PERFORMANCE AND COMPLEXITY ANALYSES OF H.264/AVC CABAC ENTROPY CODER

Ho Boon Leng

(B.Eng (Hons),NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND

COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2006

ACKNOWLEDGEMENTS

I would like to dedicate this thesis to my family, especially my parents. The journey to obtain the master degree has been tough and I am extremely grateful for their understanding and constant support.

I would also like to express my gratitude to my supervisor, Dr Le Minh Thinh for his patience, guidance, and advice in my research. He has provided constructive suggestions and recommendations for my research work.

I would also like to express my sincere thanks to my colleagues, Tian Xiaohua and Sun Xiaoxin for all the helps they have given me throughout my research work

Last but not least, I would like to express my utmost appreciations to my good friends, Ong Boon Kar and Cheong Kiat Fah for always having been there for me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

B&CM	Binarization & Context Modeling
CABAC	Context Adaptive Binary Arithmetic Coding
CAVLC	Context Adaptive Variable Length Coding
CIF	Common Intermediate Format
FMS	Finite Machine State
GOP	Group of Pictures
IS	Interval Subdivision
ISA	Instruction Set Architecture
LPS	Least Probable Symbol
MPEG	Moving Picture Expert Group
MPS	Most Probable Symbol
NRDSE	Non-residual Data Syntax Element
QCIF	Quarter Common Intermediate Format
RDO	Rate Distortion Optimization
RDSE	Residual Data Syntax Element
Y-PSNR	Luma Peak Signal-to-Noise Ratio

ABSTRACT

Context Adaptive Binary Arithmetic Coding (CABAC) is one of the entropy coding schemes defined in H.264/AVC. In this work, the coding efficiency, the computational and memory requirements of CABAC are comprehensively assessed for the different type of video encoders. The main contributions of the thesis are the reported findings from the performance and complexity analyses. These findings assist implementers in deciding when to use CABAC for a cost-effective realization of the video codec that meets their system's computational and memory resources. Bottlenecks in CABAC have also been identified and recommendations on possible complexity reductions have been proposed to system designers and software developers.

CABAC is more complex than Context Adaptive Variable Length Coding (CAVLC), and is dominated by data transfer in comparison to arithmetic and logic operations. However, it is found that the use of CABAC is only resource expensive when Rate-Distortion Optimization (RDO) is employed. For a RDO encoder, CABAC hardware accelerator will be needed if the real-time requirement is met. Alternatively, the use of suboptimal RDO techniques can reduce the computational and memory requirements of CABAC on the video encoder, making it less expensive to use CABAC in comparison to CAVLC.

CHAPTER 1 INTRODUCTION

Over the past decade, digital video compression technology has evolved tremendously, which made possible many application scenarios from video storage to video broadcast and streaming over Internet and telecommunication networks. The aim of video compression is to represent the video data with the lowest bit-rate at a specified level of reproduction fidelity, or to represent the video data at the highest reproduction fidelity with a given bit-rate.

H.264/AVC [1] is the latest international video compression standard. In comparison to the previous video compression standards such as MPEG-4 [2] and H.263 [3], it provides higher coding performance and better error resilience through the use of improved or new coding tools at different stages of the video coding. For the entropy coding stage, H.264/AVC offers two new schemes for coding its macroblock-level syntax elements: *Context Adaptive Variable Length Coding* (CAVLC) and *Context Adaptive Binary Arithmetic Coding* (CABAC). Both entropy coding schemes achieve better coding efficiency than their predecessors in the earlier standards as they employed context-conditional probability estimates. Comparatively, CABAC performs better than CAVLC in terms of coding efficiency as it encodes data with non-integer length codeword, and it adjusts its context-conditional probability estimates to adapt to the non-stationary source statistics. However, the higher coding efficiency of CABAC comes at the expense of increased complexity in the entropy coder. This is one of the reasons why the developer team of H.264/AVC excludes CABAC from the *Baseline* profile [5].

1.1 Research Work

In this work, comprehensive performance and complexity analyses of CABAC at both the entropy coder level and the video encoder/decoder levels will be conducted using software verification model. Both variable bit-rate video encoder and constant bit-rate video encoder will be considered. For the performance analyses, percentage bit-rate savings and changes in peak signal-to-noise ratio of the video luminance component (Y-PSNR) will be used. As for the complexity analyses, computational complexity, data transfer complexity and memory usage will be assessed. The goals of the analyses are:

(a) To present the computational and memory requirements of CABAC

(b) To identify "scenarios" where the use of CABAC is more cost-effective based on a co-evaluation of the system's coding efficiency and complexity performance across different configurations and encoder types.

(c) To identify the possible bottlenecks in the CABAC entropy coder and to make possible recommendations / suggestions on complexity reduction of CABAC to system designers or software developers.

1.2 Motivation

The CABAC tool is not supported in the *Baseline* profile of H.264/AVC. As such, it is commonly believed that using CABAC is computationally expensive for a video encoder. However, no work has been done on evaluating the complexity requirements of using CABAC except in [4], which gives a brief assessment of the effect of using CABAC on the video encoder's data transfer complexity. (More

details on the related works that have been carried out for H.264/AVC are given in Chapter 2.)

In [4], the additional memory requirement of using CABAC over CALVC from the perspective of the video encoder is briefly reported, and this result has been referenced by many literatures (due to the lack of works done in this area). However, the complexity evaluation of CABAC given in their work is far from being complete, as it performs a tool-by-tool add-on analysis, and CABAC is only considered for one specific encoder configuration. Moreover, it also failed to include any complexity analyses of using CABAC at the decoder.

There are also some drawbacks in evaluating the complexity increment of using CABAC over CAVLC from the perspective of the video encoder. The results can be misleading as these complexity figures also depend on the choices of coding tools used in the video encoder. This makes comparison of such figures across different configurations less meaningful. Besides, analyzing the complexity performance of CABAC from the perspective of the video encoder will be more of interest to implementers, who wish to achieve a cost-effective realization of the video codec. However, it may be less relevant for system designers of CABAC as the complexity figures do not reflect the true requirements of the entropy coder. Rather, they will be more interested in the complexity performance of CABAC from the perspective of the entropy coder.

As such, these provide the motivation for comprehensive analyses on the performance and complexity of CABAC at two levels: top-level video encoder and the entropy coder level. It is believed that analyses at the entropy coder level will be useful to system designers or software developers in understanding the CABAC

system properties, to gauge its implementation cost and for optimizing it design implementation.

1.3 Thesis Contributions

The thesis contributions have been four-fold:

(a) provided inputs - findings from co-evaluation of performance-complexity analyses of CABAC - that can assist implementer in deciding whether to use CABAC in the video encoder,

(b) identified possible bottlenecks in CABAC and suggests recommendations on complexity reduction to system designer and software developers,

(c) identified when the use of CABAC hardware accelerator may not be necessarily helpful in the video encoder, and

(d) developed a set of profiler tools based on Pin [13] for measuring instruction-level complexity and memory access frequency of any functional coding block of H.264/AVC that can also be used on other video codec.

1.4 Thesis Organization

The contents in this thesis are organized as follows. In Chapter 2, an overview of Context Adaptive Binary Arithmetic Coding (CABAC), a review of the complexity analysis methodologies that have been used for video multimedia system, and a literature review of existing works will be given. In Chapter 3, the performance of the CABAC, benchmarked against CAVLC is given for the different video configurations

so as to explore the inter-tool dependencies. In Chapter 4, the complexity analyses of using CABAC at both the entropy coder level and the video encoder/decoder levels are given. Related research work on rate-distortion optimization (RDO) extending from the complexity analyses of CABAC are given in Chapter 5. Finally, conclusions are given in Chapter 6.

CHAPTER 2 BACKGROUND

In this chapter, the role of the entropy coder is discussed and an overview of CABAC is given. This is followed by presenting the different encoder controls that can be used in the video encoder. Lastly, a review of the complexity analysis methodologies that have been used for video multimedia system, and a literature review of existing works will be given.

2.1 Entropy coder

The entropy coder may serve up to two roles in a H.264/AVC video encoder. The primary role of the entropy coder is to generate the compressed bitstream of the video file for transmission or storage. For video encoders that optimize its mode decision using rate-distortion optimization (RDO), its entropy coder performs an additional role during the mode selection stage. The entropy coder computes the bit-rates needed by each candidate prediction mode. The computed rate of information is then used to guide the mode selection. Further details are given in sub-section 2.3.2.

2.2 Overview of CABAC

Context Adaptive Binary Arithmetic Coding (CABAC) [5] is one of the entropy coding schemes in H.264/AVC, and is only supported in the *Main* profile. Fig. 2.1 shows the block diagram for encoding and decoding a single syntax element in CABAC.

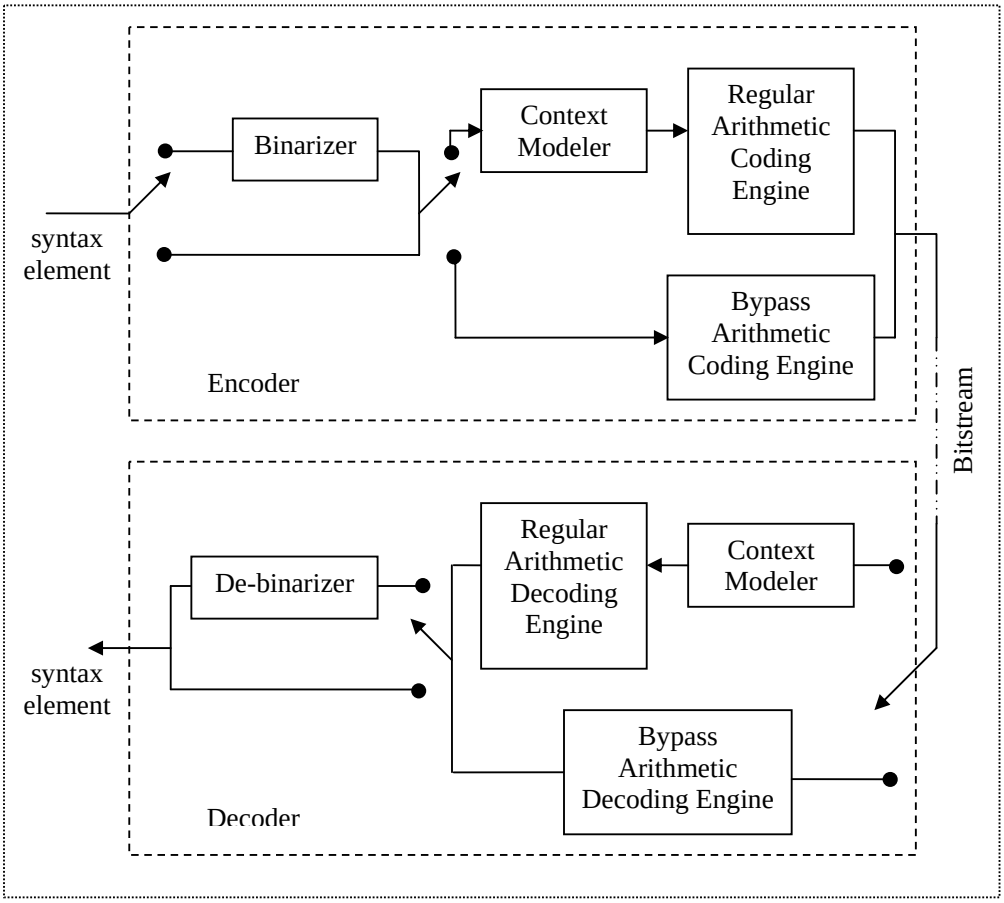


Figure 2.1: CABAC entropy coder block diagram

The encoding/decoding process using CABAC comprises of three stages: binarization, context modeling, and binary arithmetic coding.

2.2.1 Binarization

The binarization stage maps all non-binary syntax elements into a binary sequence known as bin-string using four basic binarization schemes: Unary (U), Truncated Unary (TU), k^{th} order Exp-Golomb (EGK) and Fixed Length (FL). The only exception where these binarization schemes are not used is when encoding the macroblock type and the sub-macroblock type syntax elements. For these syntax elements, unstructured binary trees are used instead of binarization.

2.2.2 Context Modeling

Each bin in a bin string is encoded in either *normal* mode or *bypass* mode depending on the semantic of the syntax. For a bypass bin, the context modeling stage is skipped because a fixed probability model is always used. On the other hand, each normal bin selects a probability model based on its context from a specified set of probability models in the context modeling stage. In total, 398 probability models are used for all syntax elements.

There are four types of context. The type of context used by each normal bin for selecting the best probability model depends on the syntax element that is encoded. The first type of context considers the related bin values in its neighboring macroblocks or sub-blocks. The second type considers the values of the prior coded bins of the bin-string. These two types of contexts are only used for non-residual data syntax elements (NRDSE). The last two types of context are only used for residual data syntax elements (RDSE). One of them considers the position of the syntax element in the scanning path of the macroblock while the other evaluates a count of non-zero encoded levels with respect to a given threshold level.

2.2.3 Arithmetic Coding

In the binary arithmetic coding (BAC) stage, the bins are arithmetic coded. Binary arithmetic coding is based on the principle of recursive sub-division of an interval length as follows:

$$E_{LPS} = P_{LPS} \cdot E \tag{2-1}$$

$$E_{MPS} = E - E_{LPS} \tag{2-2}$$

$$L_{LPS} = L + E - E_{LPS} \tag{2-3}$$

$$L_{MPS} = L \tag{2-4}$$

where E denotes the current interval length, L denotes the current lower bound of E, P_{LPS} denotes the probability of least probable symbol (LPS) from the selected probability model. E_{LPS} and E_{MPS} denote the new lengths of the partitioned intervals corresponding to LPS and the most probable symbol (MPS). L_{LPS} and L_{MPS} denote the corresponding lower bounds of the partitioned intervals. For each bin, the current interval is first partition into two as given in Eqn. 2-1 to Eqn. 2-4. The bin value is then encoded by selecting the newly partitioned length that corresponds to the bin value (either LPS or MPS) as the new current interval. E and L are also referred as the coding states of the arithmetic coder.

In H.264/AVC, the multiplication operation of interval subdivision in Eqn. 2-1 is replaced by using finite state machine (FSM) with a look-up table of pre-computed intervals as follows:

$$E_{LPS} = RangeTable[\hat{P}_{LPS}][\hat{E}] \tag{2-5}$$

The FSM consists of 64 probability states, $\hat{P}_{LPS}$ and 4 interval states, $\hat{E}$. For the normal bins, the selected conditional probability model is updated with the new statistic after the bin value is encoded.

2.2.4 Renormalization

To prevent underflow, H.264/AVC performs a renormalization operation when the current interval length, E falls below a specified interval length after coding a bin. This is a recursive operation which resizes the interval length through scaling until the current interval exceeds the specified interval length. The codeword is output on the fly each time bits are available after the scaling operation.

2.3 Encoder Control

The encoder control refers to the strategy used by the encoder in selecting the optimal prediction mode to encode each macroblock. In H.264/AVC, the encoder can select from up to 11 prediction modes: 2 Intra prediction modes and 9 Inter prediction mode, including *SKIP* and *DIRECT* modes to encode a macroblock. Note that the encoder control is a non-normative part of the H.264/AVC standard. Several encoder controls have been proposed and are reviewed below.

2.3.1 Non-RDO encoder

For a non-RDO encoder, either the sum of absolute difference (SAD) or the sum of absolute transform difference (SATD) can be used as the selection criteria. The optimal prediction mode selected to encode the macroblock corresponds to the prediction mode that minimizes the macroblock residual signal, i.e. the minimum SAD or SATD value.

2.3.2 RDO encoder

For a RDO encoder, a rate-distortion cost function is used as the selection criteria for the optimal mode and is given as

$$J = D + \lambda R \tag{2-6}$$

where J is the rate-distortion cost, D the distortion measure, λ the Lagrange multiplier, and R the bit-rate. The optimal prediction mode used to encode the macroblock corresponds to the prediction mode that yields the least rate-distortion cost. Note that to obtain the bit-rate, entropy coding has to be performed for each candidature mode. This significantly increases the amount of entropy coding performed in the video encoder.

2.3.3. Fast-RDO encoder

The fast-RDO encoder employs the fast RDO algorithm proposed in [23]. Similar to the RDO encoder, it uses the rate-distortion cost function in Eqn. 2-4 as the selection criteria. However, it does not perform an "exhaustive" search through all candidate prediction modes. Rather, it terminates the search process once the rate-distortion cost of a candidate prediction mode lies within a threshold - a value derived from the rate-distortion cost of the co-located macroblock in the previous encoded frame. The current candidate prediction mode whose rate-distortion cost lies within the threshold is selected as the optimal prediction mode, and the remaining prediction modes are bypassed. If none of the prediction modes meets the early termination criteria, the prediction mode with the least rate-distortion cost is then selected as the optimal prediction mode.

2.4 Complexity Analysis Methodologies

In this section, a review of the known complexity analysis methodologies is given. Complexity analyses are often carried out using verification models software (in the case of video standards) such as the Verification Model (VM) and the Joint Model (JM) reference software implementations for MPEG-4 and H.264/AVC, respectively. These are un-optimized reference implementations but are sufficient for analyzing the critical blocks in the algorithm for optimization and discovering the bottlenecks. On the other hand, optimized source codes are needed or preferred for complexity evaluation when performing hardware / software partitioning as in [6] or when comparing the performance-complexity between video codec as in [7].

2.4.1 Static Code Analysis

Static code analysis is one way of evaluating the computational complexity of an algorithm, a program or a system. Such analysis requires the availability of the high-level language source code such as the C codes of the Joint Model (JM) reference software of H.264/AVC. The methods based on such analysis includes counting the number of lines-of-code (LOC), counting the number of arithmetic and logical operations, determining the time complexity of the algorithms, and determining the lower or upper bound running time of the program by explicit or implicit enumeration of program paths [8]. Such analyses measure the algorithm's efficiency but do not take into considerations the different input data statistics. In order to obtain an accurate static analysis, restricted programming style such as absence of recursion, dynamic data structure and bounded loop are needed so that the maximal time spent in any part of the program can be calculated.

2.4.2 Run-time Computational Complexity Analysis

For run-time complexity analysis, profiling data are collected when the program executes at run time on a given specific architecture. The advantage of run-time complexity analysis is that input data dependency is also included. One method of run-time computational complexity analysis is to measure the execution time of the program using ANSI C clock function [9]. An alternative is to measure the execution time of the program in terms of clock cycles using tools like Intel VTune - an automated performance analyzer, or PAPI - a tool that allows access to the performance hardware counters of the processor for measuring clock cycle [10].

Function-level information can also be collected for coarse complexity evaluation using profilers such as Visual Studio Environment Profiling Tool or Gprof

[11]. These profiling tools provide information on function call frequency and the total execution time spent by each function in the program. This information allows identifying the critical functions for optimization and help partial redesign of the program to reduce the number of function calls to costly functions.

On a finer granularity, instruction level profiling can be carried out to provide the number and the type of processor instructions that are executed by the program at run-time. This can be used for performance tuning of program and to achieve more accurate complexity evaluation. However, the profiling data gathered is dependent on the hardware platform and the optimization level of the compiler. Unfortunately, there are few tools assisting this level of profiling. In [12], a simulator and profiler tool set based on SimpleScalar framework [22] was developed to measure the instruction level complexity. In our work, a set of profiler tools using Pin was developed to measure the instruction level complexity of the video codec [13].

2.4.3 Data Transfer and Storage Complexity Analysis

Data transfer and storage operations are other areas where complexity of the program can be evaluated. Such analyses are essential for data-dominant applications such as video multimedia applications where it has been shown that the amount of data transfer and storage operations are at least of the same order of magnitude as the amount of arithmetic operations [14]. For such application, data transfer and storage will have a dominant impact on the efficiency of the system realization.

Data transfer and storage complexity analyses have been performed for a MPEG 4 (natural) video decoder in [14] and H.264/AVC encoder/decoder in [4] using ATOMIUM [21], an automated tool. This tool measures the memory access frequency (the total number of data transfers from and to memory per second) and the

peak memory usage (the maximum amount of memory that is allocated by the source code) of the running program. Such analysis allows identifying memory related hotspots in the program, and optimization of the storage bandwidth and the storage size. However, the drawback of this tool is that it uses a “flat memory architectural mode”, and does not consider other memory hierarchy such as one or more levels of caches.

2.4.4 Platform Dependent /Independent Analysis

Generally, two types of complexity analyses can be performed: platform dependent complexity analysis and platform independent complexity analysis. The complexity evaluation using automated tools like VTune and Pin are platform dependent, specifically for general purpose CISC processor such as Pentium III and PentiumIV.

Platform independent analysis is generally preferred compared to platform dependent analysis as the target architecture on which the system will be realized is most likely different from that used to compile and run the reference implementation. Tools such as ATOMIUM and SIT [15] are developed with such a goal: to measure the complexity of a specific implementation of an algorithm independent from the architecture that is used to run the reference implementation. Besides these tools, a complexity evaluation methodology for video applications that is platform independent is also proposed in [16]. In its methodology, the platform-independent complexity metric used is the execution frequencies of core tasks executed in the program and is combined with the platform-dependent complexity data (e.g. the execution time of each core task on different processing platforms) for deriving the system complexity on various platforms. However, this approach requires

implementation cost measures for each single core task on different hardware platform to be available in the first place before the system complexity can be calculated. A similar platform-independent complexity evaluation methodology is also given in [17]. The difference lies in that for its platform-independent complexity data, it counts both the frequencies of the core tasks and the number of platform-independent operations performed by each core task. The platform-dependent data is a mapping table that identifies the number and types of execution subunits in each hardware platform that are capable of performing basic operations in parallel. As such, this methodology removes the needs for obtaining the implementation cost measure of each core task for the different platform but leads to a lower bound of the complexity measure, which is 2 - 3 factors lower than the actual complexity.

2.5 Existing Works

In most works, the complexity analyses of H.264/AVC are performed on general-purpose processor platforms. In [9], the complexity of H.26L (a designation of H.264 in the early stage of development) decoder is evaluated using two implementations and benchmark against a highly optimized H.263+ decoder. One of the implementations is a non-optimized TML-8 reference version, and the other is a highly optimized version. In their work, the execution time (measured using the ANSI C clock function) is used as the complexity metric. The complexity of CABAC which falls into the high complexity profile of H.26L was not evaluated.

In [17], the complexity of the H.264/AVC *baseline* profile decoder is analyzed using a theoretical approach. This approach allows the computational complexity of the decoder to be derived for various hardware platforms, thereby allowing classes of candidate platforms that are suitable for the actual implementation to be identified

easily. The number of computational operations is used as the complexity metric in their work. The theoretical approach is as follows: for each sub-function, its complexity is estimated using the number of basic computational operations it performs on a chosen hardware platform and its *call frequency*. The number of basic computational operations it performed on each hardware platform varies depending on the number of execution subunits available in each hardware platform. These execution subunits allow basic operations such as *ADD32, MUL16, OR, AND, Load* and *Store* to be performed in parallel. The draw-back of theoretical complexity analysis is that overhead operations such as loop overhead, flow control and boundary condition handling are not included. The run-time complexity of the decoder running on an Intel Pentium III platform is also measured using Intel VTune, an automated performance analyzer tool. Compared to the measured complexity by VTune, the estimated complexity of the H.26L decoder using the theoretical approach for the same platform is some factor lower, giving a lower-bound of the actual computational complexity of the decoder. The complexity of CABAC is not evaluated in their work as it does not fall into the baseline profile.

In [18], the performance and complexity of H.26L video encoder are given and are benchmark against the H.263+ video encoder. The complexity analysis is carried out at two levels: the application level and the kernel (or function) level. At the application level, the execution time (measured using the ANSI C clock function) is used as the complexity metric, whereas at the kernel level, the number of clock cycles (measured using Intel VTune) is used as the complexity metric.

