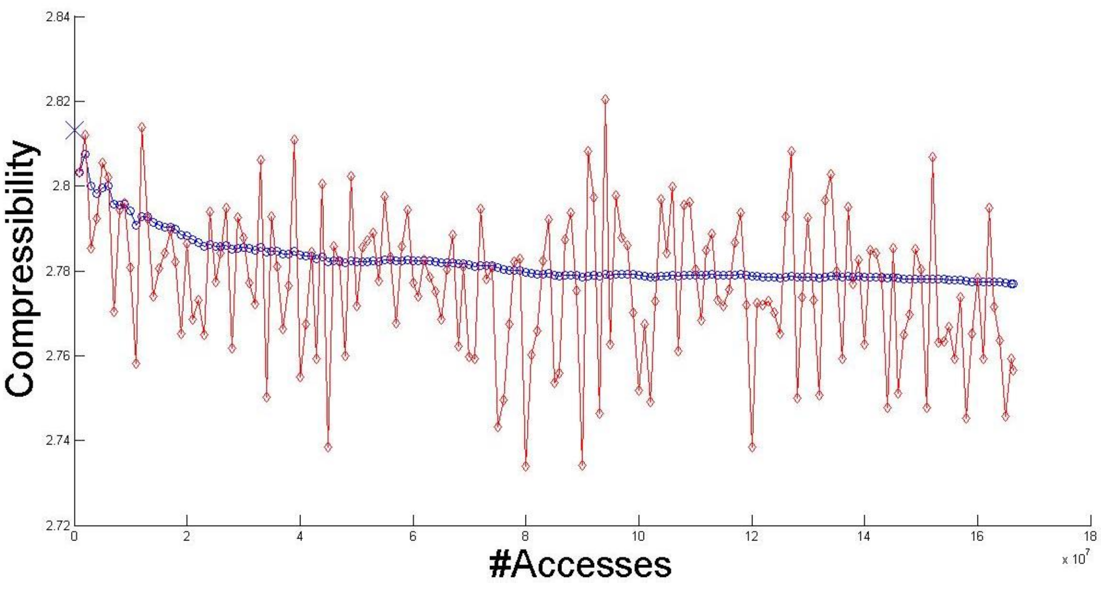
CHALMERS



Evaluation of Huffman coding in memory compression

Master of Science Thesis in Embedded Electronic System Design

YU JIANG

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, March 2014

Evaluation of Huffman coding in memory compression

YU JIANG

Examiner: Angelos Arelakis, Per Stenström

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
The compressibility variation of Huffman-based compression scheme during the execution of benchmark application omnetpp, Page 62

Department of Computer Science and Engineering
Göteborg, Sweden March 2014

Abstract

One of the main limitations of computer-systems' performance has been for almost three decades the speed gap between processors and memory, known as *memory wall*. Despite the slow improvement in memory's speed, the requirements for larger memory capacity are continuously growing. A costly solution, both in terms of area and power dissipation, is to increase the physical memory capacity. In alternative, the same physical space could be maintained, if the data were stored in a more efficient way; in other words if they were stored more densely by exploiting lossless compression techniques. However, such techniques need to encode data before storing them in memory and decode compressed data before accessing them. Therefore, the encoding and decoding operations may add further delay to the already long memory latency resulting in potential performance losses, as the memory access time becomes longer. Thus, a trade-off between the benefits in memory space (measured using *compressibility*) and losses in performance (typically measured in *execution time*) is often needed when considering memory compression solutions.

This master thesis makes a motivational analysis of using a Huffman-based compression approach in the main memory. We first trace and record the memory footprint when executing different applications from the SPEC2006 benchmark suite. Based on the obtained trace, we dynamically establish the value frequency distributions and generate Huffman coding to evaluate the potential of footprint's compressibility. The thesis studies the impact of parameters in statistical compression's design space such as data granularity, value frequency and encoding efficiency through time. A first finding is that *value locality* varies little over time despite the large amount of data stored in the main memory level. This means that once Huffman coding has been established, it works sufficiently for a long execution time (on the order of milliseconds) without any update, corroborating to a previous study on cache compression. We also evaluate different compressibility monitoring approaches as well as losses in compression efficiency when applying fixed-size compressed memory pages to address the impact of memory fragmentation problem. Overall, it is found that a high compression factor (CF) of 3.115X, on average, and up to 7.840X is possible.

Keywords: Memory, Lossless Data Compression, Statistical Compression Memory Scheme, Huffman Coding, Value Locality.

Acknowledgements

The 2 years’ master study in Chalmers is the best experience in my life. I am very thankful for all the help I got from lots of kind persons in Gothenburg.

First of all, I would like to thank to my supervisors, PhD candidate Angelos Arelakis and Professor Per Stenström. They guided me into this amazing area in which I have learned such a useful knowledge and skills. They support me a lot when I face problems both in study and life. I wish them all the best with health and success.

I would like to acknowledge Doctor Risat Pathan. You are the best teaching assistant I have ever met. I am deeply affected by your passion and sense of responsibility. You will be my role model in the future.

I am grateful to my classmate and friend, Li Kang, who introduced me to the fascinating research area of cache/memory compression and inspired me to start this thesis.

Last but definitely not the least; I would like to acknowledge my family and my girlfriend. You are always the most important to me.

Yu Jiang,

Gothenburg March, 2014

Contents

Acronyms

BDI	Base-Delta-Immediate
CF	Compression factor
$CF_{with\ VFT}$	CF with consideration of metadata of VFT
$CF_{w/o\ VFT}$	CF without consideration of metadata of VFT
CL	Codeword length
CW	Codeword
FPC	Frequent pattern compression
FVC	Frequent value compression
G1	granularity of 1 byte
G2	granularity of 2 bytes
G4	granularity of 4 bytes
LZ77	Lempel-Ziv 1977
STL	Standard Template Library
VFT	Value frequency table
VFT-all	VFT that saves all the values appear in the memory pages
VFT-X-mfv	VFT that saves the X most frequent values in the memory pages

1. Introduction

The performance of computational systems is dominated by core and memory architecture. Despite the enormous technology advancement through the technology scaling, the complexity as well as the workload requirements of the computational tasks are also growing in an enormous pace. As a result, memory with higher speed and larger capacity as well as more advanced processors are still the key solutions.

Memory wall, which is the speed gap between processor and memory, has been growing for almost three decades, as is illustrated in Figure 1.1 [1]. The latency of one memory access is now three orders of magnitude longer than executing an instruction in the processor. Thus memory has become the major bottleneck in computer performance.

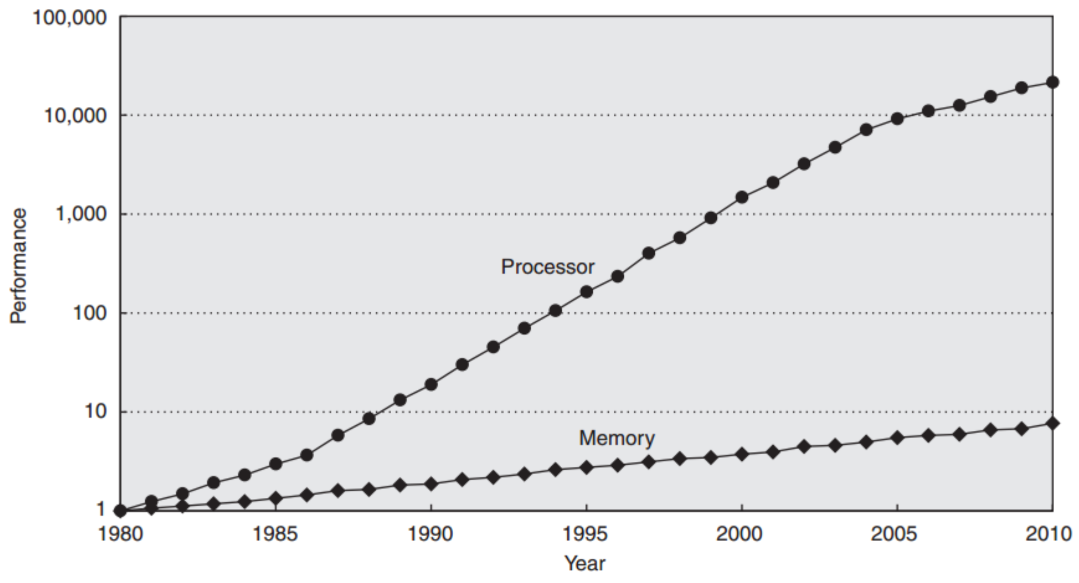


Figure 1.1: *Memory wall: The speed difference between processors and memory [2].*

This increasing speed gap makes the requirement of fast and large memory even more urgent. On the top of that, the limiting power budget constrains the demands for larger physical memories. Therefore, applying techniques that enhance memory's efficiency while keeping low area, power and economic costs are of high importance.

One such technique is data compression. Lossless data compression techniques have a huge potential to improve the utilization of main-memory resources [3]. As such techniques target to store more data in memory, the overall memory capacity is virtually increased. As a result, more accesses are handled by the memory (less page faults), instead of accessing the disk that may require even three orders of magnitude more time (order of μs). Another advantageous effect is the efficiency in power as power requirement can be kept in low levels.

The main drawback of compression is that compressed data must be decompressed before being used. One implication of this is that extra latencies may be imposed, potentially affecting overall system's performance as decompression must be applied to data that are requested by the processor (i.e., a critical access) [4]. In addition, compressors and decompressors will introduce power overheads. Therefore it is crucial to consider the trade-off between compressibility and performance/power cost when designing a compression mechanism in the context of caches/memories. Apart from some fast but inefficient algorithms, the statistical compression cache/memory schemes have a great potential compressibility at the expense of high latencies. Thus, they would be good compression candidates if modifications can be made on the compression/decompression process to address the performance losses problem.

This master thesis makes a motivational analysis of statistical compression algorithms, such as Huffman coding, in memory compression and studies the impact of several design parameters (mainly data- and algorithmic- related) on memory's content compressibility. Our results show that Huffman compression can highly improve the utilization of memory resources when applied on that level. The simulation results for 7 applications from the SPEC2006 benchmark suite show that a high compression ratio of 3.115X on average and up to 7.840X is possible. Besides, the thesis studies the impact of compression granularity (1 byte, 2 bytes and 4 bytes). Another contribution of this thesis is it evaluates the impact on compressibility when establishing Huffman coding with value frequency tables (VFTs) of different sizes. The smaller VFT introduces fewer values to be compressed trading off compressibility for reduced performance overheads due to smaller compression/decompression latency. The results are promising, as despite using a small VFT, a relatively high compression ratio is possible. This observation shows a great potential of practicality for Huffman coding technique on the main memory level.

We demonstrate that during the execution of benchmark applications, the value locality on the main memory varies little over time (on the order of hundreds of millions of committed instructions) which means once a Huffman tree has been established and the encoding has been generated, the same encoding can be used over millions of memory accesses without any update. It is meaningful to say that the impact of value granularity, VFT size as well as the duration of sampling (prior code generation) have been studied in prior works in value-aware caches [5, 6]. This thesis extends those studies in main memory, while many of the findings corroborate with or complement prior works. We also make experiments to evaluate the compressibility reduction when considering the memory fragmentation problem by applying fixed-size page compression.

The thesis first discusses the background and theory in Chapter 2. In Chapter 3, the experimental setup of this thesis is explained. Chapter 4 presents the experimental results based on different design parameters together with the result analysis. The conclusion of the thesis is made in Chapter 5.

2. Background and Theory

In this chapter, the background in memory hierarchies is first discussed to define the problem that this thesis studies (Section 2.1). Then, a review of the most important previous works about memory compression techniques is presented (Section 2.2). Finally, Section 2.3 discusses Huffman encoding and decoding, how it is generated and the advantages and disadvantages.

2.1 Principle of Locality and Memory Hierarchy

Memory space is needed during program execution to store data that are produced and data that are going to be used. Despite the technology advancement of processors and memory hierarchies, the program demands for more memory space are increasing too. Thus, larger and more efficient memory capacity is tightly bonded to high performance computing.

The great success of memories (both off-chip and on-chip) is due to the *principle of locality* that characterizes the data distribution in programs [7]. It can be divided into 1) *temporal locality*, which describes the likelihood of accessed data to be accessed again in the near future and 2) *spatial locality*, which states that once a piece of data is accessed, nearby data are likely to be referenced later.

The *principle of locality* led to the appearance of *memory hierarchy* which is organized into several levels. Figure 2.1 [2] shows the typical structure of memory hierarchy.

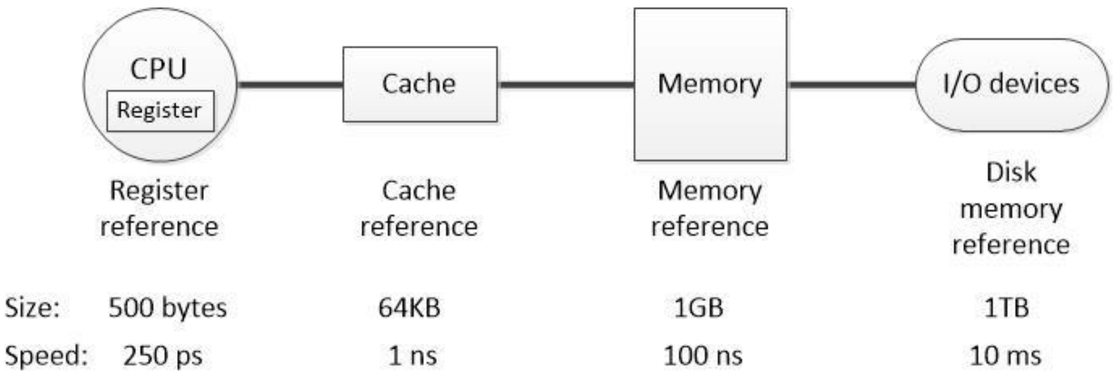


Figure 2.1: *The typical structure of memory hierarchy in computer system.*

As illustrated in Figure 2.1, there are 4 levels of memory in the hierarchy, from the register level to the hard disk level. The level closer to CPU saves data that were recently used or modified and data that may be probably used soon. For these reasons it is fast but has small capacity and is more expensive in comparison to lower level memories. The access time between adjacent levels typically differs by two orders of magnitude, while the difference can even be three or four between the main memory level and the hard disk.

When a piece of data is requested from the processor, it will be retrieved from the highest level (closest to the processor). If the data is not found, or in other words a *miss* occurs (in memories it is called *page fault*), the request is propagated further to the lower levels until the data is eventually located [8]. Thus, the large capacity and efficient utilization of the main memory can lead to fewer disk accesses thus saving millions of CPU cycles [9].

One way to increase the memory space without adding extra physical resources is by using compression techniques. Data compression shrinks the stored data and increases the effective memory capacity thus creating fewer misses (in cache space) or fewer page faults (in main memory) and possibly improving overall performance. Data compression can be used in all the levels of the memory hierarchy in Figure 2.1, but aiming in different gains.

Next section discusses the most important previous work in cache/memory compression. Our study applies the compression approach on the main memory level.

2.2 Previous Studies

Applying data compression to the cache and memory can increase their virtual capacity without adding extra area and increasing the power overheads. However, compression does not come for free as it may introduce extra latencies in the critical path possibly affecting performance as well as power overheads due to compression and decompression processes. Another obstacle of integrating compression in cache/memory is changes in the design and verification process that are required. Therefore simplicity is another important characteristic.

Previous studies are focusing on proposing in most of cases simple compression schemes targeting to improve the utilization of memory resources but trading off high compressibility for design simplicity and short decompression latency. On the other hand, this study explores the potential of using more complex statistical compression approaches, such as Huffman coding that can deliver superior compression ratios. The rest of this section discusses related works by classifying them based on the compression algorithms they employ.

Significance-based compression:

Alameldeen and Wood proposed a significance-based compression scheme called Frequent Pattern Compression (*FPC*) [10] in the cache level. FPC mainly targets to eliminate small values that are represented as big data due to fixed-size representations, by replacing them with fewer bits. That is because the few least-significant bits are enough to record all the information. This work defines 8 patterns that are common in cache data and can be represented with a 3-bit prefix. Once a word (32 bits) in the cache line matches one of the patterns, it will be encoded and stored with its prefix. For example, prefix 001 means the 32-bit word is a small signed integer that can be stored into 4 bits. Then only 7 bits (3 bits for prefix and 4 bits for data) are needed instead of 32 bits. The compression factor ($\frac{uncompressed\ cache}{compressed\ cache}$) achieved is less than 1.5X, on average, and ideally no more than 2X.

Ekman and Stenstrom exploit a variation of FPC (zero-based) [3] on the main memory level. They found that 55% of all bytes in the memory are zero for SPEC2K and few database workloads and target to eliminate zero bytes or null blocks by efficiently storing them. They also propose a small TLB-like structure that is used to quickly locate compressed blocks and pages.

Value-based compression:

Lots of previous studies on data compression can be categories as value-based compression schemes.

Yang et al. proposed a simple scheme called Frequent value compression (*FVC*) [11, 12] where they try to squeeze two cache blocks into one line based on the observation that during the execution of a program, a small number of values that is predefined using profiling occupies about 50% of references for the integer applications of SPEC95. The FVC scheme exploits this observation by compressing some most frequent 32-bits values into smaller size. For example, the 7 most frequent 32-bits values are encoded into 3 bits from 000 to 110. The other infrequent values are presented with the tag 111.

On the other hand, the Base-Delta-Immediate (*BDI*) compression scheme, which was proposed by Pekhimenko et al., exploits spatial value locality [13] to compress cache blocks [14] and recently memory space [15]. BDI compresses cache blocks that contains nearby values using delta encoding. For example, a cache block may contain the following data values 0xafaf8890, 0xafaf8898, 0xafaf8845, etc. FPC (or similar) could not compress these values, as they don’t follow an exact pattern. However, BDI uses one of them as a base and represents the rest using their difference (delta) from the selected base. BDI increase the cache capacity by 1.53X, on average, and ideally by 2X improving slightly vs. FPC.

Dictionary-based compression:

Dictionary-based compression approaches encode common values using pointers to dictionaries which keep those values. A simple dictionary-based cache compression scheme was proposed by Molina et al. [16] achieving an overall compression factor of less than 1.4X due to the overheads of the large dictionary.

A popular variation of dictionary-based compression algorithms is LZ77 [17]. Assuming there is a sequence of data to be compressed. A virtual window (dictionary) is defined with a constant length and added into the head of the sequence. That means the window is part of the sequence and its content changes due to its movement in the sequence. During the compression process, the window moves forward in the data sequence and keeps comparing the sequence data behind the window with its own content for the longest match. A match occurs when a data sequence behind the window is as same as part of its content. The matched data are encoded for compression purpose by using the corresponding pointer information of the window. For example, there is a sequence of character MFABCABCEDF. Define the length of window to be 4. While the window moves to the second character of the sequence, its content is FABC. The window searches for the match from the first character behind it which is A. In this case, the first 3 characters behind the window (ABC) is as same as part of its content which is the longest match. Then a combination of (1, 3, E) is used to represent ABC. Where 1 is the pointer of A in the window (as the pointer of F is 0), 3 presents the length of the matched data ABC, E is the first unmatched character behind the window. In the next step, the dictionary moves forwards to be ABCE and searches for a new match from character D.

IBM company realized a scheme called MXT which is a parallelized derivative of LZ77 [18]. Typically, each 1-KB uncompressed data blocks are divided into 256B parts which are operated by 4 independent compression engines. Each engine contains a 255B content-addressable memory (CAM) while all the engines use the same dictionary for compressing the data in different parts. In this way, the compression process is speed up by four times. The compression factor of MXT is close to 2X. This is the only commercial example of memory compression.

Statistical compression:

Huffman coding is a typical statistical compression algorithm. Huffman coding eliminates data redundancy using the statistical properties of data values, or in other words by encoding densely those data values that appear more frequently. After coding, more frequent values have narrower or at least equal length codewords (*CWs*) than less frequent values. A counter effect of Huffman coding is that, some infrequent values may have codewords longer than their original code. For example, it is possible for a 16-bits value to be encoded using 20 bits. As a result, the compressibility of Huffman coding depends on the value distribution. To make it more concrete, the compression result would be better if fewer values occupy higher fraction of the total frequency. In this thesis, we focus on applying Huffman coding on main memory level.

2.3 Huffman Coding

Generally, one value is stored with fixed width related to its data type. For example, one integer value (defined as *int* in C++) occupies 32 bits no matter the value is 0 or 0xffffffff. Huffman coding can densely compress data by assigning narrower codewords to more frequent values and wider codewords to less frequent values. As a result, the total code length after encoding is shorter than the original one.

Huffman codes are prefix codes which means it is impossible for one code to be the prefix of any other codes. Huffman coding is based on two observations of optimal prefix codes [19]:

1. In an optimal prefix code, more frequent symbols will have shorter (or at least the same length) codewords than symbols that occur less frequently.

2. In an optimal prefix code, the two least frequent symbols have the same length.

Huffman code is proved to be optimal prefix code for a given set of symbols.

2.3.1 Huffman Coding Algorithm

Huffman coding is generated by first building the Huffman tree, which is a binary tree, using a particular methodology. Once the tree is built, the codeword assignment of all symbols is done. An example is made by using the data in Table 2.1.

Table 2.1 *Information of the sequence to be compressed*

Symbol	Weight(frequency of occurrence)	Fixed-size code
E	2	100
C	3	010
B	8	001
A	10	000
D	12	011
F	14	101

Assume a sequence including 6 different symbols which are sorted by their frequencies of occurrence (weight) in Table 2.1. As 2 bits can only represent 4 possibilities, then at least 3 bits are needed to represent 6 symbols when using the codes of same length. The third column in Table 2.1 shows an example of fixed-size representation. The total code length of the sequence is 3*(2+3+8+10+12+14) = 147 bits.

Now consider using Huffman coding to represent the same sequence. Before building Huffman tree, all the symbols are sorted in ascending order by their weight to be a list which is already done in Table 2.1. In every step of establishing the Huffman tree, two symbols with the lowest weight are deleted from the list and added into the tree. A new node is generated in the tree as the combination of these two symbols and becomes the parent node of them. The combination node is written back to the list as a new symbol with the total weight of its two children so that the list remains sorted. Obviously, the number of symbols in the list is reduced by 1 after each step. The procedure continues until the list contains one symbol, the root of the tree.

Figure 2.2 illustrates the algorithm of building the Huffman tree by using the data in Table 2.1. The adjacent steps are connected with arrows in the figure. In the first step, the two least frequent symbols in Table 2.1, E and C are added to the tree and their combination is represented as a new node with their total weight of 5. The new symbol called EC is written back to the list. When step 2 begins, the two least frequent symbols are B and EC. Thus, EC and B are combined together. A new node called ECB is generated in the tree while its corresponding symbol ECB is added into the list. The construction of Huffman tree follows the left-right, bottom-up rule. Left-right means in each step, the symbol with lower weight is always set to a consistent side of the branch (for example, left) and assigned with a consistent binary value (in this case, 0) as one bit of its Huffman codeword. As a result, the other symbol with greater weight is put on the other side (right) with the rest binary value (1). Bottom-up rule means the Huffman codeword for each symbol is generated from the least significant bit to the most significant bit. In specific, as shown in Figure 2.2, after step 2 the least significant 2 bits of the Huffman codeword of symbol C are figured out as 01 (**bold fonts with underline**). Similarly, for symbol B, the least significant bit of its Huffman codeword is 1 (*italic fonts with underline*).

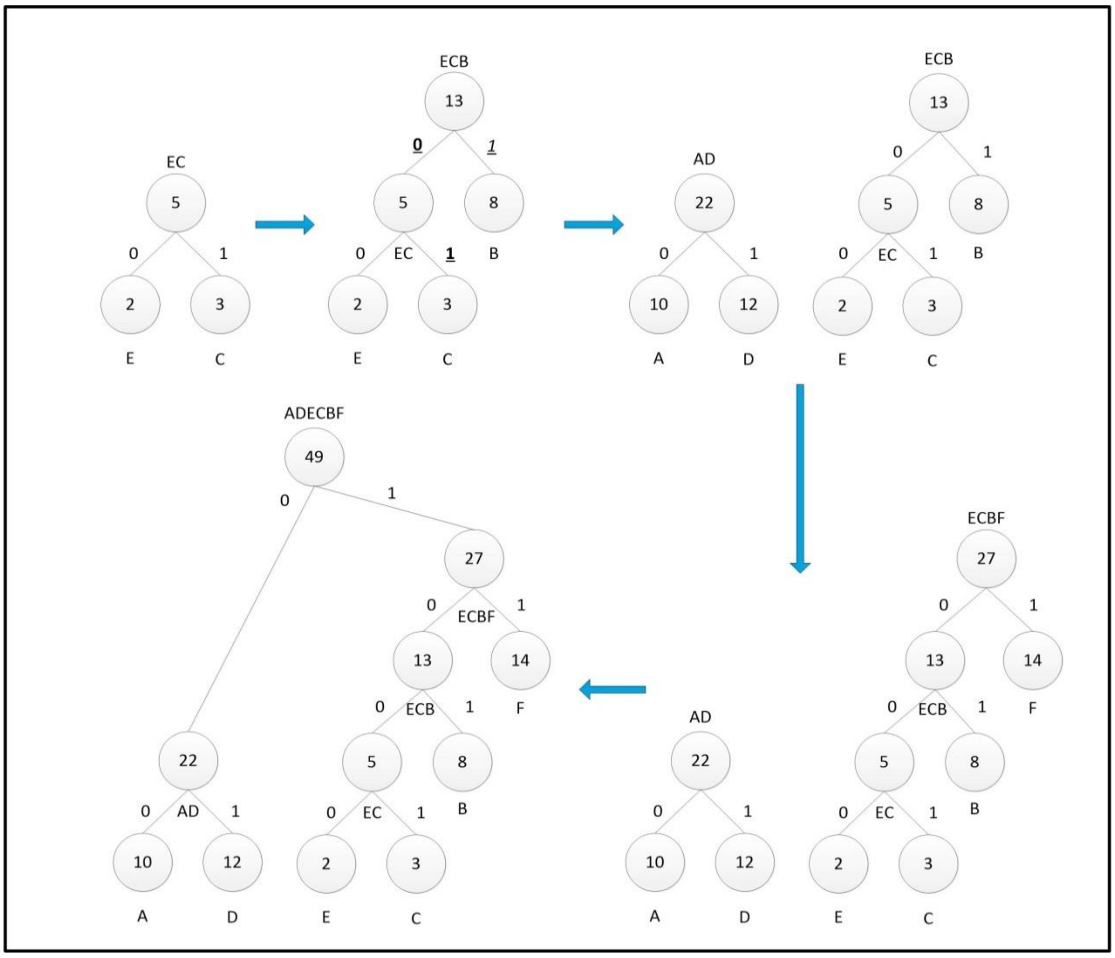


Figure 2.2: *Algorithm of building Huffman tree.*

After 5 steps, the tree is generated and the assignment of Huffman codeword is done as shown in Table 2.2.

Table 2.2 *Comparison between uncompressed/compressed symbols.*

Symbol	**Weight**	**fixed-size code**	**Total length of fixed-size codes (weight*code length)**	**Huffman codeword**	**Total length of Huffman codewords (weight*codeword length)**
E	2	100	6	1000	8
C	3	010	9	1001	12
B	8	001	24	101	24
A	10	000	30	00	20
D	12	011	36	01	24
F	14	101	42	11	28
Total	49		147		116

The total length of the sequence by using Huffman codewords is 116, the compression factor ($\frac{uncompressed\ size}{compressed\ size}$) is 147/116=1.267X which means the representation of the sequence by using fix-size codes is 1.267 times long as the one using Huffman codewords. To evaluate the performance of a compression scheme, one way is making comparison between the compression result and ideal compression limitation. With the help of principle *entropy*, the compression limitation of the sequence can be calculated with Equation 2.1:

$$\text{Limitation of sequence length} = -\sum \mathrm{W}_{\mathrm{A_i}} * \log_{\mathrm{b}} \frac{\mathrm{W}_{\mathrm{A_i}}}{\mathrm{W}_{\mathrm{total}}} \qquad (2.1)$$

W means weight (frequency of occurrence) of symbols while b is the unit of scale which can be set to 2 as we use binary code for representation. The limitation of compression size in this case is:

$$-\left(2\log_2\frac{2}{49} + 3\log_2\frac{3}{49} + 8\log_2\frac{8}{49} + 10\log_2\frac{10}{49} + 12\log_2\frac{12}{49} + 14\log_2\frac{14}{49}\right) = 114.8241$$

That means at least 114.8241 bits are required to represent the sequence. As the length of the sequence by using Huffman codewords is 116 which is close to the compression limitation 114.8241, we demonstrate that Huffman-based compression approach has a high potential compressibility.

2.3.2 Drawbacks of Huffman Coding and Considerations

Although Huffman coding has a great potential compressibility, several drawbacks of it may diminish its practicality in the context of memory compression. The thesis studies the impact of the main disadvantages of Huffman coding and explores the related questions as followed:

1) Time cost for establishing value statistics: as Huffman codewords are based on the data distribution (are data-dependent), extra time is required prior compression to establish the data value statistics. Moreover, the value statistics should reflect the value distribution of the application accurately for building an efficient Huffman tree. The question is how many values are needed to be collected for building an effective Huffman tree. In Section 4.4, the compression experiments of building Huffman tree based on 7 million write memory accesses are implemented to explore this question.
2) Overhead for storing VFT: the information of data distribution or the Huffman tree is necessary for decompression thus should be kept in the memory. Storing this information may introduce a significant area/power overhead. To address this problem, the thesis evaluates Huffman coding with the value frequency table (*VFT*) of different sizes. VFT stores the data distribution information of memory in which the values in memory are stored with their frequency of occurrence. Despite VFT-all which includes all the values in memory, the so called VFT-X-mfv only contains the X most frequent values. The values outside the table are infrequent values which are not compressed by Huffman coding scheme. By using VFT-X-mfv, fewer values are kept in the table thus introduce a smaller overhead and the memory capacity can be saved. Thus the second drawback of Huffman coding is alleviated. Another benefit of using VFT-X-mfv is that it may simplify the compression process. Assume a value to be encoded. The encoding program should search the value in the value-codeword table to find its corresponding Huffman codeword. Obviously it is easier to search a value in a small table than in a big one. Since VFT-X-mfv can introduce small value-codeword table than VFT-all, it may reduce the workload of table-scanning thus speeds up the compression process. The drawback of using a VFT-X-mfv is that the compression performance may degrade as less data values are encoded. In Section 4.3, we try different VFT sizes to find the best option for compression.
3) Latency of decompression: as the decompression process is on the critical access path, the latency comes from the decompression must be kept low. In a regular Huffman scheme, the decoding time varies based on the codeword length. For

example, a 10-bits codeword requires double decoding time than 5-bits codeword. However, the work on this thesis assumes a decompressor that is based on canonical Huffman coding [6]. This way, the decoding process is accelerated as it's broken in two pipelined stages: one detects a valid codeword (of any length) and the second retrieves the value using the previously detected codeword as index. The work of accelerating the decompression process is not discussed in this thesis.

4) Inefficiency problem: Huffman coding may be inefficient according to improper design parameters, such as compression granularity which is the width of the value under compression. For Huffman coding, it is possible for an infrequent symbol to have a wider Huffman codeword than its original representation. The improper granularity selection may lead most values assigned with longer codewords which means time and power are wasted for meaningless or even pernicious coding. This problem is studied by looking at 3 different compression granularities (1 byte, 2 bytes and 4 bytes) in Section 4.2. Our results show that as the granularity size increases, less and less values are assigned with codewords longer than their original representations thus the potential compressibility is improved.
5) Impact of value variation: as the application runs, the value distribution keeps changing. The compressibility of a static Huffman coding may degrade to an unacceptable low level during the execution of program, thus dynamic or semi-adaptive Huffman coding may be more suitable approaches. We study this issue by exploring the impact of value locality variation on compressibility which is introduced Section 4.4.

2.4 Summary

In this chapter, the typical memory hierarchy of 4 levels is first introduced. The lossless compression schemes can be implemented on all the levels of memory for different purposes while this thesis focuses on compressing data in the main memory. In Section 2.2, we discuss several important compression algorithms on cache/main memory levels with their performance comparisons. Unlike most prior proposed compression schemes which are fast but inefficient, this thesis studies the statistical compression algorithms such as Huffman coding which have a high potential compressibility at the expense of high latencies. In Section 2.3, the theory and an example of Huffman coding are presented. The main drawbacks of Huffman coding algorithm together with several questions considered in this thesis are also discussed in this section. Based on these questions, we implement different experiments to explore the characteristics of our Huffman coding compression algorithm on main memory level which are discussed in the later chapters.

3. Experimental Setup

This Chapter discusses the experimental setup of this thesis. The requirements of the experiment are first introduced. Later, we detail the methodology followed in order to setup our experimental environment combining different tools and writing our own tools. Moreover, it is explained how this setup is adapted when different parameters which this thesis analyzes, are included.

Typically, the setup required to generate Huffman coding mainly consists of 4 steps, as shown in Figure 3.1. Firstly the selected benchmark application is executed while the information of memory accesses during the execution are collected (Step1). Next, the *memory footprint* is generated in Step2 according to the collected data from Step1. In Step3, VFT of different sizes and granularities are established which store the value distribution information of memory footprint. In the last step, Huffman codewords are generated based on VFT and the compression experiments are implemented for evaluating the potential compressibility of the scheme.

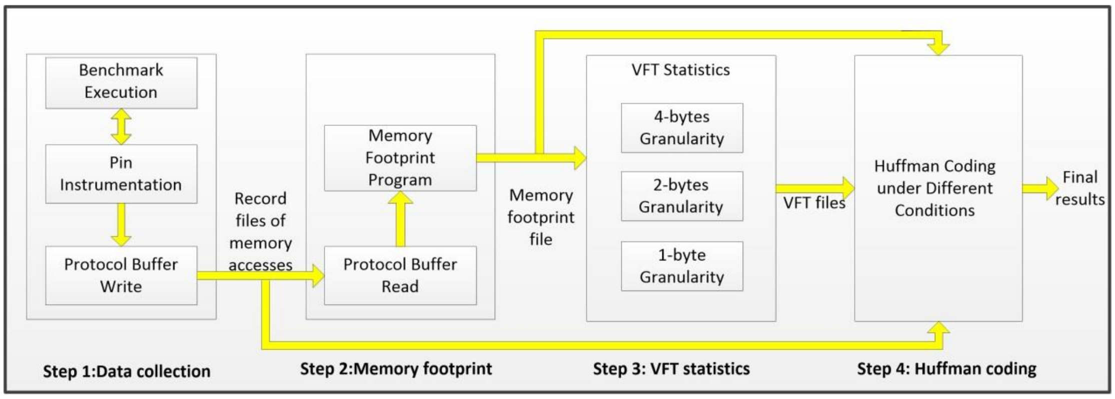


Figure 3.1: *The typical Huffman coding experiment process.*

Each step of the experiment is discussed in a unique section. Step1 is introduced in Section 3.1. To make a reliable evaluation of Huffman coding scheme, a suitable set of applications are needed as the compression targets. In Step1, 7 benchmark applications are selected and executed and the memory access operations during the executions are traced and recorded in time order. The memory accesses information obtained in this step is needed for establishing the data value statistics which is necessary for building Huffman tree.

