PH.D. THESIS

Parallelized Implementation of Full High Definition H.264 Decoder on Embedded Multi-core

임베디드 다중 코어에서의 초고화질 H.264 복호기 병렬 구현

BY

MINSOO KIM

FEBRUARY 2012

SCHOOL OF ELECTRICAL ENGINEERING AND

COMPUTER SCIENCE

COLLEGE OF ENGINEERING

SEOUL NATIONAL UNIVERSITY

Abstract

In this paper, we deal with the problem of parallelization in implementing H.264/AVC(Advanced Video Coding) decoder on embedded multi-core system. For this purpose, we suggest a parallelization strategy for embedded multi-core system. The parallelization strategy can be applied to not only H.264/AVC decoder, but also a general application for parallelization on the embedded multi-core system. In addition, we propose two specific parallelization methods called dynamic load balancing and hybrid partitioning. We show the validities of the proposed methods through the implementation of two embedded multi-core platforms for H.264/AVC decoder. One is dual-core system with 3 hardware accelerators, and the other one is quad-core system with 2 co-processors.

On dual-core system, H.264/AVC decoder is parallelized with a few hardware accelerators by the proposed parallelization strategy. For that system, functional partitioning is selected by the proposed parallelization strategy, which enables simple interface with hardware accelerator and small memory usage for inter-core communication. We also propose dynamic load balancing method for the functional partitioning. The load balancing is achieved by mapping a few selected functions to each core dynamically at macroblock level. In this case, buffer level information is enough for making decision which core runs those functions. Because of this simple decision criterion and mechanism, performance loss for load balancing process can be negligible and it is also possible to extend the proposed load balancing method to multi-core systems easily. Experimental result shows that the proposed load balancing method reduces the waiting overhead dramatically and the reduced amount is 82.3% of the total waiting overhead.

For quad-core system, we propose a new partitioning method called hybrid partitioning by adopting the proposed parallelization strategy. Partitioning is a very important issue for the mapping of application software on multi-core systems. In this paper we propose a hybrid partitioning, mixture of functional and data partitioning methods. Each module is partitioned by

functional partitioning or data partitioning depending on the module's features. Compared with functional and data partitioning, the hybrid partitioning is as powerful as data partitioning for load balancing between cores, and it is also as efficient as functional partitioning in the view point of memory requirement. Hybrid partitioning is also free from the macroblock level dependency problem which data partitioning usually has in video decoding. As a result of applying hybrid partitioning, we can reduce 86.0% of waiting overhead compared with functional partitioning. Regarding memory usage, hybrid partitioning requires 51.2% less VLIW (Very Long Instruction Word) program memory and 62.0% less CGRA (Coarse-Grained Reconfigurable Array) program memory than data partitioning. As for SDRAM (Synchronous Dynamic Random-Access Memory) bandwidth, compared with data partitioning, hybrid partitioning uses 11.6% of the whole bandwidth budget of 333MHz SDRAM memory used in experiments

Keyword: Embedded multi-core system, H.264/AVC, parallelization strategy, dynamic load balancing, hybrid partitioning, functional partitioning, data partitioning.

Student Number : 2006-30843

Contents

List of Figures

List of Tables

CHAPTER

1

Introduction and Motivation

Recently, according to widely used smart devices like smartphones, tablet, and etc., multi-core systems with embedded cores of relatively low performance have become more popular for high complexity applications. The reason is low power consumption of the multi core system compared to systems with a single high performance core. Due to intensive competition between manufacturers of the smart devices, they require more complex and powerful applications. Examples of these applications are multi-standards video decoders like HD (High Definition, 1280x720 pixels) or FHD (Full High Definition, 1920x1080 pixels). Regarding video coding standards H.264[9] is the most advanced, although most complex, one.

Although the power efficiency of a multi core system is superior to a single core system, development of an efficiently parallelized software on multi-core system is much more difficult than that on single core system. Many software developers are suffer from developing efficient parallelization of a given application including H.264/AVC decoder [15]. The difficulty level of parallelization of a given application varies a lot depending on the feature of the application. In case of H.264/AVC decoder, the main obstacle to

parallelization is data dependency and wide variation of each function's complexity.

To overcome those problems, many approaches like [4], [5], [6], [7], [8], [9], [10], [11], [16], [20], [21], [22], [23], [28], [29], [30], [31], [32], and [34] have been proposed and applied for parallel H.264/AVC decoders. The major approaches for this purpose are functional partitioning and data partitioning. The difference of two partitioning methods is described well in [4]. The functional partitioning means that all functions of a given application are distributed to different cores, therefore all cores do different tasks. In case of data partitioning, all cores execute the same task with different data [4]. Data partitioning is suitable for handling load balancing on each core because all cores perform the same job and each core processes the data for different macroblocks (MBs) depending on the load of each core. But the problem with data partitioning is synchronization between threads due to complex dependencies of MBs in H.264/AVC standard. In addition, a large amount of memory in proportion to picture size is required for synchronization and communication between cores. If the buffers for this data are in SDRAM, accessing those buffers usually consume a large amount of SDRAM bandwidth. It also requires much more program memory than functional partitioning because all cores must have all functions of the video decoder [4]. Functional partitioning, on the other hand, has no problem with synchronization and requires low memory usage compared with data partitioning [4]. However it presents issues on load balancing.

In addition to the problems of each partitioning type, another obstacle against the parallelization of H.264/AVC decoder is a very wide variation of each multi-core system. The parallelization methods should be adaptive depending on the configuration of a multi-core system.

In this paper, we suggest a parallelization strategy for H.264/AVC decoder applicable to any kind of multi-core system. In addition to the proposed parallelization strategy, we also suggest two specific parallelization

method called dynamic load balancing and hybrid partitioning to overcome the problems of each partitioning type. The proposed dynamic load balancing method can resolve a problem of unbalanced load on functional partitioning and the proposed hybrid partitioning can resolve the complex synchronization and huge memory requirement of data partitioning

To show validity of the proposed parallelization method, we developed two platforms. One is dual-core system with 3 H/W accelerators, and the other one is quad-core system with 2 co-processors. In dual-core system, there are 2 embedded DSP (Digital Signal Processor) cores called RP(Reconfigurable Processor)[13] and 3 H/W accelerator for VLD(Variable Length Decoding), MC(Motion Compensation) and deblocking. In quad-core system, there are 4 RP and two co-processors called BSPU (Bit-Stream Processing Unit) and MemPU (Memory Processing Unit). The role of two co-processors will be explained at Section 3. This platform can be provided as an IP (Intellectual Property) module for silicon chips and can be part of an AP (Application Processor) or other processors.

The objective of this paper is to find a parallelization strategy for efficient implementation of an application on embedded system including H.264/AVC decoder, and to show the validity of the proposed methods through implementation and performance evaluation on the two types of multi-core systems. Especially, we will find an efficient mapping of the H.264 video decoder on the developed dual-core and quad-core systems. For this purpose, we will consider an effective partitioning method with efficient load balancing, low memory usage, and minor SDRAM (Synchronous Dynamic Random-Access Memory) bandwidth usage.

CHAPTER
2

Review on H.264/AVC Standard

In this section, we briefly review on H.264/AVC standard which is used for validation of the proposed parallelization methods. H.264/AVC is one of the most frequently used video coding standards because it is one of the most efficient video coding standards. It is also one of the most complex video coding standards to achieve the high efficiency. H.264/AVC is the video coding standard of the ITU-T video coding expert group and the ISO/IEC moving picture expert group. The main goal of the H.264/AVC video coding standard is providing more advanced coding efficiency than the previous coding standard like MPEG2, H.263, and etc. Compared with MPEG2, it used about 50% less data than MPEG2 for the same video quality [14]. This means that H.264 is twice more efficient than MPEG2.

Figure 1 shows the block diagram of H.264/AVC decoder. The input for this decoder is H.264 bitsream. The first step of the decoder is entropy decoding with variable length code or arithmetic code. The next steps are inverse quantization (IQ) and inverse transform (IT) using the output of the entropy decoder. The inverse quantization is also called de-quantization (DQ).

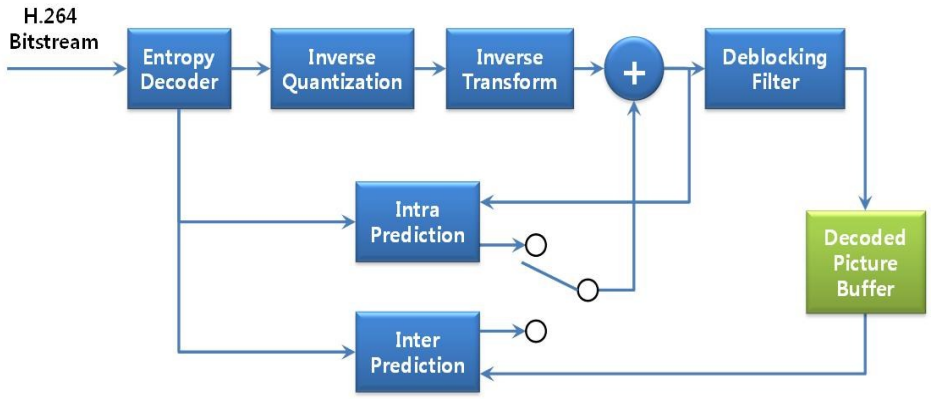


Figure 1: H.264/AVC Decoder Block Diagram [4]

The result of inverse quantization is transformed inversely to generate residual data by IT module. These residual data are added to the output of a prediction function. There are two kinds of the prediction type. One is an intra prediction, and the other is an inter prediction. The intra prediction uses neighboring pixel data in the same frame to generate predicted pixel. The inter prediction uses previously decoded frame called reference frame for the prediction of pixel data or motion compensation (MC). The last function of the H.264/AVC decoder is a deblocking filter. The role of the deblocking filter is smoothing the boundary of macroblocks or sub-blocks. The output data of the deblocking filter are stored in frame buffer called DPB(Decoded Picture Buffer) for display and inter prediction.

To achieve the high compression efficiency, H.264/AVC coding standard became much more complex than the previous coding standards like MPEG2, H.263, and etc. from the view point of the computation [4]. For better coding efficiency, H.264/AVC standard adopts much more complex entropy coding scheme called CABAC(Context-based Adaptive Binary Arithmetic Code) in addition to CAVLC(Context-based Adaptive Variable Length Code). It also adopts much more prediction scheme compared to MPEG2 standard. It also

uses smaller block size like 8x4, 4x8, or 4x4 for inter prediction. The inter prediction uses arbitrary number of blocks for composing single macroblock with 16x16, 16x8, 8x16, 8x8, 8x4, 4x8, and 4x4 in the case of luma pixel data. This means that the maximum number of motion vector for single macroblock is 32 in the case of bi-directional or B macroblock. These kinds of small blocks can handle complex motions of small objects in a video frame in the cost of high computational complexity. The effect of the small block size is 15% reduction of bitrate compared to only 16x16 block size [4].

As for motion vector, 1/4 pixel precision is used instead of 1/2 pixel precision of MPEG2. In addition to more accurate pixel precision, filter for inter prediction became higher order filter than the previous coding standards. It adopts 6-tap filter for inter prediction with the motion vector of 1/4 pixel accuracy. These higher order filter and more accurate motion vector increase decoder complexity much more than MPEG2.

The maximum number of reference frames is 2 for bi-directional frame in the case of MPEG2. In contrast to MPEG2 coding standard, H.264/AVC standard can use 15 different reference frames. It results in 15% bitrate reduction in the cost of the decoder complexity and large memory for reference frames.

H.264/AVC also adapts in-loop deblock filter to reduce blocky artifact between different blocks. The intensity of deblock filter depends on the parameter called boundary strength. The boundary strength indicates a level of difference between adjacent blocks, and it depends on the difference of motion vectors and reference indexes between those blocks.

CHAPTER

3

Design Strategy for H.264/AVC Decoder on Multi-core System

After the prediction of Gorden E Moore, the number of transistors on integrated circuit doubles approximately every 18 months [17]. With this increment of the number of transistors, operating frequency of the system also has been increased continuously. This improvement of the operating frequency is hitting the wall and the increasing speed of processing frequency is slowing down due to the limitation of power consumption and the shrinking of wire connections [15]. To overcome these problems, processor designers adopt multi-core system on a single silicon chip. Multi-core processor is a kind of innovation in processor architecture, but the introduction of multi-core processor imposes new challenges on software developers [15].

To migrate the previously developed applications from single core architecture to multi-core architecture, software developers should analyze the parallelism inside the applications and find suitable parallelization methods for those applications [15]. In this chapter, we explore various parallelization strategies for implementation of parallel application software on an embedded multi-core systems. Then we also explore various proposals for parallelization of H.264/AVC decoder. Finally, we propose a novel strategy for

parallelization of application software on the embedded multi-core system considering system configuration and features of the given applications.

3.1 MASTER/SLAVE MODEL AND DATA FLOW MODEL

For a task level parallelism inside a given application software, two dominant models can be used: one is master/slave model in which one core controls the work assignment on all other cores, and the other is data flow model in which the work flows through processing stage as in a pipeline [15]. Figure 2 shows master/slave model and Figure 3 shows data flow model. Another name of the master/slave model is walker/server model.

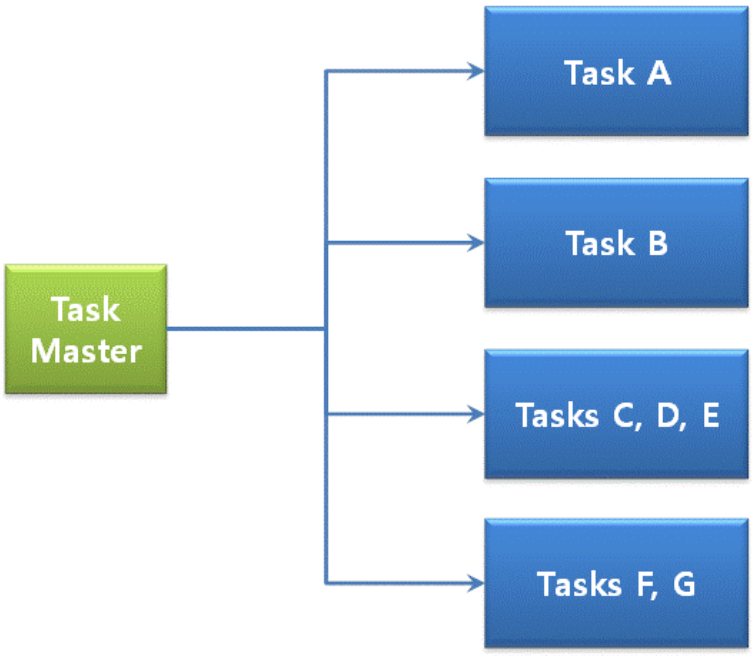


Figure 2: Master/Slave Model [15]

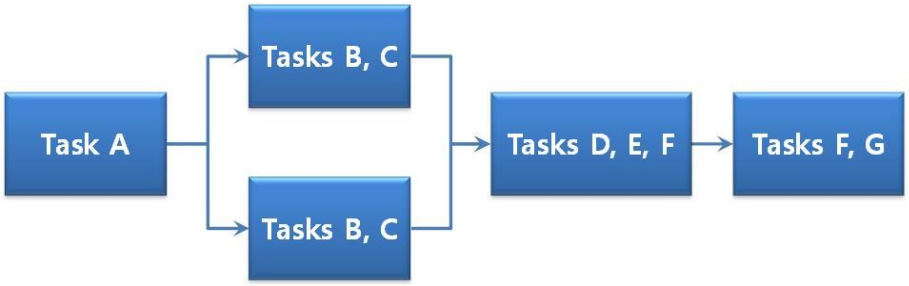


Figure 3: Data Flow Model [15]

3.2 FUNCTIONAL PARTITIONING AND DATA PARTITIONING

There is another categorization to classify multi-core mapping methods depending on task level parallelism or data level parallelism. One is functional partitioning and the other one is data partitioning. The functional partitioning means that all functions of the application are distributed to different cores, therefore all cores do different task. In the case of data partitioning, all cores execute the same task with different data

Figure 4 shows a general application consists of 3 tasks with input data on single core system. Each input data is processed by each task sequentially. Usually data 1 is processed at first by 3 tasks, then data 2 and data 3 are processed sequentially. All tasks are executed on single core one by one.

Figure 5 shows a functional partitioning of previous application on 3 core system. Core 1 executes task 1, and core 2 executes task2, and core 3 executes task 3 with all input data. Usually data are processed in pipelined manner in functional partitioning. Therefore core 3 executes task 3 with data 1 during core 2 executes task 2 with data 2 and core 1 executed task 1 with data 3. Figure 6 shows the data partitioning of the previous application on 3 cores

system. Each core executes same tasks with different input data. Core 1 executes task 1, 2, and 3 with input data 1. Core 2 and core 3 execute same tasks with data 2 and data 3 each.

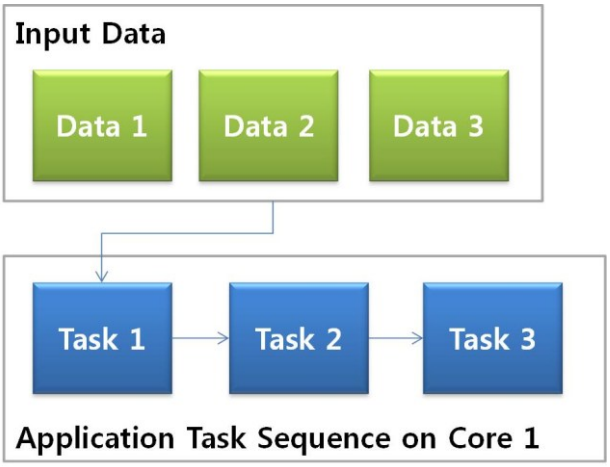


Figure 4: Data sequence and task on single core system

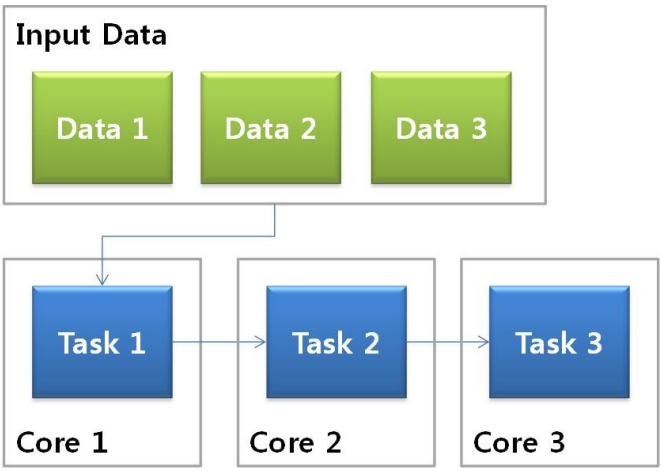


Figure 5: Functional partitioning on multiple core

As can be seen in Figure 5 and Figure 6, data partitioning is similar to master/slave processing model, and functional partitioning is similar to data flow model.

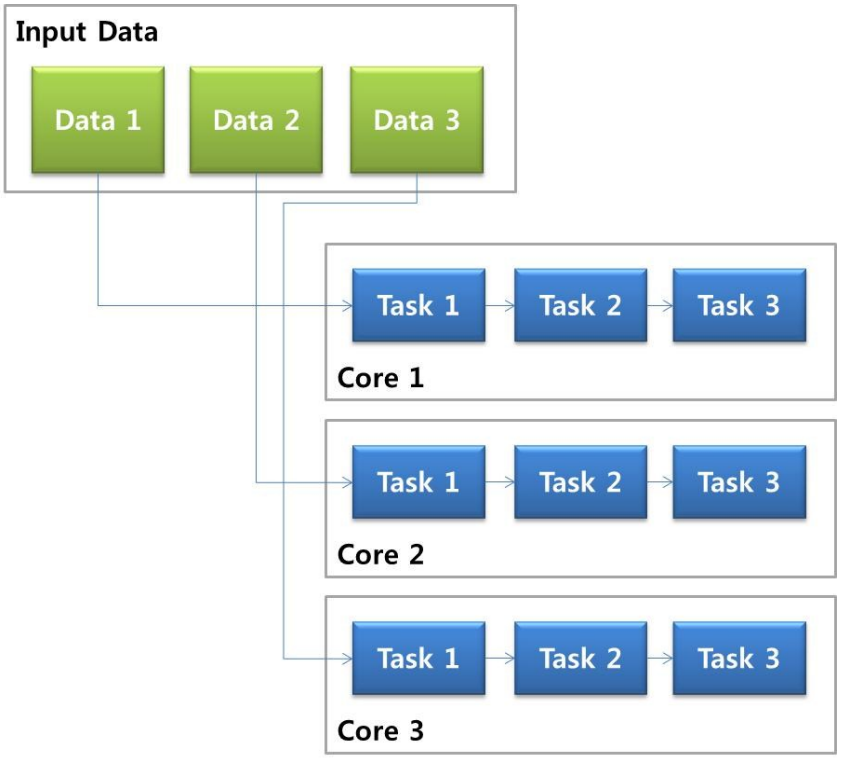


Figure 6: Data Partitioning on multiple core

3.3 RELATED WORK FOR PARALLELIZATION OF H.264 DECODER

Many approaches have been proposed for parallelizing the H264/AVC decoder on multi-core systems like [4], [5], [6], [7], [8], [9], [10], [11], [16],

[20], [21], [22], [23], [28], [29], [30], [31], [32], and [34]. In those proposals, functional partitioning method and data partitioning method are usually used. In addition to two major partitioning methods, another approach called ring model was also proposed [16].

The difference between the two methods on the H.264/AVC decoder was described in [4]. Data partitioning is suitable for handling load balancing on each core because all cores perform the same job and each core processes the data for different macroblocks (MBs) depending on the load of each core. But the problem with data partitioning is synchronization between threads due to complex dependencies of MBs in H.264/AVC standard. In addition, a large amount of memory in proportion to picture size is required for synchronization and communication between cores. If the buffers for this data are in SDRAM, accessing those buffers usually consume a large amount of SDRAM bandwidth. It also requires much more program memory than functional partitioning because all cores must have all functions of the video decoder [4]. Functional partitioning, on the other hand, has no problem with synchronization and requires low memory usage compared with data partitioning. However it presents issues with load balancing.

Many different methods have been proposed because of various multi-core system configurations like number of cores, the architectures of the multi-core systems, memory configurations, and hardware accelerators for H.264/AVC decoding. For hardware accelerator, there can be many accelerators for various functions in H.264/AVC decoder, like motion compensation accelerator, variable length decoder for CABAC and CAVLC, loop filter accelerator, and etc. In most case of the implementations of H.264/AVC decoder on multi-core system, hardware accelerator for variable length decoder is usually used, because variable length decoder cannot be parallelized within slice boundary. In the next section, details of the previous proposals for parallelization of the H.264/AVC decoder will be described in each case of the partitioning.

