Picture Parameter Set (PPS) which holds various frame-related information. And output of the entropy decoding is stored in the H264Context. Without parallelization, decoding will be performed for each macroblock, and the total amount of required data memory is 1,888 bytes which is the size of one marcoblock. The size of memory for H264Context data structure is 174,176 bytes. In case of 2D-Wave, the size of memory required for macroblocks will be equal to the total size of macroblocks in one frame, and the corresponding size will be 6,970,976 bytes and 15,580,256 bytes for HD and FHD, respectively. In SFPP, memory banks are reused. Hence, the memory requirement is much less than that of 2D-Wave. In the current implementation of SFPP, we use three different memory banks. When we partition a frame into a group of macroblocks with a size of 400 in HD, the memory requirement is 2,439,776 bytes, which is 65% less than that for 2D-Wave. We could reduce the memory requirement by 81% for FHD compared with 2D-Wave.

TABLE II
COMPARISON ON MEMORY REQUIREMENTS

	HD,1280X720, 3600 MB (byte)		
	Basic	2D wave	SFPP
Used MB(num)	1	3,600	400x3
Memory of MB(byte)	1,888x1 = 1,888	1,888x3,600 = 6,796,800	1,888x400x3 = 2,265,600
Memory of H264Context (byte)	174,176	6,970,976	2,439,776

	FHD,1920X1088, 8160 MB (byte)		
	Basic	2D wave	SFPP
Used MB(num)	1	8,160	480x3
Memory of MB(byte)	1,888x1 =1,888	1,888x8,160 =15,406,080	1888x480x3 =2,718,720
Memory of H264Context (byte)	174,176	15,580,256	2,892,896

IV. EXPERIMENTAL ENVIRONMENTS AND RESULTS

A. Experimental environments

To verify the effectiveness of SFPP, we parallelize FFmpeg H.264/AVC decoder to compare the performance. Currently FFmpeg H.264/AVC decoder is commonly used, and a popular video player, FFplay can play video decoded by FFmpeg. Video samples used in this experiment are encoded by JM-v16, and the encoding environment is based on the baseline profile provided by JM-v16. Linux Ubuntu 9.10 with kernel 2.6.31 on an Intel Quad-Core i5 processor was used as the execution platform. GCC v4.4.1 was used with OpenMP for parallelization.

TABLE III
H.264/AVC DECODING TIME BEFORE PARALLELIZATION

Sequence		Entropy decoding (μs/frame)	MC+IQ/IT IP+IQ/IT (μs/frame)	Deblocking filter (μs/frame)	AVG (μs/frame)	MIN (μs/frame)
rush_hour	FHD, 1920X1088	3929	6567	5699	16195	14746
blue_sky	FHD, 1920X1088	4184	5863	5514	15561	13694
pedestrain_area	FHD, 1920X1088	4295	6800	6440	17535	13498
sunflower	FHD, 1920X1088	3905	6301	3130	13336	11034
park_run	HD, 1280X720	5150	3926	3714	12790	10687
mobcal	HD, 1280X720	2310	3458	2564	8332	3882
stockholm	HD, 1280X720	2029	2993	2190	7212	6640
shields	HD, 1280X720	2211	2997	2210	7418	5707

TABLE IV
RESULTS OF MINIMUM EXECUTION TIMES PER FRAME OF PARALLELIZATION OF H.264/AVC DECODING

Sequence		Pipeline		2Dwave(2)		2Dwave(3)		SFPP	
		(μs/frame)	(%)	(μs/frame)	(%)	(μs/frame)	(%)	(μs/frame)	(%)
rush_hour	FHD, 1920X1088	11873	19%	12105	18%	9998	32%	6993	53%
blue_sky	FHD, 1920X1088	11018	20%	11470	16%	9558	30%	6735	51%
pedestrain_area	FHD, 1920X1088	12126	10%	10666	21%	9604	29%	8431	38%
sunflower	FHD, 1920X1088	9481	14%	9616	13%	8507	23%	6718	39%
park_run	HD, 1280X720	7813	27%	8482	21%	7441	30%	5580	48%
mobcal	HD, 1280X720	2655	32%	3154	19%	2678	31%	2172	44%
stockholm	HD, 1280X720	4996	25%	5395	19%	4563	31%	3702	44%
shields	HD, 1280X720	5046	12%	4689	18%	4160	27%	3528	38%