Section 3.2 describes Step2 of the experiment. The data in memory keeps changing during the execution of application, while the dynamic value distribution is needed for building Huffman tree. *Memory footprint* is referred to the amount of memory (on the page granularity) that has been accessed by the application for a specified execution period and is used to establish the value distribution statistics required. Hence in Step2, memory footprint is established according to the memory trace files from Step1. To make it more concrete, the addresses and data of the accessed blocks are used to build memory pages or modify them by reading the memory trace files from the beginning of an execution phase. Reading from the trace can stop at any possible time instance to perform analytical studies.

While the memory footprint is being established, we can start extracting the value distribution information using the format of the *Value Frequency Table (VFT)*. This is the goal of Step3. The VFT keeps all the values and their frequencies of occurrence as they appear in the memory footprint. Step1 records the memory footprint, i.e., the addresses requested by the application and the corresponding data content of the virtual page (the access address is aligned to) in sequence of bytes. The study investigates several value granularities. Thus in Step3, we read the footprint and establish VFTs with 3 different value granularities which are 1 byte, 2 bytes and 4 bytes. This step is presented in Section 3.3.

VFT is the premise of building Huffman tree. As VFT is generated in Step3, the Huffman tree can be established then. In Step4, we first establish the Huffman tree with different parameters, such as the granularity, VFT size and different execution time points of the applications. Based on the generated Huffman codewords, the potential compressibility of the compression scheme is evaluated. Section 3.4 makes a brief introduction of this step while more details together with the experimental results and analysis are introduced in Chapter 4.

The flow chart that depicted in Figure 3.1 is the main experimental flow that is followed in the different experiments of this thesis. However, extra steps may be included for some experiments depending on particular requirements.

3.1 Step1: Data Collection

The main work of Step1 is executing an application and tracing its memory accesses during the execution. More specifically, this step can be divided into 3 parts as shown in Figure 3.1. First, a suitable application is selected and executed (Section 3.1.1). Then, the memory accesses information of the execution is traced (Section 3.1.2) and recorded (Section 3.1.3).

3.1.1 Benchmark Execution

To make a reliable evaluation of Huffman coding scheme, a suitable set of applications are needed as the compression targets. Six integer and one floating point applications from the SPEC2006 benchmark suite are selected in this thesis. Different benchmarks are for different kinds of calculating tasks. For example, the benchmark *mcf* is used for simulating the vehicle scheduling by using a network simplex algorithm. As a result, the value distribution varies hugely between different benchmarks which make our compression results to be more universal.

SPEC2006 benchmark workloads have execution times on the order of hundreds of billions of instructions, while some of them have memory footprints on the order of GBs. Simulating or tracing for their whole execution requires significant amounts of time and space resources. The SimPoint methodology is adopted in this thesis to reduce the size of workloads by tracing in particular representative execution phases. For each application, two simulation points (simpoints) are used; each one has an interval of 10^9 committed instructions. For *leslie3d*, 12 simpoints of interval size of 250-M committed instructions are used instead due to lack of 1-B simpoints for this application. The simpoints are assigned with different *weight* according to their contribution to particular performance metrics, mainly CPI (cycles per instruction). By using the SimPoint methodology, we focus on the result of simpoints instead of observing the whole benchmark application. The interval decided by one simpoint is called one *phase*. The Huffman-based memory compression experiments are implemented in each phase-unit. The detail information of each phase for seven applications is shown in Table 3.1

Table 3.1 *Information about each phase in 7 applications.*

Benchmark-phase	Start instructionth	End instructionth	Weight
xalan-p1	187000000579	188000000584	0.339176
xalan-p2	969000003110	970000003111	0.660824
omnetpp-p1	23999999635	24999999636	0.0179533
omnetpp-p2	342000000749	343000000750	0.982047
astar-p1	133000000644	134000000648	0.619289
astar-p2	343000002234	344000002237	0.380711
mcf-p1	25000000144	26000000145	0.285266

mcf-p2	161000000751	162000000752	0.714734
bzip2-p1	92000000550	93000000570	0.924051
bzip2-p2	210000001282	211000001288	0.0759494
hmmer-p1	31000000510	32000000512	0.786268
hmmer-p2	379000006210	380000006250	0.213732
leslie3d-p1	181250010845	181500010874	0.0666848
leslie3d-p2	219750013126	220000013155	0.123551
leslie3d-p3	325250019699	325500019701	0.0627301
leslie3d-p4	329750019982	330000019992	0.0807309
leslie3d-p5	564000034050	564250034064	0.11046
leslie3d-p6	729250044718	729500044723	0.0754125
leslie3d-p7	873500053660	873750053707	0.109914
leslie3d-p8	916750056165	917000056173	0.084822
leslie3d-p9	1007500061678	1007750061683	0.0842766
leslie3d-p10	1033500063389	1033750063401	0.0724124
leslie3d-p11	1096750067162	1097000067164	0.062321
leslie3d-p12	1156750070773	1157000070775	0.0666848

3.1.2 Pin Instrumentation

In the experiment, we need to accurately trace and record all the memory accesses during the phase execution. That is because of the methodology we used in the thesis for evaluating our Huffman-based compression scheme. To be more specific, there are 2 ways to evaluate the performance of the compression scheme on different benchmark applications. The first and straightforward way is implementing Huffman-based compression/decompression for all the memory accesses during the execution and observing how much memory space could be saved. There are 2 disadvantages of this approach. First and most important, implementing encoding/decoding for the whole execution phase introduces a vast number of calculations which is time/power consuming. The second drawback is that the differences between two program runs may introduce different value distribution making it difficult to verify the correctness and guarantee the reliability of the experiment. For example, the same compression scheme may provide different compression results, possibly influencing the performance evaluation.

The second way that is applied in this thesis is a simulation approach in which the Huffman coding is generated using a record of the application’s traced data. In particular, the benchmark application executes only once. During the execution, every memory access is traced and recorded. After the execution, we review the record files and the process of value variation in memory during the execution can be simulated. Thus, we can simulate that execution process and evaluate our Huffman-based memory compression scheme on it. To evaluate the compression performance, it is no need to implement encoding/decoding anymore which is time-consuming but necessary in the first methodology we mentioned. In this way, the length of each Huffman codeword (CL) is used to calculate the compressed size of its corresponding value. Thus the compressed size of each memory page can be worked out by adding all the CLs of the values in the page together. As a result, the evaluation task is finished without encoding/decoding which is helpful for time saving. Besides, the compression would not be influenced by the different program runs anymore since the record provides a static value statistics. However, the way applied in the thesis makes it extremely important to trace and record the memory accesses accurately. If the recorded information is different from the real situation, all the experimental results could be meaningless. For that reason, a professional tool is needed for tracing the application execution.

Pin tool—a dynamic binary instrumentation tool developed by Intel Company [20] is applied for the instrumentation of the benchmark applications. Pin enables the analysis at the instruction level. Thus, every memory accesses can be traced by Pin without any influence on the execution.

Figure 3.2 shows how the Pin program traces each memory accesses in a *phase* (phase is introduced in Section 3.1.1). During the execution of the application, once there is a new instruction, the instruction counter in Pin is added by 1 and the simpoint situation is checked. If the instruction is included in the range of any simpoints (phases), the function for tracing and recording will be activated, otherwise it is kept in sleep mode.

In the next step, if the instruction is in any phase and it results in any memory accesses, the memory accesses must be traced and recorded. One memory access can be classified as 4 types:

1. *New read*: this is a read access, and the address is in a new physical memory page which has never been accessed before.

2. *Old read*: this is a read access, but the address reads from an old page.

3. *New write*: this is a write access, and the address is in a new memory page.

4. *Old write*: this is a write access, but the address writes in an old page.

The program records the access differently based on its access type. As the working environment of this thesis is Linux OS which organizes memory resource in pages of 4KB, the information of all the pages touched by the program need to be recorded for establishing *memory footprint* in the next step of the experiment (Step2). Thus, if the access is *new* (read/write), the program should record the information of the new accessed page of 4KB. The read access does not change the content of memory. Hence, to save space the program records nothing about an *old read* access. When it comes to a write access, normally several bytes in the memory are rewrote which need to be recorded by the program.

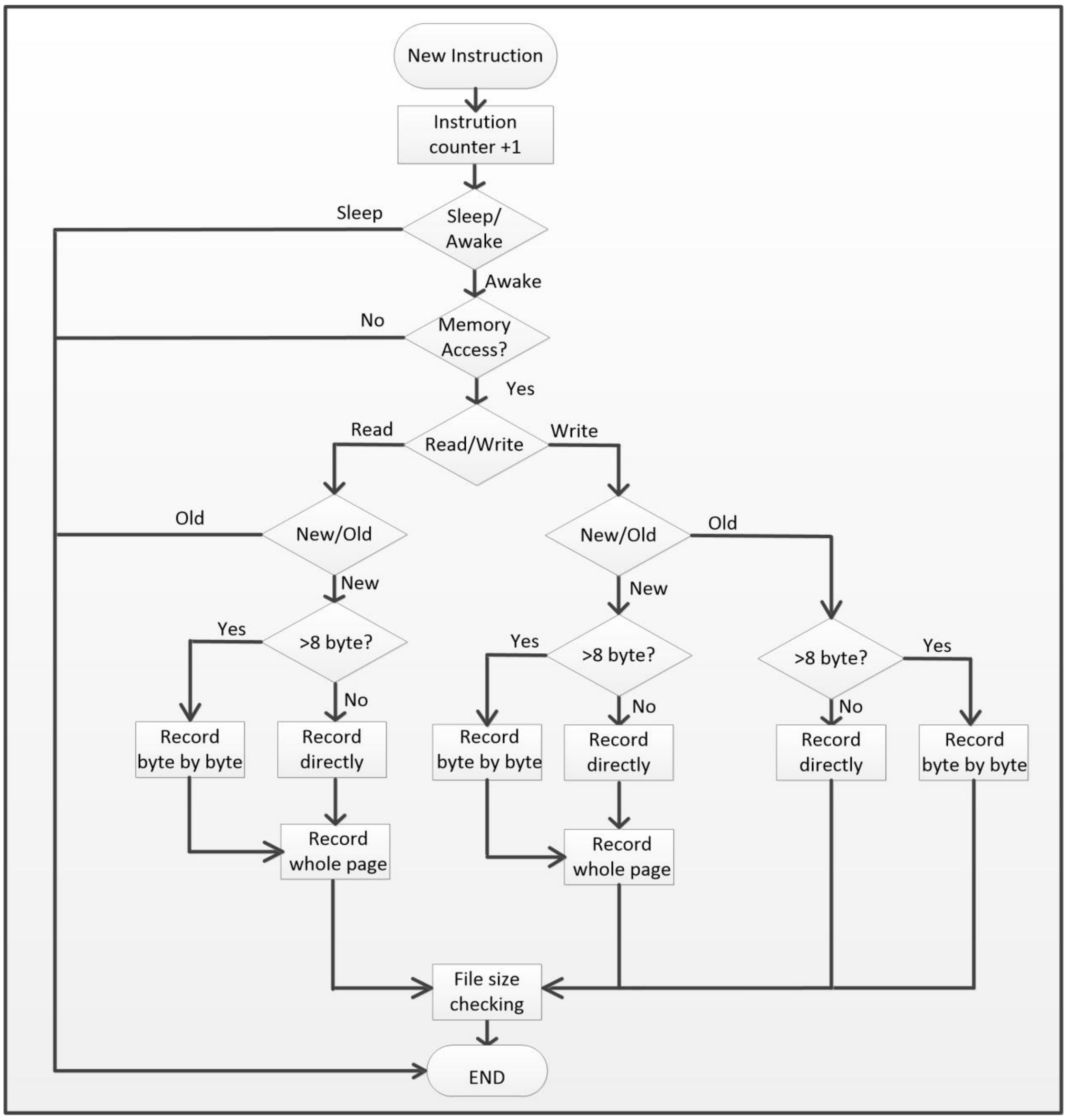


Figure 3.2: *The process of Pin instrumentation.*

As shown in Figure 3.2, the accesses of large size (more than 8 bytes) are handled specially. The biggest data type in C++ is '*long long int*' which is 8 bytes. Thus for the accesses larger than 8 bytes, it is necessary for the program to use a loop to instrument the accesses byte by byte. It must be mentioned here that as data are manipulated by X86 ISAs using little-endianness, another loop is needed to inverse the sequence of bytes before storing it into the record file.

The size of record files should be kept smaller than 67MB which is the size limit for reading by the software tool *protocol buffer* in the next part of the experiment (Section 3.1.2). Thus, a counter is set to estimate the size of the file. Once the counter reaches its upper bound, the current file will be closed and a new empty file will be generated. Thus, thousands of small record files are generated instead of a huge file.

By following the process as shown in Figure 3.2, all the memory accesses in the phases can be traced and recorded accurately. However, as the record files have a huge total size, some modifications should be done for saving the space. Despite the overlook of *old read* accesses, a modification for recording the new accessed pages is applied in the program. That is, when a new page is accessed, we only record the address of the page head followed with 4KB value information instead of recording addresses of every single byte in the page.

To further reduce the size of record files, a software tool called *protocol buffer* is applied in this thesis which is introduced in section 3.1.3.

3.1.3 Protocol Buffer

As mentioned in section 3.1.1, each phase includes 1 Billion (10^9) instructions. Although not all the instructions introduce memory accesses, there are still millions of memory accesses need to be recorded increasing the size of record files. To save the space, the Google *protocol buffer* [21] is used to store the data in an efficient way. The protocol buffer applies a scheme which stores smaller values with smaller size to meet the compression purpose. Another benefit of using protocol buffer is that the data after converting into protocol buffer type are well structured and easily to be managed.

To convert the regular record files into protocol buffer type, one message file is necessary to define the parameters in the record file as shown in Figure3.3.

```
package memorytrace;

message Record {

     message Paras  {

                     optional uint64 address = 1;

                     optional int32 accesstype = 2;

                     optional int32 size = 3;

                     optional uint64 value1 = 4;

                     optional string value2 = 5;

                    }

     repeated  Paras paras = 1;
          }
```

Figure 3.3: *Parameters used in record file.*

In the memory accesses record files, 4 parameters which are access address, access type, access size and access value are used to record the information of a memory access. To convert them into protocol buffer type, their data type should be defined in the message file. It is meaningful to say that there are two types of values defined in the message file as shown in Figure 3.3 which are *value1* and *value2* with different data types of *unit64* and *string*, respectively. That is because the large-size access problem introduced in Section 3.1.2 also occurs when processing the recorded data using protocol buffers. To address this problem, *value2* is defined for converting the access values larger than 8 bytes since *string* type has an unlimited upper bound of size. For the access smaller than 8 bytes, *value1* is used for converting the access value as its data type is *unit64* (unsigned 64 bits integer).

In Step1 of the experiments, the application, Pin tool and protocol buffer execute in parallel. Thus, at the end of the step, we obtain thousands of record files of memory accesses in protocol buffer type.

3.2 Step2: Memory Footprint

After getting the record files of memory accesses, the next work is constructing the *memory footprint* defined as Step2 in Figure 3.1. The *memory footprint* is basically which addresses that processor requests have been touched so far by the program. In the thesis, we defined *memory footprint* as the memory pages touched and the corresponding data at the end of the *phase* since data are organized in pages in the memory.

The memory footprint is constructed by reading all the memory accesses in the access time order. In particular, we first generate a data structure like a big matrix to represent the memory which is null at the beginning. Next, we review the history of data changes in memory during the execution by reading the record files from Step1. When there is an access of new type (new read/new write), we add the information of the whole page into the matrix. As the content of memory is altered by every write access, we need to modify the certain part of the matrix to emulate the changes in memory by the write accesses. To build the footprint, we should read all the memory accesses in the phase for updating the matrix.

To manage the value updates efficiently, the program was developed using the software library called Standard Template Library (STL) in C++ programming. The dedicated data structures in STL called containers as *vector* and *map* are efficient in sorting and searching. Here, the program uses the matrix-like structure as below to represent the content in memory, where *m1* is the name of a set of map containers:

```
map < unsigned long int, vector<short unsigned int> > m1;
```

The *map* implements a hash. One *map* element has two parameters which are *key* and *value*. To make it clear, we rename the *value* of the *map* element to be *weight*. Different *map* elements must have different *keys*. Thus, the *key* can be used to represent the address of one specific page in the memory and the *weight* is used to store the 4KB values in the page. Thus in *m1*, the *weight* is a *vector* which includes 4096 short unsigned integer values to store each byte in a page. In practice, once a new access occurs, its address information will be read first by the program. The most significant bits of the address as its page address will be compared with all the existing *keys*. If there is no *key* the same as this page address, it means the access occurs on a new page. Thus, a new *map* element will be generated with the *key* value of this page address. The *weight* of the new *map* element is set to be 4096 zero-bytes first, and then is updated by this access. On the other hand, if the access occurs on an old page, its page address will be found in the group of *map* elements. In this case, several bytes in the *weight* of that certain element will be rewrote according to this access.

Google protocol buffer is still needed in Step2 for reading the record files in protocol buffer type. However, as the memory footprints are usually small, they can be stored regularly without the compression by protocol buffer. At the end of Step2, the footprint for each phase is obtained.

3.3 Step3: VFT Statistics

As shown in Figure 3.1, we obtain the footprint at the end of Step2. In Step3, the value frequency table (VFT) as the premise of building Huffman tree can be generated base on the footprint. VFT records all the values appear in the memory footprint and sorts them in ascending order of their frequencies of occurrence. In other words, VFT shows the value distribution information of the memory footprint.

In the footprint, the content of each memory page is recorded byte by byte. For Huffman coding, the granularity which is the width of the value under compression could vary. Thus in Step3, VFTs of three different granularities, 1 byte, 2 bytes and 4 bytes, are generated from the same footprint. Despite the statistics of 1 byte value, the program should combine the certain number of nearby bytes for making the statistics of 2 bytes or 4 bytes values. For example, the first to fourth byte in the page are selected and combined together as a value of 4 bytes. Then the program turns to the next 4 bytes and so on. Endianess is taken into consideration when forming values of granularities larger than 1 byte.

Similarly to Section 3.2, the *map* container of STL is used again to manage the values in memory footprint and their frequencies of occurrence. In this case, the *key* is used to present one specific value in the memory and the *weight* is used to present the frequency of the *key*. Each newly generated value is compared with the previously obtained *keys*. If it is different from those existed values, the value will be added as a new *map* element with the *weight* of 1. Otherwise, when there is an element with the same *key* as the value, its *weight* will be added by 1.

As the program scans through the memory footprint, all the values appear in the memory footprint with their frequencies are worked out. After that, the program applies the *bubble sort* scheme to sort the values in ascending order of their frequencies of occurrence. It must be mentioned here that when the VFT is made based on the whole footprint at the end of the simulation phase, the value distribution statistics correspond to the memory content of that time instance. As the content in memory keeps changing during the execution, the data distribution may differ if we obverse the memory at another time point of the phase. For example, some values may be saved in memory for few cycles and then disappear during the application execution. Thus, they will not exist in the final footprint.

3.4 Brief Introduction of Step4: Huffman Coding

In Step4, the Huffman tree is established using the algorithm introduced in Section 2.3.1 based on VFT. All the values in VFT are assigned with Huffman codewords following the rule that the values with higher frequencies are assigned with narrower codewords. The information of the values with their corresponding codewords is stored in a table. Next, the table is used for the performance evaluation experiments of Huffman-based memory compression scheme. In this thesis, the compression factor (CF,$\frac{uncompressed\ size}{compressed\ size}$) is used to evaluate the scheme performance. To calculate CF, it is no need to implement encoding/decoding on the compression target such as memory footprint. Instead, we calculate the compressed size with the information of codewords length then estimate the final compression factor, decreasing the time consumption. More details will be introduced in Chapter 4.

3.5 Conclusion

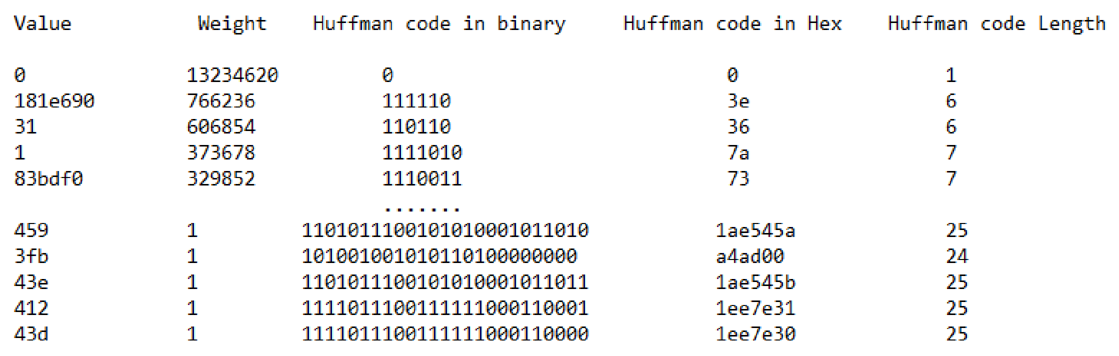
This chapter introduces the experimental setup in the thesis. Four main steps of the experiment are presented separately in details. The experiment need to be implemented in the order of steps because later steps always based on the result of previous steps. Several software tools and libraries are applied in this thesis. Some of them are essential in making the experiments more efficient.

4. Experimental Results and Analysis

This Chapter presents the results of our study after running different Huffman coding experiments together with the detailed analysis on the obtained results. It begins with the investigation of the potential compressibility based on the memory footprint using VFT of unlimited size (VFT-all), in Section 4.1. In Section 4.2, the impact of compression granularity is evaluated. In Section 4.3, as VFT-all is expected to occupy too much memory space, we study the impact of a reduced VFT size (VFT-X-mfv) on potential compressibility. To address the problem, the impact of different VFT sizes on the established value statistics and consequently on the generated Huffman coding is studied. Until Section 4.3, all the Huffman coding is based on the memory footprint at the end of each execution phase. To make our scheme more practical, in Section 4.4 we build the Huffman tree at the beginning of each simulation phase of each application. The Huffman codewords are generated after counting 7M write accesses in the obtained trace and are used for compressing the data in the rest part of application. The interval of 7M write accesses corresponds to approximately 50M committed instructions, as one store instructions occurs every 5-6 instructions. In Section 4.5, the thesis discusses the fixed-size compression to address the fragmentation problem in compression. It must be mentioned here that the impact of value granularity, VFT size as well as the duration of sampling (prior code generation) have been studied in prior works in value-aware caches [5, 6]. This thesis extends those studies in main memory, while many of the findings corroborate with or complement prior works.

4.1 Potential Compressibility of Huffman Coding

As shown in Figure 3.1, the experiment process is divided into 4 steps. After the first 3 steps, VFT is generated. In Step4, the Huffman tree is established based on VFT. The values with higher weights (frequencies of occurrence) in VFT are assigned with narrower Huffman codewords as shown in Figure 4.1.



```
Value          Weight    Huffman code in binary    Huffman code in Hex    Huffman code Length

0              13234620        0                           0                    1
181e690        766236          111110                      3e                   6
31             606854          110110                      36                   6
1              373678          1111010                     7a                   7
83bdf0         329852          1110011                     73                   7
                               .......
459            1          1101011100101010001011010       1ae545a               25
3fb            1          101001001010110100000000        a4ad00                24
43e            1          1101011100101010001011011       1ae545b               25
412            1          1111011100111111000110001       1ee7e31               25
43d            1          1111011100111111000110000       1ee7e30               25
```

Figure 4.1: *Huffman codewords assignment for xalan phase1, granularity of 4 bytes.*

Figure 4.1 presents the Huffman codewords of 5 most frequent and 5 least frequent values in VFT of application *xalan* phase 1. In this case, Huffman coding is based on the granularity of 4 bytes. As shown in Figure 4.1, the value 0x0(means 0 in hexadecimal) has the highest frequency of 13234620 occupying 42.33% space of footprint, thus assigned with the shortest Huffman codeword 0 which is only 1 bit long. As the original representation of a 4 bytes value is 32 bits long, the compression ratio of value 0x0 is 32X. From Figure 4.1 we can see that the value 0x43d only appears 1 time in the footprint and assigned with the longest Huffman codeword of 25 bits which is still narrower than 32 bits. In other words, all the values can be compressed into a smaller size by the scheme. This happens because the example value granularity is relatively coarse. Although 4 bytes (32 bits) can represent 2^{32} values, only a smaller number of values appear in the memory according to the experiment's findings. Later experiments show that for granularities of 1 byte and 2 bytes, the Huffman codewords of some values are longer than their original representations.

The selected granularity is also important as it may affect the value distribution thus encoding. For example, if the granularity selected is 1 byte, the frequencies of most values follow uniform distribution resulting in fixed coding that has similar width to the original representation's one. The maximum length of Huffman codewords is also highly related to the value distribution. For example, if the value probabilities follow the Fibonacci sequence, the depth of a Huffman tree with N values is N-1 [22], resulting in much larger codewords than the actual data width. On the other hand, Huffman coding is optimal when value probabilities are negative powers of 2.

From this discussion, we could expect that the granularity of 4 bytes is a more suitable option for statistical compression than finer granularities. On the other hand, coarser granularities allow more value representations possibly increasing the (area/processing) requirements when establishing the value statistics. This is an interesting trade-off that was studied in value-aware caches [6] and also is studied by this thesis in main memories.

As the Huffman codewords are obtained, in the next step we evaluate the compression performance based on these codewords. In this section, we make the experiments by using the memory footprint at the end of each phase as the compression target. The value granularity selected is 4 bytes. Based on the length of each Huffman codeword, we can calculate the compressed size of each 4KB memory page in the footprint. This way, the generated coding is optimal as the used value frequency is measured using the final data content of the memory footprint. Hence, we can establish the upper bound of the compressed memory footprint for the examined applications and investigate the full potential for compression. Later in the thesis, we explore the compressibility of the memory footprint but using more practical ways of sampling and establishing the value frequencies.

We use the *Compression Factor (CF,* $\frac{uncompressed\ size}{compressed\ s}$*)* metric to present the compressibility of Huffman coding. As one memory page is 4096 bytes large, the CF of page can be defined as $\frac{4096}{size\ of\ compressed\ page}$. The total CF for the whole memory can be calculated as $CF_{w/o\ VFT} = \frac{4096*n}{\sum size\ of\ compressed\ page}$, where n is the number of pages in the memory footprint. However, we should consider the cost of metadata for storing the VFT. For each value stored in the VFT, assuming 4-byte granularity, 8 bytes are needed to store the information: 4 bytes for storing the value and 4 bytes for the frequency. This way, CF is calculated using the following formula:

$$CF_{with\ VFT} = \frac{4096*n}{8*n_v+\sum\ size\ of\ compressed\ page} \qquad (4.1)$$

Where n_v is the number of values in the VFT. We evaluate the Huffman-based memory compression scheme on 7 benchmark applications. As mentioned in Section 3.1.1, for *leslie3d*, 12 simpoints of interval size of 250-M committed instructions are used instead due to lack of 1-B simpoints. The results are shown in Figure 4.2. As the 12 phases of *leslie3d* have similar compression results (measured in CF), we only present phase 1 and phase 2 of *leslie3d* in the figure. Besides, the workloads of calculating *mcf*-phase2 are too heavy due to the huge VFT in this phase. Thus, the result for *mcf*-phase2 is missing. More details about the compression are provided in Table 4.1.

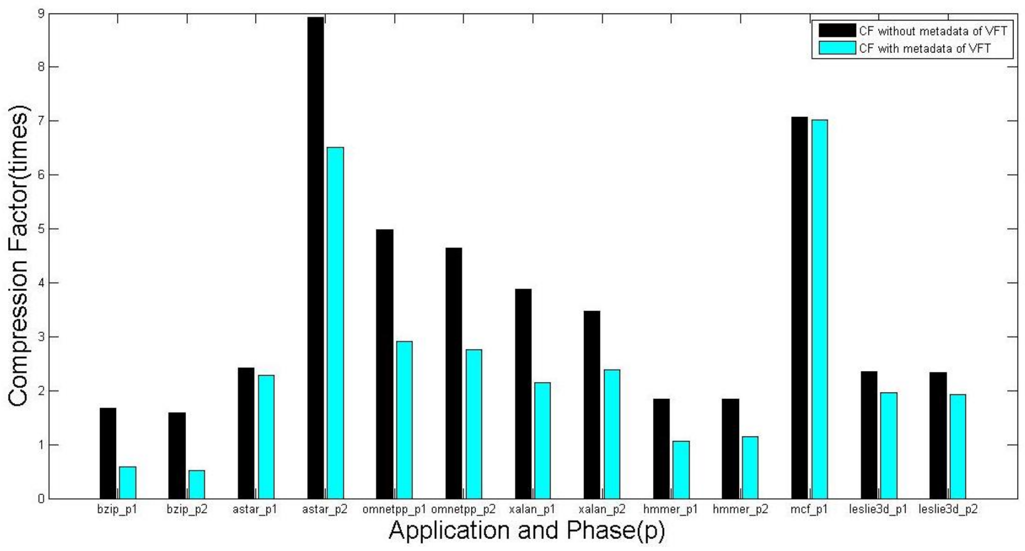


Figure 4.2: *Compression factors with and without metadata of VFT for granularity of 4 bytes.*

As shown in Figure 4.2, the best compression result without considering metadata is almost 9X in the second phase of *astar*. Besides, the footprint of all the applications can be compressed into smaller size than their original representations. However, there is significant compressibility degradation when the overhead due to metadata is considered by using equation 4.1. For *astar* phase 2, the compression factor is reduced by 28% (CF decrease of 2.5X units), while the highest $CF_{\text{with VFT}}$ of all the phases is obtained in *mcf* phase 1 instead. Moreover, the $CF_{\text{with VFT}}$ of both phases in *bzip2* is lower than 1X which means the compressed size of the footprint is even larger than uncompressed size. Hence, the compression for *bzip2* is harmful for saving memory space. In conclusion, the impact of metadata on the compressibility is significant.

Table 4.1 *Information about VFT and compression – Granularity: 4 bytes.*

Benchmark-phase	**Top10 Occurrence**	**Number of unique values**	**VFT size (MB)**	**$CF_{\text{w/o VFT}}$**	**$CF_{\text{with VFT}}$**
xalan-p1	52%	3235506	24.6850	3.8725	2.1495
xalan-p2	29%	2885387	22.0138	3.4731	2.3916
omnetpp-p1	67%	1590773	12.1366	4.9920	2.9206
omnetpp-p2	63%	2738376	20.8922	4.6500	2.7613
astar-p1	13%	71677	0.5469	2.4137	2.2794
astar-p2	82%	638161	4.8688	8.9179	6.5050
mcf-p1	81%	183045	1.3965	7.0717	7.0250

mcf-p2	74%	27431156	209.2831	NA	NA
bzip2-p1	1%	726036	5.5392	1.6724	0.5850
bzip2-p2	<1%	1569030	11.9707	1.5796	0.5113
hmmer-p1	3%	272263	2.0772	1.8454	1.0588
hmmer-p2	3%	288733	2.2029	1.8370	1.1391
leslie3d-p1	35%	649200	4.9530	2.3539	1.9611
leslie3d-p2	35%	751143	5.7308	2.3274	1.9358
Geometric mean			7.1561	3.0890	1.7530

From the results we make the following observations:

1. High potential compressibility: Huffman coding has a high potential compressibility when compared to other compression schemes. According to Table 4.1, the average CF of the 7 benchmarks with and without VFT consideration are 2.8749X and 3.8627X, respectively. We will later see that the former CF can be close to the latter after some modifications. As most previous studies report a compression factor of less than 2X, this result is quite promising.

2. Impact of value distribution: Huffman coding is data dependent and the potential compressibility of it highly depends on the established value distribution. The values in the second column of Table 4.1 are the proportion of the most 10 frequent values in each benchmark application phase which is calculated by dividing the total frequency of top 10 frequent values by total frequency of all the values and can be considered as a standard of value distribution. For example, in *astar* phase 2 and *mcf* phase1, the top 10 frequent values occupy more than 80% of the memory footprint after running the entire 1 billion instructions. As a result, Huffman coding achieves extremely high compression factors. To be more specific, the CF in *astar* phase 2 is 8.9179X if the metadata of VFT is not considered. On the other hand, if we use the equation 4.1 to calculate the CF with considering metadata, the $CF_{with\ VFT}$ of *astar* phase 2 degrades to 6.5050X and *mcf* phase1 achieves the highest $CF_{with\ VFT}$ of 7.025X among all the applications. For the application *bzip2*, the top 10 frequent values occupy only 1% of the memory footprint, resulting in a negligible CF. In such cases, when accounting for the size of the VFT, the CF is reduced to

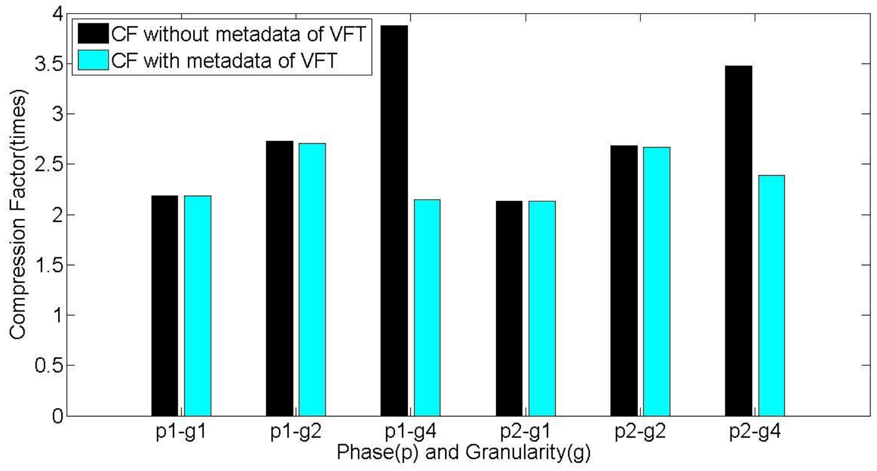
less than 1X, meaning that the overheads due to metadata were larger than the compression benefits. In such cases (*bzip2* and *hmmer*), it may be better if compression is avoided.

3. Impact of metadata overhead: as shown in Table 4.1, the potential compressibility (in terms of CF) is seriously degraded when we consider the overhead due to metadata. Fitting all the unique values appear in memory would require many MBs for the VFT. In particular, the phase 2 of *mcf* has more than 27 million unique values that require a huge VFT of 209MB. Moreover, the workload of building a Huffman tree including 27M values is prohibitive in terms of execution time as well as computational resources. The latter is the reason CF is not calculated for the second phase of *mcf*. There are two options to reduce the size of VFT. First, we can choose smaller granularity than 4 bytes. For example, if we consider the granularity of 2 bytes, there are at most 65536 (2^{16}) combinations, thus the size of VFT is no larger than 400KB. This option is tested in Section 4.2. In alternative, only the most frequent values are compressed and the rest are saved uncompressed instead. This way, less metadata is required due to VFT. This part of work is described in Section 4.3.