3.3.1 FUNCTIONAL PARTITIONING OF H.264/AVC DECODER

As described at the previous section about functional partitioning of the general applications, different cores run different tasks in the case of functional partitioning. In the case of the H.264 decoder, usually functional modules in Figure 6 can be candidate functions for functional partitioning. For example, the first core parses H.264/AVC input bitstream with variable length decoder, the second core dequantizes and invers-transforms the result of the first core. The third core calculates prediction pixel using pixel data in the reference frame or neighboring pixel in the current frame depending on prediction mode. The last core runs deblock filtering and saves the result into the decoded picture buffer. To reduce the waiting time of each core, data are processed, in a pipelined manner, macroblock by macroblock in functional partitioning. For example, the first core parses the 4th macroblock data of the current frame, when the second core dequantizes and inverse transforms the 3rd macroblock data.

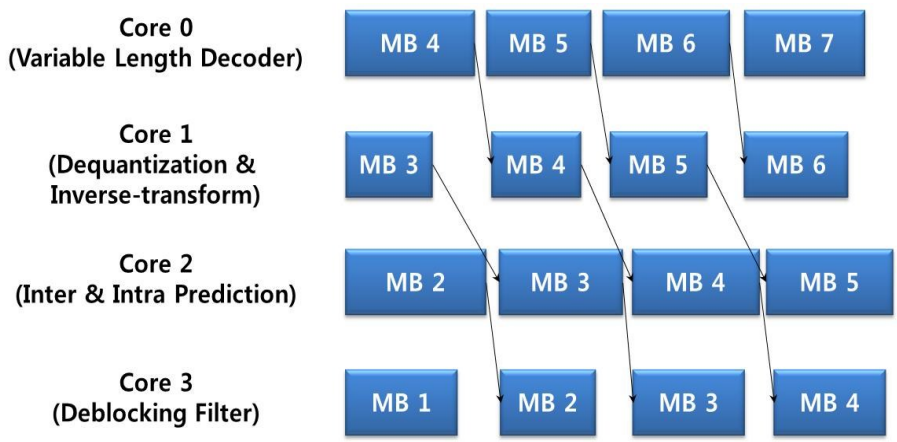


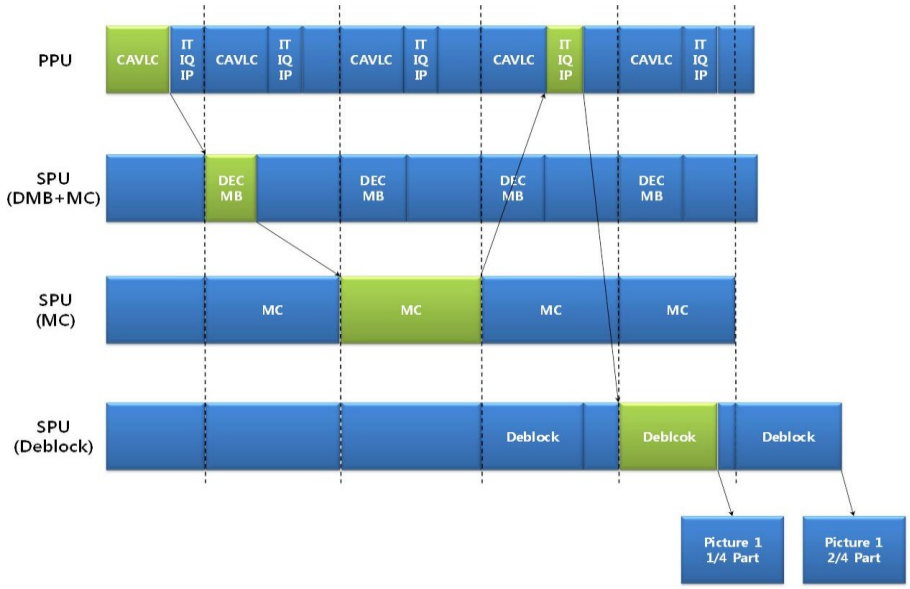
Figure 7: Pipeline of H.264/AVC Decoder with functional partitioning

At the same time, the third core calculates prediction pixel for the 2nd macroblock, and the forth core runs deblock filtering for the 1st macroblock. Figure 7
shows the example of pipelining of H.264 decoder on the functional partitioning.

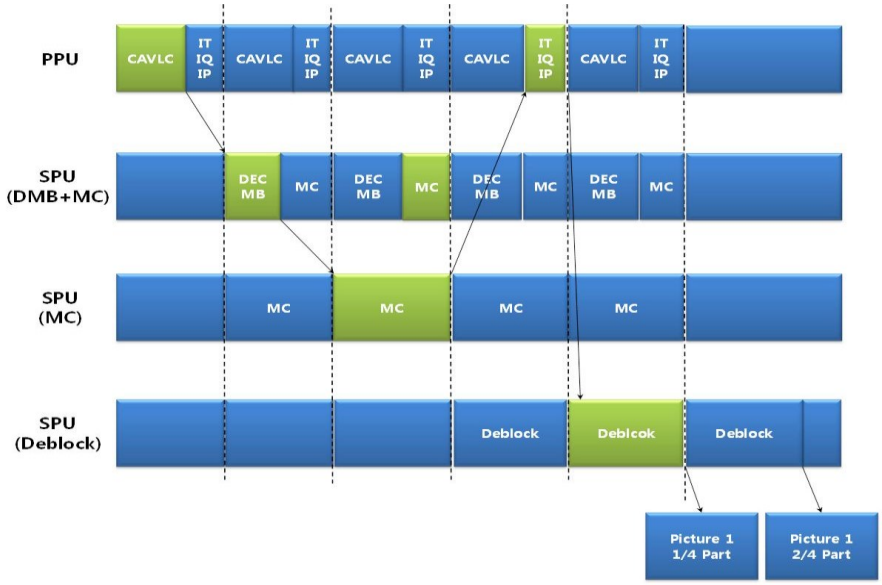
Because each core runs different functions, there are usually differences of the loads of the each core. These unbalanced loads cause waiting overhead to each core statically or dynamically. In spite of the existence of the waiting overhead, functional partitioning makes the synchronization between cores very simple. The synchronization is just waiting the processing of the previous core in functional partitioning. In addition to simple synchronization, it requires less memory usage compared with data partitioning.

As for the implementation of H.264 decoder using functional partitioning, Kim et al. proposed a dynamic load balancing scheme for functional partitioning [6]. Kim et al. used Cell processor[18] for H.264/AVC full high definition decoder. Cell processor has heterogeneous architecture with a 64 bits multi-threaded Power Processor Element(PPE) and eight Synergetic Processor Elements(SPEs)[6]. To avoid complex synchronization problem and to reduce waiting overhead between cores, they adopted a functional partitioning with quarter frame level synchronization. They also adopted dynamic load balancing in frame level to reduce frame level waiting time.

They split the motion compensation function to 2 cores for load balancing. In the case of FHD decoding, quarter frame pipelining requires a large buffer including thousands of macroblocks for inter-core communication. Figure 8 shows the functional partitioning with quarter frame pipelining and dynamic load balancing proposed by Kim et al. Baik et al. proposed hybrid functional partitioning for Cell broadband processor [9] where they used quarter frame pipelining and split the MC function to 3 cores for load balancing like [6].



(a) Normal pipeline structure with idle state



(b) Dynamically load balanced pipeline structure

Figure 8: Functional Partition with Quarter Frame Pipelining [6]

Unfortunately, this caused the same memory requirement problem as [6] because of large data buffer for quarter frame pipelining. In addition to that, it causes load balancing problem between PPE and SPE because 1 PPE, 1 SPE and 3 PPE executes different functions which have different load. Figure 9 shows the workload partitioning, and Figure 10 shows timing diagram of the proposed method by Baik et al.[9] Because of those reasons mentioned previously, it shows only 3.5 faster performance with 5 cores than that of single core.

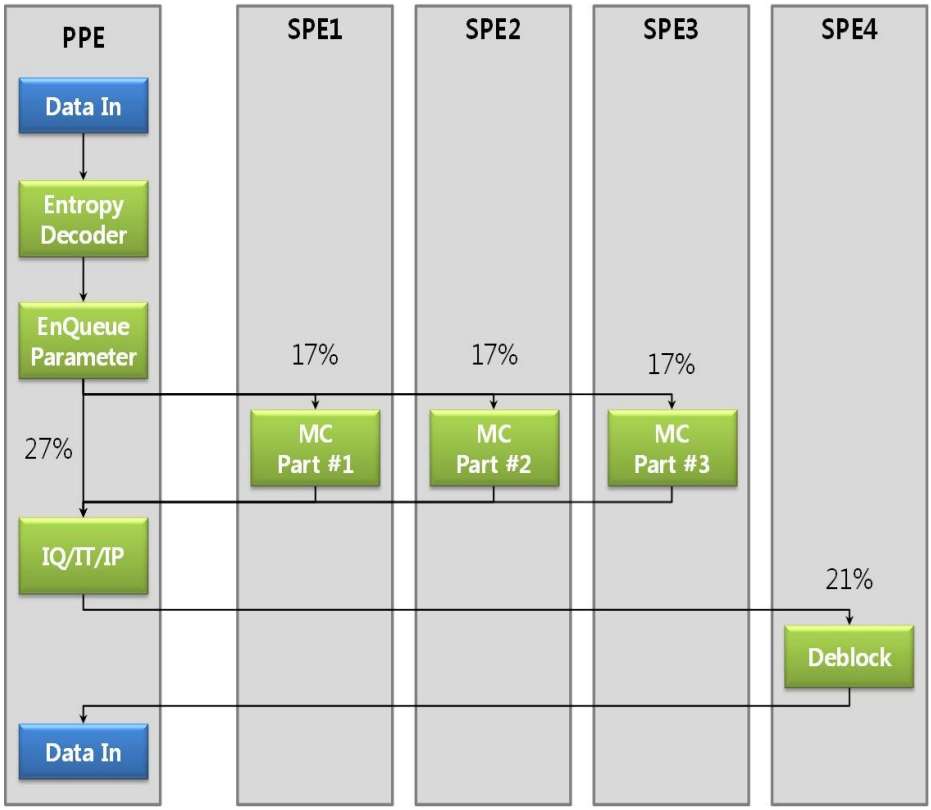


Figure 9: Workload Partitioning[9]

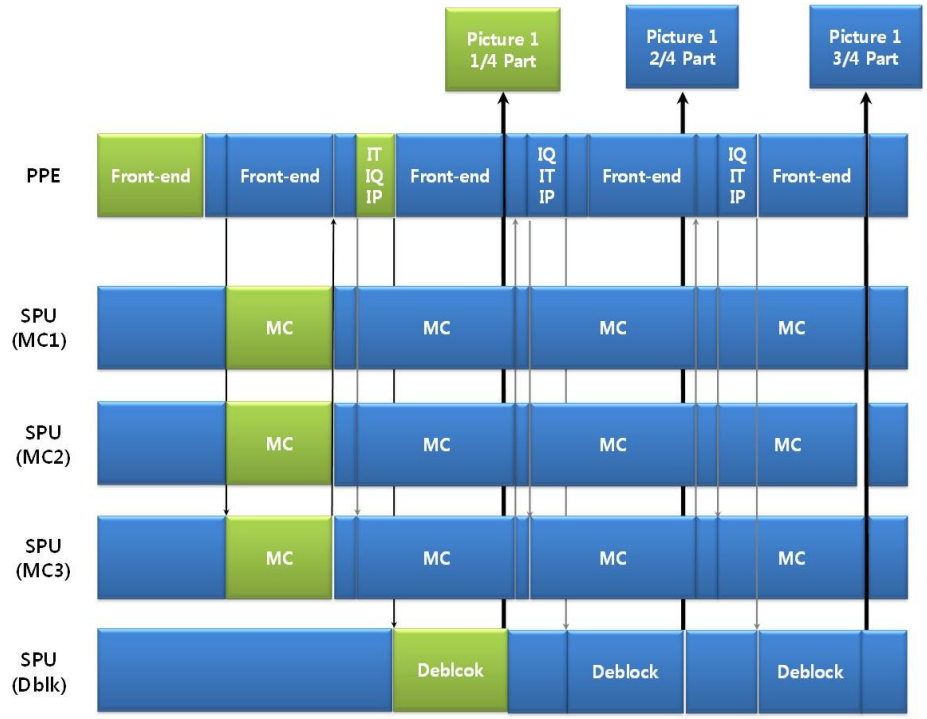


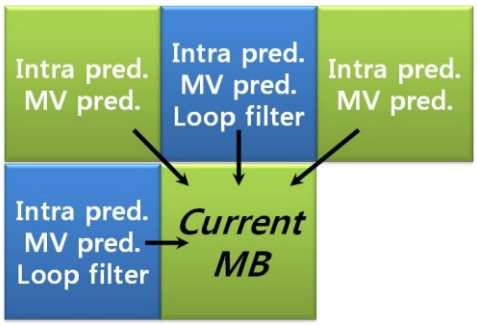
Figure 10: Timing diagram of decoder pipeline [9]

3.3.2 DATA PARTITIONING OF H.264/AVC DECODER

As described at the previous section about data partitioning of the general application, all cores run whole functions of an application, but they process different data. In the case of H.264/AVC decoder, almost all implementations adopt macroblock level data partitioning. Various macroblock level data partitioning methods are proposed for H.264/AVC decoder like [4], [5], [7], [8], and [22] because of complex macroblock level dependency of H.264/AVC coding standard.

Figure 11 shows the macroblock level dependency of the H.264/AVC decoder. There are 3 types of macroblock level spatial dependency inside single frame except variable length code. As for variable length decoder, parallelism inside

slice or frame boundary is basically impossible. Therefore many implementation using data partitioning assign single core or hardware accelerator for variable length decoder without parallelization. In some cases, ideal variable length decoder with unlimited performance is assumed like [5]. 3 types of macroblock level spatial dependency are described well in [4]. The first one is the dependency by intra prediction. In the case of intra prediction, predicted pixels are calculated using neighboring pixels depending on prediction modes. The directions of the predictions are from left to right, from top to bottom, and from top-left to bottom-right. The second one is the dependency by motion vector calculation. In the bitstream of the H.264/AVC, there are only motion vector difference values. Therefore, the motion vector of current block partition should be calculated using a motion vector predictor. The direction of prediction is similar to intra prediction. Therefore, the motion vector prediction is calculated from the motion vectors of the top, left, and top-left block of current block. The third one is the dependency by deblocking filter. Deblocking filter is performed at every 4x4 block boundary from top to bottom direction and from left to right direction. Figure 11 shows the macroblock level data dependencies which exist in H.264/AVC standards.



Intra pred.: pixel prediction for the current MB requires data from this block
MV pred.: motion vector prediction for the current MB requires data from this block
Loop filter: deblocking of the current MB requires data from this block

Figure 11: Spatial data dependencies for a macroblock [4]

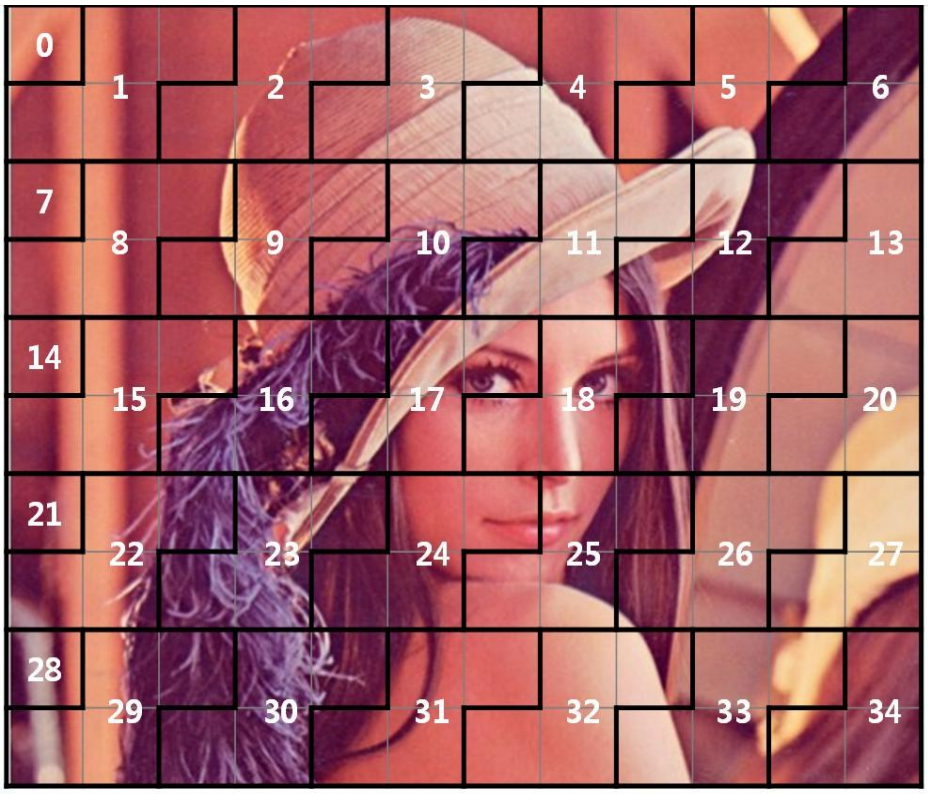


Figure 12: Stairway-shaped data partitioning [4]

To resolve complex macroblock level dependency problem, Van der Tol et al. proposed stairway shaped data partitioning method. Figure 12 shows the detail of the proposed data partitioning and Figure 13 shows the scheduling of decoder. They used 5 processors for H.264/AVC decoder. One processor is assigned to variable length decoder, and the 4 processors run same function using proposed data partitioning. Figure 14 shows the detail of the implementation on 5 processors.

Meenderinck et al. proposed similar data partitioning method [5] with that of Van der Tol et al. They also extended those data partitioning to frame level parallelism. Inside single frame Meenderinck et al. used similar data partitioning methods with [4], but they parallelized H.264/AVC decoder using

macroblock row partitioning considering macroblock dependency. They called this partitioning as 2D-wave approach. Figure 15 shows the 2D-wave proposed by [5].

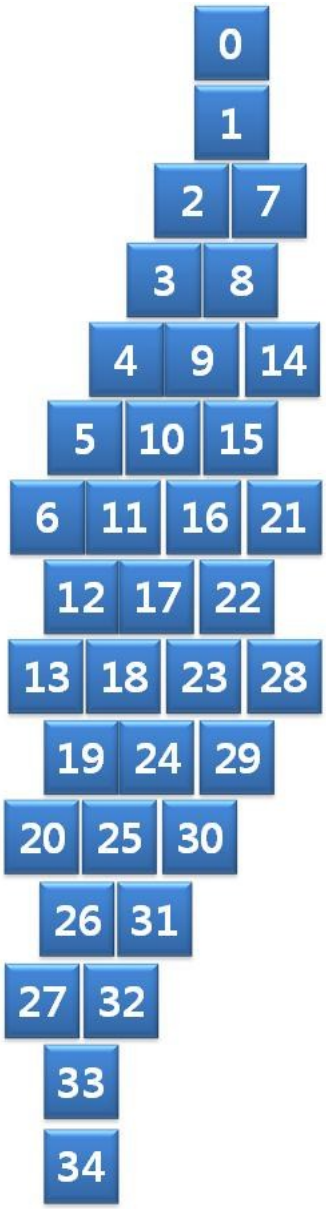


Figure 13: Schedule of decoded data partitions [4]

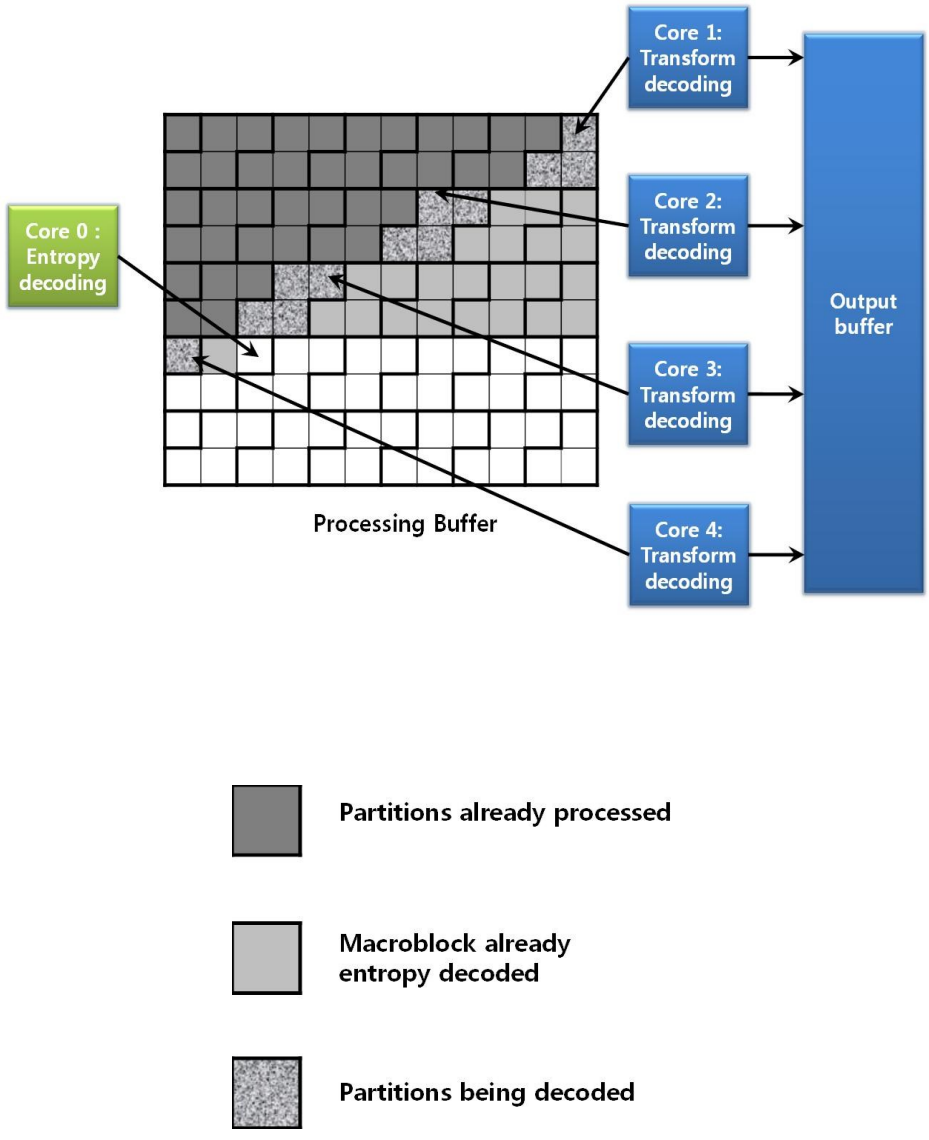


Figure 14: Data partitioning on 5 processors [4]