In [4], the performance and complexity of H.264/AVC video encoder/decoder are reported. Unlike earlier works which focus on computational complexity, this work focused on data transfer and storage requirements. Such an approach proved to

be mandatory for efficient implementation of video systems due to the data dominance of multimedia applications [19][20]. To provide the support framework for automated analysis of H.264/AVC using the JM reference implementation, the C-in-C-out ATOMIUM Analysis environment has been developed. It consists of a set of kernels that provide functionalities for data transfer and storage analysis. In this work, all the coding tools have been used, including the use of B-frame, CABAC and multi-reference frame that were not evaluated in other works. Furthermore, the complexity analysis in this work explores the inter-dependencies between the coding tools and their impact on the trade-off between coding efficiency and complexity. This is unlike earlier works where the coding tool under evaluation is tested independently by comparing the performance and complexity of a basic configuration with the use of the evaluated tool to the same configuration without it.

In [12], the instruction level complexities of the H.264/AVC video encoder/decoder are measured using a simulator and profiler tool set based on the SimpleScalar framework. Similar to [4], the complexity analysis is carried out on a tool-by-tool basis using the JM reference implementation. However, it addressed the instruction level complexity in terms of arithmetic, logic, shift and control operations that were not covered in [4]. It also proposed a complexity-quality-bit-rate performance metric for examining the relative performance among all configurations used for the design space exploration.

2.6 Conclusion

In this chapter, an overview of the main functional blocks of CABAC, and a review of the encoder controls of the video encoders have been given. This is followed by a discussion on the known methodologies used in evaluating complexity

and the existing works that have been carried out for complexity evaluation of H.264/AVC. In the next chapter, the performance of CABAC, benchmarked against CAVLC for different video encoder configurations will be presented.

CHAPTER 3 PERFORMANCE ANALYSES OF ENTROPY CODING SCHEMES

3.1 Introduction

The use of new entropy coding schemes in H.264/AVC: CABAC and CAVLC is one of the reasons for its higher coding efficiency compared to earlier video standards. Both schemes adapt to the source statistics allowing bit-rates that are closer to the source entropy to be achieved. Comparatively, CABAC outperforms CAVLC in achieving higher compression.

The CABAC scheme has been reviewed in the earlier chapter. CAVLC, on the other hand is an entropy coding scheme based on variable length coding (VLC) using Exp-Golomb code and a set of predefined VLC tables. It has been reported that CABAC reduces the bit-rate up to 16% in [5] and a lower 10% in [4]. In our work, we shall validate the performance of CABAC benchmark against CAVLC using diverse range of test sequences and different combinations of coding tools.

In particular, we analyze the performance of CABAC in a H.264/AVC video encoder that does not employed rate-distortion optimization (RDO). This has yet to be reported in any work. Furthermore, we compared the coding performance between a non-RDO encoder and an RDO encoder. The reason being the use of non-normative RDO technique has a direct influence on the workload in the entropy coding stage of the video encoder. A co-evaluation of the performance-complexity of the entropy coding scheme will also be given in the next chapter. Lastly, the performance of CABAC in a constant bit-rate video encoder is also considered, using the rate-control mechanism in the JM reference software.

3.2 Performance Metrics

The performance metrics used are the bit-rate savings and the peak signal-to-noise ratio of the luminance component (Y-PSNR). The assumption made here is that similar Y-PSNR values yields approximately the same subjective spatial video quality. The chrominance components (U and V) are not used as comparison metrics because the human visual system is less sensitive to chrominance components, which will have small effects on the perceived video quality.

3.3 Implementation

Performance analyses and complexity analyses of CABAC (which will be given in the next chapter) are both conducted using JM reference software version 9.5.

All testings were carried out on an IA32 architecture using Intel Pentium III 933 MHz processor with 512 SD-RAM in a standard Linux/C environment. The on-chip L1 data and instruction caches each have a size of 16 KB, whereas the L2 caches each have a size of 512 KB. The video encoder and decoder were compiled using GNU GCC compiler with -O2 optimization option. Note that this level of optimization does not include optimization for space-speed tradeoff such as loop unrolling and function in-lining.

3.4 Test Bench Definitions

A set of fifteen QCIF and CIF video sequences have been used for the testing as given in Table 3-1. These sequences have been categorized based on the amount of motion content in them. The table also lists the bit-rates of the sequences encoded using configuration C2 with CABAC for a non-RDO encoder (as given in Table 3-2).

Table 3-1: Test sequences and their motion content classification

Sequence	QCIF	CIF	Motion Contents	QCIF Bit-rates (kbps)	CIF Bit-rates (kbps)
Akiyo	X	X	Low	80	214
Mother & Daughter	X	X	Low	89	248
Container	X	X	Low	112	407
Carphone	X		Moderate	223	
Foreman	X	X	Moderate	224	780
Soccer		X	High		1264
Stefan		X	High		1302
Coastguard	X	X	High	289	1336
Walk	X	X	High	439	1379

The classification of the video sequences is carried out by subjective evaluation. The low-motion contents test sequences have been shaded in grey, moderate-motion content test sequences in white, and high-motion contents test sequences in black. These denotations will be used throughout this work.

Sequences *Akiyo, Mother & Daughter* and *Container* are used to represent low-motion sequences while *Coastguard, Foreman* and *Walk* contain varying degrees of camera motion. The set of CIF sequences also includes two sport sequences: *Soccer* and *Stefan*, that are representative of broadcast video content with high object motions. One is a soccer game and the other is a tennis game. Most of these sequences

have identical video content in their counterpart video format, which will be used to study the effect of picture size. All sequences comprise of 300 frames.

A large number of configurations have been used in the testing. A selected set that are representatives of these configurations are shown in Table 3-2.

Table 3-2: Encoder configuration cases

	C1	C2	C3	C4	C5	C6	C7	C8
Intra 4x4	1	1	1	1	1	1	1	1
Intra 16x16	1	1	1	1	1	1	1	1
Inter modes 16x16/16x8/8x16/8x8	1	1	4	4	4	4	4	4
Sub-partition modes 8x4/4x8/4x4	0	0	0	0	3	3	3	3
Reference frame	1	1	1	5	5	5	5	5
Search Range	8	16	16	16	16	16	32	16
Hadamard	0	0	0	0	0	1	1	1
FSBMME	0	0	0	0	0	0	0	1
B frame	0	0	1	1	1	1	1	1
Slice per frame	1	1	1	1	1	1	1	1

These configurations have been ordered from C1 to C8 so that more complex coding tools are turned on progressively. This includes the use of higher number of reference frames, larger search ranges, smaller block sizes for motion estimation, Hadamard transform and full-search block-matching motion estimation (FSBMME). With no consideration to the entropy coding schemes, configurations C1 and C2 belong to the Baseline profile, whereas the remaining configurations belong to the Main profile. In this work, a GOP is defined as 10 frames, with only the first frame being an Intra (I) frame. All subsequent frames in the GOP are Inter frames. Each frame contains only one slice. For configurations C1 and C2, no *B* frames are used in the GOP. For the remaining configurations, each *P* frame is followed by a *B* frame (in encoding order).

3.5 Performance Analyses

3.5.1 Percentage bit-rate savings by CABAC

The use of CABAC advocates a reduction in bit-rate needed to encode a sequence at the same video quality. Table 3-3 gives the bit-rate savings by CABAC, benchmarked against CAVLC for some configurations using both non-RDO and RDO video encoders. The savings obtained in the remaining configurations are similar, and hence not shown.

Table 3-3: Percentage Bit-rate Savings Due to CABAC

	Non-RDO encoder				RDO encoder			
QCIF Sequences	C2	C4	C6	C8	C2	C4	C6	C8
Akiyo	3.9	4.2	4.3	4.3	4.7	4.8	4.6	4.6
Mother & Daughter	3.8	3.7	3.8	3.8	4.7	4.3	4.1	4.1
Container	4.3	4.5	4.8	4.8	4.5	4.9	4.4	4.4
Carphone	3.3	4.6	4.8	4.8	3.5	3.0	3.5	3.6
Foreman	4.8	5.3	5.4	5.3	4.9	4.4	4.7	4.7
Coastguard	7.5	8.7	8.8	8.9	6.8	7.3	7.2	7.0
Walk	4.2	6.1	6.5	6.9	4.7	4.9	5.7	5.8
CIF Sequences								
Akiyo	6.5	5.8	5.9	5.9	6.0	5.9	5.8	5.8
Mother & Daughter	6.9	5.7	5.8	5.8	7.4	7.1	6.8	6.8
Container	5.4	6.2	6.4	6.4	6.0	6.7	6.0	6.1
Foreman	7.2	7.4	7.2	7.0	6.5	6.2	6.6	6.7
Soccer	6.3	7.0	6.9	6.8	7.1	7.8	7.6	7.2
Stefan	6.3	7.3	7.8	8.3	6.4	7.7	7.3	7.4
Coastguard	8.7	9.2	9.6	9.6	7.5	8.1	8.3	8.7
Walk	5.8	7.5	7.9	8.1	6.7	7.2	7.4	7.5

Bit-rate savings between 3-9% for QCIF sequences and 5-10% for CIF sequences have been obtained for all configurations. The effect of CABAC on the coding performance is additive as the bit-rate savings obtained for the same sequence is consistence across the configurations. In addition, the bit-rate savings obtained from the non-RDO video encoder is similar to that from the RDO video encoder for

the same sequence. This implied low correlation exists between CABAC and the use of other coding tools, and between CABAC and RDO.

Other less significant observations includes the followings: bit-rate savings obtained for low-motion content sequences are generally smaller than that of high-motion content sequences, especially when more complex coding tools are used. It is also observed that for identical video content, higher bit-rate saving is obtained for the CIF sequences compared to the QCIF sequences. This indicates that bit-rate saving increases with higher level of motion contents or the use of larger picture size.

3.5.2 Percentage bit-rate savings by RDO

The RDO technique minimizes the bit-rate budget needed by the video encoder to encode a sequence for a given video quality. Table 3-4 summarizes the bit-rate savings obtained for a RDO encoder compared to that of a non-RDO encoder.

Table 3-4: Percentage Bit-rate Savings by RDO

	C2		C4		C6		C8	
QCIF Sequences	CAVLC	CABAC	CAVLC	CABAC	CAVLC	CABAC	CAVLC	CABAC
Akiyo	4.2	4.9	2.7	3.3	1.5	1.8	1.2	1.4
Mother & Daughter	6.3	7.3	4.9	5.5	4.5	4.8	4.0	4.3
Container	10.5	10.7	4.6	3.7	2.5	2.0	1.9	2.3
Carphone	6.5	6.6	7.8	7.1	8.1	6.9	8.9	8.5
Foreman	7.7	7.9	7.8	8.2	8.2	7.5	7.9	8.3
Coastguard	8.2	7.5	6.4	5.6	5.9	4.3	6.3	6.9
Walk	4.2	4.7	6.6	7.3	7.7	6.9	8.1	6.9
CIF Sequences								
Akiyo	8.4	7.9	4.7	4.1	3.5	3.4	3.5	3.3
Mother & Daughter	8.1	8.6	5.8	6.7	5.7	6.7	6.2	6.8
Container	7.1	7.7	3.5	2.1	2.6	2.2	2.3	2.8
Foreman	11.3	10.6	7.4	8.1	8.3	7.8	8.2	8.5
Soccer	7.4	7.4	7.1	6.7	7.6	7.1	7.2	7.4
Stefan	9.5	10.3	9.9	9.5	10.1	9.7	10.2	9.7
Coastguard	5.8	4.5	6.3	7.1	8.6	7.3	8.8	9.0
Walk	6.7	7.6	8.4	8.5	8.8	8.3	8.8	8.2

From 1-11% of bit-rate can be saved when using RDO in the video encoder for selecting the optimal coding modes. Its performance is minimally affected by the entropy coding scheme used. This is shown by the small variation in bit-rate saving between its use with CAVLC and its use with CABAC. This again implies a low dependency between RDO and the entropy coding schemes.

Other less significant observations include the followings: inter-dependencies between RDO and the other coding tools do exist as shown by the variation in bit-rate savings across the configurations for the same sequence. However, it is difficult to establish the exact dependency between them. What can be deduced from the data is that for low-motion content sequences, lower bit-rate savings are obtained for more complex configurations. This means that the use of RDO for bit-rate reduction becomes less effective when more complex coding tools are used in the video encoder. For complex configurations such as C6-C8, the saving in bit-rates by RDO for low-motion content sequences is much smaller than that for high-motion content sequences. This indicates that RDO achieves better performance for high-motion content sequences.

3.5.3 Overall bit-rate saving using Baseline and Main profile configurations

For an overview, the joint performance of coding tools in improving the coding efficiency is given here. Table 3-5 summarizes the bit-rate savings obtained for different combinations of entropy coding schemes with a Baseline (configuration C2) configuration and a Main profile (configuration C6 where most complex coding tools have been turned on) configuration in a non-RDO encoder and a RDO encoder. The bit-rates obtained with the collective use of the Baseline configuration with

CAVLC in a non-RDO encoder is listed and is used as a reference by which bit-rates of other coding combinations are expressed as percentage increments.

Table 3-5: Overall bit-rate savings in percentage

QCIF Sequences	Bit-rate for Baseline@ CAVLC (kbps)	Non-RDO encoder				RDO encoder			
		Baseline (C2)		Main (C6)		Baseline (C2)		Main (C6)	
		CAVLC	CABAC	CAVLC	CABAC	CAVLC	CABAC	CAVLC	CABAC
Akiyo	80	-	3.9	6.2	10.2	4.2	8.7	7.6	11.8
Mother & Daughter	89	-	3.8	7.1	10.7	6.3	10.8	11.3	14.9
Container	112	-	4.3	12.9	17.1	10.5	14.5	15.1	18.8
Carphone	224	-	3.3	11.8	16.1	6.5	9.7	19.0	21.8
Foreman	225	-	4.8	14.4	19.0	7.7	12.3	21.4	25.1
Coastguard	288	-	7.5	13.2	20.9	8.2	14.4	18.4	24.3
Walk	438	-	4.1	17.2	22.6	4.2	8.6	23.6	27.9
CIF Sequences									
Akiyo	214	-	6.5	8.6	14.0	8.4	13.9	11.8	16.9
Mother & Daughter	247	-	6.9	7.3	12.7	8.1	14.9	12.6	18.6
Container	406	-	5.4	8.0	13.8	7.1	12.7	10.3	15.7
Foreman	780	-	7.2	22.5	28.1	11.3	17.0	28.9	33.7
Soccer	1,047	-	6.3	18.8	25.1	7.4	13.3	25.0	30.5
Stefan	1,325	-	7.5	19.6	26.5	9.5	14.7	28.6	34.8
Coastguard	1,335	-	8.7	11.2	19.6	5.8	12.8	18.7	25.5
Walk	1,369	-	5.8	19.8	26.2	6.7	12.9	26.9	32.3

For the discussion in this sub-section, all bit-rate saving are made with respect to bit-rates obtained for CAVLC with the Baseline configuration in a non-RDO encoder.

The use of CABAC and Main profile configuration achieves a 10-28% bit-rate savings with a non-RDO encoder and a higher 12-35% bit-rate savings with an RDO encoder (but at the expense of higher encoder complexity, which will be given in the next chapter). The data shows that RDO improves the bit-rate savings for all encoder configurations but is found to be generally less effective for low-motion content sequences than high-motion content sequences.

Comparatively, smaller improvements in bit-rate saving by the use of Main profile configuration are obtained for low-motion content sequences than high-motion content sequences. This indicates that the use of Main profile configuration achieves better performance for high-motion content sequences.

For the Baseline profile configuration, the use of CAVLC in a RDO encoder outperforms that of CABAC in a non-RDO encoder for almost all sequences. However, with the Main profile configuration, the same observation is obtained for only some sequences. For the other sequences, the use of CABAC in a non-RDO encoder achieves better results than CAVLC in a RDO encoder. This shows that the use of more coding tools overshadow the combined effect of CAVLC and RDO.

3.5.4 Effect of CABAC on Y-PSNR at Constant Bit-Rates

In this sub-section, the effect of using CABAC in improving the coding performance at constant bit-rate is studied. The performance metric used is the Y-PSNR. Tables 3-6a and 3-6b list the increases in Y-PSNR due to CABAC when using the Main profile configuration C6 in a non-RDO encoder as well as an RDO encoder across different constant bit-rates. All Y-PSNR improvements are made with respect to the Y-PSNR values obtained for CAVLC with Main profile configuration in a non-RDO encoder at the specified constant bit-rates.

Table 3-6a: Δ Y-PSNR due to CABAC in a non-RDO encoder at different constant bit-rates

QCIF Sequences	64 kbps	128 kbps	256 kbps	CIF Sequences	256 kbps	512 kbps	1024 kbps
Akiyo	0.30	0.23	0.31	Akiyo	0.22	0.21	0.20
Mother & Daughter	0.17	0.27	0.29	Mother & Daughter	0.25	0.24	0.20
Container	0.35	0.22	0.35	Container	0.28	0.24	0.34
Carphone	0.36	0.27	0.27	Foreman	0.44	0.32	0.32
Foreman	0.40	0.27	0.29				
Coastguard	0.51	0.33	0.42	Coastguard	0.65	0.43	0.43
Walk	0.47	0.40	0.43	Walk	0.71	0.51	0.44
				Soccer	0.53	0.46	0.43
				Stefan	0.34	0.37	0.47

Table 3-6b: Δ Y-PSNR due to CABAC in an RDO encoder at different constant bit-rates

QCIF Sequences	64 kbps	128 kbps	256 kbps	CIF Sequences	256 kbps	512 kbps	1024 kbps
Akiyo	0.92	0.77	0.86	Akiyo	0.70	0.69	0.59
Mother & Daughter	0.59	0.74	0.77	Mother & Daughter	0.72	0.67	0.52
Container	0.76	0.61	0.68	Container	0.57	0.53	0.65
Carphone	0.68	0.60	0.60	Foreman	0.80	0.68	0.63
Foreman	0.80	0.59	0.58				
Coastguard	0.71	0.56	0.71	Coastguard	0.89	0.67	0.74
Walk	1.03	0.77	0.80	Walk	1.25	0.91	0.81
				Soccer	0.87	0.80	0.92
				Stefan	0.65	0.77	0.85

The results show that at constant bit-rates, the use of CABAC improves the video quality by a negligible amount of 0.1-0.8 dB in a non-RDO encoder. Even with the collective use of RDO, only a small improvement of 0.5-1.2 dB has been obtained in the RDO encoder. This indicates that CABAC is less attractive as a tool for improving video quality at constant bit-rate than as a compression tool.

3.6 Conclusion

In this chapter, performance analyses of CABAC and RDO have been given. Benchmark against the performance of a Baseline profile configuration of

H.264/AVC, the advanced coding tools achieves saving in bit-rates up to 35%. A tool-by-tool analysis shows that CABAC alone saves 3-10% in bit-rates while RDO, another 1-11%. It is observed that these tools achieved better performance in high-motion content sequences. At constant bit-rates, the collective use of CABAC and RDO however is not effective in improving the video quality, achieving a gain of at most 1 dB. The complexity of CABAC is assessed and presented in the next chapter.

CHAPTER 4 COMPLEXITY ANALYSES

4.1 Introduction

In this chapter, the complexity analysis of CABAC is conducted using Pin and PAPI tools [24][25]. The complexity metrics are the computational complexity, the data transfer complexity and the memory usage.

Analyses are carried out at the entropy coder level and the video encoder level. The entropy coder is a generic term that refers to either the CAVLC or CABAC entropy coding functional block of the top-level video encoder. At the entropy coder level, the additional workload required by the entropy coder when CAVLC is replaced by CABAC, is measured for different configurations in both non-RDO and RDO encoders. At the top-level video encoder, the effect of using CABAC on the overall complexity of the video encoder is observed. The workload of the entropy coder is also partitioned based on their functional sub-blocks for further analyses so as to identify any critical sub-block or possible bottleneck. Besides the encoder, the complexity of the decoder is also being addressed. To achieve an exhaustive analysis of CABAC, a wide genre of video contents has been used as test sequences.

Lastly, a co-evaluation of the performance-complexity of CABAC is given, where its optimal use is recommended by considering the tradeoff between its performance and complexity.

4.2 Complexity Metric Definitions

4.2.1 Computational Complexity

The computational complexity of a functional block is given in terms of the number of *complete instruction set computer* (CISC) instructions it executed. The instructions executed by the entropy coding functional block are referred as *entropy* instructions in this work.

4.2.2 Data Transfer Complexity and Memory Usage

The data transfer complexity of a functional block is given in terms of the number of memory accesses it performed for memory read or memory write operations. As for memory usage, because dynamic memories are allocated and freed at different stages of the encoding and decoding process, peak memory usage is used as the memory requirements needed by the encoder/decoder.

4.3 Computational Complexity

In this section, the computational complexity of CABAC when used in both non-RDO encoder and RDO encoder are analyzed, and are compared with reference to CAVLC. All computational complexity measurements are expressed in terms of the average number of instructions (or entropy instructions) executed per GOP.

4.3.1 Effect of CABAC on the entropy coder

The use of CABAC requires more computation to be performed compared to CAVLC. Table 4-1 shows the percentage increase in computational complexity of the entropy coder when CABAC replaced CAVLC across different configurations.

Table 4-1: Percentage increase in computational complexity of the entropy coder due to CABAC

QCIF Sequence	Non-RDO Encoder				RDO Encoder			
	Baseline	Main			Baseline	Main		
	C2	C4	C6	C8	C2	C4	C6	C8
Akiyo	16	19	20	20	19	21	21	21
Mother & Daughter	17	24	26	26	17	20	20	20
Container	21	23	24	24	29	30	30	30
Carphone	18	26	30	30	24	24	26	26
Foreman	16	27	30	29	26	27	28	27
Coastguard	19	25	28	29	33	35	34	34
Walk	18	25	27	27	29	28	31	30
CIF Sequence								
Akiyo	17	19	20	20	12	15	15	15
Mother & Daughter	16	23	24	24	10	13	13	13
Container	19	20	20	20	26	27	28	28
Foreman	15	23	26	26	21	23	23	23
Soccer	13	18	21	21	10	16	19	19
Stefan	22	22	22	22	26	25	22	22
Coastguard	17	23	25	25	29	34	31	31
Walk	15	19	23	23	22	22	24	24

From the data, CABAC increases the computational complexity of the entropy coder by 13-30% for a non-RDO encoder and 10-35% for an RDO encoder. It is observed that smaller percentage increments are obtained with the use of Baseline profile configuration (C2) compared to Main profile configurations (C4, C6 & C8). Smaller percentage increments are also obtained for low-motion content sequences than high-motion content sequences.

4.3.2 Effect of RDO and complex coding tools on the entropy coder

The use of RDO as the video encoder control significantly increases the computational complexity of the entropy coder, whereas the use of more complex coding tools has a smaller effect on its computational complexity. Table 4-2 gives the computational complexity of the CABAC entropy coder in a non-RDO encoder and

the corresponding complexity increment factor of the CABAC entropy coder in an RDO encoder, using both Baseline profile configuration and Main profile configuration. The complexity increment factor is given by normalizing the average entropy instruction counts executed by the RDO encoder with that of the non-RDO encoder for the same configuration. Similar results have been obtained for the CAVLC entropy coder and are not shown.