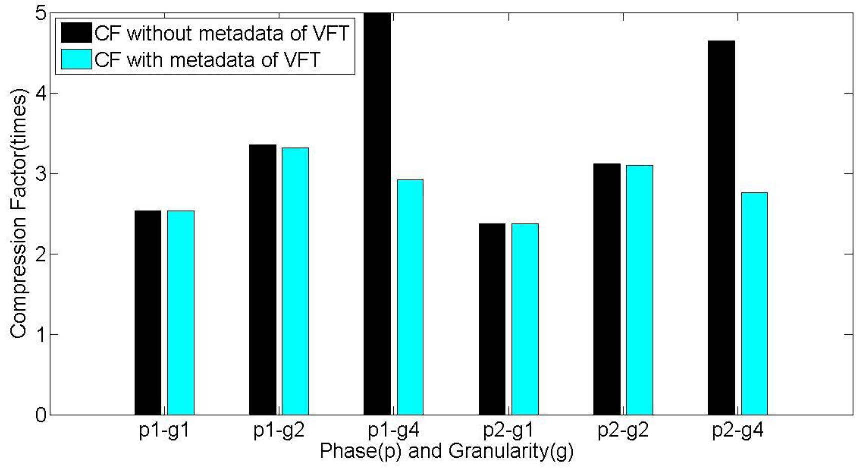
4.2 Impact of Compression Granularity

In the previous section, we observe that the Huffman-based compression approach performs well with the granularity of 4 bytes. However, we find that the VFT of 4 bytes may introduce prohibitive overheads due to metadata that have a negative impact on the potential of compressibility. Using smaller value granularities can help reducing the size of metadata. For example, a value of 2 bytes has only $2^{16} = 65536$ possibilities or even 1-byte values result in 256 combinations. If we use 4 bytes to store the weight information, then the upper bound of VFT size for 2-byte granularity is$(2 + 4)\text{bytes} * 65536 = 384\text{KB}$. Similarly, the upper bound of 1-byte VFT is 1.25KB. Consequently, coding with different granularities is a meaningful work for exploring the characteristic of the compression scheme.

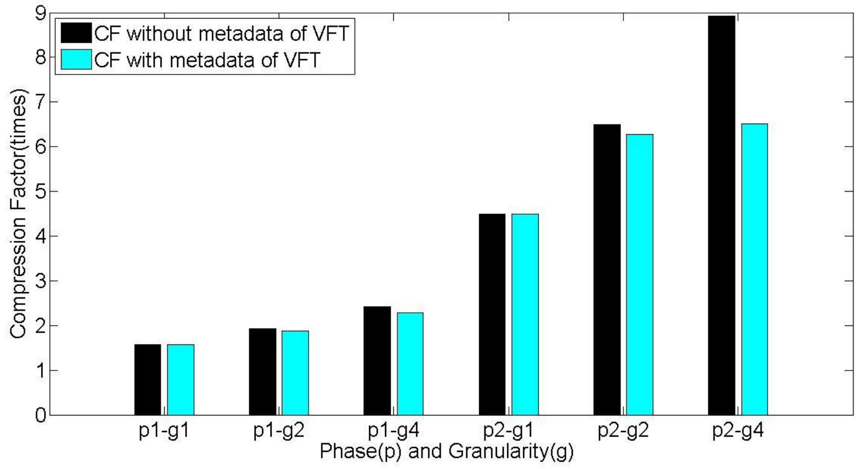
We study 3 different granularities for Huffman coding on the same benchmarks. The results are shown in Figure 4.3 and Table 4.2.



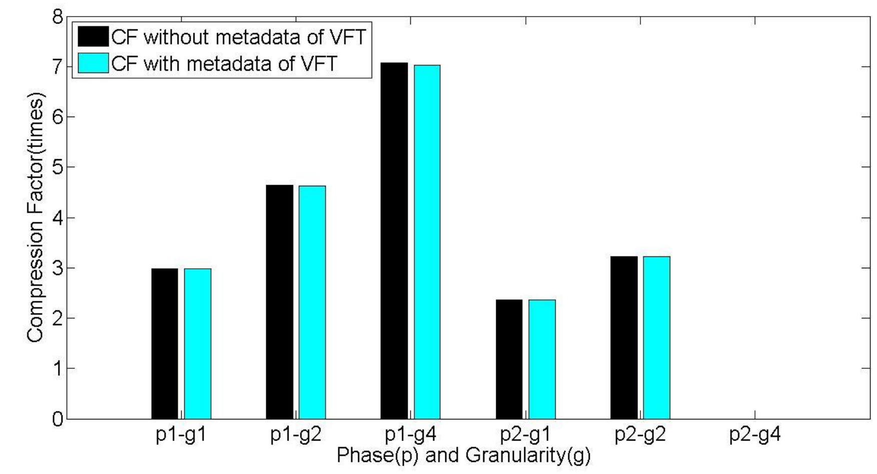
(a) xalan



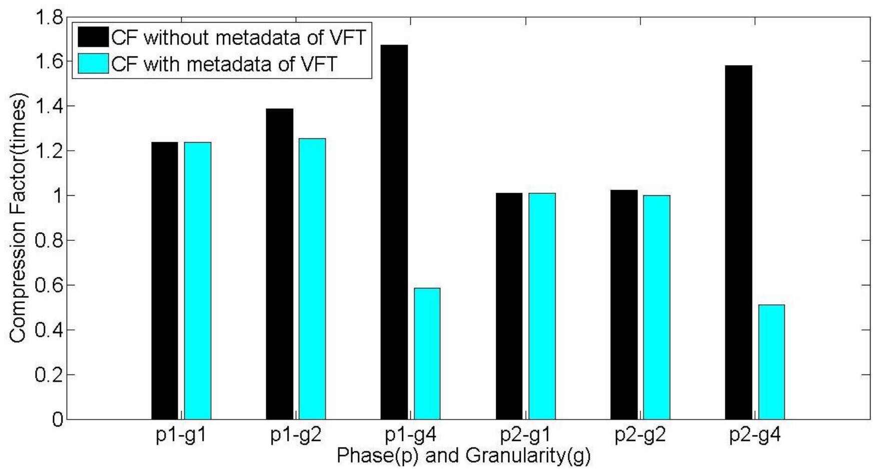
(b) omnetpp



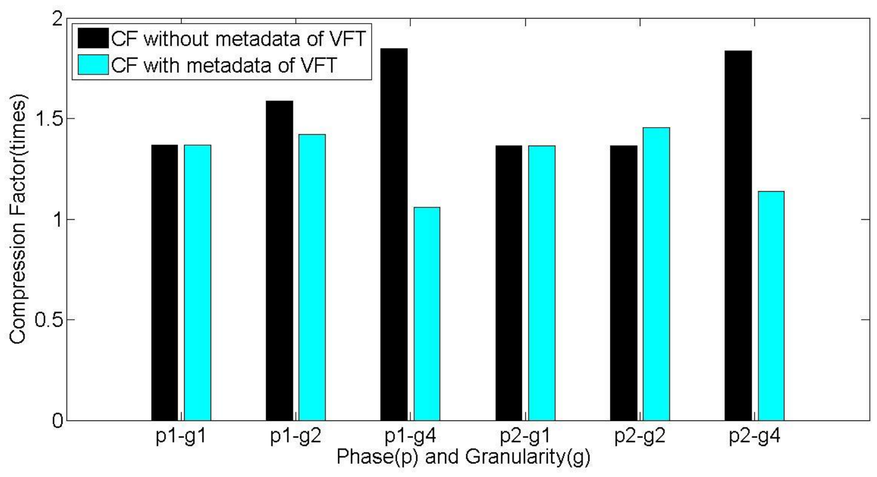
(c) astar



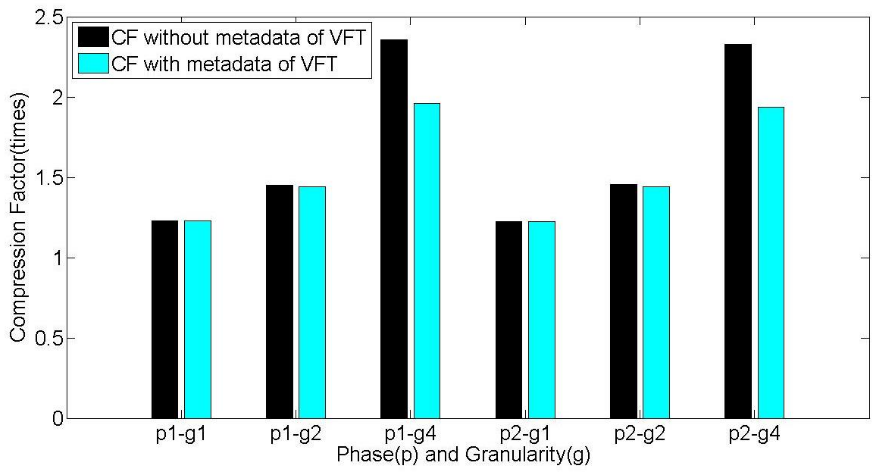
(d) mcf



(e) bzip2



(f) hmmer



(g) leslie3d

Figure 4.3: *Compression factors with and without metadata of VFT for granularity of 1 byte, 2 bytes and 4 bytes.*

Table 4.2 *Information about compression granularity, value distribution and compression result of each phase*

Benchmark -phase	**Granularity (byte)**	**Top10 Occurrence**	**CF$_{\text{with VFT}}$**	**CF$_{\text{w/o VFT}}$**	**Number of values has wider codewords / Number of values (percentage)**
xalan-p1	1	70%	2.1867	2.1867	206/256 (80.47%)
	2	60%	2.7260	2.7028	62221/65536 (94%)
	4	52%	3.8725	2.1495	0/3235506 (0%)
xalan-p2	1	73%	2.1301	2.1300	224/256 (87.5%)
	2	60%	2.6862	2.6703	63389/65536 (96.72%)
	4	29%	3.4731	2.3916	0/2885387 (0%)
omnetpp-p1	1	76%	2.5367	2.5366	208/256 (81.25%)
	2	71%	3.3502	3.3163	39434/45589 (86.5%)
	4	67%	4.9919	2.9205	0/1590773 (0%)
omnetpp-p2	1	73%	2.3686	2.3686	204/256 (79.69%)

	2	67%	3.1223	3.0968	58459/65536 (89.2%)
	4	63 %	4.6500	2.7613	0/2738376 (0%)
astar-p1	1	63%	1.5767	1.5766	213/256 (83.2%)
	2	58%	1.9216	1.8821	31342/42791 (73.24%)
	4	13%	2.4137	2.2794	0/71677 (0%)
astar-p2	1	92%	4.4834	4.4832	224/256 (87.5%)
	2	88%	6.5024	6.2697	58490/65536 (89.25%)
	4	82%	8.9178	6.5049	0/638161 (0%)
mcf-p1	1	89%	2.9814	2.9814	248/256 (96.88%)
	2	86%	4.6363	4.6322	44389/49178 (90.26%)
	4	81%	7.0717	7.0251	0/183045 (0%)
mcf-p2	1	82%	2.3576	2.3576	247/256 (96.48%)
	2	75%	3.2302	3.2279	59793/63485 (94.18%)
	4	74 %	NA	NA	NA
bzip2-p1	1	44%	1.2382	1.2378	216/256 (84.38%)
	2	41%	1.3867	1.2556	61166/65536 (93.33%)
	4	1%	1.6723	0.5850	0/726036 (0%)
bzip2-p2	1	10%	1.0092	1.0090	32/256 (12.5%)
	2	1%	1.0253,	0.9996	26717/65536 (40.77%)
	4	<1%	1.5796	0.5113	0/1569030 (0%)
hmmer-p1	1	51%	1.3687	1.3683	216/256 (84.38%)
	2	49 %	1.5867	1.4226	60635/65536 (92.52%)
	4	3%	1.8454	1.0588	0/272263 (0%)
hmmer-p2	1	51%	1.3641	1.3637	216/256 (84.38%)
	2	49%	1.5863	1.4553	60876/65536 (92.89%)
	4	3%	1.8370	1.1391	0/288733 (0%)
leslie3d-p1	1	38%	1.2266	1.2266	214/256 (83.59%)

	2	34%	1.4524	1.4389	57618/65536 (87.92%)
	4	35%	2.3539	1.9611	0/649200 (0%)
leslie3d-p2	1	38%	1.2253	1.2253	220/256 (85.94%)
	2	34%	1.4539	1.4419	57852/65536 (88.28%)
	4	35%	2.3207	1.9358	0/751143 (0%)

Interestingly, if we look at the CF without consideration of metadata due to the VFT (expressed as $CF_{w/o\ VFT}$) as the ideal potential compressibility, then for all the benchmark applications, Huffman coding in larger granularity always has higher compression potential than the one in smaller granularity. In other words, G4 (granularity of 4 bytes) is the best option for compression while G1 (granularity of 1 byte) is the worst. We don't explore coarser granularities (e.g., 8-byte) as data are typically processed in 4 bytes or smaller units in typical X86 ISAs. Thus compression with 8-byte or larger granularity would not achieve good result.

$CF_{w/o\ VFT}$ establishes a high potential compressibility for 4-byte granularity. However, the cost of metadata must be taken into account to make for more practical cases. Smaller granularity can reduce the metadata overhead. Unlike the situation of 4-byte granularity, we can see from Figure 4.3 that the compression factors with and without calculating the VFT size are extremely close when the granularity is 1-byte or 2-byte. That means the impact of metadata is negligible when granularity is small. As a result, 4 of 7 benchmarks have the highest $CF_{with\ VFT}$ (CF with consideration of metadata) when the granularity is 2-byte. Besides, *astar* has similar compression results between 2-byte and 4-byte granularities. In other words, in most applications 2-byte is the best compression granularity option according to its low metadata overhead.

Our third conclusion in this section is that for all the value granularities, the data distribution is always important to the compression performance. The data in the third column of Table 4.2 demonstrate that higher value of top 10 occurrences typically results in higher $CF_{w/o\ VFT}$ for all the compression granularities.

Another interesting observation is shown in the last column of Table 4.2. When the granularity is 1-byte or 2-byte, many values in the VFT are assigned to longer Huffman codewords than their original representations. On the other hand, when the compression granularity is set to 4-byte, it is observed that all the codewords become narrower than 4 bytes. As granularity becomes finer, we observe that: (1) the number of unique values appear in the memory grows (higher percentage of appeared unique values with reference to the total number of representable values -- Table 4.2) and (2) the amount of *efficient*[1] Huffman coding is further constrained, as the bit-width of the original binary representation becomes shorter. For example, in *astar*-p2-g2 despite the high $CF_{with\ VFT}$ of 6.27X, 90% of values in VFT are replaced by wider than 16-bits codewords. Therefore the compression algorithm builds a deep Huffman tree based on the VFT-all. As a result many values in the tree are associated to wider codewords than the initial representation! That is one of the reason that the $CF_{w/o\ VFT}$ of larger granularity is always better than smaller one. However, even if we can make all the codewords in G2 compression to be no wider than 16 bits, it is still hard for G2 to achieve a high $CF_{w/o\ VFT}$ as G4 if the application has a high potential for compressing. That is because although most values in G2 are assigned with wider codewords, their total frequency is low and the width of codewords is no wider than 18 or 19 bits in most cases. Even if all the wide codewords can be narrowed to 16 bits, the space freed up is much smaller than the saved space by applying G4 compression. The reason behind is G4 has advantage than G2 in compressing frequent values. For example, for G4 compression, 32 bits can be compressed to 1 bit while at least 2 bits are needed for G2 compression. For the applications which have high compression potentials, several most frequent values occupy most space of the footprint. As a result, G4 compression can benefit a lot from the concentrated value distribution and achieve a much higher CF than G2. However, high compressibility is not always guaranteed with coarser value granularity when metadata is considered.

In conclusion, 1-byte granularity is not a good choice as it has lowest compression potential in the 3 granularity alternatives. 2-byte granularity is competitive since it has a much smaller VFT overhead than 4-byte one. For 4-byte granularity, it has the highest compression potential due to its good performance in compressing all the values in VFT. The overhead of keeping VFT in memory is a big problem but also means there is a potential for modification. To reduce the metadata overhead, the VFT-X-mfv can be considered which only includes top X frequent values keeping a small size. Another reason for considering VFT-X-mfv is based on the observation that for small granularities, most values are assigned with wider codewords when the VFT-all is used. Thus keeping those values out of VFT may benefit in time and power at the expense of a negligible performance losses. An attempt of VFT-X-mfv is done in the next section.

[1] by *efficient* here we mean denser than the original representation's width

In general, selecting the appropriate value granularity in statistical compression is not straightforward as there are many implications. This section as well as the next one explore this problem and analyze the implications. Another implication that is not discussed, as is beyond the scope of this thesis, is that the number of compressed/decompressed values in a block or a page increases, as the granularity becomes finer. This not only affects compression, as was previously discussed, but it may also affect the processing time, energy and throughput of the memory system when (de)compressing.

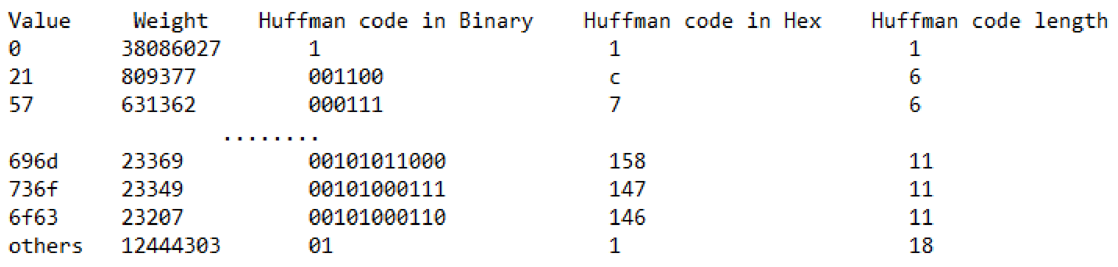
4.3 Huffman Coding with VFTs of Different Sizes

As discussed in Section 4.2, there are 3 drawbacks of using a full-size VFT for Huffman coding. First, it leads to a huge overhead for storing the information of VFT. Second, compressing all the values is not efficient when most of the codewords are wider than their original representations. Third, the processing overheads for encoding and decoding may grow due to the big VFT tables.

On the other hand, the VFT-X-mfv can be considered. In a VFT-X-mfv, only a number of most frequent values are included. As a result, only these values are encoded using Huffman coding. Infrequent values that are not tracked by VFT are kept as uncompressed. Applying such algorithm can result in several benefits. First, less overhead due to metadata are needed. Second, the processing overheads of building the Huffman tree as well as encoding and decoding can be reduced. This may happen as searching for the corresponding codeword of a value in a small table takes less time than in a big table which includes thousands of entries. Finally, since the number of values decreases, the maximum and average lengths of codewords become narrower which may simplify the decoding work. The last benefit is based on the regular Huffman coding scheme which need to decode the codewords bit by bit. As mentioned in Section 2.3.2, a well-designed decompressor can accelerate the process and decode the codewords of any length in the same time. Thus, the decoding time cannot be reduced by shortening the average codeword length.

There are two main issues by using VFT-X-mfv: (1) distinguish between compressed and uncompressed values and (2) selecting an appropriate VFT size that will reduce the metadata without trading off the high compressibility potential.

The first issue is tackled by using a unique code as prefix for all the uncompressed values to differentiate from other compressed values. As same as other Huffman codewords, the unique code should not be any other codewords' prefix. Figure 4.4 is an example of Huffman coding based on a VFT-128-mfv which only includes 128 values for 2-byte granularity. To be more specific, only the top 127 frequent values are assigned with Huffman codewords. The other infrequent values are saved uncompressed by preceding them with the same unique code. During the encoding process, frequent values are replaced by their corresponding codewords when stored. The infrequent values are stored with the prefix (unique code) and followed with their original representations. For decoding, when the decompressor detects the unique code, it will read the following 2 bytes for the information of that infrequent value.

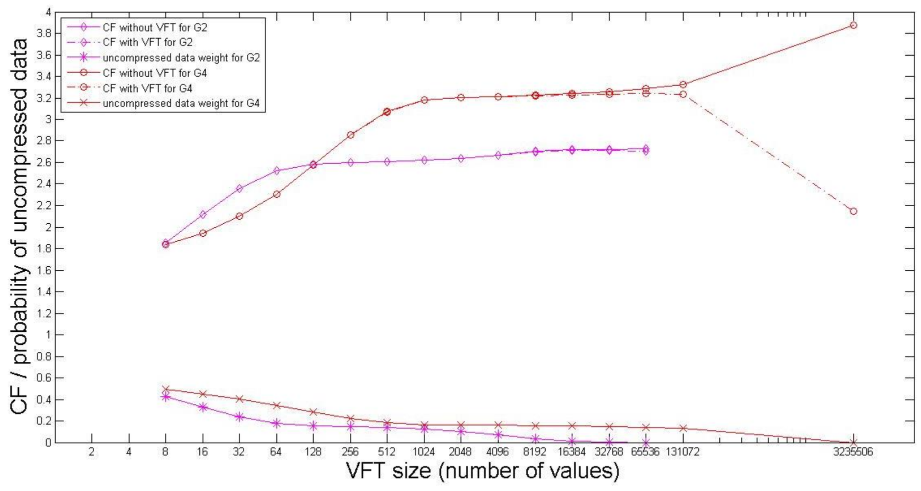


```
Value     Weight     Huffman code in Binary     Huffman code in Hex     Huffman code length
0         38086027       1                          1                       1
21        809377         001100                     c                       6
57        631362         000111                     7                       6
                   ........
696d      23369          00101011000                158                     11
736f      23349          00101000111                147                     11
6f63      23207          00101000110                146                     11
others    12444303       01                         1                       18
```

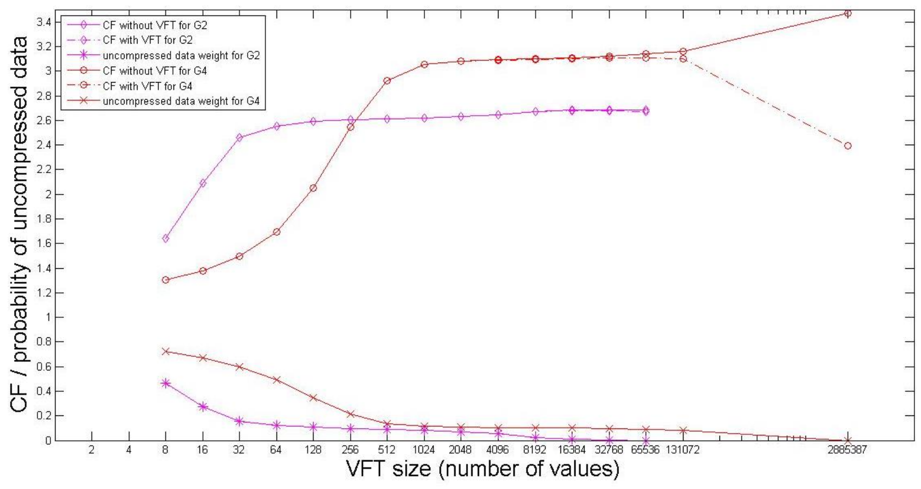
Figure 4.4: *Huffman coding on VFT-128-mfv.*

In Figure 4.4, we can see that all the uncompressed values are stored with the prefix of 01. When the program detects 01, the next 16 bits (2 bytes) will be read. The total length of a compressed value is 2+16=18 bits which is 2 bits wider than its original representation. The unique code (prefix) for uncompressed values is calculated in the same way as other values in VFT. When building the Huffman tree, we combine all the infrequent values together as a new value. The frequency of the new value is the sum of all the infrequent values' frequencies (subtracting the total count of the frequent values from the total values stored in the current memory footprint). In this case, the total frequency of infrequent values is high thus the prefix is only 2 bits long. Therefore, the more the infrequent values the narrower codewords are used to expand them.

The second issue is which VFT size is ideal to be used for compression. We experimentally study this issue by using different VFT sizes to establish the frequency distribution of value sub-sets and therefore limiting set codewords. The results are shown in Figure 4.5 and Table 4.3. As G1 (granularity of 1 byte) is proved as a bad compression option, we focus on the comparison between G2 (granularity of 2 bytes) and G4 (granularity of 4 bytes). In each sub-figure of Figure 4.5, there are 6 curves for these two granularities. For each granularity, there are 2 curves for CFs (with and without accounting for the VFT size) and a third curve that illustrates the probability of weight of values that are kept uncompressed. As the size of VFT grows, the probability of uncompressed values is reduced until becoming zero 0.

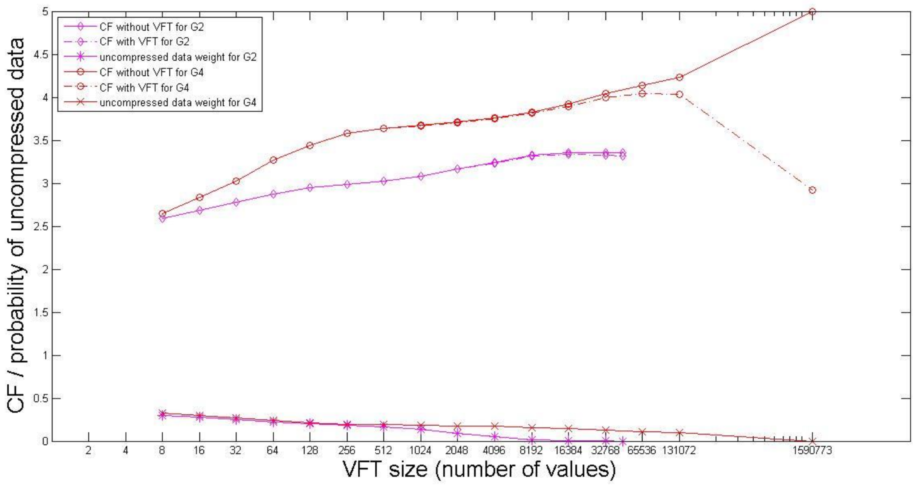


(a) xalan-phase 1

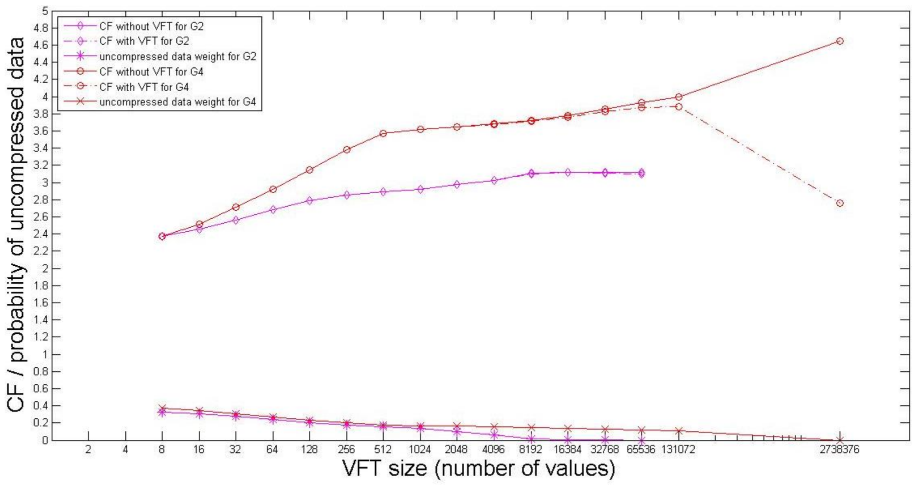


(b) xalan-phase 2

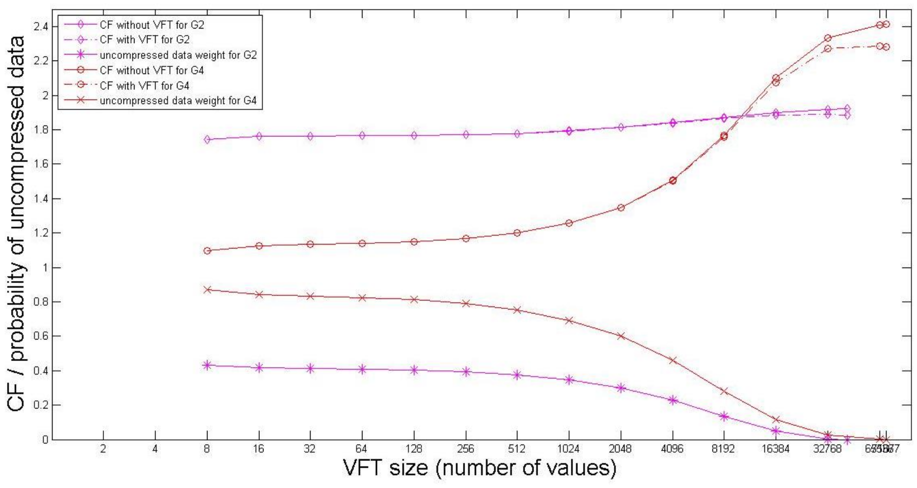
Both phases of *xalan* show similar results. That is, when the VFT size is small, the $CF_{\text{with VFT}}$ for G2 is higher than G4. As the size of VFT increases, the CF for G4 catches up then exceeds CF for G2. As the size of VFT become huge (more than 64K-entry for G4), the $CF_{\text{with VFT}}$ degrades significantly due to the high metadata overload.



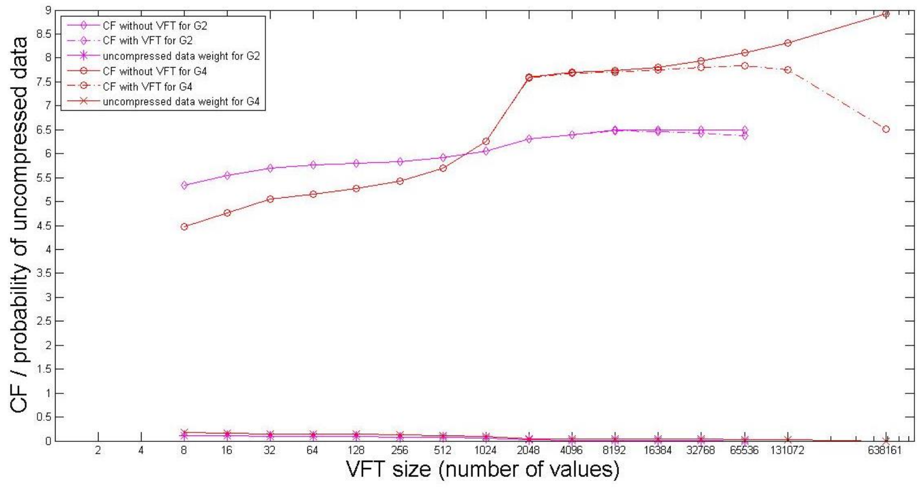
(c) omnetpp-phase 1



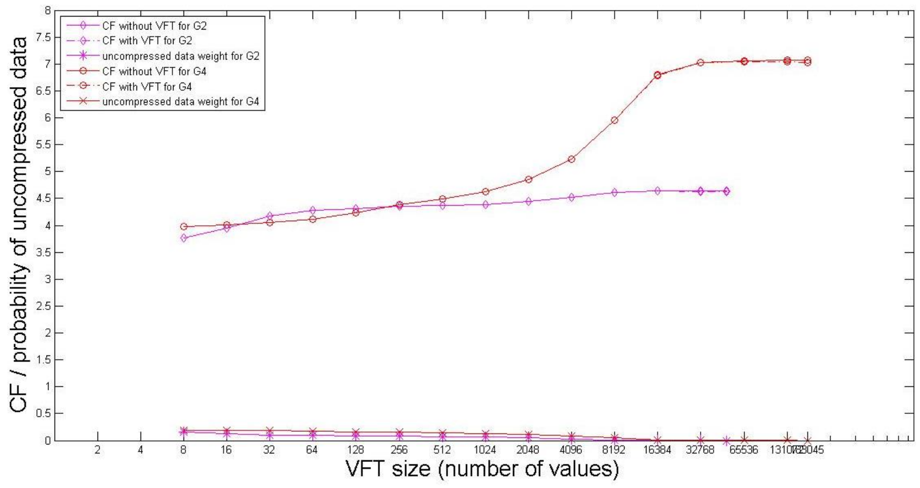
(d) omnetpp-phase 2



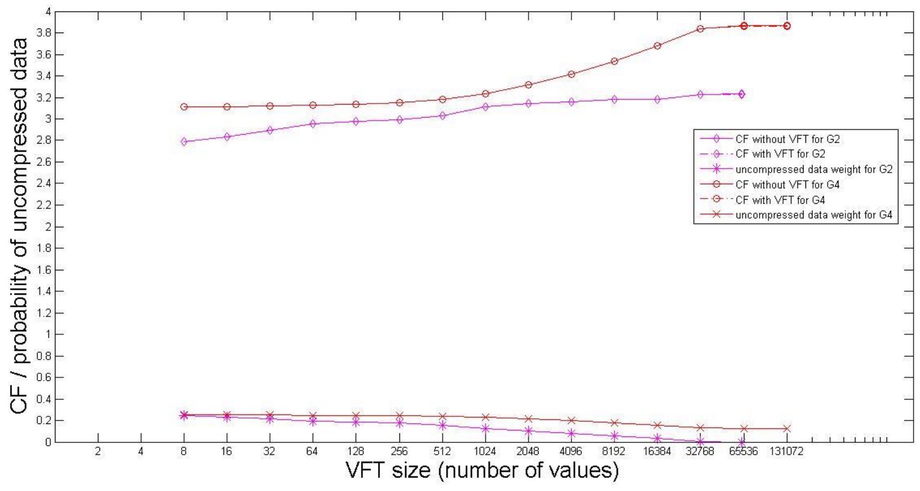
(e) astar-phase 1



(f) astar-phase 2

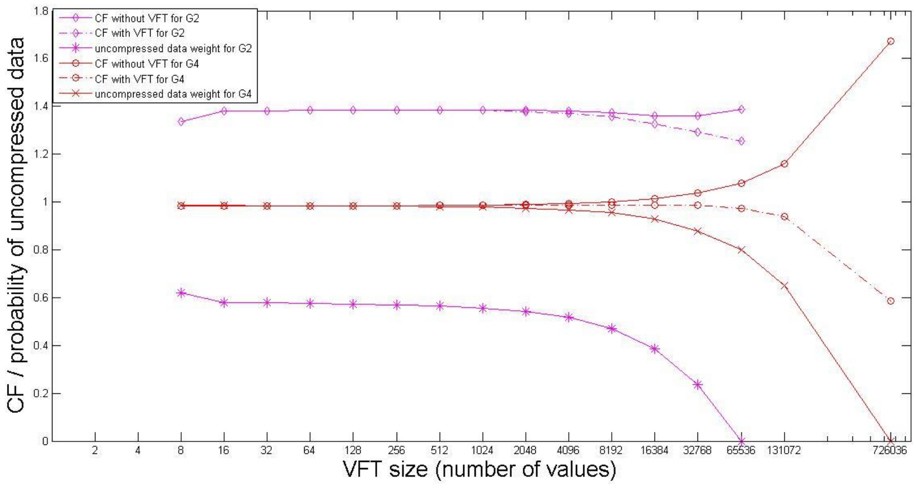


(g) mcf-phase 1

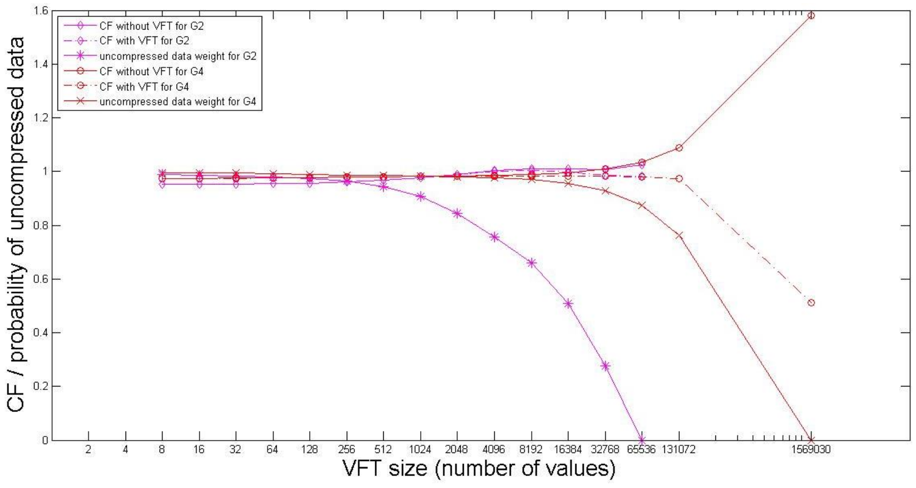


(h) mcf- phase 2

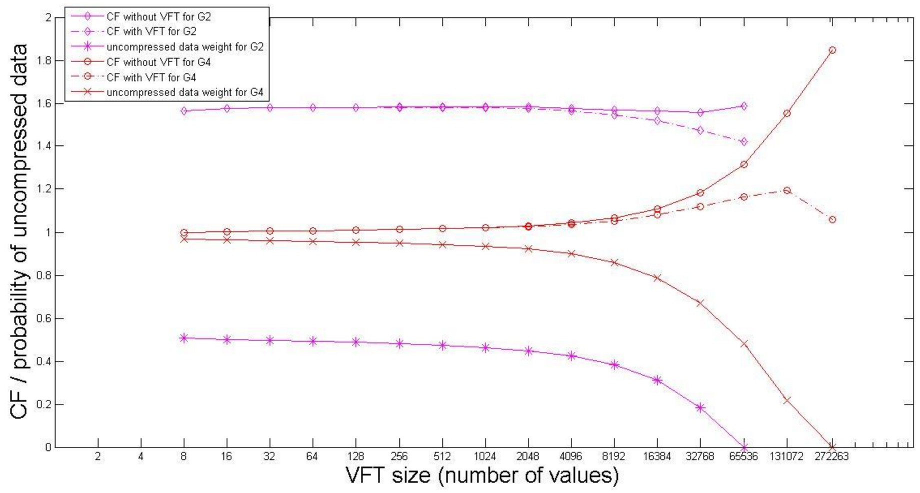
The applications *xalan, astar, omnetpp* and *mcf* have similar results. G4 can achieve the best compression result if the VFT-X-mfv is applied. On the other hand, the VFT-all introduces big overhead due to metadata and leads to a significant degradation of compressibility for G4 while the impact of metadata is much smaller for G2. That is why CF for G2 is higher than G4 for VFT-all in most cases.