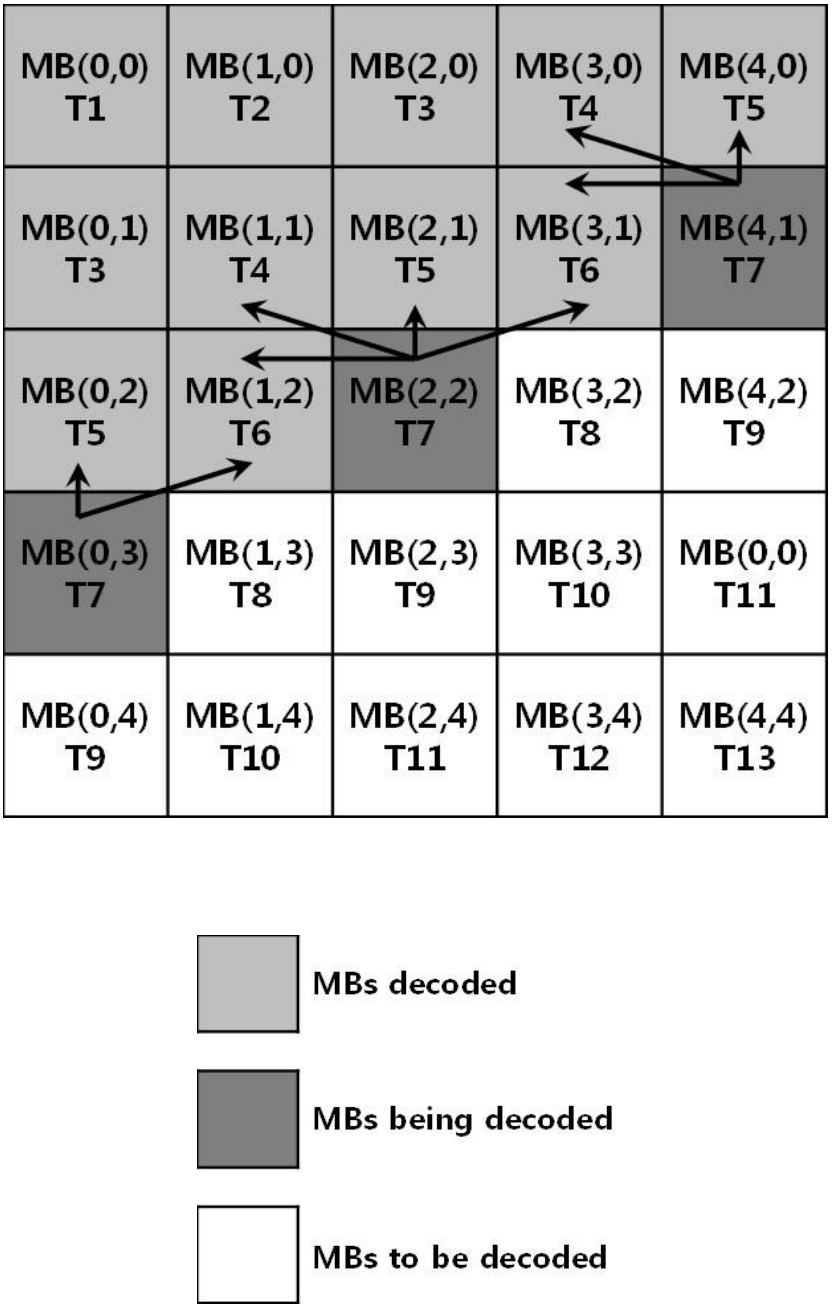


Figure 15: 2D-wave data partitioning. The arrows indicate dependency. [5]

The parallelism level of 2D-wave partitioning depends on the number of macroblock row in single frame. In the case of full high definition, the number of the macroblock rows is 6. Therefore the maximum number of the macroblocks decoded at the same time is 68 by 2D-wave approach. Figure 16 shows the macroblock level parallelism in a single full high definition frame. As can be seen in the graph, maximum parallelism appears during middle of decoding single frame only.

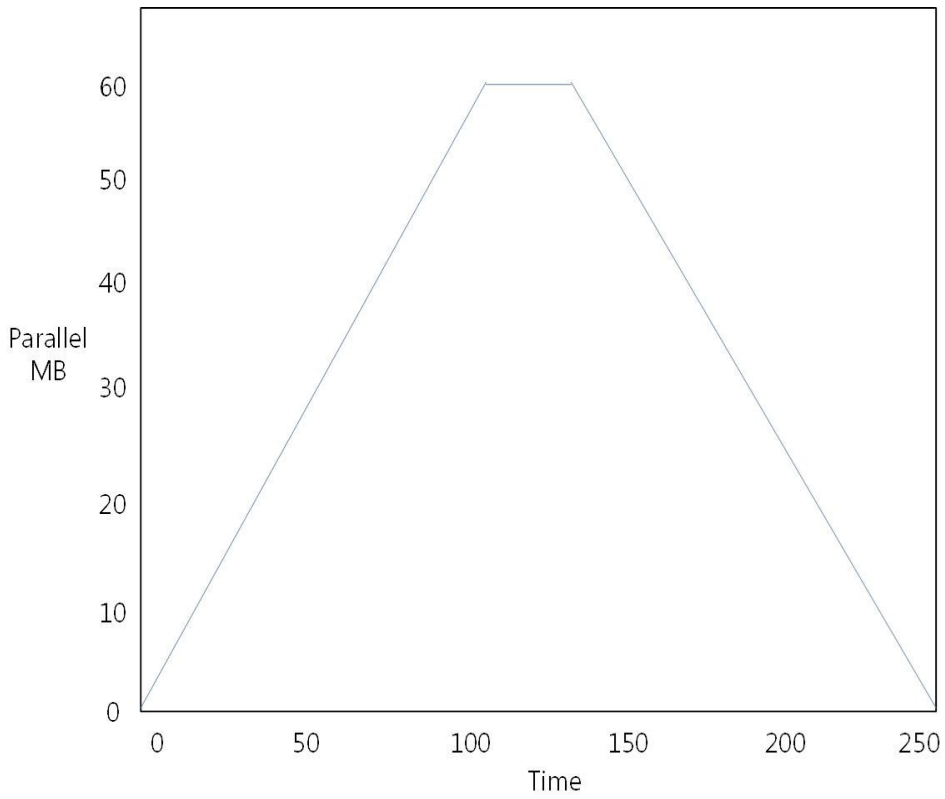


Figure 16: Macroblock parallelism for a single FHD frame using the 2D-Wave approach. [5]

As for variable length code, Meenderinck et al. assumed ideal variable length decoder which has limitless performance. To overcome that limitation of parallelism in 2D-wave approach, they also expended the proposed approach to multiple frames. Between the frames in H.264/AVC, there can be a dependency caused by motion vector calculation in the case of bi-directional frame. The motion vector of a co-located macroblock at the previous reference frame is required for motion vector calculation of the current macroblock. Therefore, the dependency between frames should be considered to expand parallelism beyond frame boundary and the shape of data partitioning should be limited. Figure 17 shows the shape of the extension of 2D-wave called 3D-wave approach.

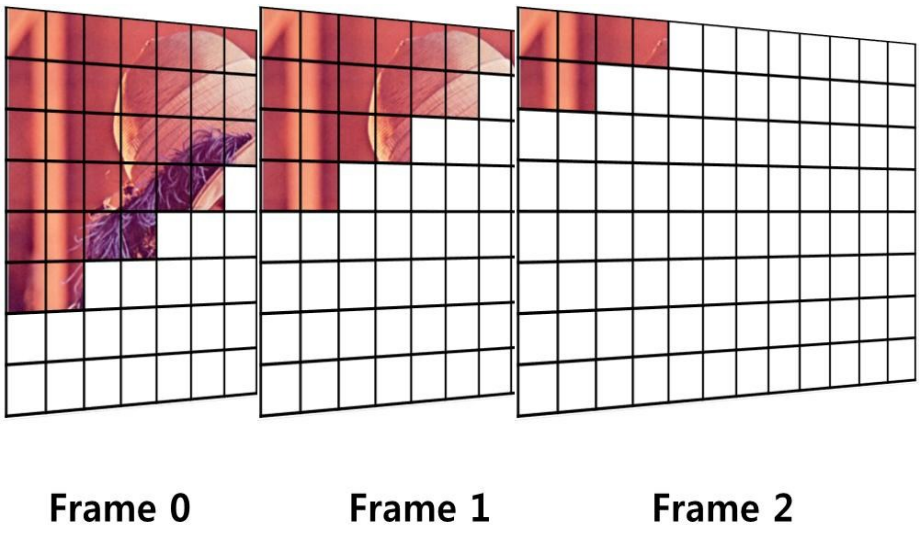


Figure 17: 3D-wave data partitioning. [5]

TABLE I.

MAXIMUM MB PARALLELISM, AND FRAMES IN FLIGHT FOR SD, HD, AND FHD RESOLUTION. ALSO THE AVERAGE MOTION VECTORS (IN SQUARE PIXELS) ARE STATED. [5]

Stream	SD		
	MBs	**Frames**	**Average MV**
rush_hour	1202	93	1,3
riverbed	1944	150	2
pedestrian	1227	104	9,2
blue_sky	1298	97	4,4
Stream	**HD**		
	MBs	**Frames**	**Average MV**
rush_hour	2831	139	1,8
riverbed	4579	228	2,2
pedestrian	2807	151	11
blue_sky	2873	140	5,1
Stream	**FHD**		
	MBs	**Frames**	**Average MV**
rush_hour	6133	218	2,2
riverbed	9169	304	2,6
pedestrian	4851	242	9,9
blue_sky	7327	253	5,6

Table I shows the result of the experiment using 3D-wave approach by [5]. In the case of FHD, maximum value of parallelism is more than 9,000, but this result is the output of simulator with the assumption of ideal variable length decoder. All result of this work is not the output of real implementation, but the output of simulation to evaluate the level of parallelism in H.264/AVC decoder is proposed by Meenderinck et al.[5] Therefore, the performance of variable length decoder in real implementation should limit the performance of the whole decoder. In addition to that limitation, large synchronization buffer including thousands of macroblocks is required in the case of FHD decoding.

Nishihara et al. proposed a data partitioning method except variable length decoder by a result of macroblock level dependency analysis [8]. Figure 18 shows the direction of the macroblock level dependency, and Table II shows the result of dependency analysis and partitioning approaches for each function. To reduce the macroblock level synchronization overhead caused by data partitioning naturally, they used coarse grained partitioning approaches. Table III shows the proposed coarse grained partitioning approaches. Figure 19 shows the partitioning for each core.

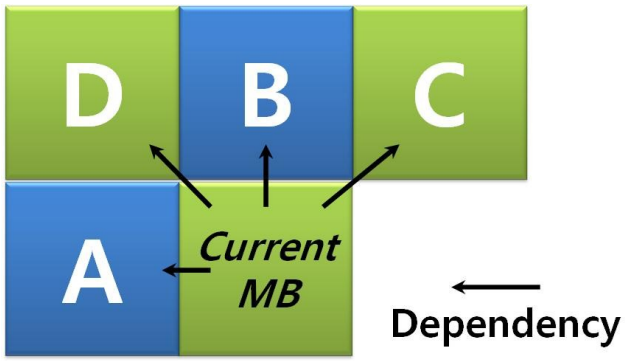


Figure 18: Dependency among MBs [8]

TABLE II

PROPERTIES OF H.264 DECODING FUNCTIONS AND PARTITIONING APPROACHES [8]

Functions	Property
VLD	A sequential execution is needed
MC	No dependence on neighboring MBs (A~D in Figure 18)
IP	Dependence on the neighboring MBs (A~D in Figure 18)
LF	Dependence on the neighboring MBs(A~C in Figure 18). Independent execution is possible.

TABLE III

PROPOSED PARTITIONING APPROACHES DEPENDING ON DEPENDENCY ANALYSIS [8]

Functions	Approach
VLD	Concurrent execution with previous LF and following MC
MC	Frame partitioning and adjustment of the partition size at runtime
IP	Frame partitioning into MB lines and execution with synchronizations at each MB
LF	Frame partitioning into some set of MB line and execution independently

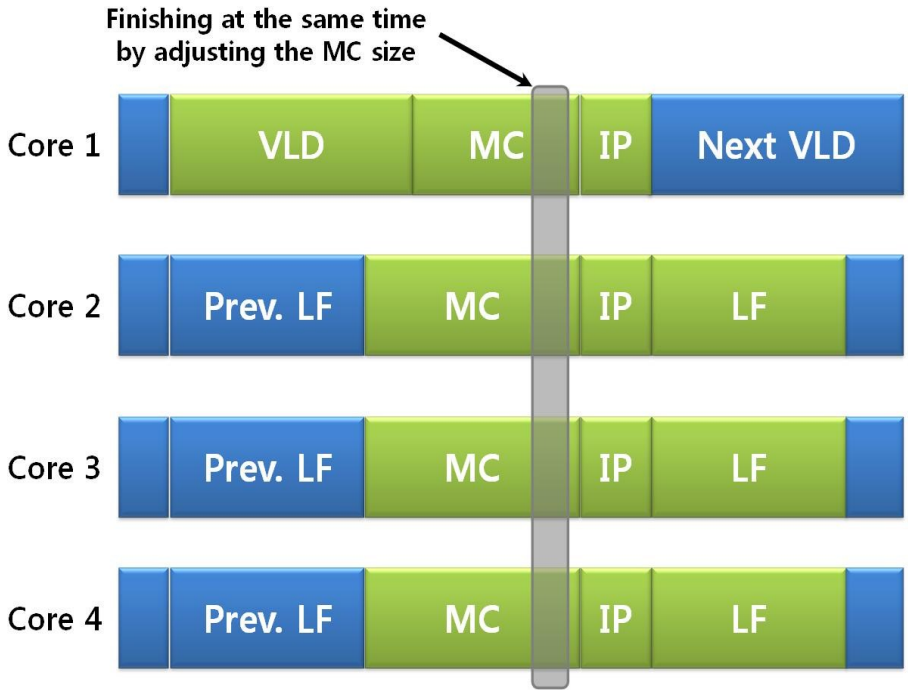


Figure 19: Partitioning of one frame process [8]

Even though they use those coarse grained partitioning approaches, it also requires large memory buffers to share the output of variable length decoder in external SDRAM. This kind of large memory buffer causes cache contention which degrades the performance of the parallelized decoder. Figure 20 shows the analysis of the degradation factor including cache contention in their implementation.

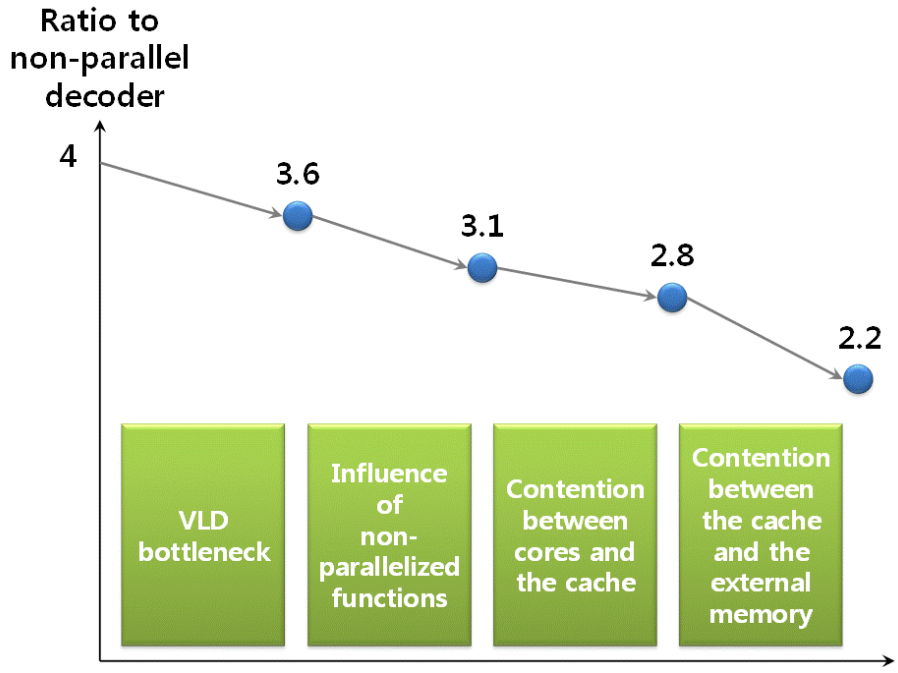


Figure 20: Degradation factor [8]

3.3.3 TASK POOL APPROACH AND RING LINE APPROACH

Chi analyzed the data partitioning methods for H.264 decoder in [16] for the parallelization on Cell Processor. He proposed two methods for assignment of macroblock data to each core. One is TP(Task Pool) approach and the other is RL(Ring Line) approach. TP approach is the same as traditional worker-sever model and it has been used already in Van der Tol et al. [4]. In worker-server model, server task check dependencies of all macroblocks processed in each worker and it allocates idle core for decoding the macroblock which all dependency problem are resolved for. Figure 21 shows the detail of the task pool algorithm.

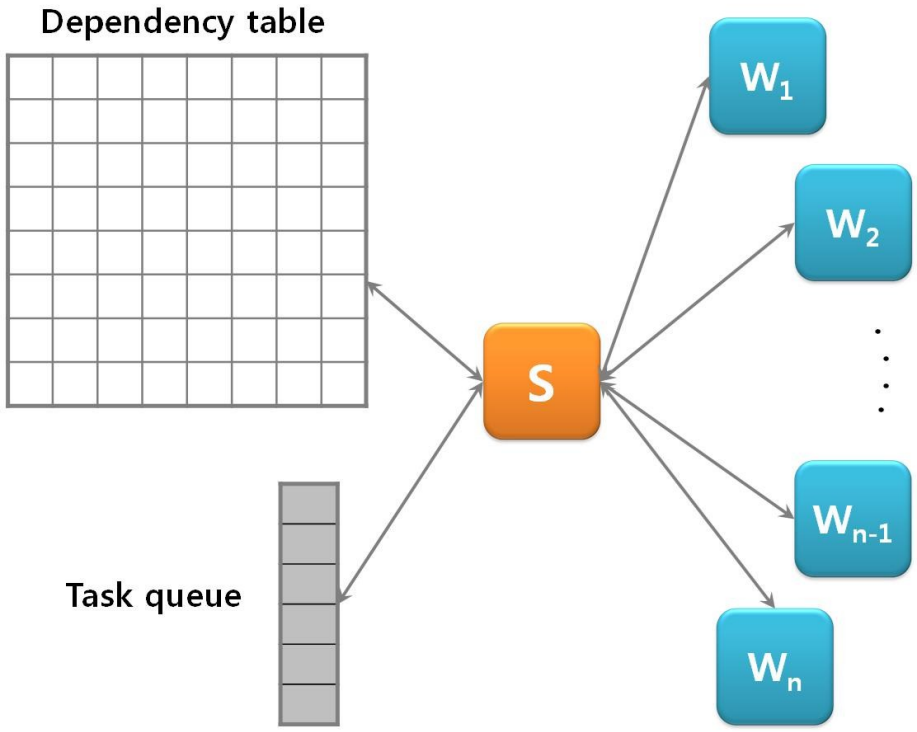


Figure 21: The task pool algorithm is based on a worker-server model. [16]

Being different from TP approach, all macroblocks in single row of the frame are decoded by one single core. During decoding single macroblock row, each core check the dependency by itself. The synchronization between cores is easy because it is enough to check the availability of the previous macroblock row. This means that one core needs to check the progress of the previous core only not all other cores. This characteristic is suitable for Cell broadband engine because Cell has ring topology for communication between cores. Figure 22 shows the architecture of Cell broadband engine.

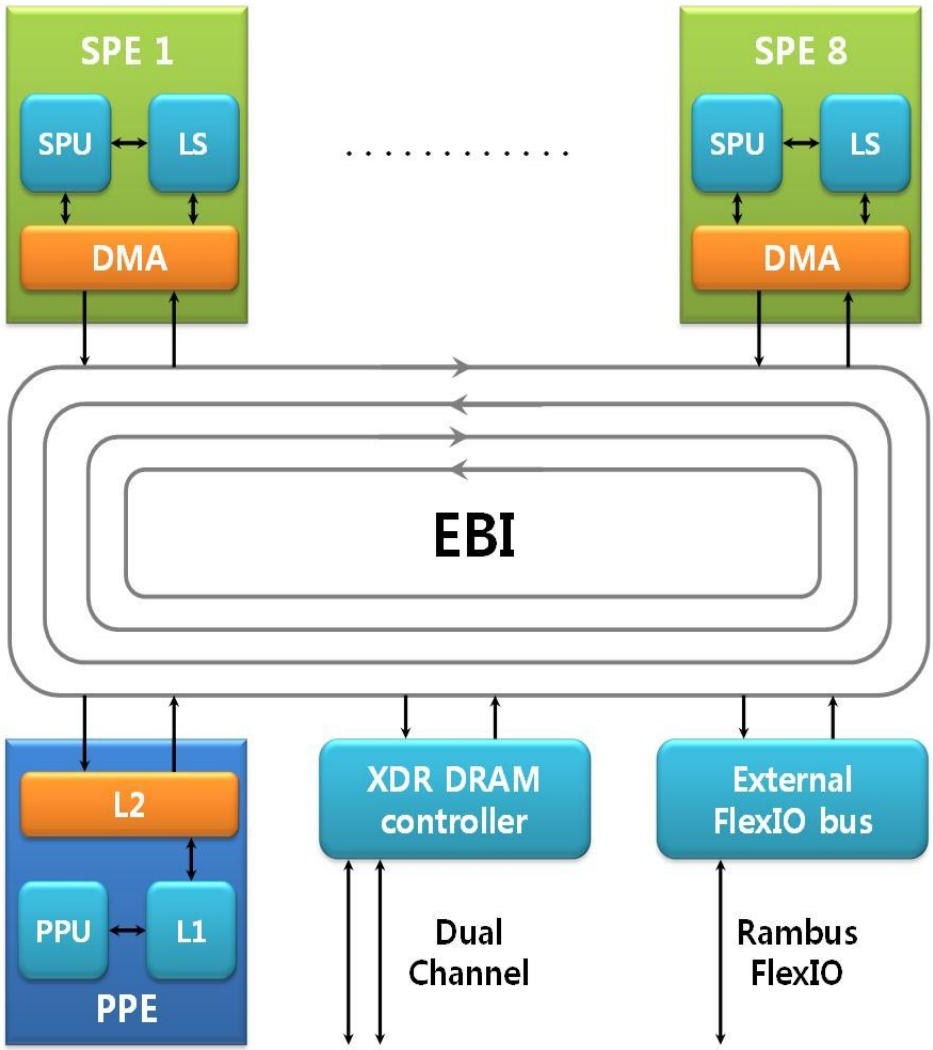


Figure 22: Schematic view of the Cell Broadband Engine architecture [16]

Figure 23 shows the dependency flow of RL algorithm. Figure 24 shows the mapping of ring algorithm. The role of C processor is giving start and stop signal at slice and frame level. Chi also extended RL approach to multi frame. Figure 25 shows the extended version of ring line approach.

