Contents

List of Figures

List of Tables

Abbreviations

CABAC	Context-based Adaptive Binary Arithmetic Coding
CAVLC	Context-based Adaptive Variable Length Coding
CBR	Constant Bit Rate
CPB	Coded Picture Buffer
DPB	Decoded Picture Buffer
DUT	Decoder under test
FIFO	First-In, First-Out
HRD	Hypothetical Reference Decoder
HSS	Hypothetical Stream Scheduler
IDR	Instantaneous Decoding Refresh
LSB	Least Significant Bit
MB	Macroblock
MBAFF	Macroblock-Adaptive Frame-Field Coding
MSB	Most Significant Bit
NAL	Network Abstraction Layer
RBSP	Raw Byte Sequence Payload
SEI	Supplemental Enhancement Information
SODB	String Of Data Bits
SVC	Scalable Video Coding
UUID	Universal Unique Identifier
VBR	Variable Bit Rate
VCL	Video Coding Layer
VLC	Variable Length Coding
VUI	Video Usability Information

Chapter 1. *Introduction*

H.264/AVC [1] has been the state of the art video compression standard of the ITU-T Video Coding Experts Group and ISO/IEC Moving Picture Experts Group (MPEG) in current video applications. It promises to outperform the earlier MPEG-4 and H.263 standard, employing many better innovative technologies such as multiple reference frame, variable block size motion estimation, in-loop de-blocking filter and context-based adaptive binary arithmetic decoding. H.264/AVC system can save the bit-rate up to 50% compared to the previous video standard such as H.263 and MPEG-4 under the same quality. Because of its high quality and compression gain technology, the more livelihood application products such as digital camera, video telephony and portable DVD player adopt H.264/AVC as its video standard as well. H.264/AVC contains two entropy decoders. One is Context-based Adaptive Variable Length Coding (CAVLC), and the other is Context-based Adaptive Binary Arithmetic Coding (CABAC) [3]. CABAC can achieve 9% to 14% bit-rate saving in average compared with CAVLC.

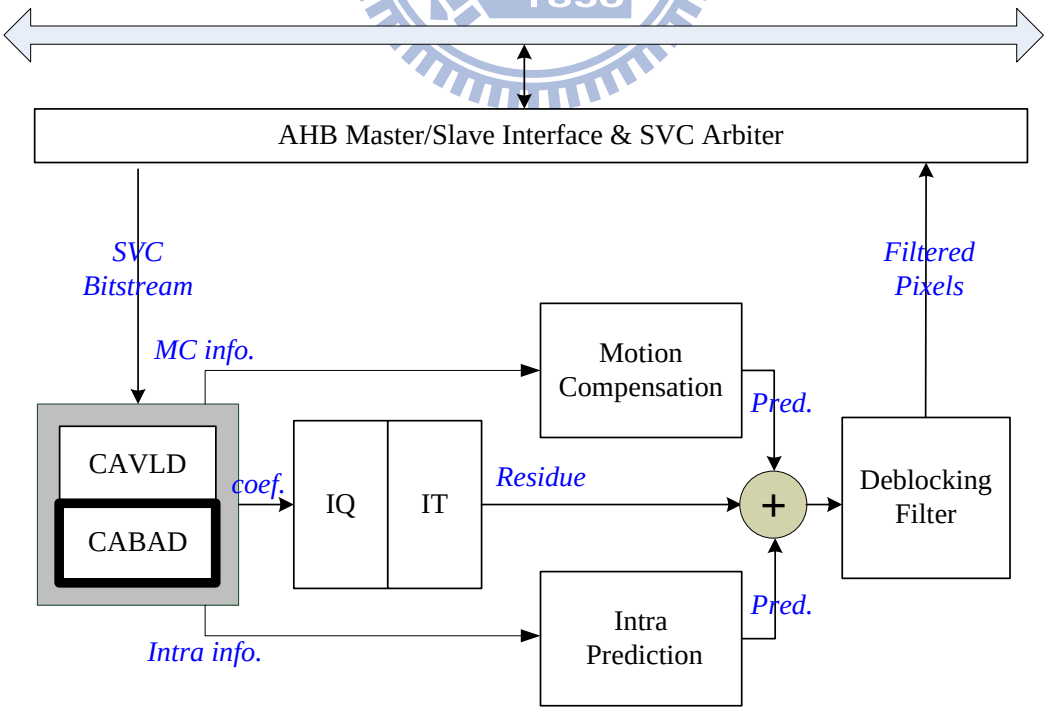


Figure 1. Block Diagram of H.264/AVC Decoder

Figure 1 shows the block diagram of H.264/AVC decoder (our system specification shows in Appendix A). The H.264/AVC has three profiles such as baseline, main and high for supporting varied video applications. The baseline profile adopts VLD to decode the MB information and the pixels coefficients which contains the universal variable decoder (UVLD) and the context-based adaptive variable length decoder (CAVLD). UVLD is one of VLD in baseline profile. It decodes not only the MB information such as the *mb_type, coded_block_pattern, intra_pred_mode,* and so on, but also the MB coefficient such as *mvd*. Because the residual data decoding occupies over 50% of the entire execution time, the residual coefficients are computed by the CAVLD architecture of the more efficiency. When it supports except baseline profile, the decoder has an advance choice except VLD. CABAD can be used in place of UVLD and CAVLD. Thus, H.264 system just needs CABAD to decode all MB information and pixel data if entropy decoding flag is assigned to CABAD.

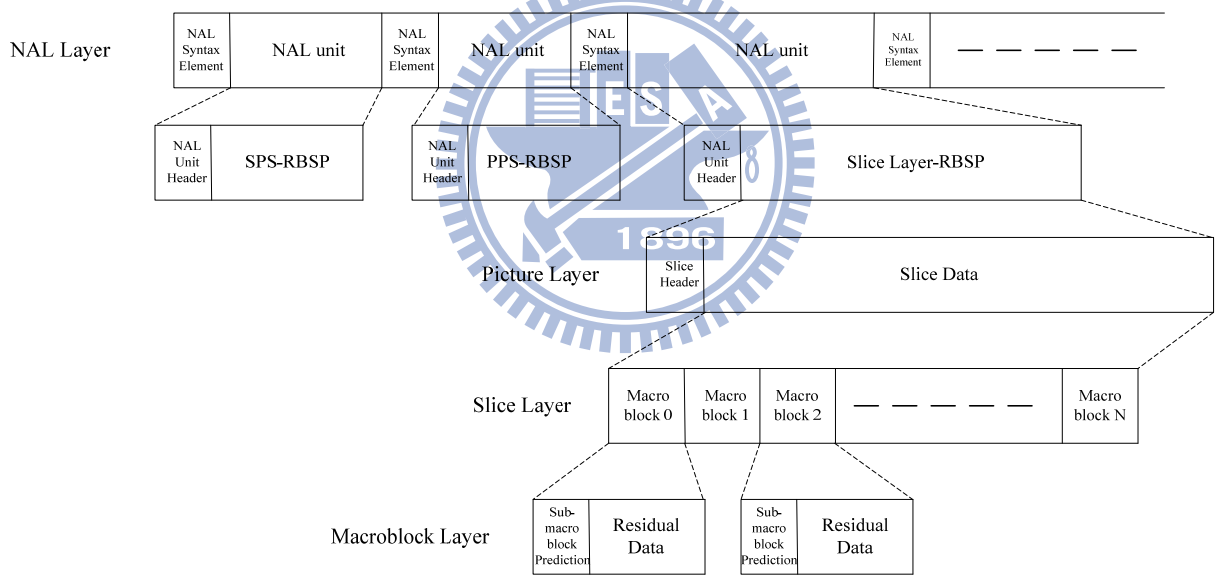


Figure 2. Bit-stream structure of H.264/AVC

In normal system architecture, the block of syntax parser employs in decoding the bit-stream on NAL layer, picture layer, and slice layer, given as Figure 2. Syntax element parser is also the top module to control all sub-system such as CABAD, VLD, intra-prediction, inter-prediction, IDCT, and so on. Hence, CABAD is the passive unit and is requested by the syntax parser and decodes the bit-stream of the macro block layer in Figure 2. The bit-stream is also fetched through the syntax element parser gets from bit-stream SRAM.

1.1 Overview of CABAC Decoding flow

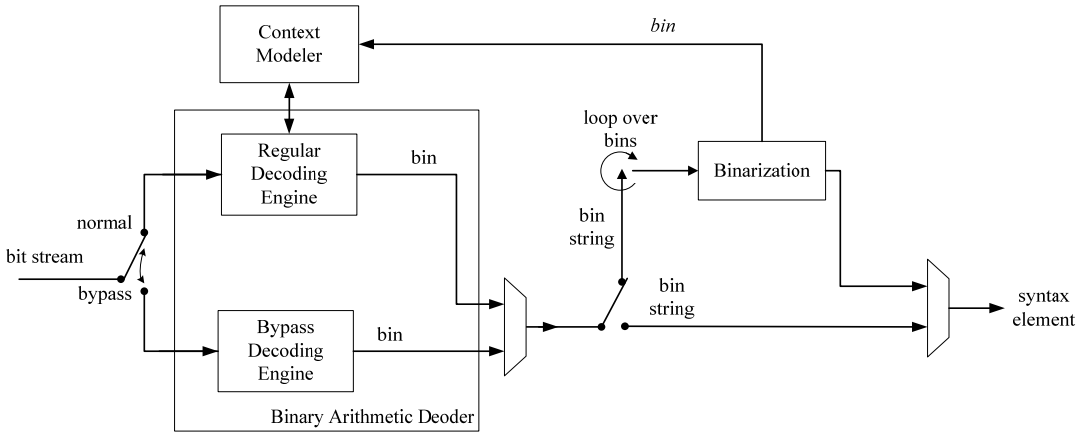


Figure 3. Block Diagram of CABAC decoding flow

In this Section 1.1, we introduce each building block of CABAC decoder and the execution flow of the CABAC decoder system. Before introducing the decoder, we have to explain the organization of bit-stream. In encoder side, all SEs of the H.264/AVC will be transferred into the binary code "bin" except flag-type SEs by binarization at first. And then, the transferred bin string encodes to the bit-stream by the binary arithmetic coder currently. So, it consists of two level decoders such as binary arithmetic decoder (AD), the de-binarization (DB) in decoder side. The binary arithmetic decoder has three different types such as regular, bypass, and terminal decoding processes. We don't show the terminal decoding process in Figure 3, because it is seldom applied in CABAC decoder system. At the beginning, the entire probabilities of the context model have to be initialized by the context model initial table when the new slice starts. In Figure 4 it has two decoding flows among the dotted lines. The first decoding flow is the arithmetic decoder which is the first stage of decoding one syntax element. It produces the bin value depending on the current range (*codlRange*) and the current value (*codlOffset*). The second decoding flow is the binarization engine. It reads the bin values to judge if the bin string forms the meaningful data. If not, the binarization engine requests the arithmetic decoder to decode one bin again and re-judges the bin string until identifying the value of the current syntax element. If completing the current slice, *codlRange* is assigned to "51210" and *codlOffset* is refilled in 9-bit bit-stream from the H.264/AVC system.

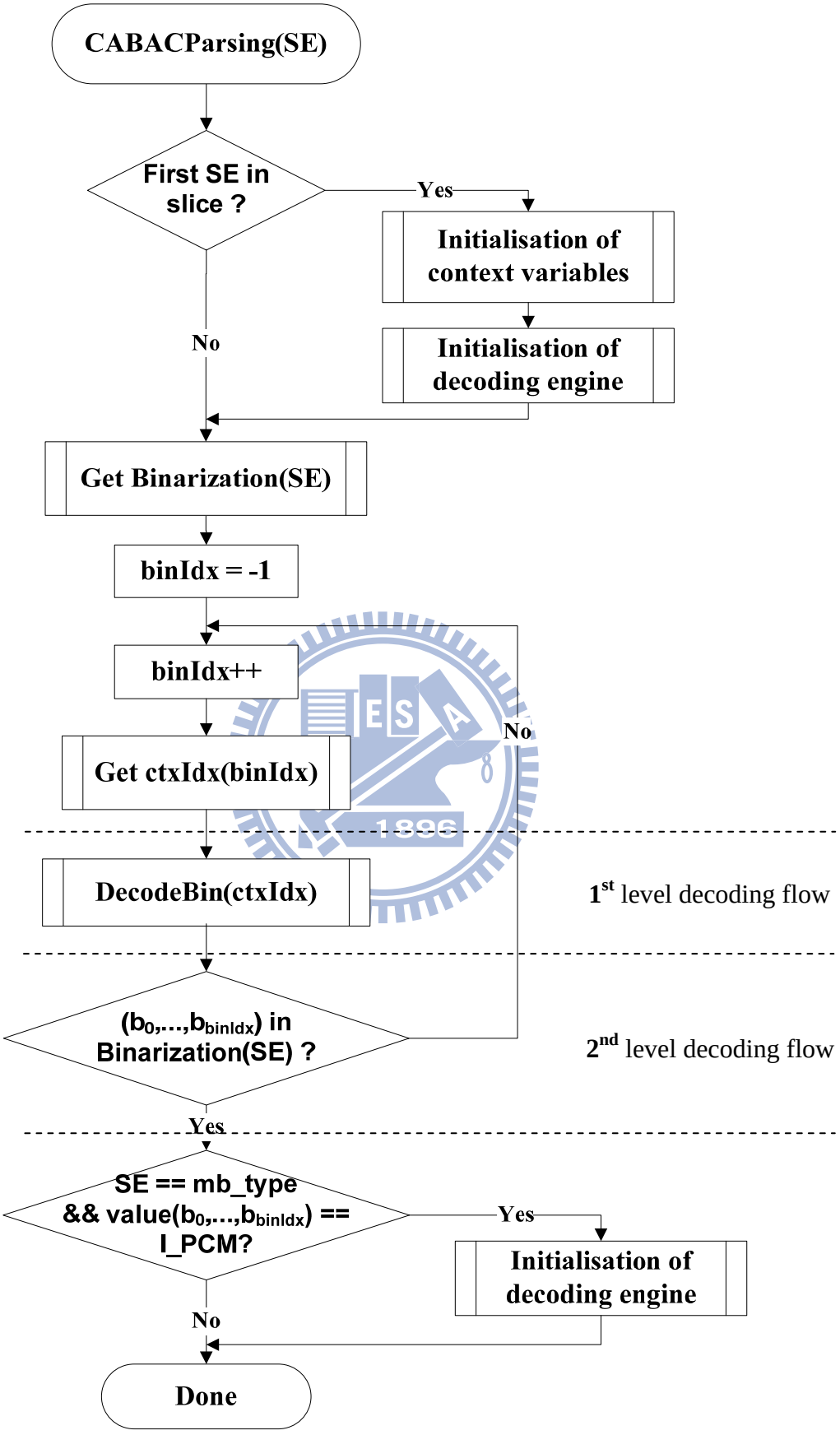


Figure 4. The flow chart of the CABAC decoding [1]

On the other hand, we collect all SEs which invoke CABAC decoder and their possible branches in Figure 5. Typically, we have four kinds of SEs including slice data, MB layer, (sub) MB pred and residual block cabac. Slice data and MB layer produce once time per macro block. (sub) MB pred and residual block cabac are produced according to block size. Therefore, we may often change our decoding order because of variable macro block type.

In slice data, we have three syntax elements such as *mb_skip_flag*, *mb_field_decoding_flag* and *end_of_slice_flag*. The *mb_field_decoding_flag* is used to recognize frame and field MB, and we produce once per MB pair. The *end_of_slice_flag* is always symbolized final syntax element of MB, and the slice will be finished when *end_of_slice_flag* equal to one. Besides, if the *mb_skip_flag* equal to one, we directly jump to *end_of_slice_flag* and skip this MB.

In MB layer, we have four syntax elements such as *mb_type*, *transform_size_8x8_flag*, *coded_block_pattern* and *mb_qp_delta*. We can recognize current block in which block size by *mb_type* and *transform_size_8x8_flag*. The *mb_qp_delta* is a parameter for inverse-quantization, and *coded_block_pattern* are represented zero distribution of residual block.

After decoding value of mb_type, we can depend on block size to judge the following status which will be mb_pred or sub_mb_pred. If we decode in sub_mb_pred, we may produce *sub_mb_type* to recognize sub-block size. And then, we may decode one or more predictor modes such as *prev_intraNxN_pred_mode_flag*, *rem_intraNxN_pred_mode*, *intra_chroma_pred_mode*, *ref_idx_lX* and *mvd_lX* for Intra or Inter predictor. ($N \in 4, 8$; $X \in 0, 1$)

Finally, we would decode the coefficient (coeff.) block in residual block cabac. The coeff. block size can be categorized into 4x4 and 8x8. So, we can get sixteen or four coeff. blocks in macro block. Different to residual block cavlc, we have to know zero coeff. position zero early. The *coded_block_pattern* may describe situation of each 8x8 block, and the *coded_block_flag* may describe that current 4x4 block contains all zero or not. After that *significant_coeff_flag* and *last_significant_coeff_flag* will scan all coeff. positions, and the *coeff_abs_level_minusl* and *coeff_sign_flag* produce the value of coeff. position which isn't equal to zero.

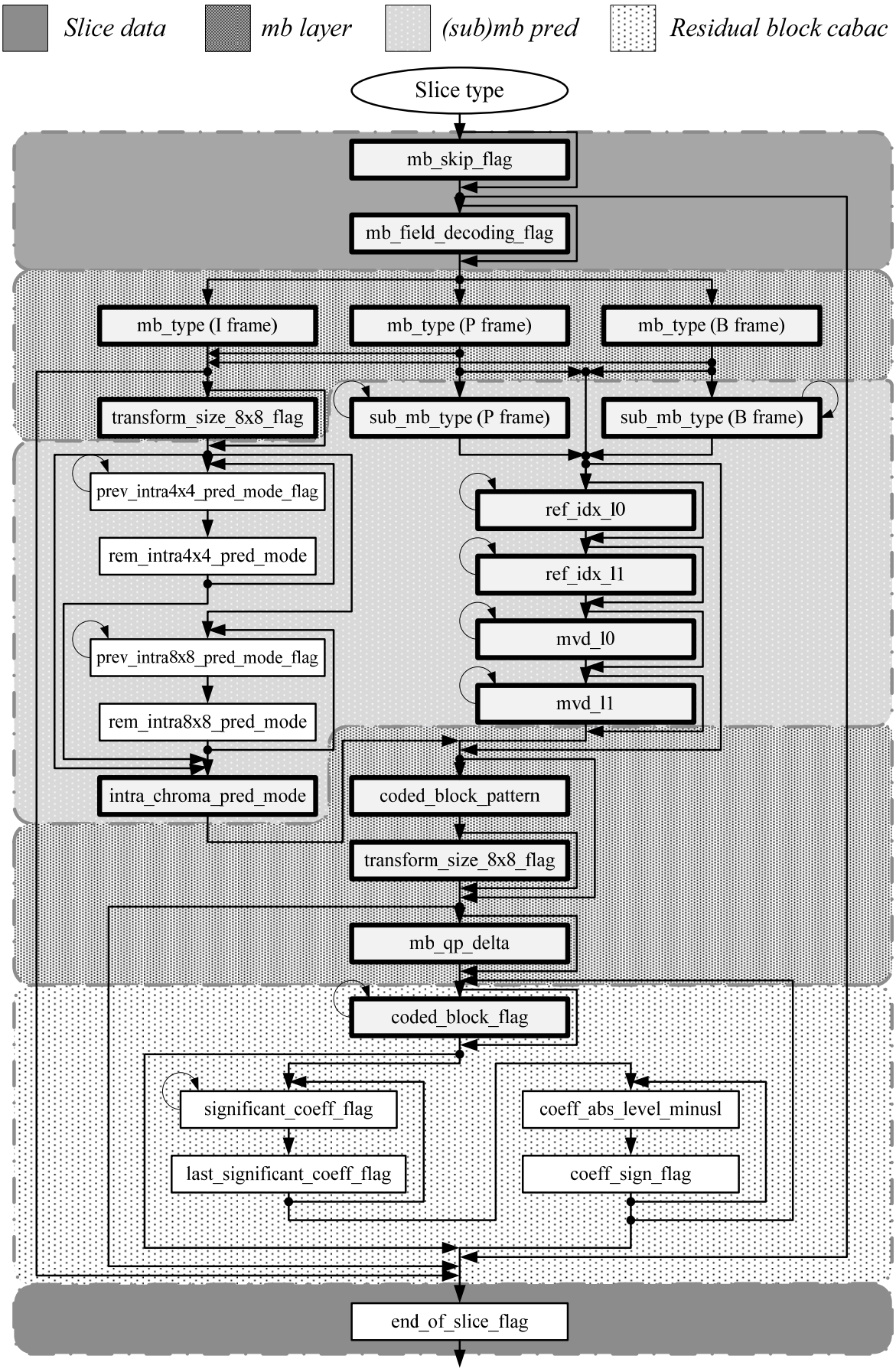


Figure 5. The flow chart of the syntax elements switching

1.2 Motivation and Design Challenges

However, the bottleneck of CABAC decoder design is the throughput for the H.264/AVC system. The arithmetic decoder pipelining is the major task for CABAC decoder. In Figure 6, the next range and value depend on current range and offset, and the table is controlled by outputted bin. So, it has notably strong data dependency to restrict throughput. Even if a DSP processor can work at 3GHz, it would be difficult to achieve the real-time CABAC decoding for HD video at 30 fps.

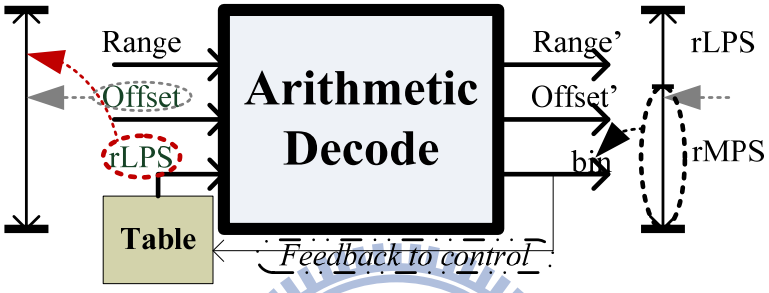
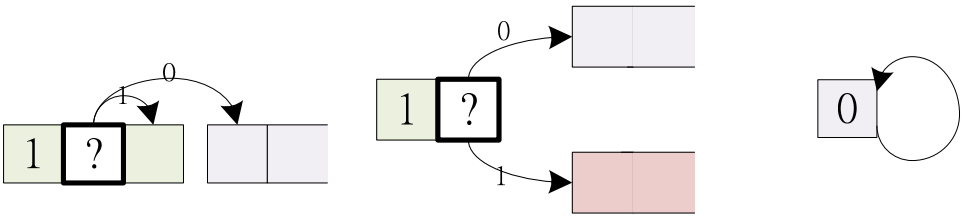


Figure 6. Block Diagram of CABAC Decoder

Besides, we go deeply into the realities for data dependency, and we observe all SEs and find out three characteristics decreased the performance in Figure 7. First, because SE has flexible code length, we can't know next bin in current SE or next SE clearly. Second, some SEs have several branches and judge depended on current SE. Third, context data is updated frequently, and it possibly require to spend some time waiting for updated data.

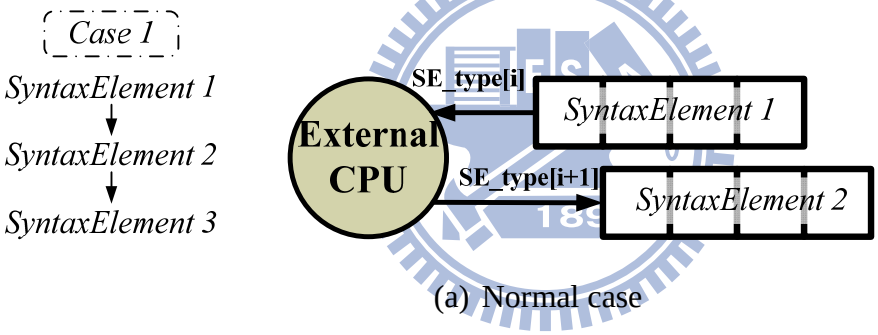


(a) (b) (c)

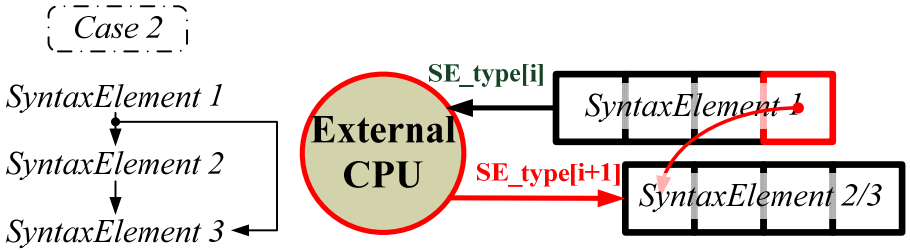
Figure 7. (a) Flexible code length (b) SE Branch Selection (c) Data update frequently

Therefore, the RAM-based context model scheduling for fetching and write-back becomes important issue in order to apply the pipeline architecture in CABAC decoder. The pipeline problems will be overcame in our proposed implementation.

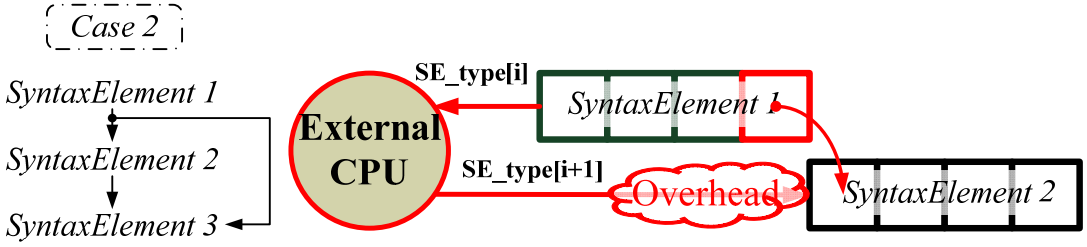
Except for drawbacks of CABAC decoder, it still exists a part which can cause performance lost. That is communication between SE parser and CABAC decoder. In [6], it calls for syntax element switching overhead (SESO). As Figure 8, we make an example for describing what situation can cause this overhead, and we use external CPU as SE parser. The case1 is the normal situation, and we already know what next SE is. The pipeline flow can be executed correctly. Actually, if we decode in the same SE continuously, it is always worked without unexpected stalls. But, as soon as we require to switch SE, it has a probability to cause data hazard. We can see in Figure 8(b). The case2 represents some SEs which have several branches, and we require previous result of SE to judge current SE branch selection. Hence, in the general solutions as Figure 8(c), we may stall some cycles to avoid data hazard. In simulation result from [6], it has more than two-thirds of probability for switching SE, and the performance would be degraded certainly.



(a) Normal case



(b) Data hazard case



(c) Solve by increasing overhead

Figure 8. Syntax Element Switching Overhead

Above analysis tells us the importance of throughput. And, we should know how many throughput is enough to real-time decode full-HD sequence. We consider a working frequency which can be accepted by system and show some different cases in Table 1. At first, we test several HD1080 sequence and find the worst case in gray part. The *riverbed* has 22.696 million bits/second (s), and throughput has to achieve 29.55 million bins/s. Actually, we require to produce 0.29 bin/cycle at 100 MHz. Second, we assume one frame contains 1 million bits, and encoder can get 1.5 compression rate (CR). And, we require to produce 0.45 bin per cycle. Finally, we consider the maximum bit-rate from standard, and we should produce 0.93 bin/cycle. It's mean we can real-time decoding full-HD by raising hardware utility even in the worst case.

Table 1 Design constraints for throughput from standard

Test Sequence		**MBits/s**	**CR**	**Mbins/s**	**bin/cycle**	
					@100 MHz	**@150 MHz**
riverbed		22.696	1.289	29.255	0.29255	0.195035
1 Mbits/frame		30.000	1.5	45.000	0.45	0.3
Level [1]	**4.1 @MP**	50.000	1.5	75.000	0.75	0.5
	4.1 @HP	62.500	1.5	93.750	0.9375	0.625

Moreover, the table-base CABAC reduce complexity significantly, but it also raises large table which have to include memory. Table 2 shows all kinds of table. However, because some tables contain large amount of data and switch frequently, they produce extra overhead to increase cost and decrease performance. In our analysis, (1) can be stored in external memory or ROM because it's seldom used and (4) can implement in internal buffer according as contained constant which occupy little gate count in hardware. Therefore, the bottleneck of memory becomes (2) and (3), and we will improve this problem in our memory system.

Table 2 Require for including table

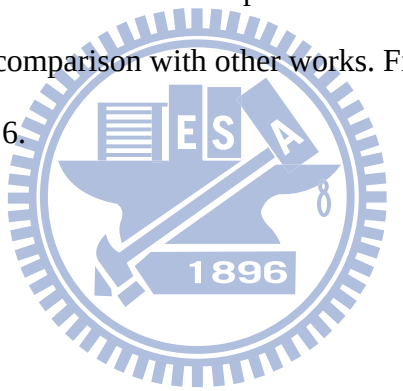
	(1) Initialization Table	**(2) Context Model**	**(3) Neighbour MB**	**(4) Range LPS & transMPS & transLPS**
Utility rate	Seldom	Very frequently	frequently	Very frequently
Max	65.92 kbits	7.21 kbits	727x(row) bits	2.048 kbits
Min	30.592 kbits	3.346 kbits		384 / 384 bits
Contain	Constant	Variable	Variable	Constant

1.3 Organization of Thesis

The rest of this thesis is organized as follows. In Chapter 2, we go through rapidly to review the specification of CABAC algorithm at first, and we describe the strategies for improvement of CABAC decoder and mention some state of the art to make an example. And, we evaluate their advantages and disadvantages.

In Chapter 3, we describe our proposed algorithm including a bin-trend-predictor scheme and optimization of memory system, and these methods get the balance with cost and performance. According to our proposed algorithm, we also take an in-depth discussion about the challenges of integrating architecture in Chapter 4.

And, we show our simulation results in Chapter 5 including verification of algorithm, implementation of architecture and comparison with other works. Finally, we make a brief conclusion and future works in the last Chapter 6.



Chapter 2. *Related Works*

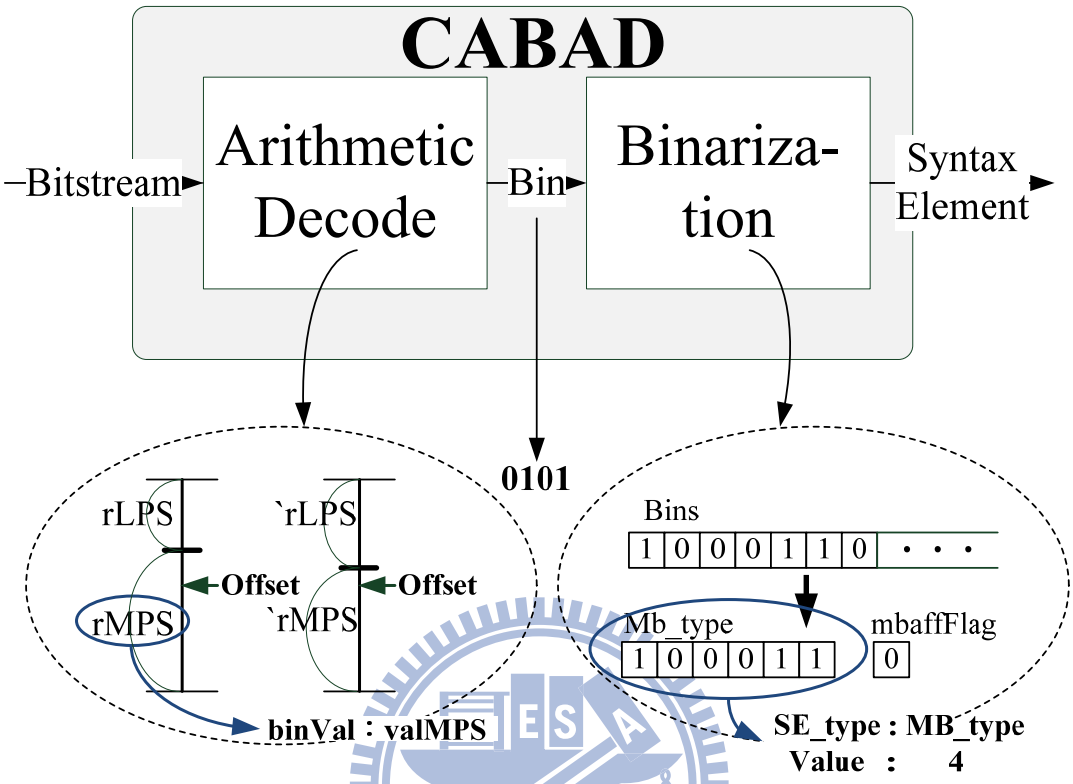


Figure 9. CABAC decoding algorithm

In this Chapter, we introduce the basic algorithm of the CABAC decoder in Section 2.1 at first. In the binary arithmetic decoder, it is executed by means of the recursive interval subdivision. It has to compute the values of *rMPS* and *rLPS* and processes the next value of *Offset*, *Range*, and the probability. After that, it decompresses the bit-stream to the *bin* value which offers the binarization to restore the syntax elements. According to H.264/AVC standard [1], we adopt the low complexity algorithm to implement the CABAC decoder circuit.

However, in order to support real-time high resolution videos, throughput still may be a bottleneck of H.264/AVC system. There are some strategies used to raise the throughput: parallel, pipeline and prediction. We introduce each of strategies and give an example, and we analyze the advantages and disadvantages in Section 2.2. Finally, we make a summary in Section 3.3 to describe the proposal in our design.

2.1 Traditional CABAC Decoding Flow

The traditional CABAC decoder engine is the sub-module of syntax element parser. When it is invoked, it schedules the timing related to the context model of reading-to and writing-back and selecting the arithmetic decoding flows and binarization flows. Figure 10 shows the finite state machine (FSM) of the traditional CABAC decoding flow [9]. The first state (state 0) is the stand-by state. The decoder waits for the request of the syntax element parser until activating the CABAC decoder system, and jumps to state 1. State 1 is required to check the type of AD. If it is the regular decoding, the binarization reads the neighbor information from the SRAM, and generates the context model index and reads the context model form the context model. And then, FSM jumps to state 2. State 2 is a binary tree where we have defined in Section 2.1.2. Based on the *bin* index (*binIdx*), the *bin* string is compared with the binary tree. If *bin* string can't find the mapped binary, the binarization engine increases *binIdx* and requests AD producing the next *bin* value to map again until the mapped binary and the suitable value of syntax element in state 3. If it finds the mapped binary value, the value of *binIdx* is initialized as "0" and waits for the request of the next syntax element.

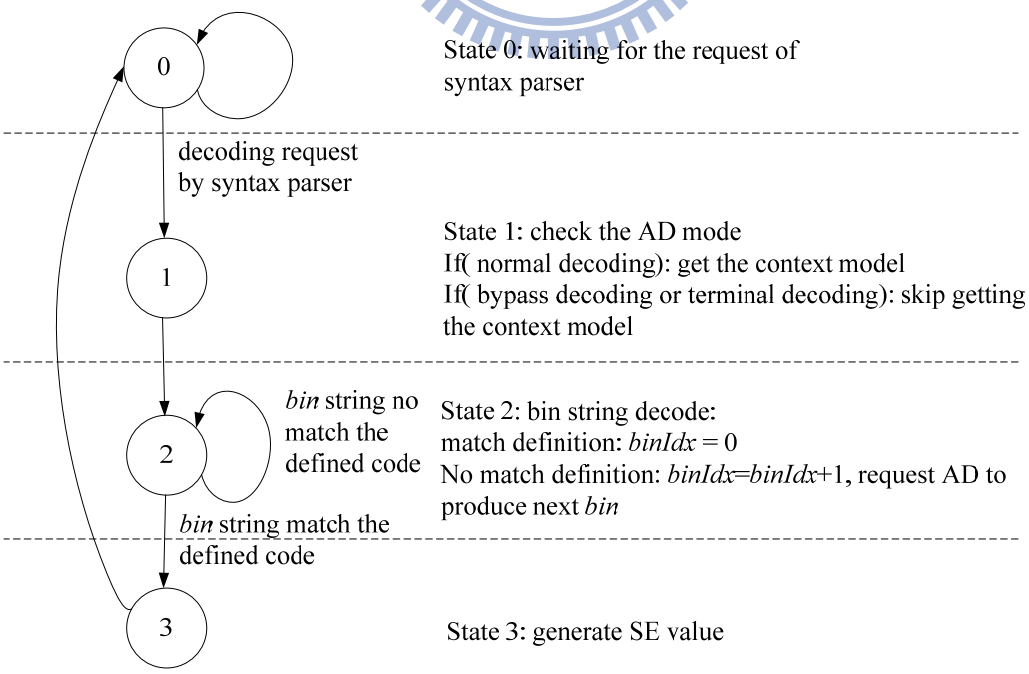


Figure 10. Traditional CABAC decoding flow [9]

2.1.1 Arithmetic Decoding Flow

In order to improve the coding efficiency, there are three kinds of the binary arithmetic decoders in H.264/AVC system such as the regular, bypass, and terminal decoding flow. We will show whole algorithms as follows.

2.1.1.1 Regular decoding process

The first algorithm is the regular decoding process which is shown in Figure 11(a). According to the H.264/AVC standard [1], the table-based method is used in place of the multiplication operation. In the regular decoding flowchart, *codlRangeLPS* looks up the table, rangeTabLPS, depending on two indexes such as *pStateIdx* and *qCodlRangeIdx*. The *pState* is defined as the probability of MPS (ρ_{MPS}) which gets from the context model. *qCodlRangeIdx* is the quantized value of the current range (*codlRange*) which is separated to four parts in this table. The second factor of the improved method is to estimate the value of ρ_{MPS} . The flowchart of Figure 11(a) also shows the table-based method to process the probability estimation. It divides into two sub-intervals such as MPS and LPS conditions. Depending on the sub-interval, it computes the next probability by the *transIdxLPS* table when the interval division is LPS and by the *transIdxMPS* table when the interval is MPS. These two probability tables are approximated by sixty-four quantized values indexed by the probability of the current interval.

In basis algorithm of binary arithmetic decoding, the interval subdivision is operated under the floating-point operation. In practical implementation, this method causes the complexity of the circuit to be increased. The advanced algorithm adopts the integer operation for H.264/AVC. The value of the next range becomes smaller than the current interval. So we use the renormalization method to keep the scales of *codlRange* and *codlOffset*. Figure 11(b) shows the flowchart of renormalization. The MSB of *codlRange* always keeps “1” in order to realize the integer operation. If the MSB of *codlRagne* is equal to “0”, the value of *codlRagne* has to be shifted left until the current bit is equal to “1”. Depending on the shifted number of *codlRagne, codlOffset* fill the bit-stream in the LSB.

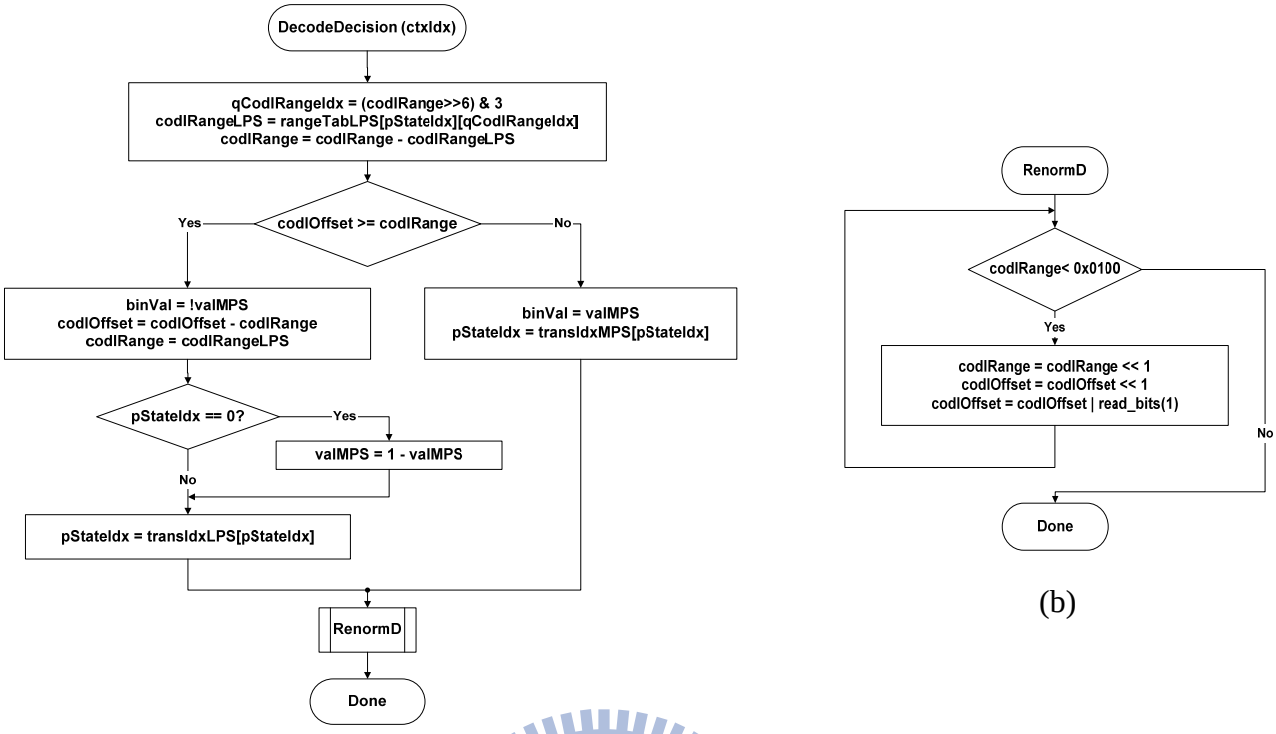


Figure 11. Flowchart of the (a) regular decoding and (b) renormalization process [1]

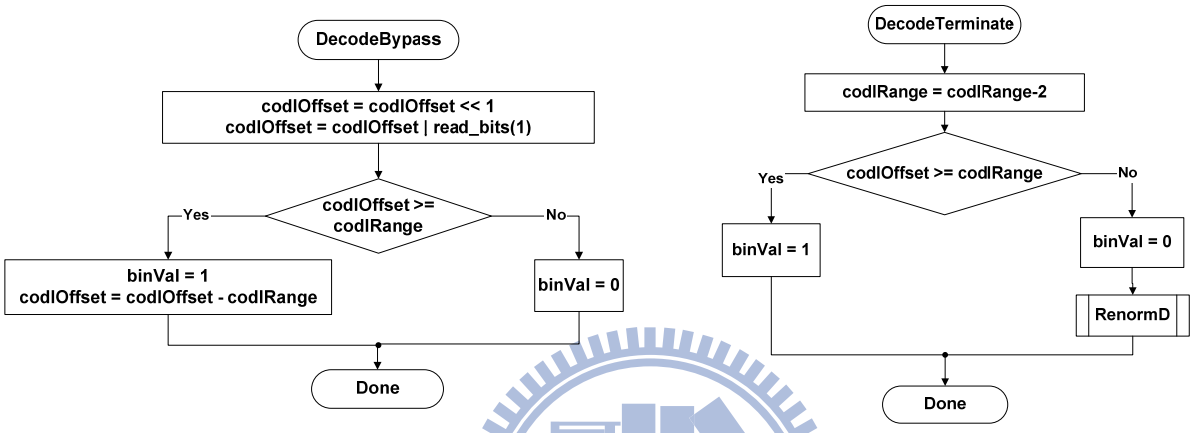
2.1.1.2 Bypass decoding process

The second algorithm is the bypass decoding process which is applied by the specified syntax elements such as suffix: *mvd, coeff_abs_level_minus, and coeff_sign_flag*. The probabilities of MPS and LPS are fair, that is, both probabilities are 0.5. It is unnecessary to refer to the context model during decoding. Figure 12 (a) shows the flowchart of the bypass decoding flow. Compared with Figure 11 (a), the bypass decoding process doesn't estimate the probability of the next interval. So we can't see the probability computation in the bypass decoding. The result of *codlRange* isn't changed which means that it has no the subdivision action in the bypass decoding. It is just used one subtraction to implement this decoding process.

2.1.1.3 Terminal decoding process

The third algorithm is the termination decoding process. Figure 12 (b) shows the flowchart of the terminal decoding flow. The terminal decoding process is very simple as well, but it has the more decoding procedure compared to the bypass decoding process. It doesn't need the context model to refer to the probability. The value of the next *codlRange* is always to subtract two from the current

codlRange depending on whether the subdivision condition belongs to MPS or not. The final values of *codlRange* and *codlOffset* are required to renormalize through the *RenormD* in this figure when it branches to the situation which defined *as codlOffset* smaller than *codlRange* (MPS condition). The flowchart results in composed of one constant subtraction, one comparator, and one renormalization. The terminal decoding process is used to trace if the current slice is ended. It occurs one time per macro block process which is seldom used during all decoding processes.



(a) (b)

Figure 12. Flowchart of (a) bypass and (b) terminal decoding process [1]

2.1.2 De-binarization Decoding Flow

In Section 2.1.2, we focus on the decoding process of the de-binarization. It reads the *bin* string to look up the suitable syntax elements. For H.264/AVC, CABAC decoder adopts five kinds of the binarization methods to decode all syntax elements. This section is organized as follows. In Section 2.1.2.1, the decoding flow of the unary code is shown at the first. The unary code is the basic coding method. Section 2.1.2.2 shows the truncated unary code which is the advanced unary coding method. It is applied in order to save the unary bit to express the current value. In Section 2.1.2.3, we introduce the fixed-length decoding flow. It is the typical binary integer method. Section 2.1.2.4 is the Exp-Golomb decoding flow. The Exp-Golomb decoding flow is only used for the residual data and the motion vector difference (*mvd*). Section 2.1.2.5 is the special definition by means of the table method. Specifically, we focus on the binary tree of the macroblock type (*mb_type*) and the sub-macroblock type (*sub_mb_type*).

2.1.2.1 Unary (U) binarization Process

Table 3 is the format of the typical unary code. If the syntax element is equal to “0”, the *bin* outputs single bit “0”. Besides the syntax element equal to “0”, the *bin* string sends “1” for *numSE* times firstly and one “0” in the end of the binary value. The value of *numSE* is equal to the syntax element. Therefore, we find the string length of the current syntax element *bin* string is *numSE+1*.

Table 3 Example for U

Syntax Element	***bin* string**			
0	0			
1	1	0		
2	1	1	0	
3	1	1	1	0
binIdx	**0**	**1**	**2**	**3**

2.1.2.2 Truncated Unary (TU) binarization Process

Table 4 is the format of the typical unary code. It is based on the unary code and has an additional factor of *cMax* which is defined as the maximum length of the current *bin* string. If the value of syntax element (*valSE)* is less than *cMax*, the TU and U are the same. Otherwise, the number “1” of the *bin* string is equal to *cMax* and there is no “0” bit to list in the current string.