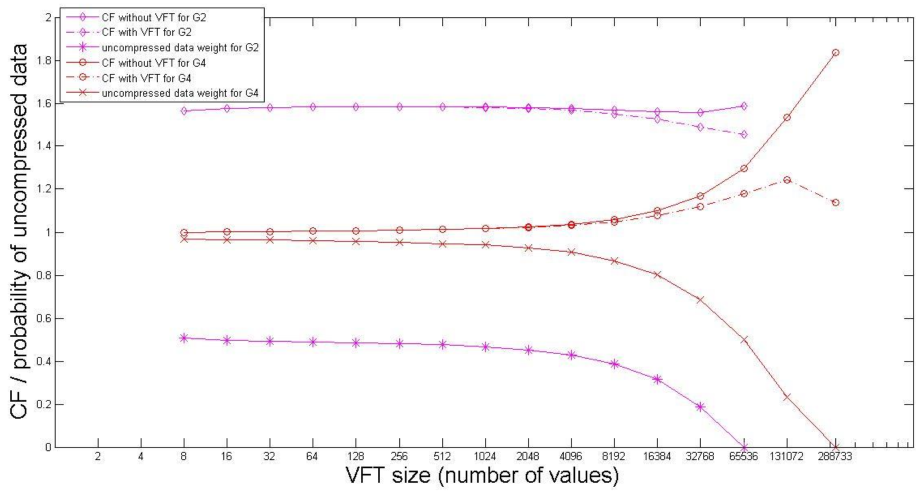
(i) bzip2-phase 1



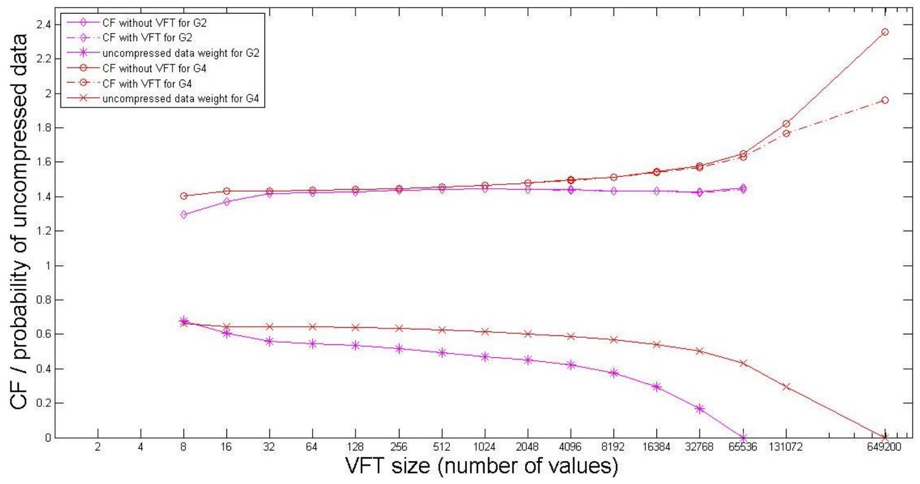
(j) bzip2-phase 2



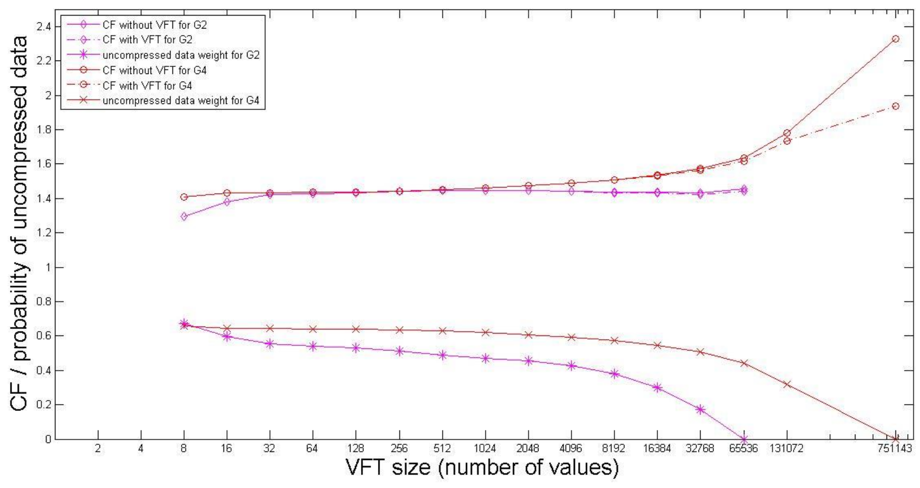
(k) hmmer-phase 1



(l) hmmer-phase 2



(m) leslie3d-phase 1



(n) leslie3d-phase 2

Figure 4.5: *Compression factor and probability of uncompressed data vs. VFT size.*

For the applications with low compression potentials such as *bzip2, hmmer* and *leslie3d*, the benefit of applying bigger VFT is much smaller than in other applications. As a result, the curves of CF keep stable with the growing of VFT size in Figure 4.5(i)-(n). More details about the compression are shown in Table 4.3.

Table 4.3 *Comparison between the compression results by applying different VFT sizes.*

Benchmark -phase	**Granularity**	**Peak $CF_{with\ VFT}$**	**$CF_{with\ VFT}$ for VFT-all**	**VFT size for peak CF with VFT /VFT-all (percentage)**
xalan-p1	2	2.7165	2.7029	16384/65536 (25%)
	4	3.2433	2.1495	65536/3235506 (2.03%)
xalan-p2	2	2.6796	2.6703	16384/65536 (25%)
	4	3.1083	2.3916	65536/2885387 (2.27%)
omnetpp-p1	2	3.3371	3.3164	16384/45589 (35.94%)
	4	4.0387	2.9206	65536/1590773 (4.12%)
omnetpp-p2	2	3.1136	3.0968	16384/ 65536 (25%)
	4	3.8820	2.7613	131072/2738376 (4.79%)
astar-p1	2	1.8883	1.8821	32768/42791 (76.58%)

	4	2.2870	2.2794	65536/71677 (91.43%)
astar-p2	2	6.4816	6.3697	8192/65536 (12.5%)
	4	7.8401	6.5050	65536/ 638161 (10.27%)
mcf-p1	2	4.6342	4.6322	16384/49178 (33.32%)
	4	7.0457	7.0251	65536/183045 (35.8%)
mcf-p2	2	3.2290	3.2279	32768/63485 (51.62%)
	4	3.8609	3.8579*	65536/ 131072 (50%)[2]
bzip2-p1	2	1.3837	1.2557	256/65536 (0.39%)
	4	0.9875	0.5850	16384/726036 (2.26%)
bzip2-p2	2	1.0035	0.9835	8192/ 65536 (12.5%)
	4	0.9820	0.5113	16384/1569030 (1.04%)
hmmer-p1	2	1.5809	1.4226	256/65536 (256/65536)
	4	1.1939	1.0588	131072/272263 (48.14%)
hmmer-p2	2	1.5822	1.4553	256/65536 (256/65536)
	4	1.2438	1.1391	131072/ 288733 (45.4%)
leslie3d-p1	2	1.4444	1.4389	1024/65536 (1.56%)
	4	1.9611	1.9611	649200/649200 (100%)
leslie3d-p2	2	1.4473	1.4419	1024/65536 (1.56%)
	4	1.9358	1.9358	751143/751143 (100%)

An exciting observation is found that for most of the applications the highest (peak) $CF_{\text{with VFT}}$ is achieved with the VFT-X-mfv instead of the VFT-all. To be more specific, all the curves in Figure 4.5 show similar trend. That is, the compression factor is at a low level when VFT is small. As the number of values in VFT increases, $CF_{\text{with VFT}}$ grows until it reaches the peak value. If we further increase the VFT size, the metadata overhead of VFT outweighs the gain of compression. Thus, $CF_{\text{with VFT}}$ begin to decrease although $CF_{\text{w/o VFT}}$ still keeps increasing as more and more values are included in VFT

[2] *For application mcf phase2, we only evaluate the compression results for VFT smaller than 131K-entry.*

and therefore compressed. By using VFT-X-mfv, we can close the difference between $CF_{with\ VFT}$ and $CF_{w/o\ VFT}$. As a result, the average $CF_{with\ VFT}$ for all the applications are 2.6087X for G2 and 3.1150X for G4, respectively. In particular, the *astar*-p2 has the highest $CF_{with\ VFT}$ of 6.4816X and 7.8401X for G2 and G4, respectively.

By applying smaller VFT, most applications obtain the highest $CF_{with\ VFT}$ by using the granularity of 4 bytes. However, for the applications with low compression potentials such as *hmmer* and *bzip2*, G2 achieves the best compression result. That is because the values in such applications distribute evenly which results in a poor compressibility for G4 compression. On the other hand, as G2 has much fewer value possibilities than G4 which introduces a relatively concentrated value distribution as shown in Table 4.2, the compression result for G2 is better than G4 in this way.

As discussed in the beginning of the section, if different Huffman coding introduces close compression factors, the one with smaller Huffman tree and fewer values compressed is preferred since it can reduce the computational overheads of encoding and decoding. If we explore Figure 4.5 carefully, in most situations a 16K-entry VFT (expressed as VFT-16K-mfv) which includes 16384 values is enough to achieve a $CF_{with\ VFT}$ close to the peak $CF_{with\ VFT}$ for G4 Huffman coding. VFT requirements can be further reduced without substantially compromising compression. For example, VFT-1K-mfv introduces the $CF_{with\ VFT}$ 0.3911X (15.21%) lower than the one for VFT-16K-mfv. For G2, the $CF_{with\ VFT}$ grows slightly as the VFT size increases. In most cases, the compression result is close to the peak value when the most frequent 8K values are kept in VFT. In general, the VFT of 128 and 8K values are two good alternatives for G2 compression. In particular, the $CF_{with\ VFT}$ for VFT-128-mfv on average is 2.4372X which is 6.25% lower than the $CF_{with\ VFT}$ (2.5894X) for VFT-8K-mfv.

Another interesting finding is that when the application has a low compression potential (smaller than 2X), G2 is the best granularity option for compression. For example, G2 has a significant advantage than G4 for the compression of *bzip2* (Figure 4.5(a)(b)) and *xalan* (Figure 4.5(i)(j)). When it comes to the applications with high compression potential, two situations should be considered separately. If VFT is small, for example less than 1K-entry (including 1024 values), G2 may achieve better compression result than G4 as shown in Figure 4.5(d)(g)(h)). As VFT size grows, G4 becomes the best granularity option and results in the peak CF for the application. It must be mentioned here that when CF for G2 and G4 are close, G4 is a better choice since it compress and decompress double size of data than G2 at one time which speeds up the encoding/decoding process.

4.4 Huffman Coding Based on 7M Write Accesses

In the previous sections, in all experiments, Huffman coding is generated based on the value distribution of the footprint established at the end of the simulation phase (Simpoint). As mentioned in Section 2.3.1, for a given set of values and based on the value frequency distribution, Huffman coding can be so optimal that when used in memory compression the footprint size can be significantly reduced. However, it is almost impossible in practice, to use optimal codewords in real-time compression, as the memory content and the used value-set changes. As the applications execute, the value distribution in memory keeps changing and static Huffman codewords may become more and more inefficient for compression. Thus, the compression performance during the execution should be monitored. If the compression factor decreases to an unacceptable level, say a set threshold, the new coding must be regenerated.

In this section, we evaluate Huffman coding in a more practical approach and study how memory compressibility is affected through time. Another dimension of this problem is the amount of memory accesses needed to be recorded for establishing an accurate data distribution statistics and building an efficient Huffman tree. In previous experiments, the footprint, at the end of each application phase, is used instead but this way the value distribution depends on all the memory accesses during the phase. In reality, the time cost for data collection should be concerned. The amount of data to be collected should be enough to reflect the data distribution of the most frequently appeared values, but at the same time it must small enough for reducing the overhead of building Huffman tree.

Based on above considerations, a new type of experiment is designed and applied in the following steps:

1. Read the memory trace files (obtained in Step1 of a typical experiment as mentioned in Section 3.1) for a certain interval of the simulation phase and monitor the value statistics of the accessed pages in this interval. Instead of establishing the statistics and generating Huffman coding at the end of the phase, it is now done after having executed 7M write accesses. The interval of 7M write accesses corresponds to approximately 50M committed instructions, as one store instructions occurs every 5-6 instructions.
2. Generate VFT based on the accessed pages' data after executing 7M write accesses.
3. Generate Huffman coding based on the VFT.

4. This step can be divided into 2 parts to evaluate two intervals (7M-write accesses / the rest part) of the phase.

a) Evaluate the compressibility of the 7M write accesses interval of the phase. Unlike previous experiments, Huffman coding is not applied on the accessed pages in footprint but on each page accessed. For each access (read/write of an address that requires a new allocated page or a modification of an existing page), the compressed size of the page where the access appears is calculated and recorded. In previous experiments, the compression of each page in the footprint was monitored only once. In this experiment, it is possible for one page to be calculated for its compressed size for as many times as it is touched by the application[3]. This way, we monitor all the possible attempts of compressing each page. We calculate the compressed page size for each access during the interval, and the average CF can be calculated with the equation 4.2:

$$CF_{average} = \frac{n_{accesses} * 4KB}{metadata + \sum compressed\ size\ of\ page} \tag{4.2}$$

Metadata is the VFT size, $n_{accesses}$ is the number of accesses in any interval we select. For example, we can calculate the $CF_{average}$ for the beginning part of the phase which includes one million accesses, thus $n_{accesses}$ is 1M. For calculating the $CF_{average}$ of the interval of 7M write accesses, $n_{accesses}$ is sum of the number of write accesses (7M) and all the *new read* accesses(defined in Section 3.1.2) which occur in the new pages that have never been touched by the program before. The $CF_{average}$ of the interval of 7M write accesses is used as a compression performance reference to see how much compression deteriorates when the same Huffman codewords are used for compression in the rest part of the phase.

b) In step (b), we evaluate the compressibility of the rest part of the simulation phase using the Huffman codewords generated from step (a). As same as step (a), for each access we track the compressed size of that accessed page. According to equation 4.2, the changes of $CF_{average}$ from the beginning to the end of the interval can be monitored. Thus, the impact of data distribution variation on the compression performance can be evaluated.

[3] For read accesses, the compressed size is calculated only for the first read that brings the page to memory.

5. In step 4(b), we focus on the *global* changing trend of the compressibility during the execution. *Global* compressibility is based on all the accesses appear in the phase. For example, when we select a time point to check the compressibility, the $CF_{average}$ until that time point is used to measure the compression performance which is calculated by counting all the compressed page sizes from the first access in the interval to the last access at the time point. If the $CF_{average}$ is high, we assume that the Huffman codewords are still efficient and there is no need for updating. However, it is possible that the Huffman codewords are not suitable anymore for compressing while this situation is covered up by the average compression result. Thus, in addition to how *global* compressibility changes, a *local* $CF_{average}$ is calculated as well. *Local* $CF_{average}$ is calculated after dividing the whole interval in smaller chunks. Thus, the *Local* $CF_{average}$ of one chunk is only based on the access in that chunk. In this way, if the *global* $CF_{average}$ is stable but the *local* $CF_{average}$ of a small interval (chunk) has a significant fluctuation, it may be important to generate new encoding to improve the compression performance, especially if the deviation from the reference compression keeps up growing for the next intervals. In this step, two types of *local* $CF_{average}$ have been calculated. First, we divide the whole interval into small chunks of 1M accesses and calculate the $CF_{average}$ of each 1M chunk. Second, as monitoring the whole process may be expensive in practice, we use a method of sampling instead. For each 100K accesses, we only evaluate $CF_{average}$ of 10K accesses. In this way, 90% of compression monitoring can be saved.

Due to time and simulation-resource limitations, we pick five of the seven applications for this experiment which include applications with different compression potentials (high/medium/low) and the experimental results are shown in Table 4.4. For *omnetpp* phase 1, there are few accesses during the phase thus we take it out of the experiment. Based on our previous discussion, for G4 granularity, we choose VFT-1K-mfv and VFT-16K-mfv instead of the VFT-all. For G2, the two alternatives are VFT-128-mfv and VFT-8K-mfv.

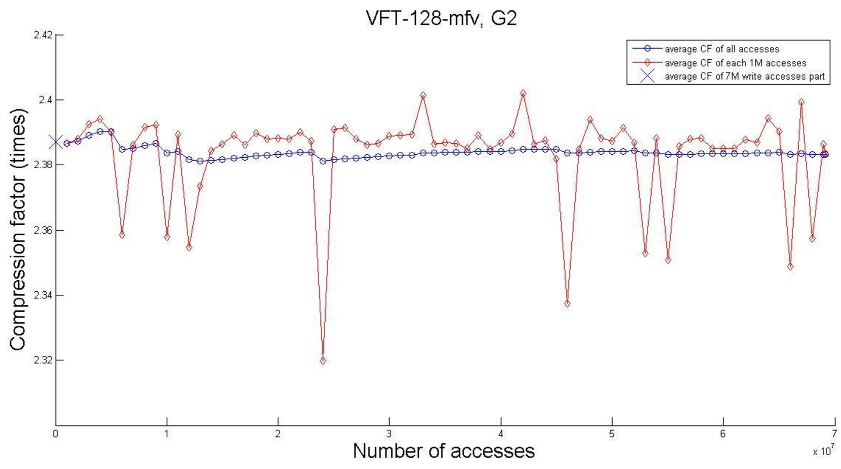
Table 4.4 *Comparison between $CF_{average}$ for both intervals (7M/rest) of each phase, range of sampling of $CF_{average}$ monitoring and $CF_{with\ VFT}$ (from Section 4.3)*

Benchmark-phase -granularity-VFT size	**$CF_{average}$ for the 7M write accesses interval**	**$CF_{average}$ for the rest part of the phase**	**Peak/bottom$CF_{average}$ when sampling the compressibility monitoring**	**$CF_{with\ VFT}$ (result from Section 4.3)**
xalan-p1-g2-128	2.3870	2.3832	2.9355/1.5870	2.5877
xalan-p1-g2-8192	2.3364	2.3232	3.1038/1.6041	2.7000
xalan-p1-g4-1024	2.1652	2.1564	3.4005/2.1315	3.1786
xalan-p1-g4-16384	2.5150	2.5060	3.4005/2.4319	3.2274
xalan-p2-g2-128	1.9756	2.0012	2.8311/1.8577	2.5939
xalan-p2-g2-8192	2.2126	2.2296	2.9297/2.0837	2.6717
xalan-p2-g4-1024	2.8168	2.8329	3.4014/2.2273	3.0518
xalan-p2-g4-16384	2.8925	2.8883	3.4577/2.3200	3.1000
omnetpp-p2-g2-128	2.4624	2.4612	2.8042/2.3451	2.7832
omnetpp-p2-g2-8192	2.5944	2.5787	2.9763/2.4594	3.1036
omnetpp-p2-g4-1024	2.5814	2.5514	3.3983/2.3609	3.6145
omnetpp-p2-g4-16384	2.8132	2.7770	3.5135/2.5920	3.7629
astar-p1-g2-128	2.5786	2.5910	2.8009/2.5379	1.7661
astar-p1-g2-8192	2.6131	2.6147	2.8011/2.5681	1.8644
astar-p1-g4-1024	2.5309	2.5052	2.7648/2.3202	1.2560
astar-p1-g4-16384	2.8125	2.7723	2.9930/2.6343	2.2731
astar-p2-g2-128	4.0814	4.0723	6.2230/.5173	5.7936
astar-p2-g2-8192	4.3423	4.3352	6.9536/3.8496	6.4816
astar-p2-g4-1024	4.1029	4.0962	6.7555/3.5348	6.2595
astar-p2-g4-16384	4.6422	4.6255	8.5006/4.0094	7.7442
mcf-p1-g2-128	4.0583	4.0079	4.1342/3.6821	4.3082
mcf-p1-g2-8192	4.2385	4.1751	4.3046/3.7765	4.6142

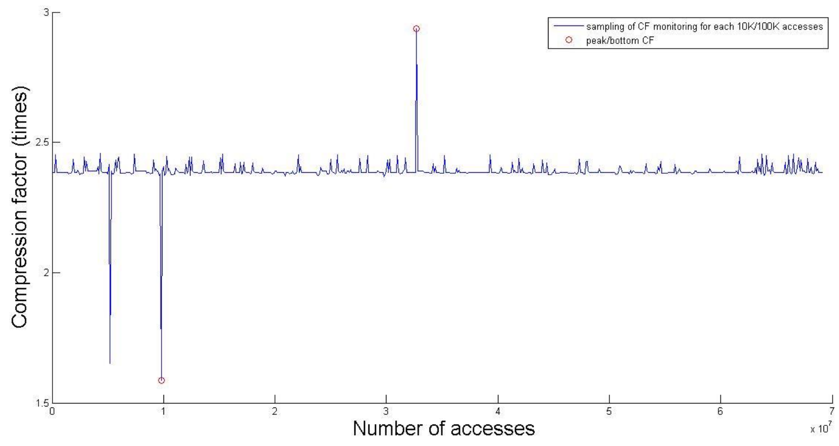
mcf-p1-g4-1024	4.3923	4.1911	4.6044/3.6090	4.6279
mcf -p1-g4-16384	5.5592	5.3348	5.7487/4.4178	6.7911
mcf -p2-g2-128	2.1952	2.1970	2.5233/2.0548	2.9761
mcf -p2-g2-8192	2.2627	2.2611	2.7053/2.0871	3.1811
mcf -p2-g4-1024	2.2031	2.2069	2.5237/2.0477	3.2348
mcf -p2-g4-16384	2.2645	2.2663	2.6741/2.0891	3.6797
bzip2-p1-g2-128	1.4831	1.1845	2.8406/1.0475	1.5269
bzip2-p1-g2-8192	1.4669	1.1807	2.8068/1.0464	1.5119
bzip2-p1-g4-1024	1.3091	1.1111	1.9641/0.9706	0.9849
bzip2-p1-g4-16384	1.2797	1.0966	1.7761/0.9743	1.0086
bzip2-p2-g2-128	1.2540	1.2290	1.6381/1.1412	1.0793
bzip2-p2-g2-8192	1.2845	1.2709	1.5791/1.1805	1.1608
bzip2-p2-g4-1024	1.2778	1.2176	1.7108/1.1016	1.1395
bzip2-p2-g4-16384	1.3584	1.2983	1.7089/1.1048	1.1510

The most important finding from the experiment is that the compression performance for static Huffman codewords keeps in a stable level during the execution of application. As shown in the second and third columns in Table 4.4, the $CF_{average}$ of the two parts (7M write accesses/rest) are close. Moreover, for the application *xalan* phase 2, the Huffman codewords based on the first 7M write accesses achieve better result for compressing the rest part of the phase.

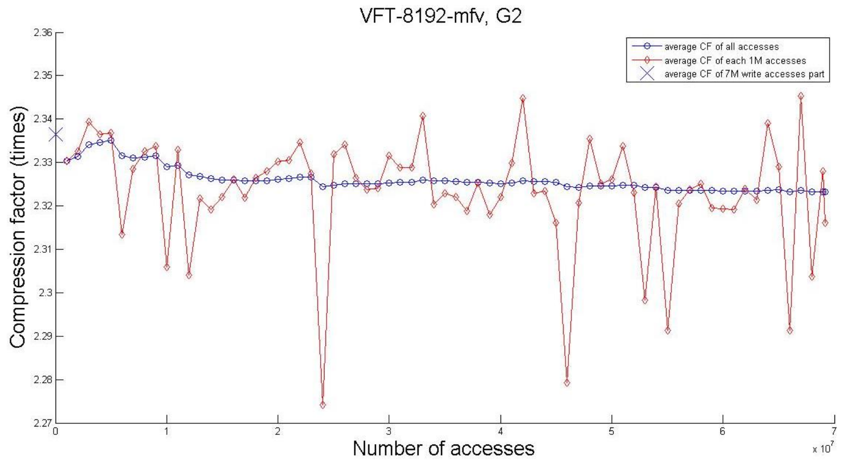
In most of cases, there is a small compressibility loss when applying Huffman coding, which is generated based on the first 7M write accesses of the phase, to the rest part of the phase. Figure 4.6 to Figure 4.14 show the results of 5 applications.



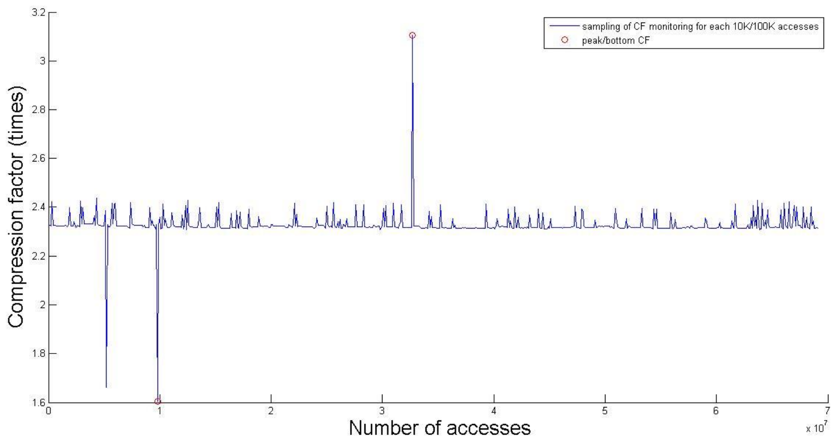
(a) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-128-mfv



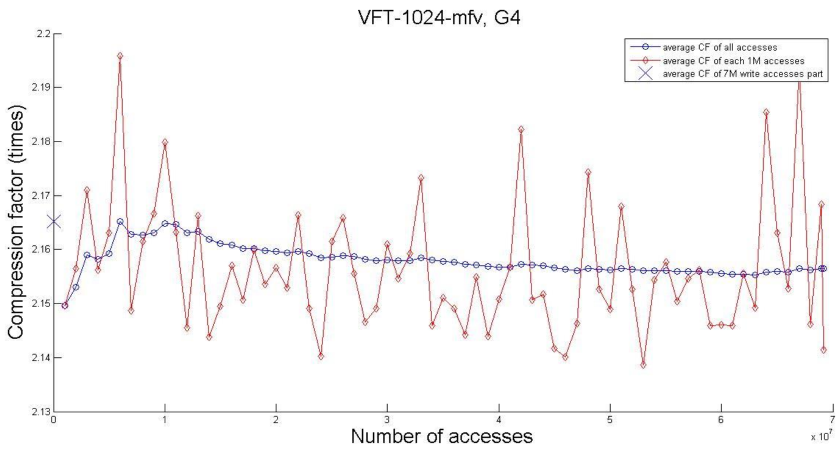
(b) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-128-mfv



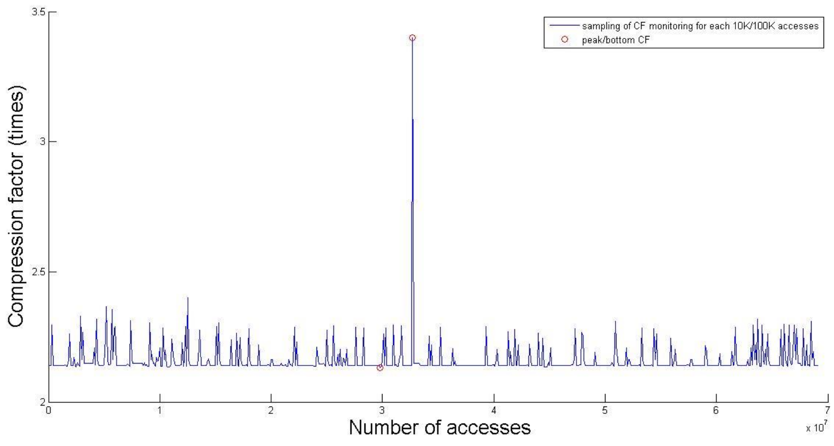
(c) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-8192-mfv



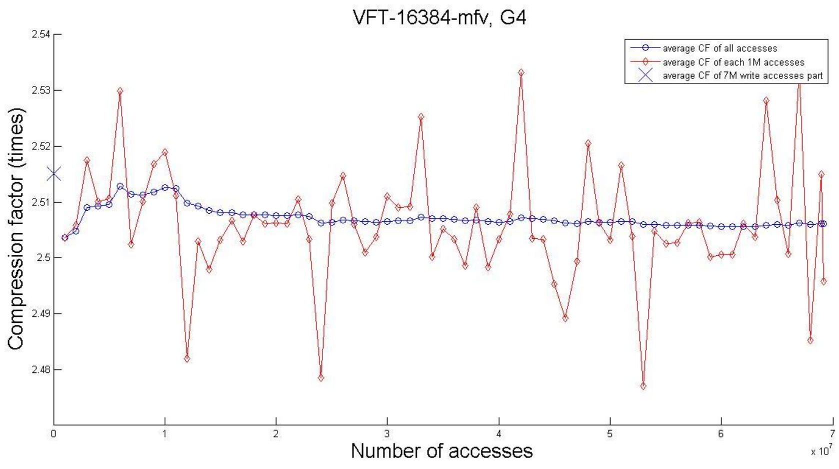
(d) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-8192-mfv



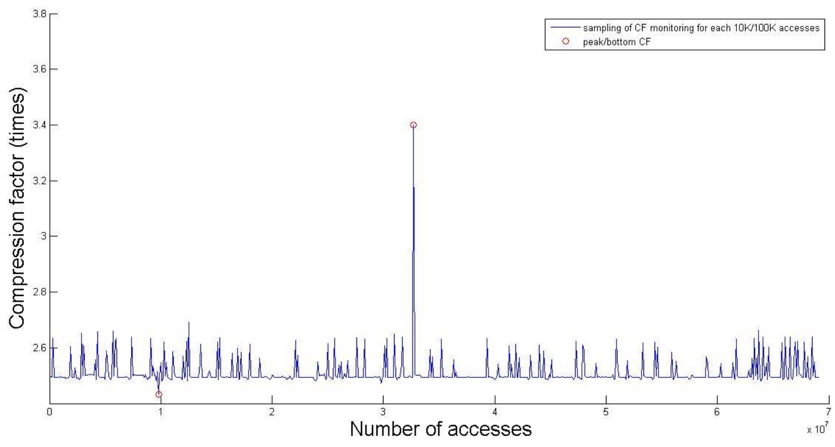
(e) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-1024-mfv



(f) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-1024-mfv



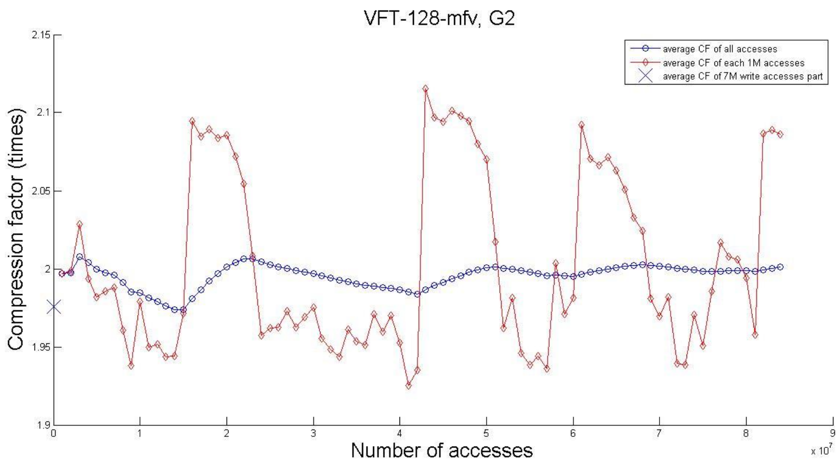
(g) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-16384-mfv



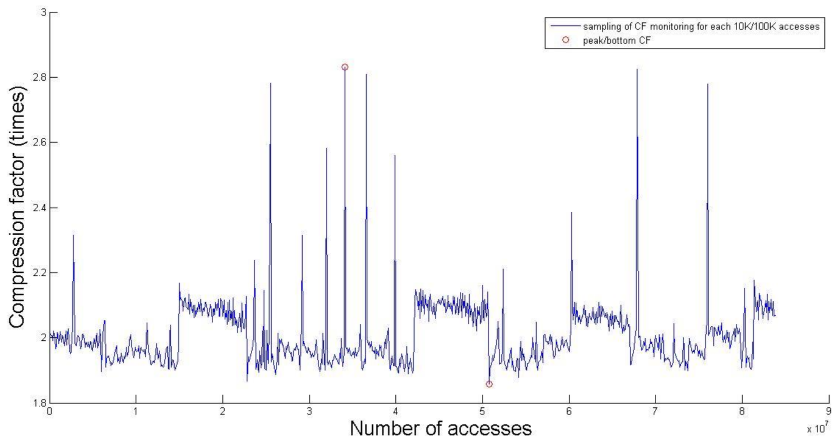
(h) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-16384-mfv

Figure 4.6: *Compression based on accesses for xalan phase1.*

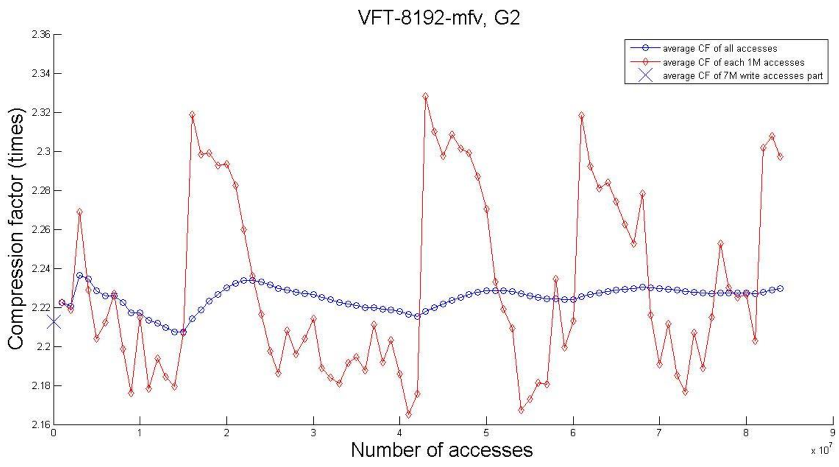
Figure 4.6 illustrates the compressibility changes for four different VFT size and 2 granularities for *xalan* phase1. Figure 4.6(g)(h) express the compressibility changes by time for G4-VFT-16K-mfv. In Figure 4.6(g), the ‘X’ mark on the Y-axis shows the $CF_{average}$ of the first 7M write accesses interval. As we use the same codewords to compress the rest part of the phase, the *global* $CF_{average}$ keeps stable which is about 0.1X lower than the X mark as the circle solid curve shows. The rolling diamond solid curve is the *local* $CF_{average}$ of every 1M accesses chunk. Figure 4.6(h) presents the sampling of $CF_{average}$ monitoring. The curve has many peaks in a rough range of $\pm$0.1X for the application *xalan* phase1 while there is a peak CF which is 35.7% higher than the $CF_{average}$ for the whole interval.