Table 4-2: Computational complexity of CABAC entropy coder in a non-RDO encoder and a RDO encoder

QCIF Sequence	Non-RDO encoder		RDO Encoder	
	Baseline (C2)	Main (C6)	Baseline (C2)	Main (C6)
	Entropy Instruction Count ($x10^3$)		Increment Factor	
Akiyo	6,125	6,120	284	298
Mother & Daughter	7,266	7,169	226	242
Container	7,817	7,056	273	315
Carphone	16,013	15,496	124	136
Foreman	16,240	15,353	128	143
Coastguard	19,338	18,406	126	141
Walk	28,792	27,000	79	92
CIF Sequence				
Akiyo	18,837	18,360	308	336
Mother & Daughter	23,084	22,630	243	265
Container	30,690	29,532	266	290
Foreman	58,305	50,236	130	161
Soccer	77,688	66,853	94	117
Stefan	74,219	63,142	98	127
Coastguard	91,822	88,150	106	118
Walk	95,335	86,272	83	100

The use of RDO increases the computational complexity of the CABAC entropy coder tremendously between 80 and 340 times. This means that the use of RDO triggered a huge workload for the entropy coder and creates a bottleneck in it. This necessitates the use of a CABAC hardware accelerator if real-time requirements are to be met with an RDO encoder.

Comparatively, changes in computational complexity of the entropy coder across the configurations are small for each video encoder. This can be seen by

comparing the data between the Baseline and the Main profile configurations, which shows that the use of Main profile configuration reduces the computational complexity of the entropy coder up to 15% in a non-RDO encoder and increases its computational complexity between 4% and 10% in a RDO encoder. This means that the choice of coding tools used (with the exception of the entropy coding scheme) in the video encoder has a small effect on the computational complexity of the entropy coder.

Encoding high-motion content sequences requires a computational complexity of the entropy coder up to 5 times more for a non-RDO encoder and up to 1.7 times or 70% more for a RDO encoder as compared to low-motion content sequences. This indicates that the computational complexity of the entropy coder increases with higher motion contents in the sequence for the same encoder configuration.

4.3.3 Overall computational complexity of the entropy coder

Table 4-3 shows the relative computational complexities of the entropy coder for different combination of entropy coding schemes with Baseline profile and Main profile configurations in a non-RDO encoder and an RDO encoder. All comparisons are made with respect to the non-RDO encoder using Baseline profile configuration with CAVLC, of which the magnitudes of entropy instructions per GOP are also given. Results have been given with accuracy up two decimal places in order to show the finer differences among the values.

Table 4-3: Computational complexities of entropy coder in different combinations of entropy coding schemes and configurations for non-RDO and RDO encoders

		Non-RDO encoder				RDO encoder			
		Baseline (C2)		Main (C6)		Baseline (C2)		Main (C6)	
QCIF Sequences	Entropy Instruction Count ($x10^3$)	CAVLC	CABAC	CAVLC	CABAC	CAVLC	CABAC	CAVLC	CABAC
Akiyo	5,280	1.00	1.16	0.97	1.16	276	329	285	346
Mother & Daughter	6,211	1.00	1.17	0.92	1.15	225	264	233	279
Container	6,463	1.00	1.21	0.88	1.09	256	330	263	344
Carphone	13,572	1.00	1.18	0.88	1.14	118	146	123	155
Foreman	14,009	1.00	1.16	0.84	1.10	117	148	122	157
Coastguard	16,255	1.00	1.19	0.88	1.13	113	150	119	159
Walk	24,404	1.00	1.18	0.87	1.11	72	93	78	102
CIF Sequences									
Akiyo	16,105	1.00	1.17	0.95	1.14	321	360	333	383
Mother & Daughter	19,908	1.00	1.16	0.92	1.14	257	282	267	302
Container	25,799	1.00	1.19	0.95	1.15	252	317	260	332
Foreman	50,701	1.00	1.15	0.79	0.99	124	150	130	160
Soccer	68,756	1.00	1.13	0.80	0.97	96	106	95	113
Stefan	74,198	1.00	1.22	0.85	1.04	72	98	88	108
Coastguard	78,482	1.00	1.17	0.90	1.12	96	124	102	133
Walk	82,903	1.00	1.15	0.85	1.04	78	95	84	104

The data provides an overview of the possible variations in computational complexity of the entropy coder due to the collective use of different video coding tools in H.264/AVC for different type of sequences.

The use of CABAC and the Main profile configuration increases the computational complexity of the entropy coder in a non-RDO encoder by small factors between 0.97 and 1.16. But the use of RDO alone increases the computational complexity of the entropy coder by more than an order of magnitude. As a result of the collective use of RDO, CABAC and Main profile configuration, the entropy coder's computational complexity in a RDO encoder using CABAC and Main profile configuration is 100 to 390 times higher than that of a non-RDO encoder which uses CALVC and Baseline profile configuration. This indicates that CABAC hardware accelerator might be needed in a RDO encoder to speed up entropy coding and to meet any real-time requirements.

4.3.4 Overall computational complexity of the video encoder

Besides the entropy coding stage, the computational complexities in the other encoding stages of the video encoder also increases due to the use of RDO and more complex coding tools. As such, the impact of the computational complexity of the entropy coder has a lesser effect on the video encoder's computational complexity. Table 4-4 shows the relative computational complexities of the video encoder for the different combinations of entropy coding schemes with Baseline and Main profile configuration, and RDO. In tandem with the analyses given in the earlier subsection, all comparisons are made with respect to the non-RDO encoder using Baseline profile with CAVLC, of which the magnitudes of total instructions executed per GOP have also been given. Results have been given with accuracy to three decimal places in order to show the finer differences among the values.

Table 4-4: Computational complexities of the non-RDO encoder and the RDO encoder using different combinations of entropy coding schemes and configurations

QCIF Sequence	Instruction Count ($x10^3$)	Non-RDO encoder				RDO Encoder			
		Baseline (C2)		Main (C6)		Baseline (C2)		Main (C6)	
		CAVLC	CABAC	CAVLC	CABAC	CAVLC	CABAC	CAVLC	CABAC
Akiyo	705,203	1.000	1.005	7.878	7.884	6.214	6.693	13.405	13.906
Mother & Daughter	719,564	1.000	1.005	8.798	8.804	6.002	6.301	14.129	14.633
Container	704,361	1.000	1.006	8.024	8.030	6.539	7.224	13.931	14.727
Carphone	771,711	1.000	1.007	9.947	9.955	5.932	6.495	15.369	15.956
Foreman	780,657	1.000	1.007	10.86	10.868	5.933	6.532	16.221	16.943
Coastguard	769,952	1.000	1.008	11.891	11.899	6.303	7.140	17.603	18.544
Walk	844,179	1.000	1.009	12.716	12.726	5.786	6.335	17.827	18.527
CIF sequence									
Akiyo	2,896,128	1.000	1.002	7.836	7.838	5.907	6.213	13.222	13.521
Mother & Daughter	2,949,502	1.000	1.002	8.698	8.701	5.848	6.157	13.945	14.220
Container	2,908,123	1.000	1.003	8.59	8.593	6.493	7.133	14.406	15.175
Foreman	3,196,301	1.000	1.003	11.284	11.288	5.998	6.307	16.565	17.093
Soccer	3,332,966	1.000	1.004	13.013	13.018	5.642	6.085	17.981	18.342
Stefan	3,380,364	1.000	1.004	12.991	13.013	5.604	6.107	17.229	17.908
Coastguard	3,228,994	1.000	1.005	12.295	12.301	6.263	7.001	18.126	18.966
Walk	3,431,219	1.000	1.005	12.673	12.678	5.668	6.327	17.563	18.113

From the viewpoint of the non-RDO encoder, the use of CABAC has a negligible effect on its computational complexity across all configurations. This is shown by the insignificant difference in computational complexity in both Baseline profile configuration and the Main profile configuration. This suggests that CABAC should always be used in a non-RDO encoder in order to take advantage of the coding gain. (The effect of CABAC on the RDO encoder's computational complexity will be given later in another set of data.)

For both entropy coding schemes, the use of Main profile configuration increases the computational complexity of the non-RDO encoder by 7 to 13 times. On the other hand, using RDO as the optimization control results in the computational complexity of the RDO encoder to be higher than that of the non-RDO encoder by about 6 times for Baseline profile configuration, and up to 2 times for Main profile configuration. As such, the computational complexity of the RDO encoder using CABAC and Main profile configuration is 13 to 19 times higher than that of a non-RDO encoder using CAVLC and Baseline profile as a result of the collective use of RDO, CABAC and Main profile configuration.

Unlike the non-RDO encoder, the use of CABAC increases the computational complexity of the RDO encoder but the degree of impact depends on the configuration that is used in the video encoder. Table 4-5 shows the percentage increase in computational complexity of the video encoder when CABAC replaced CAVLC.

Table 4-5: Percentage increase in computational complexity of the RDO encoder due to CABAC

QCIF Sequences	Baseline	Main	Main	Main
	C2	C4	C6	C8
Akiyo	6.5	4.2	3.7	0.07
Mother & Daughter	5.0	3.9	3.5	0.14
Container	10.8	7.2	5.8	0.09
Carphone	8.5	6.4	3.9	0.22
Foreman	10.2	7.8	4.3	0.28
Coastguard	12.7	9.7	6.5	0.45
Walk	10.5	7.2	3.9	0.37
CIF Sequence				
Akiyo	5.1	3.6	2.3	0.23
Mother & Daughter	5.2	3.1	2.2	0.31
Container	10.9	6.8	4.9	0.44
Foreman	6.8	4.5	3.0	0.51
Soccer	7.1	4.9	2.2	0.73
Stefan	8.9	5.4	4.1	0.97
Coastguard	12.9	7.9	4.4	1.03
Walk	12.5	7.1	3.4	1.08

For the RDO encoder, replacing CALVC with CABAC increases its computational complexity by 5-13% for the Baseline profile configuration (configuration C2). The impact is less prominent with the use of Main profile configuration as the use of more complex coding tools leads to a much higher computational complexity of the video encoder. For instance, only 2-7% increases in the computational complexity due to the use of CABAC have been obtained for Main profile configuration, C6. The increment is even less than 2% for configuration C8, which employs full-search motion estimation technique. By considering only the computational complexity, the results do encourage the use of CABAC in the RDO encoder.

4.3.5 Video Decoder

The computational complexity of the decoder is much lower than the encoder. It is observed that the use of RDO has little impact on the computational complexity

of the decoder whereas the use of CABAC, contrary to the fact that CABAC is more computationally intensive than CAVLC, actually yields a reduction in the decoder's computational complexity. Table 4-6 shows the percentage reduction in computational complexity of the decoder due to the use of CABAC for both Baseline profile and Main profile configurations.

Table 4-6: Percentage reduction in computational complexity of the video decoder due to CABAC

	RDO Off		RDO on	
QCIF Sequence	Baseline (C2)	Main (C6)	Baseline (C2)	Main (C6)
Akiyo	2.3	1.6	2.2	1.5
Mother & Daughter	3.2	1.9	3.1	2.0
Container	2.8	1.8	2.2	1.5
Carphone	6.2	3.6	5.7	3.6
Foreman	6.3	3.1	6.0	3.4
Coastguard	8.7	5.7	7.7	5.2
Walk	9.9	6.3	9.6	6.6
CIF Sequence				
Akiyo	1.7	1.2	1.5	1.0
Mother & Daughter	2.2	1.3	2.1	1.2
Container	3.1	2.5	2.1	2.5
Foreman	5.6	2.9	5.1	2.3
Coastguard	10.2	7.1	10.0	6.6
Stefan	10.4	6.6	10.3	6.1
Soccer	7.6	4.6	7.2	4.1
Walk	8.7	5.5	8.5	5.0

From the data, the use of CABAC results in a 2-11% reduction in the computational complexity of the decoder compared to CAVLC. CAVLC requires a higher computation because of the need to search through the code tables for decoding the syntax element. It is observed that larger percentage reduction in decoder's computational complexity can be obtained for high motion content sequences compared to low motion content sequences.

4.4 Data Transfer Complexity

In this section, the data transfer complexities of CABAC in both non-RDO encoder and RDO encoder are analyzed and compared with CAVLC. The analyses are carried out in two parts. The first part assumes a "flat memory architectural" model and the second part assumes a two-level memory model with the use of a 16KB L1 data cache. All data complexity measurements are expressed as the average number of memory access per GOP.

4.4.1 Effect of CABAC on the entropy coder

The use of CABAC requires the entropy coder to access the memory more frequently as compared to CAVLC. Table 4-7 shows the percentage increase in data transfer complexity of the entropy coder when CABAC replaced CAVLC across the different configurations.

Table 4-7: Percentage increase in data transfer complexity of the entropy coder due to CABAC

QCIF Sequence	Non-RDO Encoder				RDO Encoder			
	Baseline	Main			Baseline	Main		
	C2	C4	C6	C8	C2	C4	C6	C8
Akiyo	25.	29	30	31	108	111	108	108
Mother & Daughter	26	33	37	37	108	109	108	109
Container	31	32	35	35	112	112	111	114
Carphone	28	37	43	44	108	107	107	107
Foreman	26	36	43	41	108	108	108	108
Coastguard	30	37	40	42	109	110	108	109
Walk	29	35	40	41	108	108	105	110
CIF Sequence								
Akiyo	25	27	29	28	115	112	111	111
Mother & Daughter	24	31	34	36	113	110	110	110
Container	29	32	30	29	115	111	111	112
Foreman	24	32	37	37	112	109	108	109
Soccer	27	34	37	38	111	112	106	109
Stefan	22	28	32	35	109	109	104	107
Coastguard	25	29	34	37	111	109	106	106
Walk	25	28	29	31	114	110	111	112

The data shows that the use of CABAC instead of CAVLC increases the data transfer complexity of the entropy coder by 20-44% for a non-RDO encoder and a higher 104-115% for a RDO encoder. The higher percentage increase obtained for the RDO encoder is expected due to its frequent need to store, load and reset the coding states of the arithmetic coder as well as the context models. This ensures that sequential arithmetic coding process can be carried out correctly.

For a non-RDO encoder, smaller percentage increments due to the use of CABAC are obtained for Baseline profile configuration (configuration C2) compared to Main profile configurations (configurations C4, C6, & C8). However, no similar observations are obtained for the RDO encoder. Rather, the use of CABAC results in consistence percentage increase in the data transfer complexity of the entropy coder across all configurations.

4.4.2 Effect of RDO and complex coding tools on the entropy coder

The use of RDO has a large influence on the data transfer complexity of the entropy coder whereas the use of complex coding tools has a small effect on it. Table 4-8 gives the data computational complexity of the CABAC entropy coder in a non-RDO encoder and the corresponding complexity increment factor of the CABAC entropy coder in an RDO encoder, using both Baseline profile and Main profile configurations. The complexity increment factor is given by normalizing the number of memory access performed by the RDO encoder with that of the non-RDO encoder for the same configuration.

Table 4-8: Data transfer complexity of CABAC entropy coder in a non-RDO encoder and an RDO encoder

QCIF Sequence	Non-RDO encoder		RDO encoder	
	Baseline	Main	Baseline	Main
	Memory access Count ($x10^3$)		Increment Factor	
Akiyo	4,712	4,678	526	545
Mother & Daughter	5,551	5,481	433	453
Container	5,989	5,401	466	530
Carphone	12,270	11,832	218	235
Foreman	12,481	11,779	220	242
Coastguard	14,903	14,102	203	223
Walk	22,228	20,693	131	148
CIF Sequence				
Akiyo	14,378	13,992	654	683
Mother & Daughter	17,542	17,151	528	550
Container	23,601	22,524	478	508
Foreman	44,851	38,302	243	290
Soccer	59,568	51,273	178	212
Stefan	67,032	61,427	181	217
Coastguard	70,310	67,618	178	192
Walk	73,522	65,771	151	175

The use of RDO increases the data transfer complexity of the CABAC entropy coder tremendously by 130 to 700 times.

Comparatively, the choice of configuration used has a smaller effect on the data transfer complexity of the entropy coder. This can be seen by comparing the data between the Baseline and the Main profile configurations, which shows a reduction in the data transfer complexity of the entropy coder up to 18% in a non-RDO encoder and an increase in its data transfer complexity up to 6% in an RDO encoder. The reason for the reduction in data transfer complexity of the non-RDO encoder when Main profile configuration is used over the Baseline profile configuration is because the use of Main profile configuration yields better macroblock prediction. This results in a smaller residual prediction signal, which leads to less number of residual syntax elements to be entropy coded. On the other hand, this is not so for the RDO encoder because it cycles through all candidate prediction modes. As a result, more syntax

elements need to be coded in the Main profile configuration than the Baseline profile configuration.

As compared to low-motion content sequences, encoding high-motion content sequences increase the data transfer complexity of the entropy coder up to a factor of 5.1 for a non-RDO encoder but a lower factor of 1.2 or a 20% increase for a RDO encoder. Although this indicates that the data transfer complexity of the entropy coder increases with higher motion contents in the sequence for the same encoder configuration, it also shows a weaker influence the video content has on the data transfer complexity of the entropy coder in a RDO encoder.

4.4.3 Overall data transfer complexity of the entropy coder

Table 4-9 shows the relative data transfer complexities of the entropy coder for different combination of the entropy coding schemes with Baseline profile and Main profile configurations in a non-RDO encoder and an RDO encoder. All comparisons are made with reference to the non-RDO encoder using Baseline profile configuration with CAVLC, of which the magnitudes of memory access per GOP are also given. Results have been given with accuracy up two decimal places in order to show the finer differences among the values.

Table 4-9: Data transfer complexities of entropy coder in different combinations of entropy coding schemes and configurations for non-RDO and RDO encoders

	Entropy Memory Access Count ($x10^3$)	Non-RDO encoder				RDO encoder			
		Baseline (C2)		Main (C6)		Baseline (C2)		Main (C6)	
QCIF Sequence		CAVLC	CABAC	CAVLC	CABAC	CAVLC	CABAC	CAVLC	CABAC
Akiyo	3,746	1.00	1.26	0.96	1.25	317	659	326	679
Mother & Daughter	4,412	1.00	1.26	0.90	1.24	261	543	270	562
Container	4,566	1.00	1.31	0.88	1.18	289	610	296	626
Carphone	9,609	1.00	1.28	0.86	1.23	134	278	140	289
Foreman	9,923	1.00	1.26	0.83	1.19	133	276	138	287
Coastguard	11,463	1.00	1.30	0.87	1.23	126	264	132	275
Walk	17,191	1.00	1.29	0.86	1.20	81	169	87	178

CIF Sequence									
Akiyo	11,470	1.00	1.28	0.96	1.24	381	819	394	832
Mother & Daughter	14,184	1.00	1.26	0.92	1.23	306	653	316	665
Container	18,252	1.00	1.31	0.96	1.25	287	617	296	626
Foreman	36,043	1.00	1.26	0.78	1.08	143	302	148	308
Soccer	49,039	1.00	1.23	0.80	1.06	103	216	108	222
Stefan	53,228	1.00	1.26	0.97	1.17	93	195	99	203
Coastguard	55,474	1.00	1.28	0.90	1.23	107	226	113	234
Walk	58,846	1.00	1.26	0.84	1.13	89	189	95	196

The variation in data transfer complexity of the entropy coder due the use of different combination of entropy coding schemes, encoder configurations and RDO behaves similarly to its computational complexity.

The use of CABAC and Main profile configuration increases the data transfer complexity of the entropy coder in a non-RDO encoder by about 20%. But the use of RDO alone increases the data transfer complexity of the entropy coder by more than an order of magnitude. As such, the data transfer complexity of the entropy coder in a RDO encoder using CABAC and Main profile configuration is 170 to 840 times higher than that of a non-RDO encoder using CAVLC and Baseline profile configuration.

For the RDO encoder, the number of memory access performed by its entropy coder approximately doubled with the use of CABAC compared to CAVLC. This shows that data transfer have a dominant impact on the CABAC entropy coder when RDO is used. Further analysis is carried out by breaking down the data transfer operations performed by the entropy coder into memory read and memory write operations. It is consistently observed that memory read operations are approximately 50% higher than memory write operations across all configurations and video source. This suggests that redesign of the memory interface such as the memory read and write circuitries and bus widths for the CABAC entropy coder might be helpful to reduce latency.

4.4.4 Overall data transfer complexity of the video encoder

Table 4-10 shows the relative data transfer complexities of the video encoder for different combinations of entropy coding schemes with Baseline and Main profile configurations, and RDO. All comparisons are made with reference to the non-RDO encoder using Baseline profile with CAVLC, of which the magnitudes of memory access per GOP have also been given. Results have been given with accuracy to three decimal places to show the finer differences among the values

Table 4-10: Data transfer complexities of the non-RDO encoder and the RDO encoder using different combinations of entropy coding schemes and configurations

QCIF Sequence	Encoder Memory Access ($x10^3$)	Non-RDO encoder				RDO encoder			
		Baseline (C2)		Main (C6)		Baseline (C2)		Main (C6)	
		CAVLC	CABAC	CAVLC	CABAC	CAVLC	CABAC	CAVLC	CABAC
Akiyo	444,013	1.000	1.006	8.352	8.361	6.825	9.747	14.698	17.773
Mother & Daughter	453,127	1.000	1.006	9.256	9.265	6.612	9.326	15.343	18.246
Container	443,566	1.000	1.007	8.452	8.461	7.248	10.596	15.106	18.621
Carphone	484,471	1.000	1.009	10.299	10.312	6.551	9.417	16.477	19.443
Foreman	489,769	1.000	1.009	11.235	11.248	6.658	9.562	17.382	20.401
Coastguard	484,484	1.000	1.011	12.215	12.228	6.935	10.234	18.701	22.195
Walk	526,040	1.000	1.013	12.999	13.015	6.443	9.258	18.853	21.972
CIF Sequence									
Akiyo	1,797,774	1.000	1.003	8.362	8.366	6.684	8.958	14.670	17.443
Mother & Daughter	1,828,642	1.000	1.003	9.244	9.248	6.566	8.723	15.398	18.145
Container	1,806,352	1.000	1.004	9.090	9.096	7.214	10.064	15.818	19.376
Foreman	1,977,606	1.000	1.006	11.803	11.811	6.655	9.033	17.946	20.939
Soccer	2,047,461	1.000	1.007	13.461	13.470	6.348	8.576	19.243	22.016
Stefan	2.108,883	1.000	1.009	13.164	13.176	6.709	9.124	18.858	20.439
Coastguard	2,004,411	1.000	1.009	12.776	12.789	6.963	9.797	19.403	22.921
Walk	2,108,836	1.000	1.009	13.053	13.064	6.232	8.942	18.944	21.289

From the viewpoint of the non-RDO encoder, the use of CABAC has negligible effect on its data transfer complexity across all configurations. This is shown by the insignificant difference in data transfer complexity in both Baseline profile and Main profile configurations. This reaffirms that the use of CABAC is always recommended for a non-RDO encoder. (The effect of CABAC on the data transfer complexity of the RDO encoder will be given)

The use of Main profile configuration increases the data transfer complexities of the non-RDO video encoder by 8 to 14 times for both entropy coding schemes. Due to the use of RDO, the data transfer complexity of the RDO encoder is higher than that of the non-RDO encoder by 6 to 10 times for the Baseline profile configuration, and about 2 times for the Main profile configuration. The combined effect of using RDO, CABAC and Main profile configuration results in the data transfer complexity of the RDO encoder to be 17 to 23 times higher than a non-RDO encoder using CAVLC with Baseline profile configuration.

The use of CABAC over CAVLC in a RDO encoder increases its data transfer complexity but the degree of impact depends on the configuration that is used in the video encoder. Table 4-11 shows the percentage increase in data transfer complexity of the video encoder when CABAC replaced CAVLC.