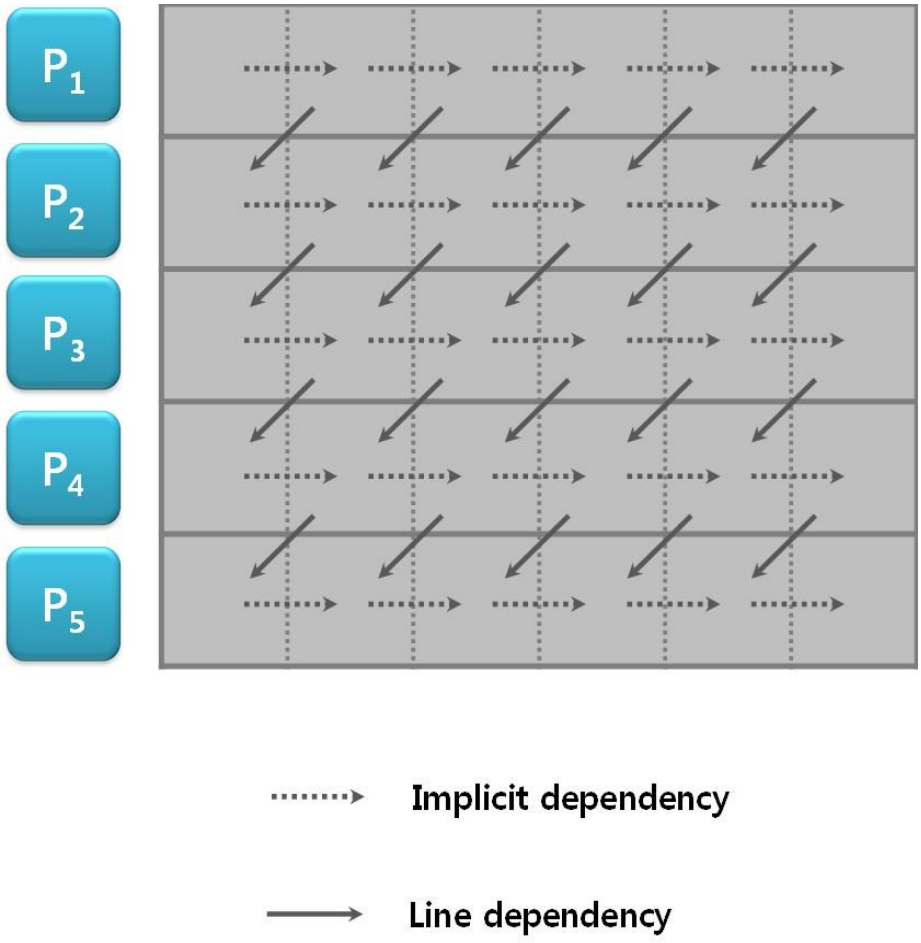


Figure 23: Simplified dependency flow of the RL algorithm. Dependencies flowing from one block to another on the same line are implicit. [16]

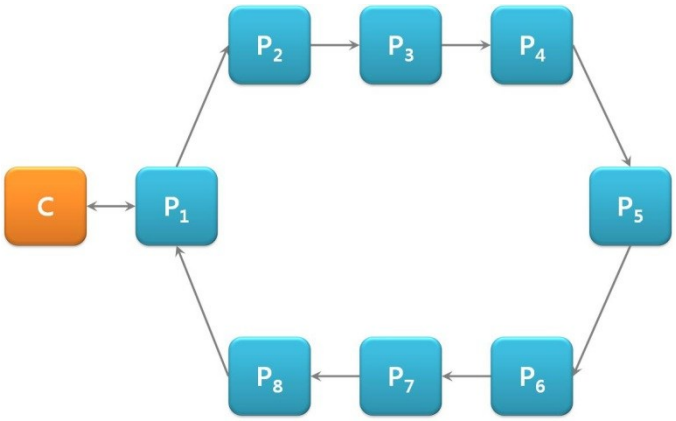


Figure 24: Uni-directional ring mapping of processing elements in the RL approach. The C-node is the control node which provides start and stop signals once a frame [16].

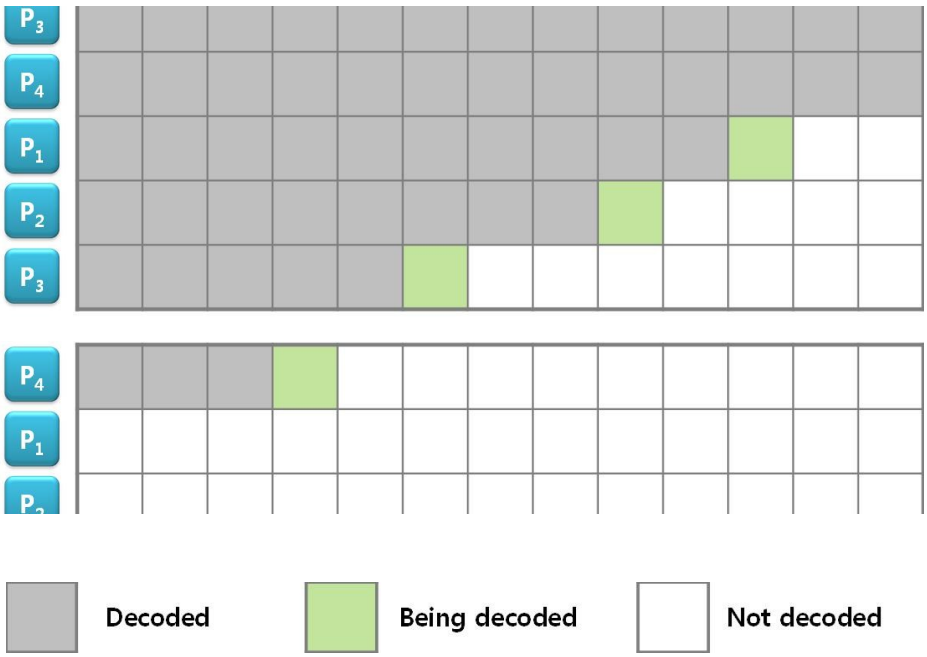


Figure 25: In multi-frame RL the decoding of the next frame start before the current frame end, which effectively negates the ramping stalls. [16]

3.4 PARALLELIZATION STRATEGY; DYNAMIC LOAD BALANCING, AND HYBRID PARTITIONING FOR EMBEDDED MULTI-CORE SYSTEM

By reviewing the previous work for the parallelization for of H.264/AVC decoder on multi-core system, it was found that there is no fixed rule or theory for mapping a general application to multi-core system, but there is empirical theory or rule of thumb to guide software developers to find efficient way of parallelization. It depends on number of cores, complexity of each functions, data dependency, and memory requirement of application. As for memory requirement, it is also related with memory limitation of multi-core system. Following items are most important criteria to decide a proper parallelization method.

1. *Number of cores*
2. *Complexity difference between each functions*
3. *Dependency between data*
4. *Memory requirement and limitation*
5. *Existence of hardware accelerator*

If a given multi-core system has many cores, then data partitioning is more preferred because of its scalability. There is no scalability in functional partitioning, therefore it is almost impossible to parallelize the given application on the many core systems like a hundred or a thousand core systems. The boundary of the number of cores for functional partitioning is about 4 in usual cases. The focus of this paper is on the embedded multi-core system with 2~4 cores, therefore applying functional partitioning is usually

possible. If there is dependency between processed data, functional partitioning is preferred because of easy synchronization between cores. In some applications, there can be spatial data dependency. The spatial data dependency means that the result of processing data of neighboring position are required for processing data of the current position.

If the application requires large size of program memory or the memory size of a given system is small, functional partitioning is also preferred because it requires small memory compared to data partitioning. If the number of functions to be distributed to all cores in the given multi-core system is less than the number of cores in the given system, then functional partitioning is impossible. If the differences of complexity between functions are large, then data partitioning is preferred. If there are special hardware accelerators in the given system, then it means already functional partitioning between software and hardware. Therefore, functional partitioning is more proper to the system which has hardware accelerators.

The program memory requirement for single core system P_s can be described as follows.

$$P_s = P_{gf} + P_{sf} \tag{3.1}$$

where P_{gf} denotes program memory requirement for general function like interrupt control routine and DMA function, and P_{sf} denotes program memory requirement for special functions of the given application. The program memory requirement for each partitioning type can be described as follows.

$$\begin{aligned} P_F &= nP_{gf} + P_{sf} + nP_{cf} \\ P_D &= n(P_{gf} + P_{sf}) + nP_{sync} \end{aligned} \tag{3.2}$$

where P_F denotes program memory requirement for functional partitioning,

P_D denotes program memory requirement for data partitioning, P_{cf} denotes program memory requirement for functions of communication between cores in functional partitioning, P_{sync} denotes program memory requirement for functions of synchronization between cores in data partitioning, and n is the number of cores in the given system.

The data memory requirement for single core system D_s can be described as follows.

$$D_s = D_{gf} + D_{sf} + D_{sh} \tag{3.3}$$

where D_{gf} denotes data memory requirement for general function, D_{sf} denotes data memory requirement for special functions of the given application, and denotes the data memory requirement for shared data between functions. The data memory requirement for each partitioning type can be described as follows.

$$\begin{aligned} D_F &= nD_{gf} + D_{sf} + D_{sh} + (n-1)D_{cbuf} \\ D_D &= n\left(D_{gf} + D_{sf}\right) + D_{sh} + nD_{sync} \end{aligned} \tag{3.4}$$

where D_F denotes data memory requirement for functional partitioning, D_D denotes data memory requirement for data partitioning, D_{cbuf} denotes data memory requirement for communication buffers between cores in functional partitioning, D_{sync} denotes program memory requirement for buffer of synchronization between cores in data partitioning.

As for many embedded multi-core system, the number of cores on that system is usually 2 or 4 up to now. For dual-core system, functional partitioning is more suitable than data partitioning because the cost for data partitioning is too much compared with gain from data partitioning. For quad-core system, it is difficult to determine which partitioning type is more proper because that number is boundary of decision. Therefore, partitioning type for quad-core system should be determined depending on the features of the

application.

Although we choose functional partitioning for parallelization of the given application by rule of thumb, there is usually a problem of load balancing. If a given system has a few cores, and the complexity of each functional module is similar, functional partitioning may be more proper than data partitioning. Even if functional partitioning shows better performance than data partitioning as parallelization strategy, the load of each core should be changing dynamically depending on input data. To resolve this problem, dynamic load balancing method will be proposed. Because the dynamic load balancing is just a general method, there will be a lot of methods for dynamic load balancing depending on applications. Therefore it is important to choose proper criteria for the decision depending on given applications. We will propose a proper method and criteria for dynamic load balancing in the case of H.264 decoder on embedded dual-core system.

If the system was many cores like 100 cores or similar number of cores, applying functional partitioning is almost impossible because the balancing among cores cannot be resolved. Therefore, data partitioning is more proper for those many-core systems. If the system has dual-cores, functional partitioning is more proper because of the reason mentioned previously. However, what about the quad-core system? It is very ambiguous to determine partitioning type for the quad-core system, because the number of cores is on the boundary of criteria. In some cases, functional partitioning is more suitable for a given application on the quad-core system, and data partitioning is more proper in the other cases. The reason is that functional partitioning and data partitioning have different features or pros & cons. What if all good features of each partitioning type can be merged by mixture of two partitioning methods? What if we can apply two partitioning method at the same time? We will call this newly proposed partitioning type as hybrid partitioning. If a proper hybrid partitioning is applied for the given quad-core system, the ambiguity will disappear and the only good features of each

partitioning type can be taken. To implement a proper hybrid partitioning, some functions are partitioned by functional partitioning, and this means that those functions are executed on different cores. On the other hand, the other functions are partitioned by data partitioning methods, and the cores allocated for data partitioning execute same functions with different data. Therefore, to determine which functions are partitioned by functional or data partitioning, the features of all functions of a given application should be analyzed precisely. We will propose a novel hybrid partitioning method for a H.264/AVC decoder based on exact analysis of all functions of the decoder and the proposed hybrid partitioning will show an impressive improvement of performance compared to applying functional partitioning or data partitioning only. The purpose of this paper is to find proper parallelization strategy for general application and to propose novel methods called dynamic load balancing and hybrid partitioning to enhance those parallelization methods.

Especially, H.264/AVC decoder on embedded quad-core system will be considered to prove a validity of the proposed methods. In the following chapters, we will apply the proposed parallelization strategy to H.264/AVC decoder on embedded dual-core system and quad-core system. After applying those parallelization strategies, we will apply dynamic load balancing for the dual-core system and hybrid partition for quad-core system. Figure 26 and Figure 27 show the summary of the previous description with flowchart. Figure 26 shows the flowchart for the dual-core system, and Figure 27 shows the flowchart for quad or more-core system. $Perf_m$ denotes the performance of the given application on multicore system, $Perf_{tgt}$ denotes the target performance, P_{sys} denotes total program memory size of the given multi-core system, and D_{sys} denotes total data memory size of the given multi-core system.

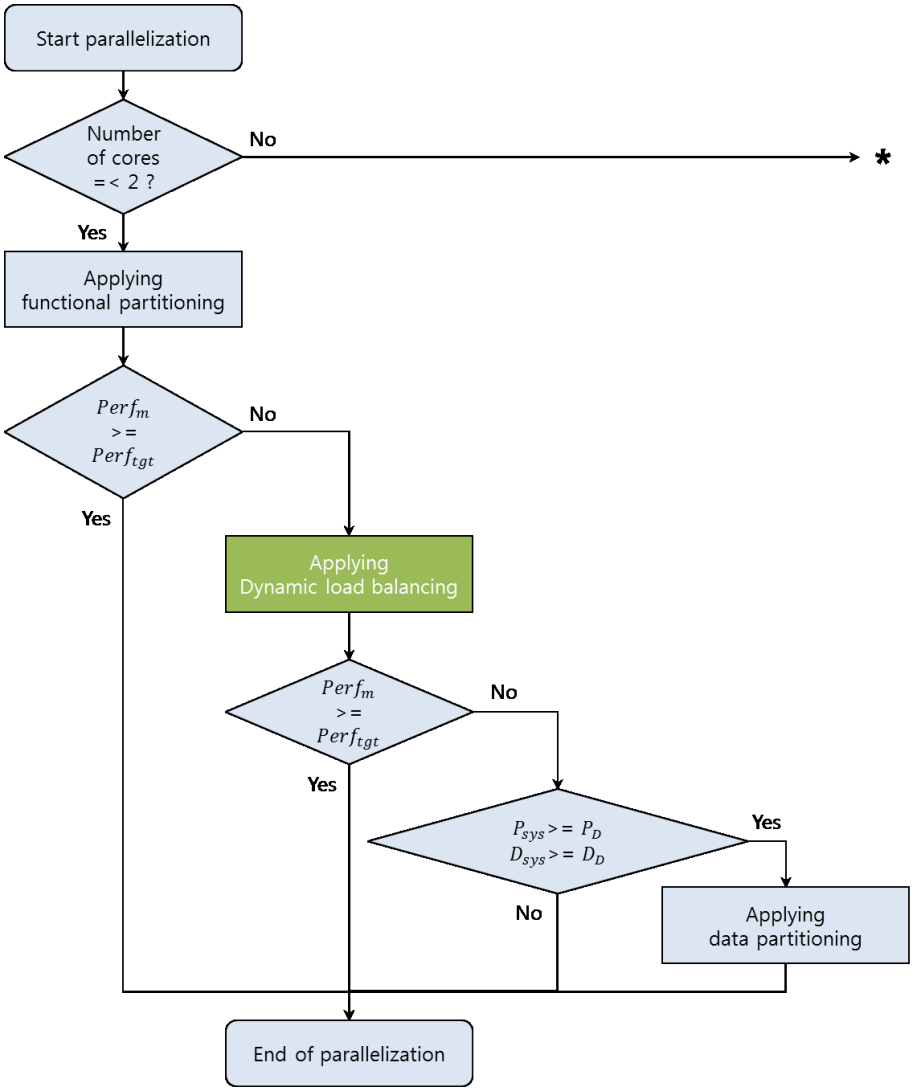


Figure 26: Flowchart for parallelization strategy in case of dual-core system

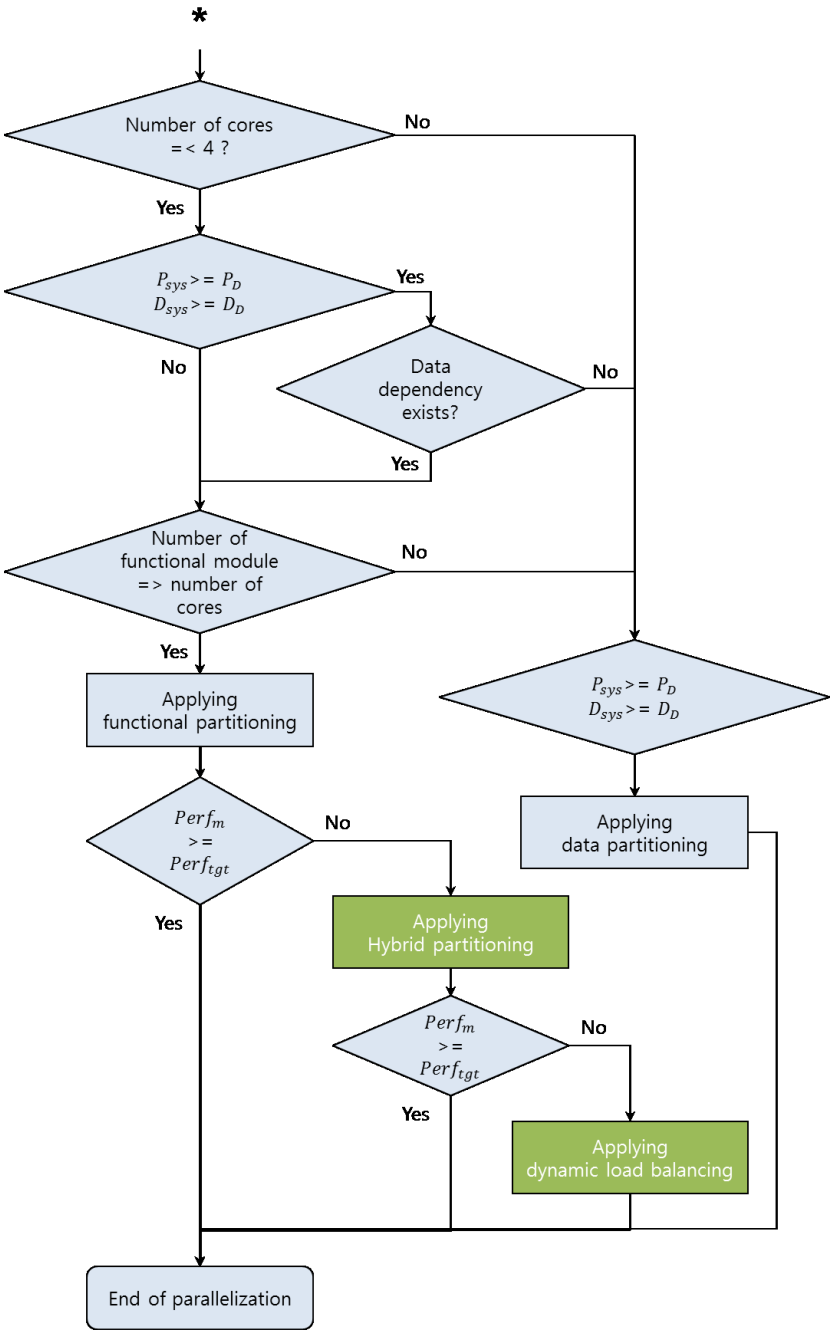


Figure 27: Flowchart for parallelization strategy in case of quad or more core system

CHAPTER

4

Parallelization of H.264 Decoder on Dual-core System with Dynamic Load Balancing

4.1 EMBEDDED DUAL-CORE SYSTEM ARCHITECTURE AND DATA FLOW OF H.264/AVC DECODER

Figure 28 shows whole system architecture of a dual-core platform used for this paper and data flow of the proposed H.264 decoder, where MV and MVD mean motion vector and motion vector difference, refIdx means reference index. Whole system consists of the processor with dual-cores, VLD HWA(Hardware Accelerator), MC HWA, deblock HWA and single SDRAM(Synchronous Dynamic Random Access Memory). In SDRAM, MB header information, MV, refIdx and decoded pictures as reference picture are stored.