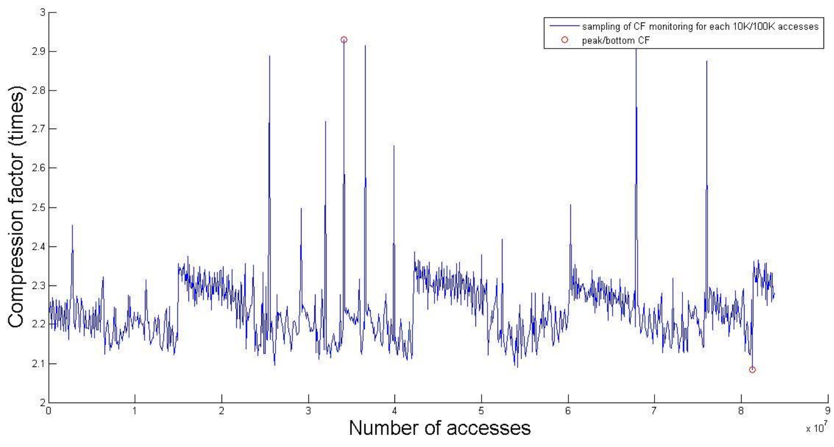
(a) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-128-mfv



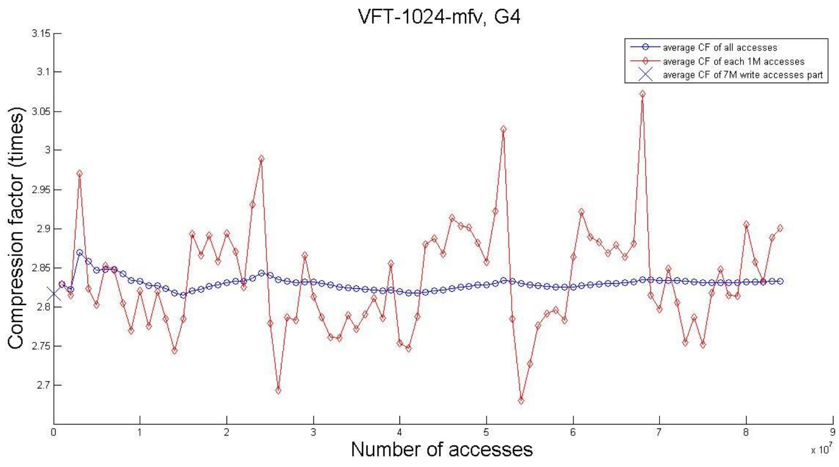
(b) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-128-mfv



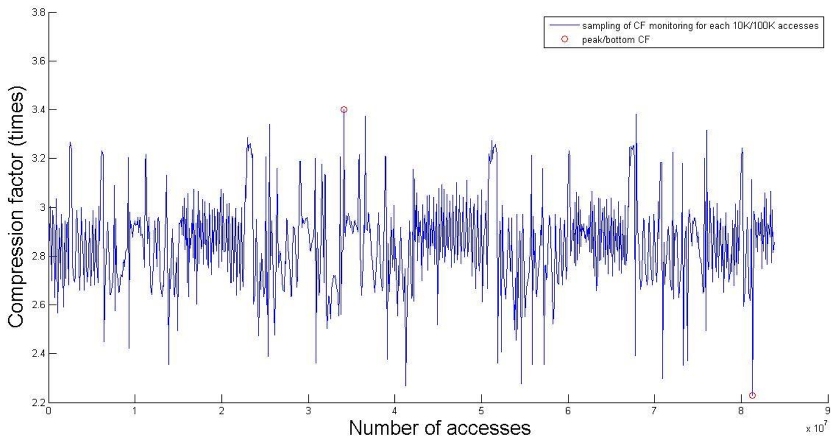
(c) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-8192-mfv



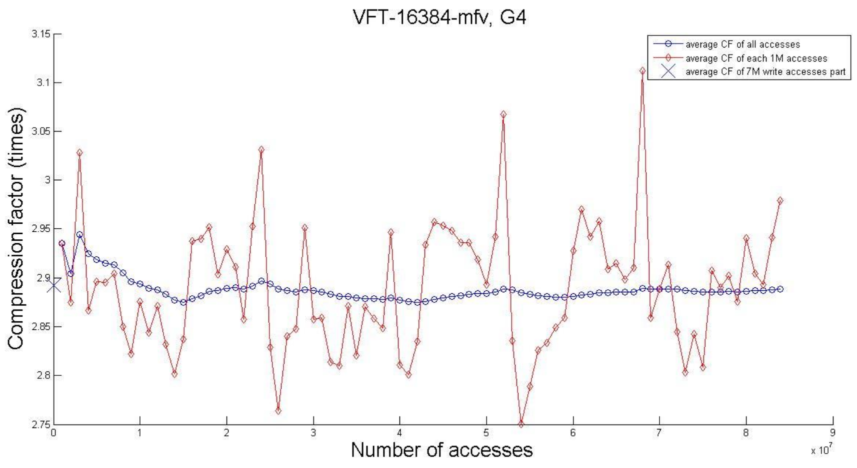
(d) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-8192-mfv



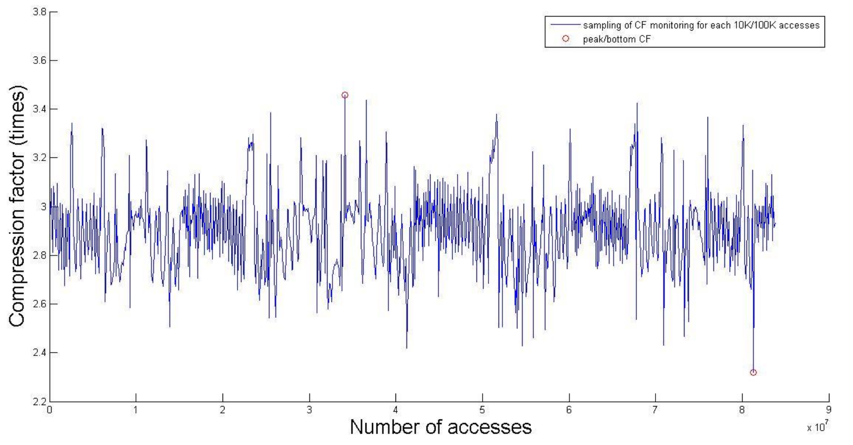
(e) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-1024-mfv



(f) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-1024-mfv



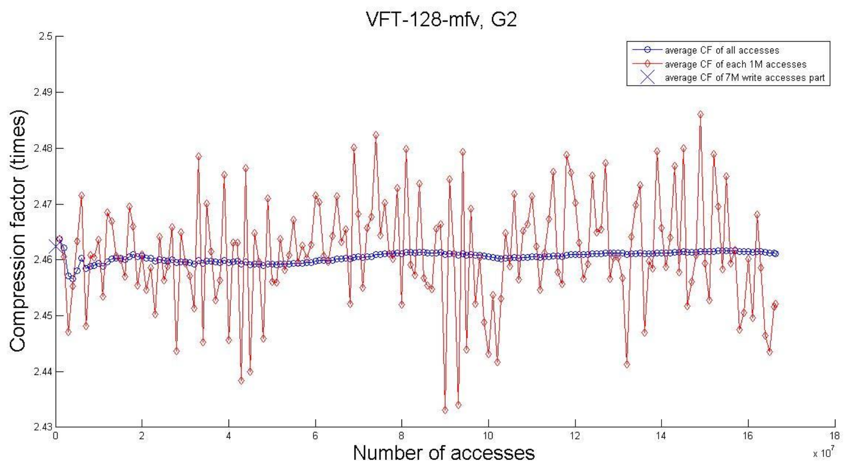
(g) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-16384-mfv



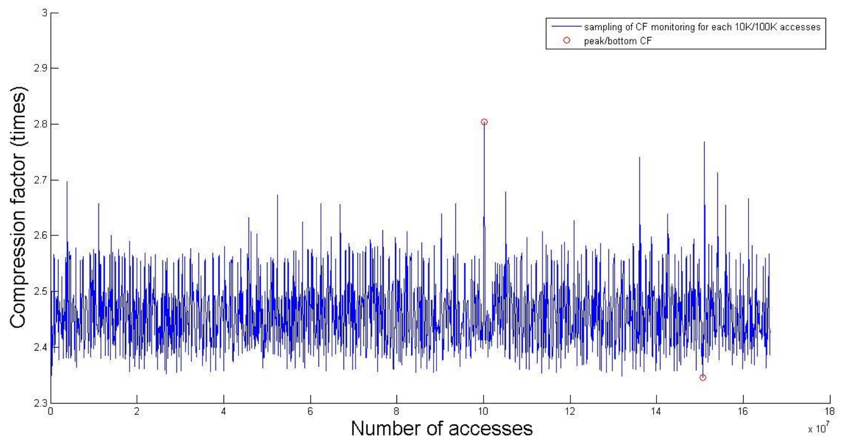
(h) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-16384-mfv

Figure 4.7: *Compression based on accesses for xalan phase2.*

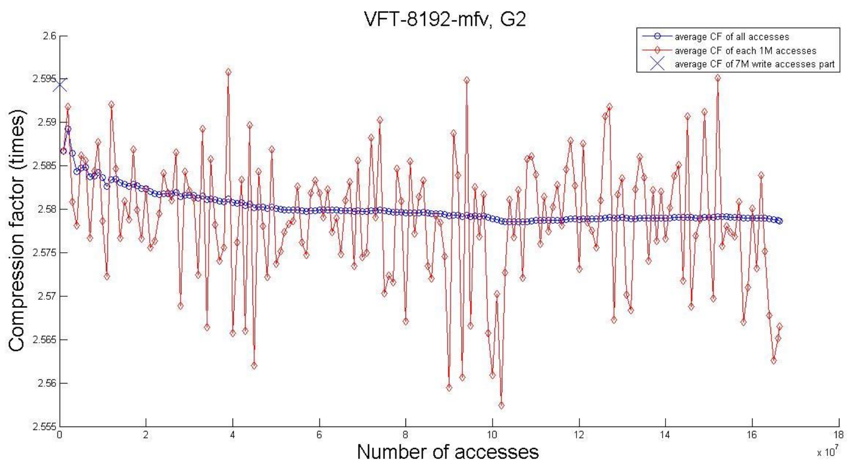
Xalan phase2 shows similar result as phase1. The difference of $CF_{average}$ between 2 intervals (7M write accesses/rest) is 0.0136X (0.55%) on average.



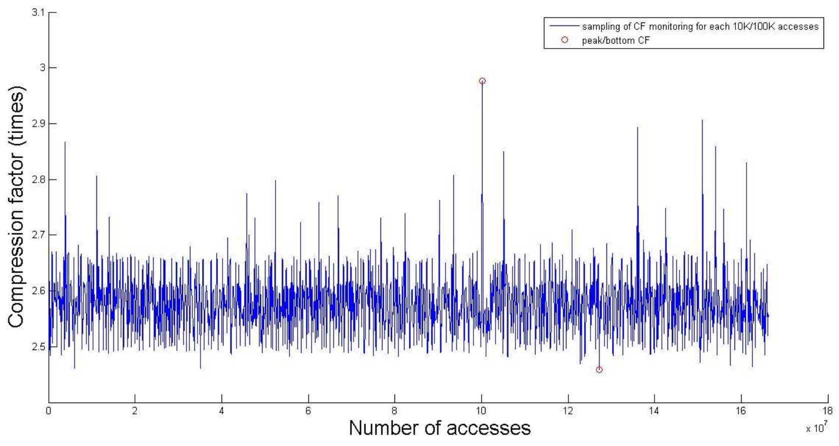
(a) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-128-mfv



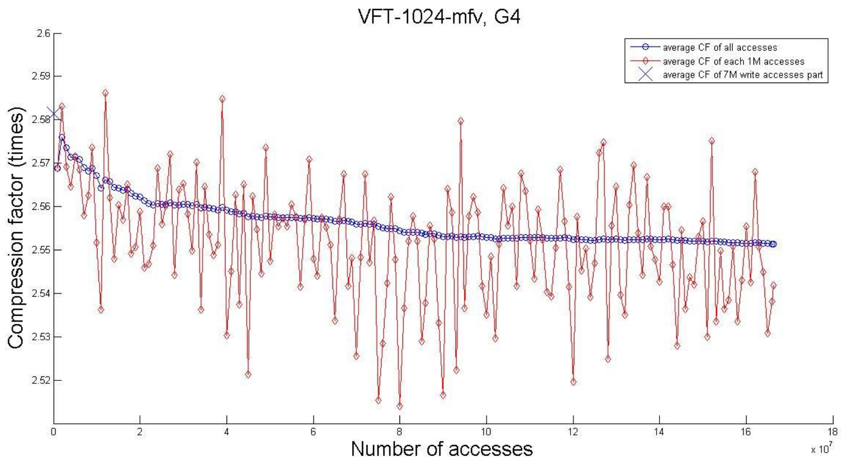
(b) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-128-mfv



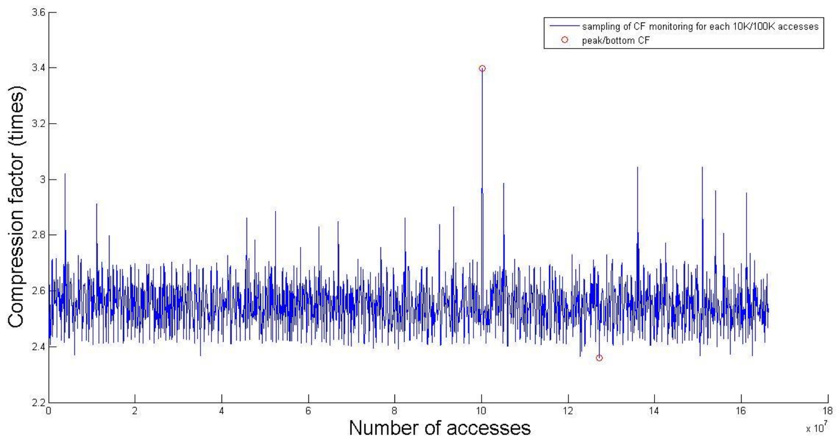
(c) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-8192-mfv



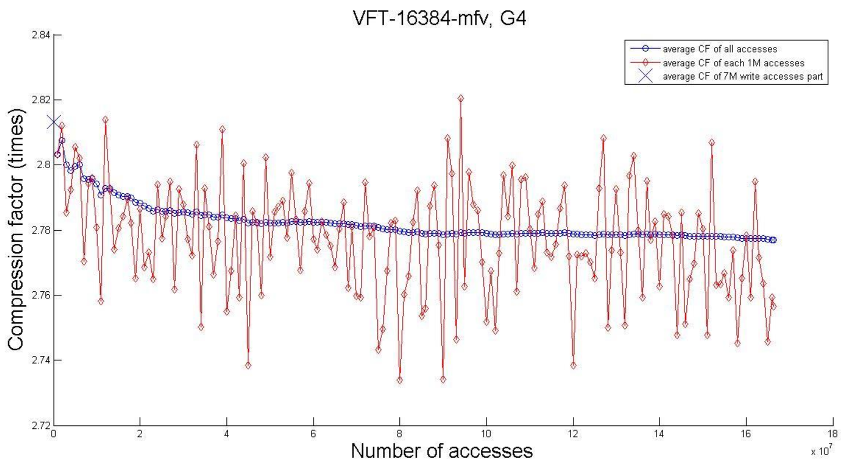
(d) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-8192-mfv



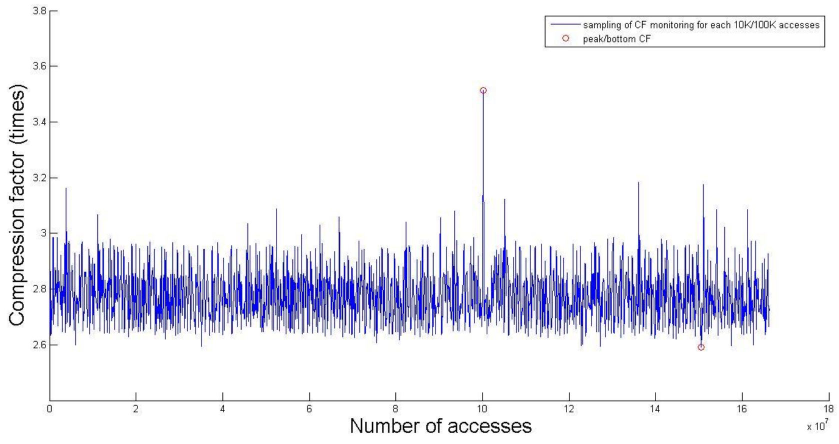
(e) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-1024-mfv



(f) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-1024-mfv

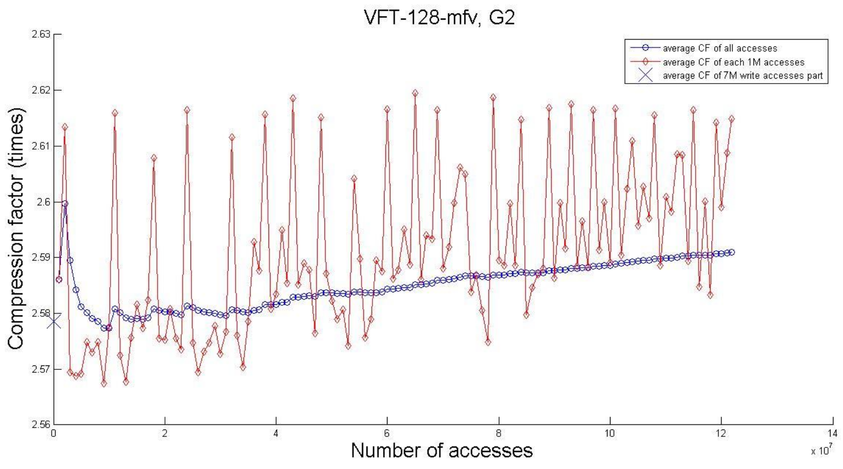


(g) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-16384-mfv

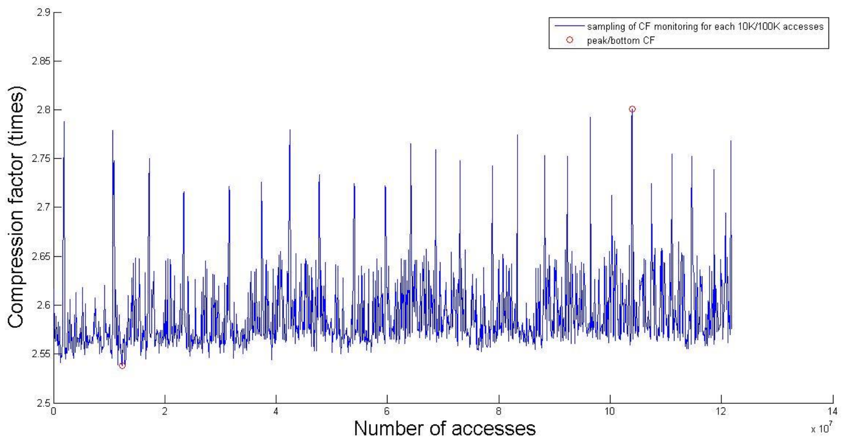


(h) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-16384-mfv

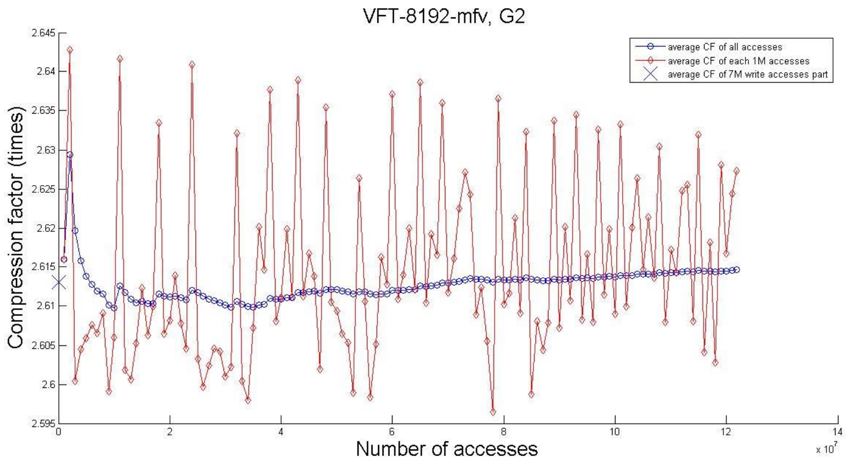
Figure 4.8 *Compression based on accesses for omnetpp phase 2.*



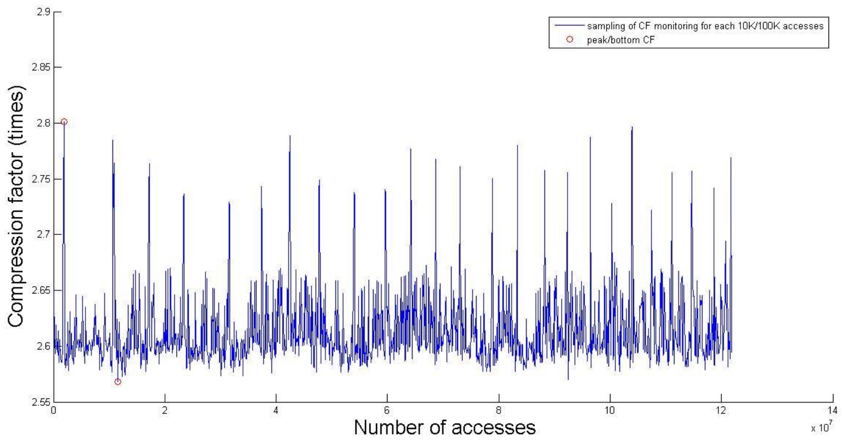
(a) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-128-mfv



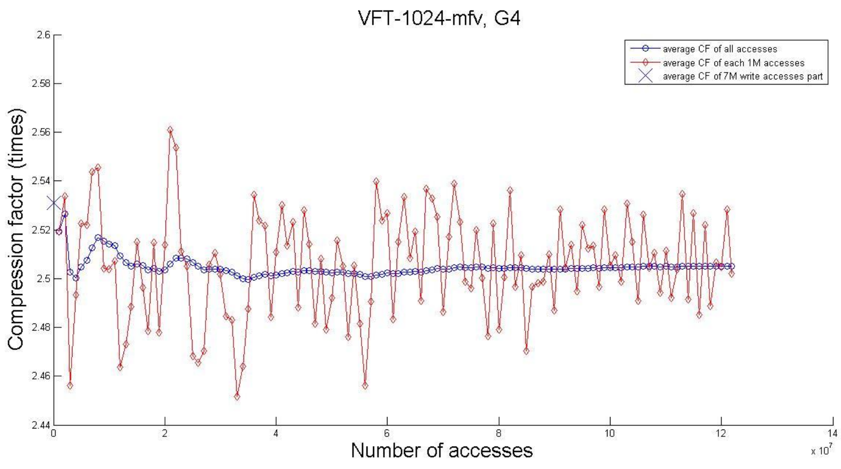
(b) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-128-mfv



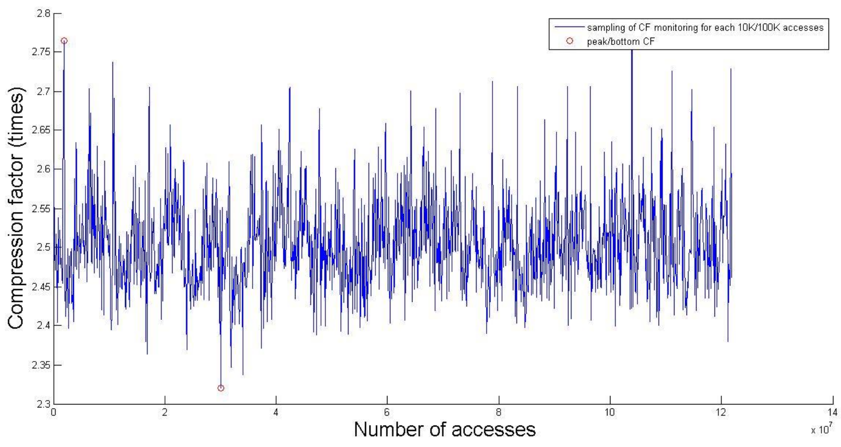
(c) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-8192-mfv



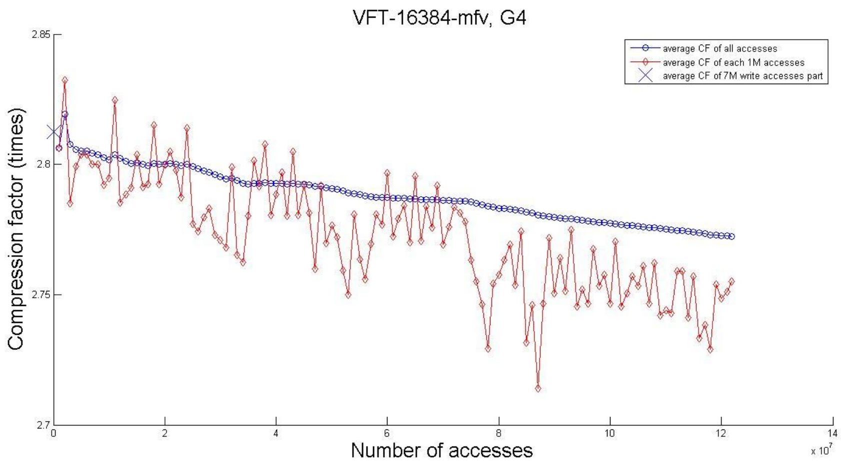
(d) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-8192-mfv



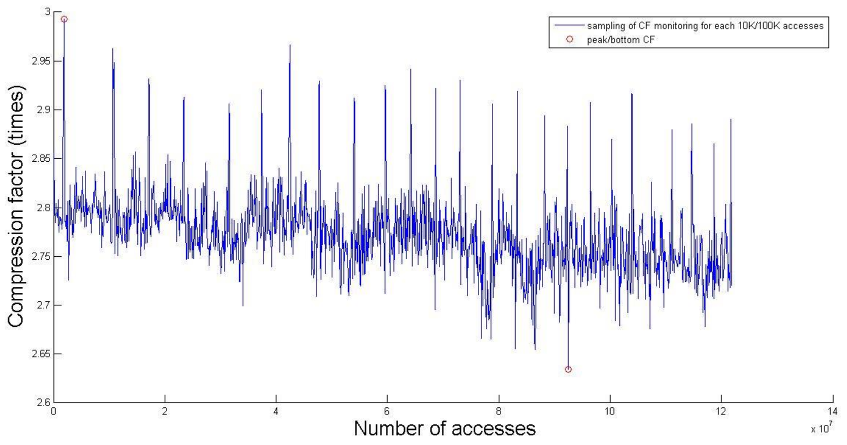
(e) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-1024-mfv



(f) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-1024-mfv



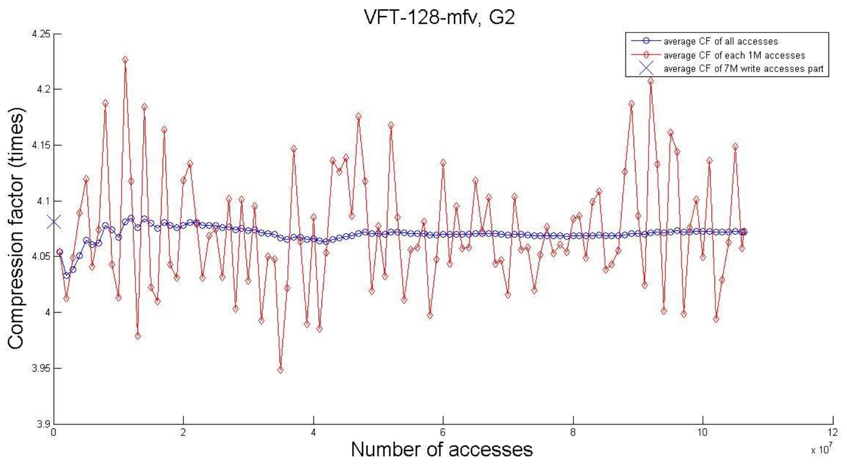
(g) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-16384-mfv



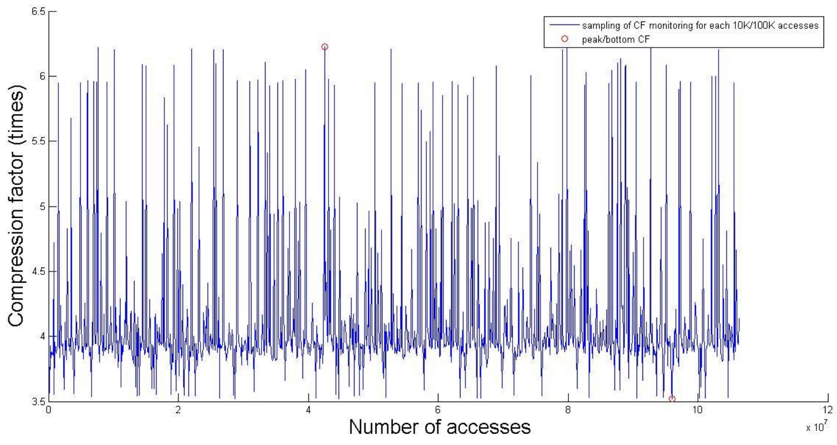
(h) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-16384-mfv

Figure 4.9: *Compression based on accesses for astar phase1.*

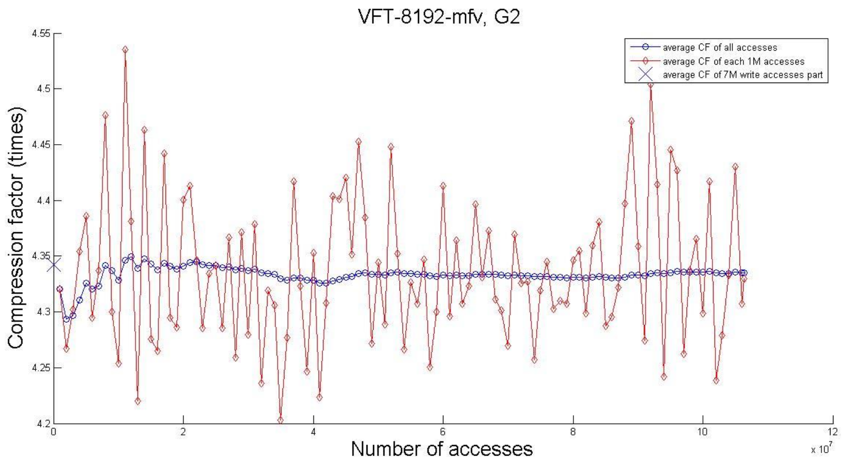
As shown in Figure 4.9(a), for astar-p1-g2-128, the $CF_{average}$ of the second interval (after 7M write accesses) of the phase is slightly higher than the $CF_{average}$ of the first interval by using the same Huffman coding from the first interval. That is because more values that are densely compressed with the established coding appear in the execution later.