Table 4-11: Percentage increase in the data transfer complexity of the RDO encoder due to CABAC

QCIF Sequences	Baseline	Main		
	C2	C4	C6	C8
Akiyo	42.1	28.8	20.9	14.6
Mother & Daughter	41.5	27.4	19.3	13.7
Container	46.0	31.0	23.2	15.9
Carphone	43.5	29.2	18.6	13.2
Foreman	43.9	28.9	17.8	13.0
Coastguard	46.8	26.5	18.6	13.6
Walk	44.7	27.5	16.3	12.2
CIF Sequence				
Akiyo	34.3	25.9	19.8	10.2
Mother & Daughter	33.5	24.7	18.1	9.7
Container	39.1	27.2	21.8	11.6
Foreman	36.6	23.5	16.6	7.8
Soccer	35.0	21.4	14.5	6.5
Stefan	35.8	21.0	14.9	6.6
Coastguard	40.1	27.8	17.6	8.2
Walk	36.8	21.1	12.4	6.3

The use of CABAC increases the data transfer complexity of the RDO encoder by 33-47% for the Baseline profile configuration whereas lower percentage increments are obtained for Main profile configurations. For instance, the use of CABAC results in a 12-24% increase in the RDO encoder's data transfer complexity for Main profile configuration C6. The use of Main profile configurations reduces the impact of CABAC on the RDO encoder's data transfer complexity as the use of more complex coding tools leads to a much higher data transfer complexity of the RDO encoder. Nonetheless, the additional data transfer complexity incurred by the video encoder due to the use of CABAC is still high. This discourages the use of CABAC in the RDO encoder.

Comparing the change in computational complexity with the change in data transfer complexity of the video encoder across all configurations, the results show that the use of CABAC results in higher increase in data transfer complexity as compared to the increase in computational complexity. This shows that CABAC is dominated more by data transfer complexity compared to computational complexity. Thus, the bottleneck can be at the memory interface.

4.4.5 Effect of using a 16KB of L1 data cache

The use of a data cache significantly reduces the number of accesses to memory which improves the average latency of data transfer. Table 4-12 gives the percentage reduction in data transfer complexity of the RDO encoder due to the use of a 16KB L1 data cache for different combination of entropy coding schemes with Baseline profile and Main profile configurations.

Table 4-12: Reduction in average memory access by the RDO encoder per GOP due to 16KB L1 data cache

QCIF sequence	Percentage (%)			
	Baseline		Main	
	CAVLC	CABAC	CAVLC	CABAC
Akiyo	99.5	94.7	99.3	96.4
Mother & Daughter	99.5	94.6	99.3	96.6
Container	99.5	95.1	99.3	96.6
Carphone	99.5	94.9	99.3	96.9
Foreman	99.5	95.0	99.4	97.1
Coastguard	99.5	95.2	99.4	97.2
Walk	99.5	95.2	99.4	96.2
CIF sequence				
Akiyo	99.4	94.4	99.2	96.3
Mother & Daughter	99.4	94.3	99.3	96.5
Container	99.4	94.8	99.3	96.6
Foreman	99.4	94.7	99.3	97.1
Soccer	99.4	94.6	99.4	97.3
Stefan	99.4	95.0	99.4	97.3
Coastguard	99.4	94.8	99.4	97.4
Walk	99.4	94.4	99.2	96.3

The use of a data cache reduces the average number of memory access per GOP in the entropy coder by more than 99% for CAVLC and between 94-98% for CABAC. The large reduction in memory access means a lower data transfer complexity of the video encoder as the time to access the cache is much faster than access to memory.

4.4.6 Video decoder

Similar to the computational complexity of the decoder, the use of RDO has negligible impact on the decoder's data transfer complexity whereas the use of CABAC over CAVLC results in a reduction in its data transfer complexity. Table 4-13 shows the percentage reduction in decoder's data transfer complexity due to the use of CABAC for Baseline profile and Main profile configurations.

Table 4-13: Percentage reduction in data transfer complexity of the video decoder due to CABAC

	RDO Off		RDO on	
QCIF Sequence	Baseline (C2)	Main (C6)	Baseline (C2)	Main (C6)
Akiyo	4.6	3.6	4.4	3.5
Mother & Daughter	5.4	3.7	5.1	3.5
Container	5.7	4.5	5.7	4.5
Carphone	10.0	6.3	10.3	6.5
Foreman	9.9	5.2	10.1	5.8
Coastguard	12.9	8.7	13.2	9.1
Walk	16.0	10.1	16.3	10.8
CIF Sequence				
Akiyo	3.1	2.4	3.0	2.4
Mother & Daughter	3.6	2.5	3.6	2.5
Container	5.4	4.9	5.5	4.9
Foreman	8.8	4.9	9.1	5.2
Soccer	11.5	7.1	12.1	7.8
Stefan	12.8	10.0	13.5	10.7
Coastguard	14.8	10.2	15.2	10.7
Walk	13.4	8.7	14.0	9.3

From the data, we saw a 3-17% reduction in the data transfer complexity of the decoder due to the use of CABAC. The lower computational complexity and lower data transfer complexity at the video decoder when CABAC is used in place of CAVLC, suggests that CABAC should be selected for offline applications in which coding time and power consumption at the video encoder are not critical but where the communication bandwidth and battery life are limited in the video decoder such as portable devices.

4.5 Memory Usage

The memory usage by the encoder / decoder comprises of three parts: memory for storing the binary codec executable and library, stack memory and dynamic memory. The memory used for the storage is pre-determined by the compiler in which the GCC compiler has been used in this work. On our Linux platform, the encoder

executable has a memory size of 0.6MB while the decoder has a memory size of 0.35MB. A total memory size of 2.2MB is needed for storing all the video codec executables and the necessary libraries.

To cater to the diverse range of encoder configurations and video source, dynamic memory is allocated and freed as needed during a coding session. The peak memory usage depends on the configuration used and the encoding frame size, but is independent of the video contents and the use of RDO as the encoder control. Figure 4.1 shows the peak memory usage during a coding session for different configurations with QCIF and CIF sequences.

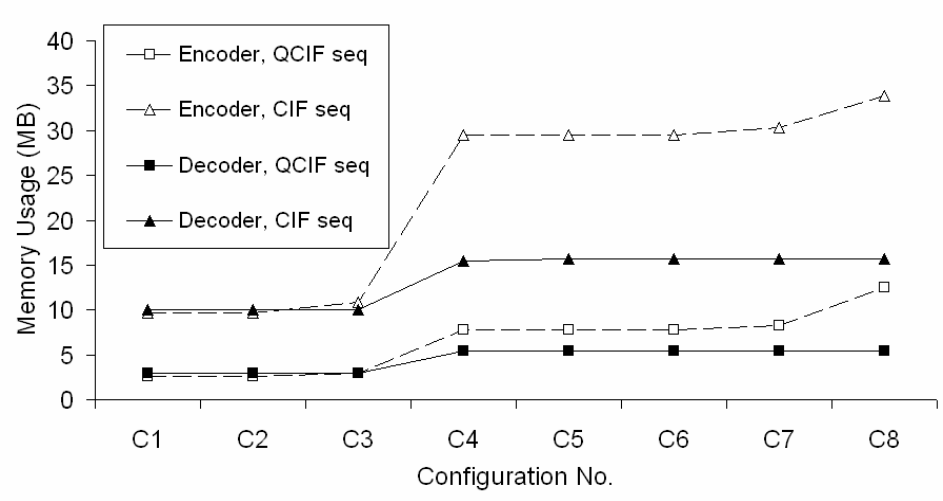


Figure 4.1: Instruction set architecture of entropy instruction executed by the CABAC entropy coder

The peak memory usage of the video encoder lies between 2–13 MB for QCIF sequences and between 9–34 MB for CIF sequences. The use of some complex coding tools (higher number of reference frames, complex motion estimation techniques, etc.) varies the buffer requirements and increases the peak memory usage of the video encoder. Compared to the Baseline profile configuration C2, the use of

Main profile configuration C6 increases the peak memory usage of the encoder about 3 times for QCIF and 3.5 times for CIF. However, the peak memory usage during a coding session for CAVLC and CABAC are almost identical, given that the rest of the coding tools used in the video encoder are the same. The profiling results show that replacing CAVLC with CABAC adds only an additional 0.06 MB of memory usage per slice for context modeling. Because the additional memory required per slice is small and each frame can only contains a limited number of slices, this explains the negligible increased in peak memory usage when CABAC is used over CAVLC

The peak memory usage for the video decoder is smaller compared to the video encoder, lying between 3-7 MB for QCIF sequences and 10-16 MB for CIF sequences. Compared to that of Baseline profile configuration C2, decoding sequences encoded with Main profile configuration C6 doubled the decoder's peak memory usage. Similarly to the video encoder, the difference in the peak memory usage of the video decoder between CAVLC and CABAC is too small to be reflected in the figure.

4.6 Functional Sub-blocks and ISA classes Analyses

4.6.1 Functional sub-blocks complexity analysis

The complexities of the functional sub-blocks of CABAC in a RDO encoder are analyzed in this sub-section. For analysis purposes, the functional sub-blocks of CABAC entropy coder have been given from a different perspective as shown in Fig. 4.2.

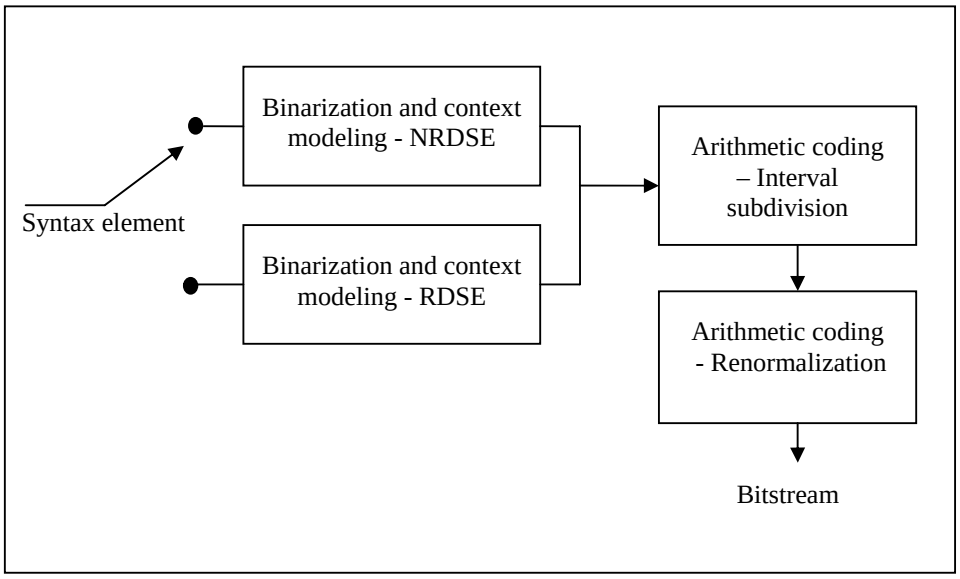


Figure 4.2: Functional sub-blocks diagram of the CABAC entropy coder

The four functional sub-blocks of CABAC entropy coder are the *Binarization and Context Modeling* (B&CM) blocks for non-residual data syntax element (NRDSE) and residual data syntax element (RDSE), and the *Arithmetic Coding* (AC) blocks for interval subdivision (AC-IS) and renormalization (AC-Renorm). NRDSE are syntax elements that contains information required for forming a macroblock prediction at the decoder, whereas RDSE are syntax elements that contain information on the residual (difference) macroblock, i.e. the transform coefficients. Note that the bypass coding stage has been integrated into each of the B&CM block for RDSEs and NRDSEs. Fig. 4.3 shows the percentage of computation spent by each sub-block when encoding CIF sequences at different QP values. Similar results are obtained for QCIF sequences and are not shown.

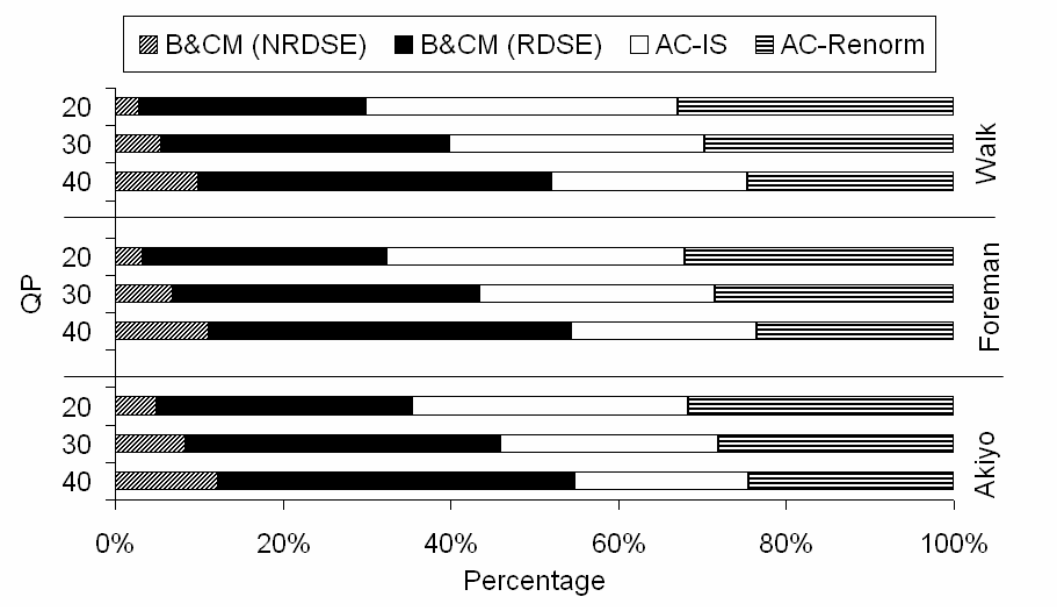


Figure 4.3: Percentage breakdown of entropy coding computation based on functional sub-blocks of CABAC entropy coder in a RDO encoder with Main profile configuration

The results show that the B&CM-NRDSE is the least critical sub-block, constituting at most 12% of the computation. At low and moderate QP values, each of the sub-blocks of B&CM-RDSE, AC-IS and AC-Renorm constitutes about one-third of the remaining computation whereas at high QP values, B&CM-RDSE is the most computationally complex block, constituting about 40% of the total computation. This means that B&CM-RDSE sub-block's computational complexity is higher than that of B&CM-NRDSE sub-block by approximately 8 times at QP=20 and approximately 4 times at QP=40. As such, in hardware-software partitioning, it is preferable to implement this functional sub-block in hardware, leaving the B&CM-NRDSE functional sub-block in software. This is because implementing the latter in hardware will require past coded data to be either stored in hardware (resulting in duplication of data) or be transferred from the software to the hardware for context modeling. This requires additional buffers or more data transfer to be performed. On the other hand,

no such data are needed by the B&CM-RDSE functional sub-block for context modeling.

4.6.2 Instruction set architecture classes

The entropy instructions have been classified into fourteen *Instruction Set Architecture* (ISA) classes. These instruction classes are given in Appendix A1. Fig. 4.4 shows the percentage of instruction classes for the executed entropy instruction in a RDO encoder. Less significant classes such as *Unconditional Branch* class and *Floating Point* class have been grouped together as '*Others*'. Similar results are obtained for the other sequences and are not shown.

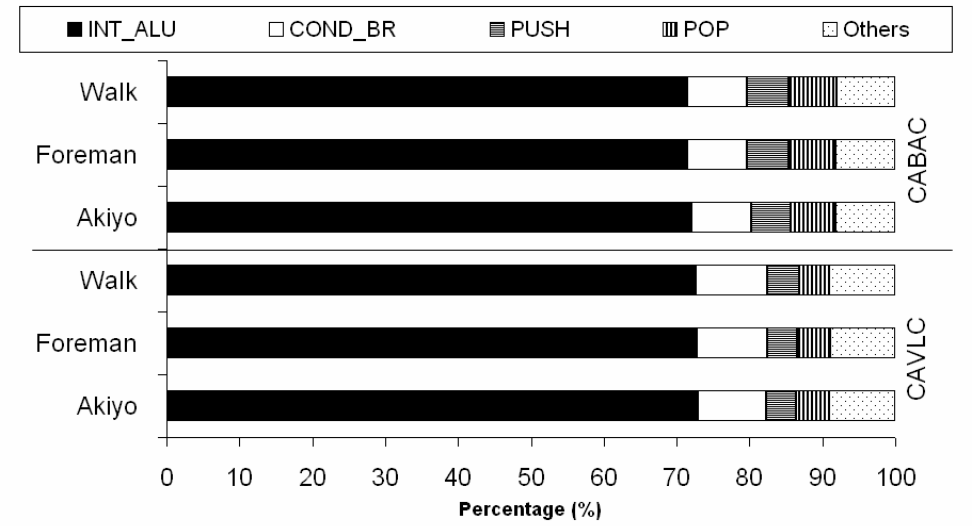


Figure 4.4: Percentage of ISA classes for the executed entropy instructions in a RDO encoder

For both CABAC and CAVLC, *Integer ALU* class makes up the bulk of the entropy instructions, constituting more than 70% of it. This is followed by the *Conditional Branch* class which makes up 8% to 11%. This shows that integer computation dominates the entropy coder's complexity whereas the use of floating

point instructions is minimal. In fact, CABAC uses no floating point operations for performing entropy coding except for adaptive initialization of context models at the start of each new slice. Comparatively, CABAC uses the stack more frequently compared to CAVLC. This is shown by the higher percentage of *PUSH* and *POP* classes with the use of CABAC, indicating that a higher memory bandwidth is required.

In the *Integer ALU* class, the *MOV* instruction dominates the integer operations making up 40% of the executed entropy instructions whereas multiplication and division operations constitute an insignificant percentage of the integer operations. A detailed ISA classification of the instructions executed in the entropy coders and the RDO encoder for CIF *Foreman* is given in Appendix A2.

4.7 Performance-Complexity Co-evaluation of CABAC

A performance-complexity co-evaluation of CABAC is presented. The objective is to identify scenarios where the use of CABAC will be more cost effective than CAVLC in software implementation. Table 4-14 summarized the performance and complexity of the non-RDO encoder and the RDO encoder for different combinations of entropy coding schemes with Baseline and Main profile configurations. All performance and complexity gains are measured with respect to the non-RDO encoder using CAVLC with Baseline profile configuration. The table has been arranged such that complexity generally increases from the lower left corner to the upper right corner.

Table 4-14: Performance-complexity table

			Baseline profile configuration C2				Main profile configuration C6			
			CAVLC		CABAC		CAVLC		CABAC	
			LMS	HMS	LMS	HMS	LMS	HMS	LMS	HMS
RDO Encoder	CIF	BR Saving (%)	7.1 – 8.4	5.8 – 9.5	12.7 -14.9	12.8 – 14.7	10.3 – 12.6	18.7 – 26.9	15.7 – 18.6	25.5 – 32.3
		Δ Y-PSNR (dB)	0.20 - 021	-0.05 – 0.16	0.22 – 0.26	-0.05 – 0.16	0.52 - 0.80	0.3 – 0.53	0.52 – 0.85	0.33 – 0.55
		EMA	6.5 – 7.2	6.3 – 6.9	8.7 – 10.0	8.5 – 9.1	14.6 – 15.8	18.8 – 19.4	17 – 20.9	20.4 – 22.9
		ECC	5.8 - 6.4	5.6 – 6.2	6.1- 7.1	6.0 – 7.0	13.2 – 14.4	17.2 – 18.1	13.5 – 15.1	17.9 – 18.3
		DCC	0.99 -1.00	0.90 – 0.92	0.97 -0.98	0.87 – 0.90	0.98 – 1.00	0.96 – 1.00	0.96 – 0.99	0.91- 0.96
		EPMU (MB)	9.0	9.0	9.0	9.0	30.0	30.0	30.0	30.0
		DPMU (MB)	10.0	10.0	10.0	10.0	15.0	15.0	15.0	15.0
	QCIF	BR Saving (%)	4.2 – 10.5	4.2 – 8.5	8.7 – 14.5	8.6 – 14.4	7.6 – 19.0	18.4 -23.6	11.8 – 18.8	24.3 – 27.9
		Δ Y-PSNR (dB)	0.13 – 0.22	-0.12 - 0.01	0.11 – 0.19	-0.08 – 0.16	0.65 – 0.85	0.16 – 0.43	0.65 – 0.88	0.22 – 0.48
		EMA	6 .6 – 7.2	6.4 – 7.2	9.3 – 10.5	9.2 -10.7	14.6 – 15.3	18.7 – 18.9	17.7 – 18.6	21.9 – 22.7
		ECC	6 – 6.5	5.7 – 6.5	6.3- 7.2	6.3 – 7.6	13.4 – 14.3	17.6 – 18	13.9 – 14.7	18.5 – 19.2
		DCC	0.98 – 1.00	0.84 – 0.88	0.97 – 1.00	0.99 – 1.02	0.97 – 1.00	0.93 – 0.99	0.96 – 0.08	0.85 – 0.93
		EPMU (MB)	2.6	2.6	2.6	2.6	8.5	8.5	8.5	8.5
		DPMU (MB)	3.0	3.0	3.0	3.0	6.0	6.0	6.0	6.0
Non-RDO Encoder	CIF	BR Saving (%)			5.4 – 6.9	5.8 – 8.7	7.3 – 8.6	15.2 – 19.8	12.7 – 14.0	19.6 – 26.2
		Δ Y-PSNR (dB)			0	0	0.26 – 0.60	0.32 – 0.47	0.26 – 0.60	0.32 – 0.47
		EMA			1.003 – 1.004	1.003 - 1.009	8.362 – 9.244	8.362 – 13.461	8.366 – 9.248	8.366 – 13.470
		ECC			1.002 - 1.003	1.004 – 1.005	7.836 -8.698	12.295 – 13.013	7.838 – 8.701	12.301 - 13.018
		DCC			0.97 – 0.98	0.90 -0.92	0.98 – 1.00	1.00 – 1.04	0.95 – 0.98	0.94 - 0.99
		EPMU (MB)			9.0	9.0	30.0	30.0	30.0	30.0
		DPMU (MB)			10.0	10.0	15.0	15.0	15.0	15.0
	QCIF	BR Saving (%)			3.8 – 4.3	4.1 – 7.5	6 – 12.9	12.9 -17.2	10.2 – 17.1	20.2 - 22.6
		Δ Y-PSNR (dB)			0	0	0.36 – 0.62	0.25 – 0.70	0.36 – 0.62	0.25 - 0.70
		EMA			1.006 - 1.007	1.011 - 1.017	8.352 – 9.256	12.196 – 12.999	8.361 – 9.265	12.216 - 13.015
		ECC			1.005 - 1.007	1.008 - 1.011	7.878 – 8.798	11.891 – 12.716	7.884 - 8.804	11.899 - 12.726
		DCC			0.97 – 0.98	0.87 – 0.91	0.97 – 1.00	0.99 – 1.02	0.96 – 0.98	0.89 – 0.95
		EPMU (MB)			2.6	2.6	8.5	8.5	8.5	8.5
		DPMU (MB)			3.0	3.0	6.0	6.0	6.0	6.0

LMS – Low-motion sequence, HMS – High-motion sequence
BR – Bit-rate; Δ Y-PSNR: quality of the luminance component
EMA – Video encoder Memory Access, ECC – Video encoder Computational Complexity,
DCC – Video decoder Computational Complexity
EPMU: Encoder Peak Memory Usage; DPMU: Decoder Peak Memory Usage
The BR saving and Δ Y-PSNR are given with respect to Baseline profile configuration with CAVLC
The values for EMA, ECC and DCC denote the increment factors with respect to Baseline profile configuration with CAVLC.

The table shows that it is not necessary true that coding performance and complexity performance have to be traded against each other. From the data, the use of CABAC, Main profile configuration and RDO improves the coding performance but have different impact on the encoder's complexity.

For a non-RDO encoder, the use of CABAC alone saves up to 9% bit-rates in Baseline profile configuration with negligible increase in encoder's complexity. The use of Main profile configuration with CAVLC achieves better bit-rate savings up to 20% at the expense of large increase in the encoder's computational complexity up to 13 times and its data transfer complexity up to 14 times. Note that the higher requirements arise solely from the use of more complex coding tools in the Main profile configuration. Up to another 10% bit-rates can be saved with the use of CABAC alone in the Main profile configuration with insignificant increase to the encoder's complexity. This gives an overall bit-rate savings up to 26% with the collective use of CABAC with Main profile configuration. The fact that bit-rates can be reduced while incurring negligible increase in encoder complexity indicates that CABAC is the most useful performance optimization tool for the non-RDO encoder and its use is always preferred.