Table 4 Example for TU with cMax = 3

Syntax Element	***bin* string**			
0	0			
1	1	0		
2	1	1	0	
3	1	1	1	
binIdx	**0**	**1**	**2**	**3**

2.1.2.3 Fixed-length (FL) Binarization Process

The fixed-length decoding flow has to refer to the value of *cMax* which defines the number size of the current syntax element. Table 5 shows the fixed-length code definition. In this table, the *cMax* equals seven because the maximum value of *binIdx* is seven.

Table 5 Example for FL with cMax = 7

Syntax Element	***bin* string**			
0	0	0	0	
1	1	0	0	
2	0	1	0	
3	1	1	0	
binIdx	**0**	**1**	**2**	**3**

2.1.2.4 k-th order Exp-Golomb (UEGk) binarization process

Table 6 shows the example of UEGk by means of the pseudo code from H.264/AVC [1]. The initial value of *k* is defined as the order of the unary Exp-Golomb coding which are named as UEGk. In the binarization decoding engine of CABAC decoder, it only applies two decoding flows such as UEG0 and UEG3. UEG0 is used by the suffix part of the residual data decoding process and UEG3 is used by the suffix part of motion vector difference one. And, the suffix part of this code doesn't always apply when the value too small.

Table 6 Example for UEGk with k = 0

Syntax Element	***bin* string**					
0	0					
1	0	1	0			
2	0	1	1			
3	0	0	1	0	0	
4	0	0	1	0	1	
5	0	0	1	1	1	
binIdx	**0**	**1**	**2**	**3**	**4**	**5**

2.1.2.5 Look up table (LUT) Binarization Process

All formats of the binarization decoding process are introduced above. But there is still a special decoding flow which we don't describe yet. In order to perform the higher video quality, the macroblock and sub-macroblock are divided into many kinds of types such as I, P, B, and SI slices. In the four basic types, it also sorts by variable block sizes. These two syntax elements are difficult to define by means of the aforementioned decoding flows. In H.264/AVC, it adopts the table-based method to define the macro and sub-macro block types. Table 7 shows an example of mb_type for P, SP slice, and gray part means suffix part are the same with table of mb_type for I slice.

Table 7 Example for mb_type (P, SP slice) [1]

Syntax Element	***bin* string**					
0 (P_L0_16x16)	0	0	0			
1 (P_L0_L0_16x8)	0	1	1			
2 (P_L0_L0_8x16)	0	1	0			
3 (P_8x8)	0	0	1			
4 (P_8x8ref0)	Na					
5(Intra, prefix only)	0	0	1	1	1	
binIdx	**0**	**1**	**2**	**3**	**4**	**5**

2.1.3 CtxIdx Model Index Calculating Flow

The values of the context model offer the probability value of MPS (*pStateIdx*) and the historical value of *bin* (*MPS*) in order to achieve the adaptive performance. In the regular decoding process of the arithmetic decoder, we have to prepare the 459 locations of the context model to record all decoding results in high profile.

$$ctxIdx = ctxIdxOffset + ctxIdxInc \quad \text{(Eq. 1)}$$

$$ctxIdx = ctxIdxOffset + ctxIdxBlockCatOffset + ctxIdxInc \quad \text{(Eq. 2)}$$

It divides into two kinds of the context model index (ctxIdx) methods to allocate the context model. (Eq. 1 is one of the index methods. Besides residual data decoding, the context model index is equal to the sum of *ctxIdxOffset* and *ctxIdxInc*. (Eq. 2 is the index method for residual data decoding. We should sum additional *ctxIdxBlockCatOffset* depend on the type of coefficient block.

2.1.3.1 ctxBlockCat and ctxIdxBlockOffset

The value of *ctxBlockCat* is the block categories for the different coefficient presentations. *maxNumCoeff* means the maximum required coefficient number of the current *ctxBlockCat*. *ctxBlockCat* is sorted six block categories in Table 8. And, the value of *ctxIdxBlockCatOffset* is defined as Table 9 which is dominated by the parameters of syntax elements and *ctxBlockCat*.

Table 8 Specification of ctxBlockCat for the different blocks [1]

coefficient type	maxNumCoeff	ctxBlockCat
luma DC	16	0
luma AC	15	1
Luma 4x4	16	2
chroma DC	4	3
chroma AC	15	4
Luma 8x8	64	5

Table 9 Assignment of ctxIdxBlockCatOffset to ctxBlockCat for SEs [1]

Syntax element of the residual data	ctxBlockCat					
	0	1	2	3	4	5
coded_block_flag	0	4	8	12	16	0
significant_coeff_flag	0	15	29	44	47	0
last_significant_coeff_flag	0	15	29	44	47	0
coeff_abs_level_minus1	0	10	20	30	39	0

2.1.3.2 Calculate for ctxIdxOffset

In both the residual data and the general decoding, the context model index is dominated by two factors such as *ctxIdxOffset and ctxIdxInc*. So, we merge *ctxIdxOffset* and *ctxBlockCatOffstet* and collect the results in Table 10. The alphabet of "na" denotes using bypass process. So, we only need to consider (Eq. 1. Depending on the syntax element, slice type, *ctxBlockCat* and some different conditions, we can find the value of *ctxIdxOffset.* And then, as soon as we calculate *ctxIdxInc*, we may compute the current *ctxIdx*.

Table 10 Syntax elements and associated types of ctxIdxOffset [1]

Image layer	Syntax element	ctxIdxOffset		
slice data	*mb_skip_flag*	(P slices only)		11
		(B slices only)		24
	mb_field_decoding_flag			70
	end_of_slice_flag			276
macroblock_ layer	*mb_type*	(I slices only)		3
		prefix	(P slices only)	14
		suffix		17
		prefix	(B slices only)	27
		suffix		32
	transform_size_8x8_flag			399
	coded_block_pattern	prefix	Luma	73
		suffix	Chroma	77
	mb_qp_delta			60
mb_pred	*prev_intraNxN_pred_mode_flag*	4x4, 8x8		68
	rem_intraNxN_pred_mode	4x4, 8x8		69
	intra_chroma_pred_mode			64
mb_pred and sub_mb_pred	*ref_idx_l0, ref_idx_l1*			54
	mvd_l0[ ][ ][ ], mvd_l1[ ][ ][ ]	prefix	x	40
		prefix	y	47
		suffix	(uses DecodeBypass)	na
sub_mb_pred	*sub_mb_type*	(P slices only)		21
		(B slices only)		36
residual block cabac	*coded_block_flag*	ALL	(ctxBlockCat < 5)	85
			(5 < ctxBlockCat < 9)	460
	significant_coeff_flag	frame	(ctxBlockCat < 5)	105
			(ctxBlockCat = = 5)	402
		field	(ctxBlockCat < 5)	277
			(ctxBlockCat = = 5)	436
	last_significant_coeff_flag	frame	(ctxBlockCat < 5)	166
			(ctxBlockCat = = 5)	417
		field	(ctxBlockCat < 5)	338
			(ctxBlockCat = = 5)	451
	coeff_abs_level_minus1	prefix	(ctxBlockCat < 5)	227
			(ctxBlockCat = = 5)	426
		suffix	(uses DecodeBypass)	na
	coeff_sign_flag	(uses DecodeBypass)		na

2.1.3.3 Calculate for ctxIdxInc

Basically, the value of *ctxIdxInc* is looked up in Table 11 by referring to the syntax element and *binIdx*. The alphabet of “na” denotes the never happened issue and the word of “Terminal” means that the decoding flow enters the terminal decoding process. If the generated *bin* is equal to “1”, the slice has to be stopped and decodes the next slice. However, we observe some SEs has several *ctxIdxInc* in Table 11. In these cases, we should refer to the left and top blocks to define the *ctxIdxInc* of the first *binIdx* such as *mb_type*, *mb_skip_flag*, *ref_idx*, *mb_qp_delta*, *intra_chroma_pred_mode*, *mb_field_decoding_flag*, and *coded_block_pattern*. According to different SEs, it may follow different kinds of principles from standard [1]. Besides, the value of *ctxIdxInc* in residual data is defined as the scanning position or look up table from the standard.

Table 11 Assignment of ctxIdxInc to binIdx for syntax elements [1]

Syntax elements	**binIdx**						
	0	**1**	**2**	**3**	**4**	**5**	**>= 6**
mb_type (I)	0,1,2	**Terminal**	3	4	5,6	6,7	7
mb_skip_flag (P)	0,1,2	na	na	na	na	na	na
mb_type (P:prefix)	0	1	2,3	na	na	na	na
mb_type (P:suffix)	0	**Terminal**	1	2	2,3	3	3
sub_mb_type (P)	0	1	2	na	na	na	na
mb_skip_flag (B)	0,1,2	na	na	na	na	na	na
mb_type (B:prefix)	0,1,2	3	4,5	5	5	5	5
mb_type (B:suffix)	0	**Terminal**	1	2	2,3	3	3
sub_mb_type (B)	0	1	2,3	3	3	3	na
mvd_lX (x:prefix)	0,1,2	3	4	5	6	6	6
mvd_lX (y:prefix)	0,1,2	3	4	5	6	6	6
ref_idx_lX	0,1,2,3	4	5	5	5	5	5
mb_qp_delta	0,1	2	3	3	3	3	3
intra_chroma_pred_mode	0,1,2	3	3	na	na	na	na
prev_intraNxN_pred_mode_flag,	0	na	na	na	na	na	na
rem_intraNxN_pred_mode,	0	0	0	na	na	na	na
mb_field_decoding_flag	0,1,2	na	na	na	na	na	na
coded_block_pattern (luma:prefix)	0,1,2,3	0,1,2,3	0,1,2,3	0,1,2,3	na	na	na
coded_block_pattern (chroma:suffix)	0,1,2,3	4,5,6,7	na	na	na	na	na
end_of_slice_flag	0	na	na	na	na	na	na
transform_size_8x8_flag	0,1,2	na	na	na	na	na	na

2.2 On-the-fly CABAC Decoding Flow

In this Section 2.2, we will introduce previous designs of CABAC decoder and use criteria of cost and performance to evaluate them. In fact, there are already a few papers which investigated the implementation of CABAC decoder such as [10] and [17]. In [17], they show that not only the arithmetic engine's (AE) peak performance but also its utilization is important for high throughput. In [10], they take an effort to evaluate several previous works and classify into three strategies: parallel, pipeline processing and prediction. In the common architecture, pipeline and parallel processing are the general solutions to enhance the throughout. However, the character of table-based CABAC algorithm is very difficult to implement parallel and pipeline structure efficiently. So, optimal methods and prediction scheme are proposed to improve this deflect. In Figure 13, we classify previous works as their strategy. Each strategy represents the major improvement in conventional CABAC decoder. Actually, we can't clearly recognize the strategies in some state-the-art solutions because some designs use multi-strategies to progress the performance. Moreover, we will discuss the strategies and analyze their benefits and drawbacks in following Section 2.2.1-2.2.3.

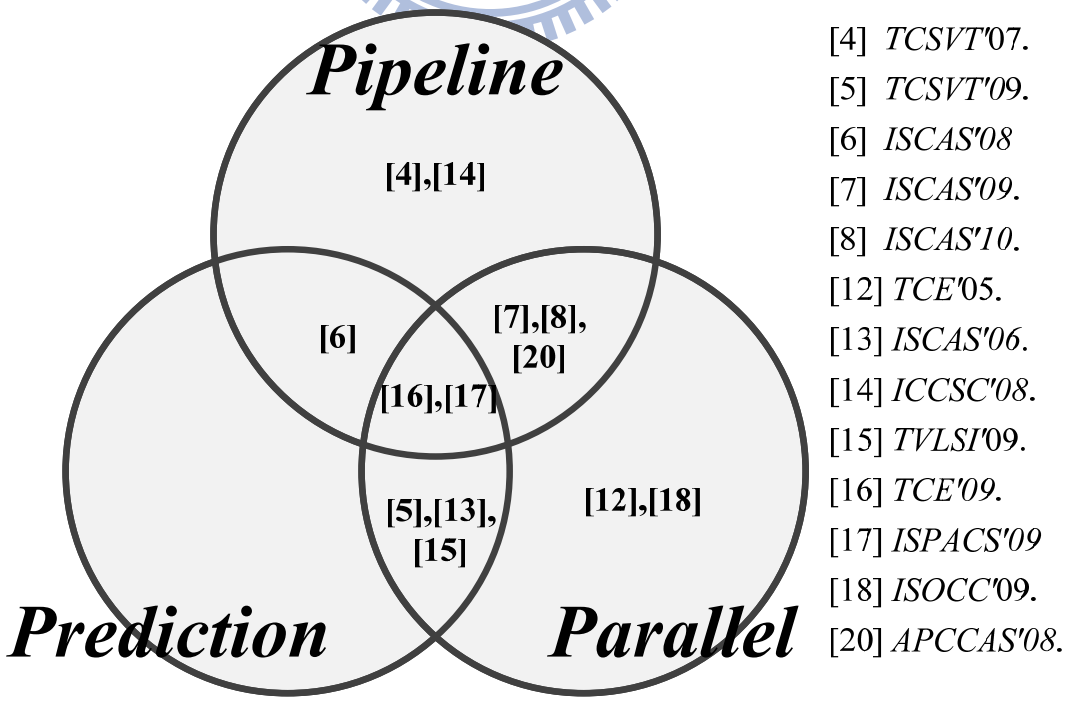


Figure 13. Implementation strategies of previous works

2.2.1 Pipeline-based CABAC Decoding flow

According to table-based CABAC algorithm, it requires 4 pipeline stages to support sequential memory accesses like Figure 14. As mentioned in [4], [14], the data hazards would be occurred according as ctxIdx relate to previous binIdx or bin, not updated context data and next SE type relate to current SE type. Those hazards would decrease performance in conventional pipeline structure.

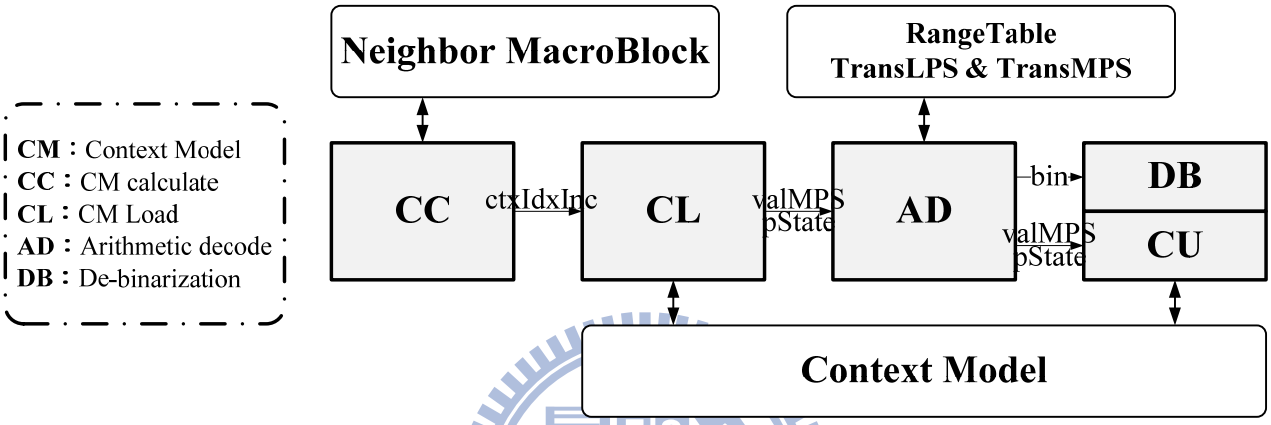


Figure 14. Pipeline Stages of conventional CABAC Decoder

To eliminate the stalls of pipeline, some previous works provide some improving methods. In [4], it parallels partial stages and provides a CM cache to reduce hazards, but it still have stalls by caches miss. And, [14] can ease data hazards efficiently by forward paths and duplication of partial CM, but it requires large SRAM. Furthermore, prediction scheme also can provide to eliminate the stalls, or multi-arithmetic decoding engine can promote throughput for pipeline structure. And, these issues will discuss in following section.

2.2.1.1 Analysis and discussion

Most of designs may use pipeline structure, because pipeline technology can shorten the critical path of design and raise working frequency. But, not all of designs make an effort to eliminate the stalls. If pipeline can work smoothly, it will reduce unnecessary overhead. However, raising pipeline structure utility may get a limit improvement and unavoidable stalls, and much higher frequency of memory accesses occupies large memory bandwidth requirement and SRAM. Therefore, merging other strategies is the best solutions for overcoming the bottleneck.

2.2.1.2 Example for pipeline structure

In Figure 15, we take an example [14] - “Pipelined Architecture Design of H.264/AVC CABAC Real-Time Decoding” to discuss the implementation of pipeline structure.

This works apply two situations and forward path to eliminate stalls. The forward path can be prepared for pre-fetching not updated CM data, and the other data hazard can be avoid by two situations. First one is used to choose binIdx++ and binIdx=0 by MUX2. Because the CABAC decoder may be required to produce one or several SE with flexible bin length, the bin of SE which will be complete can’t be known. To overcome these problems, it assumes two statuses. The SE type which impact *ctxIdxOffset* always can be known by SE parser, and each bin all consider as end of SE. It applies two context models. One contains full entries, and the other one contains partial entries which correlate binIdx=0. And, it prepares two kinds of context data all the time. Later, the correct context data can be chosen behind finishing AD process. The second situation is used to consider current ctxIdx relate to previous bin. Because *ctxIdx* always differ 1 in this situation, hazard can be eased by preparing both options. Though this adjustment, it can get high performance because of no pipeline stalls.

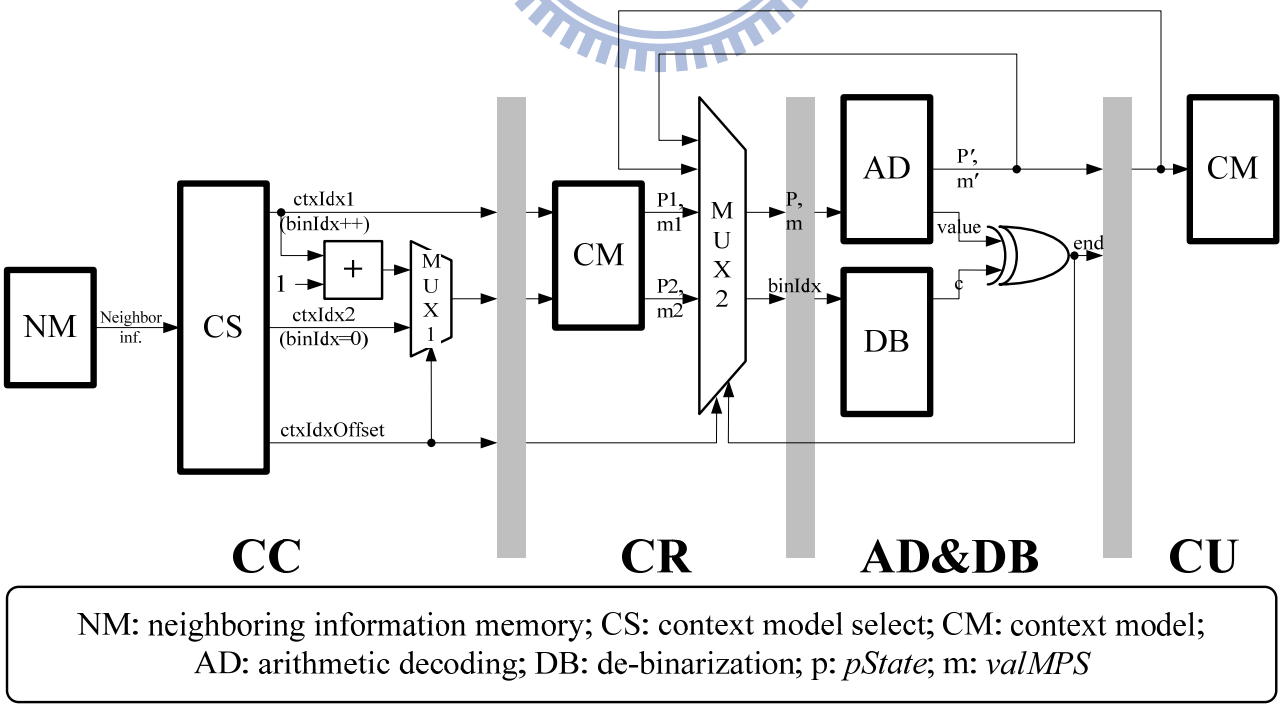


Figure 15. Pipeline architecture [14]

2.2.2 Parallel-based CABAC Decoding flow

Except for pipeline structure, parallel structure is the immediate solution to augment throughput. However, as shown in Figure 16(a), two regular bins decoding simultaneously will cause structure hazard by multi-access, because we require twice context model data in one cycle. Beside these problems, another problem will be caused by long critical path. According to correct *codIRange* and *codIOffset*, the AEs have to be cascaded. That may increase the challenge of parallel structure.

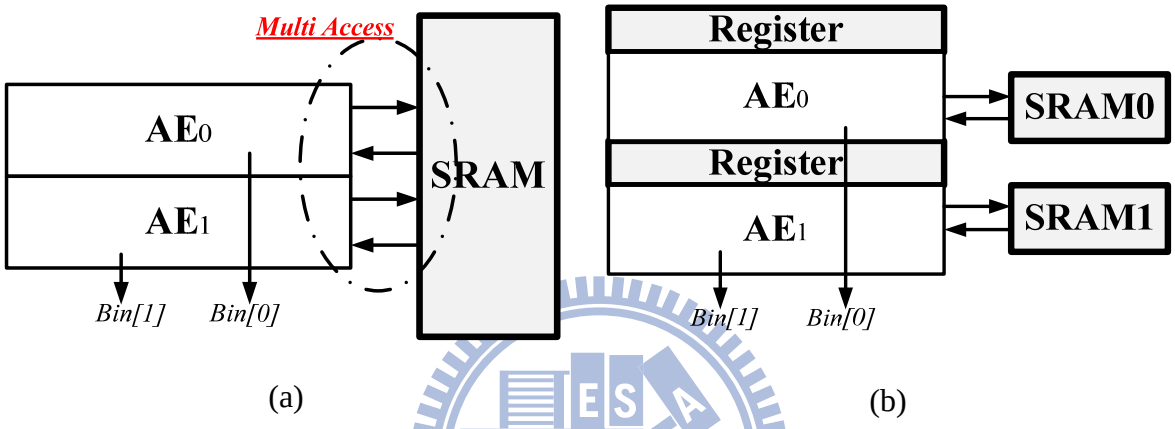


Figure 16. (a) Structure Hazard of Multi-bin (b) General Solutions of Multi-bin

[12] is the first paper proposed parallel-based CABAC decoder, and it supports several paths to decode one or two bin per cycle. [7] promotes high performance efficiently according to parallel decoding engine (TSBAD) and smooth pipeline flow, but it requires large hardware cost. [18] optimizes the critical path for parallel decoding by a symbol-prediction-bases scheme. According as above solutions, multi-accesses problem can be solved by including hybrid SRAM or CM cache like Figure 16(b), and long critical path also can be shorten by pre-fetch, pre-calculate and etc.

2.2.2.1 Analysis and discussion

According to state of the art parallel structure such as [7-8], we can get high improvement for throughput significantly. So, we can make sure parallel structure can bring high performance. But, in the other hand, it will be increased inevitable hardware cost by the internal buffer and parallel arithmetic engine. Even if we can accept the increase cost, the critical path will be another issue. Although we can find some methods to optimize the path, it is still longer than single arithmetic decoding engine. So, that will be a potential problem for raising working frequency.

2.2.2.2 Example for parallel structure

In Figure 17, we take an example [7] - “A Branch Selection Multi-symbol High Throughput CABAC Decoder Architecture for H.264/AVC” to discuss the implementation of parallel structure.

In this works, the major purpose is decoding two-symbol in one cycle efficiently. Since the CM memory is implemented in register in their proposed architecture, the CM loading or storing procedure can be merged in the same cycle [7]. By this reason, it can avoid structure hazard for multi-access, although it bring large cost. However, the most difficult part to supply multi-symbol is preparing the context data for second bin, because the 2nd ctxIdx may depend on 1st bin. Therefore, it uses pipeline structure and presents a branch selection scheme to prepare all possible options for guaranteeing the performance. Because the 1st bin is either 1 or 0, 2nd context data can be considered (1st bin=0) and (1st bin=1). Before decoding bin, TSBAD is inputted 3 context data while one for 1st bin and two for 2nd bin. And, two context data which prepare for 2nd bin will be selected correct one after decoded 1st bin. By this scheme, it makes a considerable improvement and provided an effective parallel-based CABAC decoder.

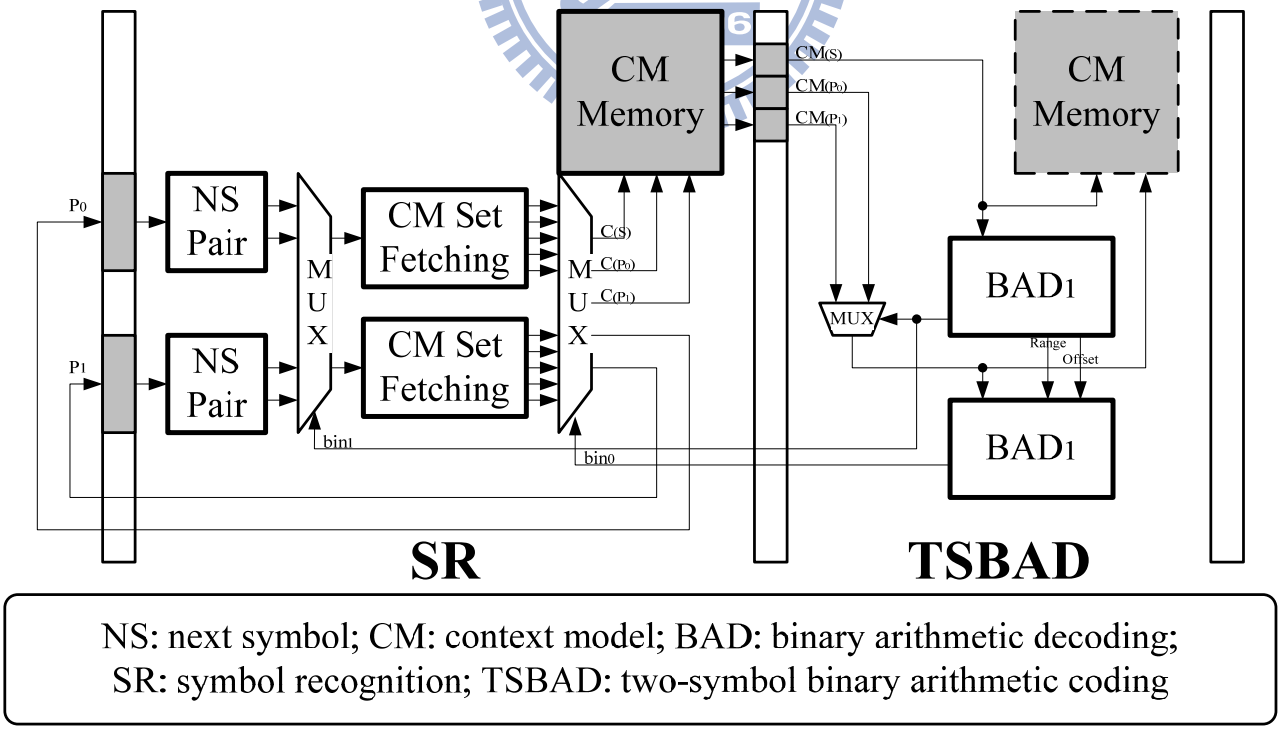


Figure 17. Parallel architecture [7]

2.2.3 Prediction-based CABAC Decoding flow

In Figure 13, we can't find the design which is only implemented by the prediction strategy, because the prediction scheme is usually an adjustment for pipeline or parallel strategy. Actually, the prediction scheme is proposed to speed up the parallel engine or hardware utility. Although the reasons for supporting prediction scheme are different for each previous works, they almost predict the value of current bin as MPS bin. Because MPS rate of total bin is more than 50% according to basic CABAC algorithm, this feature can be used to raise performance. In fact, we can classify prediction structures to two purposes: (1) using predicted value to decode multi-bin per cycle, (2) using predicted value to pre-calculate ctxIdx.

At first case, because MPS process is simpler than LPS process in BAD and has a high probability of occurrence rate. So, some previous works assume that the MPS bin can be decoded continually. According to this assumption, [5] and [13] proposed a hardware to achieve decoding 2 MPS bin in a cycle. And, [5] increases dual-series bypass paring for speeding up bypass bin. [15] exploits all the parallelism in a SE, it can decode 16 bins in a cycle mostly.

At second case, the predicted bin is used to avoid unexpected stalls. In order to avoid the impact of communication with parser and decoder, [6] proposed a SE predictor to determine next SE type by itself. The SE predictor can efficiently ease data dependency, but it wouldn't be available when it predicts miss. [16-17] apply the prediction scheme for pre-get-neighbor information, and they can avoid some idle times for loading neighbor information.

2.2.3.1 Analysis and discussion

According to previous works, prediction scheme plays an important role to balance the performance and hardware cost. The parallel structure will increase a few redundancy circuits to support multi-bin engine, and the pipeline structure can get much higher hardware utility efficiently. However, the accuracy becomes an important issue for prediction scheme. Although MPS bin has high probability occurrence rate from 50% to 70%, it may has variable performance by test pattern. Besides, even if it has 30% miss rate, the miss penalty will decay performance seriously.

2.2.3.2 Example for prediction structure

In Figure 18, we take an example [6] - “Prediction-based Real-time CABAC Decoder for High Definition H.264/AVC” to discuss the implementation of prediction structure.

This work belongs to second case as mentioned in Section 2.2.3. It discusses the overhead which is produced by communication between SE parser and CABAC decoder. To solve this problem, we have to obvious on the view of system. Because the order of SE isn’t regular and has high switching rate, the overhead can’t be avoided by improving CABAC decoder. Therefore, it applies a SE predictor which is controlled by value of bin and SE instead of traditional SE parser. The traditional parser can’t understand the meaning of each bin, so it has to wait for the value of SE which may cost several pipeline stages. Conversely, the SE predictor can be controlled by value of bin in each pipeline stages and can make sure for not only *ctxIdxInc* but also *ctxIdxOffset*. However, because of MPS-based two-bit predictor included in SE predictor, the accuracy will be impacted by MPS rate. Even throughput this deflects of SE predictor become the limit of throughput, it still brings a novel thought with the least overhead and most friendly integration solutions for H.264/AVC system.

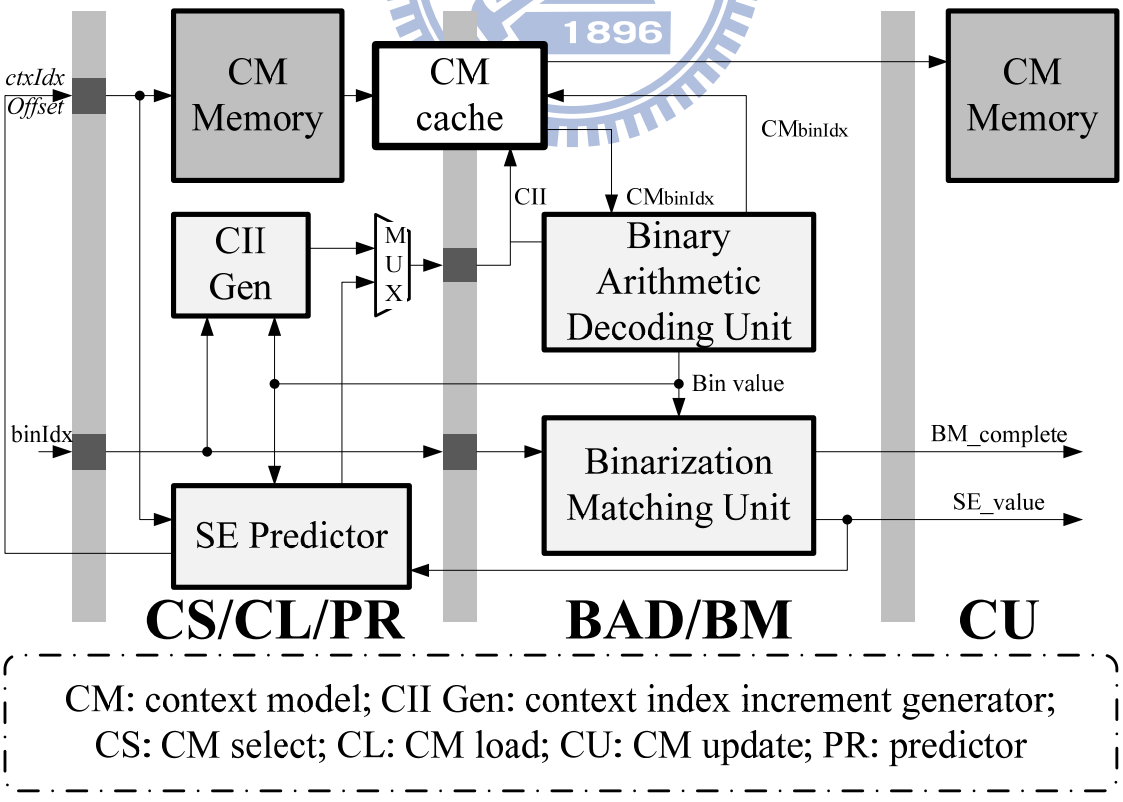
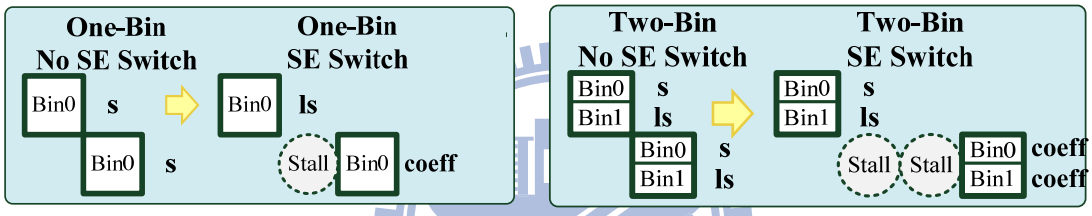


Figure 18. Prediction architecture [6]

2.3 Summary

Through above descriptions, we review the algorithm of CABAC decoder broadly and analyze the strategies for implementation. Actually, we already get high throughput by previous works such as [7] [8]. However, although we have some advanced improved strategies, it still has several potential problems while integrating to system. For example, [14] will be unavailable by unknown *ctxIdxOffset* and has to occupy large memory bandwidth requirement or internal SRAM size. And, the syntax element switching overhead (SESO) may be enhanced by multi-bin engine as shown in Figure 19. The larger CM cache adopts, the more miss penalty will be paid. And, the deeper pipeline stages is, the more idle times increase.



(a) (b)

Figure 19. (a) Single-bin engine (b)multi-bin engine

By the state of the art, we shouldn't keep going to raise much higher throughput. For our purpose, we make an effort to design CABAC decoder which is the most suitable strategy for system integration and take a balance between throughput and overhead at the premise of the acceptable throughput for real-time decoding full-HD sequences. However, systems problem will be more complex than single module behavior. It becomes serious problems which include not only communicating with SE parser as mention in previous section but also immediately getting neighboring information and so on. In fact, the real sequences can be organized by several kinds of SE, and the distribution of bin, MPS rate and SE type can be totally different for each sequence. If the design just optimize at the special statuses, the real performance will be hard to guarantee. Therefore, because of our constraints, we consider the strategy of prediction structure by doing minimum adjustment to get maximum reward.

According as [6], applying a controllable SE parser can integrate different kinds of situations which we can't improve before, and all possibly data hazard issues can be transferred to accuracy of predicted bin. Therefore, we simplify the problems and make a formula for throughput as following:

Assume

$Cycle_R$, $Cycle_B$, $Cycle_T$: *requirement of executed cycles for regular, bypass and terminal process*

Bin_R, Bin_B, Bin_T: *the amount of regular, bypass and terminal bin)*

Stall times: *the amount of stalls;* ***Idle times:*** *the amount of interruptions*

Regular Bin rate :*the ratio of regular bin in total bin*

Miss Rate: *the ratio of prediction miss*

$$\text{Total Cycle} = \text{Cycle}_R + \text{Cycle}_B + \text{Cycle}_T \quad \text{(Eq. 3)}$$

$\because \underline{(\text{Cycle}_R = \text{Bin}_R + \text{Stall times} + \text{Idle times}, \text{Cycle}_B = \text{Bin}_B, \text{Cycle}_T = \text{Bin}_T)}$

$$\therefore \quad = \text{Bin}_R + \text{Stall times} + \text{Idle times} + \text{Bin}_B + \text{Bin}_T \quad \text{(Eq. 4)}$$

$\because \underline{(\text{Total Bin} = \text{Bin}_R + \text{Bin}_B + \text{Bin}_T)}$

$$\therefore \quad = \text{Total Bin} + \text{Stall times} + \text{Idle times} \quad \text{(Eq. 5)}$$

First, we can get the formula of total executed cycles by Eq. 5.

$$\text{Bin per cycle (BPC)} = \frac{\text{Total Bin}}{\text{Total Cycle}} \quad \text{(Eq. 6)}$$

$$\underline{\text{(Eq.5 substitution)}} \quad = \frac{\text{Total Bin}}{\text{Total Bin} + \text{Stall times} + \text{Idle times}} \quad \text{(Eq. 7)}$$

$\because \underline{(\text{Stall times} = \text{Total Bin} \times \text{Regular Bin Rate} \times \text{Miss Rate})}$

$$\therefore \quad = \frac{\text{Total Bin}}{\text{Total Bin} + \text{Total Bin} \times \text{Regular Bin Rate} \times \text{Miss Rate} + \text{Idle times}}$$

$$= \frac{1}{1 + 1 \times \text{Regular Bin Rate} \times \text{Miss Rate} + \frac{\text{Idle times}}{\text{Total Bin}}} \quad \text{(Eq. 8)}$$

Second, we can get the formula of BPC by Eq. 7.

$$\text{Throughput rate} = \text{Working Frequency} \times \text{BPC} \quad \text{(Eq. 9)}$$

$$\text{(Eq.8 substitution)} \quad = \frac{\text{Working Frequency}}{1 + \text{Regular Bin Rate} \times \text{Miss Rate} + \frac{\text{Idle times}}{\text{Total Bin}}} \quad \text{(Eq. 10)}$$

Finally, we get the formula of throughout rate by Eq. 10.

In (10), we assume the conventional CABAC decoder with prediction scheme and controllable SE parser can decode one bin per cycle while no miss penalty, and we shows the formula for throughput rate with working frequency, regular bin rate, miss rate, idle times and total bin. The working frequency depends on critical path of design, or it could be restricted to rise by system constraints such as power consumption. Besides, regular bin rate and amount of total bin are depended on test sequence, so they will be unreliable parameters for various video sequences.

And then, the idle times are often caused by fetching neighboring information. If we apply an effective environment to deal with neighboring information accesses, idle times can be ignored while (total bin >> idle times). In [11], it has mentioned an improvement for time and storage efficiency by taking full use of the new found characters of SEs. Even though we use previous methods, it still has an improvement potential specifically for upgrading to full-HD. So, this will be an important issue to progress in our design.

Except for idle times, the major parameter which impacts the efficiency is miss rate. Because all the reasons for decayed performance are imputed to accuracy of prediction scheme, the miss rate will be the bottleneck of throughput. Although MPS bin has more than 50% hit rate and achieve almost 70% in test sequence in average, the miss rate still become too high to get acceptable throughput rate. Therefore, in order to guarantee the throughput and maintain basic hardware cost, providing a high accuracy prediction scheme will be an essential mission we should do in our design.

Chapter 3. *Proposed Algorithm*

In this chapter, we propose our algorithm to raise hit rate for throughput and reduce the storage for extra overhead. Because the bottleneck of throughput depends on the hit rate in the prediction-based CABAC decoder, we make the decoding flow more regular except the case we should know the decoded bin. First, we apply a SE parser unit without uncertain part and store each stage status. So, the uncertain part would be controlled by predicted bin_0. And then, we raise a bin predictor to produce bin_0 during decoding bin_1 by AD unit. Finally, we can make AD and CL working at the same time. In [6], it proposed a MPS-based two-bit predictor included in SE predictor to predict bin for the same problem, and it obtained about 70% hit rate. The 30% miss rate would be the critical part decreased the performance. Therefore, we propose three methods to improve the hit rate and describe in section 3.1.

In the other hand, we often produce extra overhead for getting neighbor information and requesting data from external memory. The overhead may be hardware cost, delay cycles or memory bandwidth requirement. To avoid unexpected performance lost, we optimize our memory system to solve this problem. We propose a CtxIdxInc pre-Calculate (CC) stage behind second pipeline stages (DB) to pre-calculate and compress the neighbor information like Figure 20. We pre-fetch and decompress when we request. And we describe in section 3.2.

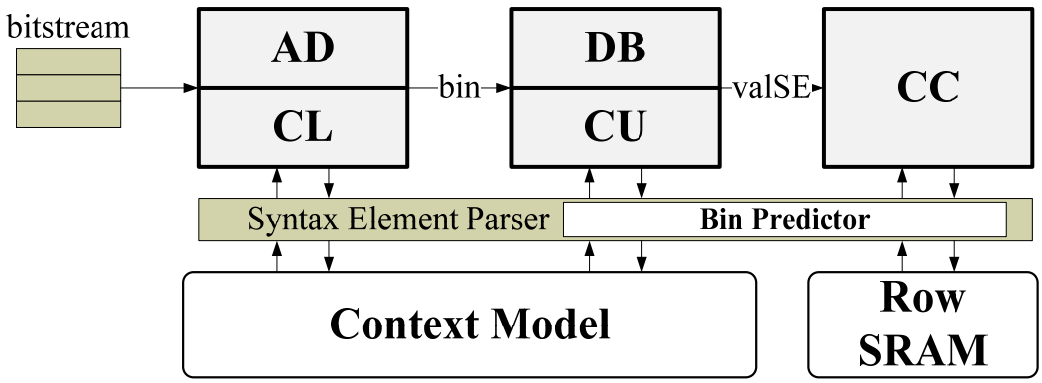


Figure 20. Pipeline Stages of Proposed CABAC Decoder

3.1 Prediction Process

Because we shouldn't need all of next bin when decoding current bin, we collect all status we have to predict next bin shown in Figure 21. First, we can categorize total bin to three kinds of bin, terminal bin, bypass bin and regular bin. However, we request ctxIdx to access Context Model (CM) only if next bin is regular bin. It may not cause pipeline stalls when next bin is bypass or terminal bin. So, we will focus on the problem in regular bin.

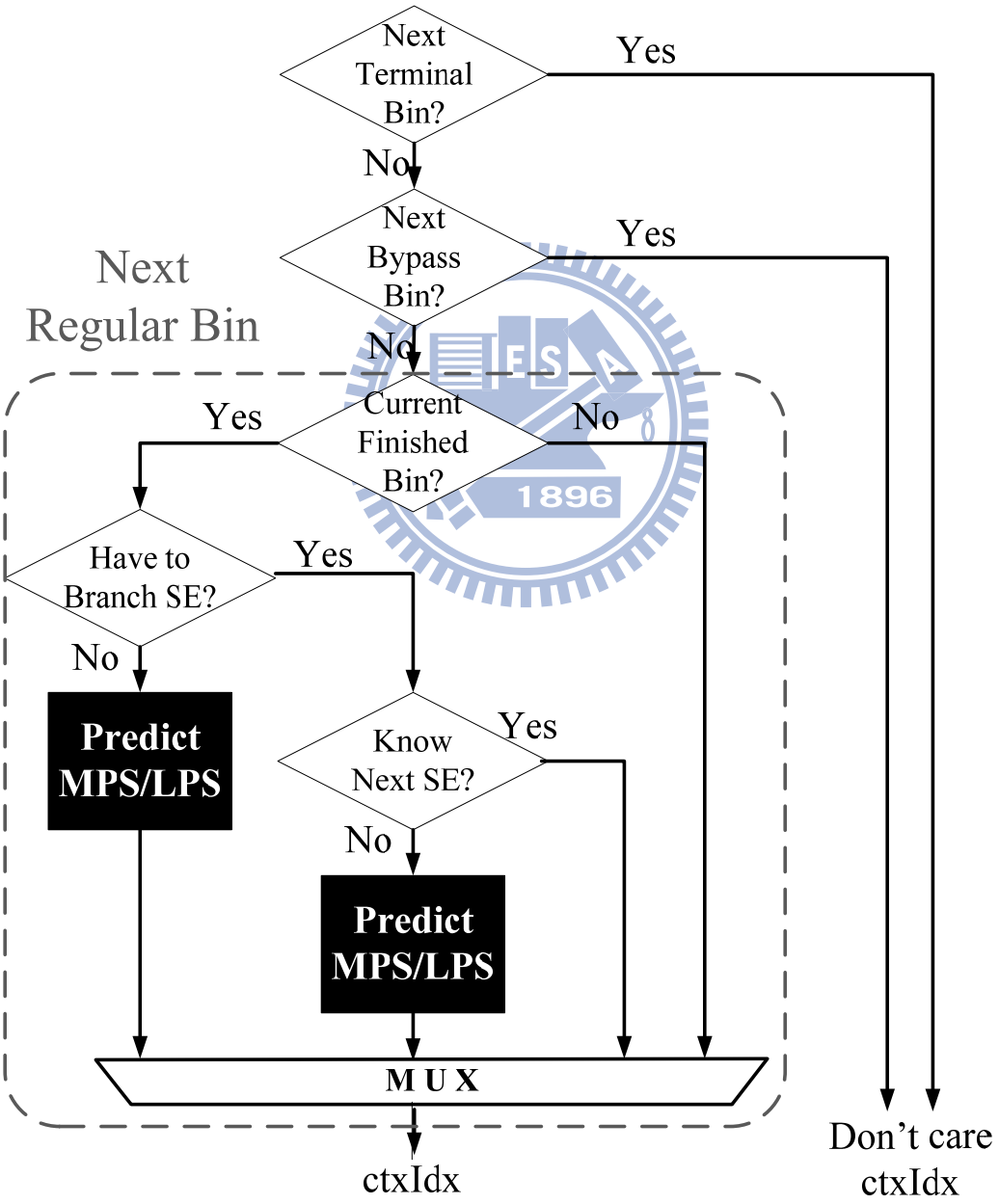


Figure 21. Flowchart of Bin Predict Process

After that, we still need to classify regular bin by some kinds of different situations. Because ctxIdx data of the same syntax element (SE) are located in neighbor, we should make sure if it may branch to next SE or not. So, we recognize the current bin which is finished bin according as SE type and binIdx. The finished bin means possible last bin in the syntax element. If the current bin isn't finished bin, we can be easy to calculate next ctxIdx by previous ctxIdx or bin. In the other hand, some SE have flexible length, and we can't really sure when the next bin will be branched to next SE. Therefore, we may require predicted bin when next bin isn't sure to SE branch in flexible-length type SE. Besides, even if the next bin is sure to branch to next SE, we still have to know what next SE is. This problem is often occurred by flag type SE. Flag type SE means this kind of SE has only one bin. In this situation, we also require predicted bin to calculate SE branch we may select. Summary, we point out two cases we require predicted bin. In the other words, if we get the high hit rate by the prediction process, we almost can calculate the ctxIdx without pipeline stalls for data dependency. Therefore, in following section we raise some methods to improve the throughput. The Section 3.1.1 describes how to raise hit rate. Even if we predict miss, we still may not have miss penalty and this method describes in the Section 3.1.2. Finally, the Section 3.1.3 describes some optimization to suit for our proposed prediction process.