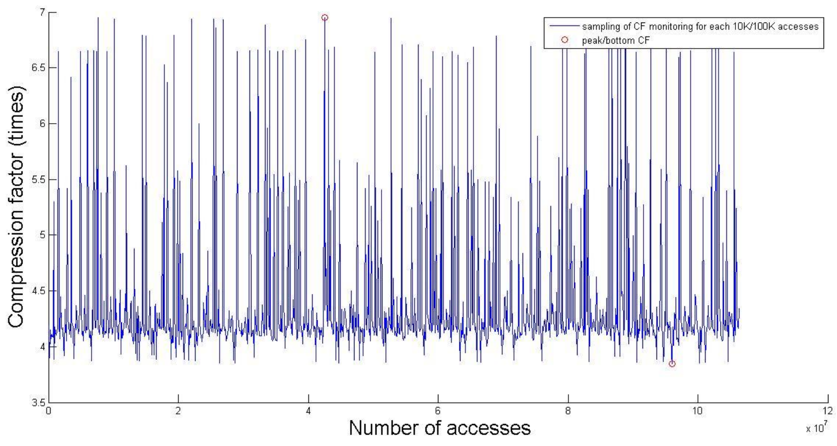
(a) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-128-mfv



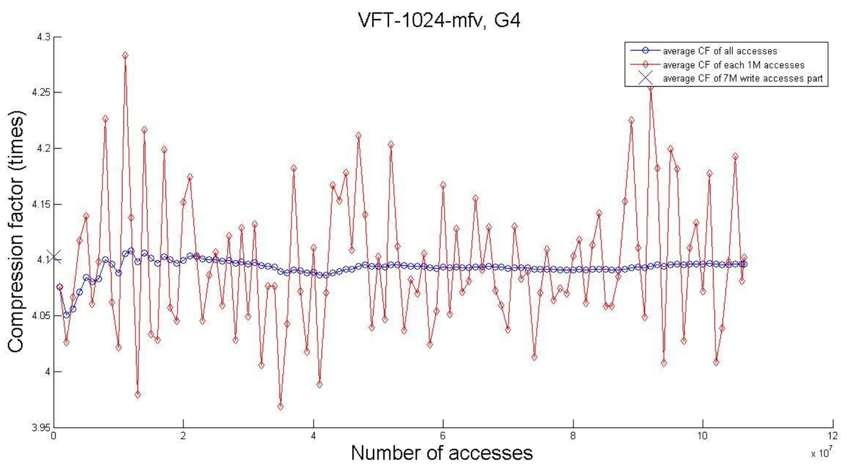
(b) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-128-mfv



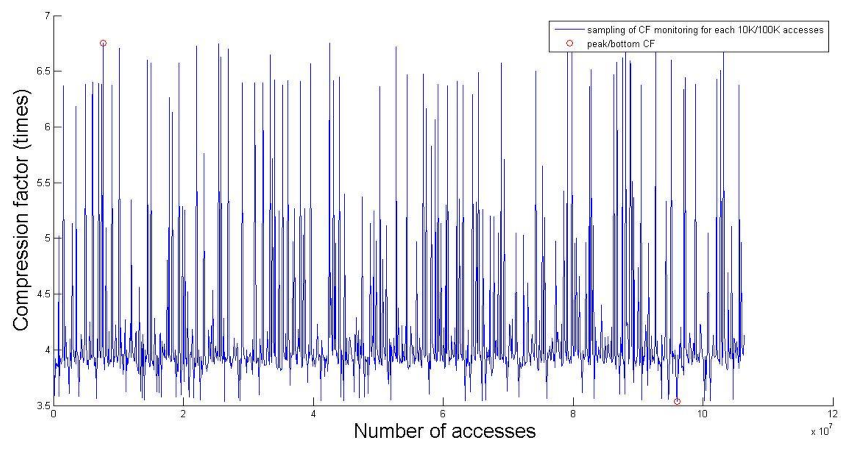
(c) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-8192-mfv



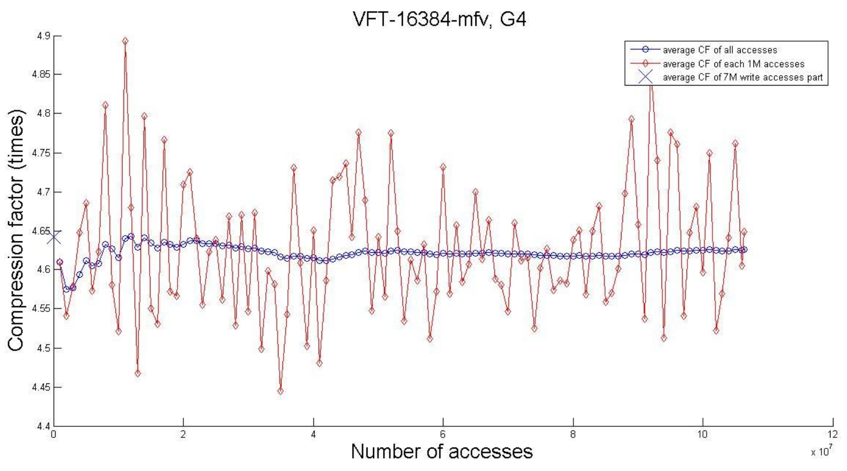
(d) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-8192-mfv



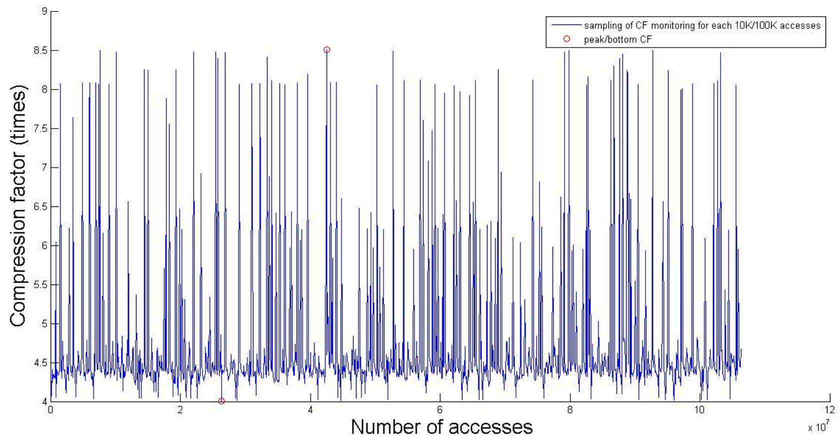
(e) Global and local CF$_{average}$ vs. time, granularity of 4 bytes, VFT-1024-mfv



(f) Sampling of CF$_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-1024-mfv



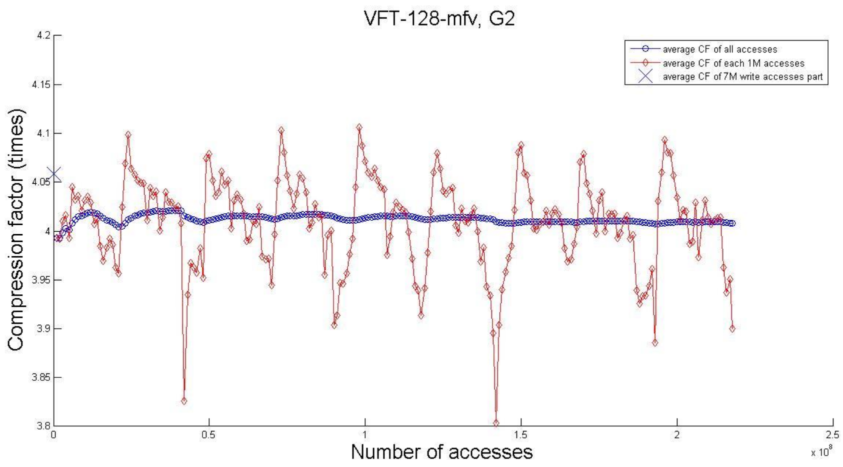
(g) Global and local CF$_{average}$ vs. time, granularity of 4 bytes, VFT-16384-mfv



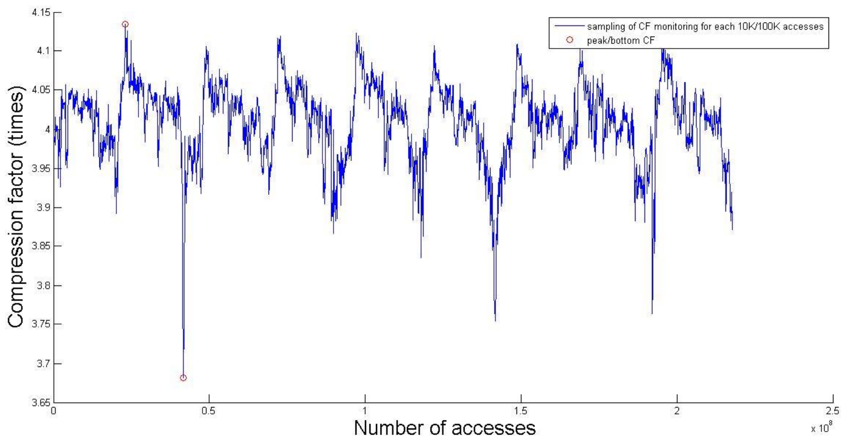
(h) Sampling of CF$_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-16384-mfv

Figure 4.10: *Compression based on accesses for astar phase 2.*

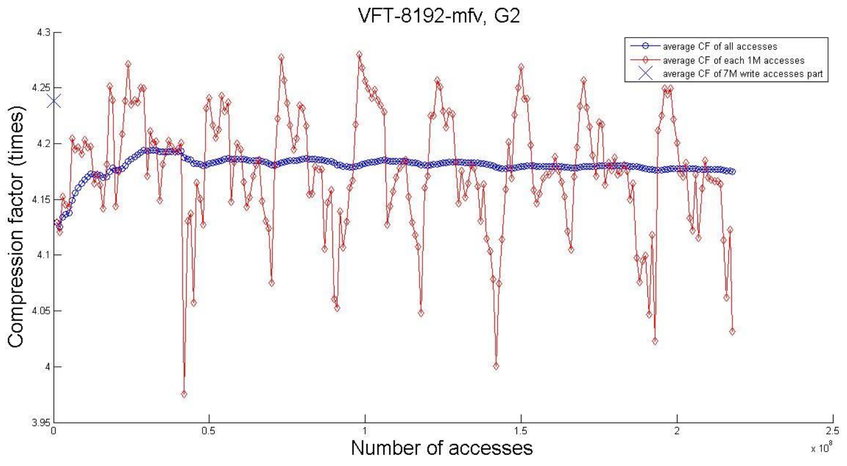
As shown in Figure 4.10, the applications with high compression potential such as *astar*-phase 2 introduce significant fluctuation of sampling of CF$_{average}$ monitoring. For example, as shown in Figure 4.10(h), the maximum value of CF$_{average}$ when sampling the compressibility monitoring is 8.5X which is 83.12% higher than CF$_{average}$ of the whole interval. Moreover, the CF$_{with\ VFT}$ obtained from previous experiment in Section 4.3 which can be considered as ideal compression result is 3.1187X (67.42%) higher than CF$_{average}$ in this case. It implies that Huffman coding in this case is inefficient. Hence, the Huffman codewords for *astar* phase 2 should be regenerated.



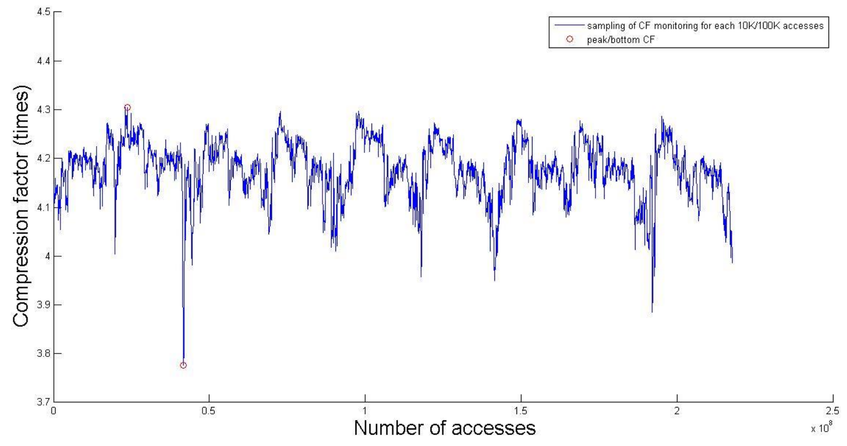
(a) Global and local CF$_{average}$ vs. time, granularity of 2 bytes, VFT-128-mfv



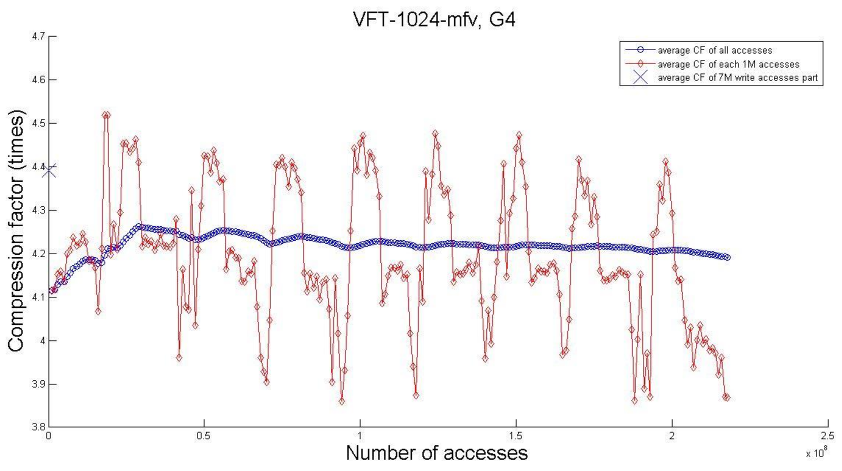
(b) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-128-mfv



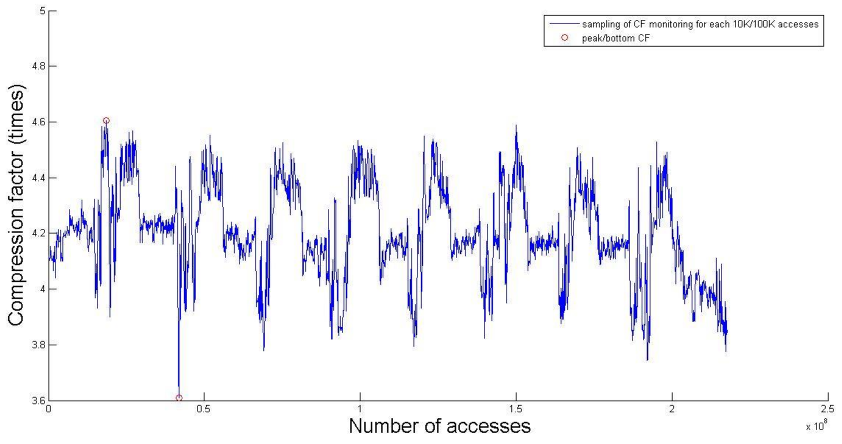
(c) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-8192-mfv



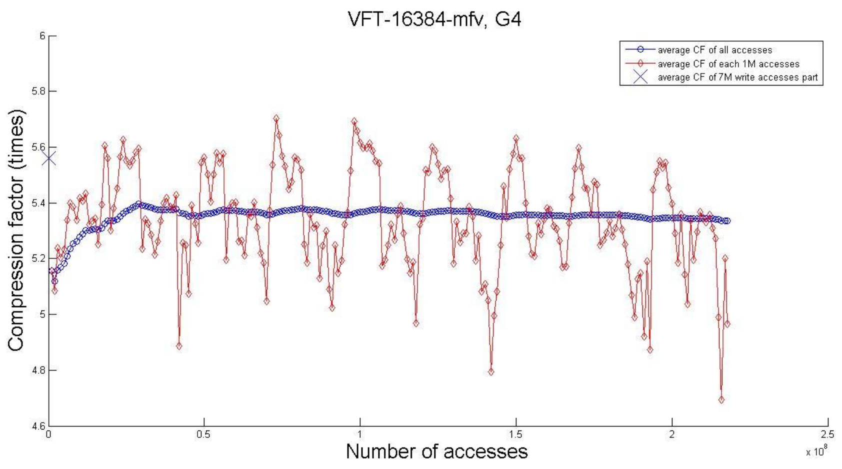
(d) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-8192-mfv



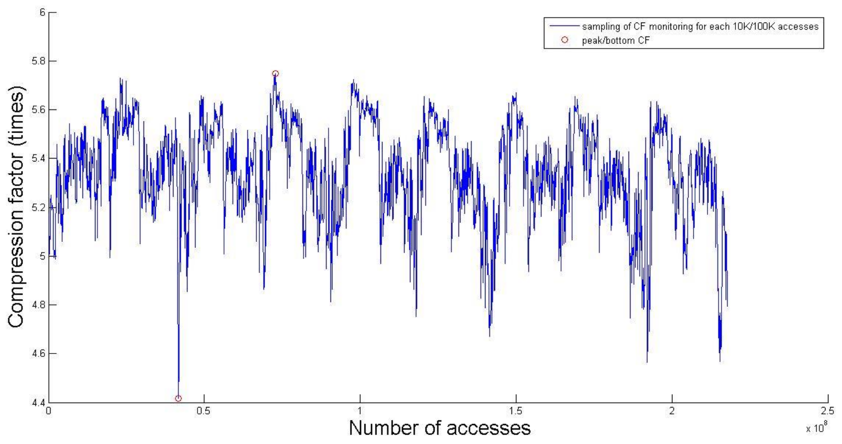
(e) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-1024-mfv



(f) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-1024-mfv

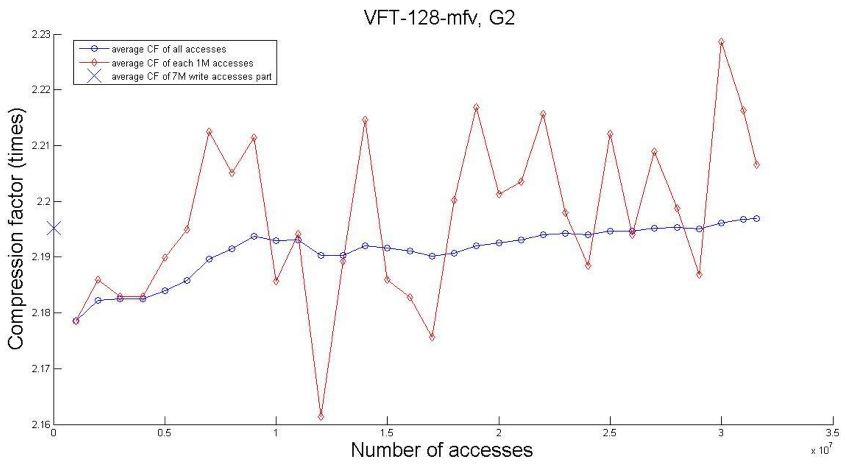


(g) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-16384-mfv

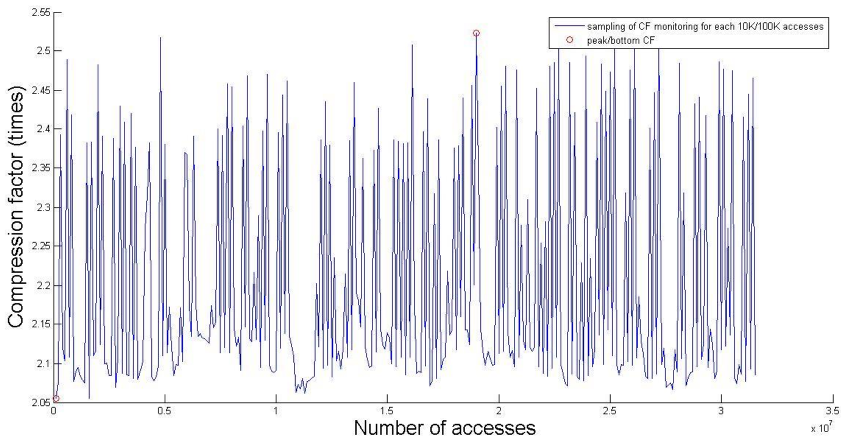


(h) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-16384-mfv

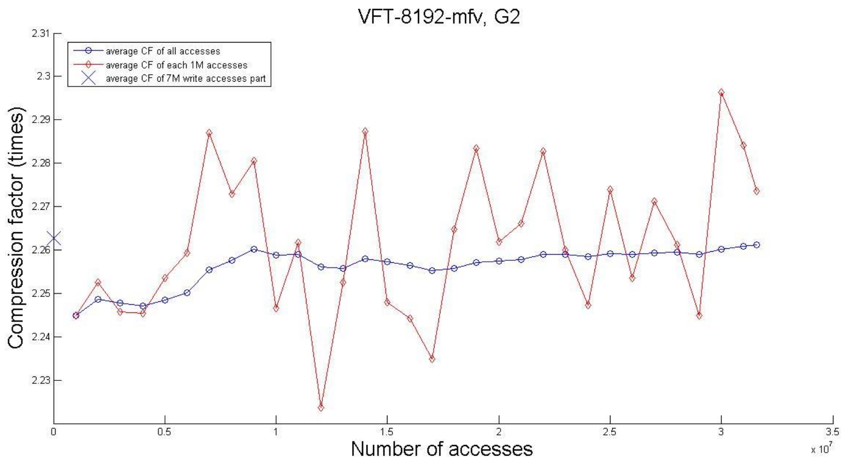
Figure 4.11: *Compression based on accesses for mcf phase1.*



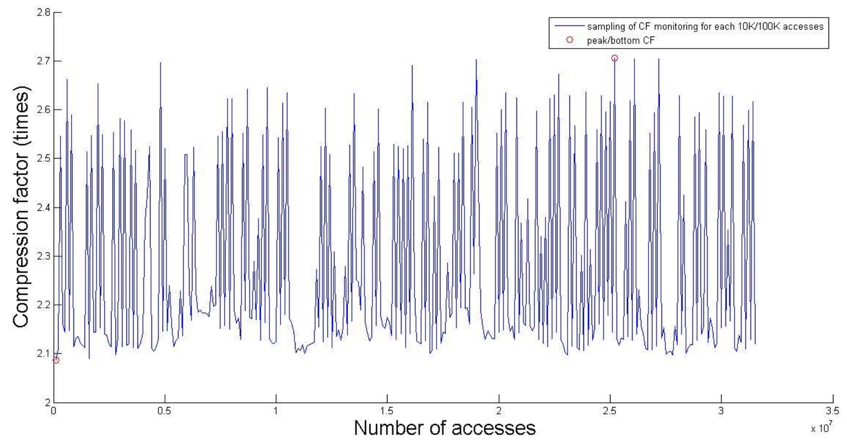
(a) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-128-mfv



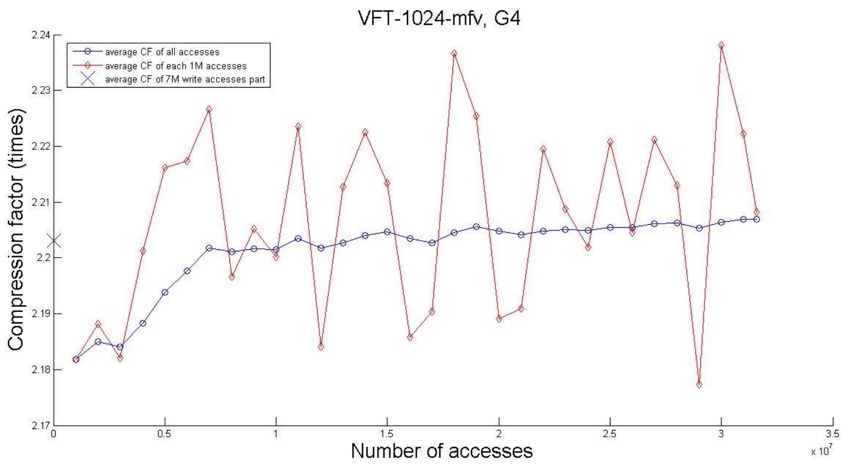
(b) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-128-mfv



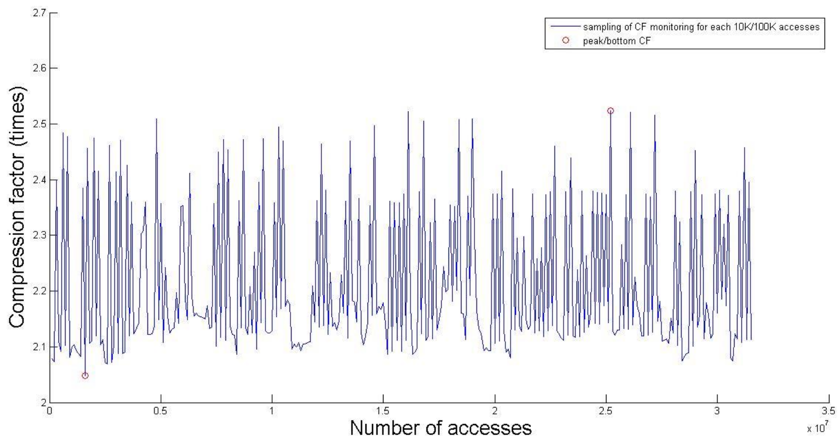
(c) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-8192-mfv



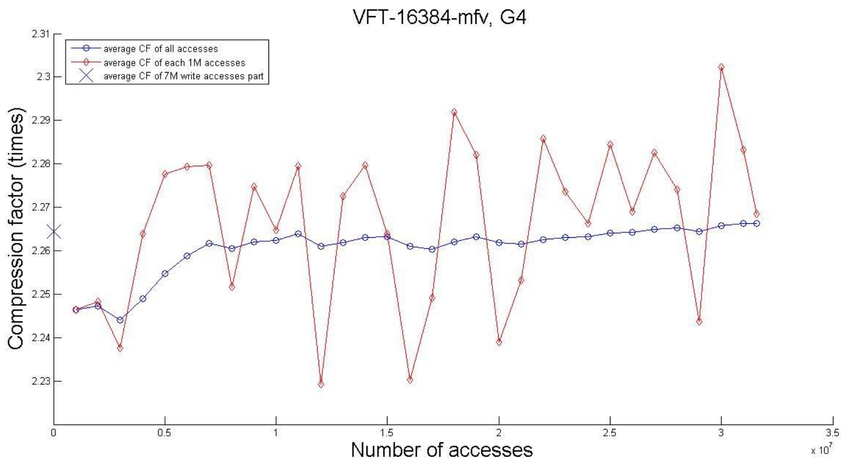
(d) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-8192-mfv



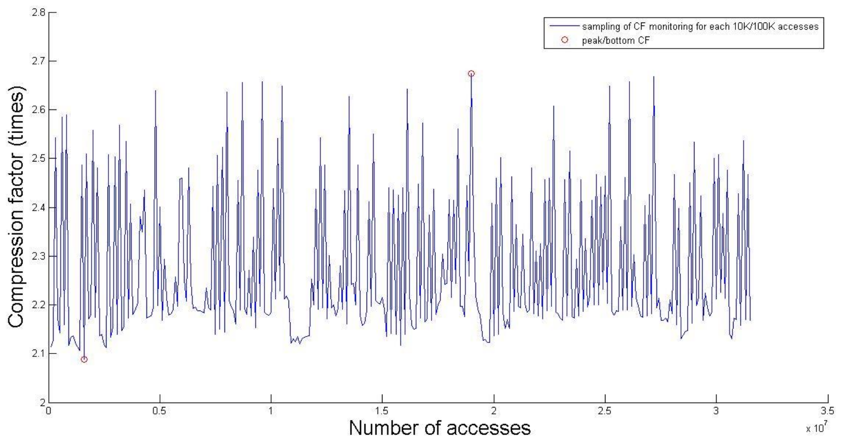
(e) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-1024-mfv



(f) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-1024-mfv

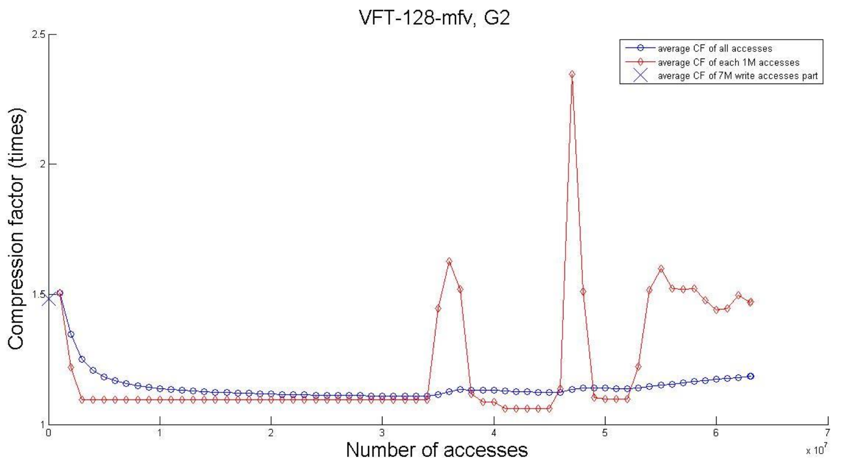


(g) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-16384-mfv

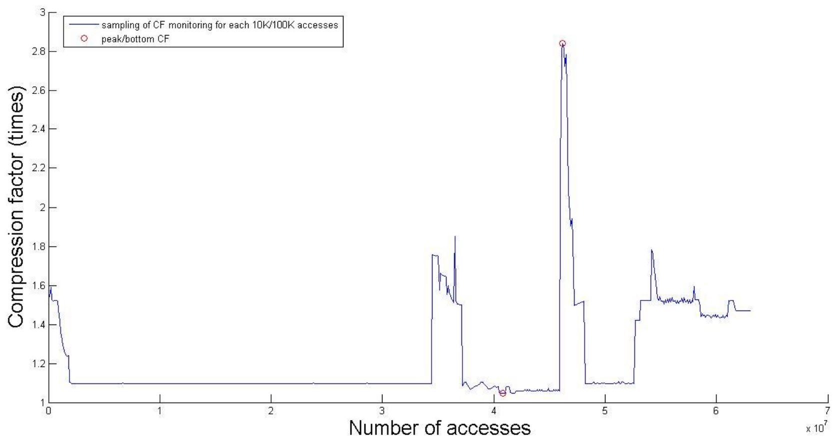


(h) Sampling of CF$_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-16384-mfv

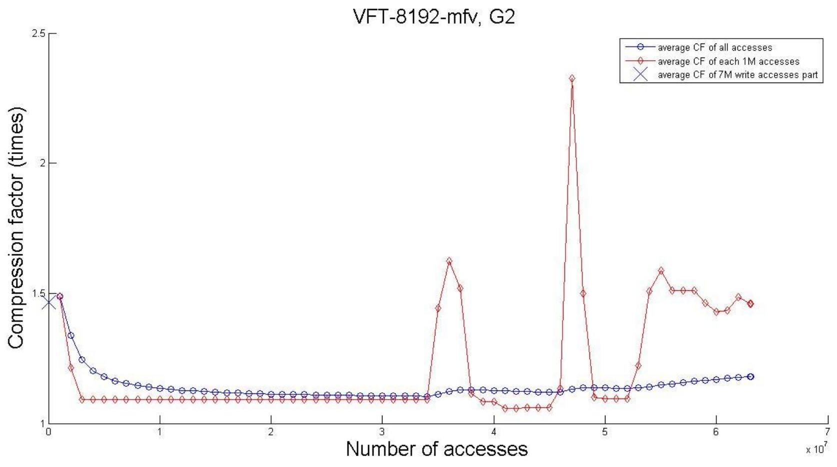
Figure 4.12: *Compression based on accesses for mcf phase2.*



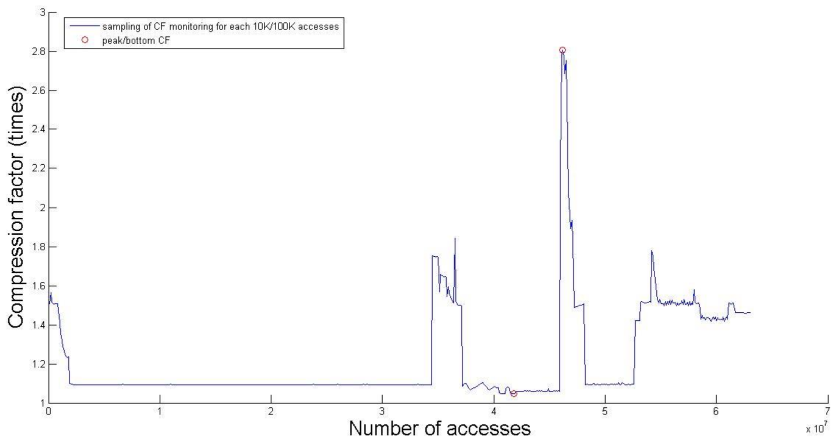
(a) Global and local CF$_{average}$ vs. time, granularity of 2 bytes, VFT-128-mfv



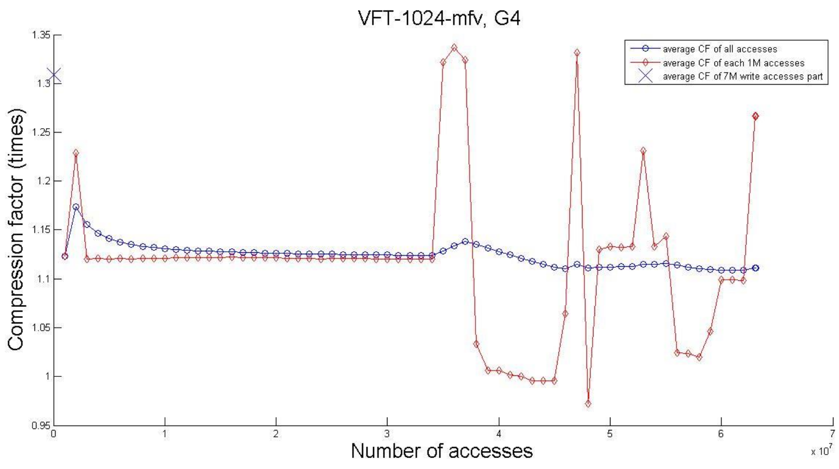
(b) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-128-mfv



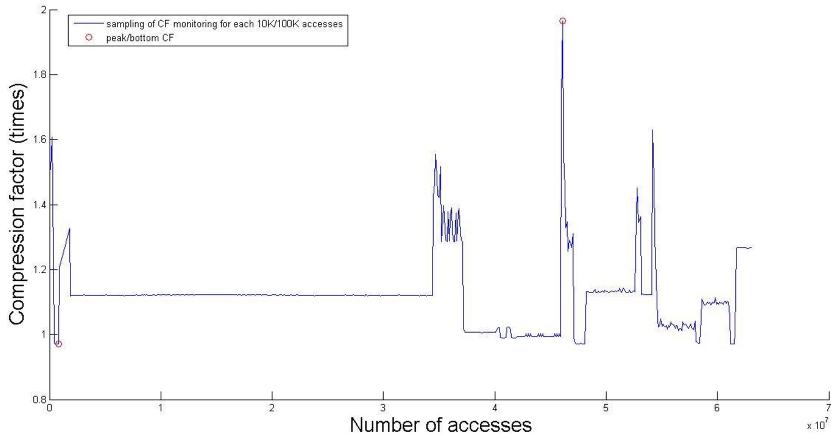
(c) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-8192-mfv



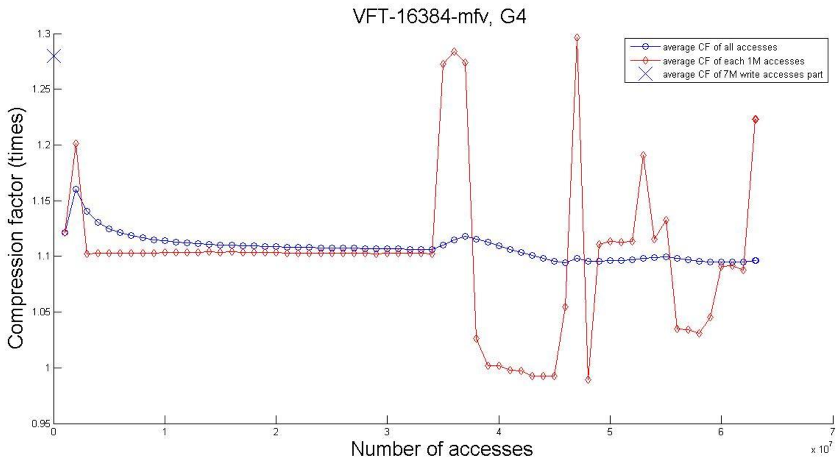
(d) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-8192-mfv



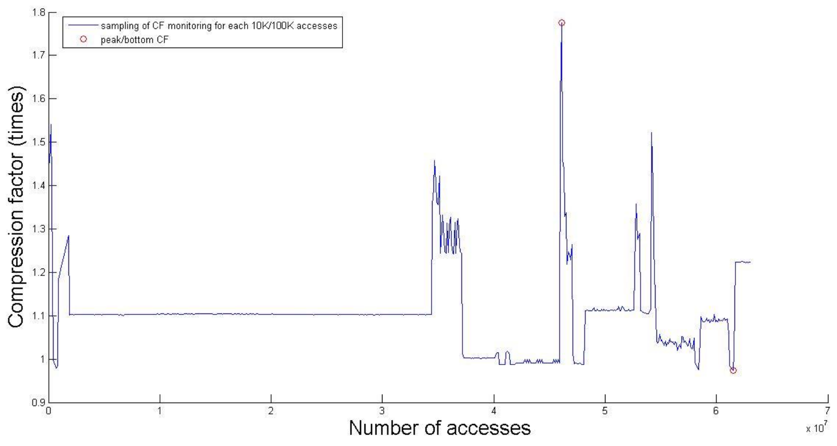
(e) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-1024-mfv



(f) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-1024-mfv

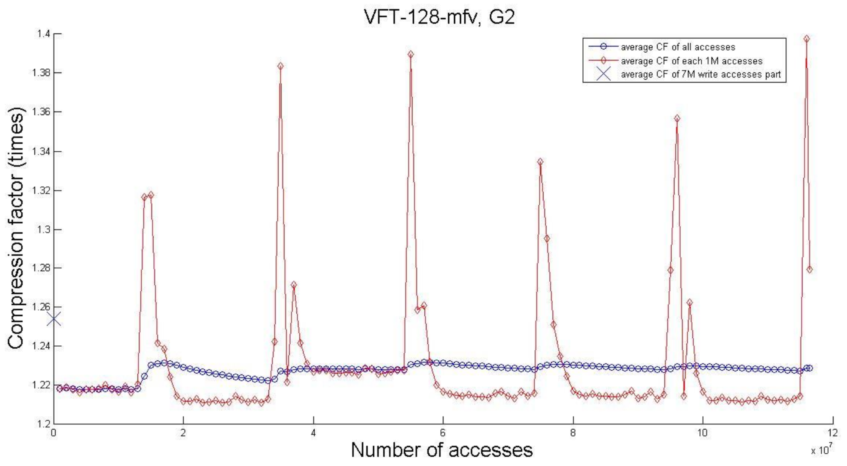


(g) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-16384-mfv

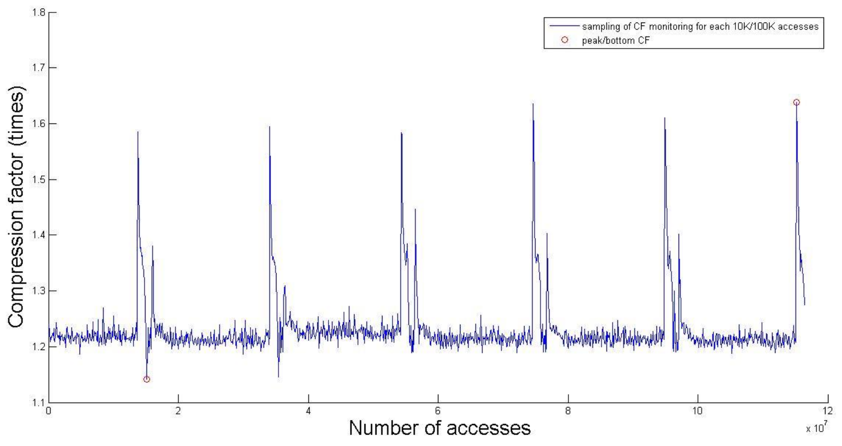


(h) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-16384-mfv

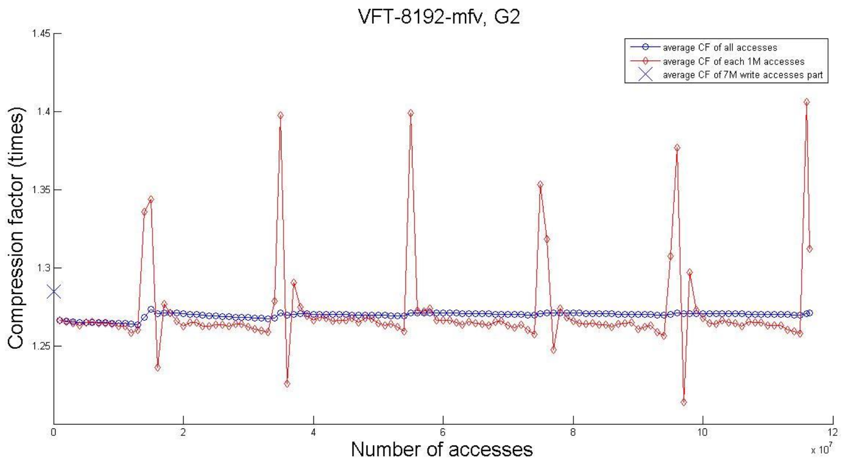
Figure 4.13: *Compression based on accesses for bzip2 phase1*.