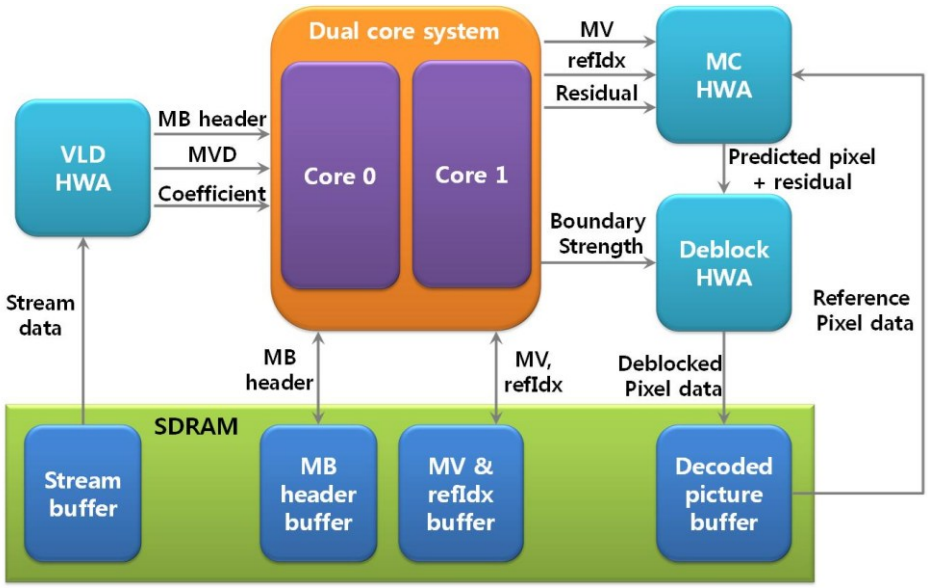


Figure 28: Block diagram of Dual-core system with 3 H/W accelerators

The targeted clock speed is 266MHz for decoding FHD H.264 main profile bitstream with 20Mbps bitrates. To reduce decoder operation cycle, VLD, MC and deblock functions are mapped to HWAs. These HWAs are originally designed for single core, so they have the interface for single core only. Core0 and Core1 are RP(reconfigurable processor)-based digital signal processor which has VLIW (very long instruction word) and CGRA(coarse-grained reconfigurable array) architecture for software acceleration with loop-level parallelism[8]. Before applying parallelization strategy for the given system, we will check the data flow of the whole system for more precise analysis of that system.

To see the data flow of the whole system including 3 hardware accelerators and dual-core subsystem precisely, the dual-core subsystem will be considered as a single processing unit at initial stage. Figure 29 shows the system with single processing unit before parallelization.

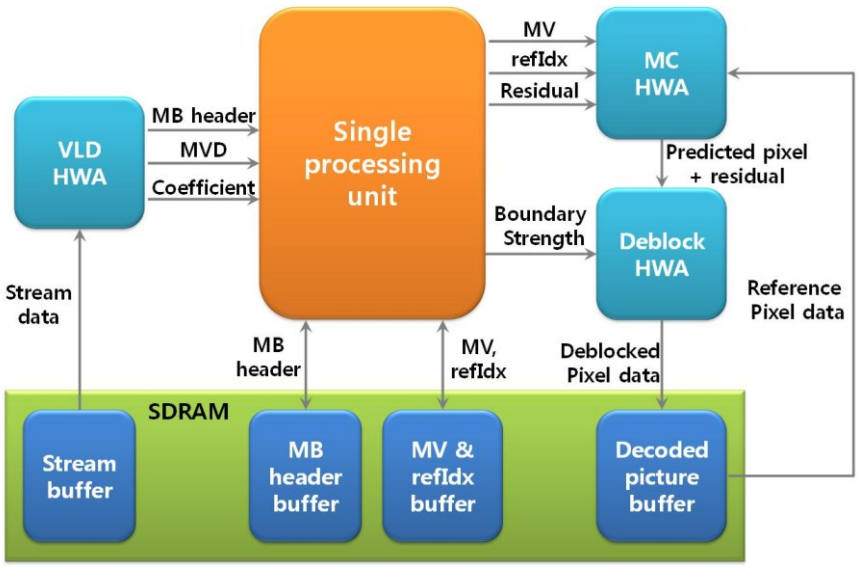


Figure 29: Data Flow and System Architecture with Single Processing Unit and 3 H/W accelerators

As can be seen in Figure 29, the bitstrem data in SDRAM are transferred to VLD HWA for parsing bitstream. After parsing bitstream by VLD HWA, the decoded data like MB header information, MVD, DCT(Discrete Cosine Transform) coefficient, and etc. are sent to a processing unit. Those data are processed by S/W(software) program in a processing unit, then MV, refIdx, residual data, BS(Boundary Strength) are generated as results of data processing. Among those data, MV, refIdx and residual data are transferred to MC HWA, and BS data are sent to deblock HWA. By using those data from processing unit and reference pixel data, MC HWA and deblock HWA generate decoded pictures and save those pictures to external SDRAM. Whole data flow is summarized as follows: VLD HWA is the beginning of whole data processing, and computing cores process the result of stream parsing. MC HWA and deblock HWA control the final step of whole decoder. If we see the data flow inside processing module, the core module calculates MV by using MB header and MVD. The processing module also

calculates BS values by using same data. Then the processing module calculates residual data with dequantization and inverse-transform step by using DCT coefficients generated by VLD HWA. Figure 30 shows the data flow inside processing module.

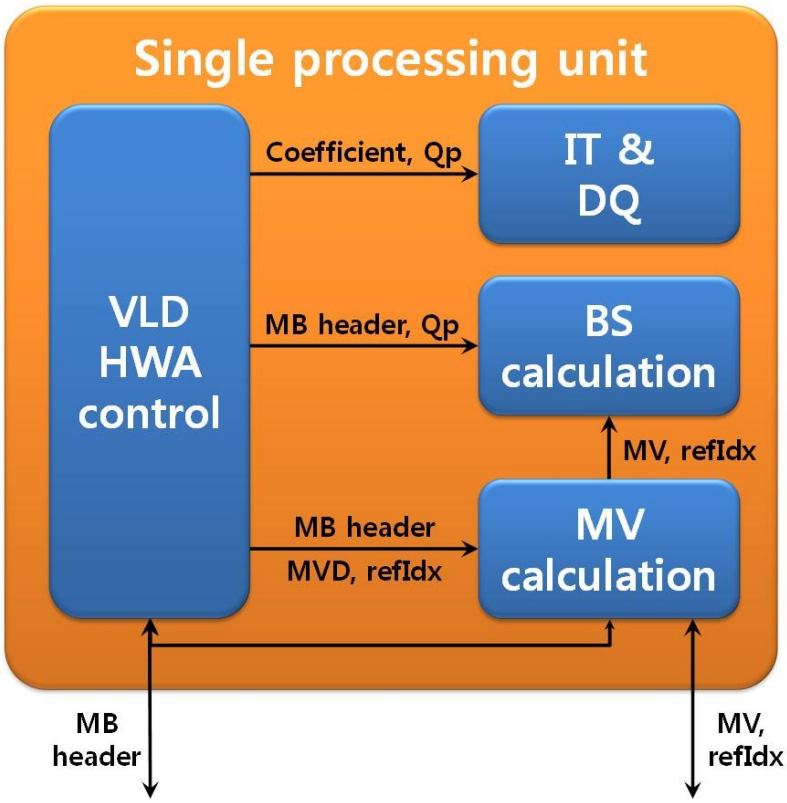


Figure 30: Data Flow inside Processing Unit

4.2 INITIAL PARALLELIZATION OF H.264 DECODER ON DUAL-CORE SYSTEM

In this section, we will apply the parallelization strategy proposed in the previous section for dual-core system. First of all, the given system is dual-

core system, therefore the functional partitioning is preferred unless there is a serious unbalance of loads among partitioned functional modules. To check the suitableness of functional partitioning for the given dual-core system with H.264/AVC decoder, we profiled the complexity of each functional module except the function mapped to HWAs by using a simulator for RP. From the profile result, major functions can be categorized to 4 parts as Table IV.

TABLE IV.

MAJOR 4 CATEGORIES FOR FUNCTIONAL PARTITIONS

Functions	**Weight(%)**
VLD HWA control	21.81
MV(Motion Vector) calculation	31.29
IT(Inverse Transform) & DQ(Dequantization)	28.72
BS(Boundary Strength) Calculation	17.53
Total	**99.35**

TABLE V

TWO CANDIDATES OF FUNCTIONAL PARTITIONING

	Candidate 1	**Candidate 2**
Functions on Core 0	VLD HWA control	VLD HWA control
	MV calculation	IT & DQ
Functions on Core 1	IT & DQ	MV calculation
	BS calculation	BS calculation
Load of Each core	Core 0: 53.10%	Core 0: 50.53%
	Core 1: 46.25%	Core 1: 48.82%

There can be two candidates for parallelization of H.264/AVC decoder on dual-core system by functional partitioning. Table V shows two candidates. All proposed candidates achieve almost 50% of load partitioning in simulation level. Figure 31 and Figure 32 show each partitioning by diagram including data flow.

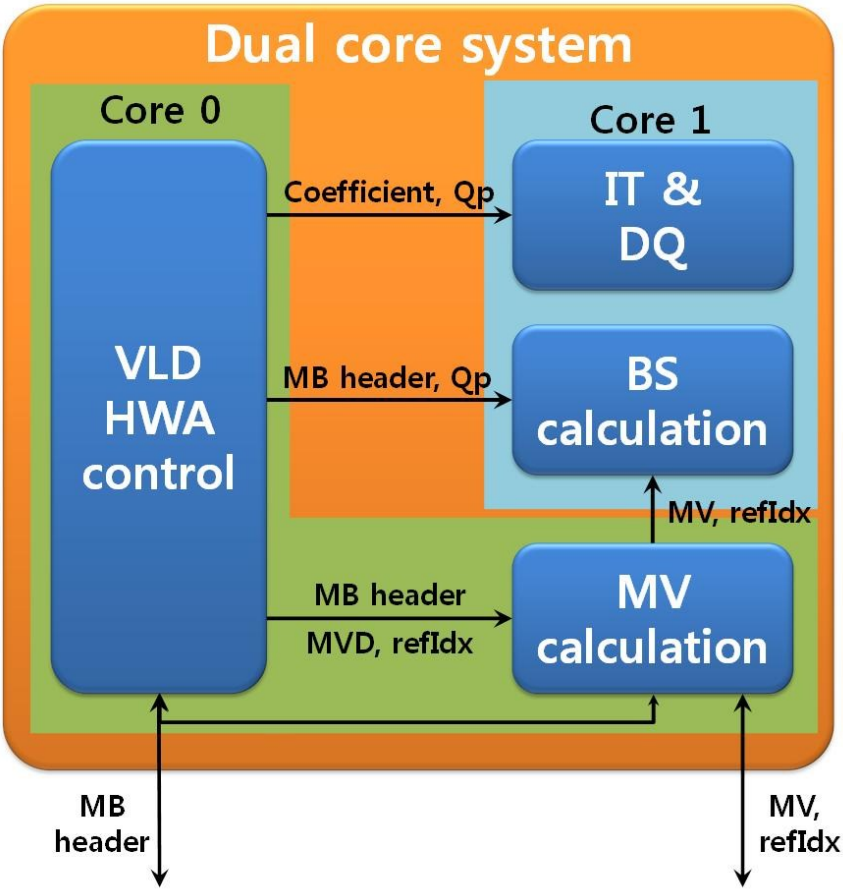


Figure 31: Functional partitioning by candidate 1

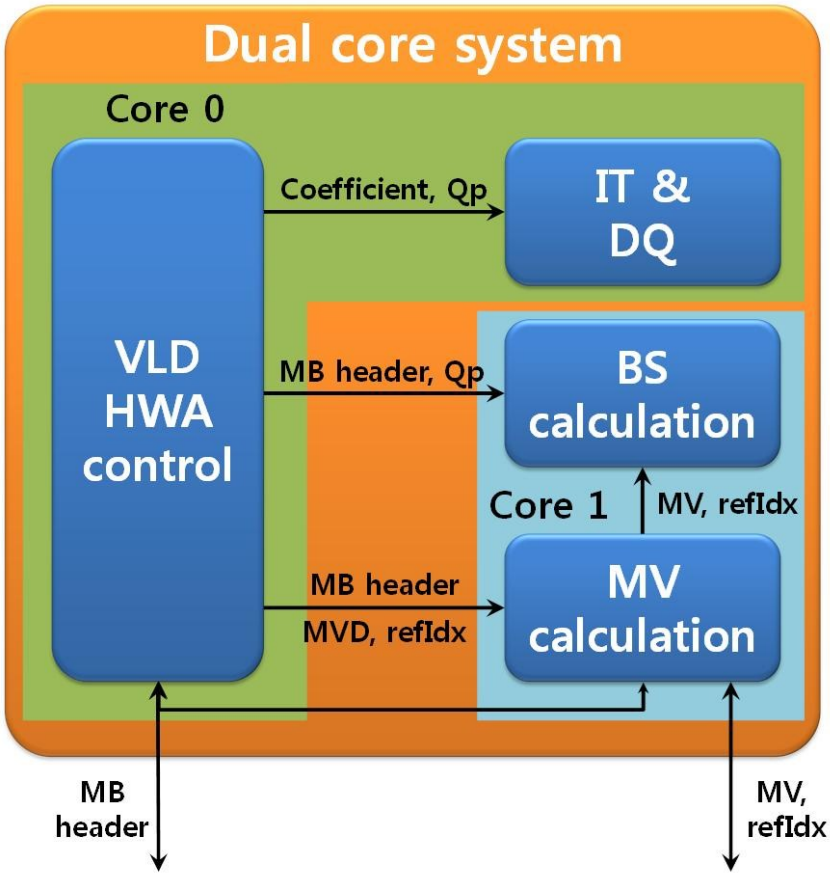


Figure 32: Functional partitioning by candidate 2

Because two candidates have similar balance of load, other criteria should be used to decide final functional partitioning. The next criteria considered for decision are overheads by each partitioning because those overheads may degrade the performance of parallelized decoder. Therefore the next step is checking two candidates from the view point of partitioning overhead. The important difference between two candidates is that the buffer in external SDRAM is managed by core 0 only in case of the first candidate. In contrast to the first candidate, all cores access the buffer in the external SDRAM in case of the second candidate.

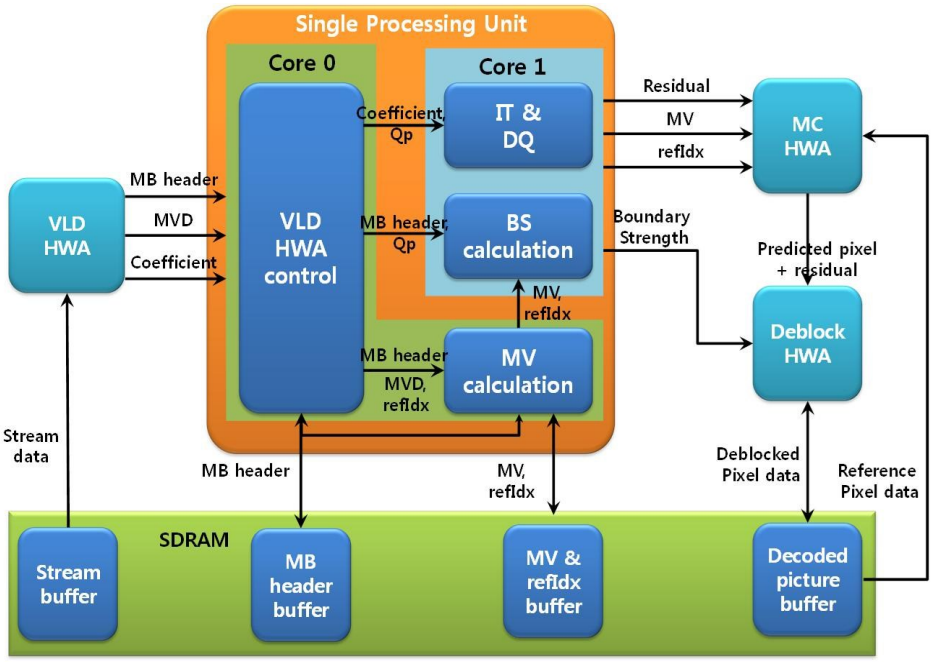


Figure 33: Data flow of whole system in candidate 1

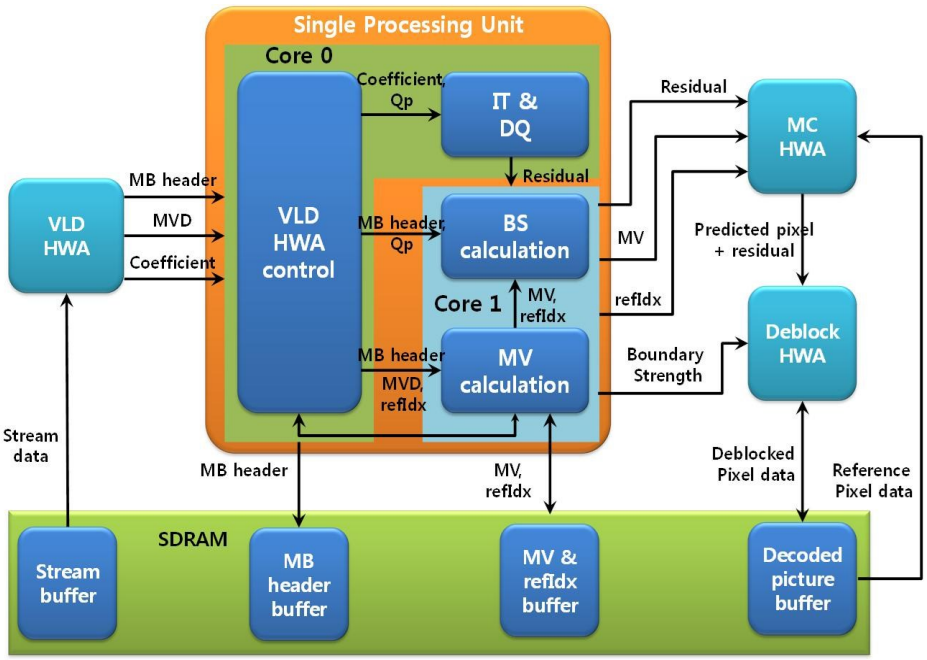


Figure 34: Data flow of whole system in candidate 2

Now we will consider the data transfer overheads of the two candidates. If all data processed by single core, there is no need to transfer the data in internal memory. In the case of quad-core system, those data should be transferred to internal memory of each other core. Figure 33 and Figure 34 show the detail of data transfer of whole system in two candidates.

Because 3 HWAs are designed in the assumption of single core system, all HWAs should be managed by individual core. In other words, the data communication for each HWA should be managed by single core. In two candidates, VLD HWA is controlled by core 0, and MC HWA and deblock HWA are controlled by core 1. Therefore that condition is satisfied. The important difference between two candidates from the view point of data flow is that residual transfer. The residual data transfer between cores happens in the case of the second candidate, which does not exist in the first candidate. In the case of the second candidate, the residual data generated by core 0 should be sent to core 1, because core 1 sends those data to MC HWA. The residual data are already pixel data, therefore the transfer size is large as pixel data. Compared to MV or similar data, residual data require a lot of data transfer between cores. The last consideration points are data consistency and easiness of implementation. As can be seen in Figure 33, all buffers in the external SDRAM are managed by core 0 in the case of the first candidate. Therefore the first one is easy to be implemented from the view point of a management of data buffer. In addition to that, core 1 and 2 HWAs(MC HWA, deblock HWA) operate as single HWA in the first candidate. In the case of the second candidate, buffers in SDRAM are managed by core 0 and core 1 separately in contrast to the first one. Especially, MB header buffer is accessed by core 0 and core 1 at the same time. Therefore it can cause problems from the view point of data consistency. To resolve those problems, one core should operate at read-only mode, and the other core should operate at write-only mode. Therefore it increases the complexity of implementation. By using this result, we could choose the first candidate as a functional

TABLE VI

INITIAL FUNCTIONAL PARTITION ON EACH CORE

	Core 0	Core 1
Functions	1. VLD HWA control 2. MV calculation	1. IT & DQ 2. BS calculation
Weight(%)	53.10	46.25

partitioning of H.264/AVC decoder as Table VI shows. Especially, easiness of controlling MV data, the data flow of the whole system, and the management of 3 HWAs are considered. Figure 35 shows the mapping of those functions on each core and data flow including communication buffers between cores. Those communication buffers are required for parallelization by functional partitioning because the processing speed of each core for single MB can be different dynamically. VLD HWA is controlled by core0 only, and also MC and deblock HWA are controlled by core1 only. So, the HWAs do not need to support the interface for multi-core. Core0 controls VLD HWA and calculates MV for each MB and sends the MB level data like MB header, MV, refIdx and transform coefficient to core1. In communication buffer, multiple MB level data are stored for core1. Buffer level means the number of valid MB data in communication buffer. The size of communication buffer is important because it can regulate the dynamic load differences between cores at macroblock level processing. If the buffer size is too small, it will increase the waiting time between cores due to the differences of processing time for different macroblock type. If it is too big, the total memory cost will increase. We determined the size of communication buffer as 32 macroblock data for FHD decoding by experiment. This buffer size is very small compared with that of the previous works like [4], [5] and [6] because those proposals should store thousands of macroblock data for inter-core communication in FHD decoding.

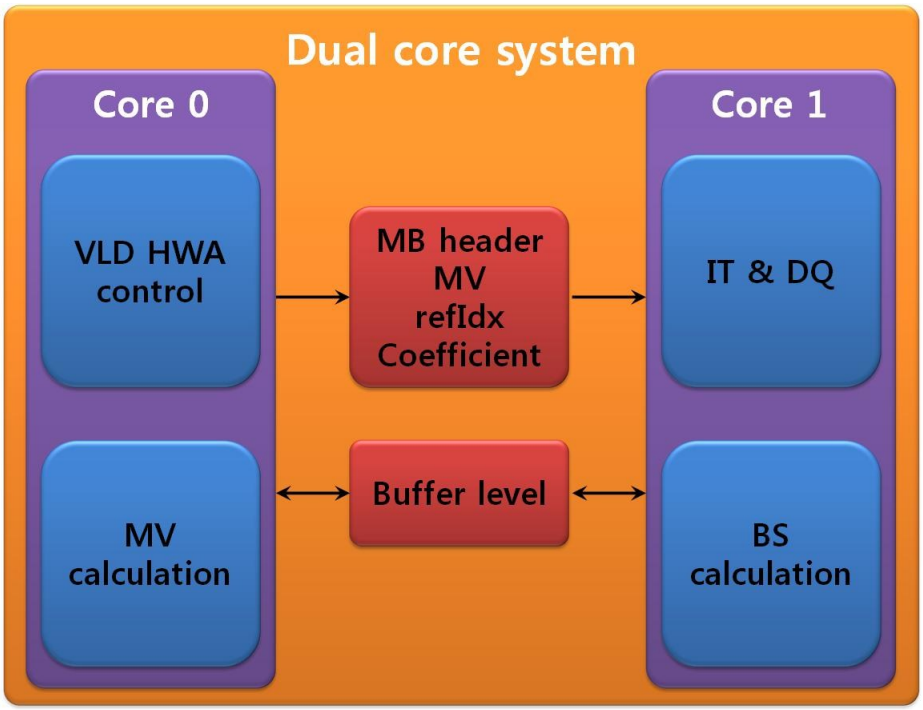


Figure 35: The decided functional partitioning and data flow between cores

4.3 DYNAMIC LOAD BALANCING ON FUNCTIONAL PARTITIONING

Depending on the partitioning in section 4.2, we mapped the proposed functional partitions to each core. The initial H.264/AVC decoder S/W was originally designed for single core system, therefore there were a lot of issues for the parallelization on the dual-core system. Especially, the communication

TABLE VII

THE FIRST OPTIMIZATION STEPS AND RESULTS.