3.1.1 Raised Hit Rate

In the Figure 22(a) and Figure 22(b), they shows the traditional arithmetic decoding flow from standard [1]. In the beginning, we have current value of Range and Offset. After we read the value of LPS range (rLPS), we can know the offset is inside the field of MPS or LPS. Then, we can decode bin value which depends on MPS or LPS, and the critical time would be waiting for rLPS from table. However, we can recognize the trend that is more possible for MPS or LPS in front of we read rLPS. Before we get the rLPS, we already have current Range and Offset. If we observe the difference between Range and Offset, we can find out MPS rate will be higher when difference is larger. Following this principle, we use two bits according as Figure 23 to recognize which result is mostly

happened at each status. Figure 23 shows the example for one of the rLPS table. At this example, we use pState equal to 0, 31 and 62 to classify, and the corresponding rLPS may be 128, 29 and 6. The difference may be transferred to 2 bit status shown in Figure 23. Actually, we can make sure result at status “00” and ‘11”, because the value exceeds the limit of standard. These statuses can significantly raise hit rate. Second, we also can use this method to observe current pState like Figure 24. Figure 24 does the opposite behavior with current pState. At the same example, we calculate rLPS in average from 0 to 31, 0 to 63 and 31 to 62, and the corresponding pState may be 13, 23 and 43. So, current pState also may be transferred to two bits. After that, we get the extra two bits that can raise hit rate when we are at status “01” or “10”. Summary, we can make sure the result at status “00” and “11” and have two-bit tips to predict result at status “01” and “10”. Finally, we get over 90% hit rate in our simulation results shown in Chapter 5 by the proposed method.

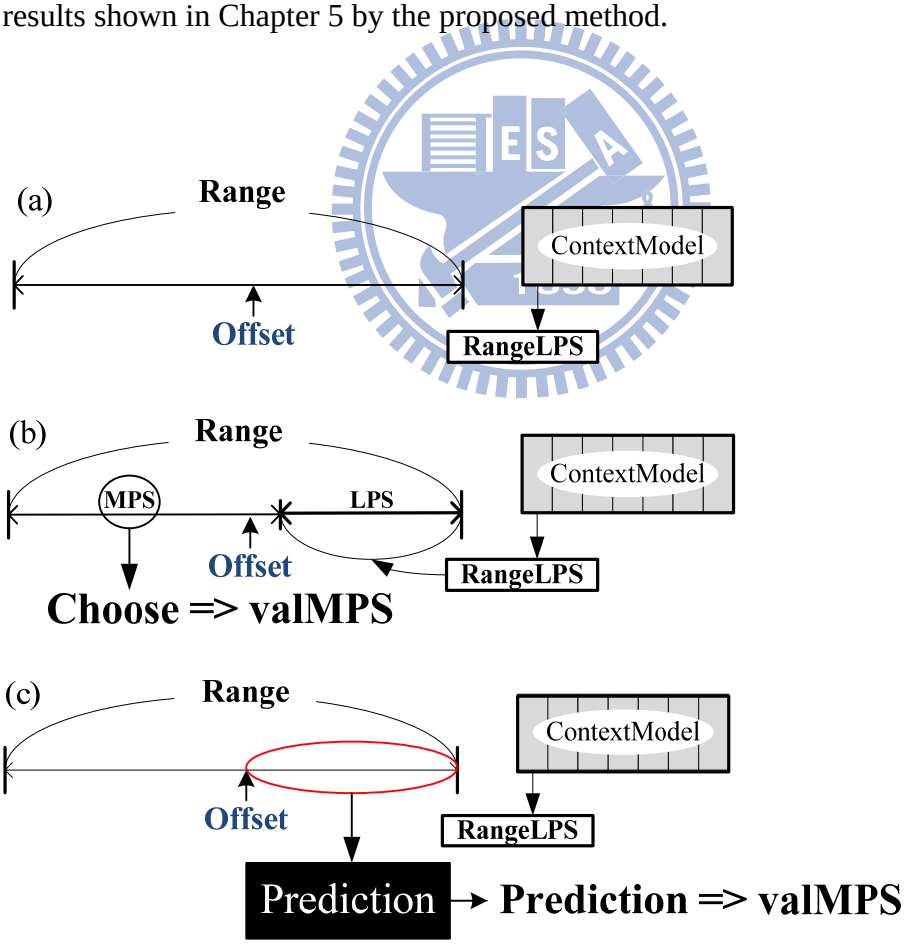


Figure 22. (a) Before decoded bin (regular process)
(b) After decoded bin (regular process) (c) Before decoded bin (prediction process)

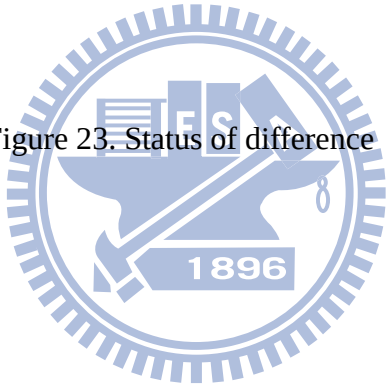


Figure 23. Status of difference

Figure 24. Status of pState

3.1.2 Reduced Stall Times

Furthermore, we try to reduce unnecessary stall times to raise throughput. Because of regular decoding flow, our prediction process depends on predicted bin_0 to calculate ctxIdx, we may get miss penalty when predicting miss. In our analysis, not all of syntax element (SE) branch need previous bin and have described what situation we request previous bin at the beginning. Therefore, we collect all type of SE to find out their finished bin. The finished-bin location is determined by binarization type, and Figure 25 is shown each kinds of binarization.

As described in Chapter 2, we recognize each kinds of binarization. And then, we may explain and use an example to point out where the finished-bin location is in the following paragraph.

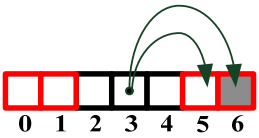
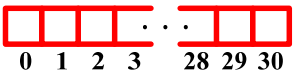
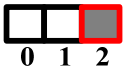
	Binarization type	Example	Finished-bin location
(a)	**Special Case (LUT)**	mb_type (I slice)	
(b)	**Unary (U)**	ref_idx	
(c)	**Truncated Unary (TU)**	Intra_chroma_mode	
(d)	**k-th order Exp-Golomb (UEGk)**	*suffix:*mvd_l0	
(e)	**Fixed-length (FL)**	rem_intraNxN_pred_mode	

Figure 25. Different kinds of finished-bin Location

In Figure 25(a), this binarization type is usually occurred in mb_type, sub_mb_type. Because these SE only can be decoded by table from standard, we may have to discuss each binIdx to recognize the finished-bin location. For example, mb_type in I slice, the finished-bin location may be at binIdx equal to 0, 1, 5 or 6, and SE must branch when binIdx equal to 6. Besides, we can recognize finished-bin location after decoding 3rd bin of SE.

In Figure 25(b) and (c), this binarization have similar characteristic. According to the rule of binarization, each of bins can be finished-bin location. However, TU have more advantage in additional condition. SE must branch when max value equal to cMax. For example, ref_idx, the finished-bin location may be at each of binIdx. Intra_chroma_mode may be finished at 1st bin or 2nd bin, because cMax equal to three.

In Figure 25(d), UEGk may be occurred in suffix of mvd and coeff_abs_level_minus1. We may change to bypass mode when we use UEGk binarization. Actually, we almost don't need finished-bin location except decoding last bin using UEGk. And, we can know where location is by previous bin. For example, suffix mvd_l0; we may get finished-bin location at 14, 16, 18, 20 or 22 from binIdx equal to 14, 15, 16, 17 or 18.

In Figure 25(e), this is most immediately binarization. We can find out the finished-bin location just by cMax, and this kind of location is sure to branch. For example, rem_intraNxN_pred_mode, the finished-bin location may be binIdx equal to two and SE have to branch at 3rd bin.

Summary, we collect all relationship between value of SE (valSE) and SE branch shown in Figure 26, each of SE with rough border may have SE dependency between current valSE and next SE, and the other SE branch can pares next SE type without current valSE.

Finally, we collect all finished-bin location of each binarization type and selected SE branch to know when we really should stall or not. Even if we predict miss, we wouldn't have miss penalty possibly. And then, we merge above method and high accuracy prediction to decrease effect of SE dependency.

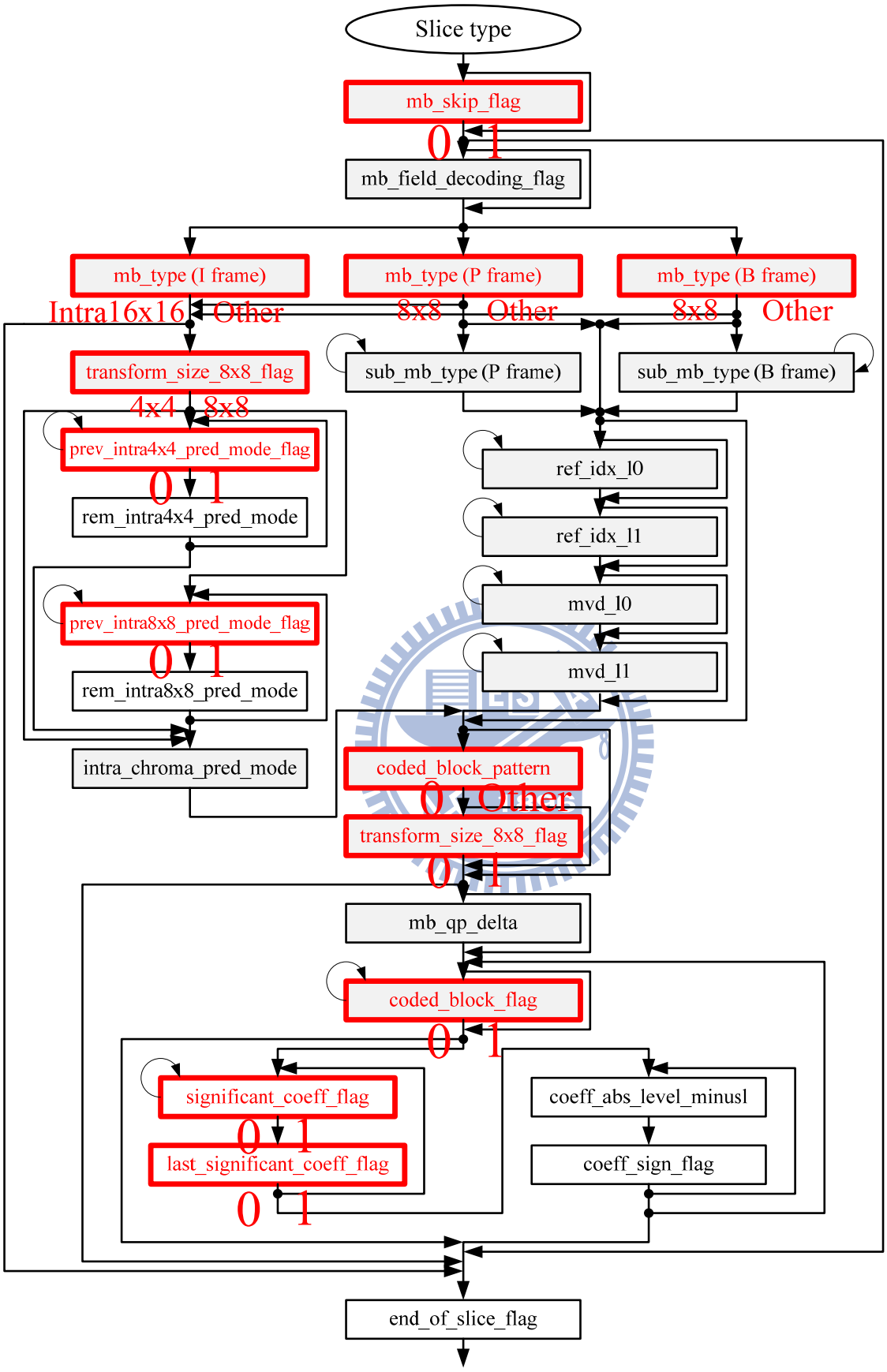


Figure 26. Relationship between value of SE and SE branch

3.1.3 Solved Data Hazard Problem

In the other hand, pipeline architecture may cause unavoidable data hazard. When we calculate next ctxIdx, we may need SE from neighbor block. With the improvement from traditional decoding flow, we avoid the worst case which we must stall to wait for decoded bin. Therefore, the other problems which cause data hazard are (1) un-decoded neighbor SE inside other pipeline stages and (2) un-updated pState in memory. Forwarding path can solve the first problem and data reuse (used buffer to hold on data) can solve the second problem. Therefore, most of data hazard can be solved by our proposed algorithm.

3.1.3.1 Forward path for (1) un-decoded neighbor SE

In our proposed pipeline architecture, we have three stages and four statuses. Pr (N) produces ctxIdx (address for CM), AD (0) produces bin, DB (1) produces valSE and CC (2) produces condTermflag. And, we need one more cycle for register. Therefore, we spend at least four cycles to handle each of neighbor information for neighbor block. However, this flow may be unavailable when Pr(N) require condTermflag during condTermflag haven't be finished. For example, coded_block_pattern, this SE immediately require neighbor information when decoding bin shown in Figure 27(a). So, we provide several forward paths to overcome this problem. We can choose data from AD, DB, CC or regular path and select by a multiplexer to cancel data hazard shown in Figure 27(b).

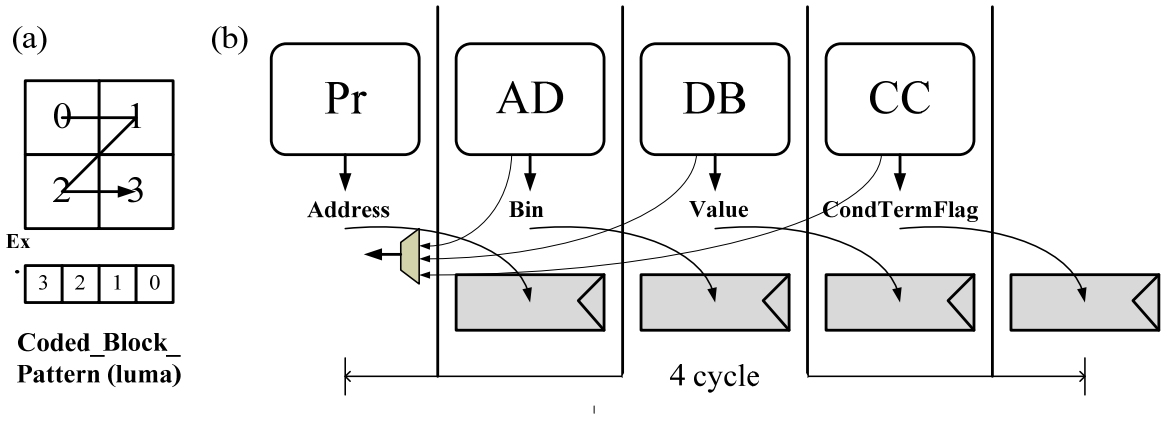


Figure 27. (a)Coded block pattern (b)Forwarding path for un-decoded neighbor SE

3.1.3.2 Data reuse for (2) un-updated pState

In the regular path like Figure 28(a), we may load context data by current ctxIdx from context model (CM) and update new context data to CM at next pipeline stage frequently. Context data include pState and valMPS. However, this path may be unavailable when load the same context data repeatedly. Because we should wait for updating new context data and loading new context data again, it may produce unnecessary latency by regular path. For that reason, we provide another path to deal with this kind of trouble.

In Figure 27(b), we show the data reuse path when we load in the same context data. Actually, we use an internal buffer to hold on updated context data. We obvious ctxIdx continually and store updated context data to buffer after decoding bin. If we find out ctxIdx is the same with previous ctxIdx, we may choose the data reuse path and use context data from internal buffer certainly until loading in different ctxIdx. According to this method, we raise a little overhead to implement, but we can reduce unnecessary stalls waiting for context data updated efficiently.

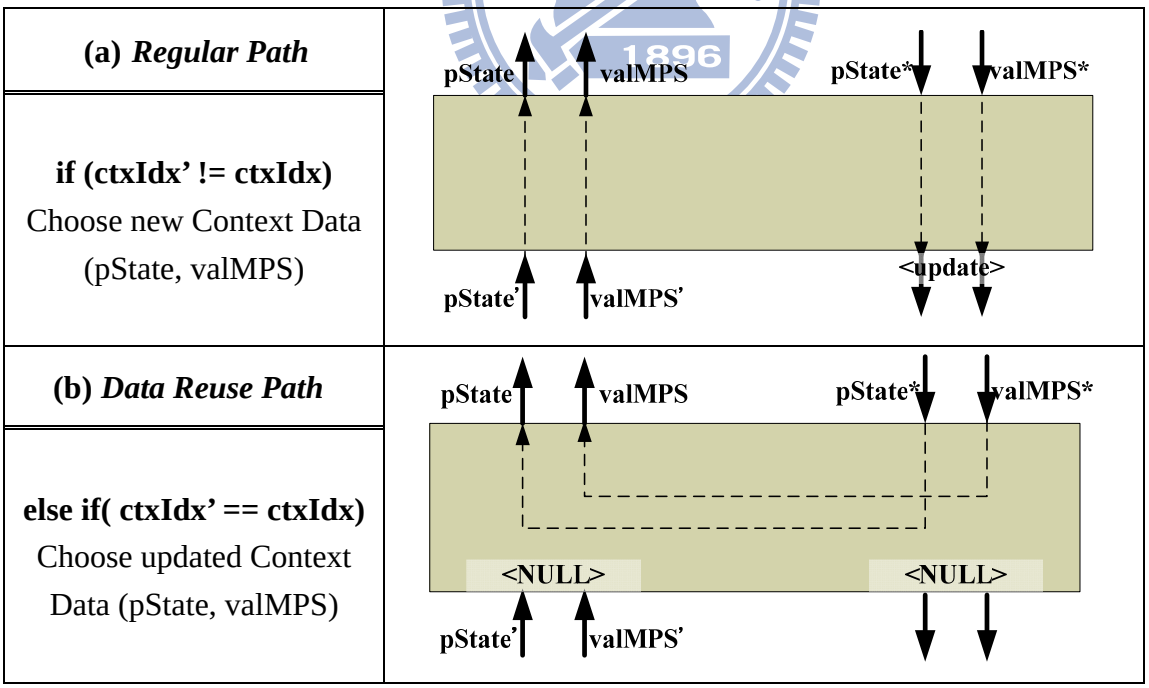


Figure 28. (a) Regular path (b) data reuse path

3.2 Memory System

In section 1.2 and Table 2, we have mentioned the bottleneck of memory system roughly. The major parts are context model and neighbor information storage. Because of our proposed method, we use single-bin engine avoiding multi-access problem. And, high hit rate enhance the hardware utilization significantly and ease data dependency efficiently. So, we don't have necessary to deal with increased overhead to supply parallel-base decoder. Therefore, we implement a context model by a two-port SRAM with 3,360 bits, and we can load and update context date simultaneously.

However, in the other issue, getting neighbor information may extend the latency when we access data from external memory. In Figure 29(a), this is a scheme to exchange neighbor information from external memory. But, when we request data from system bus, the latency may exceed our timing budget obviously. The reasons could be system clock are asynchronous with external memory, other modules occupy memory bandwidth especially for Motion Compensation and so on. Therefore, this may be a potential problem for real-time decoding, even if original storage aren't huge than the other modules.

The immediate solution is to include an internal memory to store all information we need like Figure 29(b). In Figure 29(b), we provide a scheme and store a row of MB to exchange neighbor information from internal memory. But, this method would get large overhead when we decode HD sequences. We should include almost 20 Kbits SRAM even double in MBAFF mode for CABAC decoder. So, this is in-efficient method to implement. To overcome these unexpected problems, the best solution is to reduce the stored neighbor information.

Therefore, we pre-calculate SE for neighbor macroblock (MB) to reduce neighbor information and describe in Section 3.2.1. And then, we provide a concentrated buffer to reduce redundant hardware cost and describe in Section 3.2.2. Finally, we alternately pre-fetch all neighbor information of MB to avoid waiting for neighbor information when decoding first bin of SE and describe in Section 3.2.3.

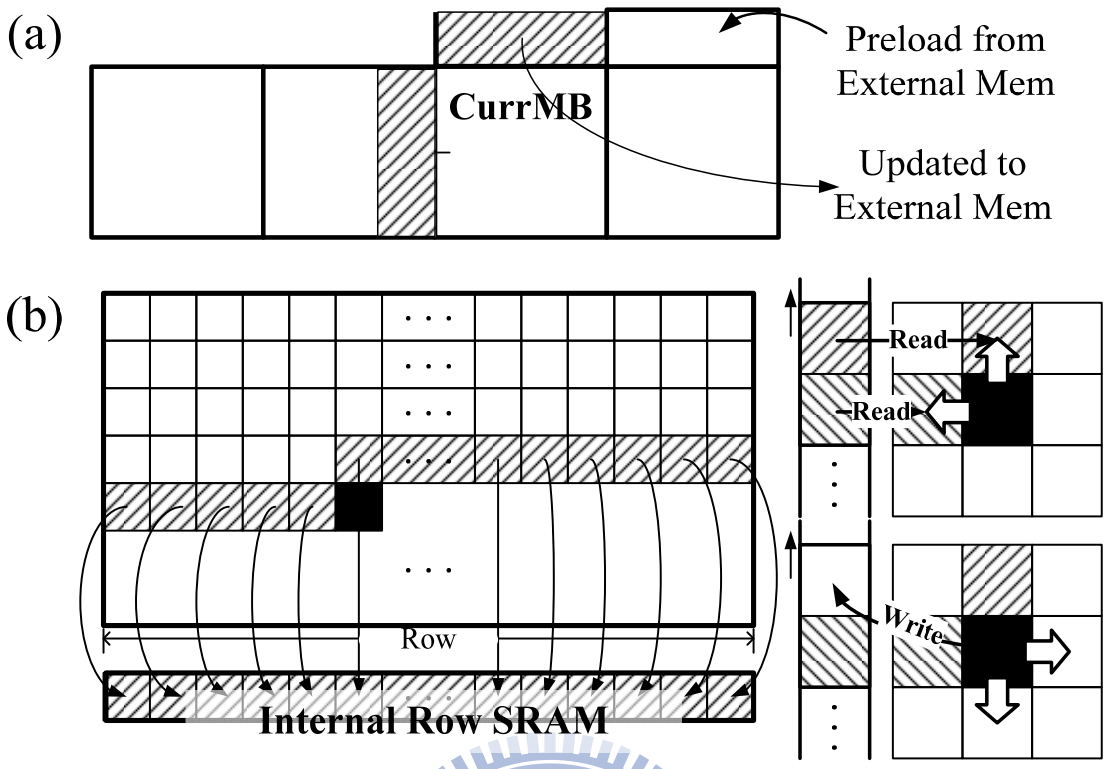


Figure 29. Stored in (a) external memory (b) internal memory

3.2.1 Reduced Memory Bandwidth Occupation

To reduce storage, the most efficient way is pre-calculated SE for neighbor block. In our analysis, most of SE can be pre-calculated for neighbor MB after decoded value of SE (valSE) except motion vector different (*mvd*). However, mvd is the biggest part of all neighbor information. We need 10 bits to store one mvd, and each MB has 16 mvd in the worst case. When we calculate current ctxIdx for mvd, we need neighbor *mvd* (A) and *mvd* (B) from left and top block. Then, we sum them to determine *ctxIdxInc* as Figure 30(a). However, we may not have to use total 10 bits to calculate ctxIdxInc in most of cases. In our simulation results, we find out most of mvd can be represented by two bits. So, we can reduce each mvd from 10 bits to 2 bits and use 5 bits to store extra mvd which is bigger than 3, as Figure 30 (b). When we need to refer neighbor mvd, we access each 2-bits-mvd and several extra-mvd from memory. Finally, we can efficiently reduce about 70% storage in our simulation result shown in Chapter 5 by this method.

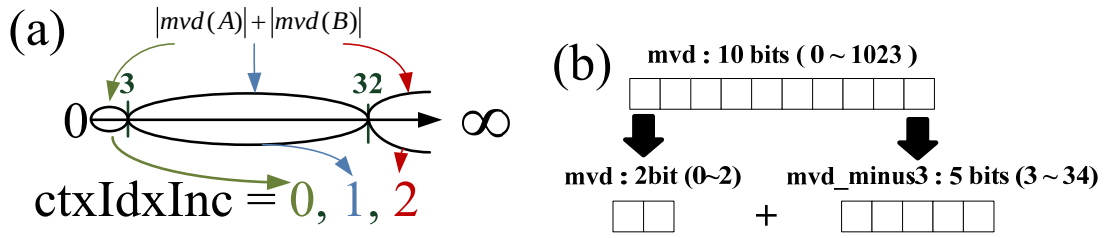


Figure 30. (a) ctxIdxInc control condition for mvd
(b) proposed mvd reduction scheme

Besides, the other SE can be pre-calculated, and we can use one bit to represent neighbor SE. Because the total neighbor storage has been reduced considerably, we can pre-fetch all neighbor information from neighbor MB we need before decoding the current MB.

3.2.2 Raised Buffer Efficiency

Furthermore, when we decode first bin of SE, we should access the same SE at the neighbor block that may be located in current MB or neighbor MB such as Figure 31. Figure 31 shows three different kinds of reference direction. Immediately, we need two kinds of buffer. The one store the data from neighbor MB, and the other one store the data from current MB. Therefore, we pre-fetch all storage of MB in the previous work, and both of them may occupy the largest percentage of buffers. During above reason, we may combine these two kinds of buffers to raise buffer efficiency.

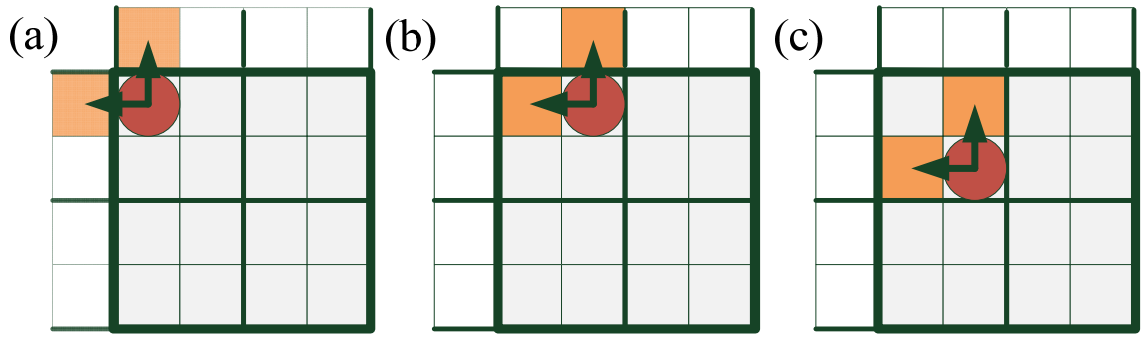


Figure 31. (a) Both in mbAddr (b) left in CurrMbAddr, top in mbAddr
(c) both in CurrMbAddr

After we read the neighbor information from the memory, the data won't be always available in various block size. When we decode except for 16x16 block size, the neighbor block may be changed from neighbor MB to current MB. So, we can reuse the buffer which stored neighbor MB information when neighbor blocks are in the current MB.

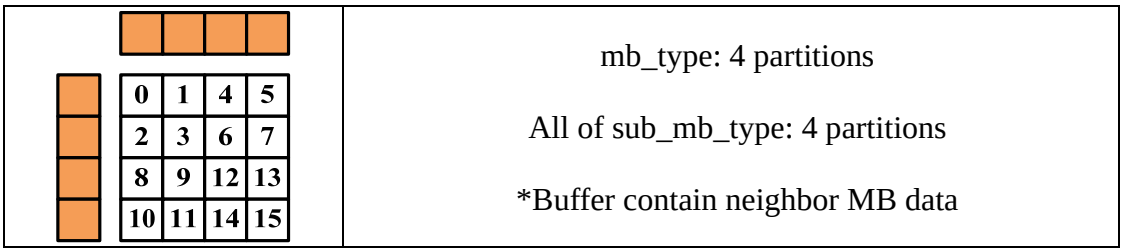


Figure 32. At the beginning of top and left buffer

In Figure 32, we show an example for a MB with 16 blocks at the beginning. We only require left and top buffer which contained 4 blocks size. In Figure 34, we show our schedule for reading and writing data in concentrated buffer. At first, current block is 0, and we have to read neighbor information from left and top buffer. After reading data, the blocks which are read can be clear to update new information. And then, we can write pre-calculate information to the empty buffer. However, we may meet terrible by following this principle. Because we change to next block by zigzag scan, the data may be covered too early. To reduce this drawback, we adjust the schedule to update the data in two buffers. Through our adjustment, the final results are shown in Figure 33. The data contained in left buffer can be reused for right MB, and the data contained in top buffer may update to SRAM for bottom MB.

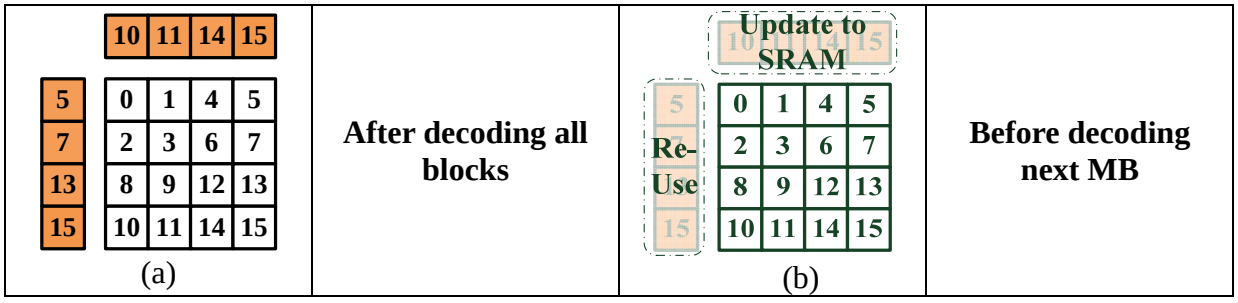
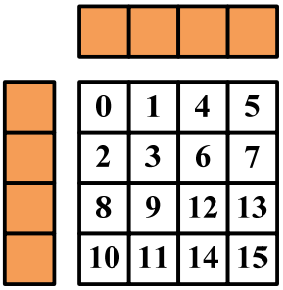


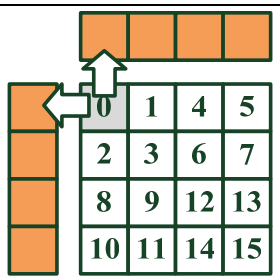
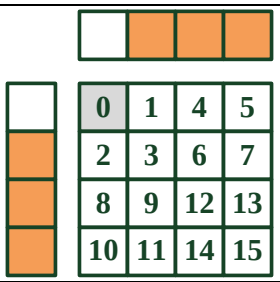
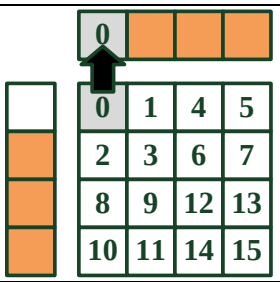
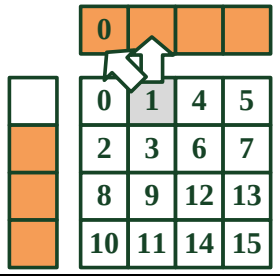
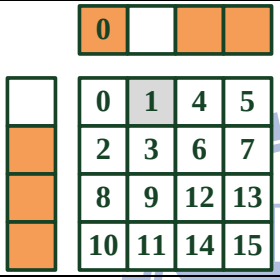
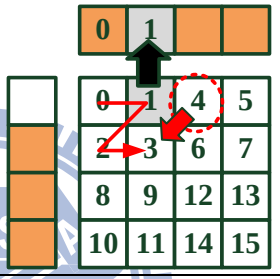
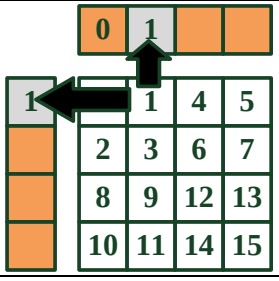
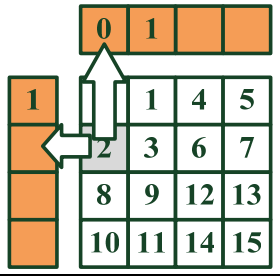
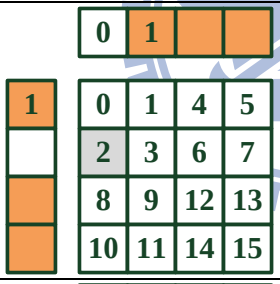
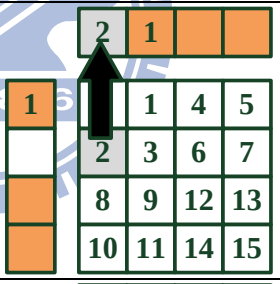
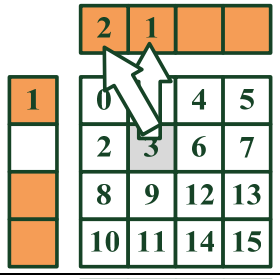
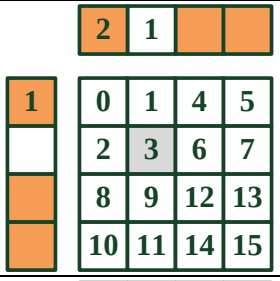
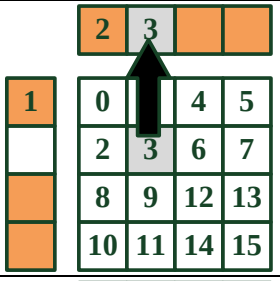
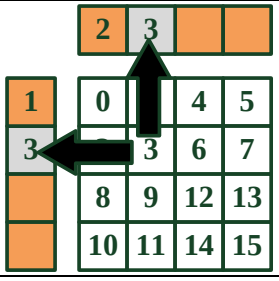
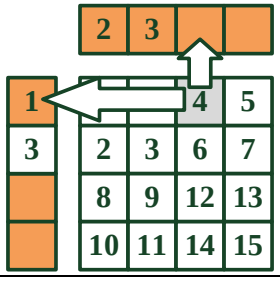
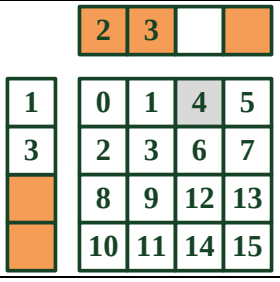
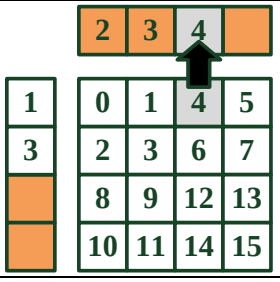
Figure 33. (a) In the end of top and left buffer (b) After decoding MB process

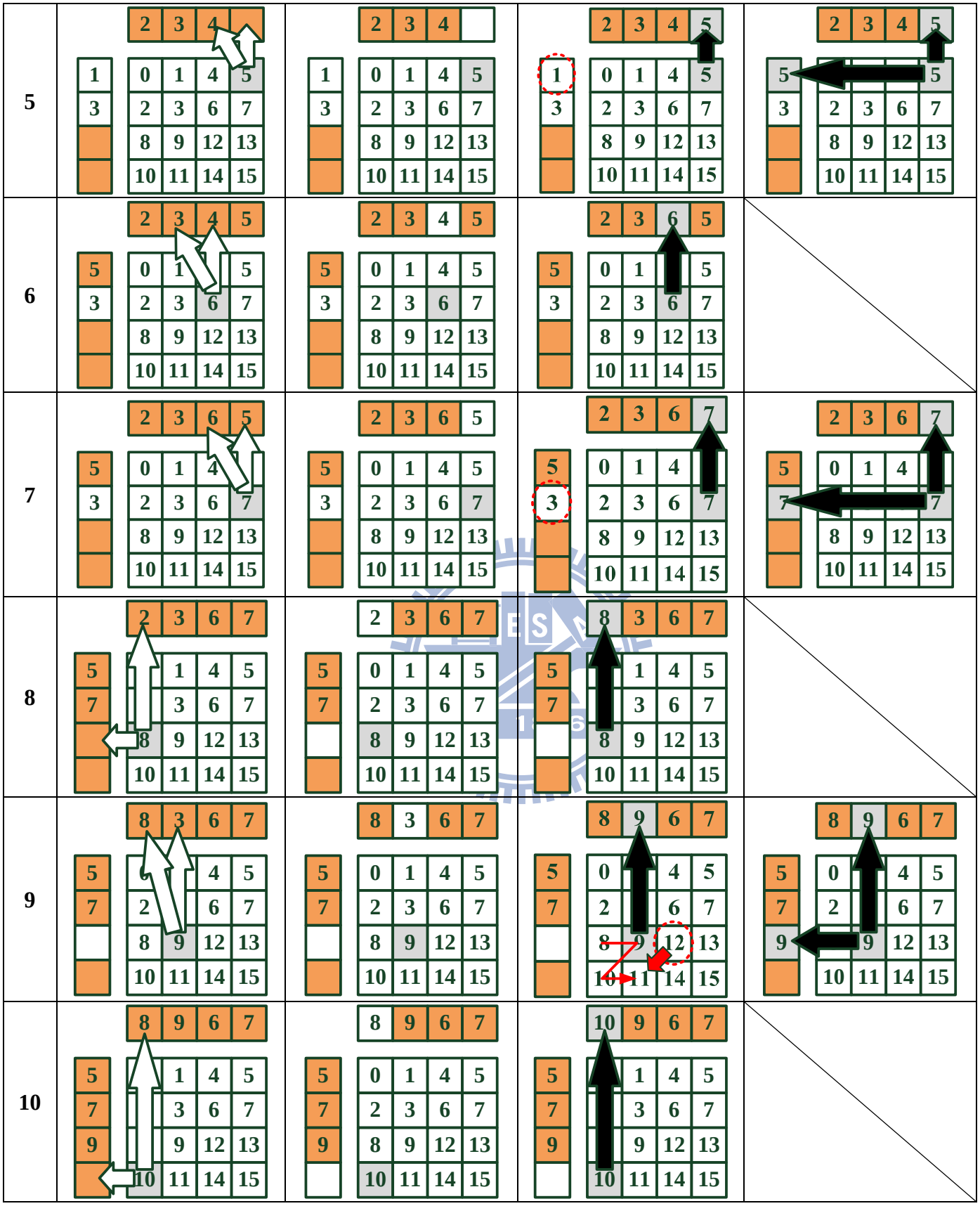
Because of this scheme and accurately scheduling, we can combine these two kinds of buffers to reduce hardware cost.

(Following discussion don't include MBAFF mode*)**



At the beginning …

	READ	CLEAR	WRITE	ADJUEST
0				
1				
2				
3				
4				


5
6
7
8
9
10

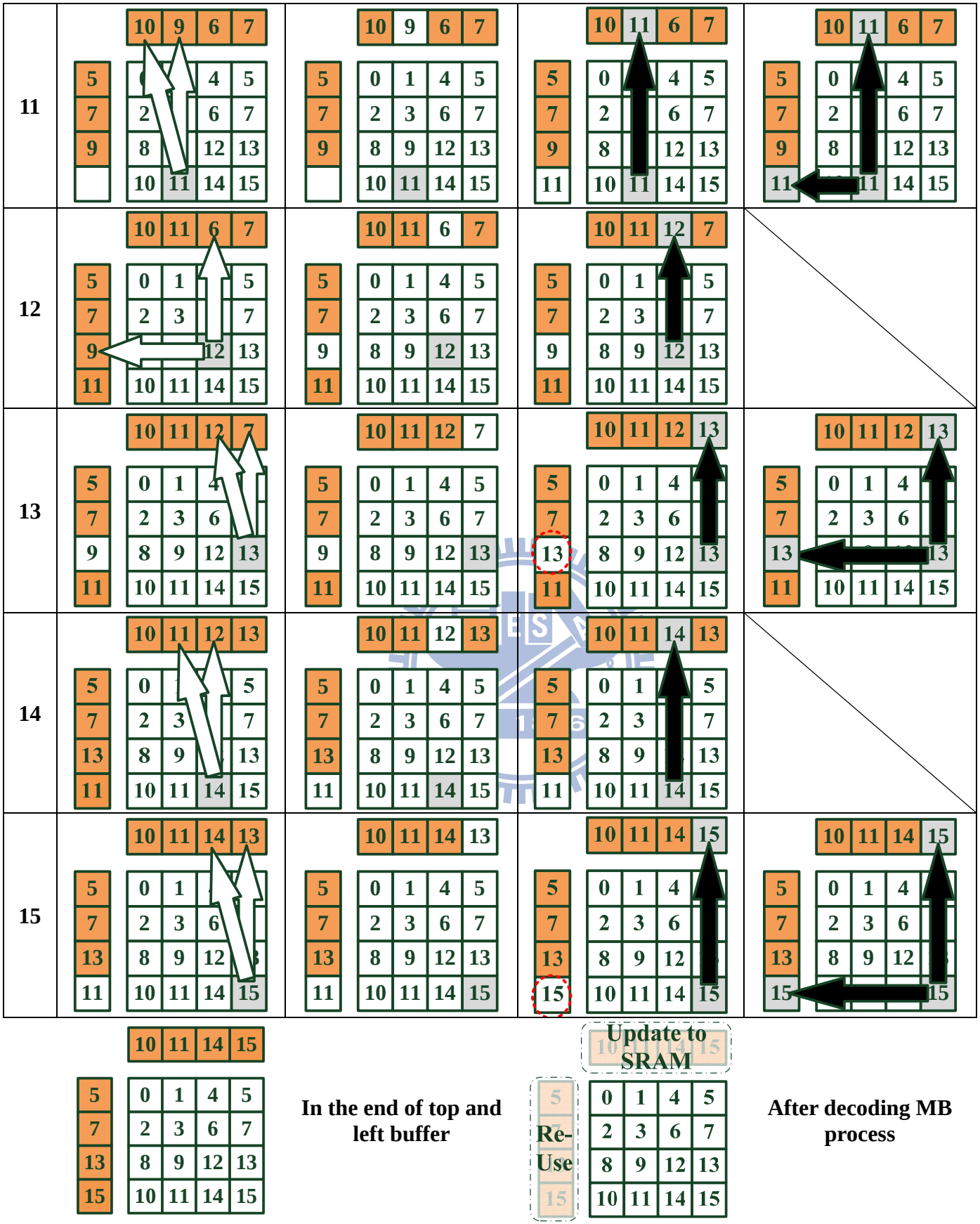


Figure 34. Schedule of concentrated buffer

3.2.3 Solved Syntax Element Switching Overhead

In [6], it clearly indicated the effect of SESO that seriously decreased performance. Prediction-base decoding flow and jointed parser and decoder are proposed to reduce this overhead. However, even if we can take care of SE branch, getting neighbor information may enhance SESO when we access data from external memory. So, we pre-calculate SE for neighbor MB described in Section 3.2.1 to reduce storage and alternately pre-fetch all neighbor information of MB like Figure 35 in a concentrated buffer described in Section 3.2.2 to avoid waiting for neighbor information when decoding first bin of SE.

We classify all pre-calculated SE to three kinds according as the order of SE. First kind of SE has the highest priority, because they require the neighbor SE at the beginning of MB shown in Figure 35. And we store this kind of SE, such as *mb_skip_flag*, *mb_type* and etc, in Row Storage 0. Because first kind of SE has low storage and high importance, they advise to supply in internal memory.

Second kind of SE contains Inter predictor mode and coefficient information. We should store 32 bits 2-bit mvd according as Section 3.2.1 and 16 bits for coefficient type SE, and we store them in Row Storage 1. Total 48 bits per MB can suit for 32 bits width bus, and we may only have to transmit one or two cycles to memory. This kind of SE can be stored in internal or external memory according as system constraint or cost.

The last kind of SE is about extra 5-bit mvd described in Section 3.2.1, and we store them to Row Storage 2. But, because not all of mvd has extra mvd, the total account of extra mvd may depend on sequence. Therefore, we should concentrate all separated extra-mvd to promote memory utility. Because this kind of SE is the largest part for neighbor information and seldom used, we can store them to external memory.

Finally, through our adjustment, the latency for getting neighbor information reduced to average 0.868 cycles per MB. Simulation results would show in Chapter 5.