For a RDO encoder, the use of CABAC increases its computational complexity up to 13% and the data transfer complexity up to 47% for an additional 3-9% bit-rate savings. Given that the RDO encoder is 2 to 10 times more complex than a non-RDO encoder, the additional data transfer requirements due to CABAC might be too high for many applications. It is observed that it is more cost-effective in terms of performance-complexity tradeoffs, to use Baseline profile configuration with CABAC for low motion sequences and Main profile configuration with CABAC for high motion sequences in a RDO encoder. This is because the use of Main profile

configuration over Baseline profile configuration in a RDO encoder doubled its computational and data transfer complexities for all sequences but obtained only a small improvement in bit-rate savings for low motion content sequences. On the other hand, more than twice the bit-rate savings can be obtained for high motion content sequences in this case.

4.8 Conclusions

In this chapter, the complexity assessment of CABAC under different encoder scenarios is given and is co-evaluated with its coding performance.

The analyses show that the use of CABAC over CALVC has negligible impact on the complexity of a non-RDO encoder which makes the use of CABAC to improve coding performance always preferable. The low complexity of CABAC from the perspective of the non-RDO encoder suggests the feasibility of implementing CABAC in software without any hardware assistance.

For the RDO encoder, the use of CABAC alone increases the encoder's computational complexity up to 13%, and its data transfer complexity up to 47%. This has yet to take into considerations the effect of using RDO, which not only increases the computational and data transfer complexities of the CABAC entropy coder tremendously by more than an order of magnitude, but also increases the complexities of other coding stages in the video encoder. As such, the RDO encoder is 2 to 6 times higher in computational complexity, and 2 to 10 times higher in data transfer complexity compared to a non-RDO encoder of a similar configuration. This suggests the need of a CABAC hardware accelerator for a RDO encoder.

It is observed for the RDO encoder, the marginal improvement in coding efficiency due to the use of additional complex coding tool saturates faster for low-

motion content sequences. This makes it more cost effective to use Baseline profile with CABAC for low-motion content sequences, and Main profile configuration with CABAC for high-motion content sequences.

It is observed that the use of CABAC results in higher increase in encoder's data transfer complexity in comparison to the increases in computational complexity. This indicates that CABAC is dominated more by data transfer complexity rather than computational complexity, and its bottleneck might lies at the memory interface. Given the relationship that memory read operations are 50% higher than memory write operations when CABAC is used, memory read and memory write circuitries and bus width can be designed accordingly. Alternatively, data cache can be used to effectively reduce the average latency of data transfer.

Within the CABAC entropy coder, up to 8 times less computation is performed for the binarization and context modeling of NRDSEs than RDSEs. For hardware-software co-design, we propose to implement the non-critical B&CM-NRDSE functional sub-block in software, whereas the remaining sub-blocks (B&CM-NRDSE and the arithmetic coder) in hardware. In addition, CABAC can be implemented without the use of floating-point instruction class. This information might be useful for hardware designers.

Given the rest of the coding tools used in the video encoder are the same, the use of CABAC over CALVC is always beneficial to the decoder as it results in lower computational and data transfer complexities of the decoder. This suggests that CABAC should be selected for off-line encoding applications where processing time and power consumption are not an issue at the encoder but at the decoder, for instance, mobile phones and PDAs.

In the next chapter, research work extending from the complexity analyses of CABAC will be given.

CHAPTER 5 RDO FOR MODE DECISION

In this chapter, separate findings and recommendations on the use of rate-distortion optimization (RDO) for mode decision in the video encoder are presented.

5.1 Predictive Coding Modes

5.1.1 Statistic of prediction modes selected for encoding

Due to the high temporal correlation between neighboring frames, it is statistically more efficient to use an Inter mode than an Intra mode to encode a macroblock. Figure 5.1 shows the percentage of prediction modes used by a RDO encoder with Main profile configuration for the different sequences. The Inter modes of SKIP/DIRECT, 16x16, 16x8, 8x16 and 8x8 are grouped as ‘Inter’ whereas Inter modes with block size smaller than 8x8 are grouped as ‘Inter P8x8’.

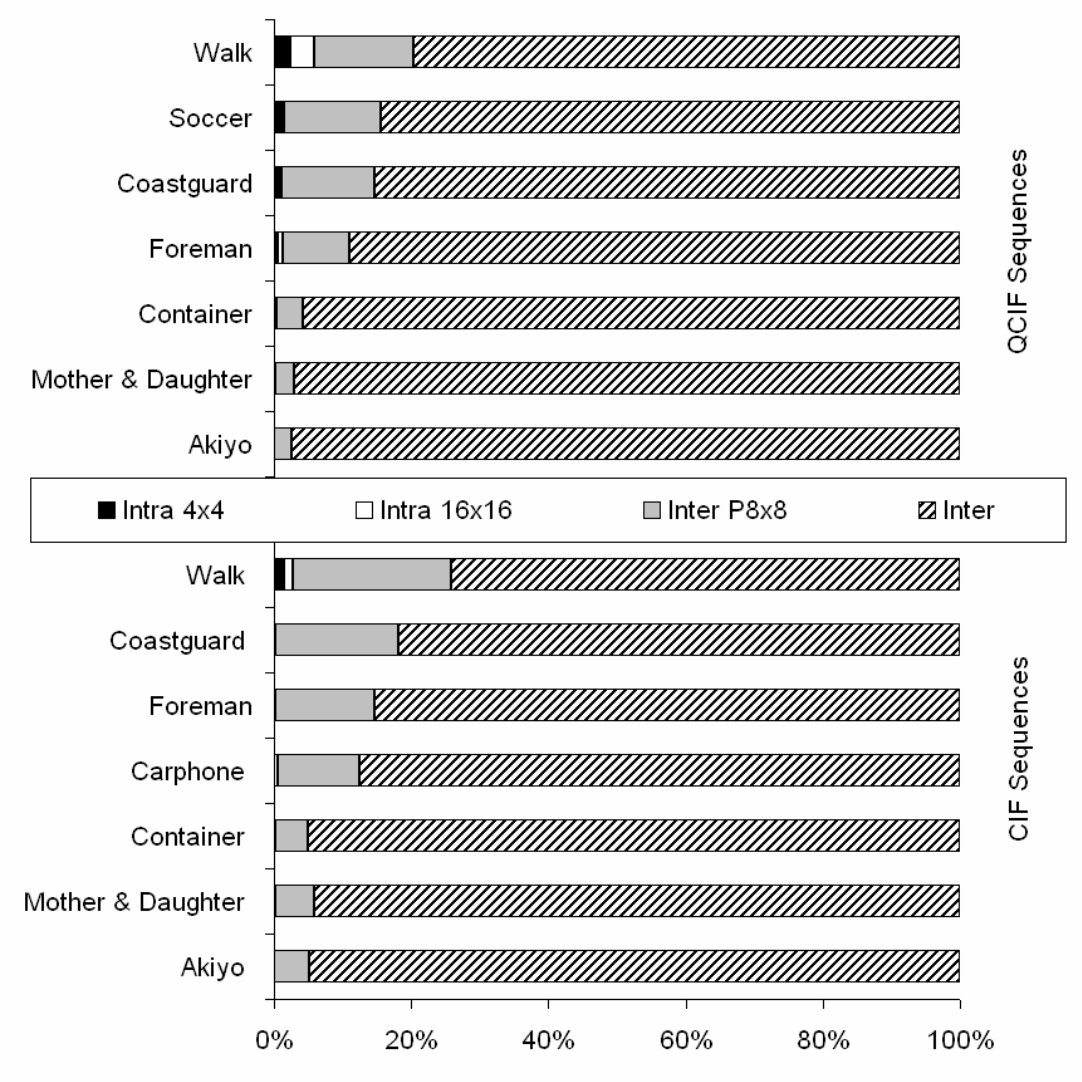


Figure 5.1: Percentage of prediction modes used in encoding QCIF and CIF sequences

The statistic shows that Intra 4x4 mode constitutes a very low percentage of the prediction modes that is selected for encoding the wide genre of video contents in our work. This means that the likelihood of using the Intra 4x4 mode to encode a macroblock in an Inter frame is low.

5.1.2 Entropy computation partitioning based on prediction modes

On the other hand, the use of RDO requires each macroblock to be entropy coded in every candidate predictive coding modes for mode decision. A partitioning of the computational complexity of the entropy coder based on the candidate predictive coding modes is given in Figure. 5.2.

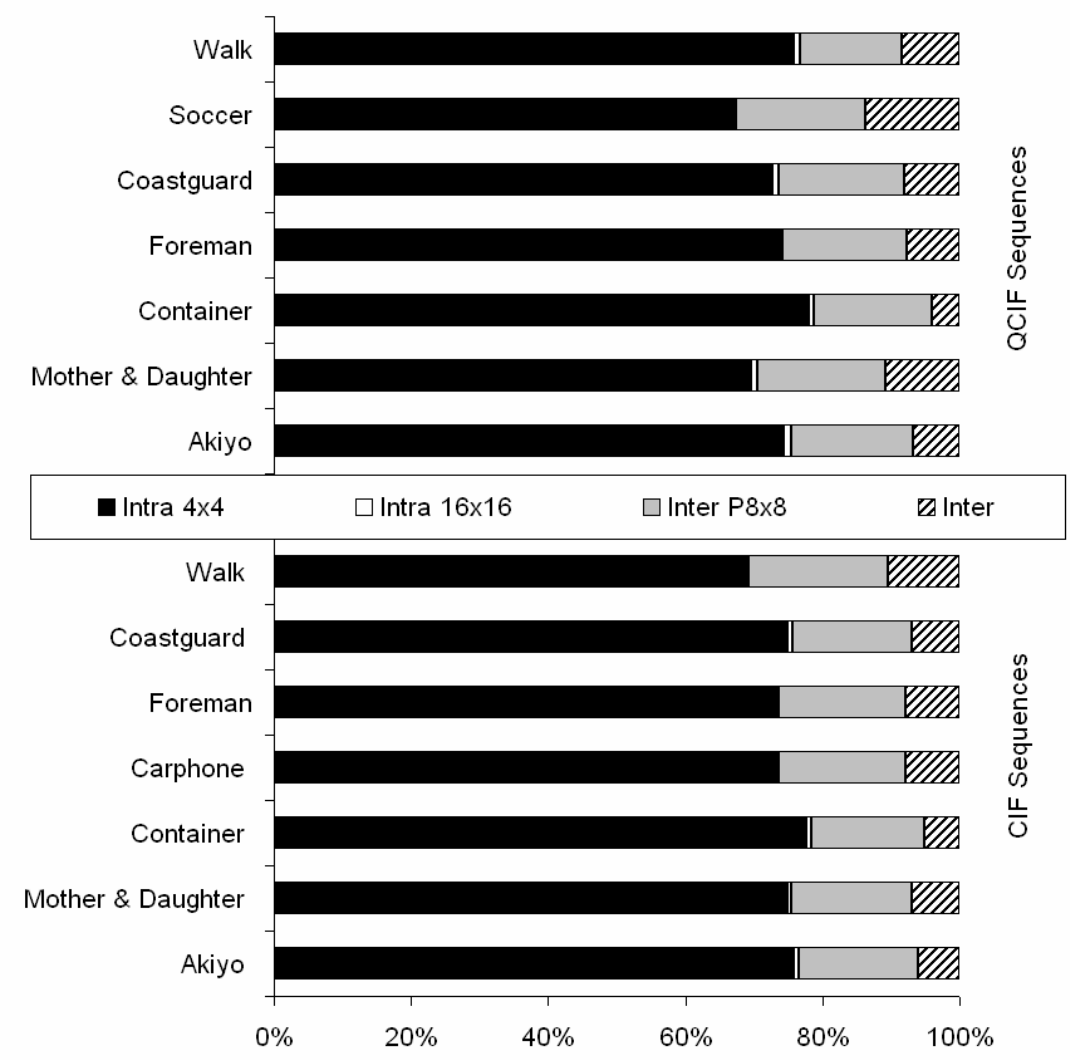


Figure 5.2: Partitioning of entropy instructions based on predictive coding modes in the RDO encoder

The data show that more than 60% of the executed entropy instructions are for coding macroblocks in Intra 4x4 mode alone. From the entropy coder's perspective,

this implies that evaluating the optimality of using Intra 4x4 mode to encode a macroblock is much more computationally complex than evaluating the optimality of other candidate predictive coding modes. This is so due to the high number of spatial directional modes employed for spatial prediction in Intra 4x4 mode. More details on spatial directional modes can be found in [26].

Due to the low probability of using Intra 4x4 mode in Inter frame, and the high accumulated complexity of evaluating the optimality of Intra 4x4 mode, it might be more cost-efficient to disable the use of Intra 4x4 directional modes in Inter frame.

5.1.3 Effect of disabling Intra4x4 mode for inter frame

Table 5-1 shows the performance degradation of the RDO encoder and the reduction in its computational and data transfer complexities, when all directional modes of Intra 4x4 are disabled in the RDO encoder. The use of Intra 4x4 DC mode is however not disabled, so as to minimize performance degradation.

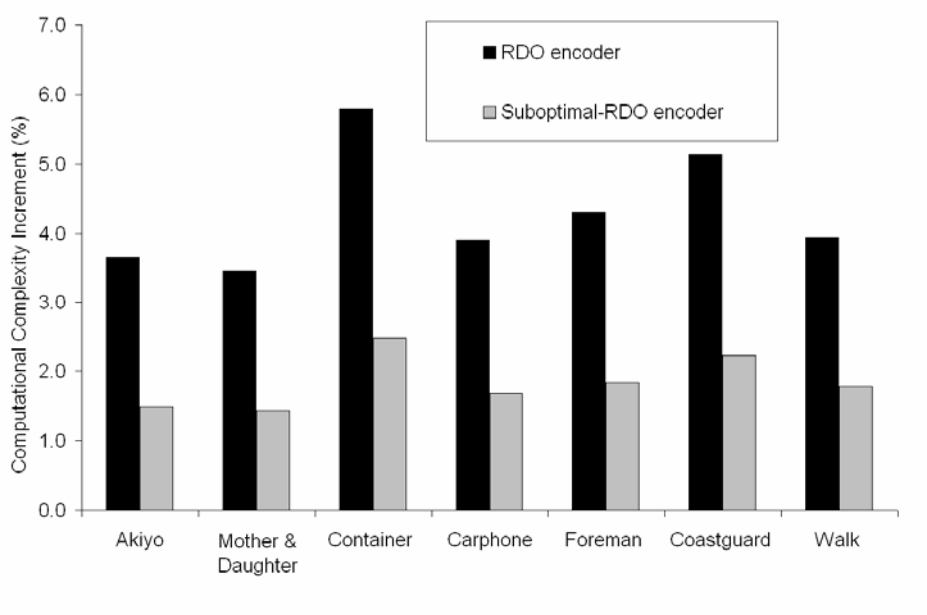
Table 5-1: Performance degradation and complexity reduction in the RDO encoder due to disabling Intra 4x4 directional modes for Main profile configuration with CABAC

QCIF sequence	Performance Degradation		Complexity Reduction	
	Δ Bit-rate (%)	Δ Y-PSNR (dB)	Δ Computational Complexity (%)	Δ Data Transfer Complexity (%)
Akiyo	0.00	0.00	22.5	31.2
Mother & Daughter	0.09	-0.01	20.6	30.1
Container	0.01	0.00	24.3	34.0
Carphone	-0.06	-0.01	19.6	30.1
Foreman	0.02	-0.02	18.6	27.7
Coastguard	-0.21	-0.01	19.2	28.6
Walk	0.19	0.00	16.7	24.9
CIF Sequence				
Akiyo	0.06	0.00	21.4	29.3
Mother & Daughter	-0.04	-0.01	19.9	27.6
Container	-0.01	0.00	23.0	31.3
Foreman	0.41	-0.03	17.9	24.8
Soccer	-0.01	-0.01	16.0	22.6
Coastguard	0.42	-0.04	18.9	25.4
Walk	0.11	-0.03	16.1	22.7

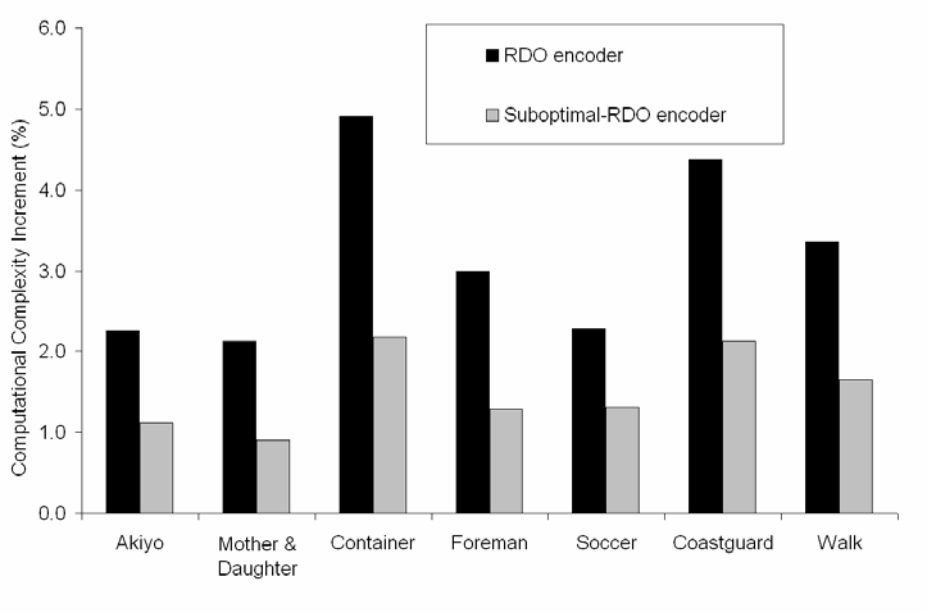
The results show that for Inter frame, disabling the use of Intra 4x4 directional modes in the RDO encoder results in negligible degradation to the coding performance but achieved a reduction in its computational complexity up to 25% and a reduction in its data transfer complexity up to 34%. These are large percentage reductions, which mean that it is much more cost-efficient to use the RDO encoder with Intra 4x4 directional modes disabled when coding an Inter frame. For easy reference, the RDO encoder with Intra 4x4 directional modes disabled shall be referred to as the *suboptimal-RDO* encoder.

5.1.4 Effect of CABAC on suboptimal-RDO encoder's complexity

The use of CABAC in the suboptimal-RDO encoder increases it's computational and data transfer complexities. Figure 5.3 compares the percentage increments in computational complexity of the RDO encoder and the suboptimal-RDO encoder due to the use of CABAC. Figure 5.4 compares the percentage increment in data transfer complexity of the RDO encoder and the suboptimal-RDO encoder due to the use of CABAC.

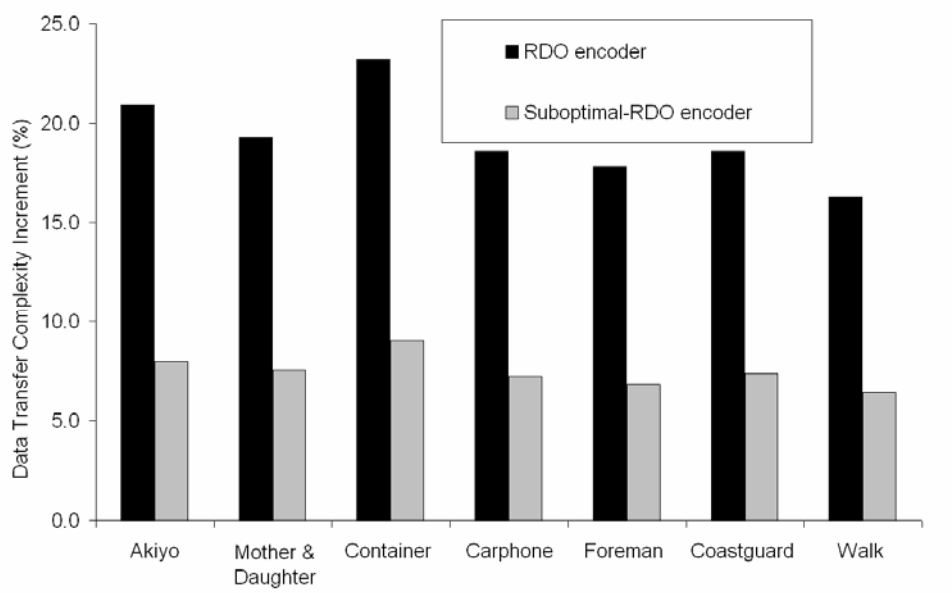


(a)

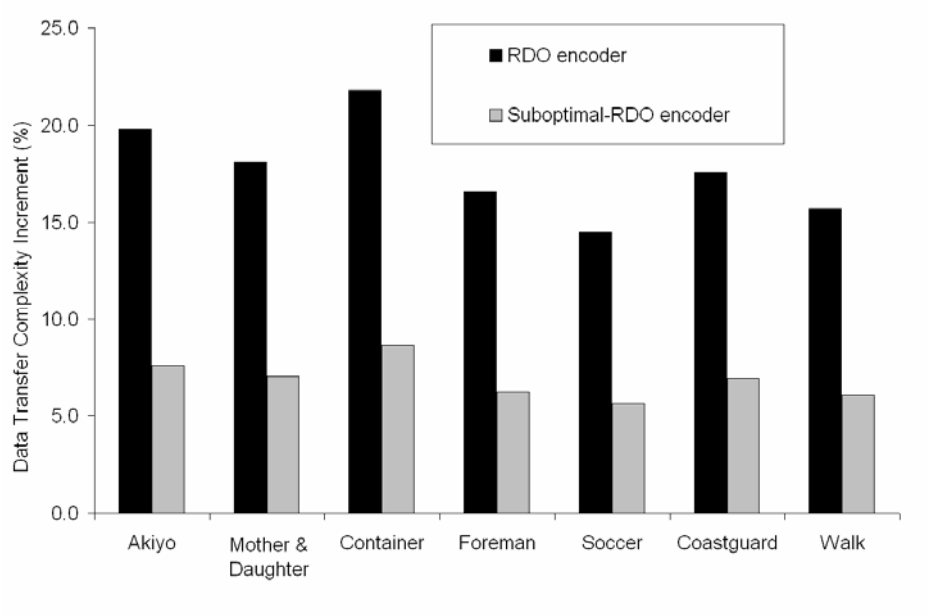


(b)

Figure 5.3: Percentage increments in computational complexity of the RDO encoder and the suboptimal-RDO encoder due to the use of CABAC for (a) QCIF sequences (b) CIF sequences



(a)



(b)

Figure 5.4: Percentage increments in data transfer complexity of the RDO encoder and the suboptimal-RDO encoder due to the use of CABAC for (a) QCIF sequences (b) CIF sequences

For the suboptimal-RDO encoder with the same Main profile configuration, the use of CABAC only increases its computational complexity by at most 2.5%, and its data transfer complexity by at most 9%. As reported earlier, the use of CABAC in the RDO encoder resulted in a 2-6 % increase in computational complexity and a 12-24% increase in data transfer complexity for Main profile configuration. This shows that a reduction in the complexity of the entropy coder diminishes the effect of using the more computationally intensive CABAC on the video encoder's complexity.

5.1.5 Bit-rate saving by CABAC in a sub-optimal RDO encoder

For both type of video encoders (RDO encoder and sub-optimal RDO encoder), similar bit-rate savings due to the use of CABAC can be obtained. This is shown in Table 5-2, which lists the bit-rate savings by CABAC for the RDO encoder and the suboptimal-RDO encoder.