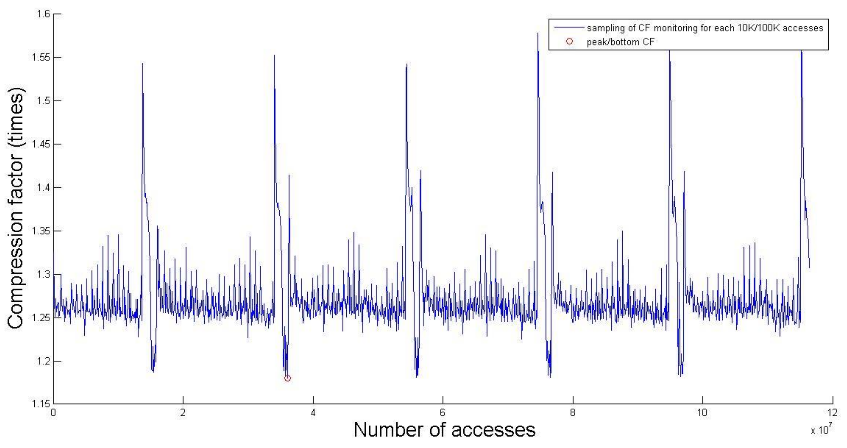
(a) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-128-mfv



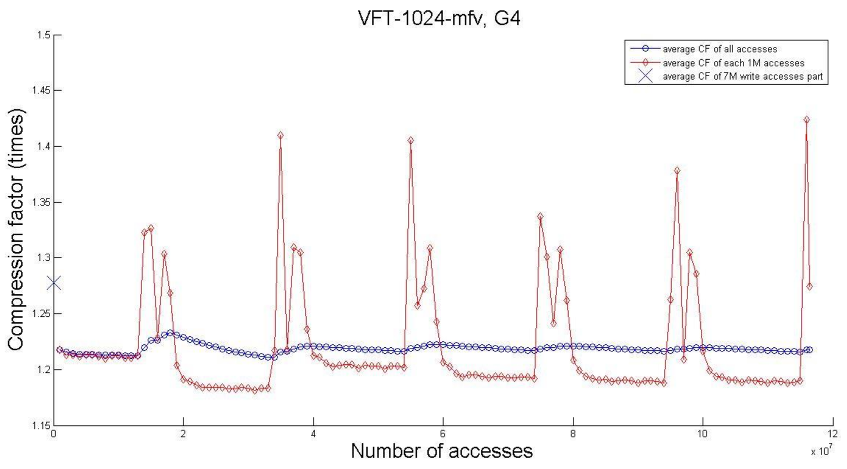
(b) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-128-mfv



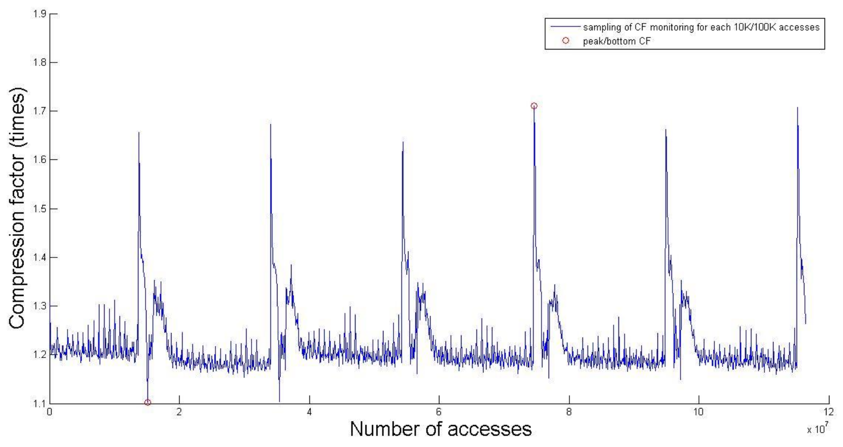
(c) Global and local $CF_{average}$ vs. time, granularity of 2 bytes, VFT-8192-mfv



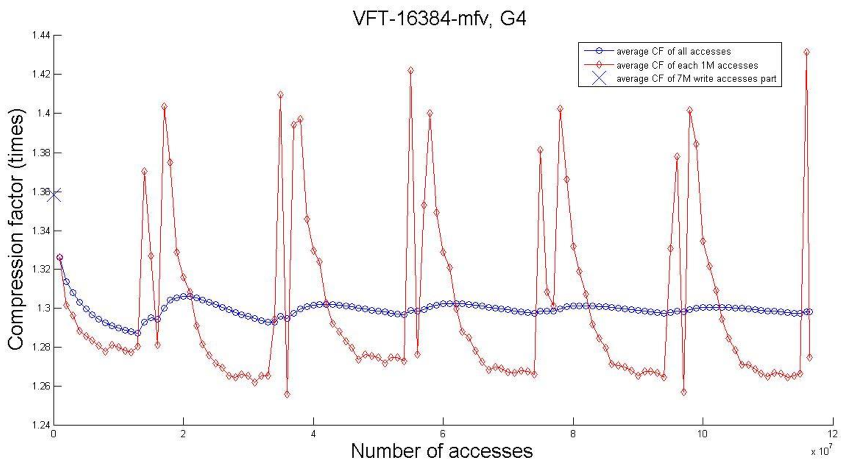
(d) Sampling of $CF_{average}$ monitoring vs. time, granularity of 2 bytes, VFT-8192-mfv



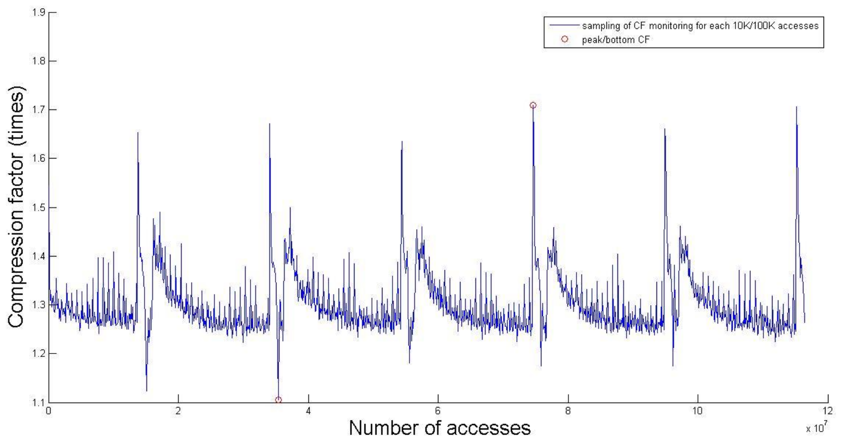
(e) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-1024-mfv



(f) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-1024-mfv



(g) Global and local $CF_{average}$ vs. time, granularity of 4 bytes, VFT-16384-mfv



(h) Sampling of $CF_{average}$ monitoring vs. time, granularity of 4 bytes, VFT-16384-mfv

Figure 4.14: *Compression based on accesses for bzip2 phase2.*

For all the applications, the sampling of $CF_{average}$ monitoring changes mostly less than 0.6X ($\pm 0.3X$ around the total $CF_{average}$). Thus, it is no need to update the Huffman tree frequently. In conclusion, the static Huffman tree works sufficiently without update in most phases. This implies that the data distribution varies little during the execution of applications which is good news for Huffman coding compression. Besides, the deviation between global and local $CF_{average}$ can be used to define the threshold of regenerating the Huffman code.

Moreover, we demonstrate that the information including 7M write accesses is enough for establishing an efficient Huffman coding. This conclusion is based on the result of the comparison between the $CF_{average}$ from this section and $CF_{with\ VFT}$ from Section 4.3. The $CF_{with\ VFT}$ is the ideal compression result based on the optimal codewords for compressing the footprint. Thus it can be used as the compression performance reference for comparing. The comparison result shows that the $CF_{average}$ is a little lower than $CF_{with\ VFT}$ in most cases. The compression performance losses on average is 13.2 %(i.e.,$\frac{\sum CF_{with\,VFT}}{\sum CF_{average}} - 1$). However, it must be noted here the Huffman codewords for *astar* phase2 are inefficient that introduce 67.42% compressibility losses. Hence, the Huffman codewords for *astar* phase2 should be regenerated. In conclusion, the Huffman codewords according to 7M write accesses are efficient in most phases. However, the significant compressibility losses may happen in particular cases such as *astar* phase 2. Thus, it is necessary for monitoring the *local* compressibility during the execution.

4.5 Fixed-size Compression

In previous experiments, the memory pages can be compressed to any possible sizes. This way of compression may increase the complexity of the memory management as the space freed up by compression has variable size which is hard to be used for storing other information efficiently. That is, as the sizes of compressed pages keeps changing during the execution, the size of freed up space (fragmentation) also changes. If the fragmented space is fully exploited by storing other compressed pages, storage problems may be introduced once the size of a compressed page grows. To deal with the fragmentation problem, fixed-size page compression can be used instead.

In this section, we define five fixed-size frames to fit compressed pages (uncompressed page = 4KB): 4KB, 2KB, 1KB, 512B and 256B corresponding to the fixed-CF of 1X, 2X, 4X, 8X and 16X respectively. A page can be only compressed to one of these five fixed-sizes. If the actual CF of one page is between two of the fixed-CF, it is compressed by the smallest factor. For example, if a page has a CF of 3.2X, it should be compressed by the factor of 2X but not 4X by using this scheme. This way, the fragmentation in memory due to compression has fixed size and can be handled easier and more efficiently. In Section 4.4, we calculate the compressed page size for each access and count all the compressed size for the $CF_{average}$. In this section, we do the similar experiment but using the fixed-CF for compressing each accessed page. The experimental results for 5 applications are presented in Table 5.1.

Table 4.5 *Comparison between $CF_{average}$ (from Section 4.4) and $CF_{fixed\text{-}size}$ by applying fixed-size compression, the CF distribution of pages for fixed-size compression.*

Benchmark-phase-granularity-VFT size	**$CF_{average}$**	**$CF_{fixed\text{-}size}$**	**Total accesses**	**Number of pages (percentage) compressed with Fixed-CF of 1X, 2X, 4X, 8X, 16X**
[4]omnetpp-p2-g2-128	2.4612	1.8466	166313233	13816001(8.3072%), 152496395(91.6923%), 837(0.0005%), 0(0%), 0(0%)
omnetpp-p2-g2-8192	2.5787	1.8466		13816002(8.3072%), 152494677(91.6912%), 2554(0.0015%), 0(0%), 0(0%)
omnetpp-p2-g4-1024	2.5514	1.8894		13816001(8.3072%), 144332864(86.7838%), 8164368(4.9090%), 0(0%), 0(0%)

[4] As omnetpp-phase 1 includes few memory accesses, we exclude it from this experiment.

omnetpp-p2-g4-16384	2.7770	1.9056		13816001(8.3072%), 141336088(84.9819%), 11161144(6.7109%), 0(0%), 0(0%)
xalan-p1-g2-128	2.3832	1.9925	69169855	462720(0.669%), 68339695(98.7998%), 294857(0.4263%), 72583(0.1049%), 0(0%)
xalan-p1-g2-8192	2.3232	1.9941		429997(0.6217%), 68335736(98.7941%), 330999(0.4785%), 73123(0.1057%), 0(0%)
xalan-p1-g4-1024	2.1564	2.0007		314930(0.4553%), 68302763(98.7464%), 347481(0.5024%), 144638(0.2091%), 60043(0.0868%)
xalan-p1-g4-16384	2.5060	2.0033		246541(0.3564%), 68327328(98.7819%), 390818(0.565%), 144936(0.2095%), 60232(0.0871%)
xalan-p2-g2-128	2.0012	1.1716	83890314	59372345(70.7738%), 24393950(29.0784%), 124019(0.1478%), 0(0%), 0(0%)
xalan-p2-g2-8192	2.2296	1.6557		17522752(20.8877%), 66212043(78.9269%), 155519(0.1854%), 0(0%), 0(0%)
xalan-p2-g4-1024	2.8329	2.0027		372881(0.4445%), 82611237(98.4753%), 782177(0.9324%), 124019(0.1478%), 0(0%)
xalan-p2-g4-16384	2.8883	2.0026		374759(0.4467%), 82609253(98.4729%), 782283(0.9325%), 124019(0.1478%), 0(0%)
mcf-p1-g2-128	4.0079	3.2539	217752746	0(0%), 49922558(22.9263%), 167830188(77.0737%), 0(0%), 0(0%)
mcf-p1-g2-8192	4.1751	3.2558		0(0%), 49766998(22.8548%), 167985748(77.1452%), 0(0%), 0(0%)
mcf-p1-g4-1024	4.1911	3.2426		0(0%), 50855038(23.3545%), 166897708(76.6455%), 0(0%), 0(0%)
mcf -p1-g4-16384	5.3348	3.2559		20(0%), 49764213(22.8535%), 167988513(77.1464%), 0(0%), 0(0%)
mcf -p2-g2-128	2.1970	1.5412	31570405	9424651(29.8528%), 22106206(70.0219%), 4781(0.0151%), 34767(0.1101%), 0(0%)
mcf	2.2611	1.5413		9424651(29.8528%), 22103761(70.0142%),

-p2-g2-8192				7226(0.0229%), 34767(0.1101%), 0(0%)
mcf -p2-g4-1024	2.2069	1.5401		9451217(29.9370%), 22084363(69.9527%), 58(0.0002%), 34767(0.1101%), 0(0%)
mcf -p2-g4-16384	2.2663	1.5410		9428439(29.8648%), 22106634(70.0233%), 565(0.0018%), 34767(0.1101%), 0(0%)
astar-p1-g2-128	2.5910	1.7003	121847816	21612698(17.7375%), 99972386(82.0469%), 262732(0.2156%), 0(0%), 0(0%)
astar-p1-g2-8192	2.6147	1.7061		21120968(17.3339%), 100464116(82.4505%), 262732(0.2156%), 0(0%), 0(0%)
astar-p1-g4-1024	2.5052	1.6946		22088350(18.1278%), 99496560(81.6564%), 262906(0.2158%), 0(0%), 0(0%)
astar-p1-g4-16384	2.7723	1.8490		10086764(8.2782%), 111494761(91.5033%), 266291(0.2185%), 0(0%), 0(0%)
astar-p2-g2-128	4.0723	2.3826	106378293	4082661(3.8379%), 66008821(62.0510%), 24201735(22.7506%), 12085076(11.3605%), 0(0%)
astar-p2-g2-8192	4.3352	2.4982		3748643(3.5239%), 64135624(60.2901%), 15631069(14.6939%), 22862957(21.4921%), 0(0%)
astar-p2-g4-1024	4.0962	2.4403		4134215(3.8863%), 65376177(61.4563%), 17287965(16.2514%), 19579936(18.4060%), 0(0%)
astar-p2-g4-16384	4.6255	3.7596		4096072(3.8505%), 7697260(7.2357%), 68221033(64.1306%), 26360142(24.7796%), 3786(0.0036%)
bzip2-p1-g2-128	1.1845	1.0131	63068975	61473322(97.47%), 1534017(2.4323%), 61636(0.0977%), 0(0%), 0(0%)
bzip2-p1-g2-8192	1.1807	1.0127		61485620(97.4895%), 1578330(2.5025%), 5025(0.00805), 0(0%), 0(0%)
bzip2-p1-g4-1024	1.1111	1.0053		62499385(99.0969%), 387344(0.6142%), 161434(0.2560%), 20809(0.0330%), 3(0%)

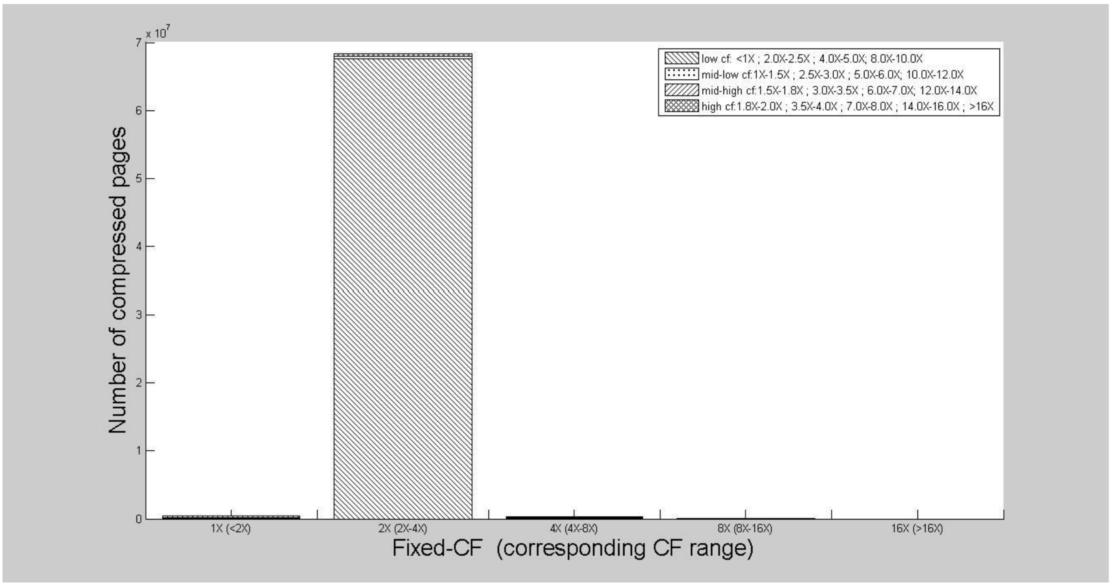
bzip2-p1-g4-16384	1.0966	1.0037		62640221(99.3202%), 363338(0.5761%), 60662(0.0962%), 4754(0.0075%), 0(0%)
bzip2-p2-g2-128	1.2290	1.0035	116452964	115671513(99.3290%), 717627(0.6162%), 63824(0.0548%), 0(0%), 0(0%)
bzip2-p2-g2-8192	1.2709	1.0026		115852560(99.4844%), 600404(0.5156%), 0(0%), 0(0%), 0(0%)
bzip2-p2-g4-1024	1.2176	1.0114		115852560(99.4844%), 600404(0.5156%), 0(0%), 0(0%), 0(0%)
bzip2-p2-g4-16384	1.2983	1.0113		115852560(99.4844%), 600404(0.5156%), 0(0%), 0(0%), 0(0%)
Average	2.5981	1.8772		

In conclusion, the fixed-size compression scheme results in a significant compressibility loss. The compressibility degradation is 38.4% from 2.5981X which is the $CF_{average}$ from Section 4.4 to 1.8872X, on average. The main reason for this degradation of compressibility is because more than 30% of the pages are actually compressed by less than a factor of 2X, thus they are eventually stored uncompressed. Unfortunately, a high potential of compressibility is traded off for easier placement of compressed pages using fixed size frames. In addition, only a negligible amount of pages (0.005%) can be compressed by the factor of 16X making this fixed-size CF option unnecessary to consider. Interestingly, increasing the VFT size or granularity can hardly improve the compressibility, in contrast to the findings from our previous experiments. We further explore this phenomenon by dividing each fixed-CF option into 4 regions to get more details about the distribution of compressibility. The detail information of CF regions is summarized in Table 4.6.

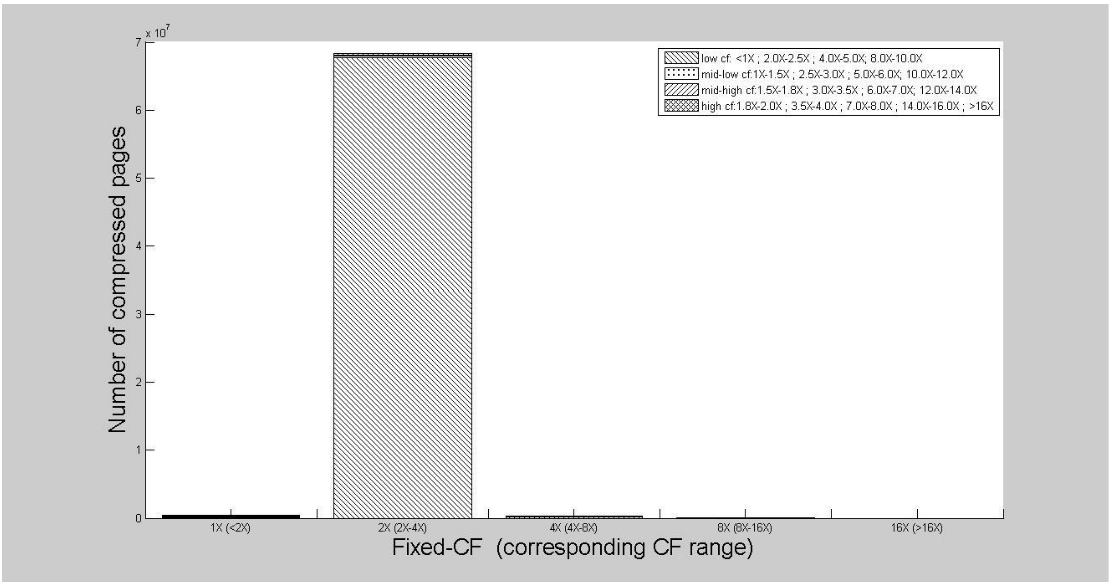
Table 4.6 *Information of regions for each fixed-CF.*

Fixed-CF	**Low**	**Mid-low**	**Mid-high**	**High**
1X	<1X	1.0X-1.5X	1.5X-1.8X	1.8X-2.0X
2X	2.0X-2.5X	2.5X-3X	3.0X-3.5X	3.5X-4.0X
4X	4.0X-5.0X	5.0X-6.0X	6.0X-7.0X	7.0X-8.0X
8X	8.0X-10X	10X-12X	12X-14X	14X-16X
16X	>16X			

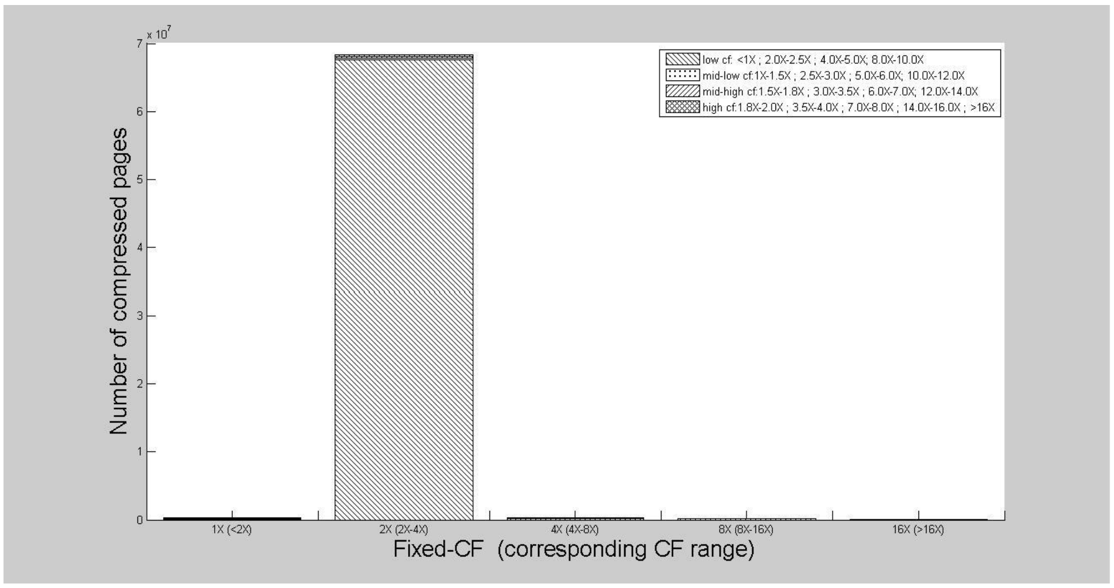
The variation of compressibility distribution due to the different granularities and VFT sizes are tested and the results are shown from Figure 4.15 to Figure 4.23.Moreover, we investigate the CF$_{\text{fixed-size}}$ changes if extra fixed-CF options of 1.5X and 3X are allowed. The results are shown from Table 4.7 to Table 4.15.



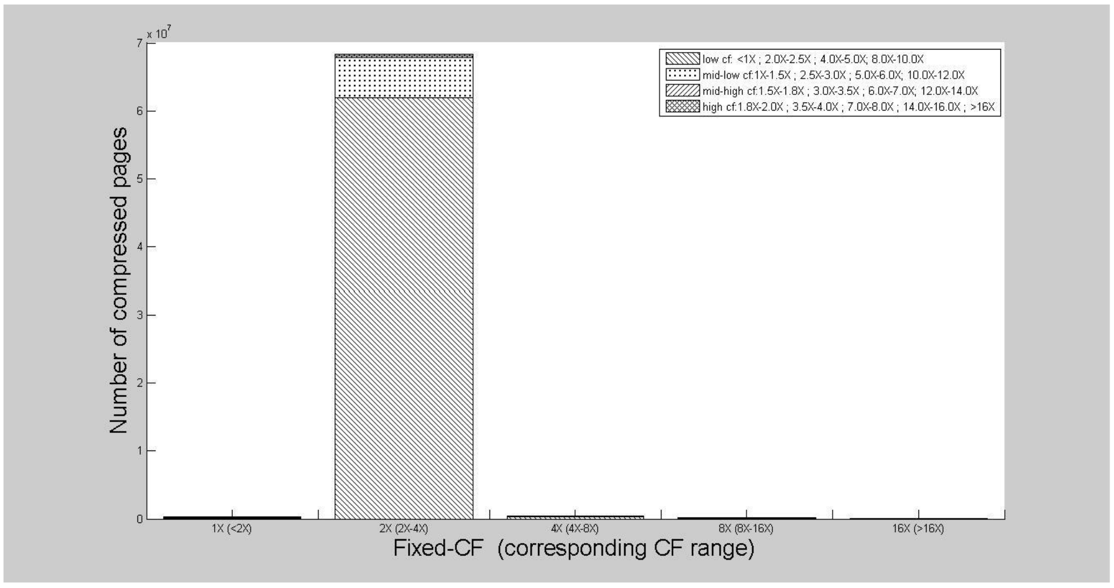
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

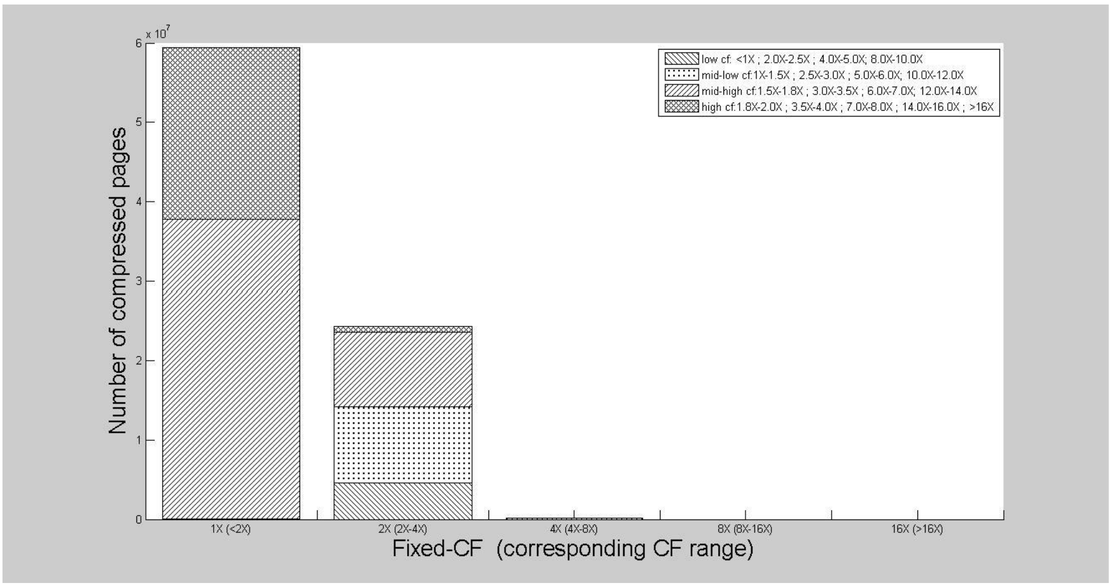
Figure 4.15: *CF distribution of pages by fixed-size compression for xalan phase1.*

More details about the fixed-size compression in *xalan* phase1 are shown in Table 4.7. The percentage of pages in each region (corresponding each pattern of each bar in the figure) of the Fixed-size CF options is listed in the table. As bigger VFT size and granularity are applied, some pages have compressibility improvement from the 2X-2.5X region to 2.5X-3X region. As a result, allowing compression by extra factor options of 1.5X and 3X introduces negligible improvement of 0.46% on the average CF as shown in the last column of Table 4.7.