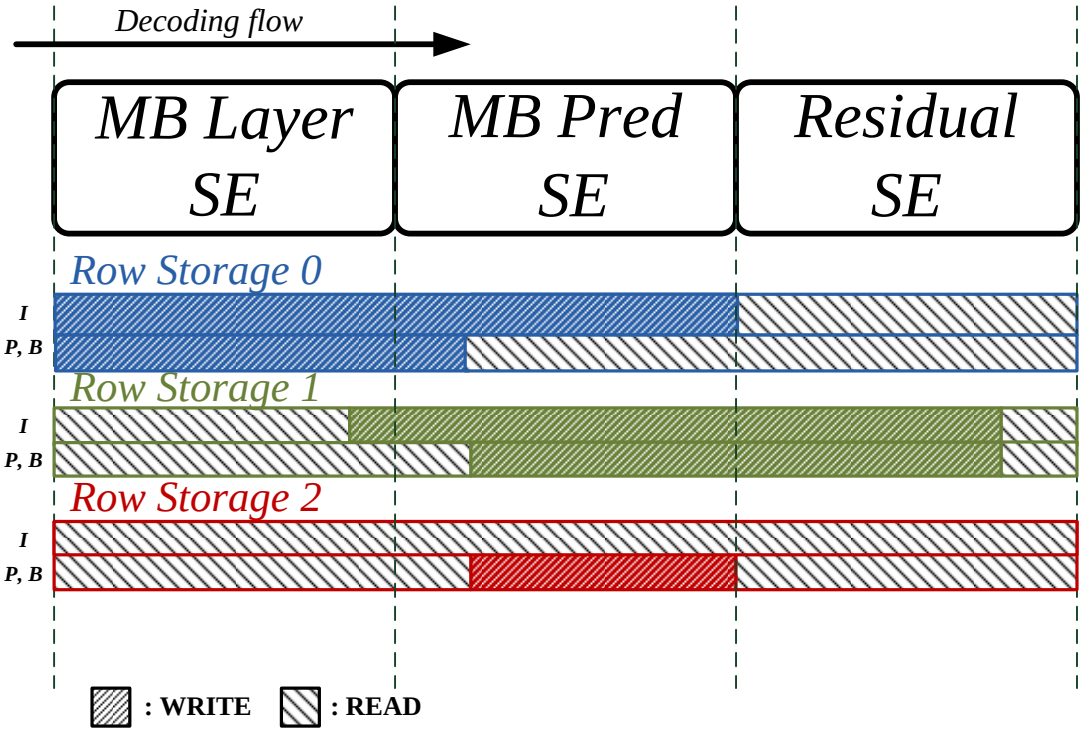


Figure 35. Alternate order for all neighbor information of MB

Table 12 Total request syntax element of macroblock

	Syntax Element	Num	Trad.	Cond.	Total bits per MB
RS_0	mb_skip_flag	1	1	1	*8 bits / MB*
	mb_field_decoding_flag	1	1	1	
	mb_type	1	6	1	
	intra_chroma_pred_mode	1	2	1	
	ref_idx_l0	2	5	1	
	ref_idx_l1	2	5	1	
RS_1	mvd_l0 (x, y)	8	10	2	*48 bits /MB*
	mvd_l1 (x, y)	8	10	2	
	transform_size_8x8_flag	1	1	1	
	coded_block_pattern (luma)	1	4	2	
	coded_block_pattern (chroma)	1	2	2	
	coded_block_flag (Y_DC)	1	1	1	
	coded_block_flag (Y_AC)	4	1	1	
	coded_block_flag (UV_DC)	2	1	1	
	coded_block_flag (UV_AC)	4	1	1	
RS_2	mvd_l0 (x, y)	8	10	5	*80 bits /MB*
	mvd_l1 (x, y)	8	10	5	

3.3 Summary

In our design, we propose a new prediction-based CABAC decoder. We predict bin value instead of syntax element and let the design flow more regular. Therefore, we just require two kinds of predict result process shown in Figure 36. When we get hit process, we may have smooth pipeline without any data hazard. Even though we get miss penalty, we may stall one cycle to calculate correct ctxIdx. Furthermore, we propose a high-accuracy prediction process to get more than 90% hit rate. Above all, we get a high throughput, because we seldom have miss process.

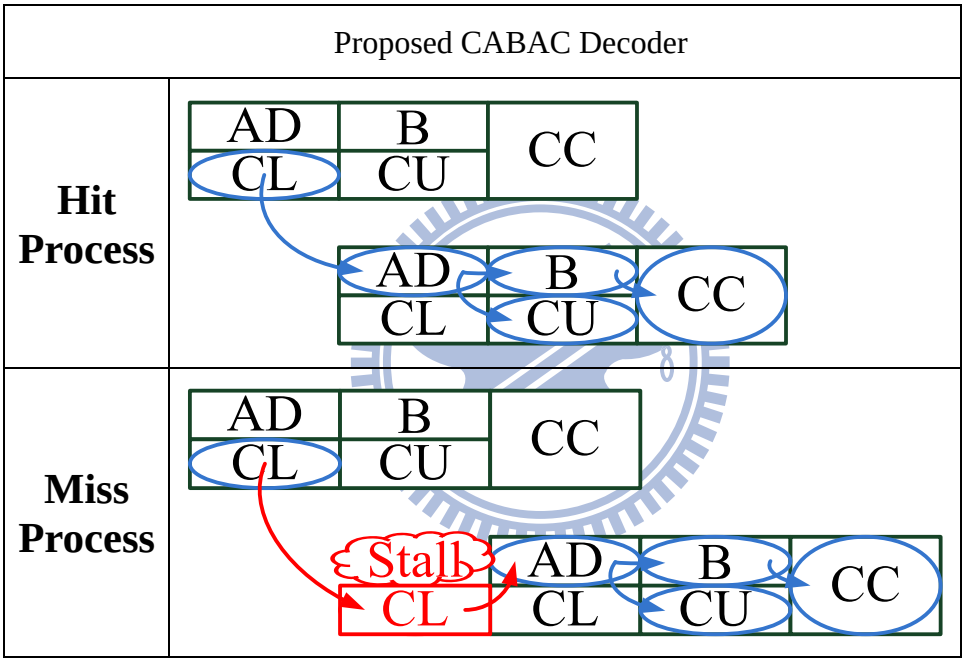


Figure 36. Proposed pipeline stage process

Besides, how to store the neighbor information and fetch them without extra latency may be more and more important when we raise the throughput continually. In our work, we reduce the storage and store all information of MB we require. So, we can get least increased latency and memory size. And, we concentrate internal buffer to reduce hardware cost. Above all, we ease the possible situation for performance lost and get the minimum hardware cost.

Chapter 4. *Proposed Architecture*

This chapter implements our proposed prediction-based CABAC decoder architecture by our proposed scheme. Figure 37 shows block diagram of our proposed CABAC decoder.

First, in order to overcome the fundamental defects of CABAC algorithm for pipeline structure, we make an effort to parallel the bin-decoded process and ctxIdx-calculated process by two independent paths. Therefore, we combine controlled parser and increase bin predictor to traditional CABAC decoder. And, the bin predictor is implemented by our proposed algorithm described in Section 3.1. By the high hit rate, we can achieve our purpose successfully. And, the details would be described in Section 4.1

Second, we consider the possibly additional overheads behind system integration. One of them is we may produce a large storage and latency for neighbor information. The storage overhead would increase hardware cost or memory bandwidth requirement. The latency overhead may cause unexpected pipeline stalls. Therefore, we optimize the memory system to reduce storage and schedule the scheme for real-time accessing neighbor information. Besides, we get acceptable throughput by simple buffer and resource. And, the details would be described in Section 4.2.

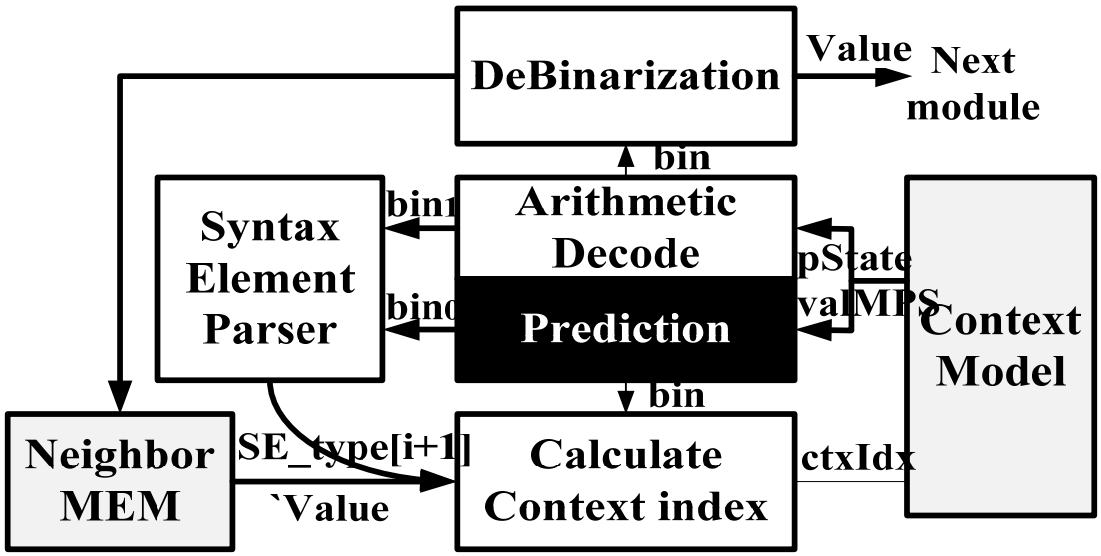


Figure 37. Block Diagram of proposed CABAC decoder

4.1 Architecture of Prediction Process

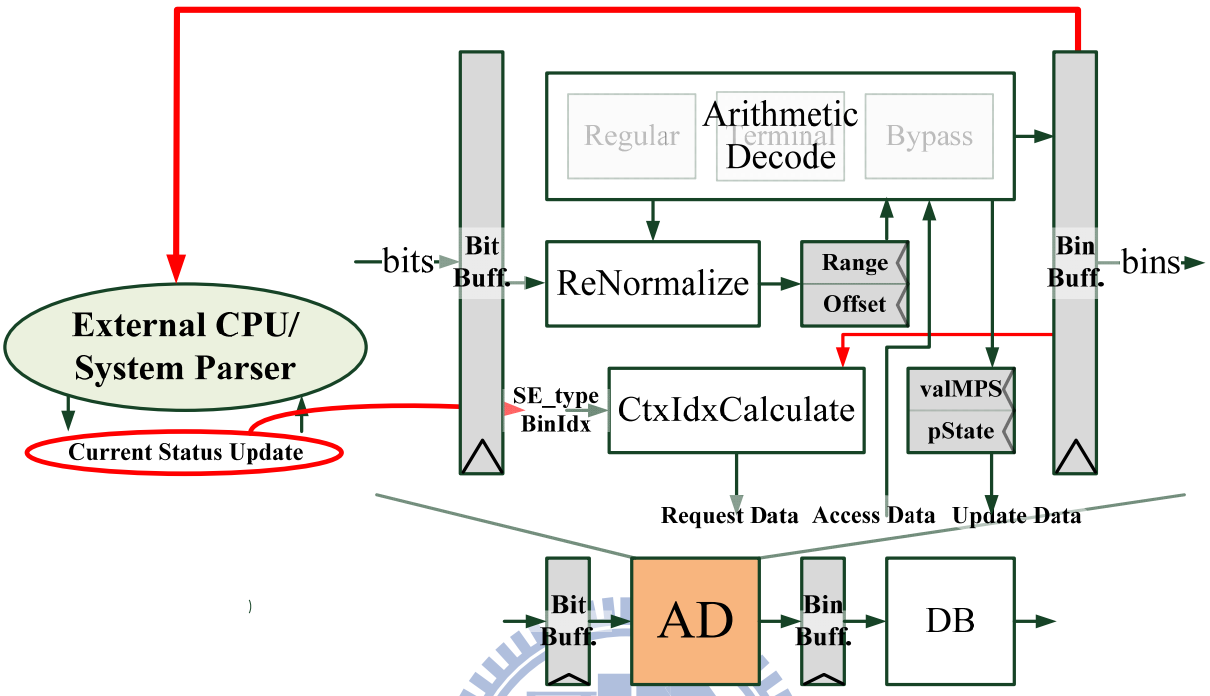


Figure 38. Traditional Arithmetic Decoder flow

At first, we focus on AD unit, because this is the critical part for throughput. For our purposed, we would like calculate current bin and next ctxIdx at the same time. And, we may have bin-decoding path and ctxIdx-calculating path to implement. The bin-decoding path inputs current values (Range, Offset) and context data (valMPS, pState) to Arithmetic Decode (AD) Unit, and it would output current bin to bin buffer. The next values (`Range, `Offset) which are outputted by AD may be adjusted by Renormalize Unit and bits. Besides, the ctxIdx-calculating path inputs current status and outputs ctxIdx by CtxIdxCalculate Unit. As we can see in Figure 38, we find out ctxIdx-calculated process may be affected by bin-decoded process. The reason is we calculate next status from decoded current bin. At the traditional decoding flow, the next status may be updated by external CPU or system buffer. So, even if we do the best optimization with the traditional CABAC decoder, we still may have a limit for throughput rate because of uncontrolled SE parser. This dependency may be major part which cause performance lost.

Therefore, we try to parallel the bin-decoded process and ctxIdx-calculated process by two independent paths. We provide a controlled SE parser and record each status of pipeline stages in shift registers to solve SESO, and we described in Section 4.1.1. And, we discuss our bin-decoding process and ctxIdx-calculating process in Section 4.1.2 and Section 4.1.3 to become two independent paths.

4.1.1 SE-parsed Process

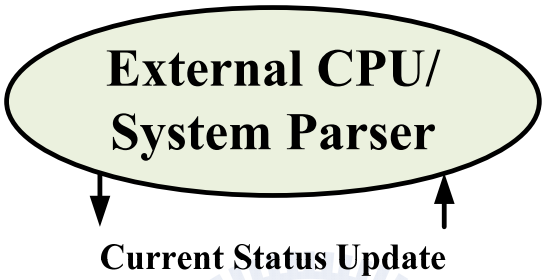


Figure 39. Traditional syntax element parser

In Figure 39, we use external CPU or system parser to determine which SE type may be decoded. Although this kind of architecture has the advantage of relatively uncomplicated implementation, this also may invoke performance lost significantly due to SESO which hasn't been considered with previous status. So, we provide a controlled SE parser instead of traditional SE parser. However, even if we consider all SE branch location, we still may get trouble at some situation, for example, the same SE in different block … and so on. As a result of some SE which is more than one in different blocks, we should record each status of stages. Hence, we collect the detail items of status which we have to store in Table 13.

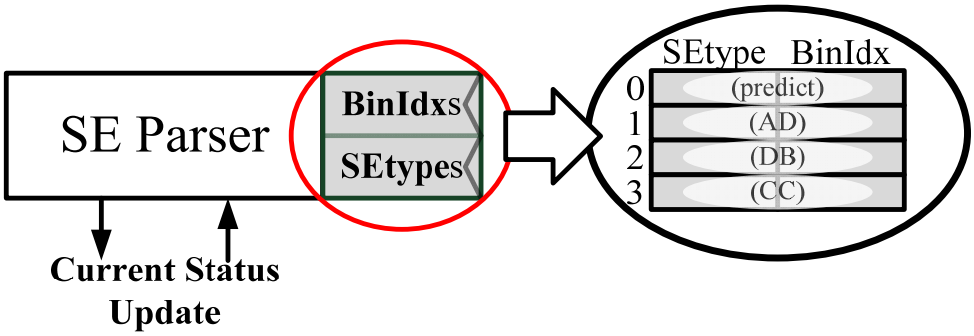


Figure 40. Controlled SE parser

Table 13 The stored status of each stage

Stages		Pr (N)	AD(0)	DB(1)	CC(2)
Stored		No	Yes	Yes	Yes
Status	*ctxIdx*	V	V		
	SE_type	V	V	V	V
	binIdx	V	V	V	V
	mbPartIdx	V	V	V	V
	subMbPartIdx	V	V	V	V
	ctxBlockCat	V	V	V	V
	CoeffBlkIdx	V	V	V	V
	UV	V	V	V	V
	levelListIdx	V	V		

4.1.2 Bin-decoded Process

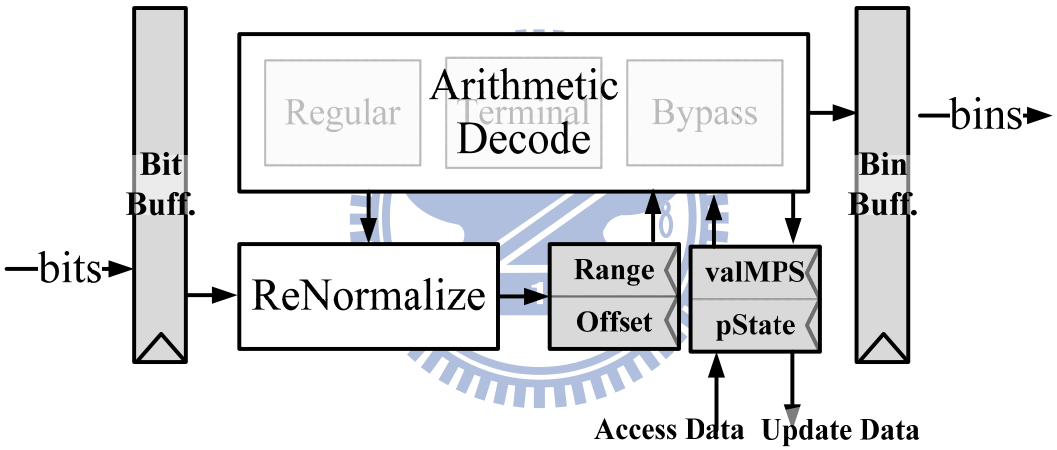


Figure 41. Data path of bin-decoded process

To implement bin-decoded process, we have to apply some components. Arithmetic Decoder unit which includes regular mode, terminal mode and bypass mode is made to decode bin described in 4.1.2.1 and 4.1.2.2. Regular mode Renormalize unit is made to adjust current value (range, offset) over correct level by bits. Besides, we may require a bit buffer which is made with FIFO structure to deal with flexible accessing bits described in Section 4.1.2.3. And, we also require some internal buffers to hold on current value (range, offset) and context data (valMPS, pState). Next value may be used soon, and previous context data may be selected to update or reuse described in Section 4.1.2.4.

4.1.2.1 Regular process

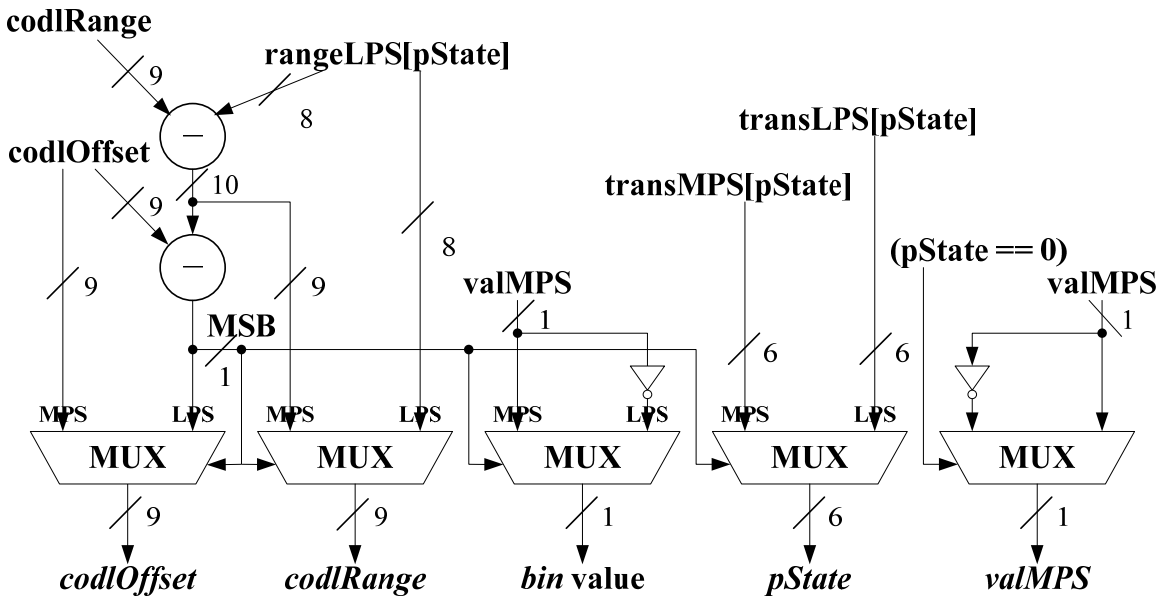


Figure 42. Regular process

According to our analysis, the regular process has more than 70% occurrence rate in total decoding process. In addition, it mostly would be the critical path restricted working frequency. Hence, we take an effort of shorting the critical path. In our design shown in Figure 42, we use two subtractors, two inverters and five 2-to-1 multiplexers. For each regular process, we input current *codIRange* and *codIOffset* which are changed by previous process repeatedly. So, the current range and offset may be updated all the time for most of regular process. Besides, we should know the history information by context model. The context data include pState and valMPS. The pState represents the index of LPS range table, and the valMPS represents the symbol with most of occurrence rate. According to context data, we can get range of LPS and next pState. So, the only thing we should do is recognizing the bin. The bin is determined which status is occurred, most possibility symbol (MPS) or least possibility symbol (LPS). And, the two statuses are determined by context data. Hence, we can find out the critical path in our design is the time we recognized MPS or LPS. Different with traditional methods, we use two subtractors, one rLPS table to get the result. This may spend much less time compared with using comparer directly.

4.1.2.2 Bypass/Terminal process

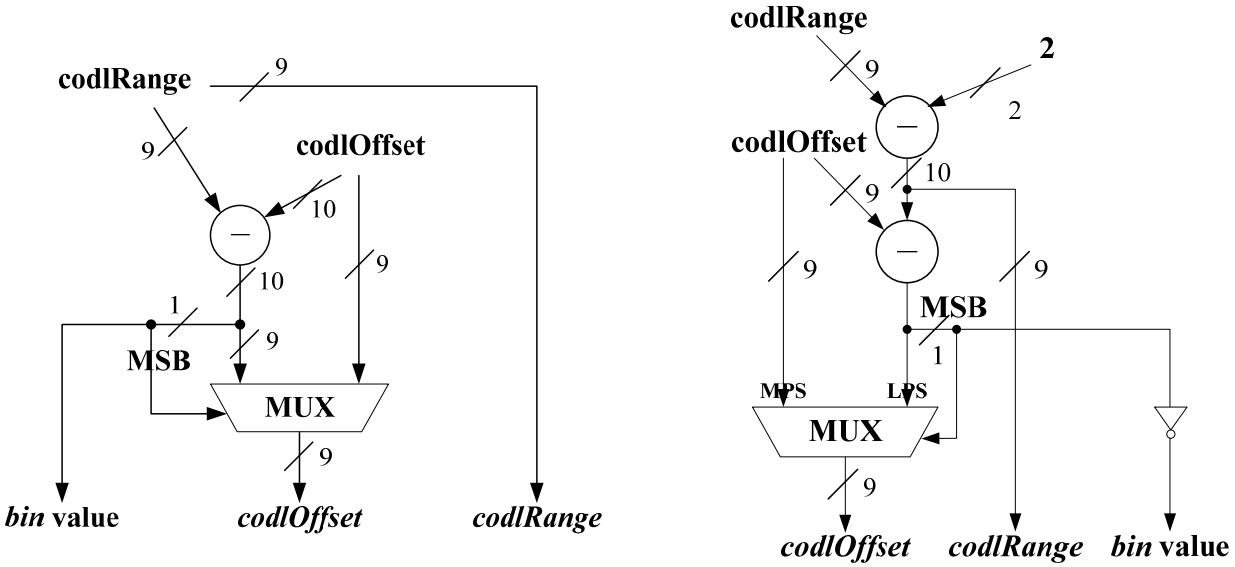


Figure 43. (a) Bypass Process (b) Terminal process

In this section describes other decoding process. Bypass process has second high occurrence rate about 15% ~ 30% shown in Figure 43(a). Terminal process may occur when decoding the end of MB and be seldom used and shown in Figure 43(b).

Because we don't require to context data, we can simplify hardware cost and data path visibly. In our bypass process, we use only one substrate and multiplexer to implement. We shouldn't change range, but we should update offset all the time. Besides, we output one when offset bigger than range and zero when opposite status.

In the other hand, terminal process is very similar with regular process. The different part may be we fix the range of LPS to 2, and we don't require context data. And, most of terminal process may be decoded zero, because one represents to end of slice. In our design, we use two subtractors, one inverter and one multiplexer to implement. Furthermore, this process doesn't have to load data from table, and it may have shorter data path than regular process.

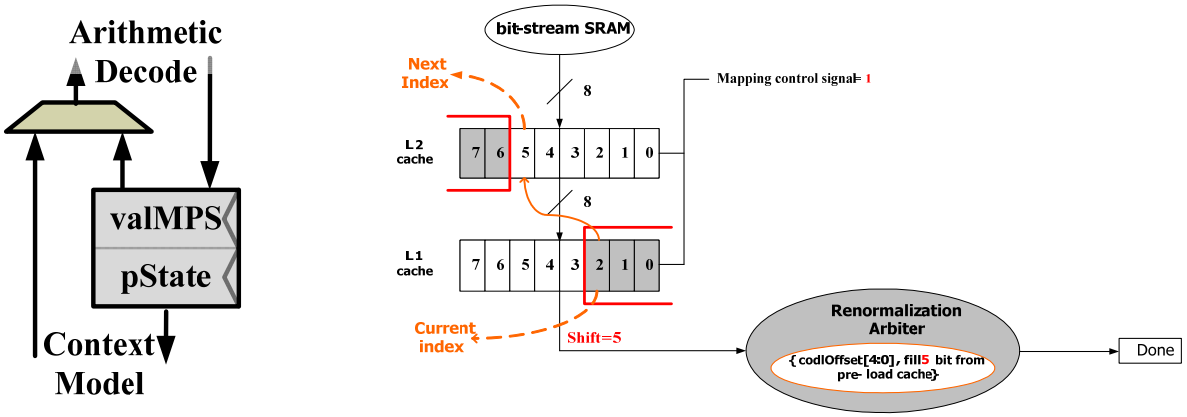
However, these two processes don't require to access data from memory. Actually, because we shouldn't care about ctxIdx, we never have miss penalty and can make sure of 1 bin per cycle in these two processes.

4.1.2.3 Context model data reused

As mentioned in Section 3.1.3.2, we require a internal buffer to control context data. The context data can be selected from new or updated one. Actually, we often decode bin by new context data in regular pipeline structure, and we store the updated one to buffer for writing context model at next stage. Therefore, we can just use a multiplex to pre-load context data just like Figure 44(a), and we compare the current ctxIdx with the previous ctxIdx to decide which one. So, we can get no delay according to the data reuse, if the address of context model is the same with previous one.

4.1.2.4 Renormalization process and bit buffer

In this part, we have to design a bit buffer to overcome flexible bit requirement. The renormalization process is used to adjust range and offset. So, the bits may be parsed in bypass process or in other process according as the range which is smaller than normalize level. In our analysis, we may require 0 ~ 6 bits in regular process, 0 ~ 1 bit in terminal process and 1 bit in bypass process for each fetching. The method we used is that we apply a L2 cache with FIFO structure shown in Figure 44(b). We always fill the buffer when containing under half of entire. Actually, we record the number of bits in buffer and monitor each fetching of requirement. If we find out that we aren't enough bits in buffer after this fetching, we would send a signal to require more bits from bit-stream SRAM. Through this buffer, we can support all possible cases for fetching the bits. Besides, we use the same renormalize unit in regular process and bypass process to achieve high hardware utility efficiently.



(a) (b)

Figure 44. (a) Data reuse buffer. (b) The structure of bit buffer [9]

4.1.3 CtxIdx-calculated Process

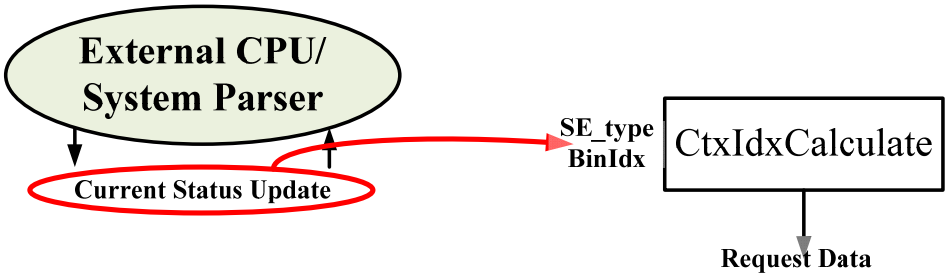


Figure 45. Conventional ctxIdx-calculated process

In fact, before we calculate ctxIdx, we should know what kind of SE type we decode. The conventional method is used external CPU/system parser to handle the complex SE branch. And, CABAC decoder may be idle until external host invoked. After that, we may get the SE type and decode bin by bit stream. However, as mention in previous, this method may produce some unexpected statuses. So, we improve the conventional process to suit our controlled SE parser.

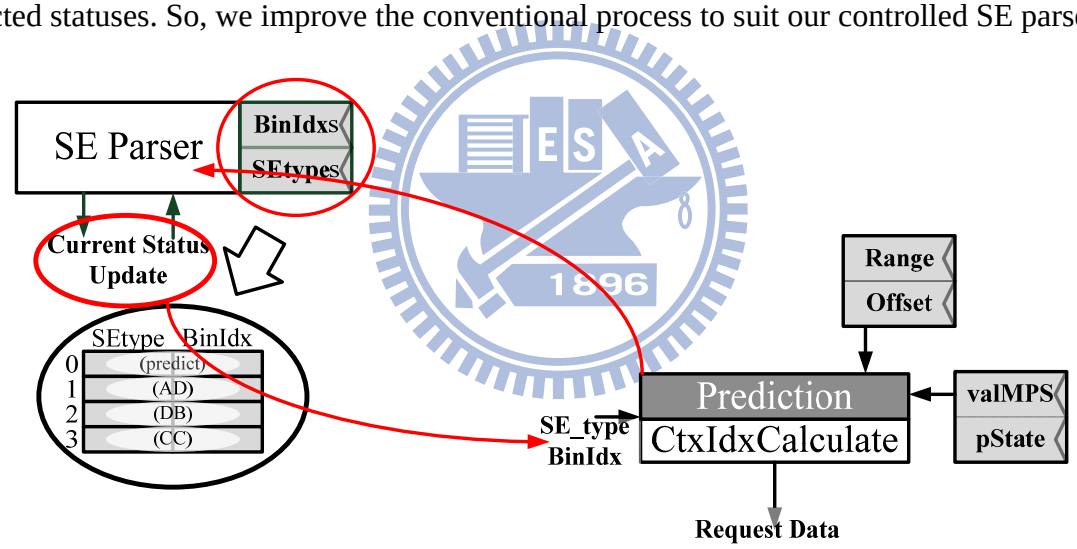


Figure 46. Proposed ctxIdx-calculated process

The controlled SE parser we has introduced in Section 4.1.1. And, we have to cut off the relationship with bin-decoded process and ctxIdx-calculated process. According to prediction unit which is made by Section 3.1, we can get the predicted bin_0 which can be produced at the same time with decoded bin_1. Instead of decoded bin_1, we use predicted bin_0 to get a next status by SE parser. And, the next status may have no dependency with decoded bin, but it may have a probability to be wrong. We can calculate ctxIdx continually until occurrence of the error predicted bin_0. Hence, we can ease data dependency, if the probability of the error predicted bin_0 is low.

4.2 Architecture of Memory System

After we implement the ctxIdx-calculated process and binIdx-process, we still have some problem to be overcome. The major problem is memory issue, and we should include several SRAM to support CABAC hardware. As mention in Section 1.2, we have two memories which are difficult to handle. One is context model, and the other is neighbor information. Because we can ease data dependency by our proposed methods, we just require a two-port SRAM which supports reading and writing at the same cycle for context model. Besides, the other problem is about how to access neighbor information when we require. In this Section, we provide a technology to access neighbor information immediately and increase a little overhead shown in Figure 47. In order to avoid waiting for loading neighbor information, we overcome this problem by including left and top neighbor buffer. The concentrated buffer has been described in Section 4.2.1. Moreover, we increase one stage (ctxIdxInc pre-calculate (CC)) to reduce the information and describe in Section 4.2.2. After we construct the pipeline architecture, we may figure out another problem. Because not all of mvd are bigger than 3, extra mvd would be too separate to achieve high storage reduction. Therefore, we implement a transfer unit described in Section 4.2.3 to reach high data compression.

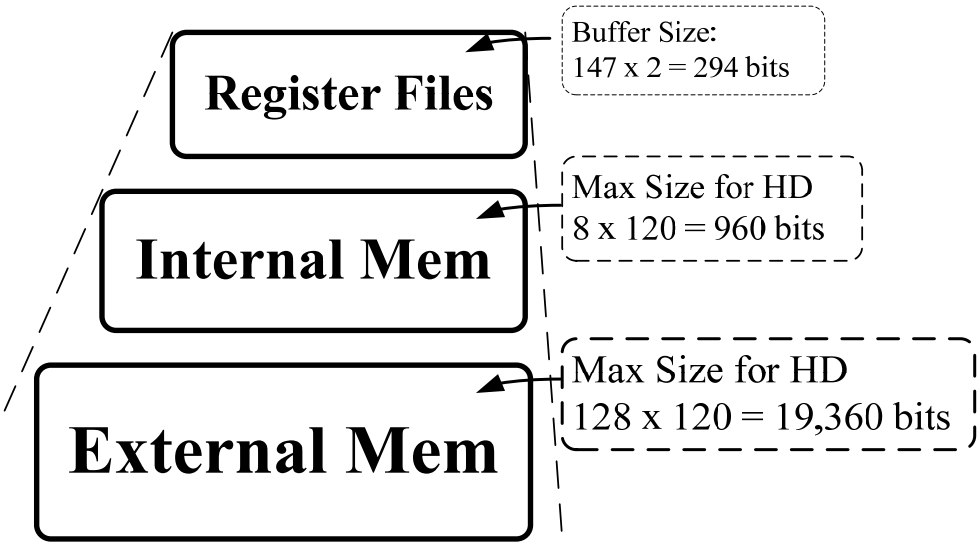


Figure 47. Memory hierarchy for neighbor information

4.2.1 Concentrated Buffer

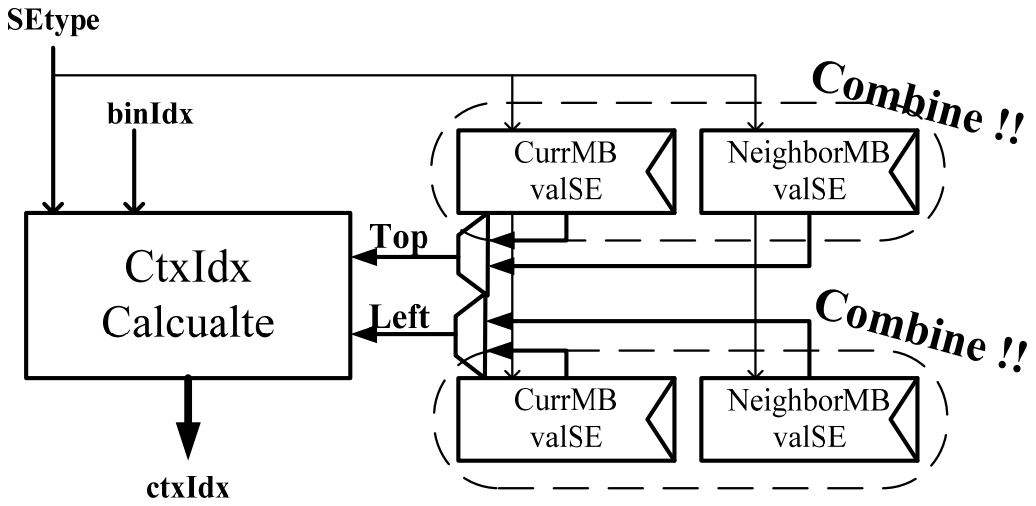


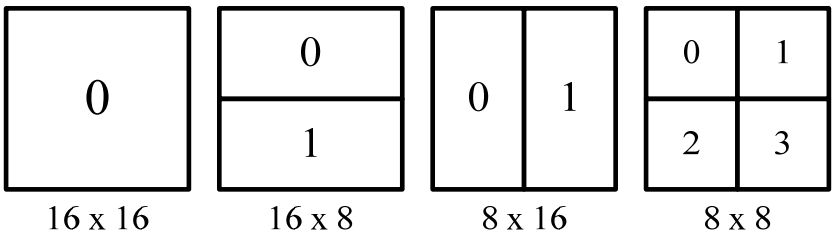
Figure 48. Combined current and neighbor MB

As soon as we decode the first bin of the SE, we may require the neighbor information to calculate ctxIdx. However, we can‘t make sure what information would be fetched. So, we usually store all information of MB to deal with flexible accessing neighbor information, and it may include a large buffer. According to supporting variable block size, we may raise two times buffer size. Hence, as mention in Section 3.2.2, we combine the two kinds of buffer and are shown in Figure 48. Actually, according as our updated schedule, we can use half of buffers to achieve the same proposed.

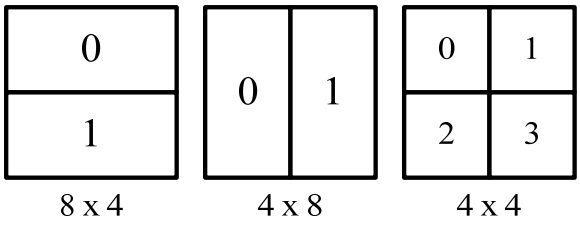
Nevertheless, this method may not be available in some situations. In Section 3.2.2, we assume all macroblock have 16 blocks, and we can update smoothly. But, H.264/AVC supports variable block size actually. And we may face different block size in neighbor. So, we should discuss this kind of situation and make an effort in Section 4.2.1.1.

On the other hand, when we upgrade over H.264/AVC main profile, and we may support a technology, Macroblock-Adaptive Frame-Field Coding (MBAFF) mode. MBAFF means that we can allow frame and field type in the same slice. Because we have to adapt for field mode, we should classify MB to top and bottom. Therefore, we should consider several possible cases in neighbor, and there is much higher complexity for supporting MBAFF mode. So, we may analysis all possible situations and discuss in Section 4.2.1.2.

4.2.1.1 Various block size



(a) macroblock partition



(b) sub-macroblock partition

Figure 49. (a) macroblock partition (b) sub-macroblock partition

In H.264/AVC, we can recognize the block size by mb_type and sub_mb_type which are two kinds of SE. We may have four types by mb_type shown in Figure 49 (a), and we even may have extra four types by sub_mb_type shown in Figure 49 (b). Therefore, when we decode current MB, we can understand what block size is. In the other hands, we may have no idea what neighbor block type is. To overcome this problem, we should make an effort for writing back information. When we decode block type except for 16x16 block size, we should assume all MB which is 16x16 block size. And, we extend the decoding block to neighbor blocks like Figure 50. Because we prepare the information in each block, we don't have to care about what block size is in neighborhood. After that, we can support various block size by increasing some data path. By the way, because not all of SE supports sub-macroblock partition, we just have to increase data path for some special cases.

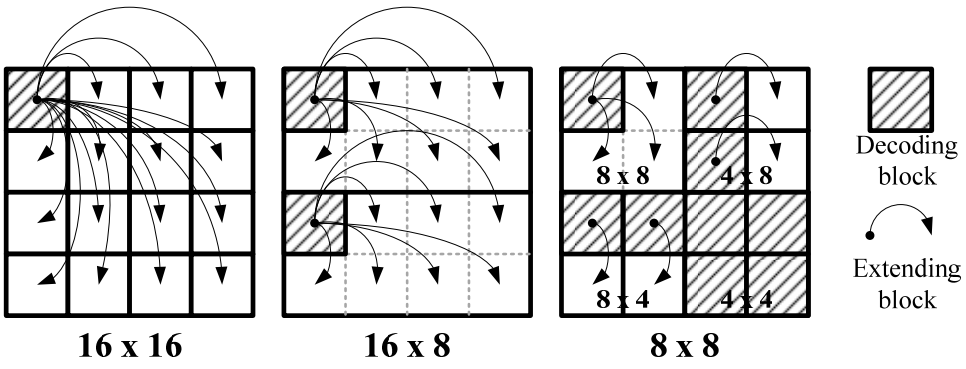


Figure 50. Example for block extension

4.2.1.2 MBAFF mode

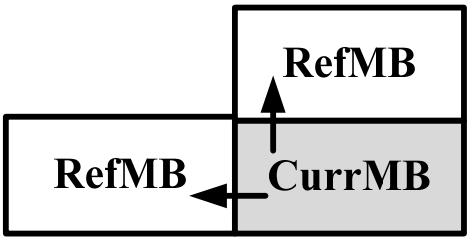


Figure 51. Without MBAFF mode

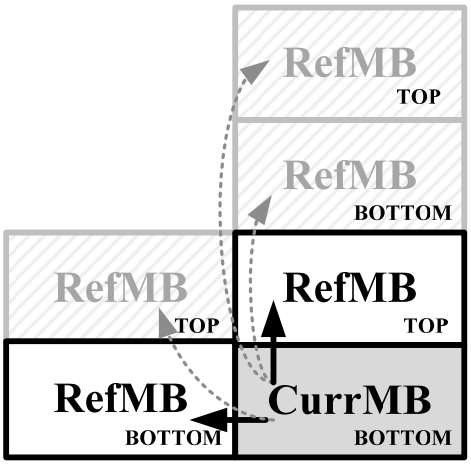


Figure 52. With MBAFF mode

Table 14. Without MBAFF mode

Case	CurrMB,	RefMB
0	Frame,	Frame
1	Field,	Field

Table 15. With MBAFF mode

Case	CurrMB,	RefMB
0	Frame,	Frame
1	Frame,	Field
2	Field,	Frame
3	Field,	Field

MBAFF is an innovative technology for coding tool, but it also increases high complexity for decoding. When we support MBAFF mode, we may consider the different with frame and field. And, the address for mapping memory, control conditions and formula from standard would be totally different. For example, Figure 51 is a case without MBAFF mode. And, we just should consider left and top neighbor MB, because the reference MB (RefMB) and current MB (CurrMB) are either frame type or field type. Figure 52 is a case with MBAFF mode, and each RefMB can be frame or field type. So, we may consider current MB which is top or bottom MB and also consider current MB which is frame or field type. Except for current MB, we also have to consider neighbor MB which is frame or field type. Therefore, above situations can change RefMB location, so we should provide more buffers to deal with increasing complexity instead of increasing system overhead. In our analysis, we should enhance 2.5 times internal buffer for the worst case shown in Figure 52.

4.2.2 CtxIdxInc pre-Calculate Stage

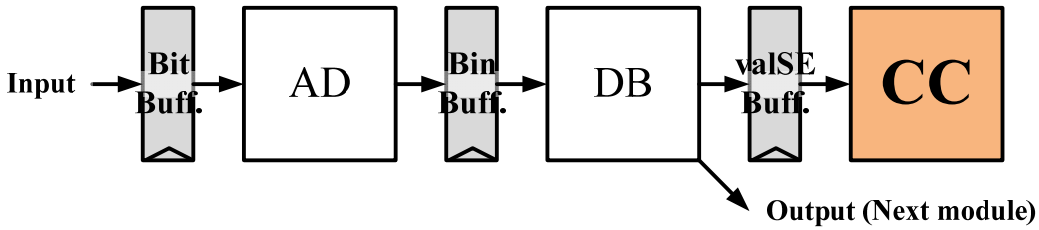


Figure 53. Incensement of third stage

To supply pre-calculated method as mention in Section 3.2.1, we provide a ctxIdxInc pre-calculate (CC) stage behind second stage shown in Figure 53. And, we output the value of SE (valSE) after second stage. It means the third stage which is used to reduce the information wouldn't affect previous pipeline stages.

In the traditional flow like Figure 54, we have to store all valSE of MB in memory. When we require neighbor information, we read them from memory. And, we calculate ctxIdx for CABAC decoder. This method may have more requirements for memory bandwidth and memory space.

In our proposed flow like Figure 55, we reorder the ctxIdxInc calculating flow. Before we store the information, we pre-calculate them and store dispersedly according as their utility rate. Hence, when we require the information, we can use them directly. And, we can simplify the data path to deal with the neighbor information.

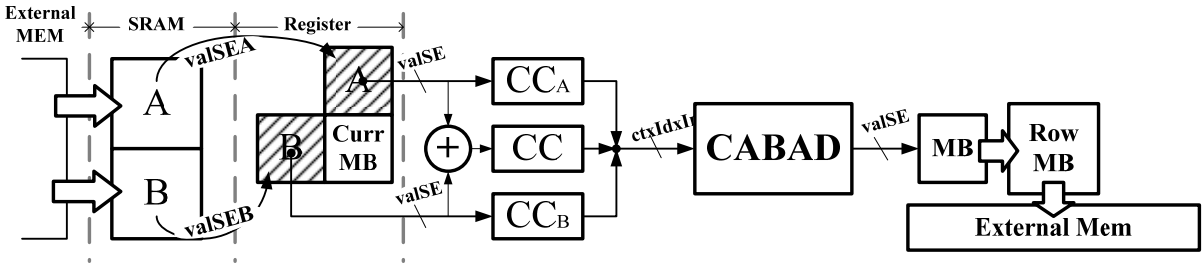


Figure 54. Traditional neighbor information calculating flow

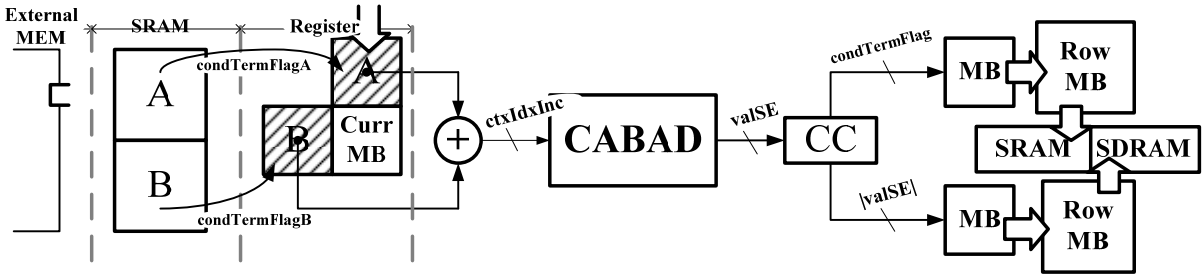


Figure 55. Proposed neighbor information calculating flow

4.2.3 Transfer Unit

After we support above technologies, we may find out that we can't get high reduction rate for neighbor information because of the dispersive extra mvd which is described in Section 3.2.1. So, we try hard to get consideration to high reduction rate and concentrated buffer, and we provide a transfer unit to compress the neighbor information.

In the transfer side, we collect at most 6 extra mvd with 5-bit and fill up from left to right to suit for 32-bit width bus shown in Figure 56. And, the transmission times can be changed from fixed 4 to variable 0 ~ 4. Because the extra mvd is about 20% of total mvd, we can reduce memory bandwidth requirement significantly. In the inverse-transfer side, we may decompress data, when we find out that 2-bit mvd equal to 3. After we received the data, we sort them according as 2-bit mvd to suit for concentrated buffer.

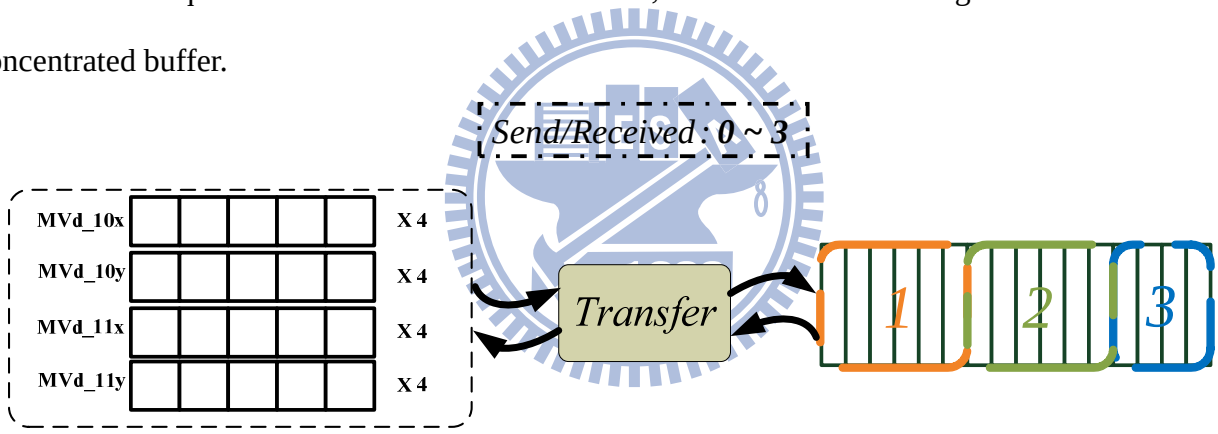


Figure 56. Transfer unit

Figure 57 is an example for compressing the dispersive extra mvd. Before our process, original extra mvd store dispersedly in different blocks, and we can just send or receive them once after our transfer process.