Table 5-2: Bit-rate savings by CABAC for the RDO encoder and the suboptimal-RDO encoder

QCIF Sequences	Bit-rate saving (%)	
	RDO encoder	Suboptimal-RDO encoder
Akiyo	4.6	4.6
Mother & Daughter	4.1	4.1
Container	4.4	4.5
Carphone	3.5	3.5
Foreman	4.7	4.8
Coastguard	7.2	7.2
Walk	5.7	5.7
CIF Sequences		
Akiyo	5.8	5.7
Mother & Daughter	6.8	6.8
Container	6.0	6.2
Foreman	6.6	6.7
Soccer	7.6	7.5
Coastguard	8.3	8.3
Walk	7.4	7.7

The use of CABAC in the suboptimal-RDO encoder saves 3-8 % bitrates. The bit-rate savings obtained for the suboptimal-RDO encoder is similar to that of the RDO encoder. This indicates that the coding performance of CABAC is not affected by the use of suboptimal-RDO encoder.

Hence, the suboptimal-RDO encoder should be used if the combined use of RDO and CABAC is desired for performance optimizations with an overall lower encoder's complexity and marginal increases in the computational complexity and small increase in data transfer complexity due to the use of CABAC.

Although our results show that there is little performance degradation, we are not suggesting that the use of Intra 4x4 directional modes is totally redundant in Inter frame. This is because the use of Intra 4x4 mode might improve the subjective quality of the video, which could be missed in our objective evaluation of video quality.

5.2 Fast RDO

A fast-RDO encoder (that has been discussed in chapter 2, section 2.3) is used to encode *"Head & Shoulder"* sequences. Its coding performance and complexity performance are presented here, benchmarked against a non-RDO encoder.

The motivation for such analysis is due to the low-motion characteristics of Head & Shoulder sequences, which makes the use of fast-RDO encoder particularly suitable for this type of sequences. To be more explicit, Head & Shoulder sequences are characterized by stationary backgrounds, slow temporal change in background scene, and low entry and exit of video objects into the scene. In a stationary background, the likelihood of using SKIP/DIRECT and Inter 16x16 modes are much higher whereas in regions of head and shoulder motion, the likelihood of using Inter

modes of smaller block size are higher. Due to the slow changing scene, there exists a strong temporal correlation between the prediction modes used in the current macroblock and the co-located macroblock of the neighboring frame. On the other hand, the early termination strategy of the fast-RDO encoder works effectively well if the candidate prediction modes can be accurately ordered based on their likelihood of being selected for encoding. Therefore, the fast-RDO encoder can reference the prediction modes used in the previous encoded frame to guide its ordering of the candidate prediction modes for the current frame.

5.2.1 Ordering of prediction mode

For this work, the fixed ordering of prediction modes used in the fast RDO encoder is modified slightly to adapt to the local statistic as given below. Let **M** denotes the prediction mode used in the co-located macroblock. Table 5-3 lists the prediction modes orderings used by the RDO encoder for the possible modes of **M**.

Table 5-3: Ordering of prediction modes for the fast-RDO encoder

M	Prediction Mode Ordering
SKIP/DIRECT Inter_16x16 Inter_16x8 Intra Modes	{SKIP, DIRECT, Inter_16x16, Inter_16x8, Inter_8x16, Inter_8x8, Inter_8x4, Inter_4x8, Inter_4x4, Intra_16x16, Intra_4x4}
Inter 8x16	{SKIP, DIRECT, Inter_16x16, Inter_8x16, Inter_16x8, Inter_8x8, Inter_8x4, Inter_4x8, Inter_4x4, Intra_16x16, Intra_4x4}
Inter 8x8 / Inter 8x4 / Inter 4x4	{SKIP, DIRECT, Inter_16x16, Inter_8x8, Inter_8x4, Inter_4x8, Inter_4x4, Inter_16x8, Inter_8x16, Intra_16x16, Intra_4x4}
Inter 4x8	{SKIP, DIRECT, Inter_16x16, Inter_8x8, Inter_4x8, Inter_8x4, Inter_4x4, Inter_16x8, Inter_8x16, Intra_16x16, Intra_4x4}

5.2.2 Coding performance

The use of a fast-RDO encoder achieves lower bit-rates than a non-RDO encoder with similar configuration. Table 5-4a summarized the bit-rate savings obtained due to the use of the fast-RDO encoder for different configurations. Configurations that use CABAC as the entropy coding scheme are indicated by an asterisk "*". For example, C1 denotes the collective use of configuration C1 with CAVLC whereas C1* denotes the collective use of configuration C1 with CABAC. Table 5-4b lists the corresponding change in Y-PSNR as a result of using the fast-RDO encoder.

Table 5-4a: Percentage bit-rate savings due to fast-RDO encoder

CIF	C1	C1*	C2	C2*	C3	C3*	C4	C4*	C5	C5*	C6	C6*	C7	C7*
Akiyo	6.9	6.4	6.9	6.6	5.8	4.9	3.6	2.8	2.8	2.5	2.7	2.1	2.6	2.3
Mother & Daughter	4.6	5.1	7.2	5.1	3.5	3.7	2.8	3.1	2.5	2.9	3.4	4.1	3.3	3.9
Silent	5.6	5.3	4.7	5.6	4.4	4.0	3.7	3.5	3.4	3.1	2.0	2.6	1.9	2.8
Paris	6.8	6.8	5.9	6.8	5.7	4.8	3.9	3.8	3.7	3.3	3.1	2.7	3.6	2.8

Table 5-4b: Δ in Y-PSNR (dB) due to fast-RDO encoder

CIF	C1	C1*	C2	C2*	C3	C3*	C4	C4*	C5	C5*	C6	C6*	C7	C7*
Akiyo	0.09	0.11	0.15	0.12	0.14	0.06	0.07	0.05	0.03	-0.01	0.04	0.07	0.04	0.06
Mother & Daughter	0.15	0.14	0.17	0.16	0.17	0.16	0.05	0.04	0.08	0.07	0.10	0.06	0.09	0.06
Silent	0.15	0.11	0.21	0.18	0.20	0.17	0.13	0.06	0.15	0.04	0.12	0.08	0.12	0.08
Paris	0.13	0.12	0.14	0.14	0.11	0.13	0.08	0.11	0.06	0.04	0.05	0.05	0.05	0.06

For Head & Shoulder sequences, savings in bit-rates between 2% and 7% can be obtained with the use of fast-RDO in the video encoder. This comes with insignificant changes to the Y-PSNR. It is observed that smaller savings are obtained for more complex configurations. This implies that the performance of fast RDO algorithm worsens with the use of more complex coding tools. Its performance however is minimally affected by the choice of entropy coding scheme used.

5.2.3 Computational complexity

Figures 5.5 – 5.9 show the computational complexities of the non-RDO encoder and the fast-RDO encoder across different configurations for the Head & Shoulder sequences. Configurations that use CABAC as the entropy coding scheme are indicated by an asterisk, "*". The percentage change in computational complexity of the video encoder due to the use of fast-RDO is listed in Table 5-5.

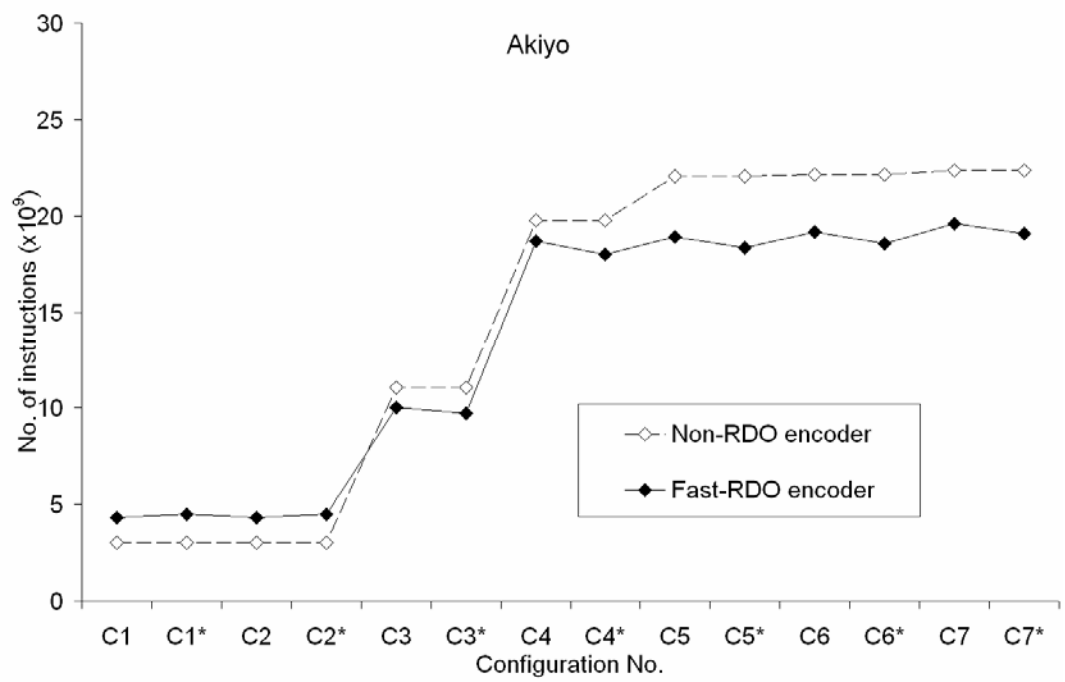


Figure 5.5: Computational complexity of the fast-RDO encoder and the non-RDO encoder for test sequence *Akiyo*

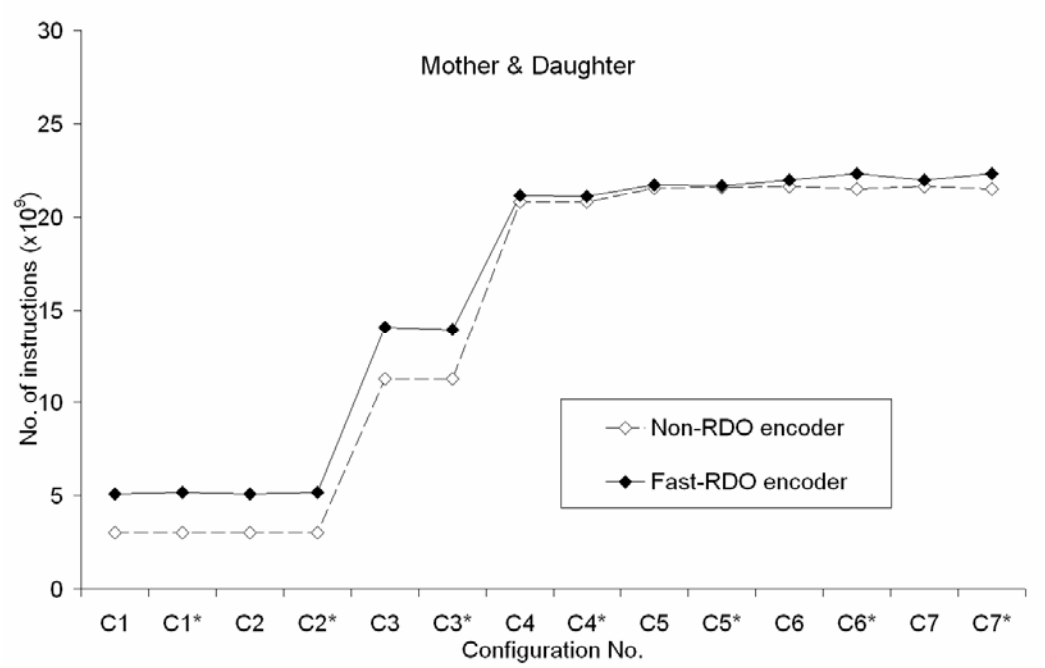


Figure 5.6: Computational complexity of the fast-RDO encoder and the non-RDO encoder for test sequence *Mother & Daughter*

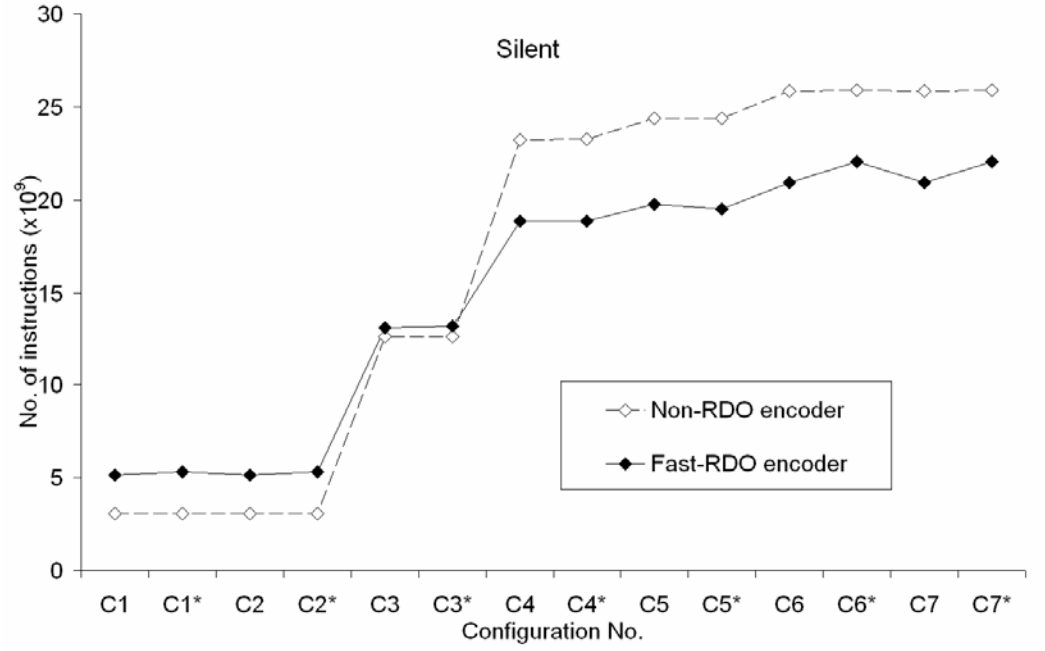


Figure 5.7: Computational complexity of the fast-RDO encoder and the non-RDO encoder for test sequence *Silent*

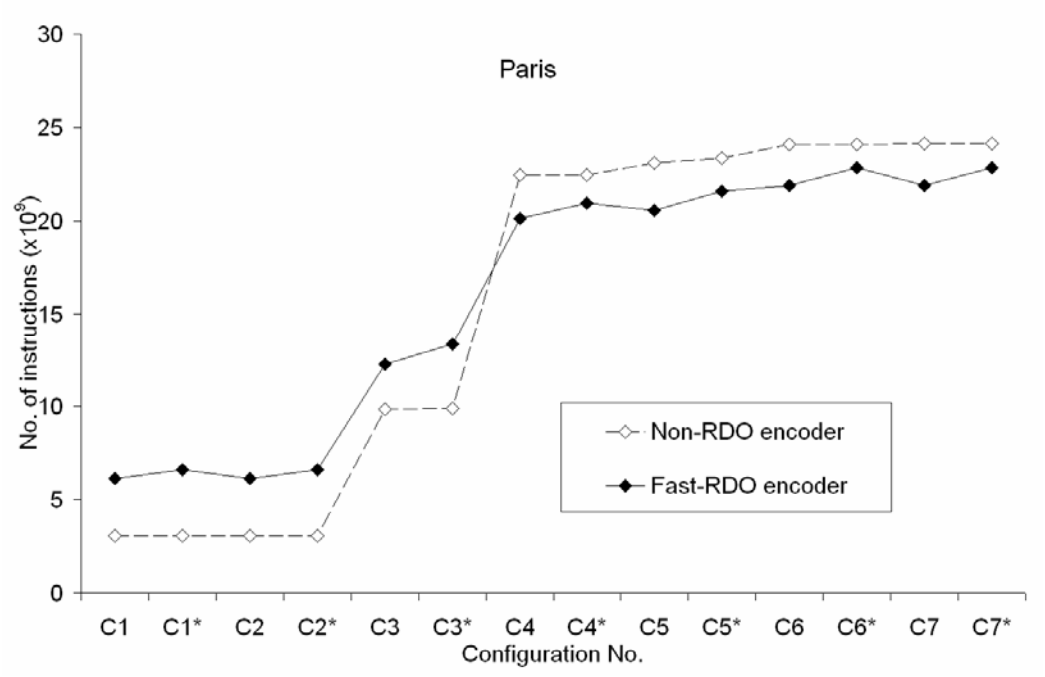


Figure 5.8: Computational complexity of the fast-RDO encoder and the non-RDO encoder for test sequence *Paris*

Table 5-5: Percentage change in computational complexity of the video encoder due to fast-RDO in comparison to a non-RDO encoder

CIF	C1	C1*	C2	C2*	C3	C3*	C4	C4*	C5	C5*	C6	C6*	C7	C7*
Akiyo	42.3	47.6	43.3	47.1	-9.5	-12.2	-5.4	-8.9	-14.2	-16.7	-13.5	-16.2	-13.3	-14.6
Mother & Daughter	67.7	69..8	68.4	72.0	24.7	23.4	1.6	1.3	0.7	0.6	1.8	3.7	1.8	1.9
Silent	67.0	73.6	67.6	72.9	3.6	4.4	-18.7	-18.9	-17.6	-20.1	-18.7	-14.9	-18.9	-14.1
Paris	98.9	114.4	99.3	115.8	24.5	34.9	-10.5	-6.8	-11.0	-7.5	-9.1	-5.2	-9.3	-5.3

For Head & Shoulder sequences, the use of fast-RDO increases the encoder's computational complexity up to 120% for simple configurations but may achieved reduction in encoder's computational complexity up to 20% for complex configurations. From the computational complexity point of view, this makes it attractive to use the fast-RDO encoder for complex configurations in comparison to a non-RDO encoder, as bit-rate savings can be achieved with negligible degradation in video quality and possible reduction in encoder's computational complexity.

5.2.4 Data transfer complexity

Table 5-6 lists the percentage increase in data transfer complexity of the video encoder due to the use of fast-RDO.

Table 5-6: Percentage increase in data transfer complexity of the video encoder due to fast-RDO in comparison to a non-RDO encoder

CIF	C1	C1*	C2	C2*	C3	C3*	C4	C4*	C5	C5*	C6	C6*	C7	C7*
Akiyo	56.9	63.4	52.0	61.9	12.6	10.5	8.2	7.6	4.8	3.1	2.7	1.2	2.9	1.3
Mother & Daughter	74.8	85.1	78.1	83.6	37.4	38.3	10.9	11.4	8.6	4.2	7.3	6..9	7.6	6.5
Silent	70.2	77.5	74.3	79.1	13.7	16.1	3.8	3.4	3.2	2.6	0.2	1.3	0.7	1.8
Paris	119.3	131.5	104.6	127.2	40.5	44.3	9.7	8.6	3.4	6.3	2.9	3.9	3.1	4.6

For Head & Shoulder sequences, the use of fast-RDO increases the data transfer complexity of the video encoder up to 140%. It is observed that the percentage increments obtained with complex configurations are relatively small. As such, it is still be preferable to use the fast-RDO encoder than the non-RDO encoder for complex configurations, so as to take advantage of the additional bit-rate savings.

5.2 Conclusion

In this chapter, a suboptimal-RDO encoder and a fast-RDO encoder for mode decision have been discussed.

The suboptimal-RDO encoder merely refers to the RDO encoder with Intra 4x4 directional modes disabled for use in Inter frames. The results showed that the use of the suboptimal-RDO encoder achieves similar bit-rate savings as the RDO encoder with negligible degradation in video quality, but with a reduction in computational complexity up to 25%, and a reduction in data transfer complexity up

to 34% in comparison to the RDO encoder. The use of CABAC in suboptimal-RDO mode increases in both encoder's computational complexity and data transfer complexity, but by smaller percentages in comparison to the use of CABAC in a RDO encoder. This comes with negligible degradation in the coding performance. As such, it is more cost-effective to use the suboptimal-RDO encoder with CABAC as compared to a RDO encoder.

The fast-RDO encoder achieves fast mode decision by adopting an early termination strategy so as to reduce the number of candidate prediction modes to a smaller subset. It is found that fast-RDO encoder works well for *Head & Shoulder* sequences. For such sequences, savings in bit-rate between 2 and 7% can be obtained by using fast-RDO encoder in comparison to a non-RDO encoder. For complex configurations, the computational complexity of the fast-RDO encoder may be lower than that of the non-RDO encoder up to 20%. The results demonstrate that more bit-rates can be saved with less computational complexity and small increases in data transfer complexity of the fast-RDO encoder when it is used for complex configurations. This makes it attractive to use fast-RDO as the video encoder control in power-limited devices when complex coding tools are turned on in the video encoder.

In the next chapter, conclusions will be given.

CHAPTER 6 CONCLUSIONS

In this thesis work, comprehensive analyses on the performance and complexity of CABAC have been conducted. A summary of the findings that have yet been reported or not highlighted in other works are given below. This is followed by recommendations and suggestions on complexity reductions for the CABAC implementation.

6.1 Findings

- The computational and data transfer requirements of a CABAC entropy coder depend largely on the encoder control used by the video encoder. Between a non-RDO encoder and a RDO encoder, these requirements vary drastically by more than an order of magnitude. Table 6-1 shows the real-time computational and data transfer requirements needed by the CABAC entropy coder in a non-RDO encoder and a RDO encoder when coding QCIF and CIF sequences at a frame rate of 30 fps.

Table 6-1: Real-time computational and memory requirements of CABAC entropy coder

Encoder Control	QCIF Sequence		CIF Sequence	
	Computational (MIPS)	Memory Access Frequency ($x10^6$ /s)	Computational (MIPS)	Memory Access Frequency ($x10^6$ /s)
Non-RDO	80	60	290	220
RDO	6420	5240	25 850	19 620

The real-time requirements suggest that when no performance optimization is used in the video encoder (i.e. no use of RDO for mode decision), it is feasible to

implement CABAC in software without using any hardware assistance. When rate-distortion optimization is used by the video encoder for optimizing mode decision, hardware accelerator of CABAC will definitely be needed to meet any real-time requirements or to speed up the entropy coding process. Alternatively, sub-optimal RDO techniques can be used instead to lower the computational and data transfer requirements of the CABAC entropy coder to near real-time but at the expense of degradation in coding performance.

- The use of CABAC is found to have a larger impact on the entropy coder's data transfer complexity than on its computational complexity. In comparison with CAVLC, CABAC increases the data transfer complexity of the entropy coder up to 44% for a non-RDO encoder and up to 115% for a RDO encoder. The higher percentage increases in data transfer for the RDO encoder is due to its more frequent need to store and reset the coding states of the context models and arithmetic coder when it is computing the bit-rates of the candidate prediction modes. This is one of the possible IO bottlenecks of CABAC.

- Analyzing the complexity increment of using CABAC over CALVC from the perspective of the video encoder can be misleading [4]. This is because the relative figures depend on the overall computational and data transfer complexities of the video encoder. To elaborate further, the impact of using CABAC on the video encoder's complexity diminishes
  - when there is a reduction in the complexity of the CABAC entropy coder, or
  - when there is an increase in the overall complexity of the video encoder as a result of using more complex coding tools (not inclusive of the RDO tool).

Nonetheless, the relative complexities of the video encoder for different combinations of entropy coding schemes and configurations show that the use of CABAC in a non-RDO encoder is not computationally expensive. CABAC adds negligible increases to the video encoder. Hence, its use is always preferred over CAVLC in a non-RDO encoder.

- It is also cost-effective in terms of coding efficiency improvements and complexity increment (in computational and data transfer) to use CABAC over CAVLC in a RDO encoder. However, given the already high computational complexity and data transfer complexity of the RDO encoder, up to 13% increases in computational complexity and 47% increase in data transfer complexity due to the use of CABAC alone can be too demanding for some systems.