TABLE V
RESULTS OF AVERAGE EXECUTION TIMES PER FRAME OF PARALLELIZATION OF H.264/AVC DECODING

Sequence		Pipeline		2Dwave(2)		2Dwave(3)		SFPP	
		(μs/frame)	(%)	(μs/frame)	(%)	(μs/frame)	(%)	(μs/frame)	(%)
rush_hour	FHD, 1920X1088	14309	12%	13390	17%	12130	25%	9236	43%
blue_sky	FHD, 1920X1088	13914	11%	13159	15%	11780	24%	8477	46%
pedestrain_area	FHD, 1920X1088	15784	10%	14264	19%	13498	23%	10875	38%
sunflower	FHD, 1920X1088	12023	10%	11572	13%	10580	21%	8822	34%
park_run	HD, 1280X720	11025	14%	10709	16%	10045	21%	6994	45%
mobcal	HD, 1280X720	7403	11%	6938	17%	6387	23%	5458	34%
stockholm	HD, 1280X720	5865	19%	6126	15%	5713	21%	4802	33%
shields	HD, 1280X720	6978	6%	6209	16%	5637	24%	4950	33%

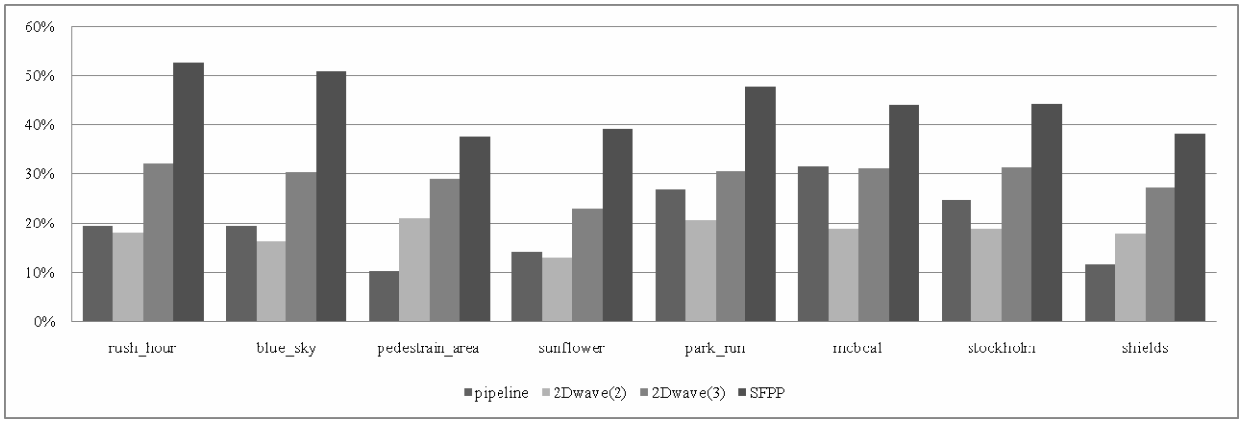

Fig. 12. Minimum execution times per frame of parallelization of H.264/AVC decoding in graphic form.

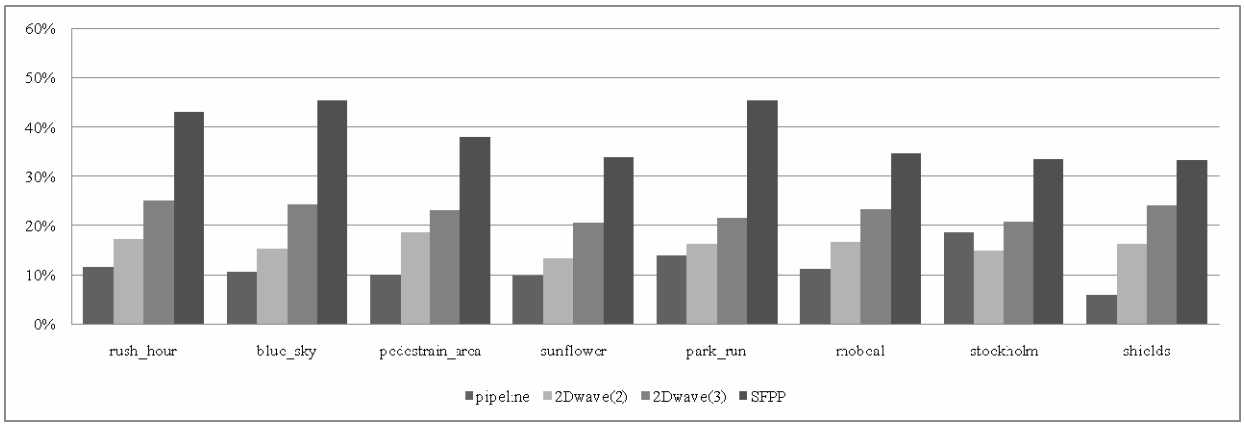

Fig. 13. Average execution times per frame of parallelization of H.264/AVC decoding in graphic form.