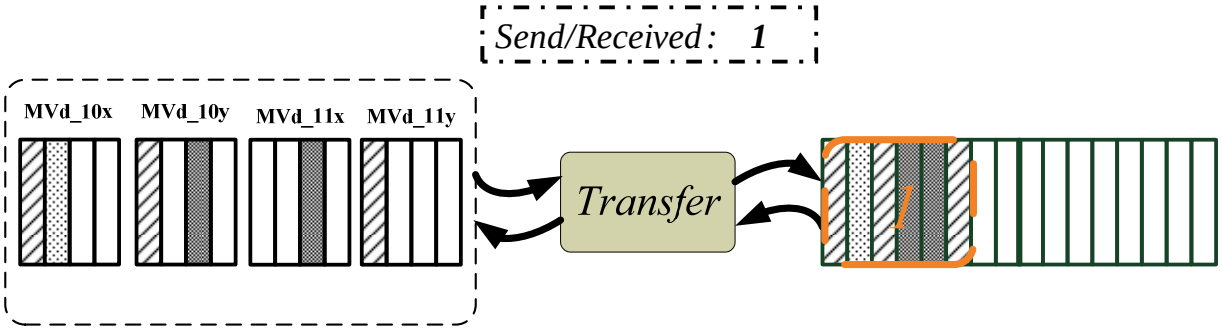


Figure 57. Example for transfer unit

4.3 Summary

After we overcome the problems of sub-unit, we may consider the issues about integration in system. We may discuss the situation about integration for each pipeline stage, memory system and entire CABAC decoding core. Different with traditional CABAC decoding flow, we combine SE parser and decoder. So, as soon as our CABAC decoder is invoked, it may decode continually until the end of slice. Besides, we may require an initialization process to initial context model data at the beginning of slice. After we finish each process we require, we make a FSM to control initialization process and decoding process.

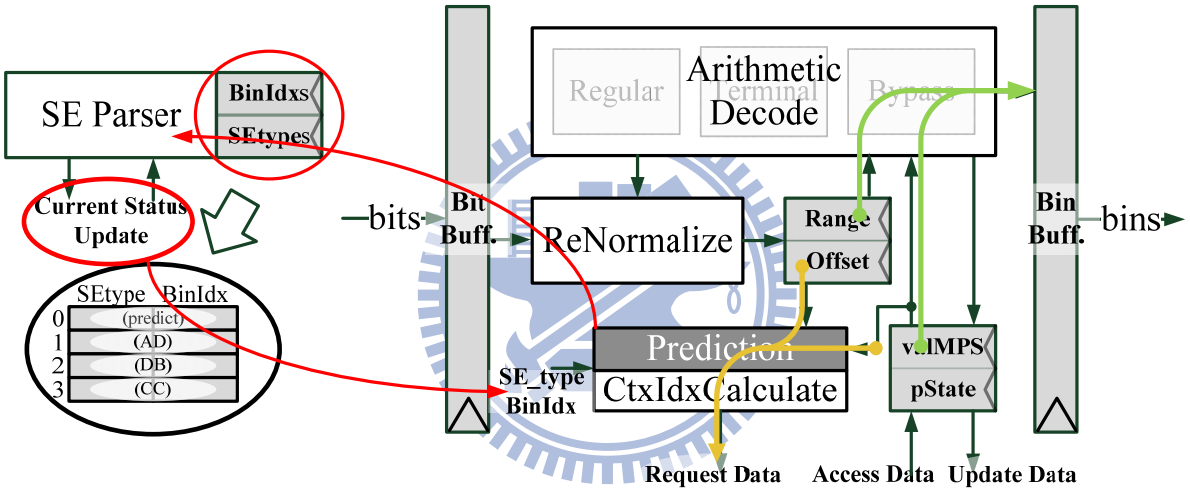


Figure 58. Integration for first pipeline stage

At first, we integrate our prediction process, bin-decoded process and ctxIdx-calculated process in first pipeline stage. And, we input the current values (range and offset) and context data to arithmetic decode and prediction unit. Because of our controlled SE parser (described in Section 4.1.1) and optimization of ctxIdx-calculated process (described in Section 4.1.1), we can decode bin and calculate ctxIdx at the same time. And then, the decoded bin may be stored in bin buffer, and the ctxIdx may be sent for requiring new context data. Besides, we may get the current values by renormalize process and bit buffer (described in Section 4.1.2.2), and the context data can be reused by internal buffer (described in Section 4.1.2.1). In Figure 58, we highlight these data paths.

However, after finishing above process, we still lack some data for calculating ctxIdx. The data is neighbor information. So, we have to supply neighbor information for ctxIdx calculate unit. As shown in Figure 59, we provide concentrated buffer (described in Section 4.2.1) and transfer unit (described in Section 4.2.3) to prepare neighbor information. And, left block of neighbor information can be reused, and top block of neighbor information may be updated with memory.

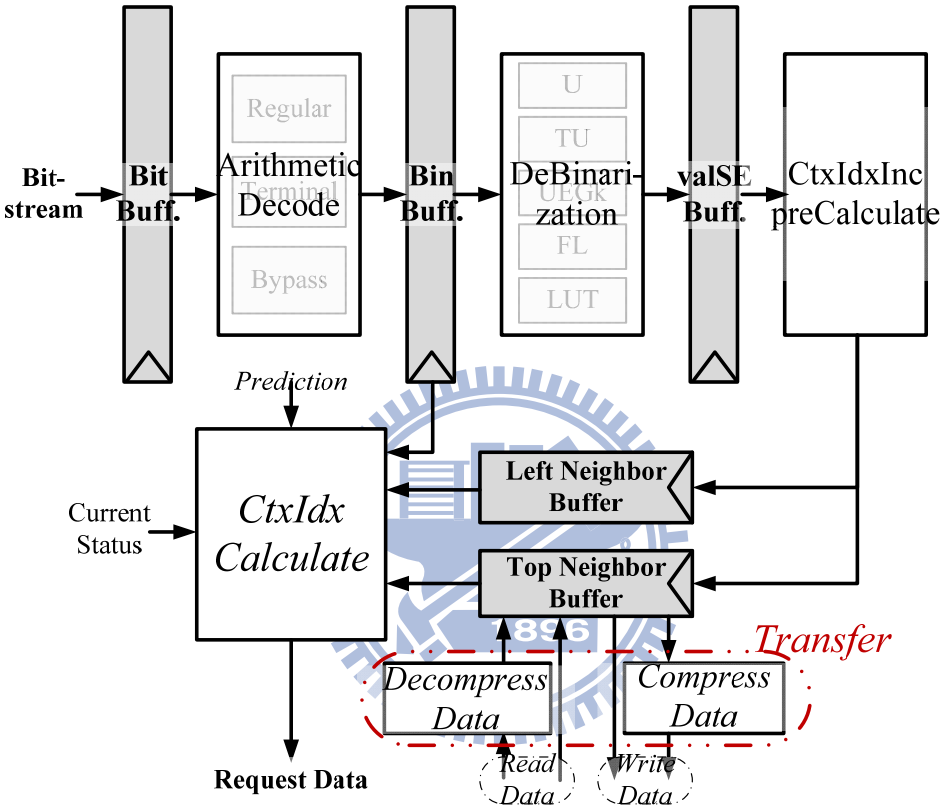


Figure 59. Integration for memory system

Second, after finishing first pipeline stage, we integrate de-binarization (DB) process and ctxIdxInc pre-calculate (CC) process (described in Section 4.2.2) to our CABAC decoding core. The DB process is inputted bins and SE type, and it may be outputted value of SE (valSE) to valSE buffer and system buffer. The CC process is inputted valSE and SE type, and it may be outputted conditional term flag to concentrated buffer for neighbor blocks. Because we have stored status of each stage in buffer of SE parser, we can make sure what SE type is in each pipeline stage. Therefore, according to low complexity and no data hazard of DB and CC process, we can implement second and third stages easily by some pipeline buffer.

Third, we integrate the whole pipeline stage (Part A) and memory system (Part B) to our proposed CABAC decoding core which is shown in Figure 60. After that, because this may cause data hazard problem in some special cases or SE, for example, coded_block_pattern, we increase a forward process to deal with this problem as mentioned in Section 3.1.3.1. Actually, we use multiplexer to select regular path or forward path to avoid data hazard problem.

In the other hand, we should have a process to deal with miss penalty. Because we adopt prediction bin_0 which has a probability to be wrong in our design, we may increase a risk for predicting miss. Therefore, we have to make a process to deal with this situation. Actually, even if we get miss penalty, it may be not things by some cases as mentioned in Section 3.1.2. However, we still have to stall in the worst case. To avoid occurring miss penalty one after another, we should stall one cycle and use previous decoded bin instead of predicted bin to calculate correct ctxIdx. And, the status buffer of SE parser shouldn't be shifted in this situation.

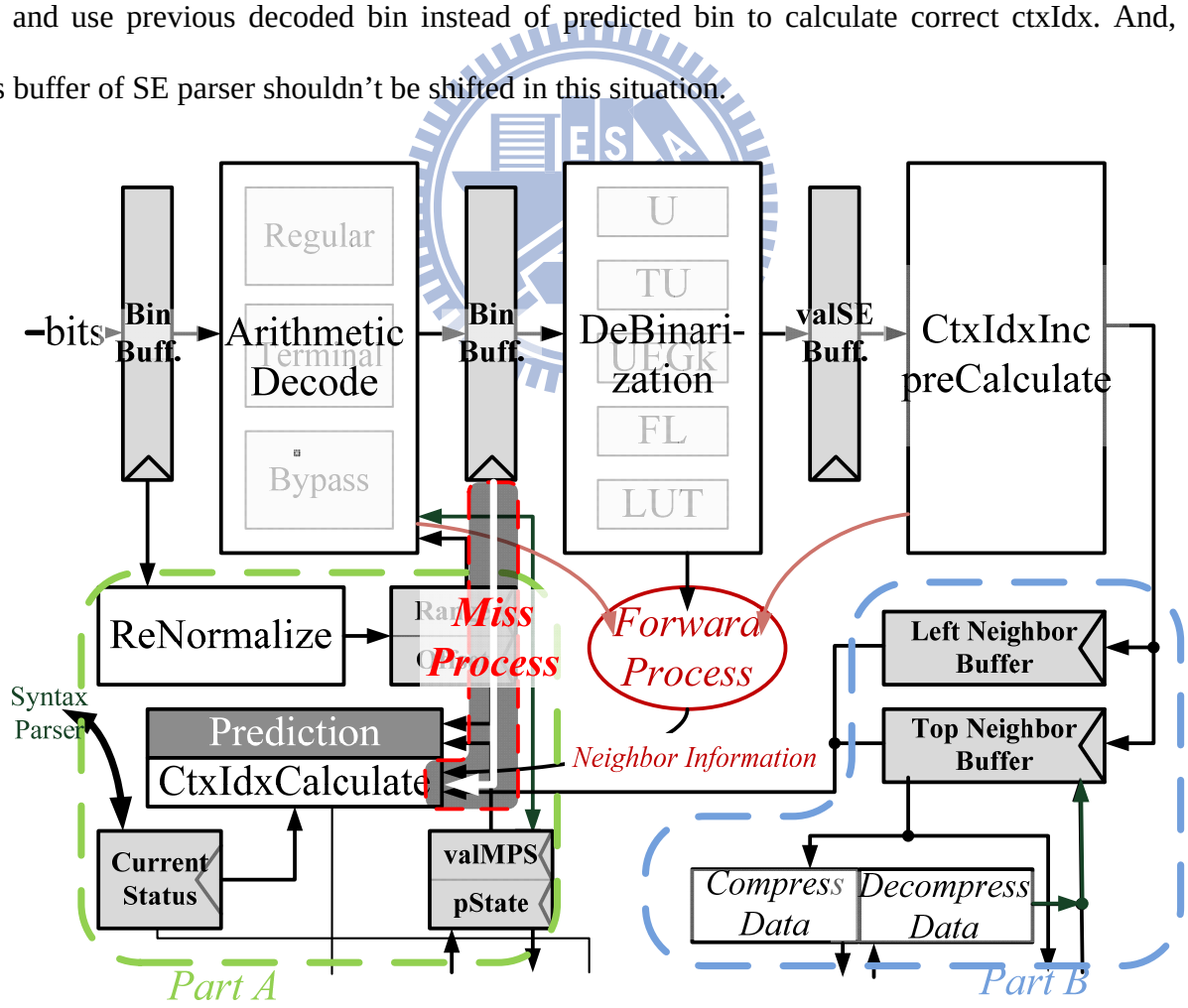


Figure 60. Integration of CABAC decoding core

After we finish our decoding process core, we still have to support a process before invoking decoding process. That is initialization process as mentioned in Section 2.1. This process is used to initial context model, and we should do it at the beginning of slice. Following is the pseudo-code from standard [1] for computing the single context model.

1. preCtxState = Clip3(1, 126, ((m * Clip3(0, 51, $SliceQP_Y$)) >> 4) + n)	(Eq. 11)

2. if(preCtxState <= 63) {
pStateIdx = 63 - preCtxState
valMPS = 0
} else {
pStateIdx = preCtxState - 64
valMPS = 1
}
	(Eq. 12)

Therefore, for our purpose, we require SliceQPy which is made from header and (m, n) which is made from Initialization table to produce initial value. According to the standard [1], the probabilities of preCtxState have to be kept between 1 and 126. Figure 61 shows the implementation of the context model initialization. It is divided into three stages, and is composed of one multiplier, one adder, and one subtractor.

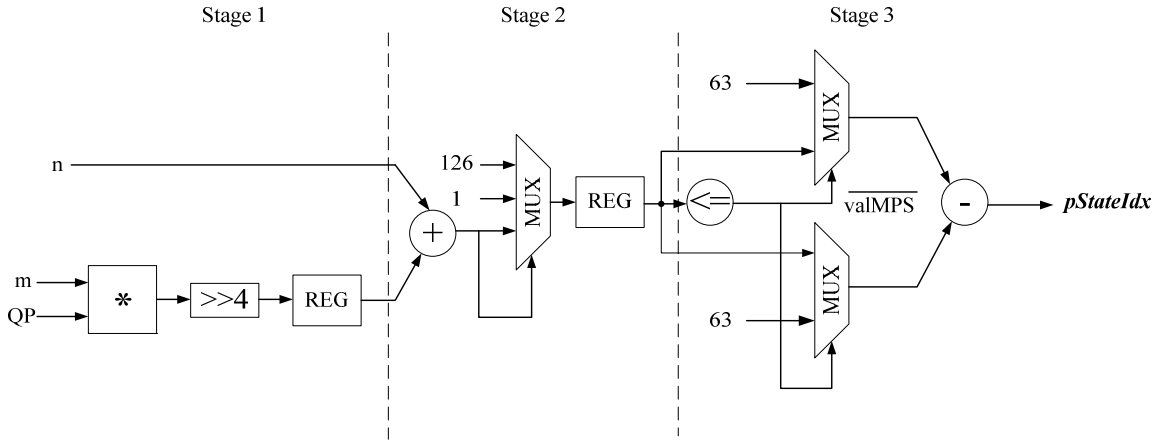


Figure 61. Initialization process [9]

Thus, the initialization can execute continuously with the pipeline. It depicts the action of three stages, and consumes 401 cycles per initialization occurrence.

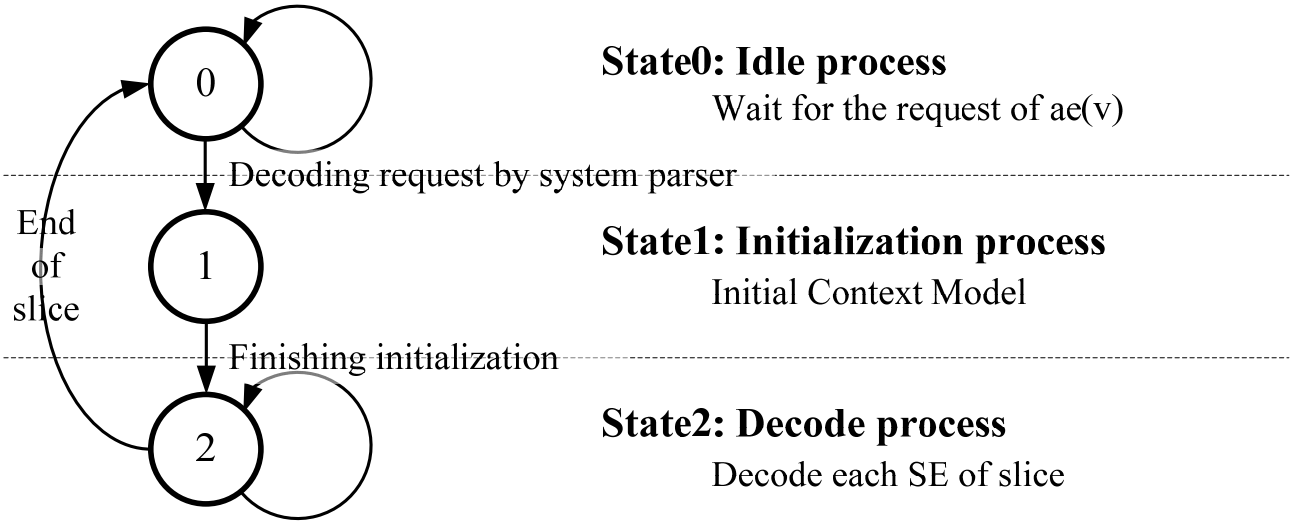


Figure 62. FSM for whole CABAC integration

Finally, after we finish our decoding process and initialization process, we may consider how to switch these processes and the memory issue. As shown in Figure 62, we supply a FSM about three states. The state 0 is supplied to shut down the decoder, when decoder isn't invoked. The system may work at the other entropy decoding in this state.

After the decoder is invoked, we may transfer to state 1. The state 1 is supplied to initial context model at the beginning of slice. We may require initialization table from ROM/external memory as shown in Figure 63. As computing each context model, initialization process would read (m, n) from initialization table. And, it may compute the initial context data to context model. Therefore, we just require working initialization process, reading parameter and writing initial context data.

As soon as we finish to initial context model, we may transfer to state 2. The state 2 is supplied to decode SE. Because of including the controlled SE parser, we shouldn't wait for external assignment. And, we can decode each SE of slice until the end of slice. Besides, we have to read and update new context model data from context model repeatedly, and we may change the neighbor information for each MB to internal and external memory shown in Figure 64. Finally, after we decode end of slice, we may return to state 0 waiting for next invoked.

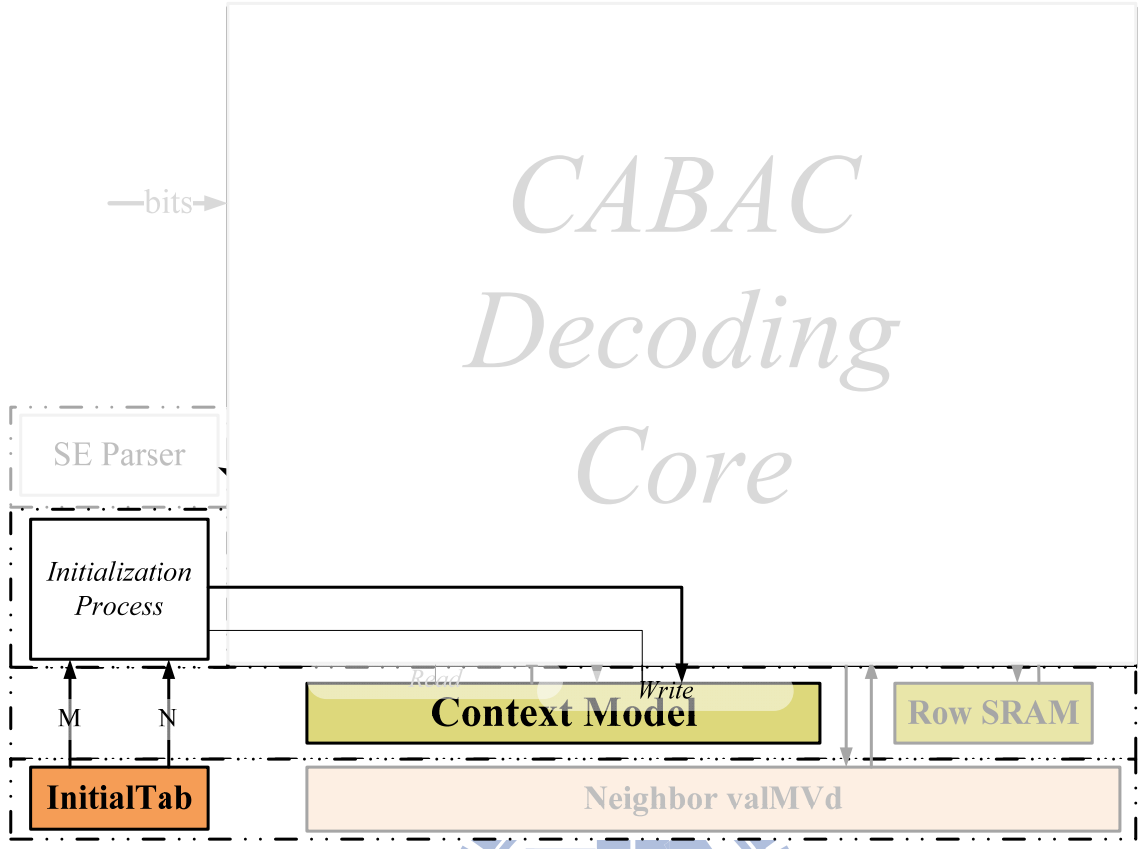


Figure 63. State 1 – Initialization Process

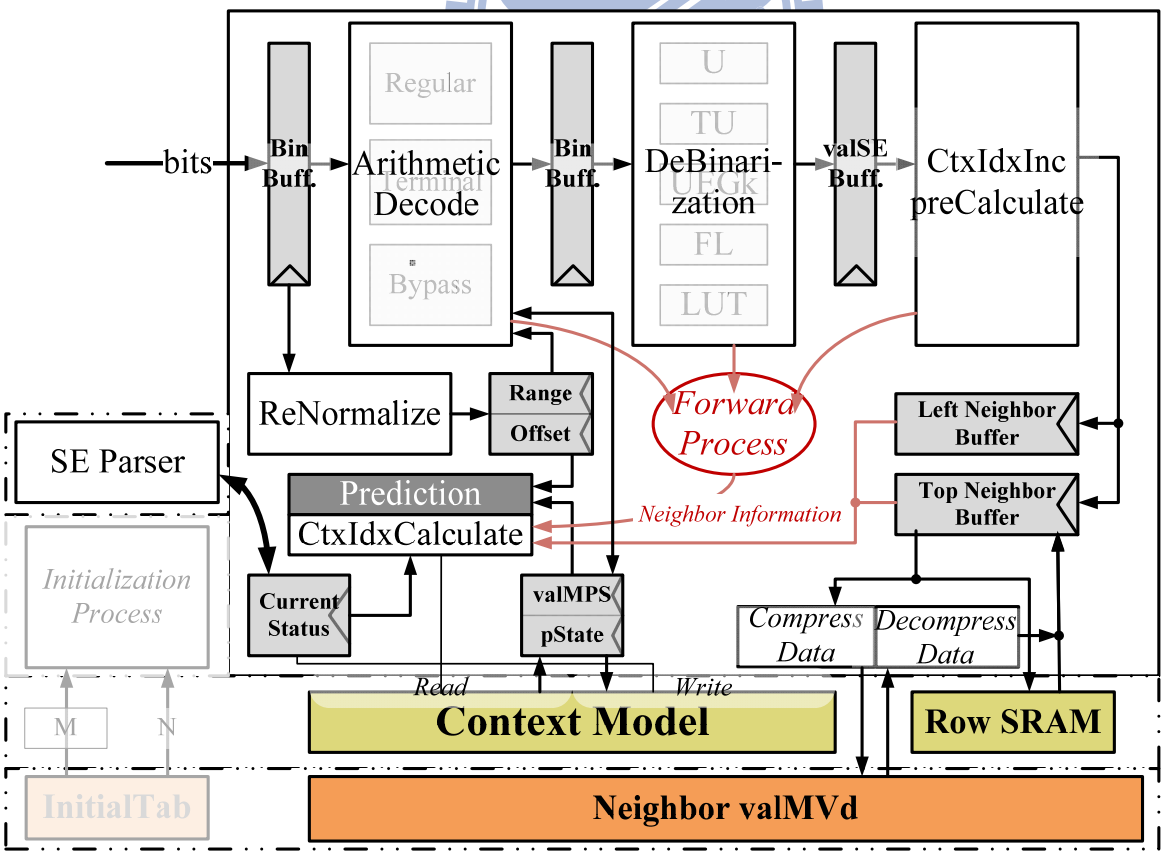


Figure 64. State 2 – Decode Process

Chapter 5. *Simulation Results*

5.1 Prediction Scheme Verification

To proof our prediction algorithm can adapt in different sequences, we make a simulation that we calculate hit rate at each bin of sequences in average with MPS rate. Figure 65 shows average hit rate of HD sequences and we get hit rate from 90.75% to 94.27% in all resolution sequences. It represents we can keep over 90% hit rate even in the worst case. Besides, we also make a simulation with variable QP in Figure 66 (fix $QP_{B,P}$) and Figure 67(fix QP_I). By the way, we put simulation results of other resolutions in Appendix B.

*(*Below simulations are tested by JM 16.1[2] with 4:2:0 color format,QP: 28, GOP: IBPBP…, Max video bit-rate and frame rate of 30 fps.)*

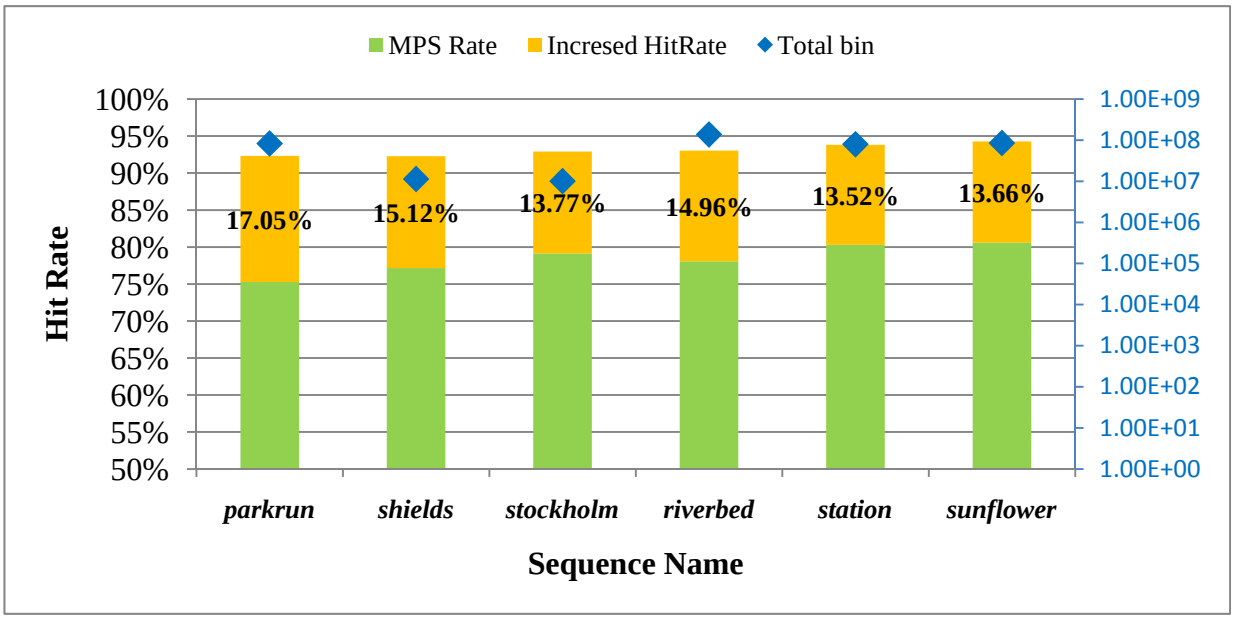


Figure 65. Hit rate of prediction process for HD sequence

Table 16 Hit rate of prediction process for HD sequence

Sequence Name	Total bin	MPS bin	Hit bin	MPS Rate	Proposed HIT Rate	Increased Hit Rate
parkrun	83001685	62474532	76624869	75.27%	92.32%	17.05%
shields	11317592	8733340	10444611	77.17%	92.29%	15.12%
stockholm	10089721	7984117	9373943	79.13%	92.91%	13.77%
riverbed	137403314	107280400	127833147	78.08%	93.03%	14.96%
station	79634578	63939062	74707361	80.29%	93.81%	13.52%
sunflower	85137642	68626372	80255375	80.61%	94.27%	13.66%

*(*Below simulations are tested by JM 16.1[2] with 4:2:0 color format,QP_{BP}: 28, GOP: IBPBP…, Max video bit-rate and frame rate of 30 fps.)*

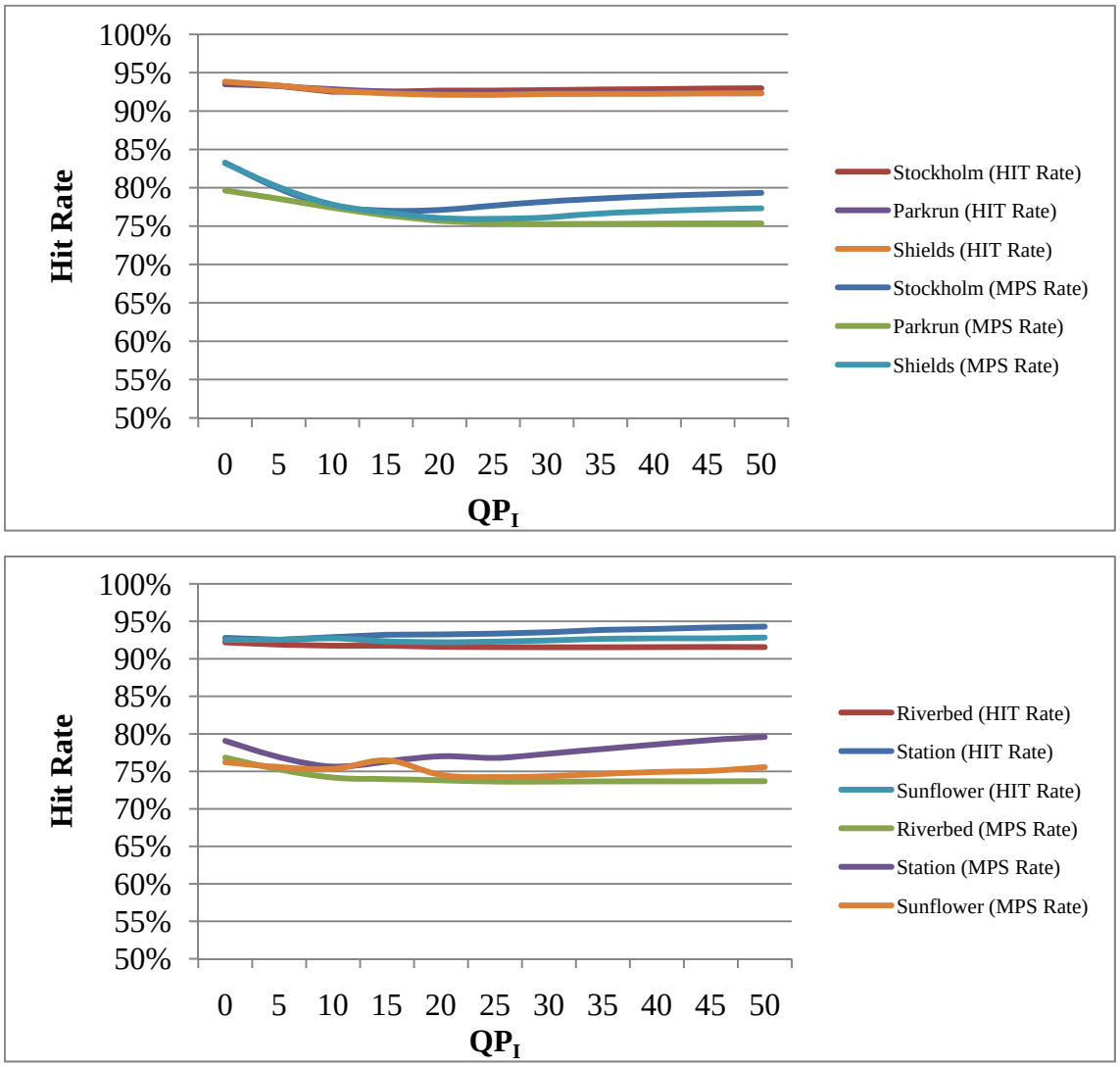


Figure 66. Hit rate of prediction process for variable QP_I

Table 17 Hit rate of prediction process for variable QP_I

	HD720p						HD1080p					
QP	**stockholm**		**parkrun**		**stockholm**		**parkrun**		**stockholm**		**parkrun**	
0	83.27%	93.72%	79.64%	93.48%	83.21%	93.83%	76.22%	92.53%	79.06%	92.81%	76.82%	92.20%
10	79.91%	93.28%	78.56%	93.29%	80.09%	93.32%	75.57%	92.56%	76.88%	92.55%	75.25%	91.90%
15	77.65%	92.51%	77.39%	92.85%	77.82%	92.64%	75.32%	92.74%	75.61%	92.90%	74.16%	91.75%
20	77.02%	92.49%	76.39%	92.55%	76.77%	92.29%	76.47%	92.34%	76.30%	93.22%	73.96%	91.75%
25	77.12%	92.67%	75.67%	92.38%	76.05%	92.12%	74.52%	92.23%	77.01%	93.26%	73.80%	91.61%
30	77.67%	92.67%	75.37%	92.30%	75.94%	92.10%	74.23%	92.30%	76.79%	93.38%	73.62%	91.56%
35	78.21%	92.73%	75.27%	92.32%	76.14%	92.24%	74.35%	92.47%	77.36%	93.56%	73.61%	91.55%
40	78.61%	92.80%	75.29%	92.33%	76.65%	92.24%	74.67%	92.67%	77.99%	93.87%	73.65%	91.55%
45	78.90%	92.85%	75.33%	92.31%	76.95%	92.23%	74.92%	92.73%	78.59%	93.99%	73.66%	91.57%
50	79.15%	92.92%	75.35%	92.31%	77.17%	92.30%	75.07%	92.74%	79.20%	94.18%	73.66%	91.58%

*(*Below simulations are tested by JM 16.1[2] with 4:2:0 color format,QP_I: 28, GOP: IBPBP…, Max video bit-rate and frame rate of 30 fps.)*

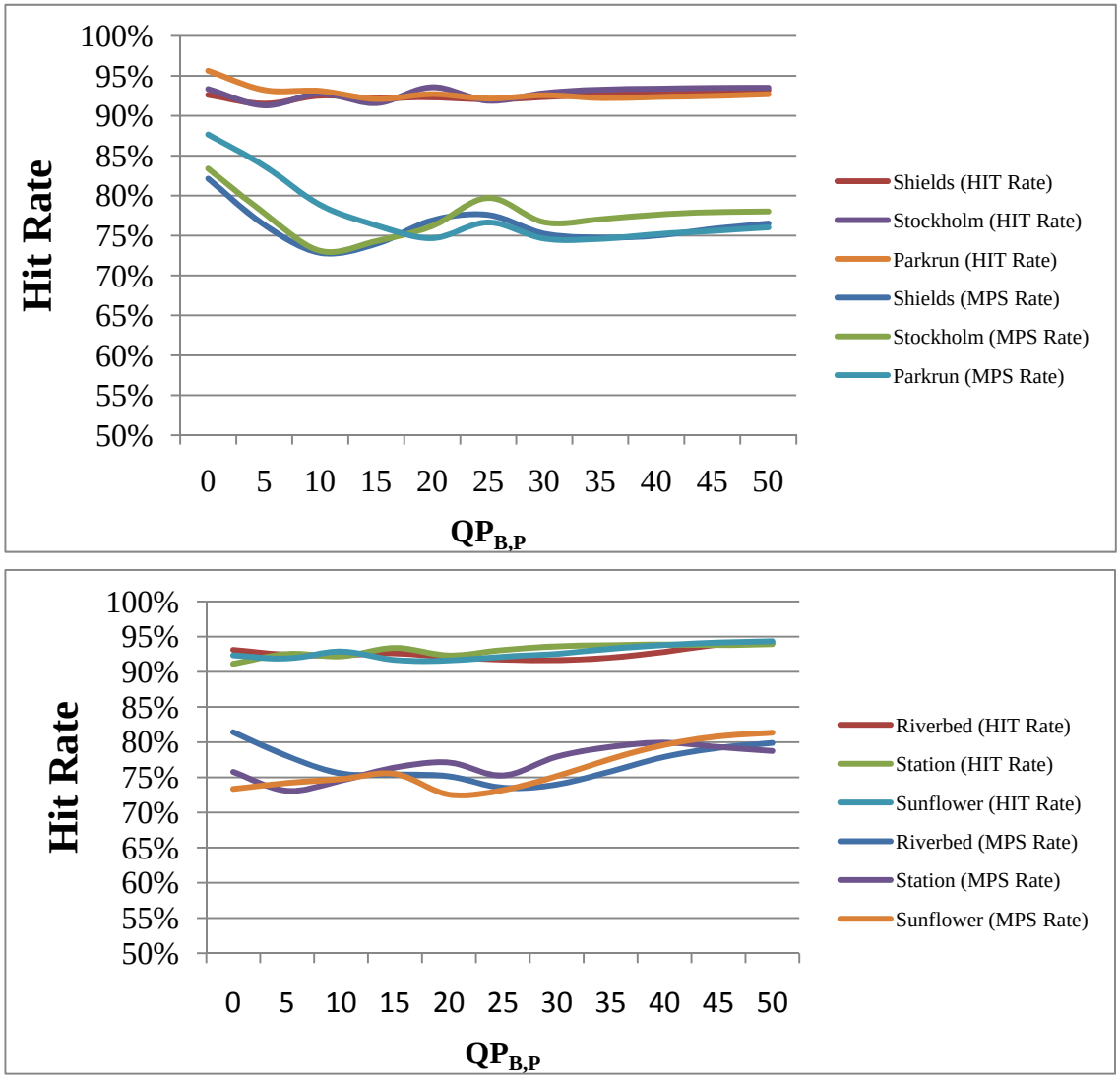


Figure 67. Hit rate of prediction process for variable $QP_{B,P}$

Table 18 Hit rate of prediction process for variable $QP_{B,P}$

	HD720p						HD1080p					
QP	**stockholm**		**parkrun**		**shields**		**sunflower**		**station**		**riverbed**	
0	83.38%	93.34%	87.65%	95.64%	82.12%	92.61%	73.34%	92.34%	75.77%	91.13%	81.42%	93.11%
10	77.81%	91.30%	83.72%	93.24%	76.34%	91.51%	74.19%	91.89%	73.08%	92.53%	78.02%	92.42%
15	73.08%	92.77%	78.83%	93.12%	72.83%	92.52%	74.75%	92.88%	74.54%	92.20%	75.54%	92.38%
20	74.30%	91.59%	76.26%	92.10%	73.95%	92.17%	75.48%	91.68%	76.40%	93.38%	75.41%	92.62%
25	76.19%	93.57%	74.66%	92.72%	76.90%	92.32%	72.54%	91.61%	77.11%	92.33%	75.14%	92.08%
30	79.71%	91.88%	76.64%	92.16%	77.58%	92.04%	73.16%	92.14%	75.26%	93.09%	73.52%	91.74%
35	76.66%	92.84%	74.61%	92.58%	75.23%	92.33%	75.17%	92.54%	77.93%	93.63%	73.97%	91.64%
40	77.04%	93.27%	74.57%	92.23%	74.74%	92.67%	77.58%	93.27%	79.33%	93.77%	75.84%	92.01%
45	77.62%	93.39%	75.16%	92.34%	74.99%	92.78%	79.63%	93.79%	79.94%	93.87%	77.91%	92.83%
50	77.92%	93.47%	75.57%	92.49%	75.82%	93.00%	80.81%	94.14%	79.31%	93.77%	79.18%	93.85%

5.2 Memory System Verification

In the other works, we reduce the storage and test in full-HD sequences. We count the range distribution of mvd in average 30 frames and show our comparison with traditional in Figure 68. And then, we get reduction rate from 58.01% to 75.13%. Although, the riverbed sequence has low reduction rate, it doesn't have large mvd to be stored actually. It means we can reduce bandwidth requirement or system buffer utilities efficiently. Besides, we also analyze the relationship between memory bandwidth requirement and internal SRAM size in Figure 69 and Figure 70. By the way, we put simulation results of other resolutions in Appendix C.