- Both the use of CABAC and RDO improve the coding efficiency. However, in terms of coding efficiency improvements and complexity increases in the video encoder, CABAC is much more useful than RDO as it provides a substantial improvement in coding efficiency without incurring a high increases in computational and data transfer complexities of the video encoder. (Refer to Tables 3-5, 4-4 and 4-10 for the supporting statistics). Furthermore, CABAC delivers consistent coding efficiency improvements regardless of the configuration used in the video encoder whereas the coding performance of RDO is dependent on the choice of coding tools used in the video encoder. It is found that the use of complex coding tools saturates the overall coding efficiency for low-motion content sequences, making the use of RDO for further bit-rate reduction less effective in such cases. However, the use of RDO has negligible impact on the

decoder's complexity. This makes the use of RDO presently more suitable for off-line encoding applications, where bandwidth is a more important issue over coding time and processing power.

- For constant bit-rate encoder, the use of CABAC in comparison to CAVLC results in only marginal improvement in video quality. This indicates that CABAC is not a useful tool for improving the video quality at constant bit-rate.

- The use of CABAC is always beneficial to the decoder as it results in lower computational and data transfer complexities of the decoder. (This was not reported in any work although in [4], similar result has been obtained for one of their test sequences). This leads to lower processing power, which is attractive for power-limited devices.

6.2 Suggestions / Recommendations

- CABAC is essentially a sequential process, which makes it difficult to accelerate the process using *Single Instruction Multiple Data* (SIMD) techniques. However, there are still parallelizations that can be exploited for its implementation using multiprocessor with *Multiple Instruction Multiple Data* (MIMD) techniques. CABAC can be carried out in parallel for every frame or slice (if more than one slice is used per frame) because the coding states of context models and arithmetic coder are reset when coding each new frame/slice, and neighboring frames/slices have no data dependencies. For a RDO encoder, parallelism can also be identified on a smaller scale at the macroblock level. The task of computing the bit-rates of all candidate prediction modes for a macroblock may be carried out in parallel, as the coding states of context models and arithmetic coder are restored back to the initial coding states before computing the bit-rate of the next candidate prediction mode. However, distinct memory locations need to be allocated for each candidate prediction mode for storing the coding states of context models and arithmetic coder so as to eliminate any data dependencies and to ensure correct coding results.

- It is found to be much more computationally complex to compute the bit-rate of Intra 4x4 mode as compared to other candidate predictive coding modes. Alternatively, it might be more efficient to just separate Intra 4x4 mode from the rest of the candidate predictive coding modes, and execute its bit-rate computation in parallel with the bit-rate computation for the remaining candidate predictive coding modes.

- Conditional branch instructions constitutes up to 11% of the entropy instructions. Though the percentage is not high, it can still degrade the performance of the CABAC system if the conditional branch instruction stalls the processor pipeline until the next instruction to be fetched is decoded. One of the conditional branches executed most frequently is the conditional branch for encoding a *least probable symbol* (LPS) / *most probable symbol* (MPS). One simple way to deal with this branch delay is to employ trivial branch prediction. Based on our observations that approximately 65-73% of the bins encoded are *MPS,* the branch predictor can predicts that the branch to LPS will always not be taken and the fetch stage continues to fetch instructions from the fall-through path of MPS without stalling. If a wrong prediction is made, the pipeline is then flushed clean and the correct instruction is fetched.

- Alternatively, delayed branches may be more useful for the LPS/MPS branch. Since the operation E - E_{LPS} is performed in both paths, their corresponding instructions are executed independent of the result of the branch instruction. These instructions can be carefully selected at compiled time to be executed in the branch delay slots. This ensures that useful instructions are executed till the branch address is available, thereby removing the penalty of executing the branch instructions.

- The ADD instruction contributes to a substantial percentage of the entropy instructions as well as the total instructions executed by the video encoder. The use of more execution subunits operating in parallel for ADD operations will enhance efficiency.

- It is found that there exists a 3:2 relationship between memory read operations and memory write operations in the entropy coder. As such, the associated memory read and memory write circuitries, and bus widths can be designed accordingly to reduce the data transfer latency. The use of multi-level memory hierarchy (i.e. registers, caches, off-chip memory and main memory) may also help to reduce the memory access time. It is observed that RDSEs are encoded much more frequent by 4 to 8 times more than NRDSE. As such, the context models corresponding to RDSEs, together with the range look-up table should be stored in fast off-chip memory where they are accessed or/and updated more often, and the remaining context models can be stored in slower main memory.

- When implementing CABAC with *Field Programmable Gate Array* (FPGA), it is recommended that context modeling of NRDSEs is performed in software using existing microprocessor, whereas the context modeling of RDSEs is realized in hardware. This is because context modeling of NRDSEs for a current macroblock involves using context information of neighboring macroblocks for selecting a probability distribution model. No such information is needed for the context modeling of RDSEs. When implemented in hardware, these context information needs to be transmitted through the system bus to the hardware. Although the number of NRDSEs is lower than the number of RDSEs, delay through system bus due to transmission of context information can still lower the efficiency of the hardware. The cost of calculation using an existing processor is much lower because all context information can be easily accessed from the buffers in the software.

BIBLIOGRAPHY

[1] ITU-T Recommendation H.264, "Advanced Video Coding for Generic Audiovisual Services", ITU, Mar 2005

[2] ISO/IEC 14496-2, "Information Technology – Coding of Audio-Visual Objects", Committee Draft, Mar 1998

[3] ITU-T Recommendation H.263, "Video Coding for Low Bit Rate Communication", ITU, 1995 (version 1), 1998 (version 2), 2000 (version 3)

[4] S. Saponara, K. Denolf, C. Blanch, G. Lafruit, and J. Bormans, "Performance and Complexity Co-evaluation of the Advanced Video Coding Standard for Cost-effective Multimedia Communications", *EURASIP*, vol. 2004, pp. 220-235, Feb. 2004

[5] D. Marpe, H.Schwarz and T. Wiegand, "Context-based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard", *IEEE Trans. Circuits Syst Video Technol.*, Vol. 13, No. 7, pp. 620-636, Jul 2003

[6] G. Stitt and F. Vahid., "Hardware/Software Partitioning of Software Binaries: A Case Study of H.264 Decode," *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 164-170, Nov. 2002.

[7] T. Wiegand, H. Schwarz, A. Joch, F. Kossentini and G. J. Sullivan, "Rate-Constrained Coder Control And Comparison of Video Coding Standards", *IEEE Trans. Circuits Systs. Video Technol.*, vol. 13, no. 7, pp. 688-702, July 2003

[8] P. Pushner and C. Koza, "Calculating the Maximum Execution Time of Real Time Program", *Journ. Trans. Signal Process.*, vol. 46, no.4, pp. 1027-1042, Apr 1998

[9] V. Lappalainen, A. Hallapuro and T. Hamalainen, “Complexity of Optimized H.26L Video Decoder Implementation”, *IEEE Circuits Syst. Video Technol.*, vol. 13, pp. 717-725, Jul 2003

[10] Dongarra, J., London, K., Moore, S., Mucci, P. and Terpstra, D., "Using PAPI for Hardware Performance Monitoring on Linux Systems," *Conference on Linux Clusters: The HPC Revolution*, Linux Clusters Institute, Urbana, Illinois, June 25-27, 2001.

[11] S. Graham, P. Kessler and M. McKusick, “Gprof: A Call Graph Execution Profiler”, *Proc. Symp. Compiler Construction (SIGPLAN)*, vol.17, pp.120-126, Jun 1982

[12] C. Xu, M.T. Le, T.T. Tay, “Instruction Level Complexity Analysis”, *IMSA 2005*, pp.341-346, Aug 2005

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005

[14] K. Denolf, P. Vos, J. Bormans, and I. Bolsens, “ Cost-efficient C-level Design of an MPEG-4 Video Decoder”, *Lecture Notes in Computer Scienc*, vol. 1918, pp. 233-242, Springer-Verlag, Heidelberg, Sep 2000.

[15] M.Ravasi and M. Mattavelli, “High-Abstraction Level Complexity Analysis And Memory Architecture Simulations of Multimedia Algorithms”, *IEEE Circuits Syst. Video Technol.*, vol. 15, pp. 673-684, May 2005

[16] H.J. Stolberg, M. Berekovic and P. Pirsch, “A Platform-independent Methodology for Performance Estimation of Streaming Media Applications”,

IEEE International Conference on Multimedia and Expo, pp. 105-108, Lausanne, Switzerland, Aug 2002.

[17] M. Horowitz, A. Kossentini and A. Hallapuro, "H.264/AVC Baseline Profile Decoder Complexity Analysis", *IEEE Circuits Syst. Video Technol.*, vol. 13, pp. 704-716, Jul 2003

[18] V. Lappalainen, A. Hallapuro and T. Hamalainen, "Performance Analysis of Low Bit-rate H.26L Video Encoder," *IEEE ICASSP'01*, pp. 1129-1132, May 2001

[19] F. Catthooer, *Custom Memory Management Methodology*, Kluwer Academic Publishers, 1998

[20] L. Nachtergaele, D. Moolenaar, B. Vanhoof, F. Catthoor and H. De Man, "System-level Power Optimization of Video Codec on Embedded Cores: A Systematic Approach," *Journ. VLSI Signal Processing*, vol.18, no. 2, pp. 89-111, 1998

[21] http://www.imec.be/design/atomium/

[22] D. Burger and T.M. Austin, "The SimpleScalar Tool Set", *Computer Architecture News*, pp. 13-15, June 1997.

[23] F. Pan, K. TW. Choo, and Thinh M. Le, "Fast Rate-Distortion Optimization in H.264/AVC Video Coding", *Knowledge-Based Intelligent Information & Engineering Systems: Multimedia Compression*, Springer – series of Lecture notes in Computer Sciences, pp. 425-432, 2005.

[24] http://rogue.colorado.edu/pin

[25] http://icl.cs.utk.edu/papi

[26] J. Osternabb, J. Bormans, P. List, D. Marple, M. Narroschke, P. Pereira, T. Stockhammer, and T. Wedi, "Video Coding with H.264/AVC: Tools,

Performance, and Complexity," *IEEE Circuits System Magazine*, First Quarter 2004

APPENDICES

A1: Instruction Set Architecture Class

Table A-1: Intel ISA used in the entropy coders and the video encoder

INT_ALU		COND_BR	STRINGOP	X87_ALU		FCMOV	CMOV
ADC	MUL	JB	CMPSB	FABS	FNSTCW	FCMOVE	CMOVB
ADD	NEG	JBE	MOVSB	FADD	FNSTSW	CALL	CMOVBE
AND	NOT	JL	MOVSD	FCHS	FRNDINT	CALL_NEAR	CMOVL
BSR	OR	JLE	MOVSW	FCOMP	FST	RET	CMOVLE
CDQ	RDTSC	JNB	SCASB	FDIV	FSTP	RET_NEAR	CMOVNB
CLD	SAHF	JNBE	STOSB	FDIVR	FSUB	INTERRUPT	CMOVNBE
CMP	SBB	JNL	STOSD	FDIVRP	FSUBR	INT	CMOVNLE
CWDE	SETBE	JNLE	SHIFT	FILD	FUCOM	POP	CMOVNS
DEC	SETL	JNP	SAR	FIST	FUCOMI	POP	CMOVNZ
DIV	SETLE	JNS	SHL	FISTP	FUCOMIP	PUSH	CMOVS
IDIV	SETNL	JNZ	SHLD	FLD	FUCOMP	PUSH	CMOVZ
IMUL	SETNLE	JP	SHR	FLD1	FUCOMPP	SEMAPHORE	
INC	SETNZ	JrCXZ	SHRD	FLDCW	FXAM	CMPXCHG	
LEA	SETZ	JS		FLDLG2	FXCH	XCHG	
LEAVE	SUB	JZ		FLDLN2	FYL2X		
MOV	TEST	UNCOND_BR		FLDZ	FYL2XP1		
MOVSX	XCHG	JMP		FMUL			
MOVZX	XOR			FMULP			

- The ISA class name are shaded in grey.

A2: ISA Classification for CIF Foreman

Table A2-1: ISA classification using configuration C6, CABAC and RDO

	CABAC Entropy Coder		Video Encoder	
Op code	No. of entropy instructions	% of total entropy instructions	No. of instructions	% of total instructions
INT_ALU				
ADC	-	-	1876055893	0.11433
ADD	**10732713484**	**4.15798**	**179643209868**	**10.94767**
AND	2001504542	0.77541	7224193942	0.44025
BSF	-	-	983	0.00000
BSR	-	-	1977	0.00000
BSWAP	-	-	22	0.00000
CDQ	384799928	0.14908	3502478637	0.21345
CLD	371773962	0.14403	806105776	0.04913
CMP	**10423891946**	**4.03834**	**61958655151**	**3.77583**
CWDE	-	-	344124984	0.02097
DEC	3583840025	1.38842	14979411997	0.91286
DIV	384799928	0.14908	812646960	0.04952
IDIV	384799928	0.14908	1626432819	0.09912
IMUL	98846573	0.03829	8625459524	0.52565
INC	3976202285	1.54043	36061083660	2.19761
LEA	7121818932	2.75908	43934824342	2.67744
LEAVE	-	-	309	0.00000
MOV	**110446701410**	**42.78837**	**739103681163**	**45.04186**
MOVSX	1576391111	0.61071	4568302493	0.27840
MOVZX	7303831667	2.82959	97568682252	5.94595
MUL	-	-	12895	0.00000
NEG	91618539	0.03549	7655087023	0.46651
NOT	-	-	21802396	0.00133
OR	816829717	0.31645	959126407	0.05845
RDTSC	-	-	1	0.00000
SAHF	-	-	154564275	0.00942
SBB	-	-	1704654	0.00010
SETBE	81606050	0.03162	81960866	0.00499
SETL	-	-	346785253	0.02113
SETLE	-	-	303	0.00000
SETNBE	-	-	1007	0.00000
SETNL	2140903	0.00083	23604883	0.00144
SETNLE	257496426	0.09976	350829551	0.02138
SETNZ	1641175884	0.63581	1838631438	0.11205
SETS	-	-	1	0.00000
SETZ	506767541	0.19633	534652317	0.03258
SUB	7570209150	2.93279	87958850967	5.36032
TEST	**11661086734**	**4.51764**	**50548656773**	**3.08050**
XOR	2104028034	0.81513	8968740331	0.54657
COND_BR				
JB	443113	0.00017	788865	0.00005
JBE	2675652033	1.03658	2811426385	0.17133
JL	394708763	0.15291	5624559247	0.34277

JLE	912565282	0.35354	31617705733	1.92682
JNB	38453046	0.01490	706208288	0.04304
JNBE	2354425539	0.91213	2487644247	0.15160
JNL	491408343	0.19038	10420902048	0.63506
JNLE	479824304	0.18589	4126609601	0.25148
JNP	-	-	21260448	0.00130
JNS	798802920	0.30947	1859831278	0.11334
JNZ	4505546742	1.74550	15364363057	0.93632
JP	-	-	93288	0.00001
JrCXZ	87766074	0.03400	171358780	0.01044
JS	933104266	0.36150	21082077739	1.28477
JZ	7604261599	2.94598	23702941401	1.44449
UNCOND_BR				
JMP	3243812650	1.25669	20295519651	1.23683
INTERRUPT				
INT	-	-	27	0.00000
PUSH / POP				
POP	**16514466706**	**6.39790**	**35642772950**	**2.17211**
PUSH	**14744121125**	**5.71205**	**34555448316**	**2.10585**
CALL/ RET				
CALL_NEAR	5146471043	1.99380	15202826371	0.92648
RET_NEAR	5287401421	2.04840	15154745473	0.92355
SEMAPHORE				
CMPXCHG	-	-	216472	0.00001
XCHG	1157092478	0.44827	6819790064	0.41561
SHIFT				
SAR	1104679930	0.42797	11052119739	0.67353
SHL	3830869470	1.48412	11085893687	0.67559
SHLD	213249135	0.08262	223257007	0.01361
SHR	1457122309	0.56451	4961586082	0.30236
SHRD	162496854	0.06295	162575610	0.00991
STRINGOP				
CMPSB	-	-	11	0.00000
MOVSB	-	-	90935670	0.00554
MOVSD	284007888	0.11003	660120049	0.04023
MOVSW	-	-	23775985	0.00145
SCASB	-	-	1340	0.00000
STOSB	87766074	0.03400	185785862	0.01132
STOSD	87766074	0.03400	157857367	0.00962
CMOV				
CMOVB	-	-	107203	0.00001
CMOVBE	-	-	1340	0.00000
CMOVL	-	-	203	0.00000
CMOVLE	-	-	1022	0.00000
CMOVNB	-	-	300	0.00000
CMOVNBE	-	-	20559	0.00000
CMOVNLE	-	-	3756	0.00000
CMOVNS	-	-	1780	0.00000
CMOVNZ	-	-	101300	0.00001
CMOVS	-	-	1968	0.00000
CMOVZ	-	-	3311	0.00000
FCMOVE				

FCMOVE	-	-	7	0.00000
X87_ALU				
F2XM1	-	-	50830	0.00000
FABS	-	-	51987	0.00000
FADD	-	-	5078477	0.00031
FADDP	-	-	67954525	0.00414
FCHS	-	-	7	0.00000
FCOMP	-	-	51987	0.00000
FDIV	-	-	19676879	0.00120
FDIVP	-	-	10493	0.00000
FDIVR	-	-	22389837	0.00136
FDIVRP	-	-	19630543	0.00120
FILD	-	-	312169673	0.01902
FIST	-	-	116	0.00000
FISTP	-	-	41231479	0.00251
FLD	-	-	660314233	0.04024
FLD1	-	-	713423	0.00004
FLDCW	-	-	85909116	0.00524
FLDLG2	-	-	1153	0.00000
FLDLN2	-	-	4	0.00000
FLDZ	-	-	1416979	0.00009
FMUL	-	-	157739417	0.00961
FMULP	-	-	963821	0.00006
FNSTCW	-	-	23325104	0.00142
FNSTSW	-	-	154721978	0.00943
FRNDINT	-	-	1850812	0.00011
FSCALE	-	-	50830	0.00000
FSQRT	-	-	93288	0.00001
FST	-	-	213611	0.00001
FSTP	-	-	591840922	0.03607
FSUB	-	-	44792531	0.00273
FSUBR	-	-	1426621	0.00009
FUCOM	-	-	3402628	0.00021
FUCOMI	-	-	80684	0.00000
FUCOMIP	-	-	79527	0.00000
FUCOMP	-	-	49419147	0.00301
FUCOMPP	-	-	101690517	0.00620
FXAM	-	-	157699	0.00001
FXCH	-	-	203975821	0.01243
FYL2X	-	-	51934	0.00000
FYL2XP1	-	-	53	0.00000
Total	258123159880	100.00000	1640926133666	100.00000

(Instructions that are executed frequently is highlighted in bold)

Table A2-2: ISA Classification using configuration C6, CAVLC and RDO

	CAVLC Entropy Coder		Video Encoder	
Op code	No. of entropy instructions	% of total entropy instructions	No. of instructions	% of total instructions
INT_ALU				
ADC	-	-	1875155833	0.11891
ADD	**8974011088**	**4.68502**	**178353943382**	**11.31037**
AND	1237711665	0.64617	6465829212	0.41003
BSF	-	-	981	0.00000
BSR	-	-	1980	0.00000
BSWAP	-	-	22	0.00000
CDQ	401178789	0.20944	3516263630	0.22298
CLD	158865498	0.08294	592074445	0.03755
CMP	**5921325084**	**3.09132**	**57531643520**	**3.64839**
CWDE	-	-	344124984	0.02182
DEC	3421714601	1.78636	14852217178	0.94186
DIV	321203037	0.16769	748447377	0.04746
IDIV	401178789	0.20944	1641117908	0.10407
IMUL	12634980	0.00660	8566398352	0.54324
INC	2736535969	1.42865	34831898862	2.20888
LEA	4834480004	2.52392	41680517280	2.64318
LEAVE	-	-	309	0.00000
MOV	**87700139163**	**45.78522**	**717303996014**	**45.48804**
MOVSX	356676288	0.18621	3350892550	0.21250
MOVZX	2221256511	1.15964	92733970579	5.88075
MUL	-	-	12948	0.00000
NEG	284755695	0.14866	7861034404	0.49851
NOT	150957268	0.07881	172636176	0.01095
OR	2077684887	1.08469	2230678241	0.14146
RDTSC	-	-	1	0.00000
SAHF	7935657	0.00414	163100750	0.01034
SBB	-	-	1704676	0.00011
SETBE	-	-	354816	0.00002
SETL	-	-	346972771	0.02200
SETLE	11249608	0.00587	11249911	0.00071
SETNBE	-	-	1005	0.00000
SETNL	-	-	21399996	0.00136
SETNLE	94388712	0.04928	188756426	0.01197
SETNZ	39613608	0.02068	237239693	0.01504
SETS	-	-	1	0.00000
SETZ	25399626	0.01326	53088925	0.00337
SUB	3989224709	2.08264	84560606088	5.36244
TEST	**10416286274**	**5.43798**	**49353533903**	**3.12977**
XOR	2882275102	1.50474	9749436797	0.61826
COND_BR				
JB	2	0.00000	3305710	0.00021
JBE	84974337	0.04436	220387560	0.01398
JL	1382831093	0.72193	6607078044	0.41899
JLE	1203073731	0.62808	31916748712	2.02401
JNB	95420573	0.04982	763868432	0.04844
JNBE	165196759	0.08624	298503820	0.01893
JNL	995332845	0.51963	10963084219	0.69523

JNLE	629097024	0.32843	4311302941	0.27340
JNP	-	-	21260448	0.00135
JNS	1529232736	0.79836	2590238826	0.16426
JNZ	3437846766	1.79478	14340261407	0.90939
JP	-	-	93288	0.00001
JrCXZ	6158128	0.00321	89754208	0.00569
JS	2239670957	1.16926	22403721376	1.42074
JZ	6960056286	3.63361	23077988369	1.46350
JMP	2673194139	1.39558	19745962620	1.25220
INTERRUPT				
INT	-	-	27	0.00000
PUSH / POP				
POP	**8437226192**	**4.40479**	**27620973193**	**1.75159**
PUSH	**7866006060**	**4.10657**	**27727551519**	**1.75835**
CALL / RET				
CALL_NEAR	3883956400	2.02768	13974535699	0.88620
RET_NEAR	4004282404	2.09050	13904314649	0.88175
SEMAPHORE				
CMPXCHG	-	-	215232	0.00001
XCHG	516721097	0.26976	6201699497	0.39328
SHIFT				
SAR	1783938998	0.93133	11740871356	0.74455
SHL	2173206718	1.13456	9425290046	0.59771
SHLD	-	-	9960850	0.00063
SHR	1553666921	0.81112	5072138476	0.32165
SHRD	13699262	0.00715	15135725	0.00096
STRINGOP				
CMPSB	-	-	11	0.00000
MOVSB	-	-	90935672	0.00577
MOVSD	70986059	0.03706	445179464	0.02823
MOVSW	-	-	23717314	0.00150
SCASB	81721311	0.04266	81722651	0.00518
STOSB	6158128	0.00321	104181290	0.00661
STOSD	6158128	0.00321	76252795	0.00484
CMOV				
CMOVB	-	-	106584	0.00001
CMOVBE	81721311	0.04266	81722574	0.00518
CMOVL	-	-	203	0.00000
CMOVLE	-	-	1020	0.00000
CMOVNB	-	-	300	0.00000
CMOVNBE	81721311	0.04266	81741780	0.00518
CMOVNLE	-	-	3754	0.00000
CMOVNS	-	-	1781	0.00000
CMOVNZ	-	-	100660	0.00001
CMOVS	-	-	1968	0.00000
CMOVZ	-	-	3310	0.00000
FCMOVE				
FCMOVE	-	-	7	0.00000
X87_ALU				
F2XM1	-	-	50830	0.00000
FABS	-	-	51987	0.00000