Optimization method	Performance (MHz)
Initial parallelization of single core S/W (w/ 634MHz@single core)	1317
Increasing communication buffer size	1219
Direct copy to DMA (core 1 transfer)	884
Direct copy to DMA (core 0 transfer)	567
Move communication buffer position to internal memory from external SDRAM	528
Applying more optimized single core decoder (515MHz@single core)	457
32bits DMA to 64bits DMA	445
Parallelizing DMA of core 1	420
Parallelizing DMA of core 0	381
Applying DMA to communication between cores and HWAs	364

between two cores caused many problem and overhead to degrade decoder performance. To get enough effect from the dynamic load balancing approach described in chapter 2, we had to optimize initial H.264/AVC decoder on the given system. This first optimization step was mainly to reduce the overhead of the communication between two cores. Changing data transfer method to DMA(Direct Memory Access) from direct copy by cores, parallelizing DMA transfers as many as possible, increasing data transfer width to 128 bits from 64 bits, and changing data transfer between cores and HWAs to DMA were applied as optimization methods. The result of these steps of optimization are described in Table VII.

Even after applying first optimization steps, the target operation speed of 266MHz was not achieved, we should apply the dynamic load balancing described in chapter 2 for further enhancement of performance. To find proper dynamic load balancing method, we analyzed the overhead due to functional partitioning on dual-core system. As you can see in Table VIII, the overhead can be divided into two parts, one is the overhead by data transfer between cores and the other one is that by waiting for the other core due to unbalanced load.

TABLE VIII

DUAL-CORE MAPPING OVERHEAD ON CORE 0

Overhead type	MHz	Weight
Data transfer overhead	47.5	55.2
Waiting overhead	38.5	44.8
Total	86.0	100

The data transfer overhead can be reduced easily by parallelizing DMA(direct memory access) transfer efficiently. So we decided to focus on the waiting overhead. Before reducing the waiting overhead, we analyzed the overhead of each core depending on frame type. Figure 36 shows the waiting cycle at each frame. As you can see in Figure 36, the cycles for 'wait core 1', which means the cycles that core 0 waits for vacancy in communication buffer, is very large at intra-frames(frame 0 and frame 28). From this result, we could identify the major factor to determine waiting overhead, namely MB type. Figure 37 shows the relationship between the number of I-MB(intra-MB) in each frame and 'wait core1' cycles, and they are perfectly matched. In case of,

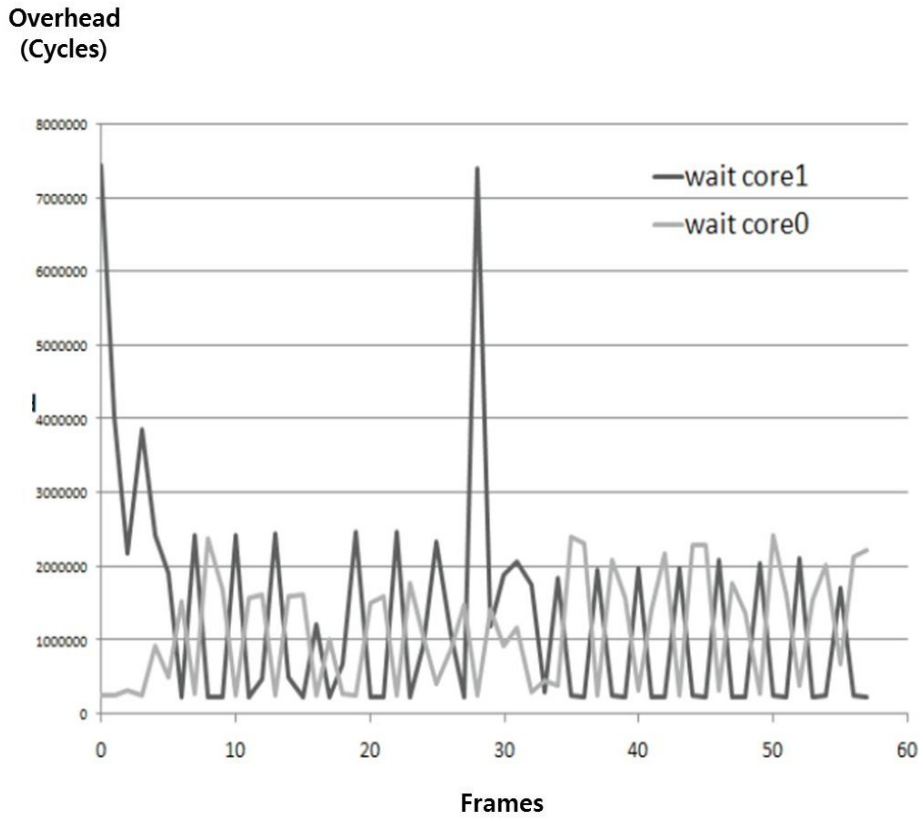


Figure 36: Waiting cycles on each core

I MB, the load for MV calculation is very small, and the load for IT and DQ became relatively large. Therefore load difference between cores will increase. Because of these unbalanced loads in I-MB, the cycles for 'wait core1' became large. To reduce the waiting cycles at I-MB, we mapped IT & DQ functions on both core and made core 0 decide dynamically which core runs those functions for each I-MB.

The buffer level between cores is used as decision criteria for dynamic load balancing because it reflects the load difference between cores. The buffer level will increase if core 0 runs faster than core1, and it will decrease if not. So, if buffer level is higher than a certain threshold, core0 will decide to run IT & DQ functions by itself and send message to core1 about that. In addition to dynamic load balancing for I-MB, we also mapped BS calculation at both core and used the same method for load balancing in the case of inter-MB.

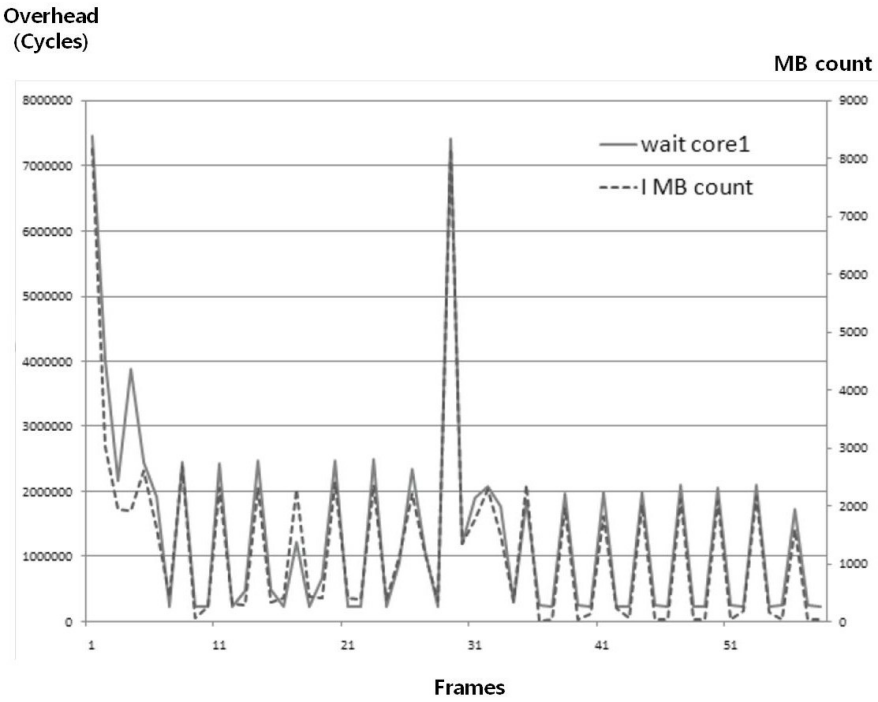


Figure 37: 'wait core 1' cycle and I-MB counts

The detailed decision mechanism and MB decoding routine are showed in Figure 38. Because the load balancing process is simple with buffer level only, the performance loss due to making decision for load balancing is negligible. In addition to small performance loss, it is easy to extend the proposed load balancing method to quad-core system because each core can balance the load with neighbor core by itself if there is only buffer level information. Figure 39 shows the final functional partitioning including the proposed dynamic loaf balancing methods

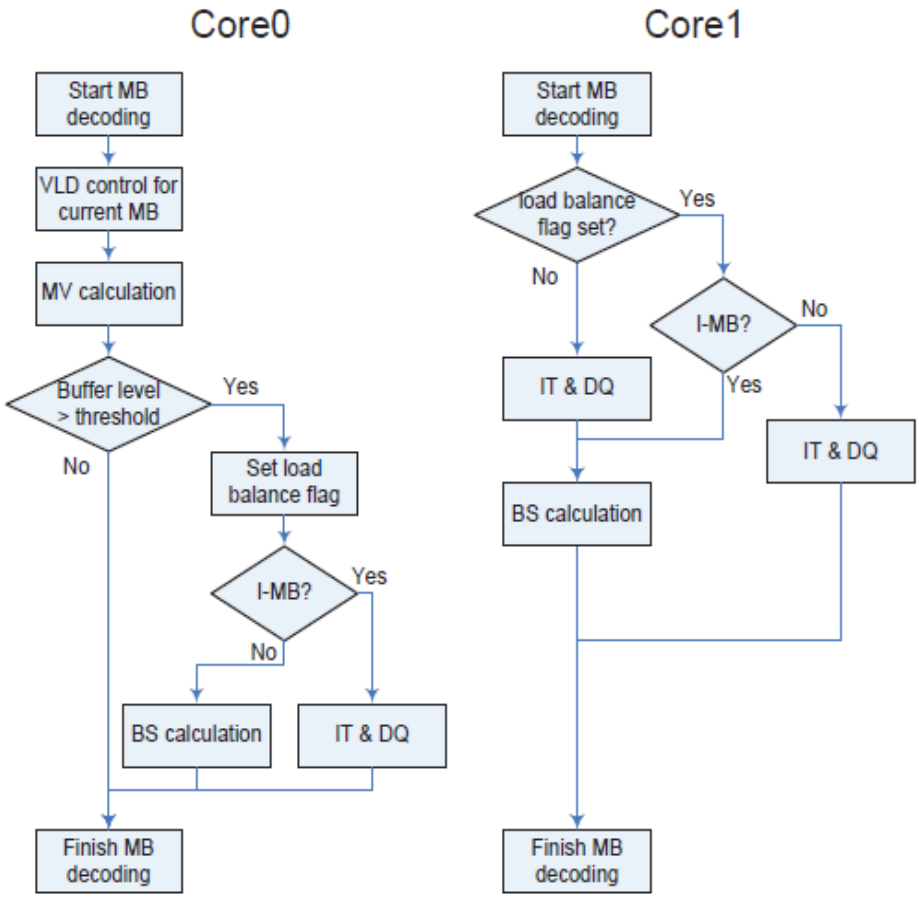


Figure 38: Flowcharts for MB decoding with dynamic load balancing on each core

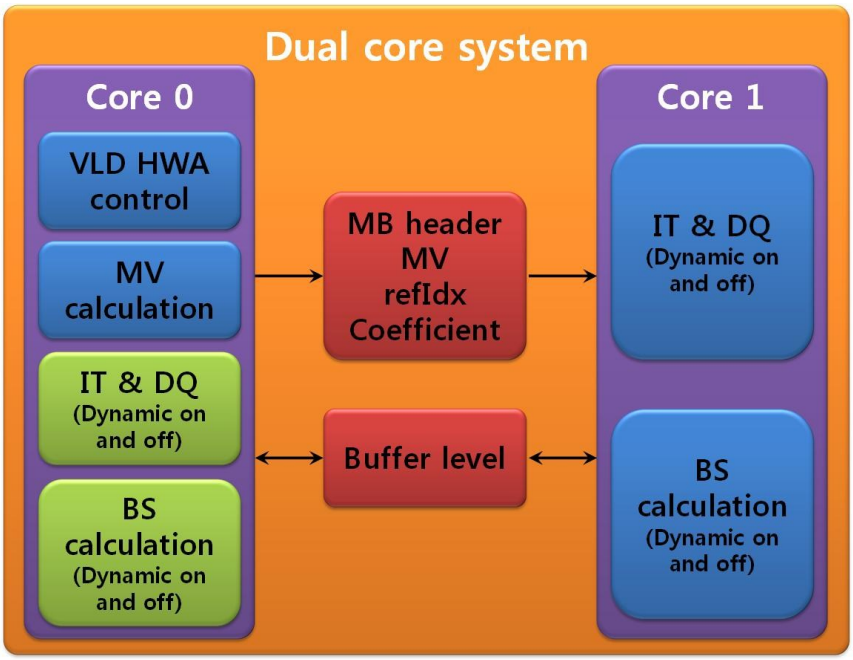


Figure 39: Proposed Dynamic Load Balancing

4.4 EXPERIMENTAL RESULT

In this section, we will show the result of the proposed load balancing method in functional partitioning. Fig. 31 shows the waiting cycles at each frame after the proposed load balancing method was applied and the previous waiting cycle also. As you can see in Figure 40, waiting cycles were reduced dramatically and distributed evenly to all frames. In case of peak value, 7.4MHz at frame 0 became 0.4MHz at frame 1 and there was 94.6% reduction. As can be seen in Table IX, the total amount of waiting overhead became 6.8MHz from 38.5MHz, and there was 82.3% reduction compared with the previous value. After applying dynamic load balancing, we also applied parallel DMA and other optimization techniques, and finally we achieved 263MHz for FHD H264 main profile decoder on our architecture.

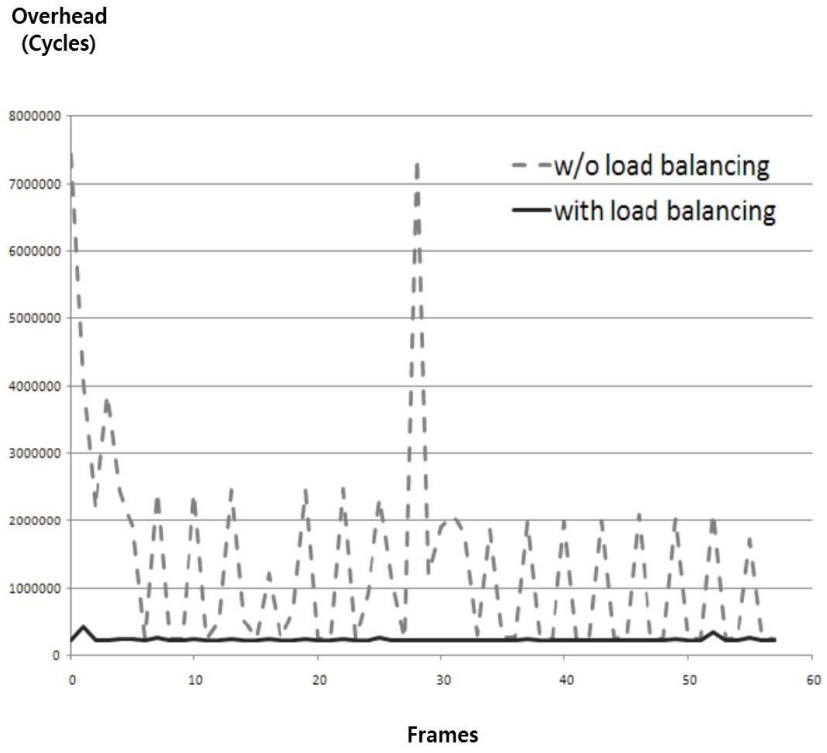


Figure 40: Waiting cycle with and w/o the dynamic load balance.

TABLE IX

REDUCTION OF WAITING OVERHEAD BY APPLYING THE PROPOSED DYNAMIC LOAD BALANCING

Conditions	Waiting overhead (MHz)
Without dynamic load balancing	38.5
With dynamic load balancing	**6.8**

CHAPTER

5

Parallelization of H.264 Decoder on Quad-core System with Hybrid Partitioning

5.1 EMBEDDED QUAD-CORE SYSTEM ARCHITECTURE

Figure 41 shows the whole system architecture of the embedded quad-core system, and data flow of the H.264 FHD decoder. The embedded quad-core system for this chapter is enhanced version of the previously used dual-core system in chapter 4. The quad-core system was developed to support more video coding standards including H.264/AVC than dual core system. To increase the flexibilities of the whole system, the quad-core system is composed of only programmable cores without hardware accelerators. Compared to dual-core system, hardware accelerators for motion compensation and deblock filtering were removed. To compensate this lack of performance, the number of cores in the system was increased to 4 from 2.

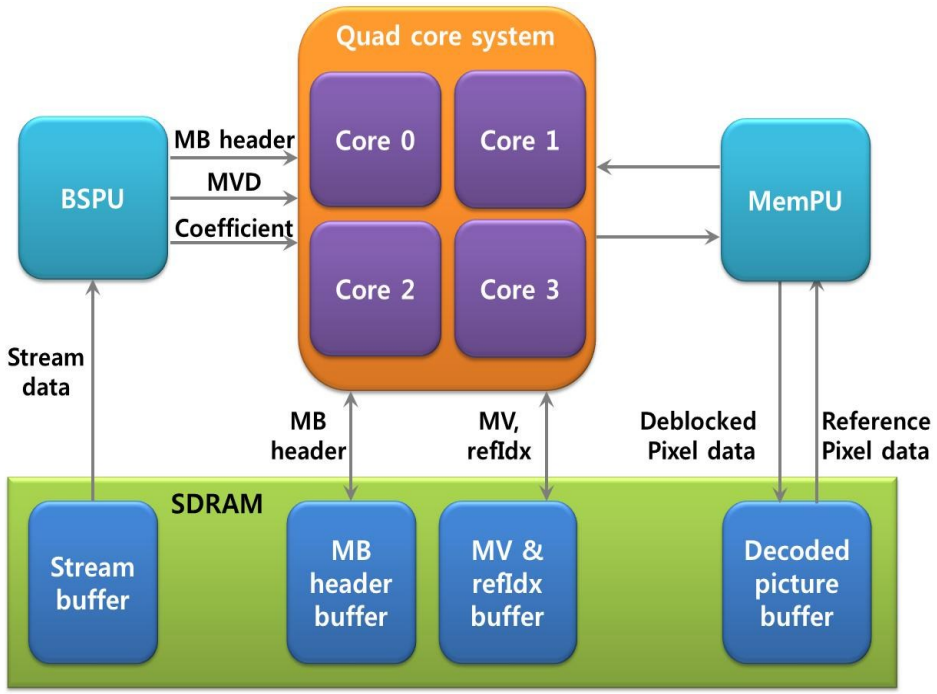


Figure 41: Embedded quad-core system architecture.

Bitstream, decoded pictures, motion vectors, and the other main data for video decoding are stored in external SDRAM, and all cores share those data. The whole system has two special co-processors, one being BSPU, and the other, MemPU. BSPU can parse general video bitstream efficiently. A role of MemPU is to provide an efficient method of accessing memory for motion compensation and deblock filtering, which have complex access patterns. BSPU and MemPU are programmable to support many video coding standards. Bitstream parsing is a very common function of all video decoders, but it is not suitable for general DSP, because almost all of its operations are conditional branches and jump routines, which break the pipeline of DSP. As a result, a special co-processor called BSPU suitable for bitstream parsing was

developed, which did not lose programmability for other video coding standards. The other co-processor, MemPU, has special functions to support complex memory access patterns of motion compensation and deblock filtering efficiently. Motion compensation and deblock filtering are also common in video decoders. MemPU can access memory more efficiently than general DSP, thus it can reduce the burden of the quad-core system for memory access [25]. With the exception of bitstream parsing, all the other computational functions will be distributed to the quad-core.

Each core is an RP-based DSP that has VLIW (Very Long Instruction Word) and CGRA (Coarse-Grained Reconfigurable Array) architectures, for software acceleration with loop-level parallelism [13]. The architecture of the cores is same as that of dual-core system, and only the number of cores is increased. In the RP core, there are two types of program memory. One is VLIW program memory, and the other one is CGRA program memory. VLIW program memory stores general functions, like DMA (Direct Memory Access) transfer routines, and interrupt service routines. It also stores the functions for computational operation, which cannot be accelerated by CGRA architecture. In CGRA program memory, there are major operations for the video decoder, like motion compensation, deblock filtering, etc. These functions can be accelerated by CGRA architecture.

5.2 COMPLEXITY ANALYSIS AND INITIAL MAPPING ON QUAD-CORE

To find the proper partitioning method for the quad-core system, the H.264 FHD decoder was profiled with an RP simulator. Compared to dual-core system, functions for motion compensation and deblock filtering were added

TABLE X

MAJOR FUNCTIONS OF H.264 FHD DECODER

Functions	Weight(%)
Motion compensation	53.9
Deblock filtering	24.1
Inverse transform	6.3
Entropy syntax analysis	5.9
MV calculation	5.7
BS calculation	4.1

to the list. Table X shows the portion of major functions in the H.264 decoder. As you can see, the complexity of the motion compensation function is very high. By following the parallelization strategy proposed in chapter 3, it seems better to apply data partitioning method for the H.264/AVC decoder on quad-core system. In case of a regular functional partitioning, the motion compensation module cannot be divided into multiple sequential functions simply because the motion compensation routine calls different sub-functions according to types of MB and the 8x8 block. Therefore, these large differences in load between functions may cause a bad load balancing result between cores in regular functional partitioning. On the other hand, if data partitioning is chosen, this will cause an MB level dependency problem, and requires large buffers in SDRAM for storing all MB level data. These buffers in SDRAM create a large amount of access to SDRAM. These kinds of problems just cause the degradation of performance, therefore it seems possible to apply data partitioning to take the performance degradation until

TABLE XI

INITIAL FUNCTIONAL PARTITIONING ON EACH CORE

	Functions	Weights(%)
Core 0	Entropy syntax analysis MV calculation BS calculation	**15.7**
Core 1	Inverse transform	**6.3**
Core 2	Motion compensation	**53.9**
Core 3	Deblock filtering	**24.1**

that moment. In addition to those problems, data partitioning also increases the program memory size of the whole system [4]. Next step is checking memory requirement of data partitioning. When the total amount of CGRA program memory required by data partitioning was checked, it exceeded the real size of CGRA program memory implemented on our quad-core system. In the RP core, there is no instruction cache for CGRA program memory. Therefore, the size of that program memory must be increased, if the data partitioning is applied. These kinds of changes were almost impossible because they would increase the total gate counts of the whole silicon chip and they would require to re-design the chip which repeats many steps of chip development and causes additional cost. The comparison of program memory size of each partitioning type will be shown at section 5.4.