*(*Below simulations are tested by JM 16.1[2] with 4:2:0 color format,QP: 28, GOP: IBPBP…, Max video bit-rate and frame rate of 30 fps.)*

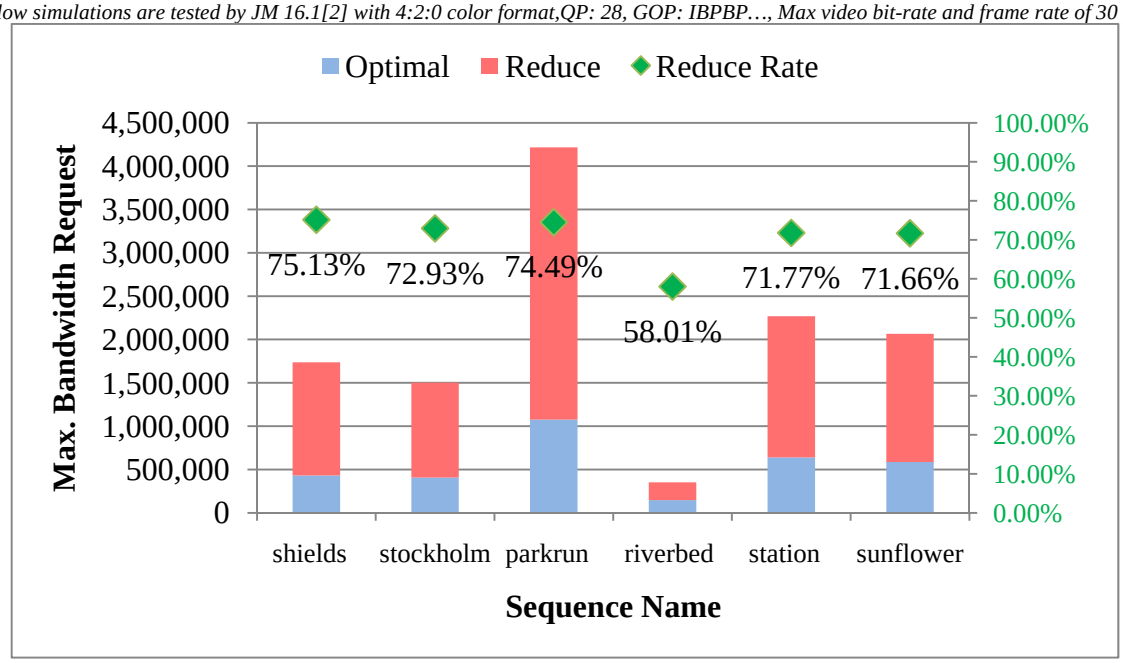


Figure 68. Max. B.W. of memory system for HD sequences

Table 19 Max. B.W. of memory system for HD sequences

Sequence Name	Total MVd	hMVd (MVd<2)	fMVd (MVd>2)	BandWidth@30fps		Reduction Rate
				Traditional	Optimal	
shields	578866	522504	56362	1736598.00	431862.60	75.13%
stockholm	500732	429883	70849	1502196.00	406712.70	72.93%
parkrun	1405608	1250702	154906	4216824.00	1075723.80	74.49%
riverbed	117080	65599	51481	351240.00	147469.50	58.01%
station	756052	631556	124496	2268156.00	640375.20	71.77%
sunflower	688546	573641	114905	2065638.00	585485.10	71.66%

*(*Below simulations are tested by JM 16.1[2] with 4:2:0 color format,QP: 28, GOP: IBPBP…, Max video bit-rate and frame rate of 30 fps.)*

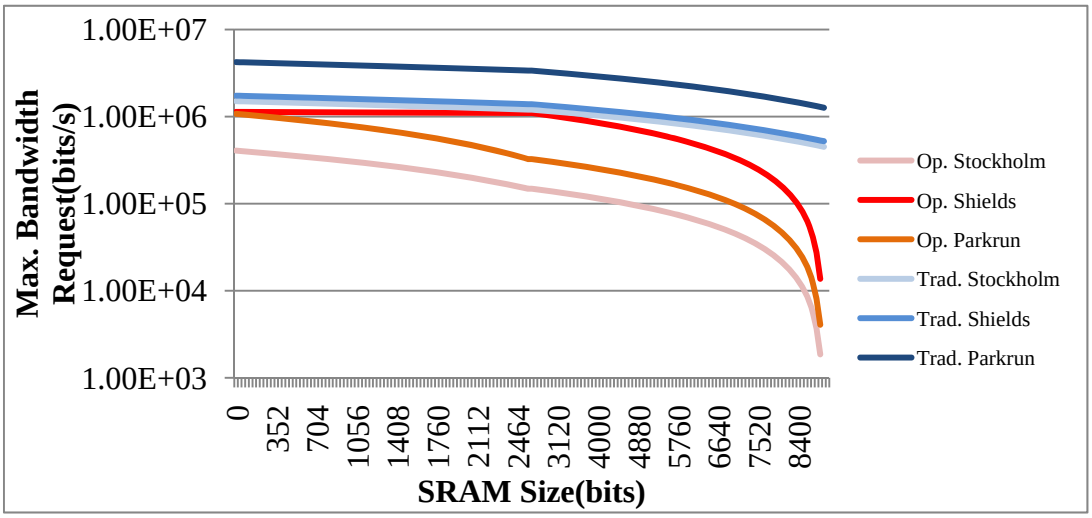


Figure 69. Max. B.W. of memory system for SRAM size for HD 720p sequences

Table 20 Max. B.W. of memory system for SRAM size for HD 720p sequences

SRAM size	HD 720p					
	Stockholm		Shields		Parkrun	
	Traditional	Optimal	Traditional	Optimal	Traditional	Optimal
0	1,502,196	406,713	1,736,598	1,131,076	4,216,824	1,075,724
160	1,483,419	390,592	1,714,891	1,128,962	4,164,114	1,028,822
320	1,464,641	374,471	1,693,183	1,126,848	4,111,403	981,921
480	1,445,864	358,351	1,671,476	1,124,735	4,058,693	935,020
640	1,427,086	342,230	1,649,768	1,122,621	4,005,983	888,119
800	1,408,309	326,110	1,628,061	1,120,508	3,953,273	841,217
960	1,389,531	309,989	1,606,353	1,118,394	3,900,562	794,316
1,120	1,370,754	293,868	1,584,646	1,116,281	3,847,852	747,415
1,280	1,351,976	277,748	1,562,938	1,114,167	3,795,142	700,513
1,440	1,333,199	261,627	1,541,231	1,112,053	3,742,431	653,612
1,600	1,314,422	245,507	1,519,523	1,109,940	3,689,721	606,711
1,760	1,295,644	229,386	1,497,816	1,107,826	3,637,011	559,809
1,920	1,276,867	213,265	1,476,108	1,105,713	3,584,300	512,908
2,080	1,258,089	197,145	1,454,401	1,103,599	3,531,590	466,007
2,240	1,239,312	181,024	1,432,693	1,101,486	3,478,880	419,105
2,400	1,220,534	164,904	1,410,986	1,099,372	3,426,170	372,204
2,560	1,201,757	148,783	1,389,278	1,097,258	3,373,459	325,303
2,960	1,154,813	139,484	1,335,010	1,028,680	3,241,683	304,971
3,360	1,107,870	130,185	1,280,741	960,101	3,109,908	284,640
3,760	1,060,926	120,886	1,226,472	891,522	2,978,132	264,308
4,160	1,013,982	111,587	1,172,204	822,944	2,846,356	243,977
4,560	967,039	102,288	1,117,935	754,365	2,714,580	223,646
4,960	920,095	92,989	1,063,666	685,787	2,582,805	203,314
5,360	873,151	83,690	1,009,398	617,208	2,451,029	182,983
5,760	826,208	74,391	955,129	548,629	2,319,253	162,651
6,160	779,264	65,093	900,860	480,051	2,187,477	142,320
6,560	732,321	55,794	846,592	411,472	2,055,702	121,988
6,960	685,377	46,495	792,323	342,893	1,923,926	101,657
7,360	638,433	37,196	738,054	274,315	1,792,150	81,326
7,760	591,490	27,897	683,785	205,736	1,660,374	60,994
8,160	544,546	18,598	629,517	137,157	1,528,599	40,663
8,560	497,602	9,299	575,248	68,579	1,396,823	20,331
8,960	450,659	0	520,979	0	1,265,047	0

*(*Below simulations are tested by JM 16.1[2] with 4:2:0 color format,QP: 28, GOP: IBPBP…, Max video bit-rate and frame rate of 30 fps.)*

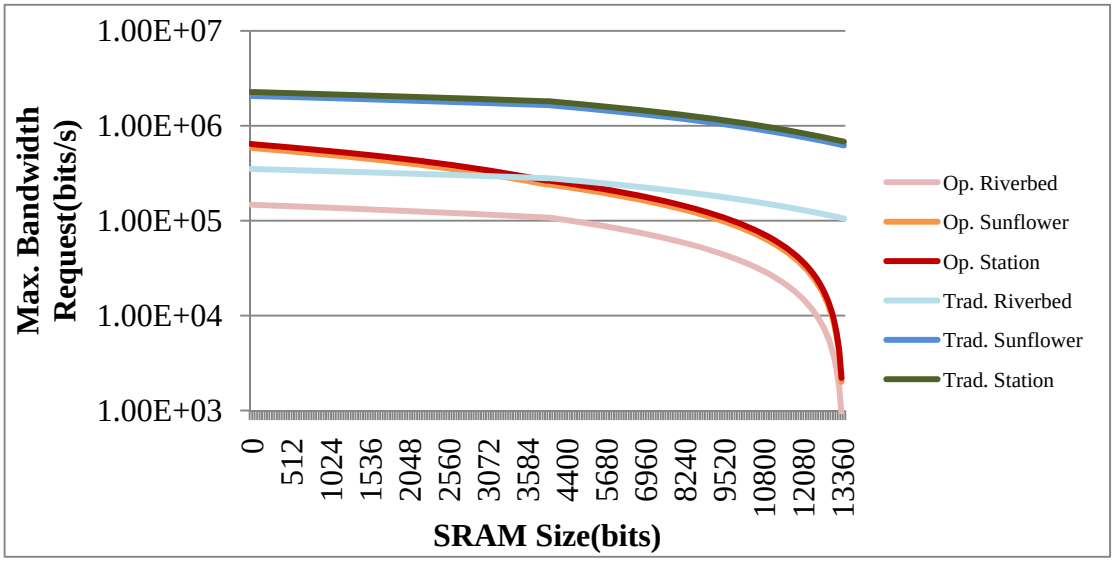


Figure 70. Max. B.W. of MEM. system for SRAM size for HD 1080p sequences

Table 21 Max. B.W. of MEM. system for SRAM size for HD 1080p sequences

SRAM size	HD 1080p					
	Riverbed		Sunflower		Station	
	Traditional	Optimal	Traditional	Optimal	Traditional	Optimal
0	351,240	147,470	2,065,638	585,485	2,268,156	642,887
256	346,557	144,846	2,038,096	562,539	2,237,914	617,692
512	341,874	142,222	2,010,554	539,594	2,207,672	592,496
768	337,190	139,598	1,983,012	516,648	2,177,430	567,301
1,024	332,507	136,974	1,955,471	493,703	2,147,188	542,106
1,280	327,824	134,350	1,927,929	470,757	2,116,946	516,911
1,536	323,141	131,726	1,900,387	447,811	2,086,704	491,715
1,792	318,458	129,102	1,872,845	424,866	2,056,461	466,520
2,048	313,774	126,478	1,845,303	401,920	2,026,219	441,325
2,304	309,091	123,854	1,817,761	378,974	1,995,977	416,130
2,560	304,408	121,230	1,790,220	356,029	1,965,735	390,934
2,816	299,725	118,606	1,762,678	333,083	1,935,493	365,739
3,072	295,042	115,982	1,735,136	310,137	1,905,251	340,544
3,328	290,358	113,358	1,707,594	287,192	1,875,009	315,348
3,584	285,675	110,734	1,680,052	264,246	1,844,767	290,153
3,840	280,992	108,110	1,652,510	241,301	1,814,525	264,958
3,840	280,992	108,110	1,652,510	241,301	1,814,525	264,958
4,480	269,284	100,903	1,583,656	225,214	1,738,920	247,294
5,120	257,576	93,695	1,514,801	209,127	1,663,314	229,630
5,760	245,868	86,488	1,445,947	193,040	1,587,709	211,966
6,400	234,160	79,281	1,377,092	176,954	1,512,104	194,302
7,040	222,452	72,073	1,308,237	160,867	1,436,499	176,639
7,680	210,744	64,866	1,239,383	144,780	1,360,894	158,975
8,320	199,036	57,659	1,170,528	128,694	1,285,288	141,311
8,960	187,328	50,451	1,101,674	112,607	1,209,683	123,647
9,600	175,620	43,244	1,032,819	96,520	1,134,078	105,983
10,240	163,912	36,037	963,964	80,434	1,058,473	88,319
10,880	152,204	28,829	895,110	64,347	982,868	70,655
11,520	140,496	21,622	826,255	48,260	907,262	52,992
12,160	128,788	14,415	757,401	32,173	831,657	35,328
12,800	117,080	7,207	688,546	16,087	756,052	17,664
13,440	105,372	0	619,691	0	680,447	0

5.3 Hardware Architecture Verification

The synthesis results of proposed architecture and the performance comparison with previous works are shown in Table 22. [7] and [8] use the same algorithm and get an increment throughput significantly by the method of branch selection hardware structure. But, they may get large hardware cost and still deal with SESO difficultly. Conversely, [5][6] use the high occurrence rate of MPS bin. [5] produces two bins per cycle (BPC) at continuous MPS bin and gets 0.86 BPC. And, [6] provides a SE predictor to really deal with SESO and has 71.4% of hit rate for overall bin switching.

The RTL simulation result shows that the proposed design can decode 0.93 BPC in average with 17k gate counts and 3,360 bits SRAM. However, we can achieve Level 5.0 MP in our estimation. Max. thoughput of the proposed design is 239.4 Mbins/s and Maximum working frequency is 232.5 MHz. In the other hand, we optimize our memory system for getting neighbor information, and we only need 0.868 cycle delay for each macro block. By the way, we put simulation results of other resolutions in Appendix D.

Table 22 Comparison of the proposed design and previous works

		ISCAS 10' [8]	ISCAS 09' [7]	ISCAS 08'[6]	CSVT 09' [5]	Proposed
Spec.		1920x1088@30fps	1920x1088@30fps	1920x1088@25fps	1920x1088@30fps	**1920x1088@30fps**
Technology		UMC 90nm	UMC 90nm	N/A	TSMC 0.18 um	**UMC 90nm**
Mechanism		Parallel-based	Parallel-based	Prediction-based	Prediction-based	**Prediction-based**
Frequency	**w/o SE parser**	MAX:264 MHz	MAX:222MHz	N/A	105 MHz (MAX:140 MHz)	**150 MHz (MAX: 286.5MHz)**
	with SE parser	N/A	N/A		N/A	**MAX:232.5 MHz**
Gate Count	**w/o Context Model,**	N/A	N/A	N/A	34,955	**17,022 (19,549@232.5MHz)**
	with Context Model,	42,372	82,400	N/A	76,333	**24,407 (28,150@232.5MHz)**
	SE Parser	N/A	N/A	N/A	N/A	**1,699**
MEM. System	**Get Neighbor Data Delay**	N/A	N/A	N/A	N/A	**0.868 idle/MB (*idle : 1 cycle)**
	Context Model	Hybrid SRAM	Register File	N/A	SRAM (3,528 bits)	**SRAM (3,360 bits)**
Average Bins/Cycle		1.83	1.95 ~ 1.98	0.8333 (Hit Rate: 71.4%)	0.71(740x480@4Mb/s) 0.86(1920x1088@60Mb/s)	**0.9295 (Hit Rate: 91.57%)**
Maximum Throughput		483.1 (@ 264MHz)	410.0 (@ 222MHz)	N/A	120.4 (@140Mhz)	223.6 **(@232.5MHz)**

Chapter 6. *Conclusion and Future Works*

6.1 Conclusion

In this works, we apply single-bin engine efficiently to get acceptable throughput by our proposed high-accuracy prediction scheme. And, we implement a bin-trend-predicted CABAC decoder hardware architecture with the balance of low overhead and acceptable throughput on the system point of view and compare with other designs in Figure 71.

We provide a bin-trend predictor to speculate the bin which is MPS or LPS, and we can get more than 90% of hit rate in our simulation results. Besides, we also provide a self-controlled SE parser which can output SE type [i+1] by inputting SE type [i] and current bin. After that, we can use the predicted bin to break the relationship between bin-decoded and ctxIdx-calculated process. Therefore, we can get high hardware utility rate efficiently.

Furthermore, we also optimize memory system to reduce the overhead for system, and it reduces about 70% of information to be stored. And, we also decrease the latency for getting neighbor information to avoid unexpected stalls. Finally, we use only 17K gate counts with 3,360 bits two-port SRAM to achieve maximum 223.6 Mbins/s throughputs for real-time decoding full-HD sequences.

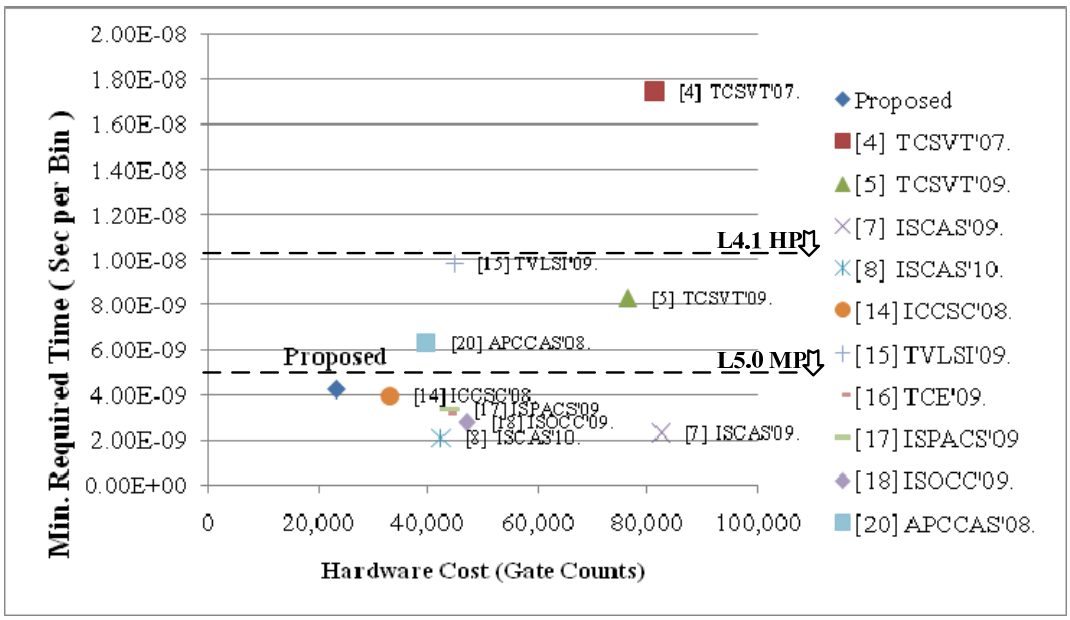


Figure 71. Comparison of the proposed design and previous works

6.2 Future Works

In order to achieve the QFHD or Ultra-HD videos, the throughput rate may not correspond to the requirement. We can estimate the specification of throughput in Table 23. If we should support higher resolution, the maximum bitstream would be upgraded. When we assume the average compression rate (CR) is 1.5, it means the maximum throughput should get at least 360 Mbins/s for QFHD. Furthermore, the ultra-HD will be supported by next standard – H.265/HEVC, and it has high probability to continue using CABAD for entropy coding. If we assume H.265/HEVC can save the bit-rate up to 50% compared with H.264/AVC, H.265/HEVC requires at least 720 Mbins/s.

Table 23 The specification [1] for QFHD and Ultra-HD at 30 fps

Resolution	**Frame size**	**FPS**	**MBits**	**CR**	**MBins**	**bin/cycle**	
						100 MHz	**333 MHz**
QFHD	4096x2048	30	240	1.5	360	3.6	1.08
Ultra-HD	8192x4096	30	*480	1.5	720	7.2	2.16

However, even if we overcome the miss penalty and idle problems as shown in (Eq. 10), the maximum throughput may equal to working frequency by single bin engine as (Eq. 13).

$$\text{Assume } (\text{MIssRate} \approx 0, \text{Idle times} \ll \text{Total Bin})$$

$$\text{Max. Throughput (Single bin)} = \frac{\text{Working Frequcy}}{(1 + \text{RegularModeRate} \cdot 0 + 0)} \quad \text{(Eq. 13)}$$

In other words, the frequency may at least work at 720 MHz for supporting Ultra-HD. Even though we don't care about the critical path of design, the system may not accept this working frequency. Therefore, we require an advantage technology to combine the advantage of prediction-base and parallel-base CABAC decoder to break the limit of throughput as (Eq. 15).

$$\text{Max. Throughput (Single bin)} = \text{Working Frequency} \times 1 \quad \text{(Eq. 14)}$$

$$\text{Max. Throughput (Multi bin)} = \text{Working Frequency} \times \text{N (N} > 1) \quad \text{(Eq. 15)}$$

Actually, parallel-based strategy may raise the miss rate and idle times, and the performance wouldn't be double as our estimation when integrating to system.

In Figure 72, we show the block diagram with prediction-based and parallel-based techniques. We apply a predictor which can predict how many MPS bin will be produced, and the decoder can support variable bin rate and high working frequency. Therefore, we separate the status of “11” to five sub-intervals, and we depend on sub-intervals to determine which engine will be chose. Besides, we prepare four kinds of context model and pre-calculate the context data for continually MPS bin, and critical path is no longer than LPS process. After that, we can guarantee to decode 1~4 bin per cycle by our prediction process, it can be used to parallel-based decoder actually.

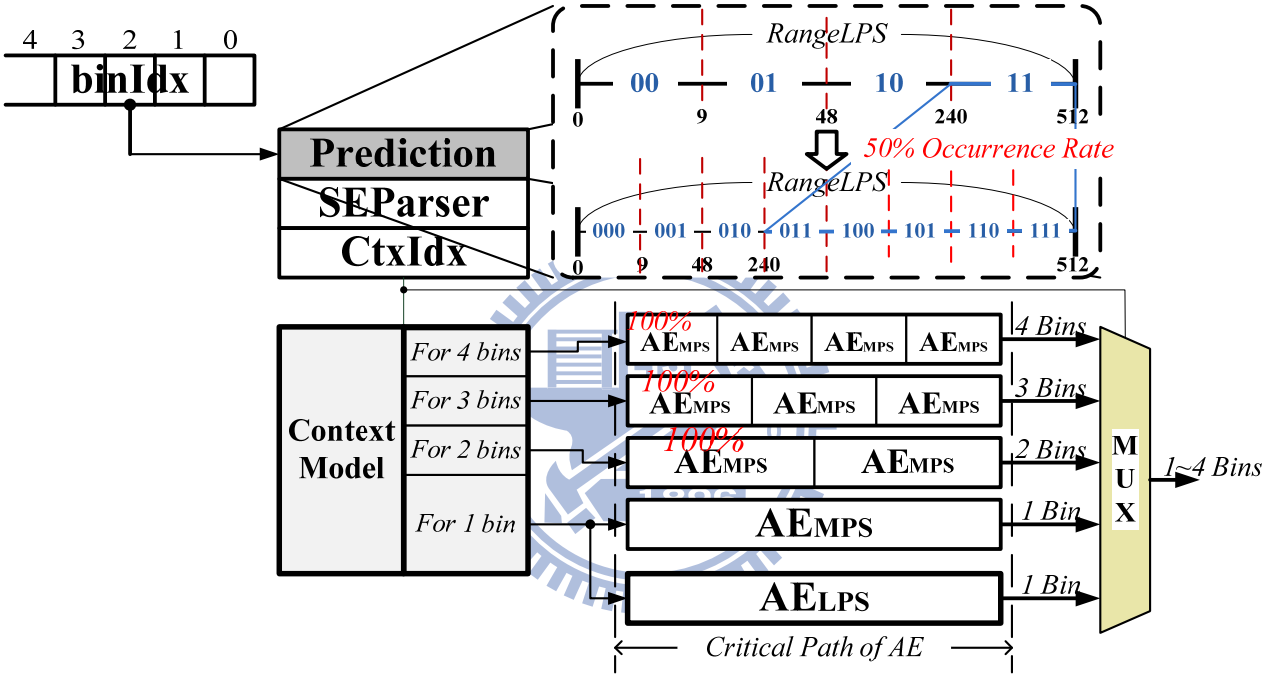


Figure 72. Combined prediction-based and parallel-based CABAC decoder

References

[1] Joint Video Team (JVT) of ISO/IEC MPEG&ITU-T VCEG, “Joint Draft ITU-T Rec. H.264 | ISO/IEC 14496-10/Amd.3 Scalable video coding,” Jul. 2007.

[2] Joint Video Team (JVT) Reference Software JM 16.

[3] Detlev Marpe, Heiko Schwarz and Thomas Wiegand, “Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard,” *IEEE Trans. on Circuits and System for Video Technology*, vol. 13, pp. 620-636, Jul. 2003.

[4] Yongseok Yi and In-Cheol Park, “High-Speed H.264/AVC CABAC Decoding,” *IEEE Trans. on Circuits and System for Video Technology*, vol. 17, pp. 490-494, Apr. 2007.

[5] Yao-Chang Yang and Jiun-In Guo, “High-Throughput H.264/AVC High-Profile CABAC Decoder for HDTV Applications,” *IEEE Trans. on Circuits and System for Video Technology*, vol. 19, pp. 1395-1399, Sep. 2009.

[6] Won-Hee Son and In-Cheol Park, “Prediction-based Real-time CABAC Decoder for High Definition H.264/AVC,” *IEEE International Symposium on Circuits and Systems*, pp. 33-36, May 2008.

[7] Pin-Chih Lin, Tzu-Der Chuang and Liang-Gee Chen, “A Branch Selection Multi-symbol High Throughput CABAC Decoder Architecture for H.264/AVC,” *IEEE International Symposium on Circuits and Systems*, pp. 365-368, May 2009.

[8] Yuan-Hsin Liao, Gwo-Long Li and Tian-Sheuan Chang, “A High Throughput VLSI Design with Hybrid Memory Architecture for H.264/AVC CABAC Decoder,” *IEEE International Symposium on Circuits and Systems*, pp. 2007–2010, May 2010.

[9] Yi-Hong Huang, “Context Adaptive Binary Arithmetic Decoder of H264/AVC for Digital TV application,” *Master thesis*, Hsinchu, Taiwan, Jul. 2005.

[10] Weiyi Xia, Xiaoliang Chen and Xiaofeng Lu, “Implementation strategies for CABAC decoder of H.264 for HD resolution video,” *IEEE International Symposium on Circuits and Systems*, pp. 596-600, Apr. 2010.

[11] Yan Zheng, Shibao Zheng, Zhonghua Huang and Ziliang Zhao, “A Time and Storage Optimized Hardware Design for Context-Based Adaptive Binary Arithmetic Decoding in H.264/AVC,” *IEEE International Conference on Multimedia & Expo*, pp. 1567-1570, Jul. 2007.

[12] Wei Yu and Yun He, “A high performance CABAC decoding architecture,” *IEEE Trans. on Consumer Electronics*, Vol. 51, No. 4, pp. 1352-1359, Nov. 2005.

[13] Chung-Hyo Kim and In-Cheol Park, “High speed decoding of context-based adaptive binary arithmetic codes using most probable symbol prediction,” *IEEE International Symposium on Circuits and Systems*, pp. 1707-1710, May 2006.

[14] Bing Shi, Wei Zheng, Hoang-Son Lee, Dong-Xiao Li and Ming Zhang, “Pipelined Architecture Design of H.264/AVC CABAC Real-Time Decoding,” *IEEE International Conference on Circuits & Systems for Communication8*, pp. 492-496, May 2008.

[15] Peng Zhang, Don Xie and Wen Gao, “Variable-Bin-Rate CABAC Engine for H.264/AVC High Definition Real-Time Decoding,” *IEEE Trans. on Very Large Scale Integration Systems*, vol. 17, pp. 417-426, Mar. 2009.

[16] Jian-Wen Chen and Youn-Long Lin, ”A high-performance hardwired CABAC decoder for ultra-high resolution video,” *IEEE Trans. on Consumer Electronics*, vol. 55, issue 3, pp. 1614-1622, Aug. 2009.

[17] Kai-Hsiang Chang and Youn-Long Lin, ”A very high throughput fully hardwired CABAC decoder,” *International Symposium on Intelligent Signal Processing and Communications Systems*, pp.200-203, Jan. 2009.

[18] Yu Hong, Peilin Liu, Hang Zhang, Zongyuan You, Dajiang Zhou and Goto S, “A 360Mbin/s CABAC decoder for H.264/AVC level 5.1 applications,” *International SoC Design Conference*, pp. 71-74, Nov. 2009.

[19] Xu Mei-hua, Cheng Yu-lan, Ran Feng, and Chen Zhang-jin, “Optimizing Design and FPGA Implementation for CABAC Decoder,” *High Density packaging and Microsystem Integration*, pp. 1-5, Jun. 2007.

[20] Yuan-Teng Chang, “A novel pipeline architecture for H.264/AVC CABAC decoder,” *High Density packaging and Microsystem Integration*, pp. 308-311, Nov. 2008.

Appendix A. System Specification

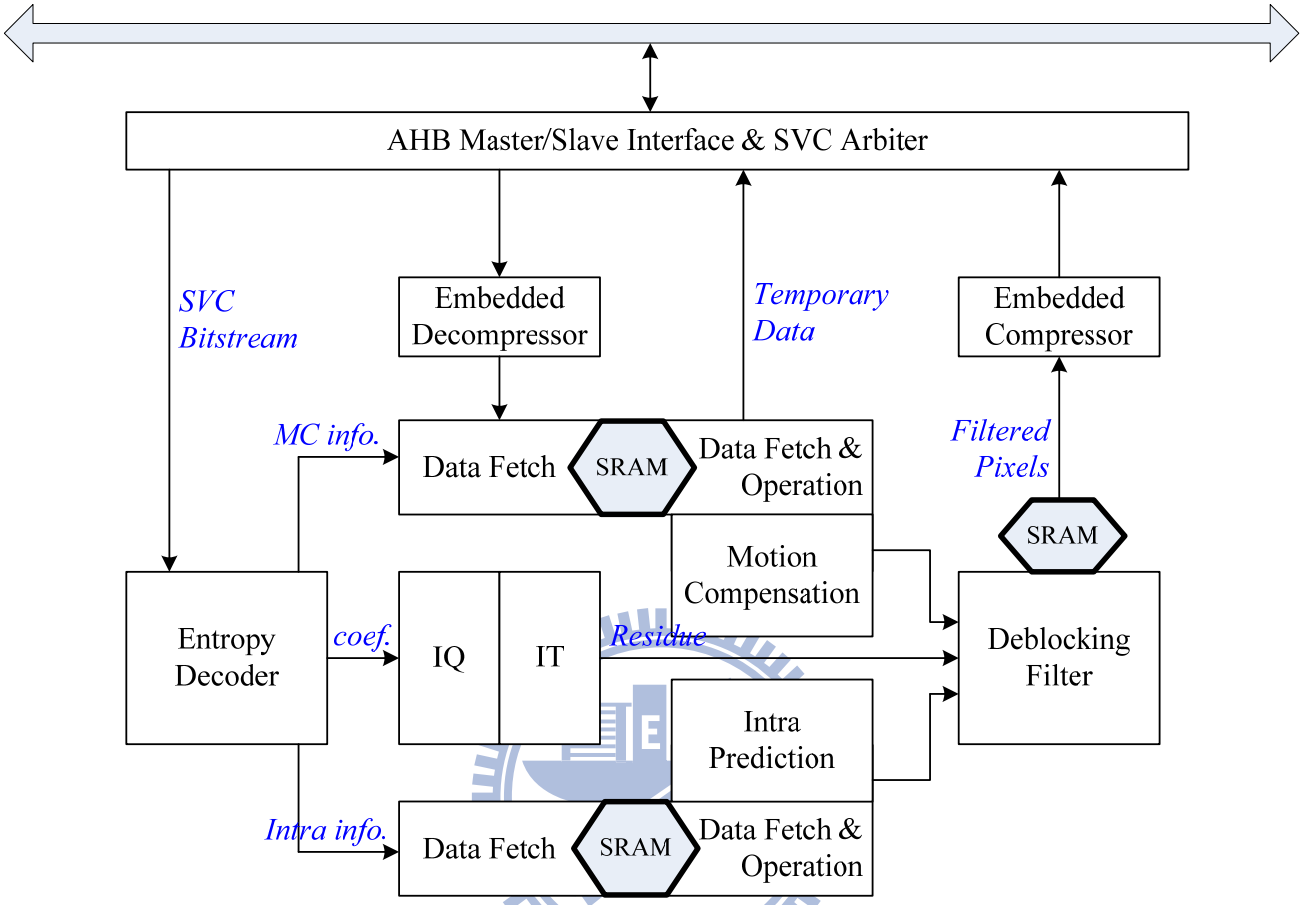


Figure 73. Block Diagram of Si2 H.264/SVC Decoder

Table 24. Our H.264/SVC system decoder specification

Si2 H.264/ SVC Decoder System			
Resolution:			
H.264/AVC		**SVC**	
HD1080, 30fps		HD720 – HD1080, 30fps (maximum resolution)	
		Others	Ex1: qcif – cif – 4cif – HD720
			Ex2: cif – 4cif – 16cif
			< 352800 MBs/s
Working Frequency:			
100 MHz		150 MHz	
External Memory and Bus:			
SDRAM	128Mb, 32-bit per entry	Bandwidth	32-bit/cycle

Appendix B. Simulation Result of Prediction Process

B.1 All Sequences of QCIF & CIF

QCIF (1/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 240 kbit/s*

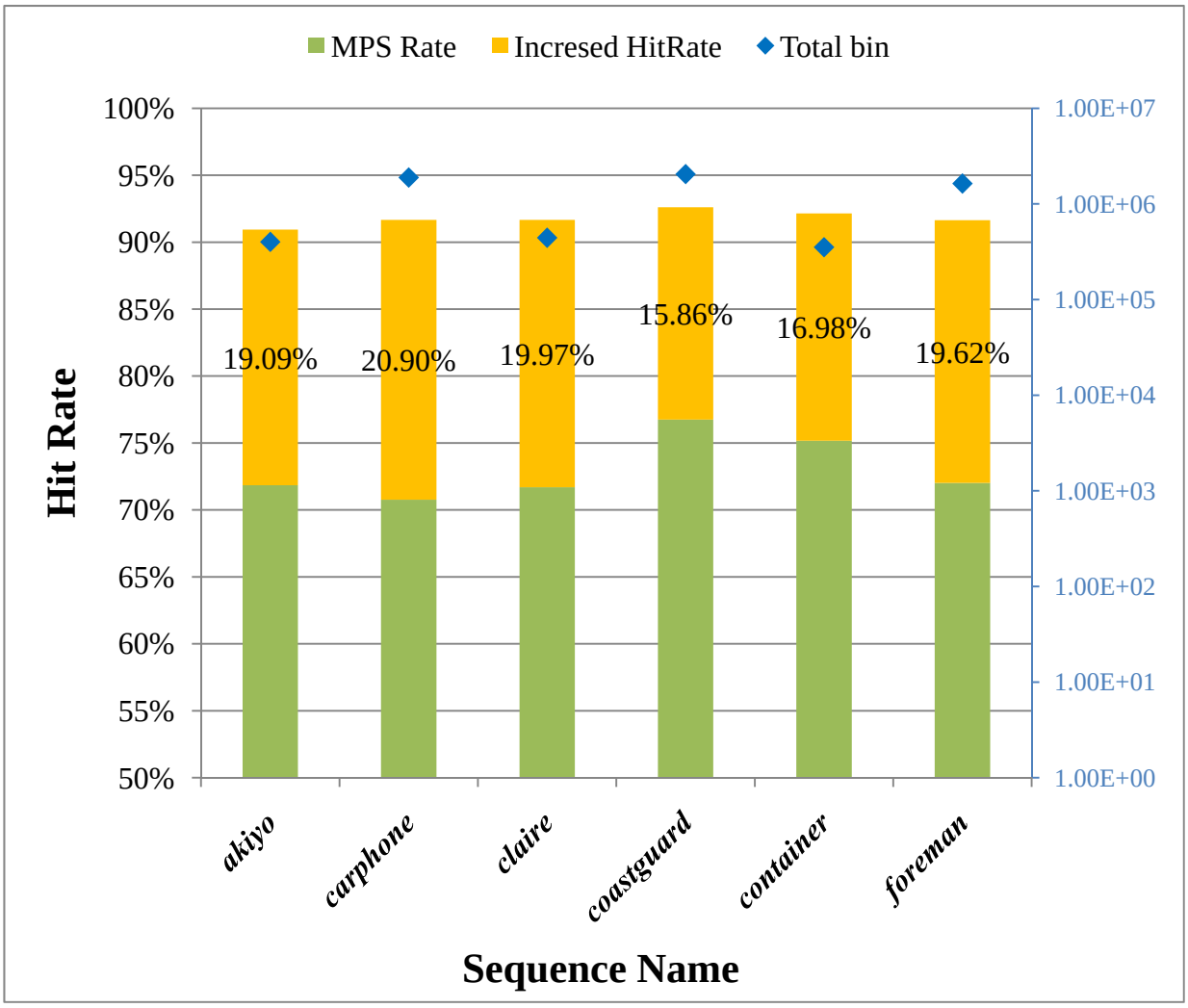


Figure 74. Hit rate of prediction process for QCIF sequences (1/4)

Table 25 Hit rate of prediction process for QCIF sequences (1/4)

Sequence Name	Total bin	MPS bin	Hit bin	MPS Rate	Proposed Hit Rate	Increased Hit Rate
akiyo	401188	288254	364849	71.85%	90.94%	19.09%
carphone	1888711	1336535	1731293	70.76%	91.67%	20.90%
claire	442900	317544	405973	71.70%	91.66%	19.97%
coastguard	2050659	1573894	1899053	76.75%	92.61%	15.86%
container	352983	265334	325270	75.17%	92.15%	16.98%
foreman	1632549	1175665	1496015	72.01%	91.64%	19.62%

QCIF (2/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 240 kbit/s*

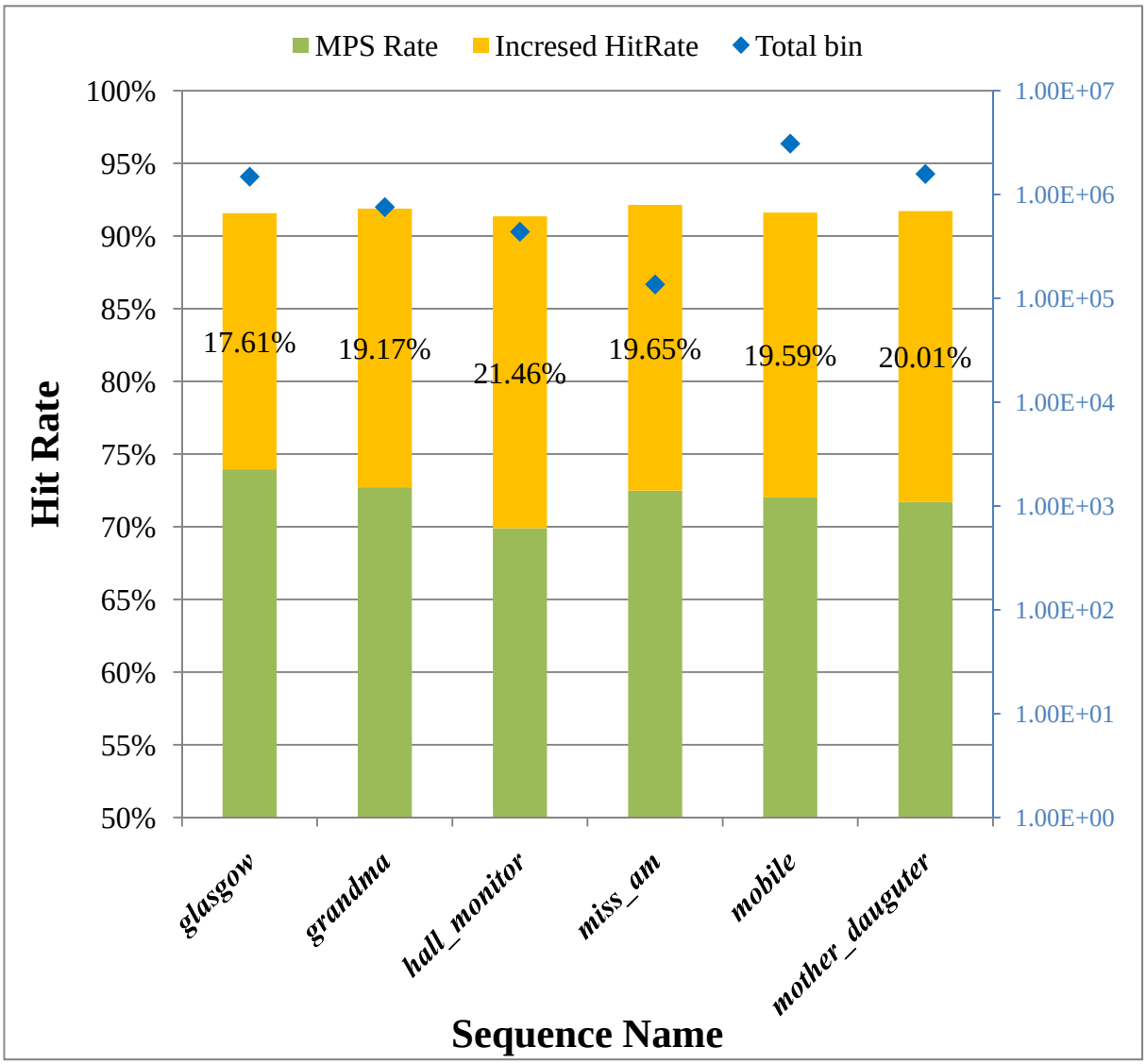


Figure 75. Hit rate of prediction process for QCIF sequences (2/4)

Table 26 Hit rate of prediction process for QCIF sequences (2/4)

Sequence Name	Total bin	MPS bin	Hit bin	MPS Rate	Proposed Hit Rate	Increased Hit Rate
glasgow	1483251	1096963	1483251	73.96%	91.57%	17.61%
grandma	756790	550260	756790	72.71%	91.88%	19.17%
hall_monitor	437924	306081	437924	69.89%	91.36%	21.46%
miss_am	136214	98737	136214	72.49%	92.14%	19.65%
mobile	3085784	2222294	3085784	72.02%	91.61%	19.59%
mother_ dauguter	1575658	1129898	1575658	71.71%	91.72%	20.01%

QCIF (3/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 240 kbit/s*

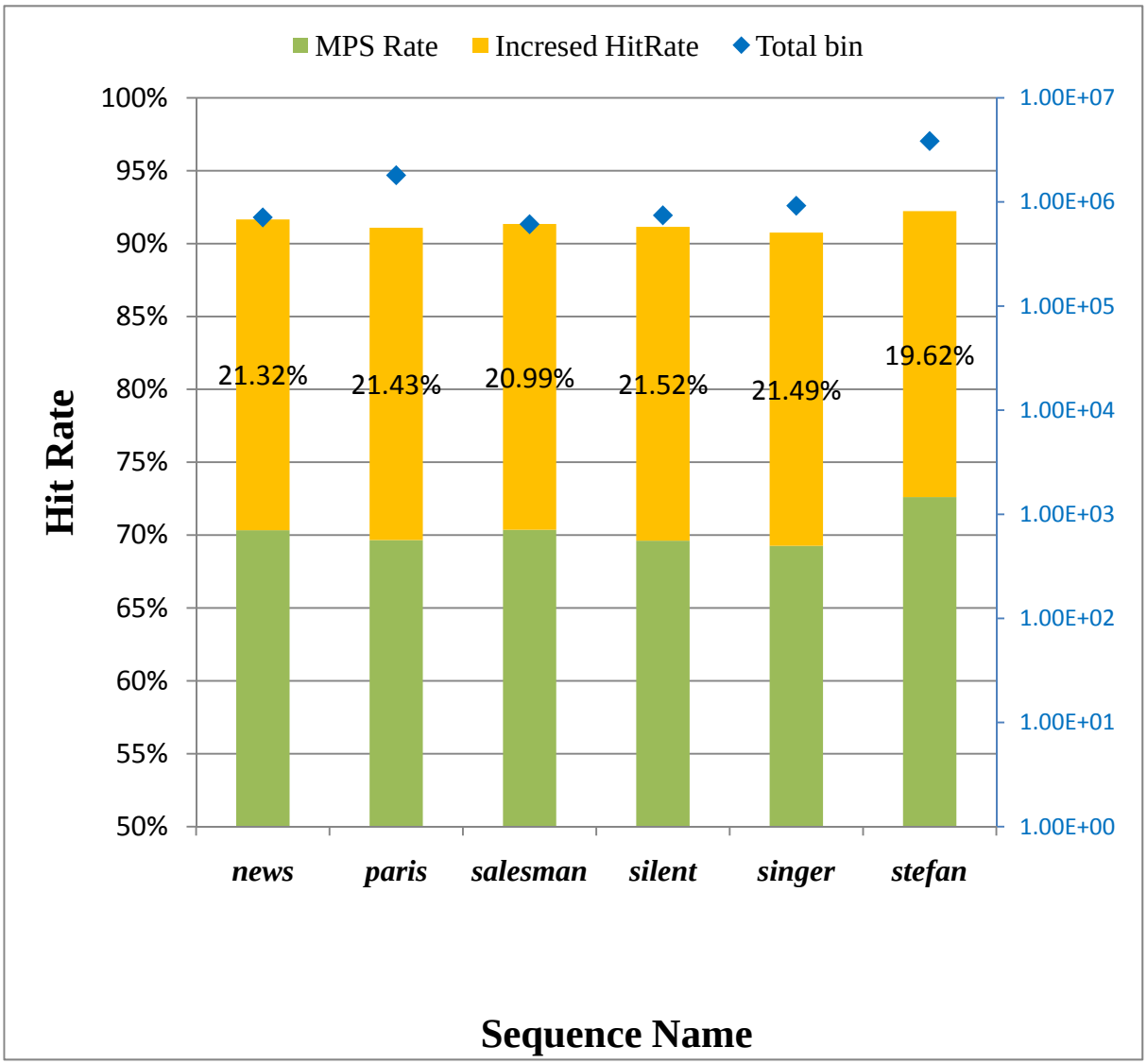


Figure 76. Hit rate of prediction process for QCIF sequences (3/4)

Table 27 Hit rate of prediction process for QCIF sequences (3/4)

Sequence Name	Total bin	MPS bin	Hit bin	MPS Rate	Proposed Hit Rate	Increased Hit Rate
news	711037	500118	651711	70.34%	91.66%	21.32%
paris	1806577	1258482	1645573	69.66%	91.09%	21.43%
salesman	611919	430568	558983	70.36%	91.35%	20.99%
silent	744340	518223	678414	69.62%	91.14%	21.52%
singer	919938	637155	834846	69.26%	90.75%	21.49%
stefan	3845883	2792411	3546934	72.61%	92.23%	19.62%

QCIF (4/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 240 kbit/s*

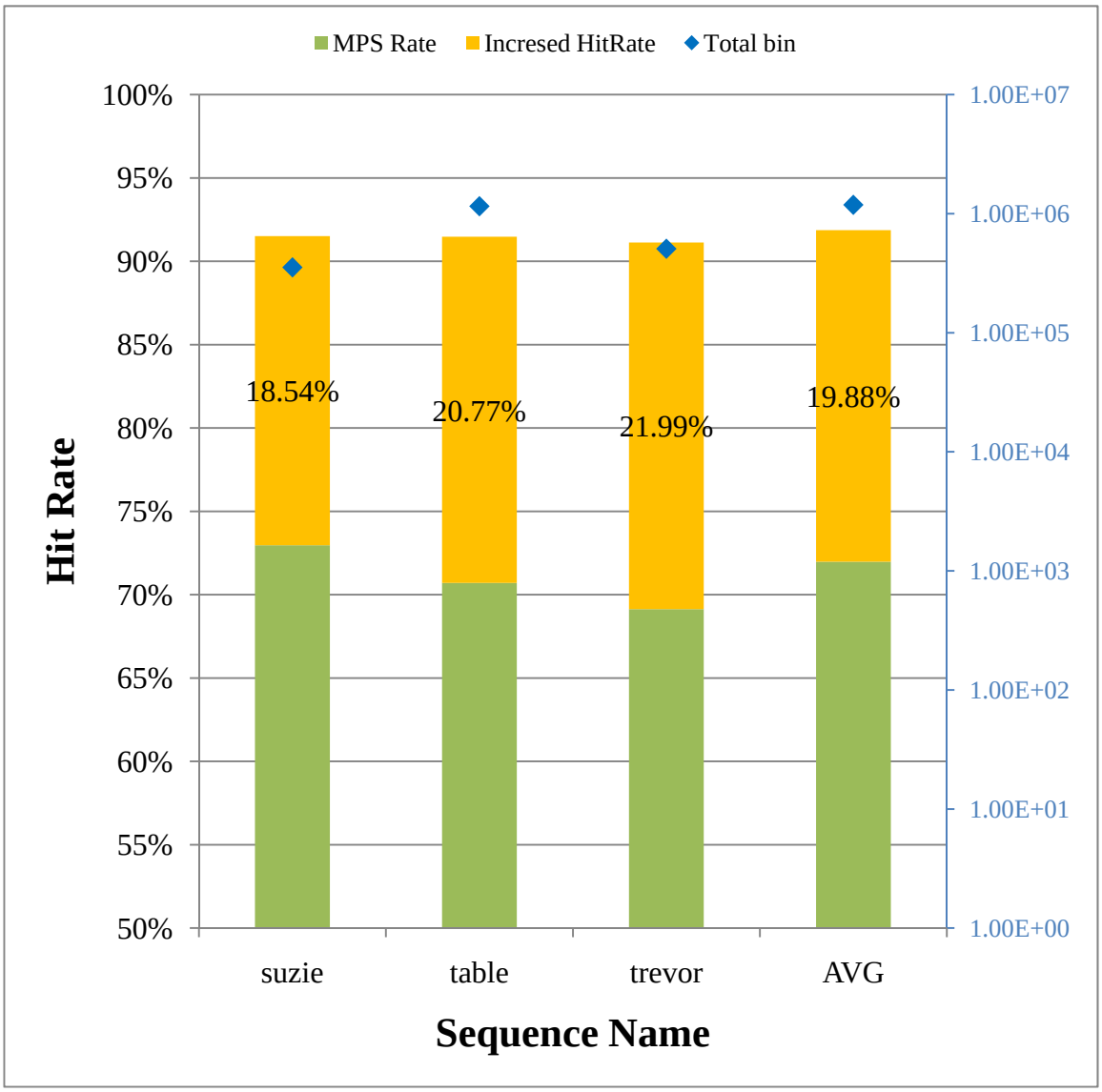


Figure 77. Hit rate of prediction process for QCIF sequences (4/4)

Table 28 Hit rate of prediction process for QCIF sequences (4/4)

Sequence Name	Total bin	MPS bin	Hit bin	MPS Rate	Proposed Hit Rate	Increased Hit Rate
suzie	354026	258335	323976	72.97%	91.51%	18.54%
table	1153592	815668	1055292	70.71%	91.48%	20.77%
trevor	507733	351005	462659	69.13%	91.12%	21.99%
AVG	1185698	853496.4	1087237	71.98%	91.70%	19.88%