FADD	-	-	4365353	0.00028
FADDP	-	-	67952181	0.00431
FCHS	-	-	7	0.00000
FCOMP	-	-	51987	0.00000
FDIV	-	-	19676879	0.00125
FDIVP	-	-	10493	0.00000
FDIVR	-	-	22221646	0.00141
FDIVRP	-	-	19630543	0.00124
FILD	28344659	0.01480	341900718	0.02168
FIST	-	-	116	0.00000
FISTP	28344659	0.01480	72200351	0.00458
FLD	219961334	0.11483	895701912	0.05680
FLD1	-	-	299	0.00000
FLDCW	40818004	0.02131	130189372	0.00826
FLDLG2	-	-	1153	0.00000
FLDLN2	-	-	4	0.00000
FLDZ	40818004	0.02131	44984111	0.00285
FMUL	23148569	0.01209	180126234	0.01142
FMULP	-	-	291009	0.00002
FNSTCW	20409002	0.01065	45465232	0.00288
FNSTSW	36280316	0.01894	194227325	0.01232
FRNDINT	-	-	1850812	0.00012
FSCALE	-	-	50830	0.00000
FSQRT	-	-	93288	0.00001
FST	40818004	0.02131	44493867	0.00282
FSTP	232434679	0.12135	838572761	0.05318
FSUB	-	-	44456149	0.00282
FSUBR	-	-	373	0.00000
FUCOM	-	-	3397940	0.00022
FUCOMI	20409002	0.01065	22220812	0.00141
FUCOMIP	20409002	0.01065	22219655	0.00141
FUCOMP	7935657	0.00414	57694424	0.00366
FUCOMPP	-	-	101956403	0.00647
FXAM	28344659	0.01480	31126571	0.00197
FXCH	116405936	0.06077	327230007	0.02075
FYL2X	-	-	51934	0.00000
FYL2XP1	-	-	53	0.00000
Total	191546819774	100.00000	1576906736934	100.00000

(Instructions that are executed frequently is highlighted in bold)

A3: Pin Tools Program Codes

```
/*      INS.c
*===============================================
*       Instruction count / Function Call Frequency
*===============================================
*/

#include <iostream>
#include <fstream>
#include <iomanip>
#include "pin.H"


#define CNT_SIZE 14
#define TOTINS 0

UINT64 functcall[CNT_SIZE];
UINT64 counter[CNT_SIZE];
UINT32 status[CNT_SIZE];
UINT32 semaphore = 0 ;
UINT8 bisymbol=0;
UINT32 CNT_SIZE_MINUS_ONE = CNT_SIZE -1;
UINT8 flag=0;
UINT64 temp_counter=0;

string F1 = "writeIPredMode_CABAC";
string F2 = "Get_Rate_8x8blk";
string F3 = "write_one_macroblock";
string F4 = "writeMBLayer";
string F4 = "Get_Rate_MB";
string F5 = "store_coding_state";
string F6 = "reset_coding_state";
string F7 = "CheckAvailabilityOfNeighborsCABAC";
string F8 = "terminate_slice";


void Global_Init()
{
        int i;
        for (i=0;i<CNT_SIZE;i++)
        {
                functcall[i] =0;
                counter[i]=0;
                status[i]= 0;
        }
        status[TOTINS]=1;
}

void docount(UINT32 idx, UINT32 state)
{
        if (state==1)
                flag=1;
```

```
	if(flag)
		counter[idx]++;

	if(state==0)
	{
		flag=0;
		functcall[idx]++;
	}
}


void checkdocount(INT32 c)
{

	counter[TOTINS] += c;

	if(semaphore>0)
	{
		counter[semaphore] += c;

	}
}



void switch_status(UINT8 idx, UINT8 state)
{
	if(state == 1)
	{
		functcall[idx]++;
	}

	//single semaphore
	if(semaphore == 0 && state==1)
	{
		semaphore = idx;
	}
	else if (idx == semaphore && state == 0)
	{
		semaphore = 0;
	}

	//more than one semaphores
	//if(semaphore < idx && state==1)
	//		semaphore = idx;
	//else if(semaphore == idx && state ==0)
	//		semaphore--;

}



#define START_STOP(funct_num)\
{\
```

```
	if(RTN_Name(rtn).c_str()==F ## funct_num)\
	{\
	INS_InsertCall(ins_head, IPOINT_BEFORE, (AFUNPTR)docount,IARG_UINT32,
funct_num, IARG_UINT32, 1, IARG_END);\
	for(INS ins=INS_Next(ins_head);INS_Valid(ins);ins=INS_Next(ins))\
	{\
		if(INS_IsRet(ins))\
			INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount,
IARG_UINT32, funct_num, IARG_UINT32, 0, IARG_END);\
		else\
			INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount,
IARG_UINT32, funct_num, IARG_UINT32, 1, IARG_END);\
		}\
}\
else{\
INS_InsertCall(ins_head, IPOINT_BEFORE, (AFUNPTR)docount,IARG_UINT32, 2,
IARG_UINT32, 2, IARG_END);\
	for(INS ins=INS_Next(ins_head);INS_Valid(ins);ins=INS_Next(ins))\
		{\
			INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount,
IARG_UINT32, 2, IARG_UINT32, 2, IARG_END);\
		}\
	}\
}


void Routine(RTN rtn, void *v)
{
	INS ins_head;

	RTN_Open(rtn);
	ins_head = RTN_InsHead(rtn);
	START_STOP(1);
	START_STOP(2);
	START_STOP(3);
	START_STOP(4);
	START_STOP(5);
	START_STOP(6);
	START_STOP(7);
	START_STOP(8);
	RTN_Close(rtn);
}


void Trace(TRACE trace, void *v)
{
   RTN rtn = TRACE_Rtn(trace);

   for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
   {
      BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)checkdocount,
IARG_UINT32, BBL_NumIns(bbl), IARG_END);

   }
```

```
}


#define PRINT2FILE(K, out)\
{\
        out << F ## K << " = " << counter[K] << " | " << functcall[K] << endl;\
}


void Fini(INT32 code, VOID *v)
{

        ofstream out("INS.out",ios_base::app);

        out << "Tot instruction = "  << counter[0]/1000000 << endl;
        PRINT2FILE(1, out);
        PRINT2FILE(2, out);
        PRINT2FILE(3, out);
        PRINT2FILE(4, out);
        PRINT2FILE(5, out);
        PRINT2FILE(6, out);
        PRINT2FILE(7, out);
        PRINT2FILE(8, out);
        out << "Tot Coding instruction = " <<   compute_tot_codingins(&percentage) <<
endl;
        out << setw(30) << "Percentage    = " <<  percentage << endl;
        out.close();

}



int main(int argc, char * argv[])
{

    PIN_InitSymbols();

    PIN_Init(argc, argv);

   Global_Init();

   RTN_AddInstrumentFunction(Routine,0);

   TRACE_AddInstrumentFunction(Trace, 0);

    PIN_AddFiniFunction(Fini, 0);

    PIN_StartProgram();

    return 0;
}
```

```
/* MA.c
*================================================================
*
*  Memory access in functional blocks
*
*================================================================
*/

#include "pin.H"
#include "instlib.H"
#include <unistd.h>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>

using namespace INSTLIB;

typedef UINT64 COUNTER;

#define CNT_SIZE 14

COUNTER MemRead ;
COUNTER MemWrite;
COUNTER MemTotal;
UINT32 semaphore = 0;

string F1 = "Get_Rate_4x4Intrablk";
string F2 = "Get_Rate_8x8blk";
string F3 = "writeMBLayer";
string F4 = "store_coding_state";
string F5 = "reset_coding_state";


typedef class BBLSTATS
{
        public:
        COUNTER _bblcnt;
        COUNTER _mem_rd_cnt;
        COUNTER _mem_wr_cnt;

        public:
                BBLSTATS(COUNTER mem_rd_cnt, COUNTER
mem_wr_cnt):_mem_rd_cnt(mem_rd_cnt), _mem_wr_cnt(mem_wr_cnt)
                {
                        _bblcnt=0;
                }
};

vector<const BBLSTATS*> statsList;


void docount(COUNTER* counter)
{
        if(semaphore > 0)
```

```
        {
                (*counter)++;

        }
}


void switch_status(UINT8 idx, UINT8 state)
{
        if(semaphore == 0 && state ==1)
        {
                semaphore = idx;
        }
        else if(semaphore == idx && state == 0)
        {
                semaphore = 0;
        }

}


void Global_Init()
{
                MemRead = 0;
                MemWrite = 0;
                MemTotal = 0;
}


void Trace(TRACE trace, VOID *v)
{
   for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
   {
        COUNTER cnt_mem_rd = 0;
        COUNTER cnt_mem_wr = 0;

        const INS head = BBL_InsHead(bbl);

        if (! INS_Valid(head)) continue;


        for (INS ins = head; INS_Valid(ins); ins = INS_Next(ins))
        {
                        if(INS_IsMemoryRead(ins))
                                cnt_mem_rd ++;

                        if(INS_IsMemoryWrite(ins))
                                cnt_mem_wr++;
          }

        BBLSTATS* bblstats = new BBLSTATS(cnt_mem_rd, cnt_mem_wr);
        INS_InsertCall(head,    IPOINT_BEFORE,    (AFUNPTR)docount,    IARG_PTR,
&(bblstats->_bblcnt), IARG_END);
        statsList.push_back(bblstats);
  }
```

```
}


void Fini(int, VOID * v)
{
	std::ofstream out("mem_access.txt",ios_base::app);
	COUNTER total_access_mem = 0;

	statsList.push_back(0);

	for(vector<const BBLSTATS*>::iterator bi = statsList.begin(); bi!=statsList.end();
bi++)
	{
		const BBLSTATS* bblstats = (*bi);
		if(bblstats==0) continue;

		MemRead += (bblstats->_bblcnt * bblstats->_mem_rd_cnt);
		MemWrite += (bblstats->_bblcnt * bblstats->_mem_wr_cnt);

	}
	MemTotal = MemRead + MemWrite;
	total_access_mem += MemRead;
	total_access_mem += MemWrite;

out << "Mem [rd] [wr] [tot] access" << " ; " << MemRead/1000 << " ; " << MemWrite/1000
<< " ; " << MemTotal/1000 << endl;\
	out << "Mem [rd] [wr] [tot] access" << " ; " <<
(float)MemRead/(float)MemTotal*100.0 << " ; " << (float)MemWrite/(float)MemTotal
*100.0<< " ; " << (float)MemTotal/(float)MemTotal*100.0 << endl;\
 out.close();
}


#define START_STOP(funct_num)\
{\
	if(RTN_Name(rtn).c_str()==F ## funct_num)\
	{\
		INS_InsertCall(ins_head, IPOINT_BEFORE, (AFUNPTR)switch_status,
IARG_UINT32, funct_num, IARG_UINT32, 1, IARG_END);\
		for(INS ins=INS_Next(ins_head);INS_Valid(ins);ins=INS_Next(ins))\
		{	\
			if(INS_IsRet(ins))\
			{\
			INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)switch_status,
IARG_UINT32, funct_num, IARG_UINT32, 0, IARG_END);\
			printf("Return instruction found for %s\n",(F ## funct_num).c_str());\
			}\
		}\
	}\
}



void Routine(RTN rtn, VOID * v)
{
```

```
        INS ins_head;

        RTN_Open(rtn);
        ins_head = RTN_InsHead(rtn);
        START_STOP(1);
        START_STOP(2);
        START_STOP(3);
        START_STOP(4);
        START_STOP(5);
        RTN_Close(rtn);

}


int main(int argc, CHAR *argv[])
{
        PIN_InitSymbols();

        PIN_Init(argc, argv);

        Global_Init();

        RTN_AddInstrumentFunction(Routine,0);

        TRACE_AddInstrumentFunction(Trace, 0);

        PIN_AddFiniFunction(Fini, 0);

        PIN_StartProgram();

        return 0;
}
```

```
/*         ISA.c
/* ======================
/* ISA Class Classfication
/* ======================
*/

#include "pin.H"
#include "instlib.H"
#include <unistd.h>
#include <vector>
#include <iostream>
#include <iomanip>
#include <fstream>

using namespace INSTLIB;

typedef COUNTER COUNTER;

const UINT32 MAX_INDEX = 4096;
const UINT32 INDEX_SPECIAL = 3000;

UINT32 semaphore = 0;

string F1 = "Get_Rate_4x4Intrablk";
string F2 = "Get_Rate_8x8blk";
string F3 = "writeMBLayer";
string F4 = "store_coding_state";
string F5 = "reset_coding_state";

UINT32 StringLength(BBL bbl, BOOL memory_acess_profile)
{
   UINT32 count = 0;

   for (INS ins = BBL_InsHead(bbl); INS_Valid(ins); ins = INS_Next(ins))
                    count++;
   return count;
}


class STATS
{
 public:
   COUNTER instr[MAX_INDEX];
   COUNTER category[MAX_INDEX];

   void Clear()
   {
      for ( UINT32 i = 0; i < MAX_INDEX; i++)
      {
         instr[i] = 0;
         category[i] = 0;

      }
   }
};
```

```
STATS GlobalStats;

typedef class BBLSTATS
{
 public:
   COUNTER _counter;
   const UINT16 * const _stats;
        const UINT16 * const _stats_category;

 public:
   BBLSTATS(UINT16 * stats, UINT16 * stats_category) : _counter(0), _stats(stats),
_stats_category(stats_category) {};

};


LOCALVAR vector<const BBLSTATS*> statsList;


void docount(COUNTER * counter)
{
        if(semaphore > 0)
                (*counter) ++;
}


void switch_status(UINT8 idx, UINT8 state)
{
        if(semaphore == 0 && state ==1)
        {
                semaphore = idx;
        }
        else if(semaphore == idx && state == 0)
        {
                semaphore = 0;
        }

}


#define START_STOP(funct_num)\
{\
        if(RTN_Name(rtn).c_str()==F ## funct_num)\
        {\
                INS_InsertCall(ins_head,    IPOINT_BEFORE,    (AFUNPTR)switch_status,
IARG_UINT32, funct_num, IARG_UINT32, 1, IARG_END);\
                for(INS ins=INS_Next(ins_head);INS_Valid(ins);ins=INS_Next(ins))\
                {       \
                        if(INS_IsRet(ins))\
                        {\
                        INS_InsertCall(ins,   IPOINT_BEFORE,   (AFUNPTR)switch_status,
IARG_UINT32, funct_num, IARG_UINT32, 0, IARG_END);\
                        }\
```

```
                }\
        }\
}


VOID Routine(RTN rtn, VOID * v)
{
        INS ins_head;

        RTN_Open(rtn);
        ins_head = RTN_InsHead(rtn);
        START_STOP(1);
        START_STOP(2);
        START_STOP(3);
        START_STOP(4);
        START_STOP(5);
        RTN_Close(rtn);

}


VOID Trace(TRACE trace, VOID *v)
{

   for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
   {
      INS head = BBL_InsHead(bbl);
      if (! INS_Valid(head)) continue;

            UINT32 n = StringLength(bbl, 1);

      UINT16 *stats = new UINT16[ n + 1];
      UINT16 *stats_end = stats + (n + 1);
      UINT16 *curr = stats;

       UINT16 *stats_category = new UINT16[n + 1];
       UINT16 *stats_category_end =  stats_category + (n + 1);
       UINT16 *curr_category = stats_category;


      for (INS ins = head; INS_Valid(ins); ins = INS_Next(ins))
      {
                *curr++ = INS_Opcode(ins);
                        *curr_category++ = INS_Category(ins);
      }


      *curr++ = 0;
       *curr_category++ = 0;

      ASSERTX( curr == stats_end );
      ASSERTX( curr_category == stats_category_end );


      BBLSTATS * bblstats = new BBLSTATS(stats, stats_category);
```

```
      INS_InsertCall(head, IPOINT_BEFORE, AFUNPTR(docount), IARG_PTR, &(bblstats->_counter), IARG_END);
      statsList.push_back(bblstats);
   }
}


void PrintStats(ofstream& out, STATS& stats)
{


   for ( UINT32 i = 0; i < INDEX_SPECIAL ; i++)
   {
      stats.instr[INDEX_SPECIAL] += stats.instr[i];
                  stats.category[INDEX_SPECIAL]+= stats.category[i];
   }

   for ( UINT32 i = 0; i < INDEX_SPECIAL; i++)
   {
      if( stats.instr[i] == 0) continue;

      out << i << " " <<  ljstr(OPCODE_StringShort(i),20) << " " << stats.instr[i] << endl;
      cout << i << " " << ljstr(OPCODE_StringShort(i),20) << " " << stats.instr[i] << endl;

   }

   for ( UINT32 i = 0; i < INDEX_SPECIAL; i++)
   {
      if( stats.category[i] == 0) continue;

      out << i << " " <<  ljstr(CATEGORY_StringShort(i),20) << " " << stats.category[i] << endl;
      cout << i << " " << ljstr(CATEGORY_StringShort(i),20) << " " << stats.category[i] << endl;

   }
       out << "Total " << stats.instr[INDEX_SPECIAL]/1000000 << endl;
       out  << "Total" << stats.category[INDEX_SPECIAL]/1000000 << endl;

}


void Fini(int, VOID * v)
{

   std::ofstream out("ISA.txt", ios_base::app);
   statsList.push_back(0);


   for (vector<const BBLSTATS*>::iterator bi = statsList.begin(); bi != statsList.end(); bi++)
   {
      const BBLSTATS *b = (*bi);

      if ( b == 0 ) continue;
```

```
        for (const UINT16 * stats = b->_stats; *stats; stats++)
        {
           GlobalStats.instr[*stats] += b->_counter;
        }

        for (const UINT16 * stats_category = b->_stats_category; *stats_category;
stats_category++)
        {
           GlobalStats.category[*stats_category] += b->_counter;
        }
     }

   PrintStats(out, GlobalStats);
   out.close();
}

int main(int argc, CHAR *argv[])
{
   PIN_InitSymbols();

   PIN_Init(argc,argv);

    RTN_AddInstrumentFunction(Routine,0);

   TRACE_AddInstrumentFunction(Trace, 0);

   PIN_AddFiniFunction(Fini, 0);

   PIN_StartProgram();

   return 0;
}
```

```
/*      MU.c
 ==================================================================
          Trace dynamic memory allocation and compute peak memory usage
 ==================================================================
*/


#include "pin.H"
#include "instlib.H"
#include "unistd.h"
#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>

using namespace INSTLIB;

std::ofstream op("PMU.txt");
UINT32 Cur_mem_type =0;
ADDRINT Cur_mem_size =0;
ADDRINT Tot_mem_size =0;
ADDRINT Peak_mem_size = 0;

typedef class MEMSTAT
{
          public:
          ADDRINT _mem_addr;
          ADDRINT _mem_size;

          public:
          MEMSTAT(ADDRINT addr,ADDRINT size):_mem_addr(addr),_mem_size(size){}
};

vector<MEMSTAT*> statList;


void dyn_mem_size(UINT32 dyn_mem_funct, ADDRINT mem_size, ADDRINT elem_size)
{

          if(dyn_mem_funct == 1)
          {
                    Cur_mem_size = mem_size;
                    Cur_mem_type = dyn_mem_funct;
          }
          else
          {
                    Cur_mem_size =  mem_size * elem_size;
                    Cur_mem_type = dyn_mem_funct;
          }
}

void dyn_mem_addr(UINT32 dyn_mem_funct, ADDRINT mem_addr)
{
          ADDRINT stored_mem_size = 0;
          INT32 pos;
```

```
	UINT32 p;

	if(dyn_mem_funct==Cur_mem_type)
	{
		Tot_mem_size += Cur_mem_size;
		MEMSTAT* memstat = new MEMSTAT(mem_addr, Cur_mem_size);
		statList.push_back(memstat);

		if(Tot_mem_size > Peak_mem_size)
			Peak_mem_size = Tot_mem_size;

	}
	else if (dyn_mem_funct == 0)
	{
		p = 0;
		pos = -1;
		for(vector< MEMSTAT*>::iterator bi= statList.begin(); bi!=statList.end();
bi++,p++)
		{
			MEMSTAT* ms = *bi;
			if(ms->_mem_addr == mem_addr)
			{
				pos = p;
				stored_mem_size = ms->_mem_size;
				Tot_mem_size -= stored_mem_size;
				ms->_mem_addr = 0;
				ms->_mem_size = 0;
			}
		}
		if(p>=0)
		statList.erase(statList.begin()+pos);
	}
}


void Image(IMG img, void *v)
{
	RTN rtn;

	rtn = RTN_FindByName(img,"malloc");
	if(RTN_Valid(rtn))
	{
		RTN_Open(rtn);
		RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)dyn_mem_size,
IARG_UINT32, 1, IARG_G_ARG0_CALLEE, IARG_ADDRINT, 0, IARG_END);
		RTN_InsertCall(rtn, IPOINT_AFTER, (AFUNPTR)dyn_mem_addr,
IARG_UINT32, 1, IARG_G_RESULT0, IARG_END);
		RTN_Close(rtn);
	}

	rtn = RTN_FindByName(img,"calloc");
	if(RTN_Valid(rtn))
	{
		RTN_Open(rtn);
```

```
                RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)dyn_mem_size,
IARG_UINT32, 2, IARG_G_ARG1_CALLEE, IARG_G_ARG0_CALLEE, IARG_END);
                RTN_InsertCall(rtn, IPOINT_AFTER, (AFUNPTR)dyn_mem_addr,
IARG_UINT32, 2, IARG_G_RESULT0, IARG_END);
                RTN_Close(rtn);
        }

        rtn = RTN_FindByName(img,"free");
        if(RTN_Valid(rtn))
        {
                RTN_Open(rtn);
                RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)dyn_mem_addr,
IARG_UINT32, 0, IARG_G_ARG0_CALLEE, IARG_END);
                RTN_Close(rtn);
        }


}


void Fini(int, void *v)
{
        COUNTER count=0;
        COUNTER memsize=0;

        op << dec << noshowbase << "Peak Memory Usage = " << Peak_mem_size << endl;
        op << "Total Memory Leakage = " << Tot_mem_size << endl;

        for (vector<MEMSTAT*>::iterator bi =  statList.begin();bi!=statList.end(); bi++)
        {
                MEMSTAT* ms = *bi;
                if(ms==0) continue;
                memsize += ms->_mem_size;
                count++;
                op << count << " : " << ms->_mem_size << endl;
        }

        op << "Total mem left in statlist = " << memsize << endl;
        op << "statlist size = " << count << endl;

        op.close();

}


int main(int argc, char** argv)
{

        PIN_InitSymbols();

        PIN_Init(argc, argv);

        IMG_AddInstrumentFunction(Image, 0);

        PIN_AddFiniFunction(Fini, 0);
```

```
	PIN_StartProgram();

	return 0;
}


/*  CM.c
/ ==========================
/	Codec memory size
/ ==========================
*/

#include "pin.H"
#include "instlib.H"
#include <iostream>
#include <fstream>


std::ofstream op("codec_memsize.txt");

void Image(IMG img, void *v)
{

	op << ljstr("Image name",15) << " = " << IMG_Name(img) << endl;
	op << ljstr("Low address",15) << " = " << IMG_LowAddress(img) << endl;
	op << ljstr("High address",15) << " = " << IMG_HighAddress(img) << endl;
	op << ljstr("Image size", 15) << " = " << dec << noshowbase << IMG_SizeMapped(img) << endl;

}


int main(int argc, char ** argv)
{
	PIN_InitSymbols();

	PIN_Init(argc, argv);

	IMG_AddInstrumentFunction(Image, 0);

	PIN_StartProgram();

	return 0;
}
```