B. Experimental Results

We compared performance of H.264/AVC decoding before and after we parallelize the decoder. We measured the minimum and the average decoding time per frame. The results after running without any parallelization technique were summarized in Table III. From the results, we observe that the processing time varies depending on videos, and the execution time for an individual subtask is different. Table IV shows performance comparison results of the minimum decoding time per frame for various parallelization techniques. Table V shows performance comparison results of the average decoding time per frame for various parallelization techniques. Pipeline is a task-level pipelined parallelization technique. 2D-Wave(2) is a parallelization technique with two threads, and 2D-Wave(3) is a parallelization technique with three threads. And SFPP is the proposed technique of this paper.

Compared with the implementation without parallelization, in terms of the minimum execution time per frame, Pipeline showed 12~32% improvement, 2D-Wave(3) showed 23~32% improvement, but the proposed SFPP showed 38~53% performance improvement. In terms of the average execution times per frame, Pipeline showed 6~19% improvement, 2D-Wave(3) showed 21~25% improvement, but the proposed SFPP showed 33~46% performance improvement. Fig. 12 shows a chart for the comparison results in Table IV, and Fig. 13 shows a chart for the comparison results in Table V. From the experimental results, we conclude that our proposed technique is consistently and significantly better than other parallelization techniques.

V. CONCLUSION

In this paper, we propose a novel technique called SFPP for parallel H.264/AVC decoding for high video resolutions. Compared with conventional parallelization techniques, SFPP is better in the sense that it has less synchronization overhead since we process a partitioned frame instead of macroblocks individually. Also we allocate an independent thread to each pipeline stage. Also SFPP uses memory much less than popular methods such as 2D-Wave. We carried out experiments on an Intel Quad-Core system with frequently used H.264/AVC decoder. As a future work, we want to extend this idea to heterogeneous multi-core systems.

ACKNOWLEDGMENT

This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2010-C1090-1031-0009).

REFERENCES

[1] ITU-T Recommendation H.264, SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS Infrastructure of audiovisual services-Coding of moving video, May 2003.

[2] ISO, Information Technology-Coding of Audio-Visual Objects, Part10—Advanced Video Coding, ISO/IEC 14496-10.

[3] Thomas Wiegand, Gary J. Sullivan, Gisle Bjøntegaard, and Ajay Luthra, Senior Member, "Overview of the H.264/AVC Video Coding Standard," IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, no. 7, pp. 560-576, July 2003

[4] Michael Horowitz, Anthony Joch, Faouzi Kossentini, and Antti Hallapuro, "H.264/AVC Baseline Profile Decoder Complexity Analysis," IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, no. 7, pp. 704-716 July 2003.

[5] Y.-K. Chen, X. Tian, S. Ge, and M. Girkar, "Towards Efficient Multi-level Threading of H.264 Encoder on Intel Hyperthreading Architectures," in Proceedings International Parallel and Distributed Processing Symposium, Apr 2004.

[6] Zhuo Zhao and Ping Liang, "A Highly Efficient Parallel Algorithm for H.264 Video Encoder," Proc. Of the 2006 IEEE International Conference on Acoustics, Speech and Signal Processing, vol. 5, 14-19 May 2006 pp. 489-492

[7] Y. Chen, E. Li, X. Zhou, and S. Ge, "Implementation of H.264 Encoder and Decoder on Personal Computers," Journal of Visual Communications and Image Representation, vol. 17,2006.

[8] T. Jacobs, V. Chouliaras, and D. Mulvaney, "Thread-parallel mpeg-2, mpeg-4 and h.264 video encoders for soc multiprocessor architectures," IEEE Trans. on Consumer Electronics, vol. 52, no. 1, pp. 269–275, Feb. 2006.