CIF (1/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 960 kbit/s*

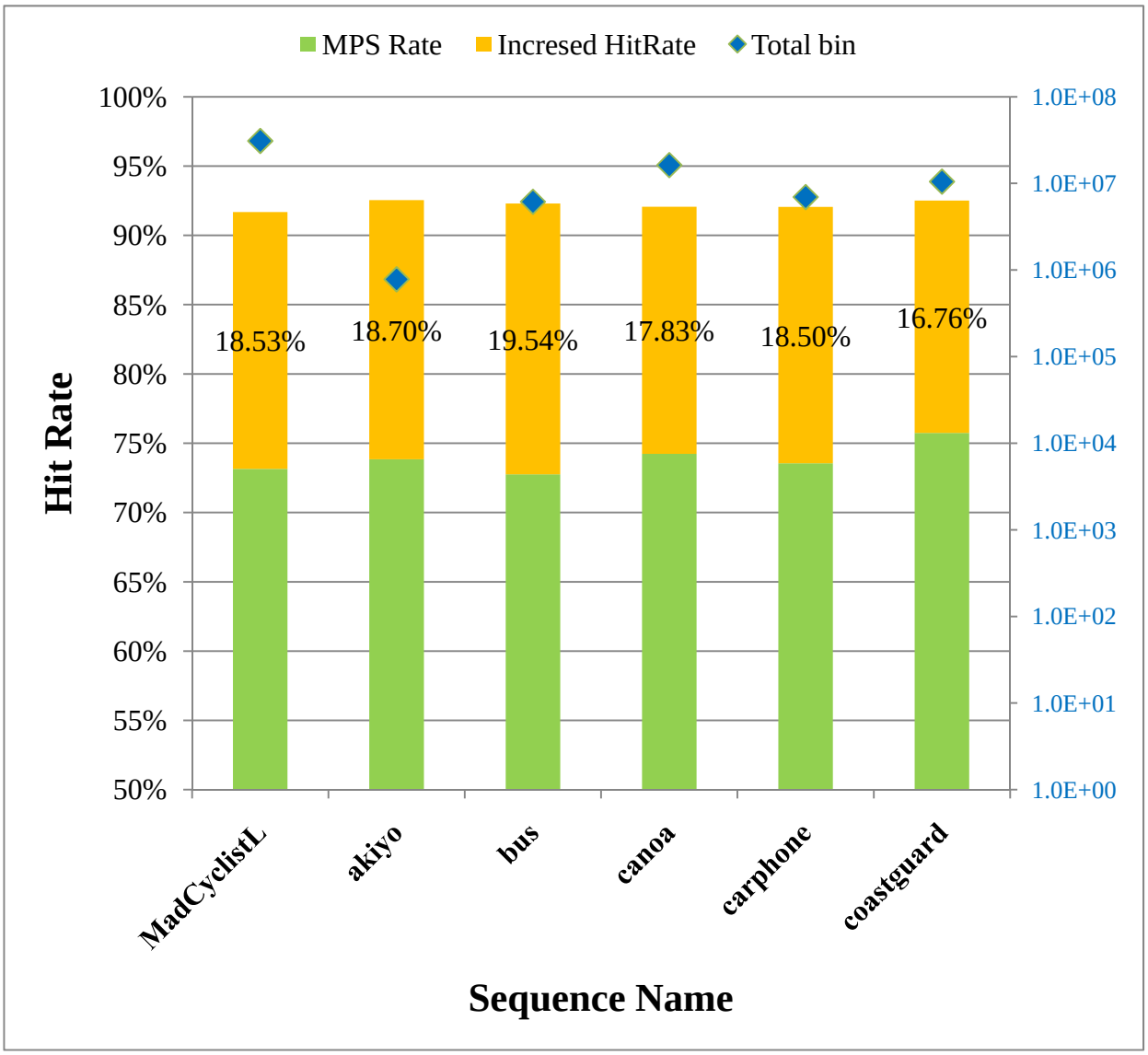


Figure 78. Hit rate of prediction process for CIF sequences (1/4)

Table 29 Hit rate of prediction process for CIF sequences (1/4)

Sequence Name	Total bin	MPS bin	Hit bin	MPS Rate	Proposed Hit Rate	Increased Hit Rate
MadCyclistL	30912062	22612471	28341503	73.15%	91.68%	18.53%
akiyo	781084	576774	722873	73.84%	92.55%	18.70%
bus	6138347	4465976	5665328	72.76%	92.29%	19.54%
canoa	16259037	12069746	14968955	74.23%	92.07%	17.83%
carphone	6955450	5115658	6402589	73.55%	92.05%	18.50%
coastguard	10485846	7942913	9700303	75.75%	92.51%	16.76%

CIF (2/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 960 kbit/s*

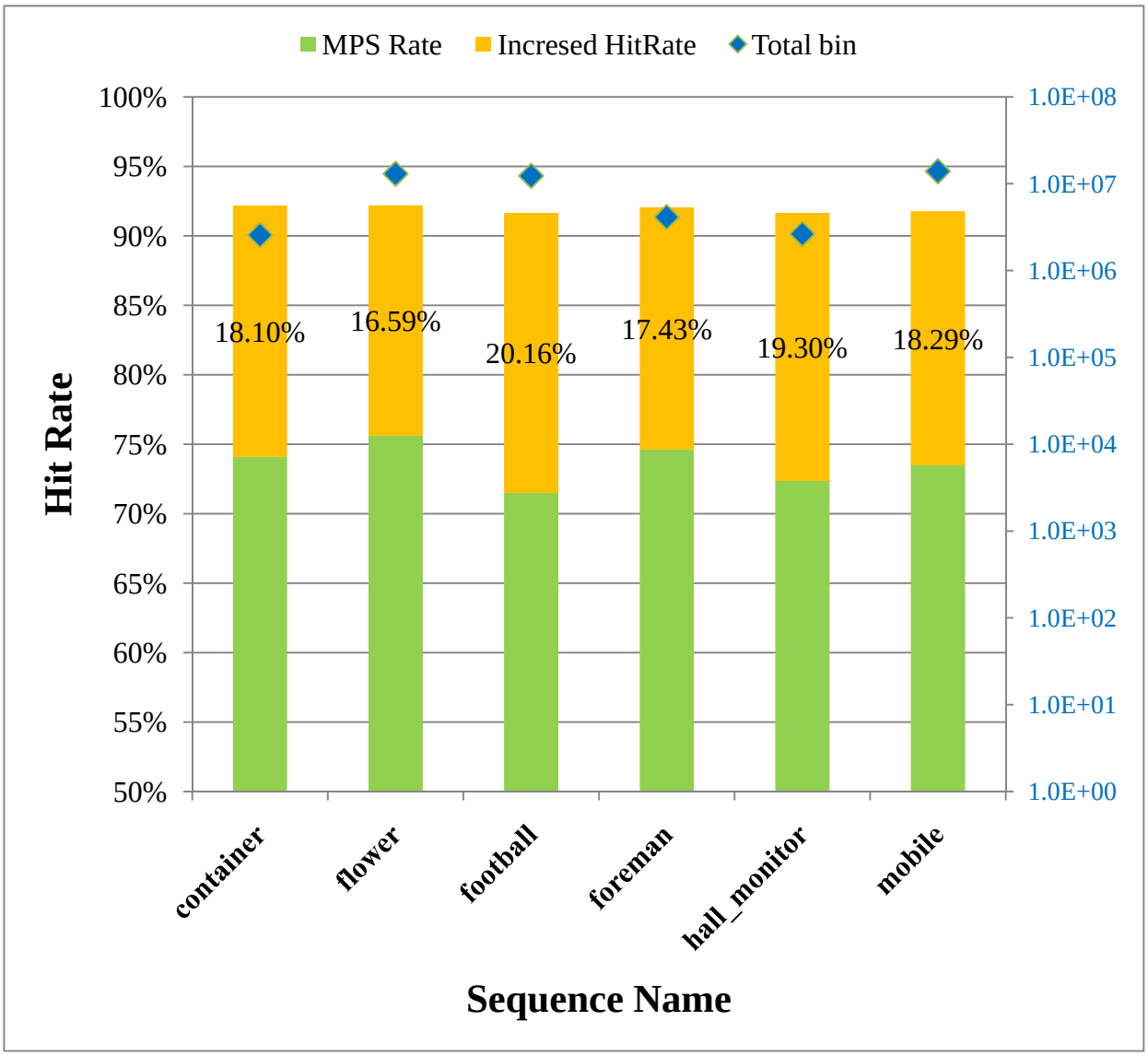


Figure 79. Hit rate of prediction process for CIF sequences (2/4)

Table 30 Hit rate of prediction process for CIF sequences (2/4)

Sequence Name	Total bin	MPS bin	Hit bin	MPS Rate	Proposed Hit Rate	Increased Hit Rate
container	2562245	1898159	2361878	74.08%	92.18%	18.10%
flower	13027346	9850217	12011075	75.61%	92.20%	16.59%
football	12317558	8806890	11289817	71.50%	91.66%	20.16%
foreman	4130911	3082408	3802422	74.62%	92.05%	17.43%
hall_monitor	2639871	1910072	2419528	72.35%	91.65%	19.30%
mobile	13913132	10223841	12768531	73.48%	91.77%	18.29%

CIF (3/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 960 kbit/s*

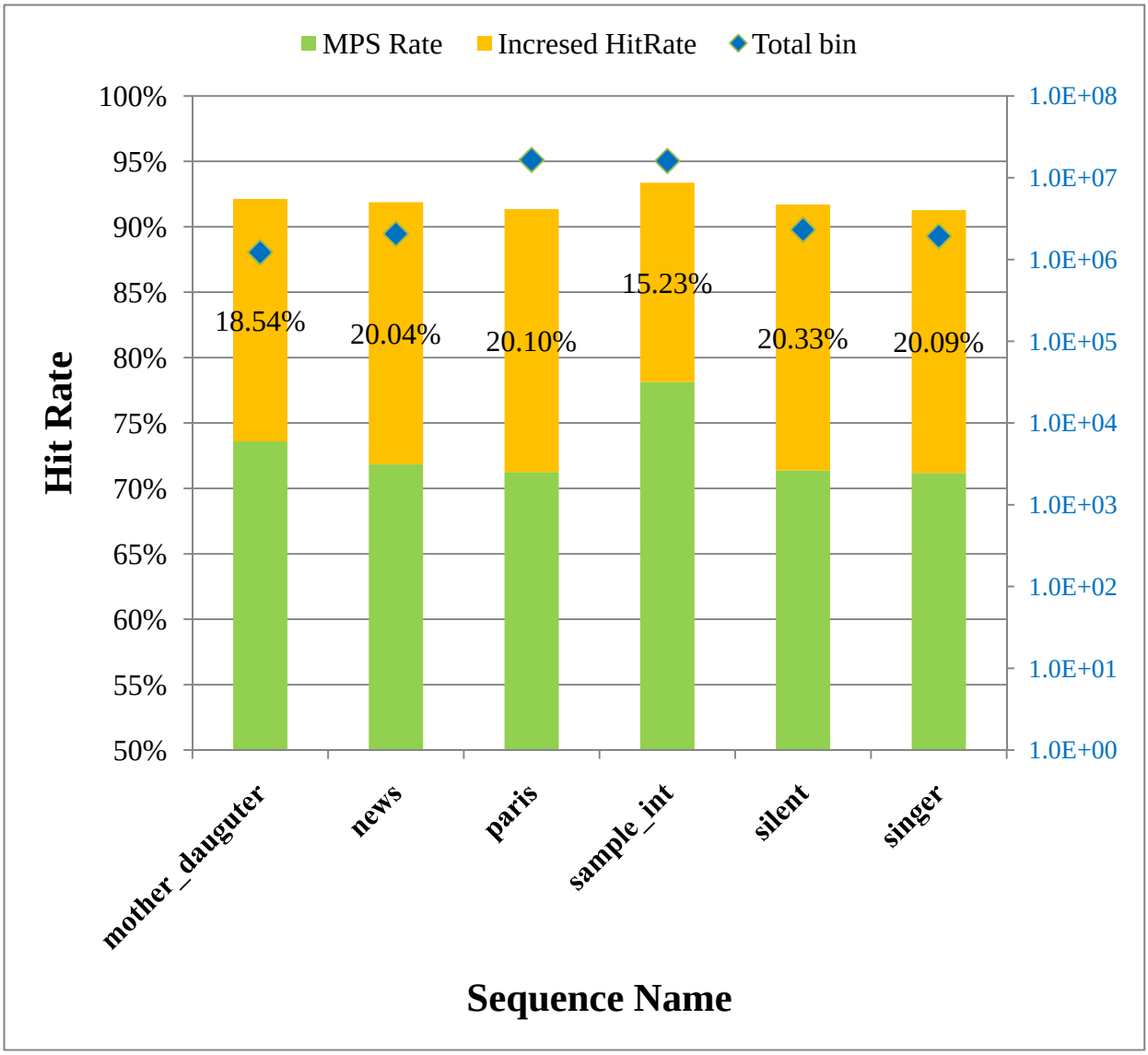


Figure 80. Hit rate of prediction process for CIF sequences (3/4)

Table 31 Hit rate of prediction process for CIF sequences (3/4)

Sequence Name	Total bin	MPS bin	Hit bin	MPS Rate	Proposed Hit Rate	Increased Hit Rate
mother_ dauguter	1220462	898149	1124388	73.59%	92.13%	18.54%
news	2058735	1479086	1891674	71.84%	91.89%	20.04%
paris	16494159	11752391	15067837	71.25%	91.35%	20.10%
sample_int	16021784	12517653	14958343	78.13%	93.36%	15.23%
silent	2313493	1651163	2121392	71.37%	91.70%	20.33%
singer	1928067	1372545	1759814	71.19%	91.27%	20.09%

CIF (4/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 960 kbit/s*

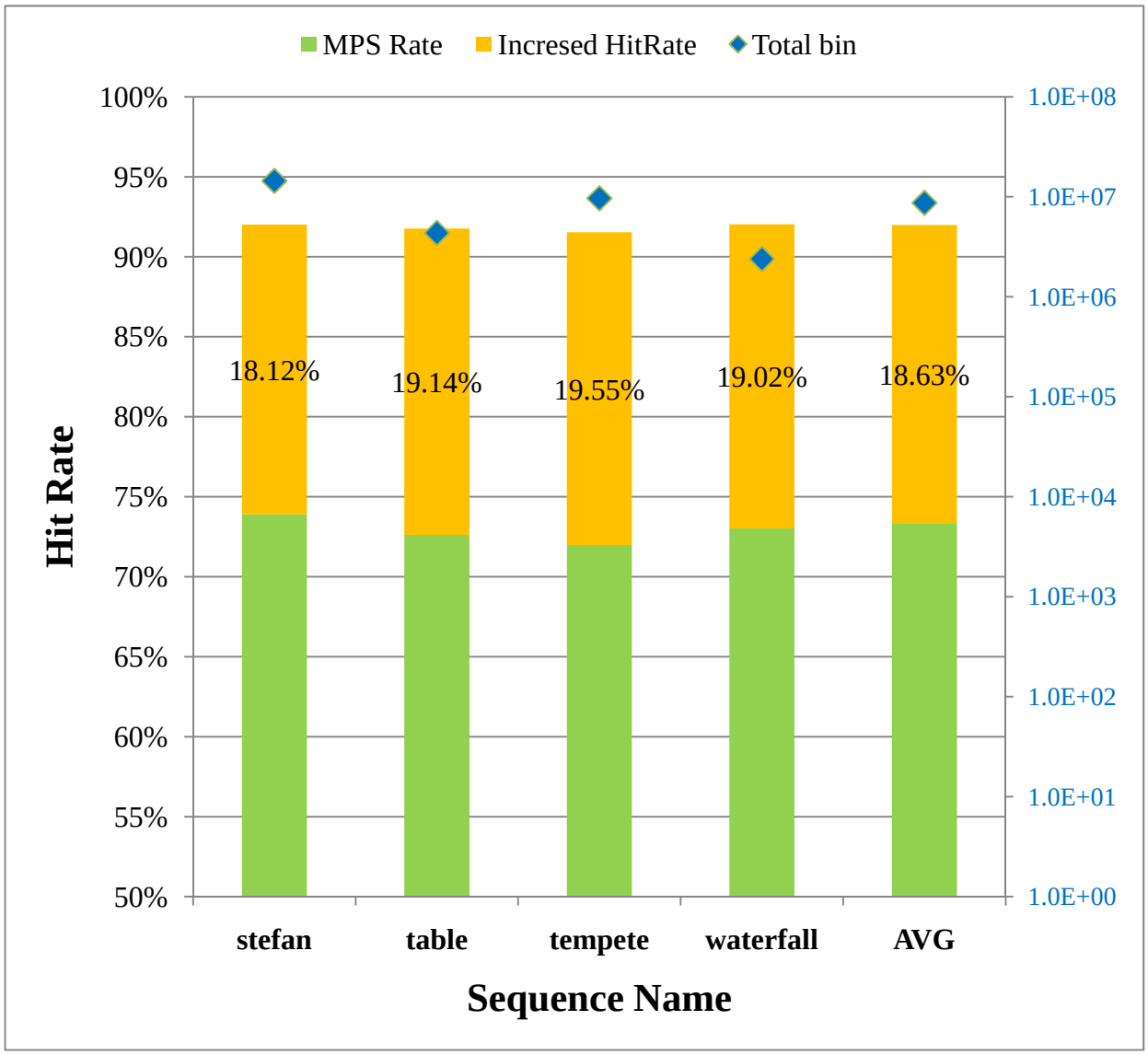


Figure 81. Hit rate of prediction process for CIF sequences (4/4)

Table 32 Hit rate of prediction process for CIF sequences (4/4)

Sequence Name	Total bin	MPS bin	Hit bin	MPS Rate	Proposed Hit Rate	Increased Hit Rate
stefan	14415583	10652610	13264112	73.90%	92.01%	18.12%
table	4344952	3155185	3986853	72.62%	91.76%	19.14%
tempete	9637761	6937195	8821796	71.98%	91.53%	19.55%
waterfall	2384654	1741221	2194676	73.02%	92.03%	19.02%
AVG	8679206	6396015	7983895	73.36%	91.99%	18.63%

B.2 All Various $QP_{I,B,P}$ of QCIF & CIF

QCIF *@ jm16.1[2] $QP_{B,P}$:28 GOP: IBPBP… BR: 240 kbit/s*

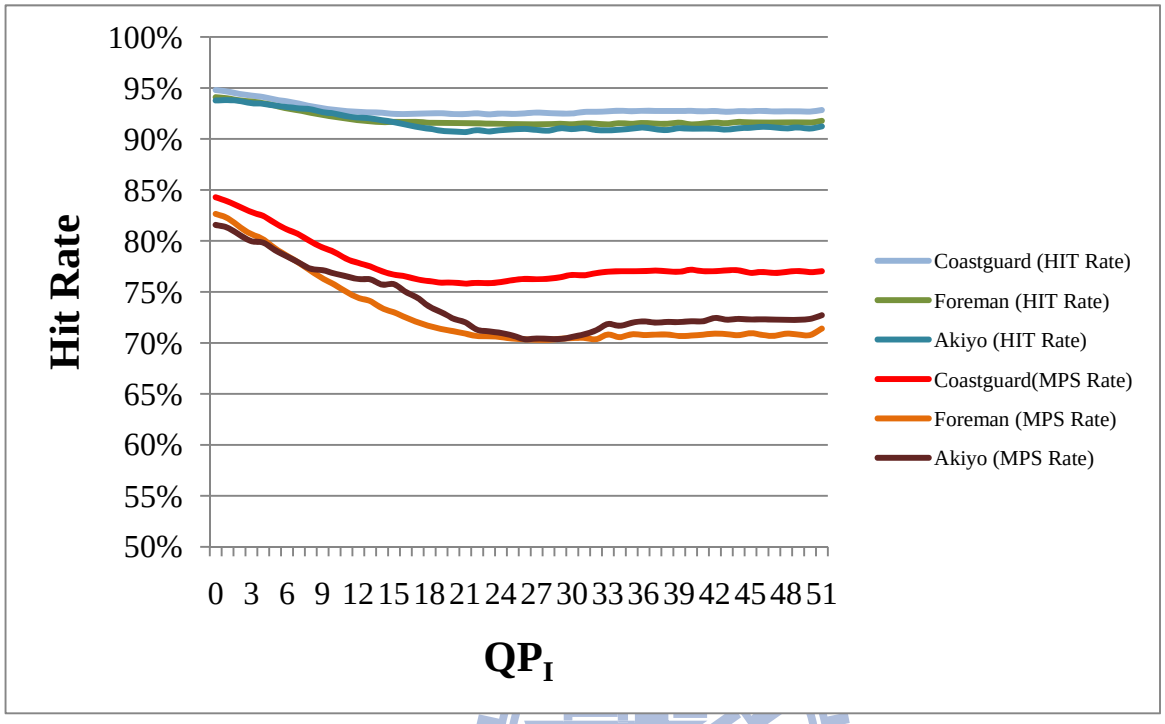


Figure 82. Hit rate of prediction process for various QP_I (QCIF)

CIF *@jm16.1[2] $QP_{B,P}$:28 GOP: IBPBP…BR: 960 kbit/s*

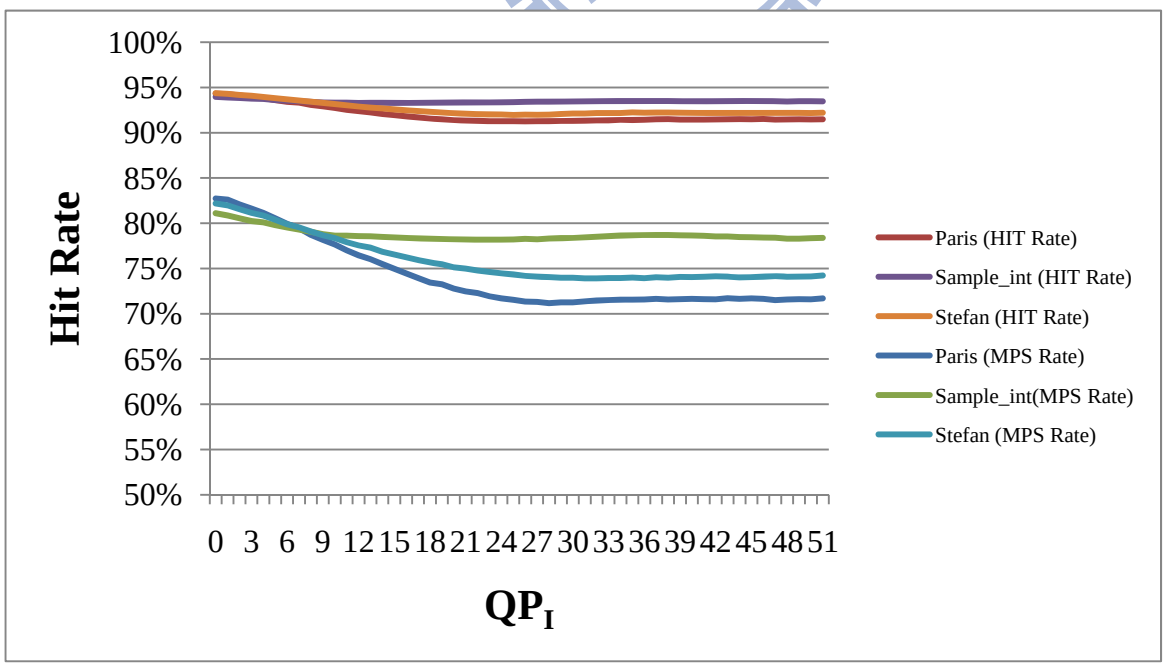


Figure 83. Hit rate of prediction process for various QP_I (CIF)

Table 33 Hit rate of prediction process for various QP_I

	QCIF						CIF					
Seq.	**coastguard**		**foreman**		**akiyo**		**sample_int**		**stefan**		**paris**	
QP_I	**MPS**	**Hit**	**MPS**	**Hit**	**MPS**	**Hit**	**MPS**	**Hit**	**MPS**	**Hit**	**MPS**	**Hit**
0	84.29%	94.81%	82.66%	94.09%	81.57%	93.78%	81.11%	93.97%	82.18%	94.37%	82.73%	94.31%
1	83.88%	94.66%	82.24%	93.97%	81.31%	93.81%	80.86%	93.90%	81.97%	94.30%	82.60%	94.28%
2	83.36%	94.43%	81.39%	93.78%	80.61%	93.74%	80.54%	93.84%	81.55%	94.18%	82.08%	94.12%
3	82.84%	94.26%	80.68%	93.66%	79.98%	93.51%	80.24%	93.77%	81.14%	94.09%	81.62%	93.97%
4	82.46%	94.12%	80.16%	93.50%	79.84%	93.46%	80.10%	93.73%	80.85%	93.98%	81.15%	93.84%
5	81.74%	93.87%	79.26%	93.26%	79.10%	93.27%	79.77%	93.61%	80.39%	93.82%	80.56%	93.62%
6	81.13%	93.70%	78.55%	92.99%	78.48%	93.15%	79.53%	93.54%	79.88%	93.67%	79.94%	93.42%
7	80.65%	93.48%	77.85%	92.80%	77.88%	93.00%	79.31%	93.49%	79.55%	93.57%	79.46%	93.32%
8	79.93%	93.21%	77.07%	92.56%	77.26%	92.91%	79.06%	93.42%	79.09%	93.42%	78.73%	93.08%
9	79.37%	93.00%	76.32%	92.37%	77.14%	92.63%	78.79%	93.35%	78.66%	93.31%	78.18%	92.92%
10	78.90%	92.87%	75.71%	92.13%	76.80%	92.49%	78.63%	93.34%	78.35%	93.19%	77.66%	92.74%
11	78.25%	92.74%	75.00%	92.01%	76.53%	92.23%	78.63%	93.34%	77.90%	93.02%	77.01%	92.53%
12	77.85%	92.65%	74.42%	91.84%	76.26%	92.11%	78.58%	93.30%	77.56%	92.89%	76.45%	92.39%
13	77.50%	92.62%	74.10%	91.72%	76.24%	92.02%	78.56%	93.31%	77.31%	92.78%	76.03%	92.26%
14	77.03%	92.57%	73.40%	91.65%	75.72%	91.83%	78.49%	93.30%	76.84%	92.69%	75.48%	92.08%
15	76.70%	92.46%	72.99%	91.68%	75.76%	91.64%	78.43%	93.30%	76.57%	92.57%	74.96%	91.95%
16	76.53%	92.45%	72.49%	91.67%	74.99%	91.42%	78.37%	93.29%	76.23%	92.50%	74.46%	91.83%
17	76.22%	92.48%	72.01%	91.68%	74.41%	91.15%	78.33%	93.32%	75.90%	92.37%	73.95%	91.70%
18	76.07%	92.51%	71.65%	91.58%	73.54%	91.01%	78.30%	93.30%	75.70%	92.32%	73.45%	91.56%
19	75.91%	92.53%	71.35%	91.58%	73.01%	90.79%	78.26%	93.33%	75.47%	92.26%	73.25%	91.50%
20	75.93%	92.44%	71.14%	91.56%	72.37%	90.73%	78.20%	93.34%	75.13%	92.17%	72.79%	91.39%
21	75.82%	92.45%	70.91%	91.55%	72.01%	90.67%	78.21%	93.34%	74.98%	92.09%	72.46%	91.35%
22	75.90%	92.51%	70.66%	91.54%	71.29%	90.86%	78.20%	93.34%	74.78%	92.04%	72.28%	91.32%
23	75.86%	92.42%	70.65%	91.50%	71.13%	90.73%	78.19%	93.34%	74.62%	92.04%	71.92%	91.28%
24	75.98%	92.50%	70.56%	91.50%	70.97%	90.86%	78.19%	93.37%	74.46%	92.01%	71.69%	91.28%
25	76.16%	92.46%	70.40%	91.43%	70.71%	90.93%	78.21%	93.39%	74.36%	91.97%	71.55%	91.27%
26	76.28%	92.52%	70.36%	91.45%	70.36%	90.98%	78.28%	93.41%	74.17%	92.00%	71.35%	91.27%
27	76.24%	92.59%	70.29%	91.42%	70.42%	90.88%	78.23%	93.43%	74.11%	91.98%	71.32%	91.28%
28	76.30%	92.55%	70.30%	91.47%	70.40%	90.81%	78.32%	93.44%	74.05%	92.00%	71.16%	91.27%
29	76.45%	92.50%	70.40%	91.49%	70.38%	91.05%	78.37%	93.44%	73.98%	92.07%	71.26%	91.32%
30	76.67%	92.50%	70.46%	91.44%	70.59%	90.97%	78.37%	93.45%	73.99%	92.13%	71.27%	91.32%
31	76.64%	92.66%	70.48%	91.53%	70.84%	91.06%	78.43%	93.47%	73.91%	92.13%	71.37%	91.34%
32	76.84%	92.67%	70.35%	91.50%	71.22%	90.87%	78.51%	93.50%	73.92%	92.17%	71.47%	91.37%
33	76.99%	92.71%	70.81%	91.42%	71.85%	90.84%	78.57%	93.49%	73.94%	92.18%	71.51%	91.37%
34	77.03%	92.77%	70.55%	91.54%	71.67%	90.90%	78.65%	93.51%	73.96%	92.18%	71.55%	91.43%
35	77.04%	92.72%	70.84%	91.49%	71.96%	91.01%	78.67%	93.50%	74.00%	92.25%	71.57%	91.40%
36	77.05%	92.77%	70.77%	91.57%	72.11%	91.12%	78.69%	93.51%	73.92%	92.22%	71.57%	91.43%
37	77.09%	92.75%	70.80%	91.50%	71.99%	90.94%	78.70%	93.50%	74.03%	92.23%	71.65%	91.49%
38	77.03%	92.74%	70.82%	91.49%	72.05%	90.88%	78.70%	93.51%	73.98%	92.22%	71.58%	91.52%
39	76.98%	92.74%	70.66%	91.60%	72.04%	91.05%	78.67%	93.50%	74.06%	92.22%	71.61%	91.46%
40	77.18%	92.76%	70.71%	91.42%	72.12%	91.00%	78.64%	93.49%	74.05%	92.22%	71.65%	91.46%
41	77.03%	92.71%	70.79%	91.50%	72.12%	91.02%	78.62%	93.49%	74.09%	92.18%	71.60%	91.45%
42	77.04%	92.74%	70.91%	91.60%	72.44%	91.00%	78.54%	93.50%	74.14%	92.15%	71.59%	91.48%
43	77.11%	92.67%	70.85%	91.55%	72.27%	90.91%	78.54%	93.50%	74.11%	92.20%	71.72%	91.49%
44	77.13%	92.73%	70.75%	91.67%	72.35%	91.04%	78.48%	93.49%	74.02%	92.21%	71.64%	91.50%
45	76.87%	92.72%	70.95%	91.60%	72.30%	91.09%	78.46%	93.52%	74.05%	92.19%	71.69%	91.48%
46	76.97%	92.76%	70.77%	91.61%	72.31%	91.19%	78.43%	93.50%	74.08%	92.18%	71.65%	91.52%
47	76.87%	92.69%	70.69%	91.63%	72.26%	91.13%	78.40%	93.49%	74.17%	92.20%	71.51%	91.45%
48	76.96%	92.72%	70.90%	91.59%	72.28%	91.03%	78.29%	93.46%	74.08%	92.19%	71.58%	91.47%
49	77.06%	92.72%	70.81%	91.64%	72.25%	91.12%	78.29%	93.49%	74.10%	92.20%	71.61%	91.50%
50	76.94%	92.69%	70.76%	91.60%	72.35%	91.01%	78.35%	93.48%	74.13%	92.19%	71.59%	91.47%
51	77.04%	92.84%	71.40%	91.78%	72.71%	91.23%	78.39%	93.47%	74.23%	92.22%	71.70%	91.49%

QCIF@ *jm16.1[2] QP_I:28 GOP: IBPBP… BR: 240 kbit/s*

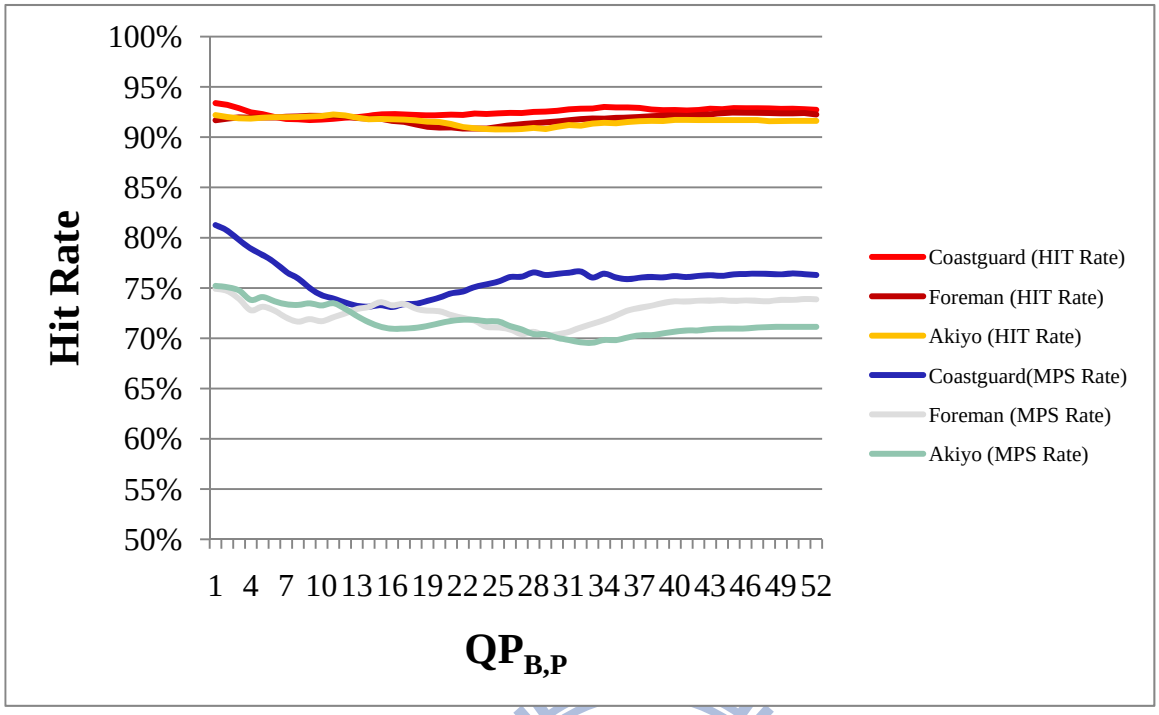


Figure 84. Hit rate of prediction process for various $QP_{B,P}$ (QCIF)

CIF@*jm16.1[2] QP_I:28 GOP: IBPBP…BR: 960 kbit/s*

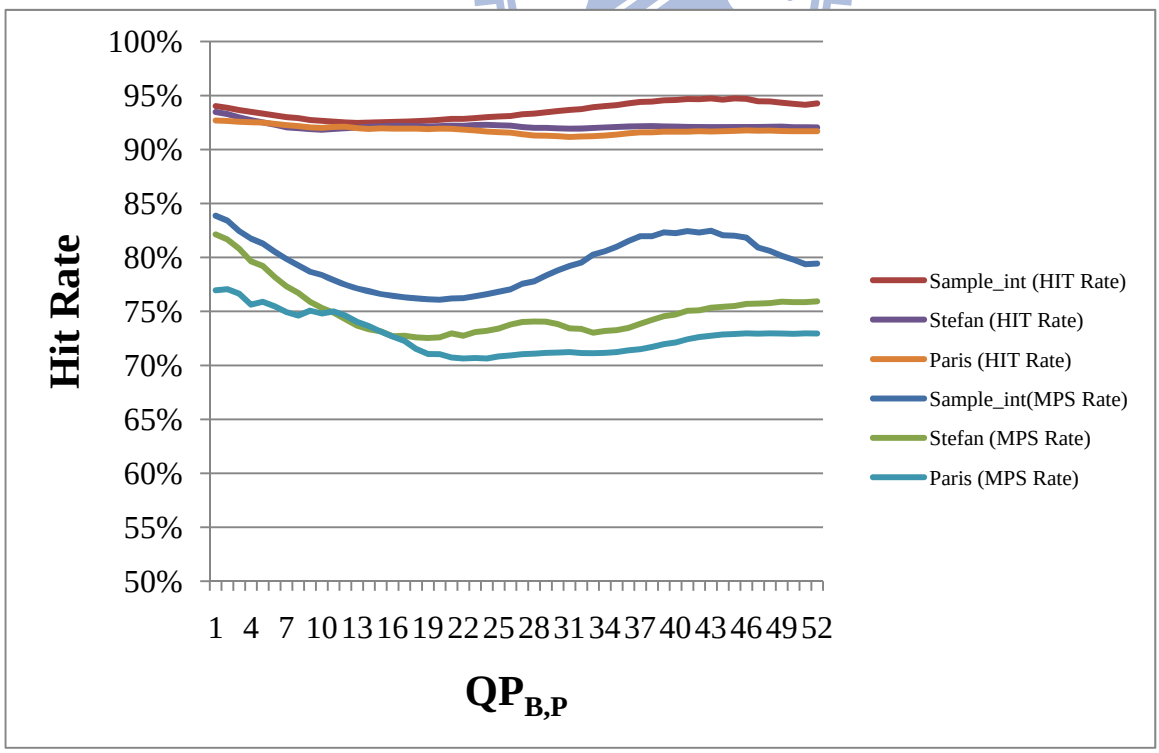


Figure 85. Hit rate of prediction process for various $QP_{B,P}$ (CIF)

Table 34 Hit rate of prediction process for various $QP_{B,P}$

	QCIF						CIF					
Seq.	**coastguard**		**foreman**		**akiyo**		**sample_int**		**stefan**		**paris**	
$QP_{B,P}$	**MPS**	**Hit**	**MPS**	**Hit**	**MPS**	**Hit**	**MPS**	**Hit**	**MPS**	**Hit**	**MPS**	**Hit**
0	81.26%	93.39%	74.92%	91.68%	75.21%	92.20%	83.86%	94.02%	82.13%	93.48%	76.96%	92.68%
1	80.72%	93.22%	74.70%	91.85%	75.08%	92.00%	83.40%	93.86%	81.65%	93.28%	77.06%	92.64%
2	79.76%	92.88%	73.91%	91.97%	74.75%	91.88%	82.45%	93.65%	80.81%	92.98%	76.65%	92.56%
3	78.93%	92.46%	72.78%	91.94%	73.81%	91.85%	81.75%	93.49%	79.64%	92.71%	75.62%	92.53%
4	78.27%	92.30%	73.13%	91.96%	74.10%	91.94%	81.28%	93.33%	79.21%	92.51%	75.89%	92.49%
5	77.57%	92.00%	72.74%	91.96%	73.68%	91.99%	80.53%	93.16%	78.18%	92.30%	75.47%	92.38%
6	76.60%	91.82%	72.06%	92.03%	73.38%	91.99%	79.86%	92.99%	77.32%	92.05%	74.93%	92.25%
7	75.96%	91.79%	71.65%	92.07%	73.31%	92.01%	79.26%	92.91%	76.72%	91.98%	74.61%	92.17%
8	75.00%	91.70%	71.90%	92.13%	73.47%	92.04%	78.68%	92.73%	75.89%	91.88%	75.07%	92.06%
9	74.30%	91.75%	71.69%	92.11%	73.26%	92.10%	78.38%	92.66%	75.32%	91.82%	74.80%	92.00%
10	73.95%	91.82%	72.08%	92.13%	73.48%	92.25%	77.90%	92.59%	74.87%	91.90%	74.99%	92.10%
11	73.56%	91.94%	72.47%	92.07%	72.93%	92.16%	77.46%	92.52%	74.27%	91.97%	74.61%	92.10%
12	73.23%	91.99%	72.90%	91.90%	72.21%	91.95%	77.12%	92.47%	73.68%	92.02%	74.04%	91.97%
13	73.15%	92.12%	73.10%	91.84%	71.60%	91.79%	76.87%	92.49%	73.34%	92.11%	73.63%	91.91%
14	73.30%	92.26%	73.58%	91.85%	71.15%	91.82%	76.61%	92.53%	73.14%	92.17%	73.14%	91.97%
15	73.11%	92.29%	73.27%	91.62%	70.94%	91.78%	76.45%	92.55%	72.70%	92.19%	72.67%	91.94%
16	73.40%	92.28%	73.41%	91.53%	70.96%	91.75%	76.30%	92.59%	72.74%	92.18%	72.27%	91.92%
17	73.43%	92.20%	72.95%	91.27%	71.04%	91.67%	76.21%	92.63%	72.59%	92.17%	71.52%	91.93%
18	73.72%	92.17%	72.73%	91.03%	71.22%	91.55%	76.13%	92.67%	72.54%	92.13%	71.06%	91.89%
19	74.05%	92.19%	72.69%	90.95%	71.49%	91.51%	76.08%	92.74%	72.59%	92.17%	71.04%	91.93%
20	74.47%	92.24%	72.30%	90.98%	71.72%	91.31%	76.20%	92.83%	72.97%	92.21%	70.72%	91.92%
21	74.65%	92.21%	72.01%	90.86%	71.84%	91.02%	76.23%	92.84%	72.75%	92.19%	70.63%	91.84%
22	75.10%	92.35%	71.75%	90.85%	71.83%	90.90%	76.40%	92.91%	73.08%	92.27%	70.69%	91.77%
23	75.37%	92.31%	71.13%	90.86%	71.69%	90.81%	76.60%	92.99%	73.21%	92.29%	70.63%	91.66%
24	75.64%	92.37%	71.07%	91.01%	71.67%	90.78%	76.82%	93.06%	73.41%	92.25%	70.83%	91.61%
25	76.09%	92.42%	70.87%	91.18%	71.22%	90.76%	77.04%	93.10%	73.78%	92.21%	70.91%	91.57%
26	76.13%	92.41%	70.41%	91.32%	70.87%	90.81%	77.56%	93.27%	74.02%	92.08%	71.03%	91.42%
27	76.55%	92.51%	70.64%	91.38%	70.41%	90.93%	77.79%	93.32%	74.06%	92.00%	71.08%	91.30%
28	76.30%	92.55%	70.30%	91.47%	70.40%	90.81%	78.32%	93.44%	74.05%	92.00%	71.16%	91.27%
29	76.40%	92.63%	70.39%	91.60%	70.04%	91.04%	78.80%	93.56%	73.82%	91.96%	71.19%	91.23%
30	76.52%	92.77%	70.63%	91.68%	69.83%	91.22%	79.21%	93.67%	73.43%	91.94%	71.24%	91.16%
31	76.65%	92.83%	71.07%	91.79%	69.59%	91.15%	79.53%	93.74%	73.39%	91.94%	71.15%	91.21%
32	76.04%	92.85%	71.43%	91.86%	69.55%	91.35%	80.26%	93.92%	73.02%	91.99%	71.13%	91.23%
33	76.43%	93.02%	71.79%	91.84%	69.83%	91.44%	80.58%	94.02%	73.18%	92.07%	71.16%	91.30%
34	76.05%	92.96%	72.27%	91.93%	69.82%	91.38%	80.99%	94.11%	73.27%	92.08%	71.23%	91.38%
35	75.88%	92.95%	72.75%	91.96%	70.07%	91.52%	81.52%	94.27%	73.48%	92.13%	71.39%	91.50%
36	76.03%	92.91%	73.02%	92.01%	70.29%	91.58%	81.97%	94.41%	73.86%	92.17%	71.50%	91.59%
37	76.09%	92.76%	73.22%	92.10%	70.30%	91.62%	81.97%	94.44%	74.21%	92.17%	71.71%	91.58%
38	76.06%	92.68%	73.51%	92.14%	70.49%	91.61%	82.32%	94.55%	74.55%	92.13%	71.96%	91.65%
39	76.19%	92.70%	73.67%	92.16%	70.65%	91.72%	82.24%	94.58%	74.72%	92.12%	72.12%	91.65%
40	76.09%	92.66%	73.64%	92.11%	70.78%	91.72%	82.45%	94.68%	75.05%	92.10%	72.42%	91.66%
41	76.21%	92.71%	73.74%	92.19%	70.78%	91.71%	82.30%	94.67%	75.11%	92.11%	72.63%	91.70%
42	76.27%	92.84%	73.72%	92.26%	70.91%	91.70%	82.47%	94.74%	75.34%	92.08%	72.74%	91.67%
43	76.21%	92.79%	73.78%	92.38%	70.96%	91.70%	82.06%	94.62%	75.43%	92.06%	72.86%	91.70%
44	76.36%	92.89%	73.71%	92.47%	70.96%	91.70%	82.01%	94.73%	75.50%	92.06%	72.91%	91.73%
45	76.40%	92.88%	73.76%	92.42%	70.97%	91.71%	81.83%	94.69%	75.69%	92.09%	72.97%	91.76%
46	76.43%	92.88%	73.70%	92.43%	71.05%	91.71%	80.91%	94.47%	75.72%	92.09%	72.94%	91.74%
47	76.41%	92.86%	73.68%	92.40%	71.13%	91.60%	80.61%	94.46%	75.77%	92.10%	72.97%	91.75%
48	76.36%	92.82%	73.82%	92.37%	71.12%	91.60%	80.14%	94.34%	75.90%	92.12%	72.95%	91.71%
49	76.45%	92.83%	73.80%	92.38%	71.13%	91.62%	79.79%	94.23%	75.86%	92.06%	72.93%	91.70%
50	76.38%	92.78%	73.90%	92.40%	71.13%	91.62%	79.37%	94.14%	75.86%	92.06%	72.97%	91.69%
51	76.30%	93.39%	73.86%	92.25%	71.14%	91.63%	79.43%	94.27%	75.92%	92.05%	72.95%	91.69%

Appendix C. Simulation Result of Memory System

C.1 All Sequence of QCIF & CIF

QCIF (1/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 240 kbit/s*

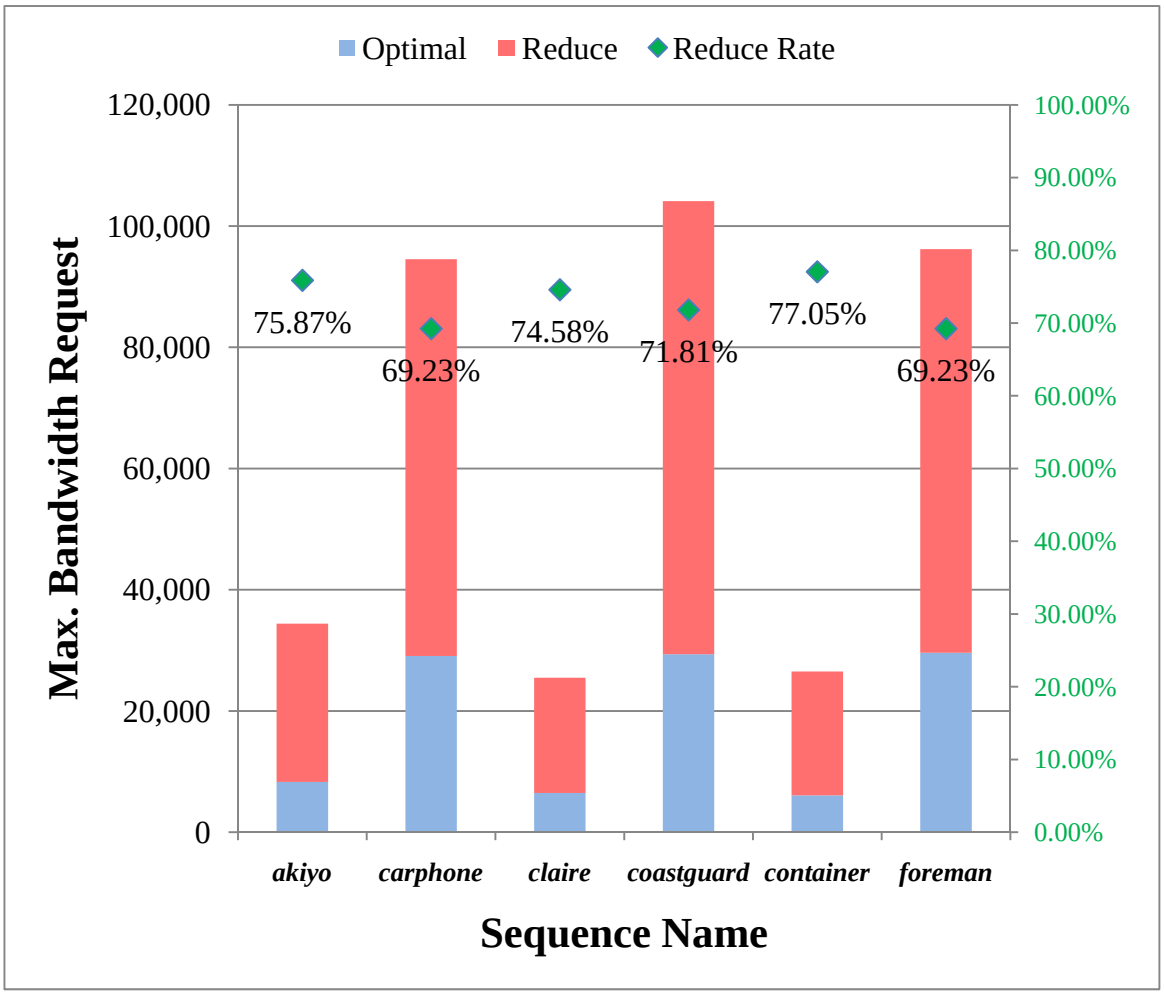


Figure 86. Max. B.W. requirement of memory system for QCIF sequences (1/4)