Because of these problems, regular data partitioning is not suitable for the quad-core system. To overcome those problems, we decided to apply the hybrid partitioning on the given quad-core system by following the

parallelization strategy proposed at chapter 3. Therefore, the goal of initial partitioning is to find the best functional partitioning as the first step before applying the hybrid partitioning. Considering the data flow and the loads of all the cores, initial partitioning was determined, as shown in Table XI. This is the most effective partitioning, when applying regular functional partitioning. As for the data flow of the initial partitioning, it will be explained in detail in section 5.3. Figure 42 shows the concepts of initial functional partitioning, and Figure 43 shows the data flow between each core

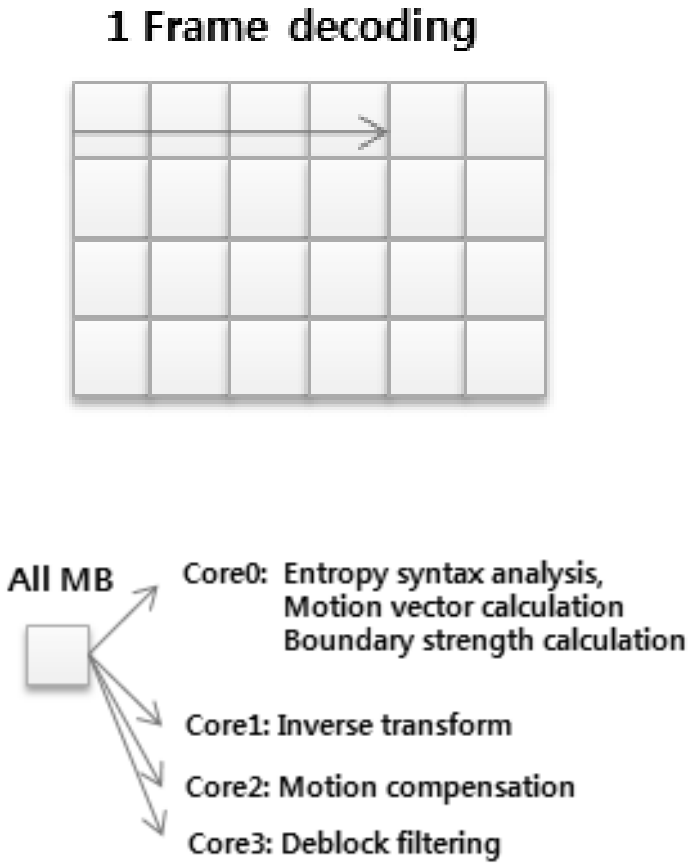


Figure 42: Initial functional partitioning and the role of each core.

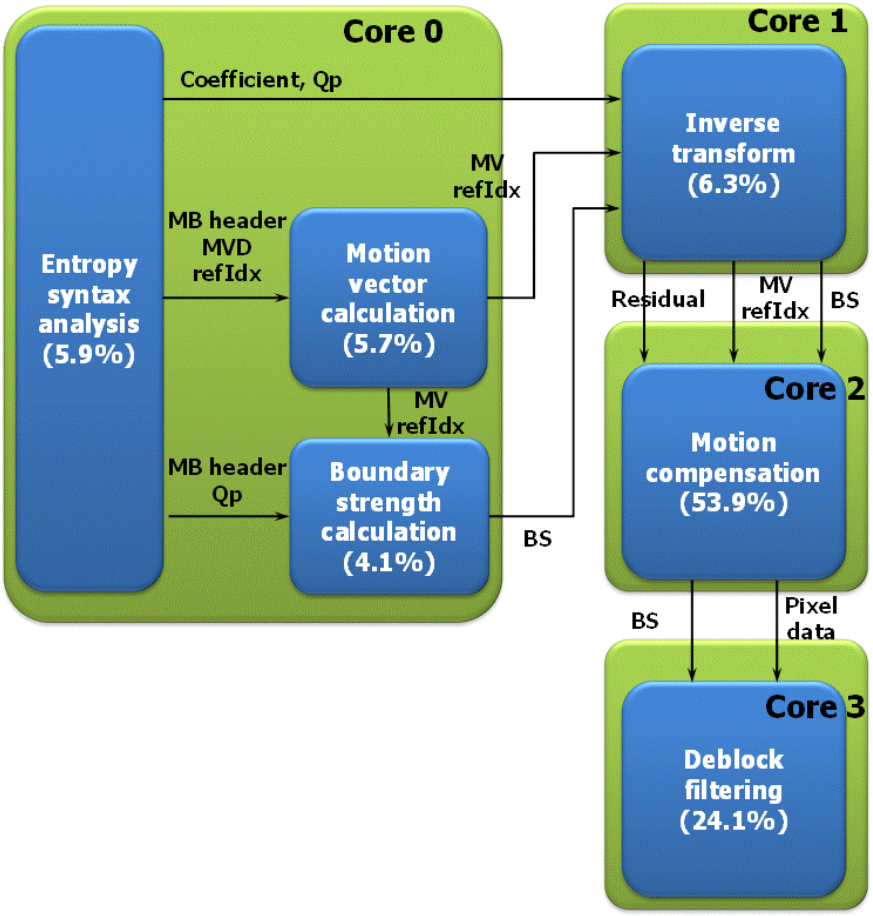


Figure 43: Initial functional partitioning and data flow between cores.

5.3 DATA FLOW AND MESSAGE STRUCTURE BETWEEN CORES

One of the important aspects of mapping a video decoder on an embedded quad-core system, is reducing the access to SDRAM by the cores. The RP core in our platform has scratch pad memory instead of a data cache, therefore

a lot of random access to external SDRAM by the core can cause a critical degradation of performance, because of the very long latency of SDRAM. Therefore, it is better to place the data buffers in internal scratch pad memory, instead of SDRAM, as much as possible. With initial partitioning, each core has its own input and output buffers inside for the MB level data. The MB level data processed by each core are stored in its own output buffer, and then those data are transferred to an input buffer of the next core, by DMA transfer. By doing this, almost all MB level data were processed inside of internal scratch pad memory, without access to external SDRAM. Figure 44 shows the structure of MB level data, the buffer distribution, and the data flow of each core. Inside the MB level data, there are data for control, MB header, MV (Motion Vector), refldx (Reference Index), Qp (Quantization Parameter), BS (Boundary Strength), and the coefficient (or pixel). The data for control include additional information, like the number of coefficients or similar for inter-core communication.

For initial functional partitioning, 3 major functions were assigned to core 0 to reduce the SDRAM bandwidth, even though the burden of core 0 was much higher than core 1. Those 3 functions (entropy syntax analysis, motion vector calculation, and boundary strength calculation) share the same data, especially the MV and refldx of the current and neighboring MBs. The MV and refldx data of neighboring MBs are stored in SDRAM. If different cores are allocated for those 3 functions separately, those cores would access SDRAM redundantly for the same data. By allocating core 0 for those 3 functions, initial partitioning can reduce the access to SDRAM. In our implementation, core 0 loads the data to internal scratch pad memory once, by DMA transfer. At initial functional partitioning, BSPU transfers the parsed MB level data to the internal scratch pad memory in core 0 directly, to reduce the access to SDRAM, as can be seen in Figure 44. In data partitioning, when BSPU parses the bitstream for a MB, it is not determined yet which core will be allocated for processing that MB level data, because of dynamic MB

assignment [4]. Therefore, MB level data produced by BSPU should be stored in SDRAM in data partitioning. To access the MB level data produced by BSPU, additional access to SDRAM is unavoidable. It consumes large SDRAM bandwidth, compared with initial partitioning, even though the DMA transfer routine is used to load the MB level data from SDRAM. The bandwidth loss by this additional access in data partitioning will be shown in section 5.7.

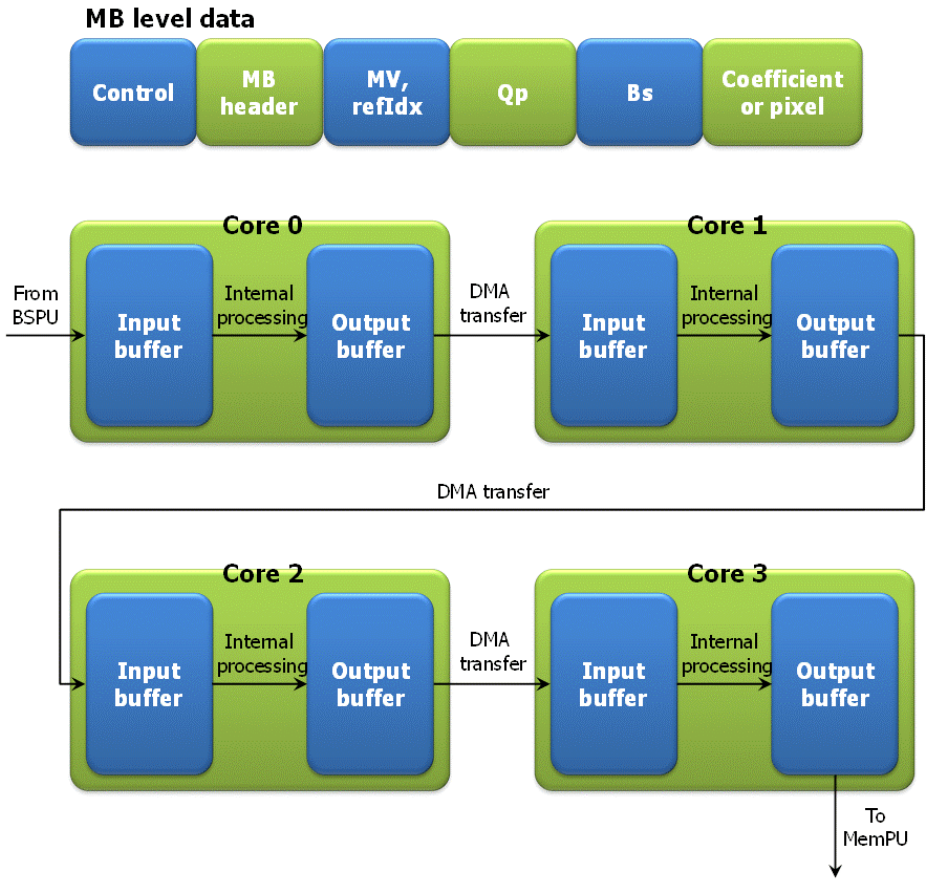


Figure 44: MB level data structure and buffers on each core. Each input or output buffer has multiple MB level data. The contents of 'coefficient or pixel' data can be changed by the operation of each core.

5.4 APPLYING INITIAL PARTITIONING

After determining initial functional partitioning, the FHD H.264 main profile decoder was ported on the real quad-core system implemented on the general-purpose FPGA (Field- Programmable Gate Array) board. The waiting overhead was very large, due to the unbalanced load between cores. Particularly, the load on core 2 or the motion compensation function was the main reason for the large waiting cycles of the other cores. Table XII shows the waiting overhead of each core, and Figure 45 shows the operation cycles and waiting cycles of the H.264 FHD decoder graphically. After applying initial partitioning, this problem must be resolved. From the viewpoint of load balancing, adapting data partitioning would be a solution. But data partitioning causes the MB level dependency problem, and it also requires additional SDRAM bandwidth and large program memory as mentioned in section 5.2 and 5.3. Therefore, adapting regular data portioning is not a proper solution.

TABLE XII

WAITING OVERHEAD OF INITIAL FUNCTIONAL PARTITIONING

	Functions	**Waiting overhead(MHz)**
Core 0	Entropy syntax analysis MV calculation BS calculation	**606**
Core 1	Inverse transform	**1,378**
Core 2	Motion compensation	**174**
Core 3	Deblock filtering	**1,011**

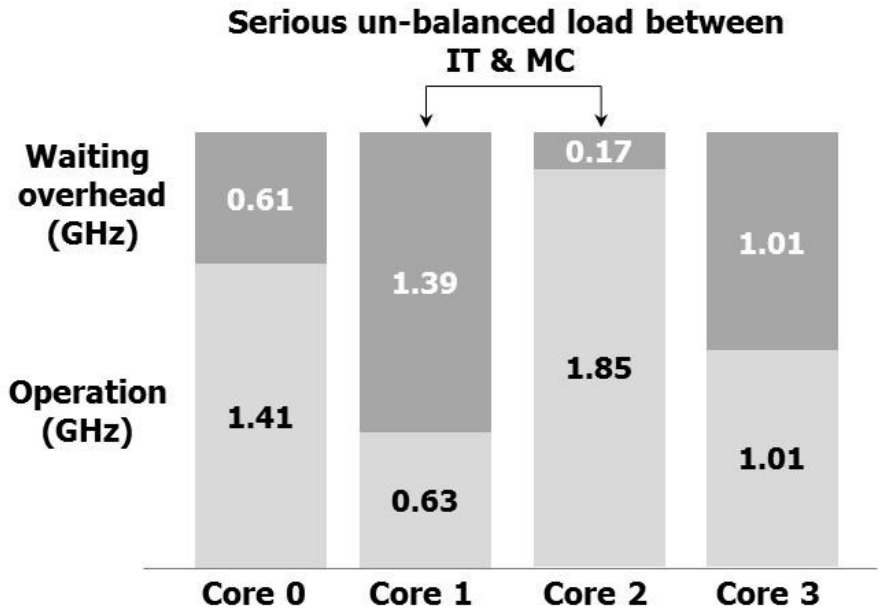


Figure 45: Operation and waiting overhead of initial partitioning.

5.5 HYBRID PARTITIONING

Instead of using regular data partitioning, the positive features of data partitioning and functional partitioning are considered by hybrid partitioning. If functional partitioning or data partitioning is used for each functional module separately, it may be possible to mix only the good features of each partitioning method. In the case of the "motion compensation" and "inverse transform" function, there is no MB level dependency problem [8]. Therefore, if data partitioning is applied on those functions only, it does not need to check the complex dependency of the neighboring MB. In addition, it seems possible to not break the whole structure and data flow of initial functional partitioning. This means that this new partitioning method does not require as

large MB level buffer as data partitioning. Now a novel partitioning method called “hybrid partitioning” is proposed in detail for embedded quad-core system, which is as powerful as data partitioning for load balancing, and as efficient as functional partitioning from the viewpoint of memory usage and SDRAM bandwidth. Figure 46 shows the detailed partitioning method. In hybrid partitioning, core 1 executes the inverse transform and motion compensation for odd MBs, and core 2 does the same for even MBs. The other cores do the same job as initial partitioning.

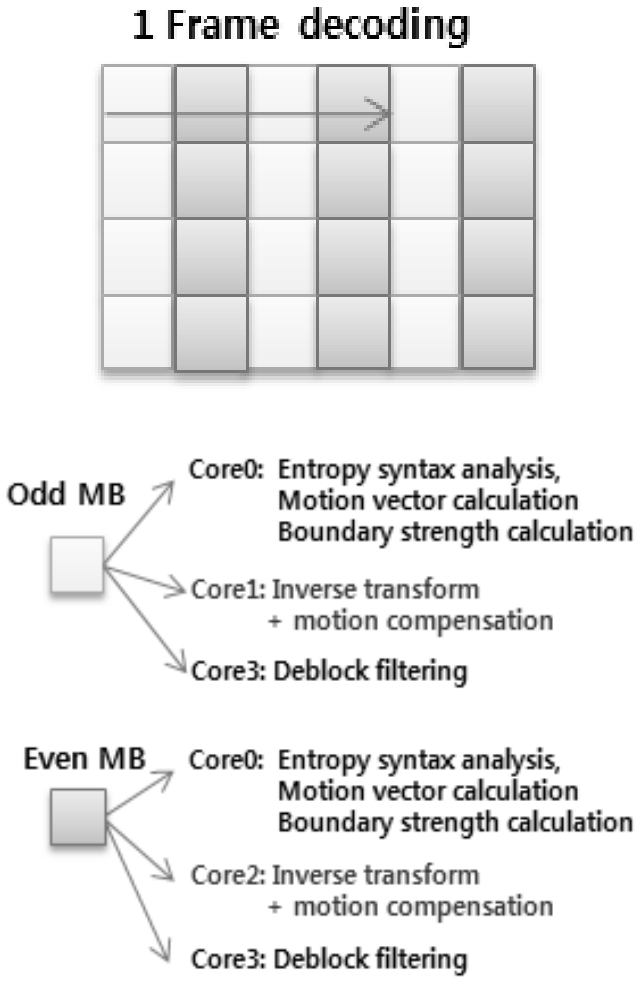


Figure 46: Hybrid partitioning of H.264 decoder

It can be expected that the proposed hybrid partitioning will change the computational loads of core 1 and core 2 to half the sum of the inverse transform load and motion compensation load. The expected value was 1.24GHz (half of 0.63GHz + 1.85GHz) on a real board. This is similar to the computational load of core 0 and core 3. From this number, it can be expected that the waiting cycles will be reduced significantly. It can also be expected that it will require almost the same SDRAM bandwidth as functional partitioning, because it has a similar data flow; and that it will require a little bit larger program memory than functional partitioning, because only core 1 and core 2 have the same functions as each other. The effects of hybrid partitioning will be shown in section 5.6.

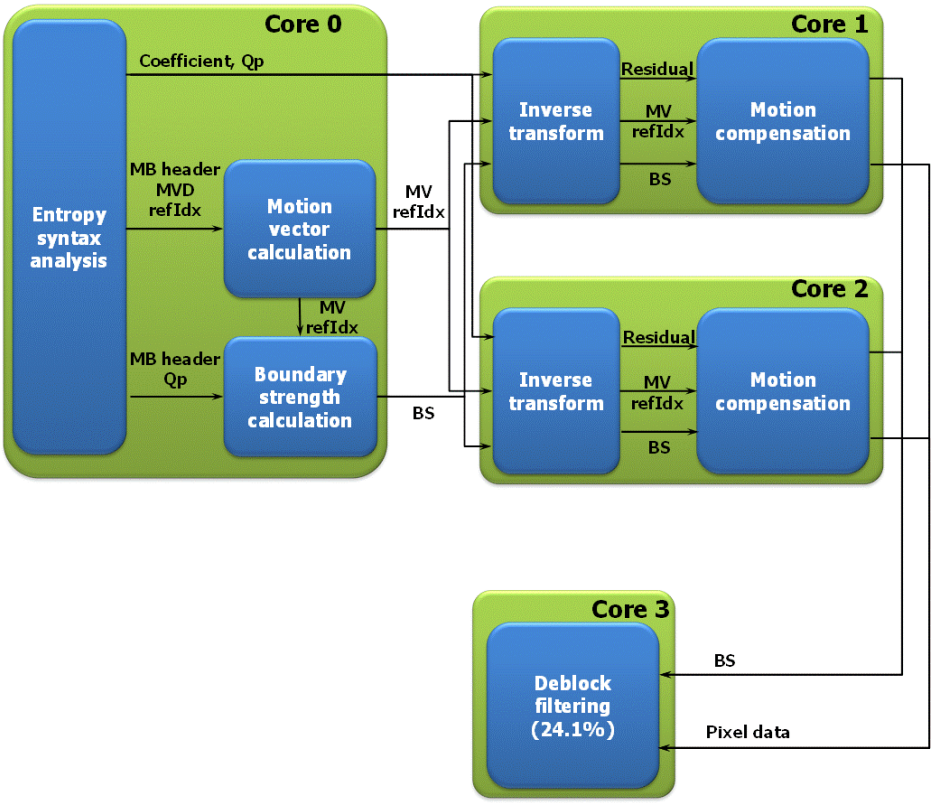


Figure 47: Role of each core and data flow between cores in hybrid partitioning

Figure 47 shows the data flow of the proposed hybrid partitioning for the H.264 decoder. The data flow of hybrid partitioning is similar to that of initial functional partitioning, except that core 0 sends the MB level data to core 1 or core 2 alternately, and core 3 receives MB level data from core 1 or 2 alternately. The MB level data processed by BSPU are transferred to the input buffer of core 0 directly as the initial partitioning. Therefore, the required SDRAM bandwidth is the same as for initial partitioning. While applying hybrid partitioning, the DMA transfer routine is also optimized. From the profile result on a real board, it was found that the waiting cycle for DMA operation was one of the major parts of the operation cycle, and this was not counted in the simulation stage. These waiting cycles can be easily reduced by parallelizing DMA. In the next section, experimental results of applying hybrid partitioning with simple DMA optimization will be shown.

5.6 EXPERIMENTAL RESULT

In this section, the result of applying the proposed hybrid partitioning will be shown. Table XIII shows the reduction of the waiting cycles by applying hybrid partitioning and simple DMA optimization. As a result of applying hybrid partitioning, the waiting overhead is reduced by 86.0% on average. This large reduction means that loads are evenly distributed to all cores. This result shows that the proposed hybrid partitioning is better than functional partitioning, from the viewpoint of load balancing. Figure 48 shows the overhead reduction from hybrid partitioning graphically

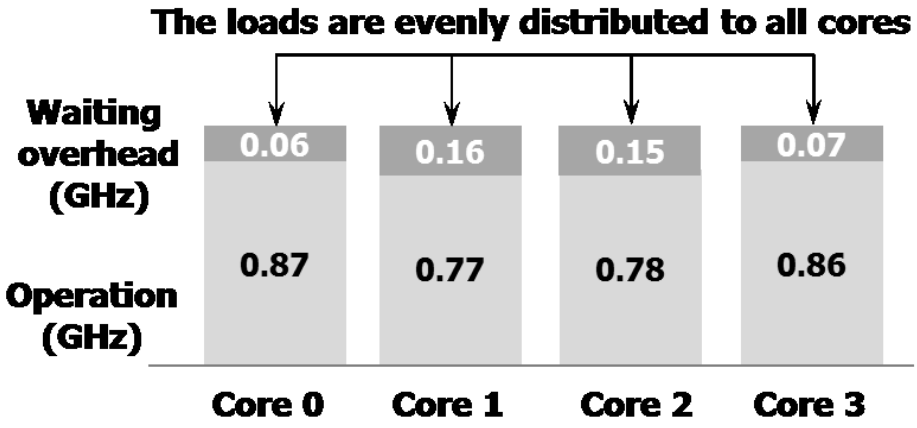


Figure 48: Operation and waiting overhead of hybrid partitioning

TABLE XIII

WAITING OVERHEAD REDUCTION BY HYBRID PARTITION

Conditions	Waiting overhead(MHz)				
	Core 0	Core 1	Core 2	Core 3	**Total**
W/O hybrid partition	606	1,378	174	1,011	**3,169**
W/ hybrid partition	61	165	144	74	**444**
Reduction ratio(%)	89.9	88.0	17.2	92.7	**86.0**

From the viewpoint of memory, hybrid partitioning requires less program memory and less SDRAM bandwidth than data partitioning. Tables XIV - XVI show the comparisons of program memory size of the three partitioning types, and the additional SDRAM bandwidth loss of data partitioning. Table XIV shows the comparison of VLIW program memory size of the three partitioning types. As can be seen in table XIV, hybrid partitioning requires just 17.1% more memory than functional partitioning, but data partitioning requires more than twice. In other words, hybrid partitioning requires 51.2% less memory than data partitioning. Table XV shows the comparison of the other program memory or CGRA program memory sizes of the three partitioning types. In the case of CGRA program memory, data partitioning requires much more memory than functional partitioning. Data partitioning requires exactly 4 times larger memory than functional partitioning. In the case of functional partitioning, each core typically does a different job, but there should be the same regular functions, like the DMA transfer routine, interrupt service routine, etc. These functions are usually stored in VLIW program memory. Therefore, the required size of VLIW program memory for data partitioning is not 4 times of that required for functional partitioning. In contrast to VLIW program memory, there are no regular functions in CGRA program memory, therefore the CGRA program memory size of data

TABLE XIV

COMPARISON OF VLIW PROGRAM MEMORY SIZE

Conditions	Program memory size (Kbytes)	Increasing ratio compared with functional partition(%)
Functional partition	1,124.1	0
Data partition	2,695.7	139.8
Hybrid partition	1,316.0	**17.1**

partitioning is 4 times larger than that of functional partitioning, because all cores would have all functions of the video decoder in data partitioning [4]. The proposed hybrid partitioning requires 62.0% less CGRA program memory than data partitioning. Table XVI shows the loss of SDRAM bandwidth in data partitioning by the additional SDRAM access for MB level data, described in section 5.2. Data partitioning requires 38.6MHz more SDRAM bandwidth than hybrid partitioning or functional partitioning, and this is 11.6% of the total bandwidth budget of 333MHz SDRAM used for the experiment.