[9] Shu-Sian Yang, Sung-Wen Wang, Hong-Ming Chen, and Ja-Ling Wu, "A Parallelism Encoding Framework for The Temporal Scalability of H.264/AVC Scalable Extension," Proceedings of IEEE Workshop on Scalable Video Coding & Transport, December 2007

[10] E. van der Tol, E. Jaspers, and R.Gelderblom, "Mapping of H.264 decoding on a multiprocessor architecture," Image and Video Communications and Processing, pp.707-718, May, 2003.
[11] Seung-Won Jung, Yeo-Jin Yoon, Haechul Choi, Aldo W. Morales,Sung-Jea Ko, "A novel post-processing algorithm for parallel-decoded U-HDTV video sequences," IEEE Trans. on Consumer Electronics, vol. 55, no. 1, pp.185-190,Feb.2008.
[12] M. Roitzsch, "Slice-Balancing H.264 Video Encoding for Improved Scalability of Multicore Decoding," in Work-in-Progress Proceedings of the 27th IEEE, 2006
[13] Klaus Schömann, Markus Fauster, Oliver Lampl, Laszlo Böszörmenyi, "An Evaluation of Parallelization Concepts for Baseline-Prole Compliant H.264/AVC Decoders," in Lecture Notes in Computer Science. Euro-Par 2007 Parallel Processing, August 2007.
[14] J. Chong, N. R. Satish, B. Catanzaro, K. Ravindran, and K.Keutzer,"Effcient parallelization of h.264 decoding with macro block level scheduling," in 2007 IEEE International Conference on Multimedia and Expo, July 2007.
[15] Kosuke Nishihara, Atsushi Hatabu, Tatsuji Moriyoshi, "Parallelization of H.264 video decoder for embedded multicore processor," In Proceedings of ICME'2008. pp.329~332
[16] Kue-Hwan Sihn, Hyunki Baik, Jong-Tae Kim, Sehyun Bae, Hyo Jung Song, "Novel approaches to parallel H.264 decoder on symmetric multicore systems," IEEE International Conference on Acoustics, Speech and Signal Processing, 2009
[17] A. Azevedo, C. Meenderinck, B. Juurlink, A. Terechko, J. Hoogerbrugge, M. Alvarez, and A. Rammirez, "Parallel H.264 Decoding on an Embedded Multicore Processor," in Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers -HIPEAC, Jan 2009.
[18] Subbarao Palacharla, Norman P. Jouppi and James E.Smith. "Complexity-Eective Superscalar Processors," In 24th International Symposium on Computer Architecture, pp. 206-218, June 1997.
[19] Tsung-Han Tsai, Chun-Nan Liu, Jui-Hong Hung, "VLIW-Aware Software Optimization of AAC Decoder on Parallel Architecture Core DSP (PACDSP) Processor," IEEE Trans. on Consumer Electronics, vol. 54, no. 2, pp.933-939, May 2008.
[20] Chunhua Liao, Zhenying Liu, Lei Huang, and Barbara Chapman. "Evaluating OpenMP on Chip MultiThreading Platforms," In First international workshop on OpenMP, Eugene, Oregon USA, June 2005.
[21] Matthew Curtis-Maury, Xiaoning Ding, Christos Antonopoulos, Dimitrios Nikolopoulos, D. "An evaluation of OpenMP on current and emerging multithreaded/multicore processors," In Proceedings of the First International Workshop on OpenMP (IWOMP), Eugene, Oregon USA, June 2005.

BIOGRAPHIES

Won-Jin Kim received the BS in mechanical engineering with a minor in Electronic engineering from Hanyang University, Ansan, Korea in 2002, and an MS in Electronic Engineering from Hanyang University, Seoul, Korea in 2008.

He was an Engineer at SENA Corp. in Seoul from 2002 to 2004, and was an Engineer at MGAME Corp. in Seoul from 2004 to 2006. Since 2009, he has been taking up a PhD course at Hanyang University, Seoul, Korea. His research interests include low power embedded system design, image processing, parallelization, and multi-core architecture.

Keol Cho received his BS in Media Communication Engineering from Hanyang University, Seoul, Korea in 2009. Since 2009, he has been taking up an MS course at Hanyang University, Seoul, Korea.

His research interests include low power embedded system design, parallelization, image processing, and embedded multi-core architecture.

Ki-Seok Chung received his BE in Computer Engineering from Seoul National University, Seoul, Korea in 1989, and PhD in Computer Science from University of Illinois at Urbana-Champaign in 1998.

He was a Senior R&D Engineer at Synopsys, Inc. in Mountain View, CA from 1998 to 2000, and was a Staff Engineer at Intel Corp. in Santa Clara, CA from 2000 to 2001. He also worked as an Assistant Professor at Hongik University, Seoul, Korea from 2001 to 2004. Since 2004, he has been an Associate Professor at Hanyang University, Seoul, Korea. His research interests include low power embedded system design, multi-core architecture, image processing, reconfigurable processor and DSP design, SoC-platform based verification and system software for MPSoC.