Table 35 Max. B.W. requirement of memory system for QCIF sequences (1/4)

Sequence Name	Total MVd	hMVd (MVd<2)	fMVd (MVd>2)	BandWidth@30fps		Reduction Rate
				Traditional	Optimal	
akiyo	34412	31573	2839	34412.00	8301.90	75.87%
carphone	120386	94464	25922	94543.98	29087.59	69.23%
claire	41964	37416	4548	25484.21	6477.81	74.58%
coastguard	104128	87067	17061	104128.00	29356.10	71.81%
container	26504	24941	1563	26504.00	6082.30	77.05%
foreman	128272	100634	27638	96204.00	29605.05	69.23%

QCIF (2/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 240 kbit/s*

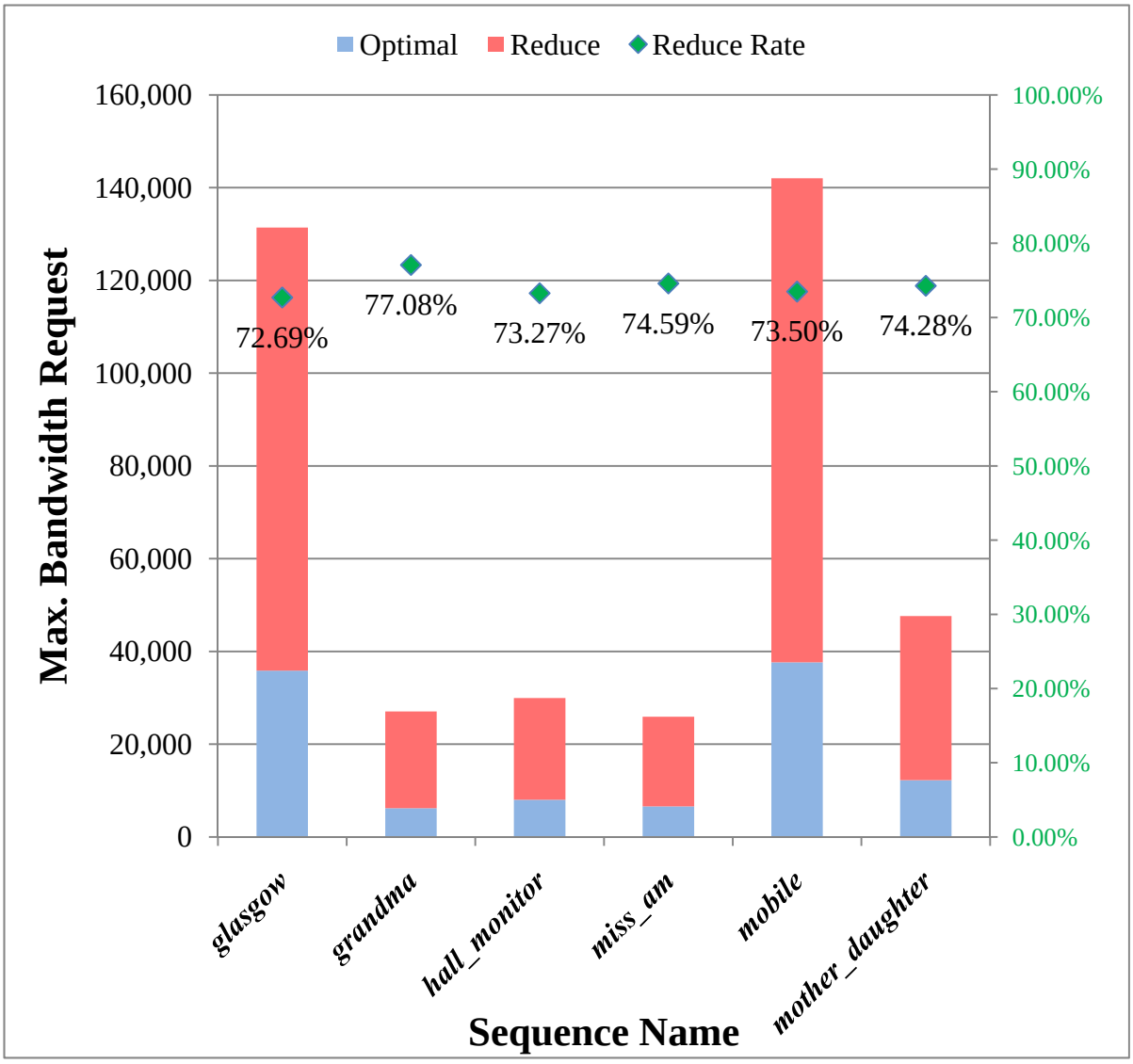


Figure 87. Max. B.W. requirement of memory system for QCIF sequences (2/4)

Table 36 Max. B.W. requirement of memory system for QCIF sequences (2/4)

Sequence Name	Total MVd	hMVd (MVd<2)	fMVd (MVd>2)	BandWidth@30fps		Reduction Rate
				Traditional	Optimal	
glasgow	43798	37399	43798	131394.00	35877.30	72.69%
grandma	78518	73931	78518	27075.17	6205.90	77.08%
hall_monitor	29950	25921	29950	29950.00	8004.50	73.27%
miss_am	12960	11559	12960	25920.00	6585.00	74.59%
mobile	142026	123550	142026	142026.00	37643.20	73.50%
mother_daughter	152616	135151	152616	47642.87	12254.64	74.28%

QCIF (3/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 240 kbit/s*

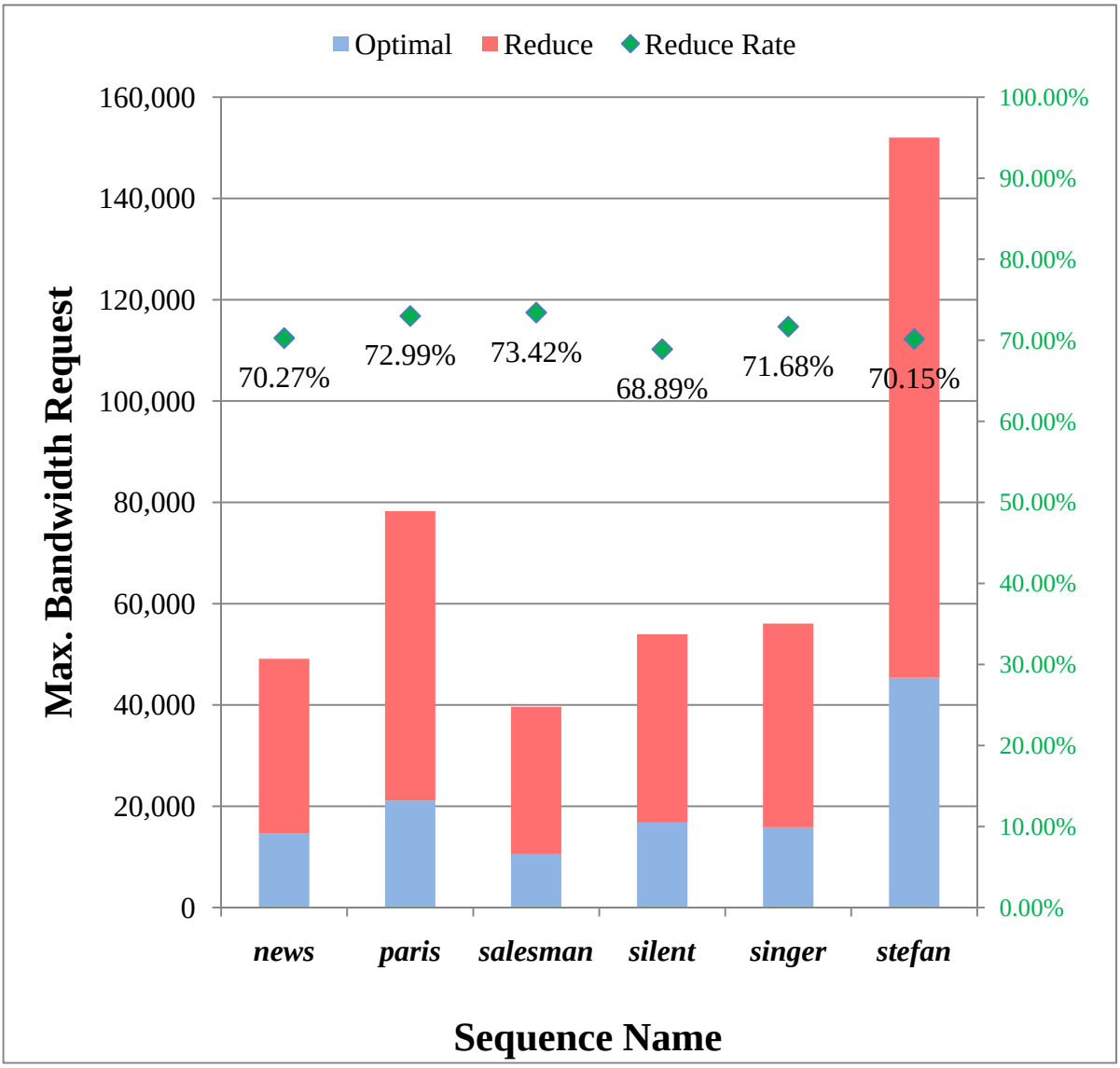


Figure 88. Max. B.W. requirement of memory system for QCIF sequences (3/4)

Table 37 Max. B.W. requirement of memory system for QCIF sequences (3/4)

Sequence Name	Total MVd	hMVd (MVd<2)	fMVd (MVd>2)	BandWidth@30fps		Reduction Rate
				Traditional	Optimal	
news	49112	39552	9560	49112.00	14602.40	70.27%
paris	104338	89703	14635	78253.50	21138.83	72.99%
salesman	59302	51492	7810	39622.72	10533.67	73.42%
silent	53950	41967	11983	53950.00	16781.50	68.89%
singer	46724	38947	7777	56068.80	15879.96	71.68%
stefan	152028	122068	29960	152028.00	45385.60	70.15%

QCIF (4/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 240 kbit/s*

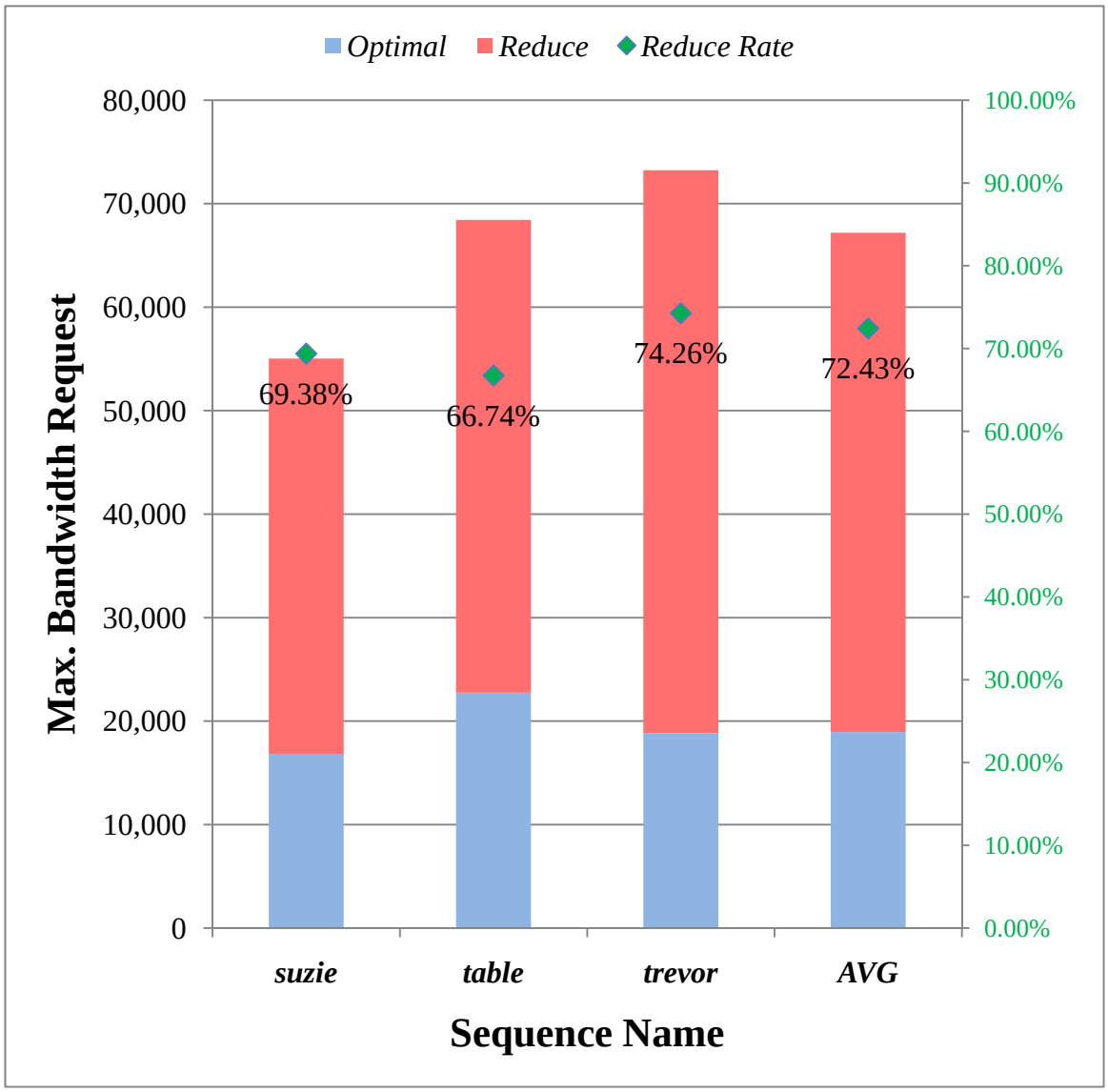


Figure 89. Max. B.W. requirement of memory system for QCIF sequences (4/4)

Table 38 Max. B.W. requirement of memory system for QCIF sequences (4/4)

Sequence Name	Total MVd	hMVd (MVd<2)	fMVd (MVd>2)	BandWidth@30fps		Reduction Rate
				Traditional	Optimal	
suzie	27512	21666	5846	55024.00	16850.80	69.38%
table	68420	50275	18145	68420.00	22756.50	66.74%
trevor	36616	29615	7001	73232.00	18847.00	74.26%
AVG	72073.14	60423.38	11649.76	67190.25	18964.65	72.43%

CIF (1/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 960 kbit/s*

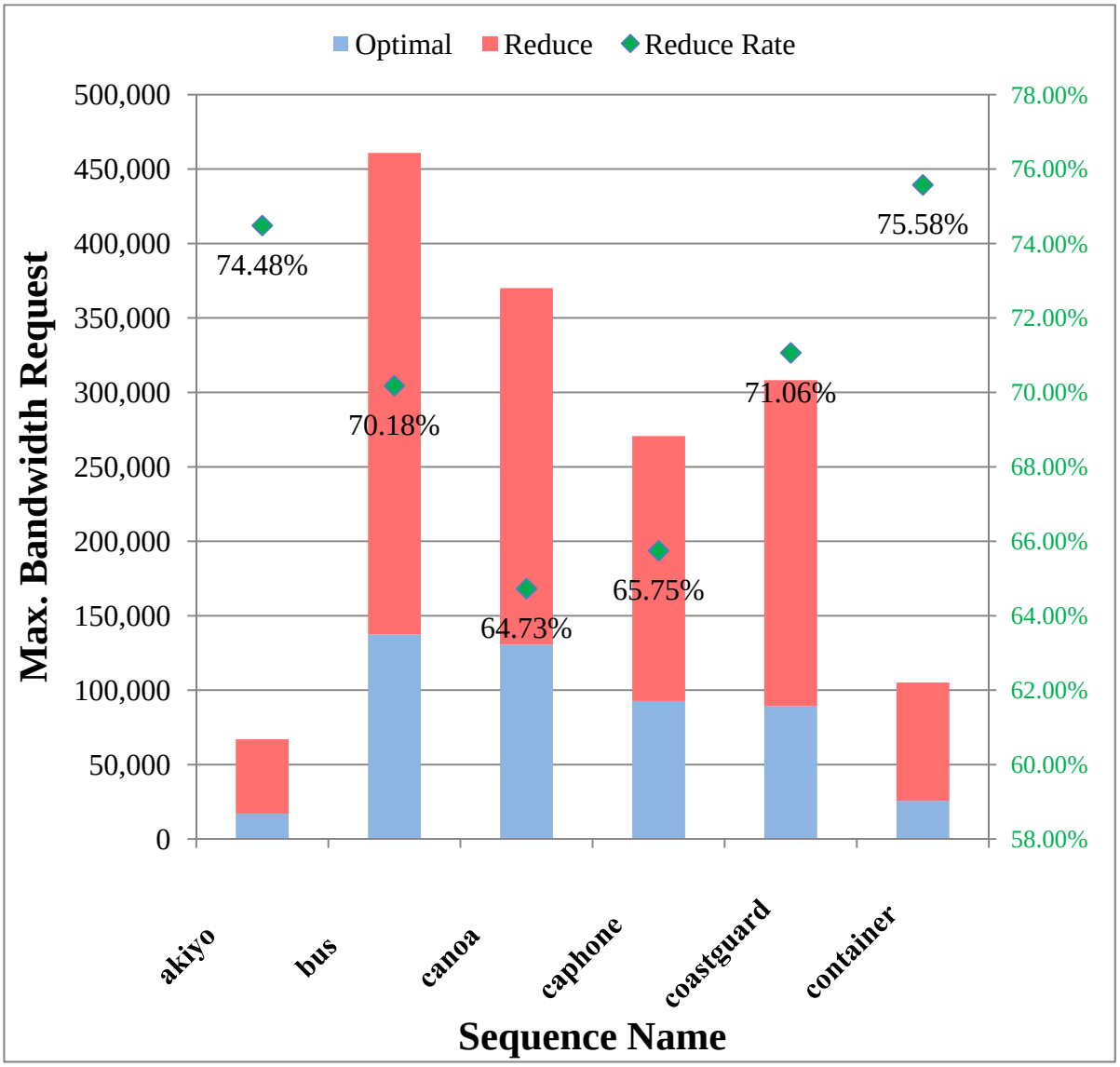


Figure 90. Max. B.W. requirement of memory system for CIF sequences (1/4)

Table 39 Max. B.W. requirement of memory system for CIF sequences (1/4)

Sequence Name	Total MVd	hMVd (MVd<2)	fMVd (MVd>2)	BandWidth@30fps		Reduction Rate
				Traditional	Optimal	
akiyo	66976	59586	7390	66976.00	17090.20	74.48%
bus	230424	185152	45272	460848.00	137441.60	70.18%
canoa	271408	188494	82914	370101.82	130552.64	64.73%
caphone	360848	257972	102876	270636.00	92705.70	65.75%
coastguard	308228	253134	55094	308228.00	89192.60	71.06%
container	105124	95822	9302	105124.00	25675.80	75.58%

CIF (2/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 960 kbit/s*

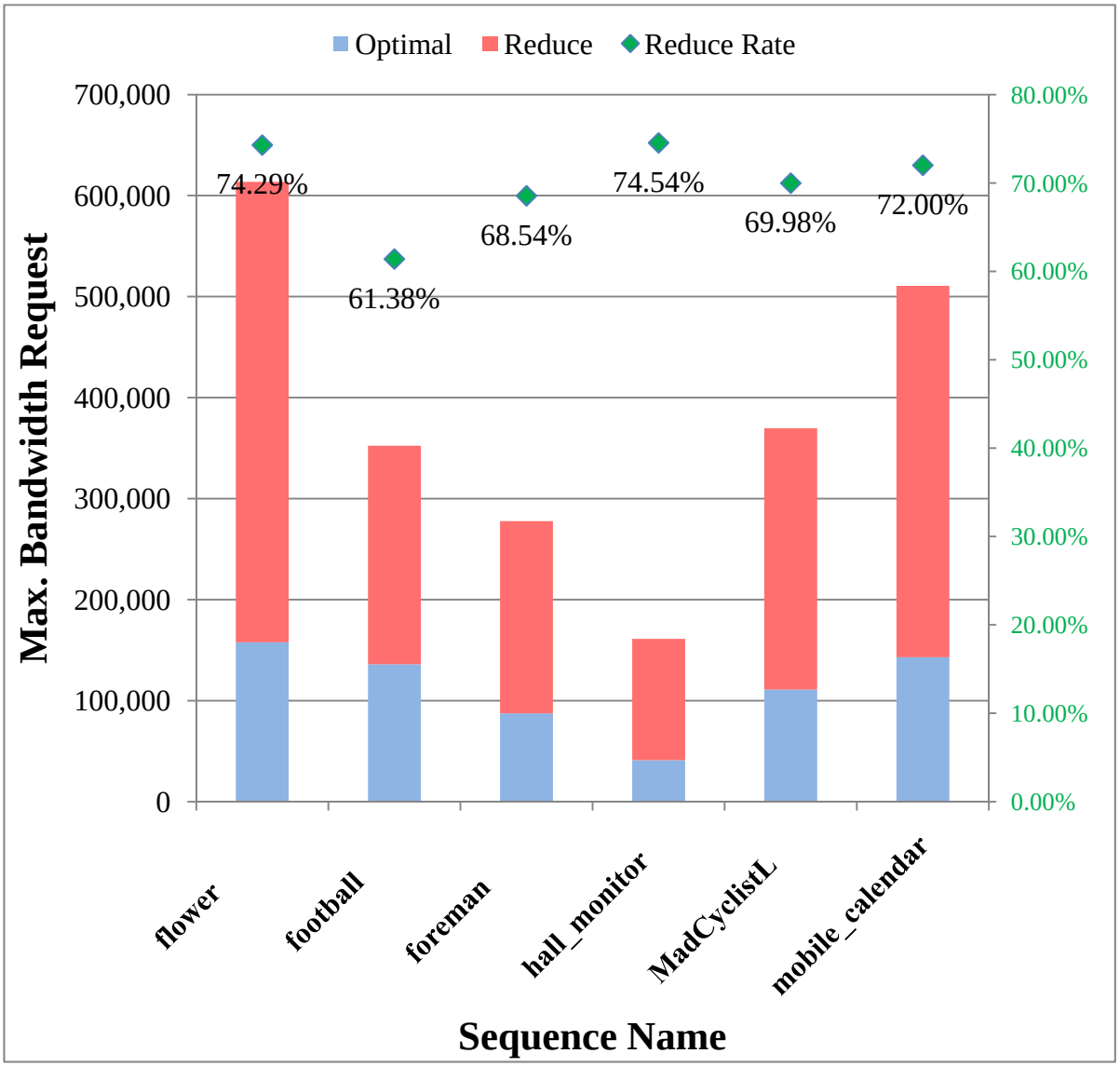


Figure 91. Max. B.W. requirement of memory system for CIF sequences (2/4)

Table 40 Max. B.W. requirement of memory system for CIF sequences (2/4)

Sequence Name	Total MVd	hMVd (MVd<2)	fMVd (MVd>2)	BandWidth@30fps		Reduction Rate
				Traditional	Optimal	
flower	511382	452952	58430	613658.40	157789.68	74.29%
football	305438	191716	113722	352428.46	136094.54	61.38%
foreman	277770	214119	63651	277770.00	87379.50	68.54%
hall_monitor	161218	143602	17616	161218.00	41051.60	74.54%
MadCyclistL	1108682	886449	222233	369560.67	110950.97	69.98%
mobile_calendar	510730	429057	81673	510730.00	142982.50	72.00%

CIF (3/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 960 kbit/s*

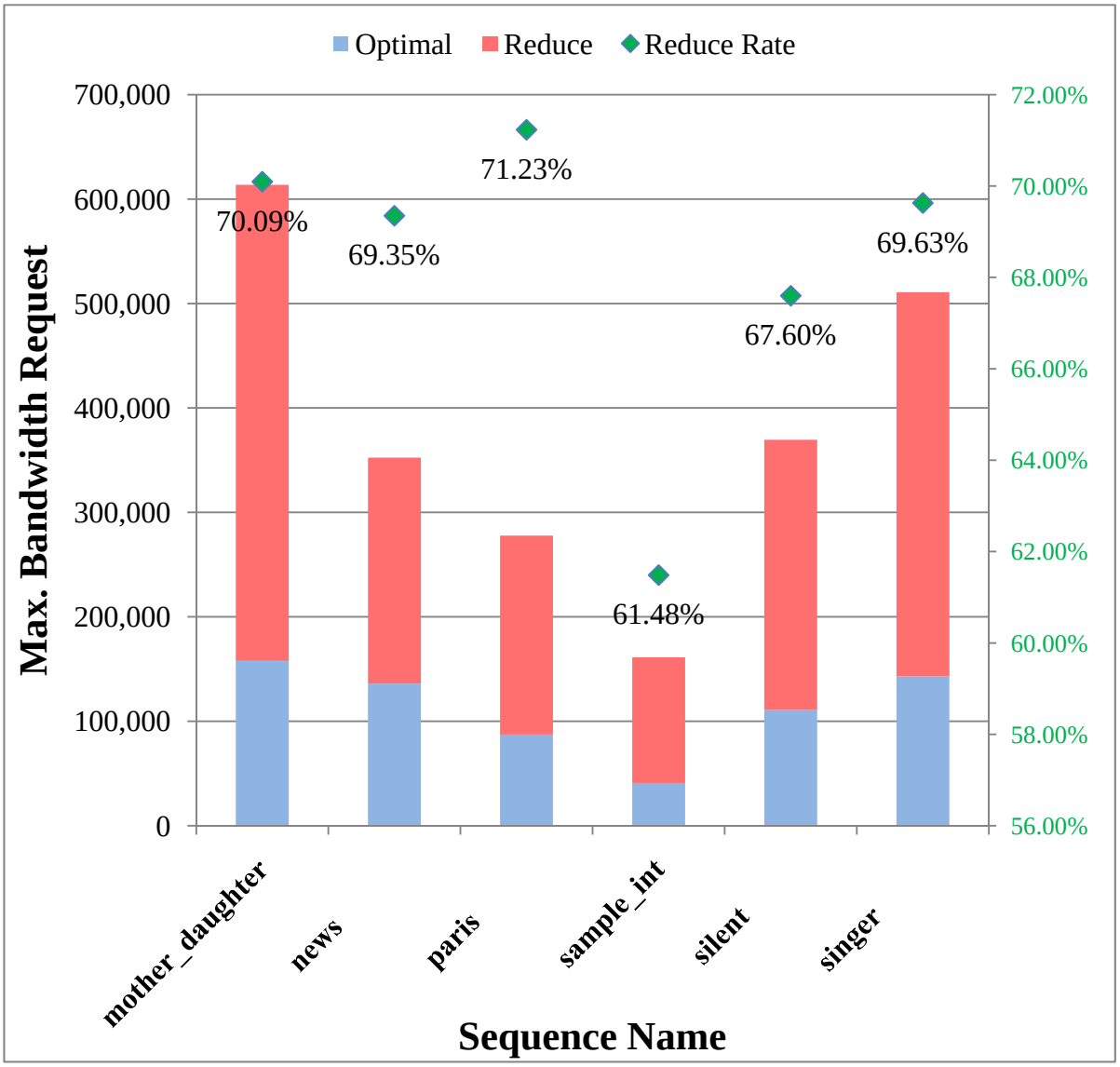


Figure 92. Max. B.W. requirement of memory system for CIF sequences (3/4)

Table 41 Max. B.W. of memory system for CIF sequences (3/4)

Sequence Name	Total MVd	hMVd (MVd<2)	fMVd (MVd>2)	BandWidth@30fps		Reduction Rate
				Traditional	Optimal	
mother_daughter	107398	86117	21281	107398.00	32120.10	70.09%
news	130074	102363	27711	130074.00	39870.30	69.35%
paris	647084	533577	113507	182277.18	52442.34	71.23%
sample_int	206366	129945	76421	281408.18	108386.86	61.48%
silent	145992	109777	36215	145992.00	47305.90	67.60%
singer	118798	94153	24645	142557.60	43298.52	69.63%

CIF (4/4) *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 960 kbit/s*

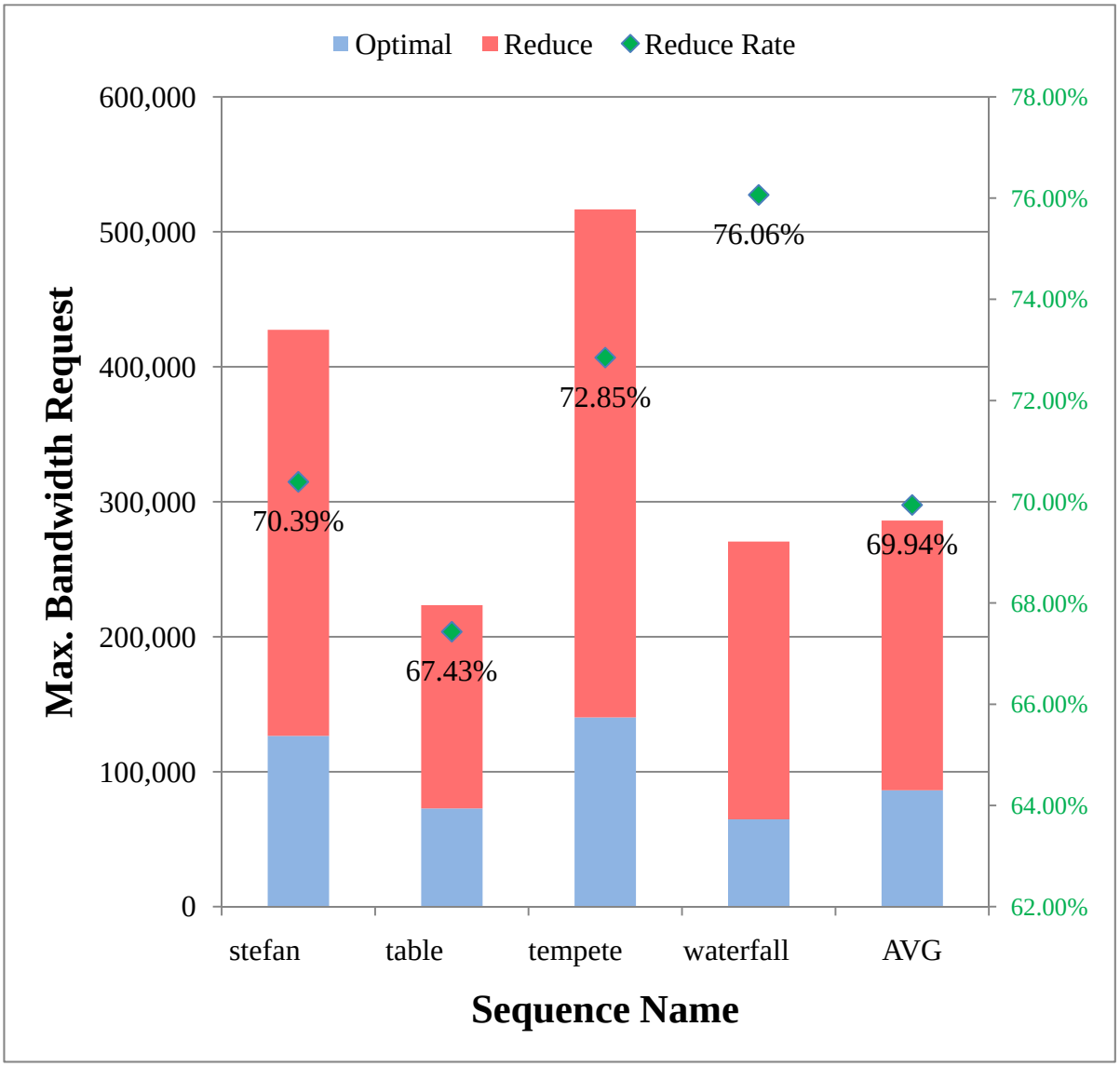


Figure 93. Max. B.W. requirement of memory system for CIF sequences (4/4)

Table 42 Max. B.W. requirement of memory system for CIF sequences (4/4)

Sequence Name	Total MVd	hMVd (MVd<2)	fMVd (MVd>2)	BandWidth@30fps		Reduction Rate
				Traditional	Optimal	
stefan	427456	345333	82123	427456.00	126552.70	70.39%
table	223420	167266	56154	223420.00	72761.00	67.43%
tempete	447692	383655	64037	516567.69	140257.96	72.85%
waterfall	234498	216026	18472	270574.62	64771.85	76.06%
AVG	313955	251194	62761	286136.57	86212.49	69.94%

C.2 Optimization for Memory and Bandwidth

QCIF *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 240 kbit/s*

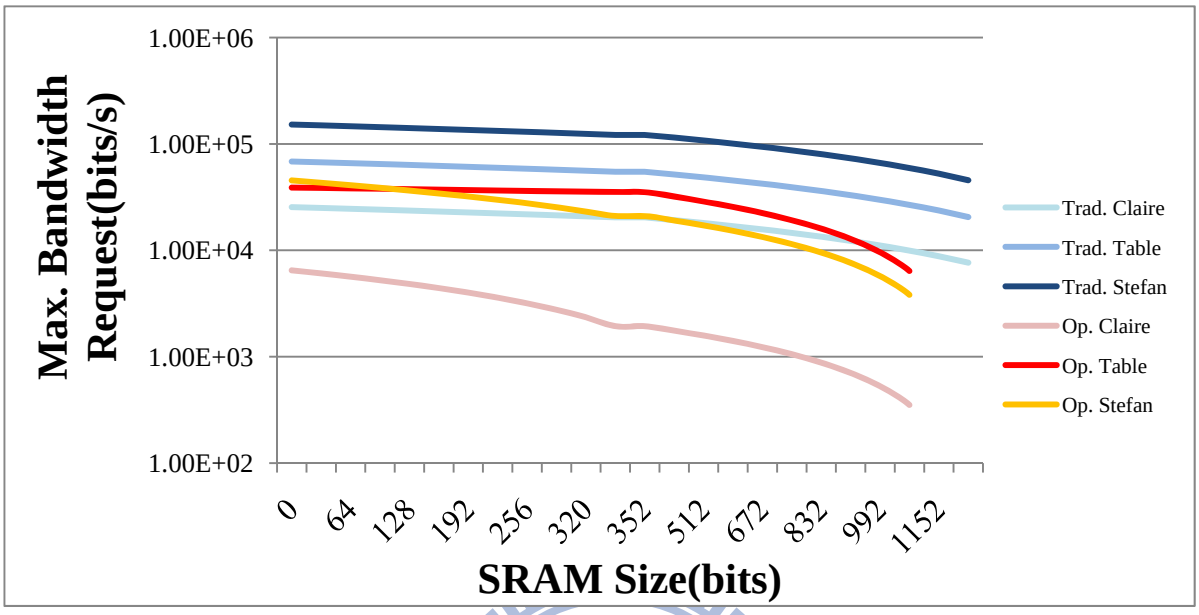


Figure 94. Max. B.W. requirement of memory system for SRAM size for QCIF seq.

Table 43 Max. B.W. requirement of memory system for SRAM size for QCIF seq.

SRAM size	QCIF					
	Claire		Table		Stefan	
	Traditional	Optimal	Traditional	Optimal	Traditional	Optimal
0	25484.21	6477.81	68420.00	38821.50	152028.00	45385.60
32	25020.86	6064.68	67176.00	38491.59	149263.85	43166.18
64	24557.51	5651.55	65932.00	38161.68	146499.71	40946.76
96	24094.16	5238.42	64688.00	37831.77	143735.56	38727.35
128	23630.81	4825.29	63444.00	37501.86	140971.42	36507.93
160	23167.46	4412.15	62200.00	37171.95	138207.27	34288.51
192	22704.11	3999.02	60956.00	36842.05	135443.13	32069.09
224	22240.77	3585.89	59712.00	36512.14	132678.98	29849.67
256	21777.42	3172.76	58468.00	36182.23	129914.84	27630.25
288	21314.07	2759.62	57224.00	35852.32	127150.69	25410.84
320	20850.72	2346.49	55980.00	35522.41	124386.55	23191.42
352	20387.37	1933.36	54736.00	35192.50	121622.40	20972.00
352	20387.37	1933.36	54736.00	35192.50	121622.40	20972.00
432	19229.00	1757.60	51626.00	31993.18	114712.04	19065.45
512	18070.62	1581.84	48516.00	28793.86	107801.67	17158.91
592	16912.25	1406.08	45406.00	25594.55	100891.31	15252.36
672	15753.88	1230.32	42296.00	22395.23	93980.95	13345.82
752	14595.50	1054.56	39186.00	19195.91	87070.58	11439.27
832	13437.13	878.80	36076.00	15996.59	80160.22	9532.73
912	12278.76	703.04	32966.00	12797.27	73249.85	7626.18
992	11120.38	527.28	29856.00	9597.95	66339.49	5719.64
1072	9962.01	351.52	26746.00	6398.64	59429.13	3813.09
1152	8803.64	175.76	23636.00	3199.32	52518.76	1906.55
1232	7645.26	0.00	20526.00	0.00	45608.40	0.00

CIF *@ jm16.1[2] QP:28 GOP: IBPBP… BR: 960 kbit/s*

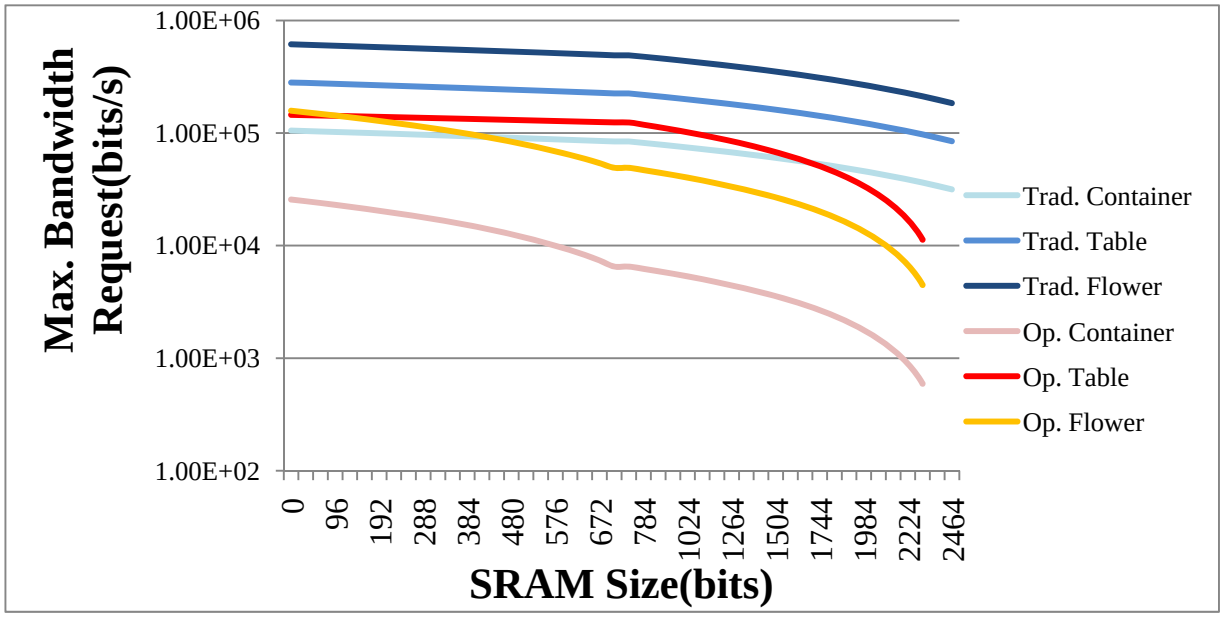


Figure 95. Max. B.W. requirement of memory system for SRAM size for CIF seq.

Table 44 Max. B.W. requirement of memory system for SRAM size for CIF seq.

SRAM size	CIF					
	Container		Sample_int		Flower	
	Traditional	Optimal	Traditional	Optimal	Traditional	Optimal
0	105124.00	25675.80	281408.18	144880.50	613658.40	157789.68
64	103212.65	23933.58	276291.67	142985.76	602500.97	147907.09
128	101301.31	22191.36	271175.16	141091.03	591343.55	138024.50
192	99389.96	20449.15	266058.64	139196.29	580186.12	128141.91
256	97478.62	18706.93	260942.13	137301.56	569028.70	118259.32
320	95567.27	16964.71	255825.62	135406.82	557871.27	108376.73
384	93655.93	15222.49	250709.11	133512.09	546713.85	98494.15
448	91744.58	13480.27	245592.60	131617.35	535556.42	88611.56
512	89833.24	11738.05	240476.08	129722.62	524399.00	78728.97
576	87921.89	9995.84	235359.57	127827.88	513241.57	68846.38
640	86010.55	8253.62	230243.06	125933.14	502084.15	58963.79
704	84099.20	6511.40	225126.55	124038.41	490926.72	49081.20
864	79320.84	5919.45	212335.26	112762.19	463033.16	44619.27
1024	74542.47	5327.51	199543.98	101485.97	435139.59	40157.35
1184	69764.11	4735.56	186752.70	90209.75	407246.03	35695.42
1344	64985.75	4143.62	173961.42	78933.53	379352.47	31233.49
1504	60207.38	3551.67	161170.14	67657.31	351458.90	26771.56
1664	55429.02	2959.73	148378.86	56381.10	323565.34	22309.64
1824	50650.65	2367.78	135587.58	45104.88	295671.77	17847.71
1984	45872.29	1775.84	122796.30	33828.66	267778.21	13385.78
2144	41093.93	1183.89	110005.02	22552.44	239884.65	8923.85
2304	36315.56	591.95	97213.74	11276.22	211991.08	4461.93
2464	31537.20	0.00	84422.45	0.00	184097.52	0.00

Appendix D. Hardware Verification

Table 45. Simulation result for I slice

Sequence Name	***I Slice***											
	Total Bin	Total Cycle	Terminal Cycle	Bypass Cycle	Regular Cycle			MPS bin	MPS rate	Hit bin	Hit rate	BPC
					Normal	Idle	Stall					
Total	221087	237346	495	35149	185443	737	15522	131466	70.89%	169922	91.63%	0.931497
Foreman	29218	31513	99	4398	24721	205	2090	17582	71.12%	22632	91.55%	0.927173
Akiyo	24816	26876	99	3755	20962	148	1912	14292	68.18%	19050	90.88%	0.923352
Carphone	26172	28204	99	3829	22244	175	1857	15349	69.00%	20387	91.65%	0.927953
Stefan	58967	62996	99	9550	49318	100	3929	35410	71.80%	45389	92.03%	0.936044
Mobile	81914	87757	99	13617	68198	109	5734	48833	71.60%	62464	91.59%	0.933418

Table 46. Simulation result for P slice

Sequence Name	***P Slice***											
	Total Bin	Total Cycle	Terminal Cycle	Bypass Cycle	Regular Cycle			MPS bin	MPS rate	Hit bin	Hit rate	BPC
					Normal	Idle	Stall					
Total	43950	47462	495	6435	37020	171	3243	27175	73.41%	33778	91.24%	0.926004
Foreman	4981	5365	99	801	4081	27	357	2778	68.07%	3725	91.28%	0.928425
Akiyo	270	371	99	6	165	97	5	141	85.45%	160	96.97%	0.727763
Carphone	5624	6068	99	903	4622	23	421	3304	71.48%	4201	90.89%	0.926829
Stefan	17650	18918	99	2802	14749	17	1251	10888	73.82%	13498	91.52%	0.932974
Mobile	15425	16740	99	1923	13403	7	1209	10064	75.09%	12194	90.98%	0.921446

Table 47. Simulation result for B slice

Sequence Name	***B Slice***											
	Total Bin	Total Cycle	Terminal Cycle	Bypass Cycle	Regular Cycle			MPS bin	MPS rate	Hit bin	Hit rate	BPC
					Normal	Idle	Stall					
Total	8184	9112	495	952	6737	381	548	4787	71.06%	6189	91.87%	0.898156
Foreman	1403	1554	99	135	1169	64	88	832	71.17%	1081	92.47%	0.902831
Akiyo	198	299	99	0	99	101	0	99	100.00%	99	100.00%	0.662207
Carphone	1881	2061	99	243	1539	67	113	1116	72.51%	1426	92.66%	0.912664
Stefan	3550	3912	99	444	3007	69	293	2118	70.44%	2714	90.26%	0.907464
Mobile	1152	1286	99	130	923	80	54	622	67.39%	869	94.15%	0.895801

Table 48. Summary of I,P,B slice

Sequence Name	Total Bin	Total Cycle	Terminal Cycle	Bypass Cycle	Regular Cycle			MPS bin	MPS rate	Hit bin	Hit rate	BPC
					Normal	Idle	Stall					
Total	***273221***	***293920***	***1485***	***42536***	***229200***	***1289***	***19313***	***163428***	***71.30%***	***209889***	***91.57%***	***0.929576***

Biography

姓名: 郭明諭

戶籍地: 台灣 新北市

出生日期: 1985.02.05

學歷: 2003.09 ~ 2008.06 元智大學 電機工程學系 學士

2008.09 ~ 2010.12 國立交通大學 電子工程研究所

系統組 碩士班

發表論文:

- **Ming-Yu Ku**, Yao Li, Chen-Yi Lee, "An Area-efficiently High-accuracy Prediction-based CABAC Decoder for H.264AVC," *IEEE International Symposium on Circuit and System (ISCAS'11)*, May 2011.