After applying hybrid partitioning to reduce waiting overhead, many optimization methods were applied to achieve the 333MHz operation clock, which was the same as target system clock. Table XVII shows a few main methods used for optimization, and required cycles for decoding the FHD stream after applying those methods. MemPU was implemented after applying hybrid partitioning and it reduced a lot of cycles for accessing memory. Special instructions were designed for accelerating motion compensation and deblock filtering routines. After applying dynamic load balancing [2], the required cycles for FHD decoder became 328MHz. Finally, the H.264 FHD decoder with 30fps (frames per second) for 20Mbps stream was achieved on an embedded quad-core system, with a 333MHz system operation clock. The maximum decoding speed for a 20Mbps stream was 30.5fps on a quad-core system with a 333MHz system clock.

Table XVIII and Table XIX shows the precise profile of each core, after achieving the final goal of a 333MHz operation cycle. Mapping overhead due to multi-core mapping is 13.5% of the total cycles. The ‘boundary strength calculation’ is distributed to core 0, core 1, and core 2, for dynamic load balancing [2]. Table XX shows a comparison of the proposed method and the previous works. As can be seen, the increase of performance by parallelization with hybrid partitioning is 3.5. This large increase of performance is due to the low waiting overhead of the proposed method. This result is relatively

high compared to the previous works, but these increasing factors can vary depending on complicated conditions like parallelization methods, multi-core system configuration, architecture of used cores, and etc.

TABLE XV

COMPARISON OF CGA PROGRAM MEMORY SIZE

Conditions	Program memory size (Kbytes)	Increasing ratio compared with functional partition(%)
Functional partition	107.0	0
Data partition	428.0	300.0
Hybrid partition	162.6	**52.0**

TABLE XVI

LOSS OF SDRAM BANDWIDTH IN DATA PARTITIONING

Bandwidth loss for MB level data access (MHz)	**Ratio with total budget of 333MHz SDRAM (%)**
38.6	**11.6**

TABLE XVII

MAIN OPTIMIZATION METHODS AND REQUIRED CYCLES.

Optimization Methods	**Required Cycles for decoding FHD stream (MHz)**
Applying MemPU	**626**
Special Instructions	**412**
Dynamic Load Balancing	**328**

TABLE XVIII

PRECISE PROFILING RESULT – CORE 0, CORE 1

		Core 0	Core 1
Total(MHz)		**328.3**	**328.3**
Computation	**Sub total(MHz)**	**284.5**	**292.9**
	Entropy syntax analysis	77.5	0
	Motion vector calculation	165.2	0
	Boundary strength calculation	10.0	20.7
	Inverse transform	0	39.1
	Motion compensation	0	199.3
	Deblock filtering	0	0
	DMA wait	31.8	19.7
	MemPU wait	0	14.1
Mapping Overhead	**Sub total(MHz)**	**43.8**	**35.4**
	Wait previous core	0	5.9
	Wait next core	20.0	21.4
	Communication	23.8	8.1

TABLE XIX

PRECISE PROFILING RESULT – CORE 2, CORE 3

		Core 2	Core 3
Total(MHz)		**328.3**	**328.3**
Computation	**Sub total(MHz)**	**286.8**	**292.5**
	Entropy syntax analysis	0	0
	Motion vector calculation	0	0
	Boundary strength calculation	31.3	0
	Inverse transform	32.4	0
	Motion compensation	187.3	0
	Deblock filtering	0	264.2
	DMA wait	19.9	11.1
	MemPU wait	16.0	17.3
Mapping Overhead	**Sub total(MHz)**	**41.5**	**35.8**
	Wait previous core	10.3	35.8
	Wait next core	24.6	0
	Communication	6.5	0

TABLE XX

COMPARISON OF VARIOUS H.264 PARALLEL DECODERS

	Our approach	**Nishihara[8]**
Processor	RP[12]	ARM11 MPCore
Maximum Resolution	1920x1080	320x240
Bitrate of Bitstream	20Mbps	256Kbps
Number of Cores	4	4
Increase of Performance by parallelization	**3.5**	**2.2**
	Baik[9]	**Jo[10]**
Processor	Cell Processor	x86 processor
Maximum Resolution	1920x1080	1920x1088
Bitrate of Bitstream	2.5Mbps	N/A
Number of Cores	5	4
Increase of Performance by parallelization	**3.5**	**2.9**

CHAPTER 6

Concluding Remarks

In this paper we proposed the parallelization strategy for multi-core embedded system implementation including H.264/AVC decoder. Depending on system configurations and the features of applications, we can choose proper parallelization methods with the proposed parallelization strategy. The system configurations are illustrated by the number of cores, memory configurations, existence of hardware accelerators, and etc. As for the features of applications, complexity differences between functions and data dependency in applications can be examples. To show the validity of the proposed strategy, we applied the strategy to H.264/AVC decoder on the dual-core systems and quad-core system. We also proposed a novel dynamic load balancing method to overcome the problems in the case of applying functional partitioning, and a novel hybrid partitioning method to merge only good features of functional and data partitioning methods.

For H.24/AVC decoder on the dual-core system, we applied the functional partitioning method. The proposed partitioning is suitable for dual or quad-core system with a few HWAs. By controlling each HWAs with only one core, the proposed partitioning enables HWAs to have simple communication interface. The proposed partitioning makes it possible to use

the small sized inter-core communication buffer including only tens of MB data for FHD decoding, and it is very small compared with the previous works. To improve the performance of the H.264/AVC decoder parallelized by the proposed functional partitioning method on the dual-core system, we also applied the dynamic load balancing method. Because the dynamic load balancing is very general strategy, we should find a proper load balancing method and proper criteria for dynamic decision. For this purpose, we assigned inverse-transform and dequantization functions for intra MBs, and boundary strength calculation function for inter MBs dynamically depending on communication buffer level. The proposed dynamic load balancing method can reduce 82% of waiting overhead due to unbalanced load. This good result depends on the proper analysis of the relationship between the waiting overhead and dynamic change of MB type. In addition to good performance, the good feature of the proposed method is a simple decision criterion. Because the buffer level between cores is enough as decision criterion, the load balancing process is simple and does not waste core performance. In addition to that, it is easy to extend the proposed method to multi-core systems.

In addition to the dynamic load balancing method, we also proposed a novel partitioning method called hybrid partitioning method, and we applied the partitioning method to H.264/AVC decoder on quad-core system. We also proposed the hybrid partitioning method for the FHD H.264 decoder. The proposed partitioning method is as powerful as data partitioning for load balancing, and as efficient as functional partitioning from the viewpoint of memory usage. By applying hybrid partitioning with DMA parallelization, 86.0% of the total waiting cycle was reduced. Whereas hybrid partitioning resolves the load balancing problem, like data partitioning, it requires 51.2% less VLIW program memory and 62.0% less CGRA program memory, than data partitioning. Hybrid partitioning also conserves the SDRAM bandwidth of 38.6MHz, compared with data partitioning. The parallelized decoder with

hybrid partitioning on an embedded quad-core system was 3.5 times faster than that on a single core. This result depends on the proper analysis of the H.264 decoder, and the features of each module, like dependency properties. By selecting the proper partitioning method for each module, the waiting cycles were minimized to almost ideal values. By applying many kinds of optimization methods, like dynamic load balancing etc., a 30fps H.264 main profile FHD decoder on the quad-core system with 333MHz was finally achieved. In the future, a dynamic MB assignment will be applied to the proposed partitioning method. The dynamic MB assignment means assigning MBs to core 1 or core 2 dynamically, instead of the current odd and even MB assignment. This is different from the dynamic load balancing on functional partitioning [2], which was already applied. The proposed method will be extended to a 16-core system for UHD (Ultra-High Definition) resolution video.

Bibliography

[1] Minsoo Kim, Joon Ho Song, Do-Hyung Kim, and Shihwa Lee, “Hybrid partitioned H.264 full high definition decoder on embedded quad-core”, *IEEE Transactions on Consumer Electronics*, Vol. 58, No. 3, pp. 1038 – 1044, August 2012.

[2] Minsoo Kim, Joonho Song, DoHyung Kim, and Shihwa Lee, “H.264 Decoder on embedded dual core with dynamically load-balanced functional partitioning”, *Proceedings of 17th IEEE International Conference on Image Processing (ICIP)*, pp. 3749-3752, September 2010.

[3] Minsoo Kim, Joon Ho Song, Do-Hyung Kim, and Shihwa Lee, “Hybrid partitioned H.264 full high definition decoder on embedded quad-core”, *Proceedings of IEEE International Conference on Consumer Electronics (ICCE)*, pp. 279-280, January 2012.

[4] Erik B. van der Tol, Egbert G.T. Jaspers, and Rob H. Gelderblom, “Mapping of H.264 Decoding on a Multiprocessor Architecture”, *Proceedings of SPIE Conference on Image and Video Communications and Processing*, pp. 707-718, 2003.

[5] Cor Meenderinck, Arnaldo Azevedo, Mauricio Alvarez, Ben Juurlink, and Alex Ramirez, "Parallel Scalability of H.264", *Proceedings of the first Workshop on Programmability Issues for Multi-Core Computers*, January 2008.

[6] Yun-il Kim, Jong-Tae Kim, Sehyun Bae, Hyunki Baik, and Hyo Jung Song, "H. 264/AVC decoder parallelization and optimization on asymmetric multicore platform using dynamic load balancing", *Proceedings of IEEE International Conference on Multimedia and Expo*, pp. 1001— 1004, 2008.

[7] Jike Chong, Nadathur Satish, Bryan Catanzaro, Kaushik Ravindran, and Kurt Keut, "Efficient parallelization of h. 264 decoding with macro block level scheduling", *Proceedings of IEEE international conference on multimedia and expo*, pp. 1874-1877, 2007.

[8] Kosuke Nishihara, Atsushi Hatabu, and Tatsuji Moriyoshi, "Parallelization of H.264 video decoder for embedded multicore processor", *Proceedings of IEEE International Conference on Multimedia and Expo*, pp. 329-332, 2008.

[9] Hyunki Baik, Kue-Hwan Sihn, Yun-il Kim, Sehyun Bae, Najeong Han, and Hyo Jung Song, "Analysis and Parallelization of H.264 decoder on Cell Broadband Engine Architecture", *Proceedings of IEEE International Symposium on Signal Processing and Information Technology*, pp. 791-795, 2007.

[10] Song Hyun Jo, Seongmin Jo, and Yong Ho Song, "Efficient coordination of parallel threads of H.264/AVC decoder for performance improvement", *IEEE Transaction on Consumer Electronics*, vol. 56, no 3, pp. 1963-1971, 2010.

[11] Won-Jin Kim, Keol Cho and Ki-Seok Chung, “Stage-based frame-partitioned parallelization of H.264/AVC decoding”, *IEEE Transaction on Consumer Electronics*, vol. 56, no 2, pp. 1088-1096, 2010.

[12] ITU-T VCEG and ISO/IEC MPEG, “Advanced video coding for generic audiovisual services”, *ITU-T Recommendation H.264 and ISO/IEC 14496-10(MPEG-4 AVC)*, Version 5, Feb. 2009.

[13] Bingfeng Mei and Andy Lambrechts, Jean-Yves Mignolet, Diederik Verkest, and Rudy Lauwereins, “Architecture Exploration for a Reconfigurable Architecture Template,” *IEEE Design & Test of Computers*, vol. 22, no 2, pp. 90-101, Mar. 2005.

[14] Thomas Wiegand, Gary J. Sullivan, Gisle Bjøntegaard, and Ajay Luthra, “Overview of the H.264/AVC Video Coding Standard,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13 , no 7, 560 – 576, July 2003.

[15] David Bell, Greg Wood, “Multicore Programming Guide”, *Application Report SPRAB27A*. Texas Instruments, 2009.

[16] C. C. Chi, “Parallel h.264 decoding strategies for cell broadband engine”, Master's thesis, Computer Engineering group, Delft University of Technology, the Netherlands, 2009.

[17] Moore, Gordon E, “Cramming more components onto integrated circuits”, *Electronics Magazine*, vol. 38, Number 8, April 19, 1965.

[18] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki, "Synergistic Processing in Cell's Multicore Architecture”, *IEEE Micro*, vol. 26, Issue 2, pp. 10-24, 2006.

[19] Maria Koziri, Dimitrios Zacharis, Ioannis Katsavounidis, Member, and Nikos Bellas, "Implementation of the AVS Video Decoder on a Heterogeneous Dual-Core SIMD Processor", *IEEE Transactions on Consumer Electronics*, Vol. 57, No. 2, pp. 673-681, May 2011.

[20] Florian H. Seitner, Michael Bleyer, Ralf M. Schreier, and Margrit Gelautz, "Evaluation of data-parallel splitting approaches for H.264 decoding", *Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia*, pp. 40-49, 2008.

[21] Mauricio Alvarez, Alex Ramirez, Mateo Valero, Arnaldo Azevedo, Cor Meenderinck, and Ben Juurlink, "Performance evaluation of macroblock-level parallelization of H.264 decoding on a cc-NUMA multiprocessor architecture", *4th Colombian Computing Conference*, pp. 108-117, 2009.

[22] Guan Hui and Wang Hongpeng, "Research of parallel decoding algrithm in h. 264 on tile64", *2nd IEEE International Conference on Broadband Network & Multimedia Technology*, pp. 500-503, October 2009.

[23] Ding-Yun Chen, Chen-Tsai Ho, Chi-Cheng Ju, and Chung-Hung Tsai, "A novel parallel H.264 decoder using dynamic load balance on dual core embedded system", *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2313 – 2316, March 2012.

[24] Yongdong Zhang, Chenggang Yan, Feng Dai, and Yike Ma, "Efficient Parallel Framework for H.264/AVC Deblocking Filter on Many-Core Platform", *IEEE Transactions on Multimedia*, Vol. 14, No. 3, pp. 510-524, June 2012.

[25] Won Chang Lee, Joon Ho Song, Do-Hyung Kim, and Shihwa Lee, "Memory processing unit in video decoding system", *Proceedings of IEEE International Conference on Consumer Electronics (ICCE)*, pp. 538-539, January 2012.

[26] Joon Ho Song, Won Chang Lee, Do-Hyung Kim, and Shihwa Lee, "Low-power video decoding system using a reconfigurable processor", *Proceedings of IEEE International Conference on Consumer Electronics (ICCE)*, pp. 532-533, January 2012.

[27] Sangjo Lee, Joonho Song, Minsoo Kim, Dohyung Kim, and Shihwa Lee, "H.264/AVC UHD decoder implementation on multi-cluster platform using hybrid parallelization method", *Proceedings of 18th IEEE International Conference on Image Processing (ICIP)*, pp. 381 - 384, September 2011.

[28] Arnaldo Azevedo, Cor Meenderinck, Ben Juurlink, Andrei Terechko, Jan Hoogerbrugge, Mauricio Alvarez, and Alex Ramirez, "Parallel H. 264 decoding on an embedded multicore processor", *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HIPEAC)*, January 2009.

[29] Mauricio Alvarez Mesa, Alex Ramirez, Arnaldo Azevedoz, Cor Meenderinckz, Ben Juurlinkz, and Mateo Valero, "Scalability of Macroblock-level Parallelism for H.264 Decoding", *Proceedings of the 15th International Conference on Parallel and Distributed Systems*, 2009.

[30] Kue-Hwan Sihn, Hyunki Baik, Jong-Tae Kim, Sehyun Bae, and Hyo Jung Song, "Novel approaches to parallel H.264 decoder on symmetric multicore systems", *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2017 - 2020,

April 2009.

[31] Arnaldo Azevedo, Ben Juurlink, Cor Meenderinck, Andrei Terechko, Jan Hoogerbrugge, Mauricio Alvarez, Alex Ramirez, and Mateo Valero, "Highly Scalable Parallel Implementation of H.264", *Transactions on High-Performance Embedded Architectures and Compilers IV*, Lecture Notes in Computer ScienceVolume 6760, pp 111-134, 2011.

[32] Chi Ching Chi and Ben Juurlink, "A QHD-capable parallel H.264 decoder", *Proceedings of the international conference on Supercomputing*, pp. 317-326, 2011.

[33] J. Y. Lee, J. J. Lee, and S. M. Park, "Multi-core platform for an efficient H. 264 and VC-1 video decoding based on macroblock row-level parallelism", *IET Circuits, Devices & Systems*, Vol. 4 , No. 2, pp. 147-158, March. 2010.

[34] Florian H. Seitner , Michael Bleyer, Margrit Gelautz, and Ralf M. Beuschel, "Evaluation of data-parallel H.264 decoding approaches for strongly resource-restricted architectures", *Multimedia Tools and Applications*, Vol. 53, No. 2, pp 431-457, June 2011.

국문 초록

본 논문에서는 임베디드 다중 코어 시스템에서의 H.264/AVC 복호기의 병렬화 구현을 다룬다. 해당 목적을 위하여 임베디드 다중 코어 시스템을 위한 병렬화 전략을 제안한다. 제안된 병렬화 전략은 H.264/AVC 복호기 뿐만 아니라, 임베디드 다중 코어 시스템에서의 일반적인 응용 프로그램의 병렬화에도 또한 적용이 가능하다. 여기에 더하여, 동적 부하 분배와 하이브리드 파티션이라 명명된 두 가지 추가적인 병렬화 기법도 제안한다. 제안된 방법의 타당성 검증을 위해 H.264/AVC 복호기를 두 가지의 다중코어 시스템에서 구현하였다. 하나는 3가지의 하드웨어 가속기를 가지는 듀얼 코어 시스템이고, 나머지 하나는 두 개의 코프로세서를 지닌 쿼드 코어 시스템이다.

듀얼 코어 시스템에서 H.264/AVC 복호기는 다수의 하드웨어 가속기와 함께 제안된 병렬화 전략에 의해 병렬화되었다. 해당 시스템을 위하여, 병렬화 전략에 따라 기능적 파티셔닝이 선택되었으며, 이는 단순한 인터페이스를 가지는 하드웨어 가속기와 코어간 통신을 위한 메모리의 소형화를 가능케 한다. 기능적 파티셔닝을 위해서 동적 부하 분배 기법이 제안되었는데, 이는 선택된 일부의 함수들을 매크로블럭 레벨에서 다수의 코어에 동적으로 분배하는 방식이다. 이 경우 해당 함수를 어느 코어에서 구동할 지의 여부를 결정함에 있어 단지 버퍼 레벨 정보를 이용하기만 하면 충분하다. 이런 단순한 결정 방법을 채택함으로써, 부하분배 결정에 의한 코어의 성능 저하는 미미한 수준이 될 수 있으며 다중 코어로의 확장 또한 용이하다. 실험을 통하여 제안된 동적 부하 분배 기법이 대기 오버헤드의 82.3%를 절감함을 확인할 수 있었다.

쿼드 코어 시스템을 위해서는 제안된 병렬화 전략을 적용하여 하이브리드 파티셔닝이라는 새로운 파티셔닝 방법을 제안하였다. 파티셔닝은 다중 코어 시스템에서의 응용 프로그램의 병렬화를 위한 매우 중요한 이슈이다. 본 논문에서는 기능적 파티셔닝과 데이터 파티셔닝을 결합한 하이브리드 파티셔닝 기법을 제안한다. 각각의 기능적 모듈들은 각 모듈의 특성에 따라 기능적 파티셔닝 혹은 데이터 파티셔닝에 따라 분할 된다. 기능적 파티셔닝 및 데이터 파티션닝과 비교하여, 하이브리드 파티셔닝은 부하 분배의 측면에서는 데이터 파티셔닝만큼 효과적이며, 메모리 사용의 측면에서는 기능적 파티셔닝만큼 효율적이다. 또한 하이브리드 파티셔닝은 비디오 복호기에서 데이터 파티셔닝을 적용할 때 일반적으로 발생하는 매크로블럭 단위의 의존성 문제로부터도 자유롭다는 장점을 지닌다. 하이브리드 파티셔닝을 적용한 결과, 기능적 파티셔닝과 비교하여 86.0%의 대기 오버헤드가 감소하였다. 메모리 사용량 측면에 있어서는, 데이터 파티셔닝과 비교하여 51.2% 더 적은 VLIW 프로그램 메모리와, 62.0% 더 적은 CGRA 프로그램 메모리를 요구하였다. SDRAM 대역폭 측면에서는, 333MHz SDRAM 대역폭 기준으로, 데이터 파티셔닝과 비교하여, 11.6% 더 적은 대역폭을 사용하였다.

주요어: 임베디드 멀티코어 시스템, H.264/AVC, 병렬화 전략, 동적 부하 분배, 하이브리드 파티셔닝, 기능적 파티셔닝, 데이터 파티셔닝.

학번: 2006-30843