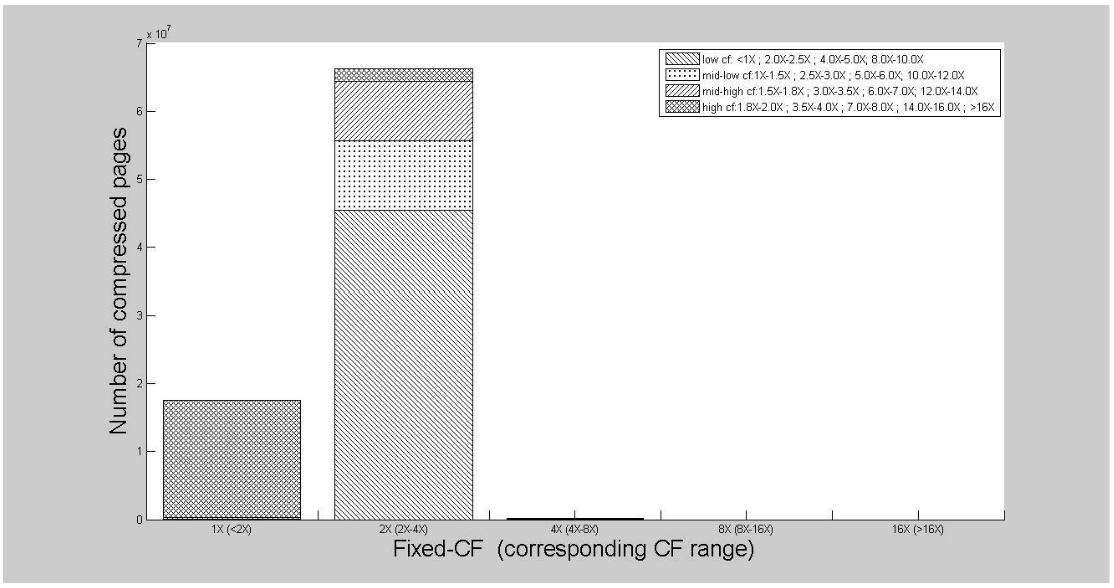
Table 4.7 *CF distribution of all compressed pages for xalan phase 1.*

Number of compressed pages	Granularity -VFT size	Fixed-size CF option	Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high	$CF_{fixed\text{-}size}$	$CF_{1.5+3X}$
69169855	G2-128	1X	3599(0.78%), 150751(32.58%), 227982(49.27%), 80388(17.37%)	1.9925	2.0024
		2X	67654004(99.00%), 274174(0.40%), 97395(0.14%), 314122(0.46%)		
		4X	101681(34.48%), 52613(17.84%), 29393(9.97%), 111170(37.70%)		
		8X	7664(10.56%), 10735(14.79%), 9262(12.76%), 44922(61.89%)		
		16X	0(0%)		
	G2-8192	1X	3599(0.84%), 148473(34.53%), 194753(45.29%), 83172(19.34%)	1.9941	2.0031
		2X	67668808(99.02%), 274534(0.40%), 100773(0.15%), 291621(0.43%)		
		4X	137290(41.48%), 52829(15.96%), 29633(8.95%), 111247(33.61%)		
		8X	7936(10.85%), 10772(14.73%), 2269(3.10%), 52146(71.31%)		
		16X	0(0%)		
	G4-1024	1X	3599(1.14%), 58829(18.68%), 176829(56.15%), 75673(24.03%)	2.0007	2.0100
		2X	67587197(98.95%), 245353(0.36%), 234225(0.34%), 235988(0.35%)		
		4X	243469(70.07%), 30861(8.88%), 63013(18.13%), 10138(2.92%)		
		8X	130835(90.46%), 2054(1.42%), 10512(7.27%), 1237(0.86%)		
		16X	60043(100%)		

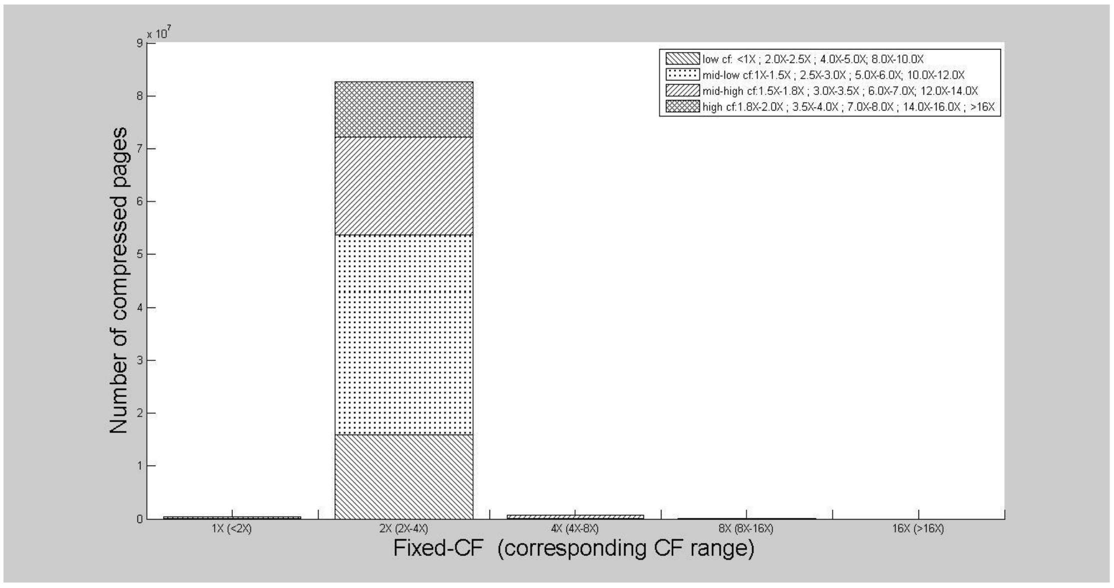
	G4-16384	1X	2907(1.18%), 58752(23.83%), 78441(31.82%), 106441(43.17%)	2.0033	2.0115
		2X	61943054(90.66%), 5901713(8.64%), 204218(0.30%), 278343(0.41%)		
		4X	257383(65.86%), 35714(9.14%), 78293(20.03%), 19428(4.97%)		
		8X	96083(66.29%), 36965(25.50%), 6258(4.32%), 5630(3.88%)		
		16X	60232(100%)		



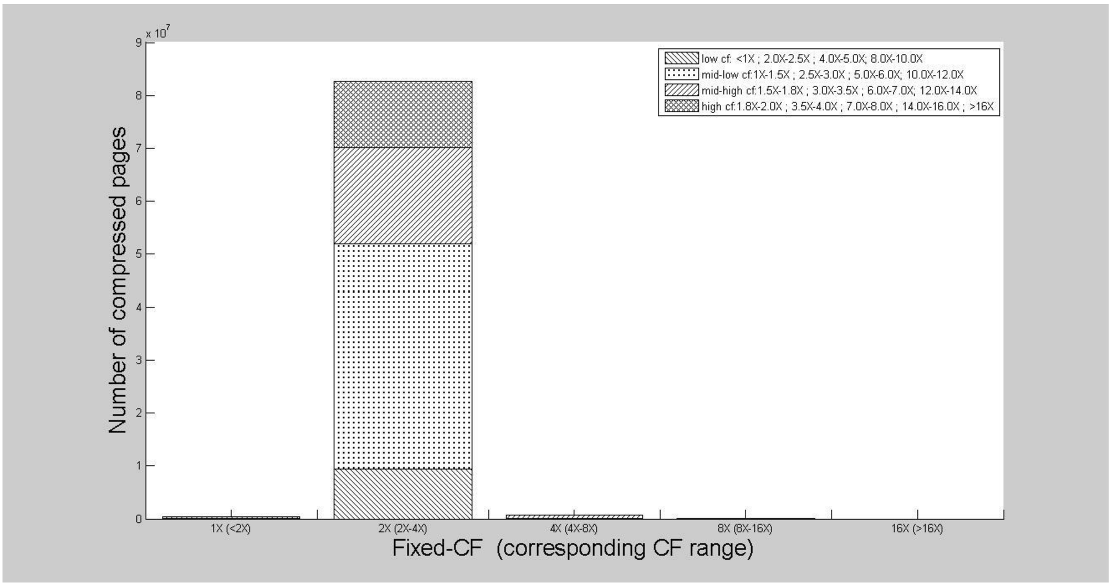
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

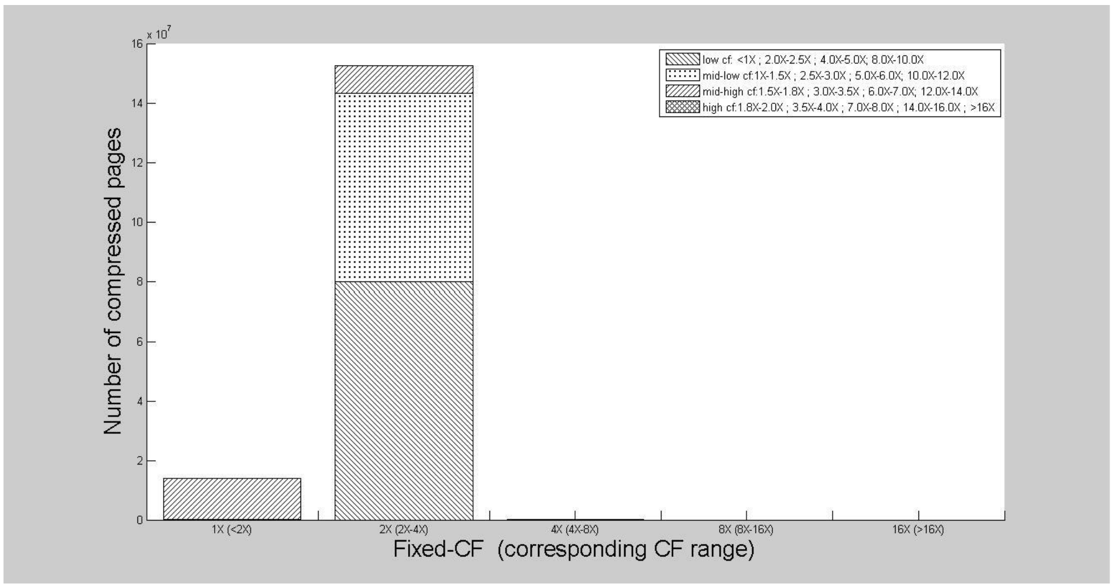
Figure 4.16: *CF distribution of pages by fixed-size compression for xalan phase2.*

More details about the fixed-size compression in *xalan* phase 2 are shown in Table 4.8. As bigger VFT size and granularity are applied, most pages can be compressed by the fixed factor of 2X. Besides, as most pages have actual CF in the region of 1.5-2X for G2-VFT-128-mfv, there is a significant compressibility improvement of 42.9% by allowing extra fixed-CF option of 1.5X.

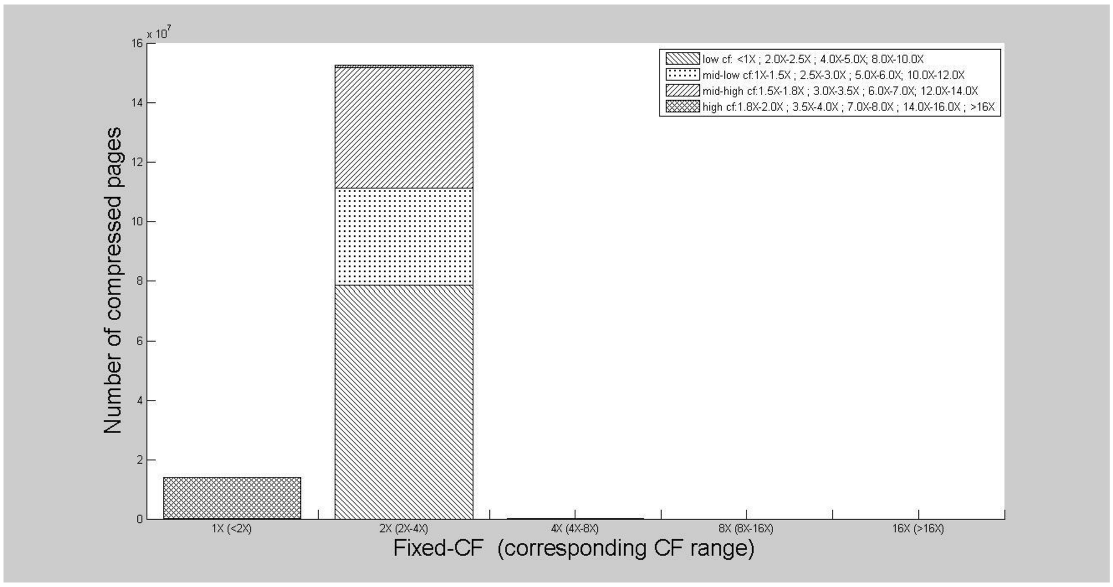
Table 4.8 *CF distribution of all compressed pages for xalan phase 2.*

Number of compressed pages	Granularity -VFT size	Fixed-size CF option	Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high	$CF_{fixed\text{-}size}$	$CF_{1.5+3X}$
83890314	G2-128	1X	11008(0.02%), 0(0%), 37716538(63.53%), 21644799(36.46%)	1.1716	1.6742
		2X	4637152(19.01%), 9578054(39.26%), 9397556(38.52%), 781188(3.20%)		
		4X	0(0%), 0(0%), 124019(100.00%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	11008(0.06%), 0(0%), 348085(1.99%),	1.6557	1.9482

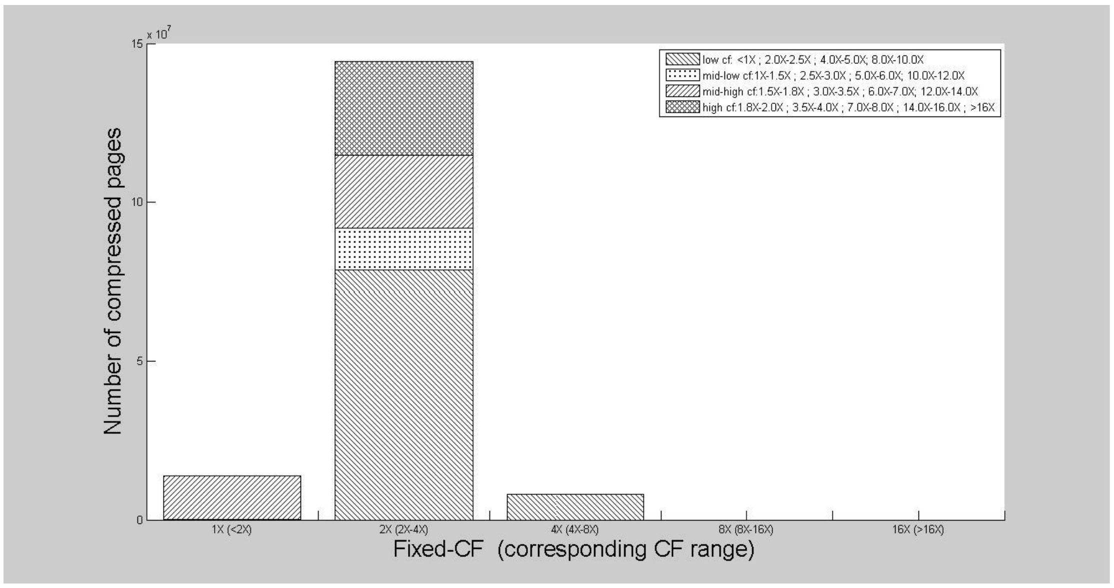
			17163659(97.95%)		
		2X	45386060(68.55%), 10233901(15.46%), 8736259(13.19%), 1855823(2.80%)		
		4X	31500(20.25%), 0(0%), 124019(79.75%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	11008(2.95%), 0(0%), 318882(85.52%), 42991(11.53%)	2.0027	2.2716
		2X	15985358(19.35%), 37602795(45.52%), 18567508(22.48%), 10455576(12.66%)		
		4X	1439(0.18%), 0(0%), 780738(99.82%), 0(0%)		
		8X	124019(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-16384	1X	10688(2.85%), 320(0.09%), 320302(85.47%), 43449(11.59%)	2.0026	2.2888
		2X	9457858(11.45%), 42462216(51.40%), 18172896(22.00%), 12516283(15.15%)		
		4X	1355(0.17%), 190(0.02%), 780738(99.80%), 0(0%)		
		8X	124019(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		



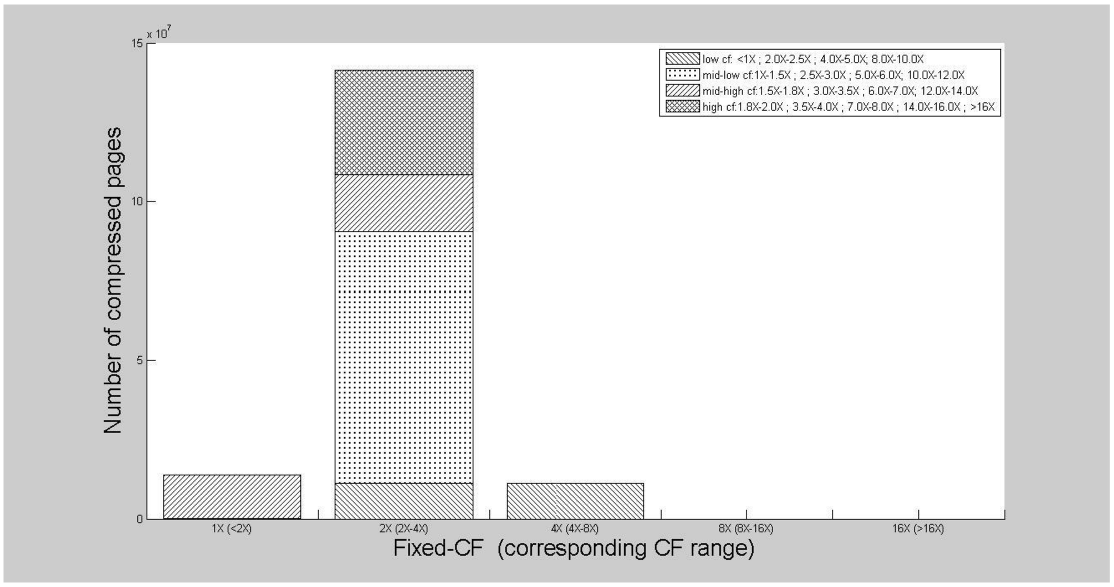
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

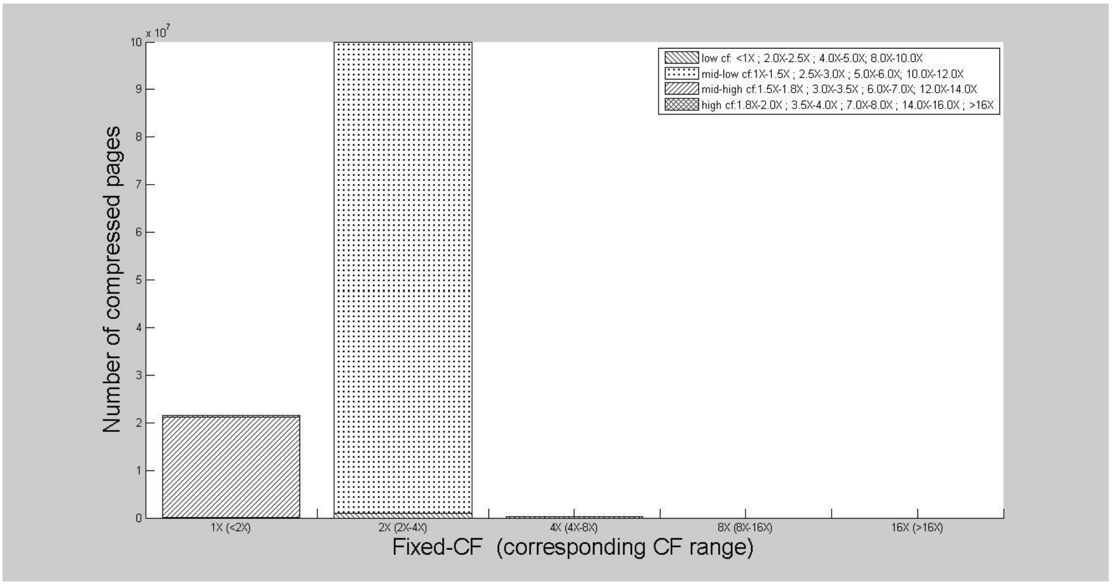
Figure 4.17: *CF distribution of pages by fixed-size compression for omnetpp phase2.*

More details about the fixed-size compression in *omnetpp* phase 2 are shown in Table 4.9.

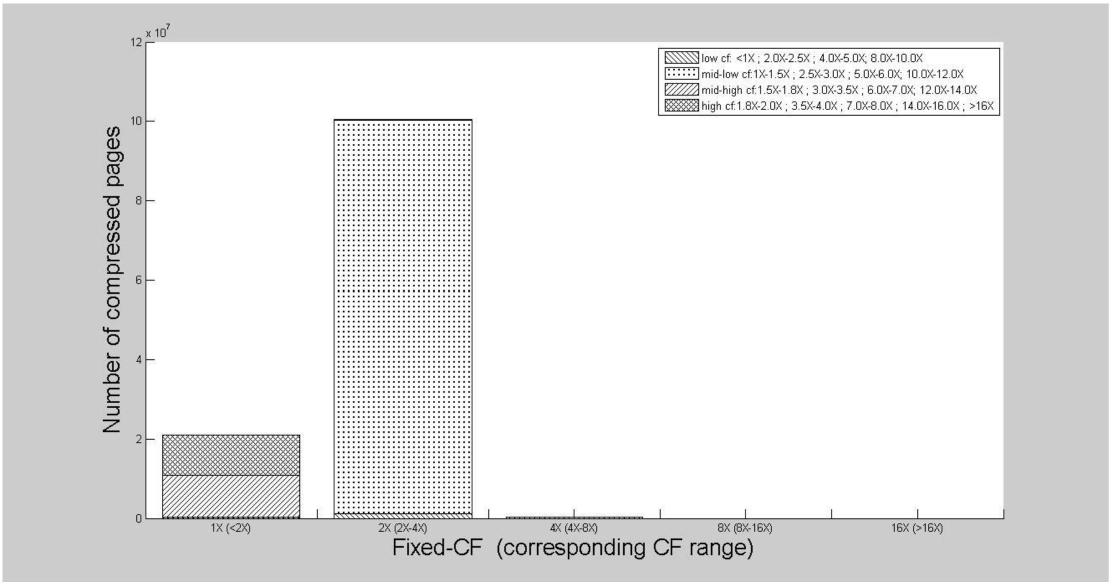
Table 4.9 *CF distribution of all compressed pages for omnetpp phase 2.*

Number of compressed pages	**Granularity -VFT size**	**Fixed-size CF option**	**Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high**	**$CF_{fixed\text{-}size}$**	**$CF_{1.5+3X}$**
166313233	G2-128	1X	6(0.00%), (0.00%), 3815991(100.00%), 0(0%)	1.8466	1.9815
		2X	79775073(52.31%), 63586128(41.70%), 9126790(5.98%), 8404(0.01%)		
		4X	837(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	8(0.00%), 1(0.00%), 4(0.00%), 13815989(100.00%)	1.8466	2.1176
		2X	78554462(51.51%), 32449628(21.28%), 40656673(26.66%), 833914(0.55%)		
		4X	2554(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	6(0.00%), 4(0.00%), 13815991(100.00%), 0(0%)	1.8894	2.2277
		2X	78552437(54.42%), 13254552(9.18%), 22994690(15.93%), 29531185(20.46%)		
		4X	8164284(100.00%), 84(0.00%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-16384	1X	6(0.00%), 4(0.00%), 13815991(100.00%), 0(0%)	1.9056	2.2415
		2X	11315969(8.01%), 79232643(56.06%), 17916649(12.68%), 32870827(23.26%)		
		4X	11161050(100.00%), 94(0.00%), 0(0%),		

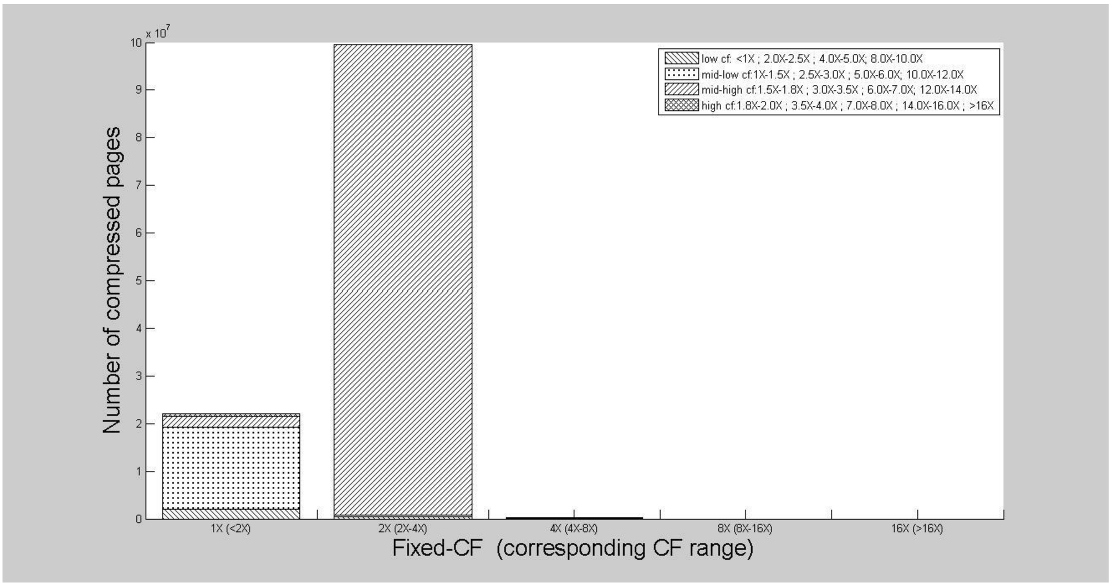
			0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		



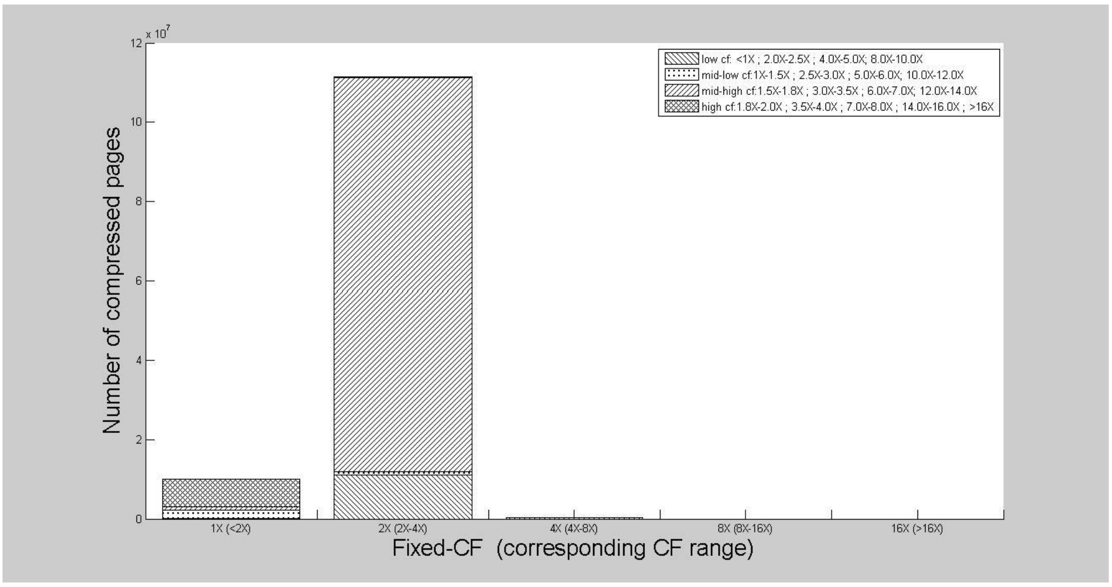
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

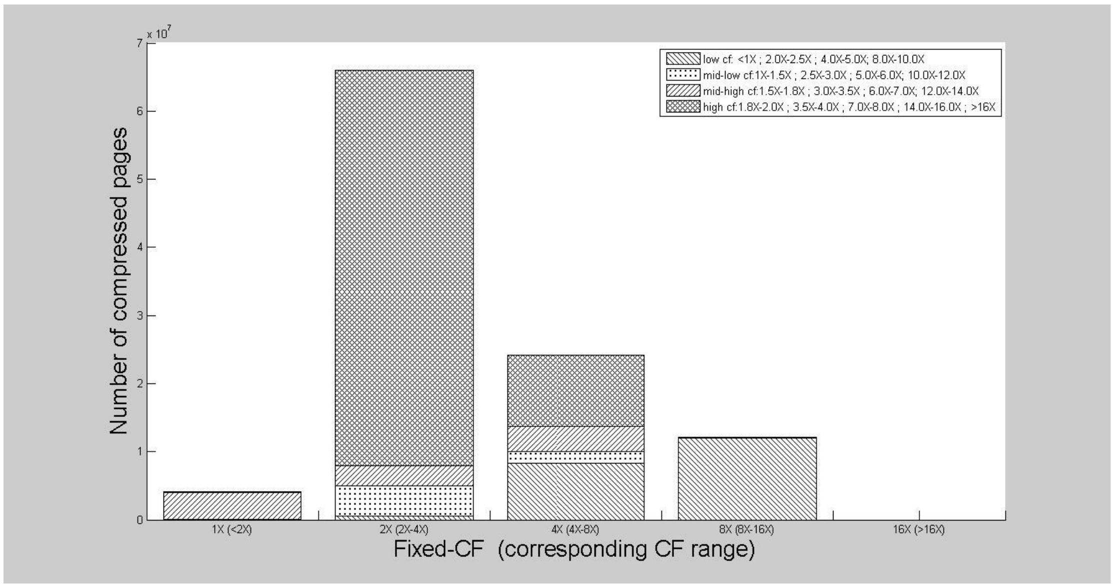
Figure 4.18: *CF distribution of pages by fixed-size compression for astar phase1.*

More details about the fixed-size compression in *astar* phase1 are shown in Table 4.10. For most pages, the CF improvement due to larger VFT size and granularity is not enough for the pages to be compressed by 4X from 2X. As a result, the $CF_{fixed\text{-}size}$ for different granularities and VFT sizes are almost the same. On the other hand, extra fixed-CF option of 3X is helpful to improve the compressibility. As a result, the $CF_{1.5+3X}$ for G4-16K-mfv is 38.13% higher than the one for G2-128-mfv.

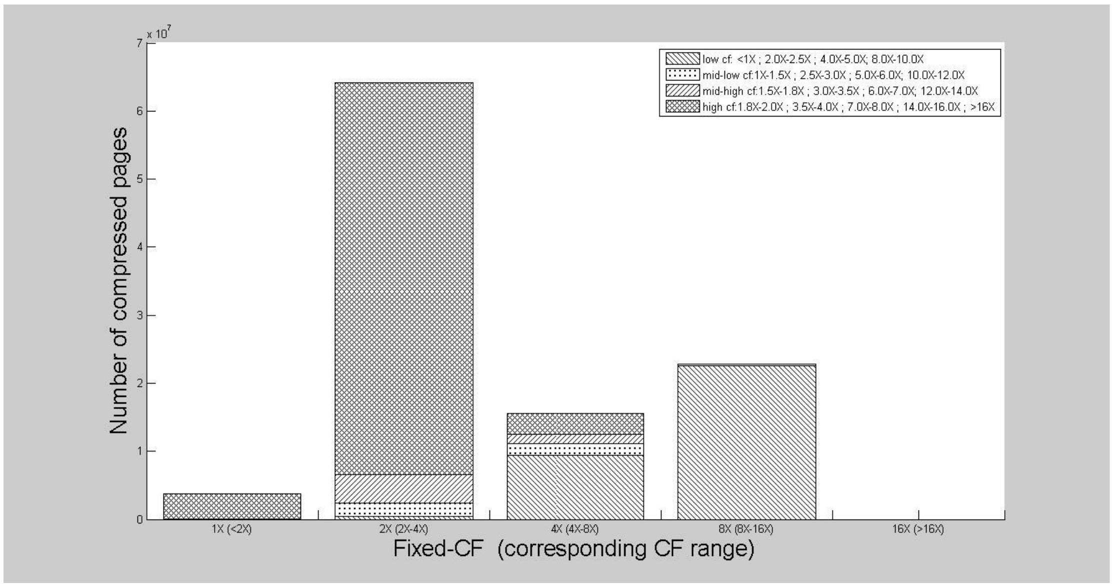
Table 4.10 *CF distribution of all compressed pages for astar phase 1.*

Number of compressed pages	**Granularity -VFT size**	**Fixed-size CF option**	**Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high**	**CF$_{\text{fixed-size}}$**	**CF$_{1.5+3X}$**
121847816	G2-128	1X	0(0%), 34422(0.16%), 21104290(97.65%), 473986(2.19%)	1.7003	1.8906
		2X	937486(0.94%), 98917503(98.94%), 117374(0.12%), 23(0.00%)		
		4X	262156(99.78%), 576(0.22%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	0(0%), 360056(1.70%), 10567381(50.03%), 10193531(48.26%)	1.7061	1.8899
		2X	1100594(1.10%), 99230993(98.77%), 132408(0.13%), 121(0.00%)		
		4X	262156(99.78%), 576(0.22%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	2073993(9.39%), 17291733(78.28%), 2167127(9.81%), 555497(2.51%)	1.6946	2.2342
		2X	527074(0.53%), 275903(0.28%), 98691933(99.19%), 1650(0%)		
		4X	7(0.00%), 262312(99.77%), 11(0.00%), 576(0.22%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-16384	1X	49809(0.49%), 2107582(20.89%), 984149(9.76%), 6945224(68.85%)	1.8490	2.6114
		2X	11003298(9.87%), 963722(0.86%), 99216344(88.99%), 311397(0.28%)		

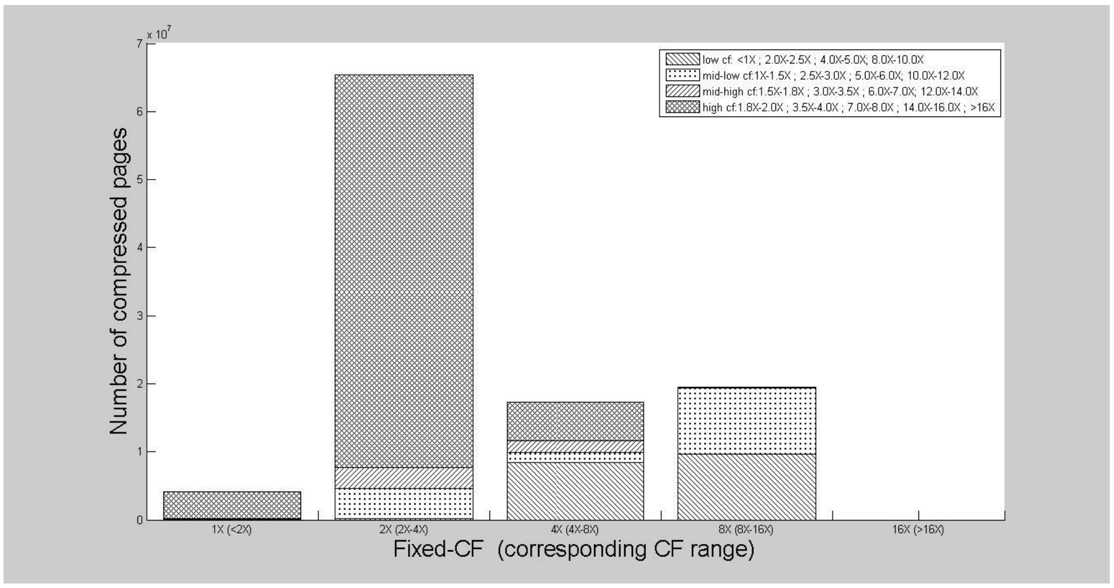
		4X	265586(99.74%), 129(0.05%), 576(0.22%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		



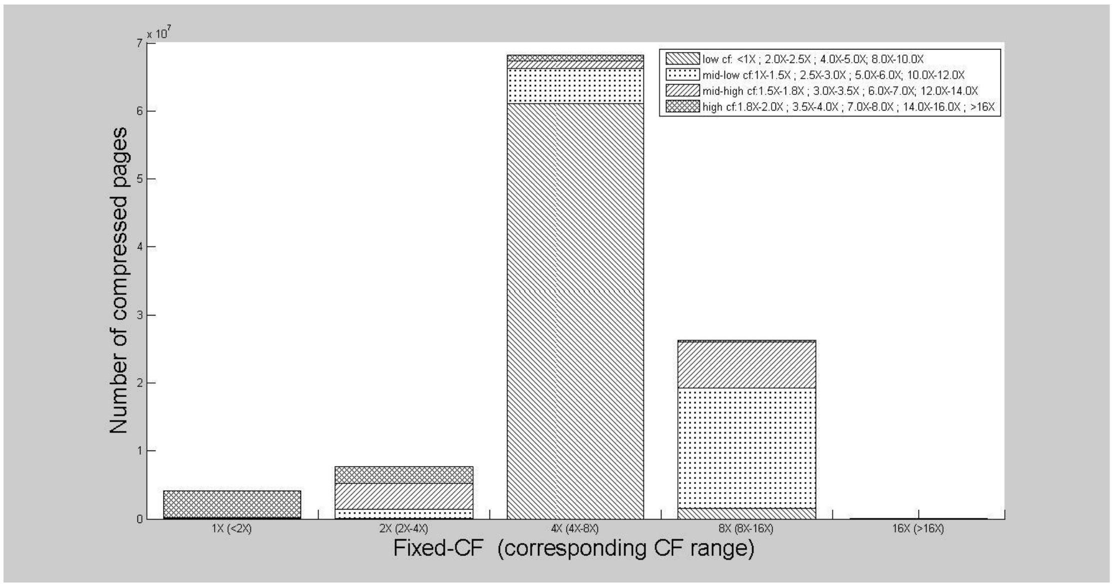
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

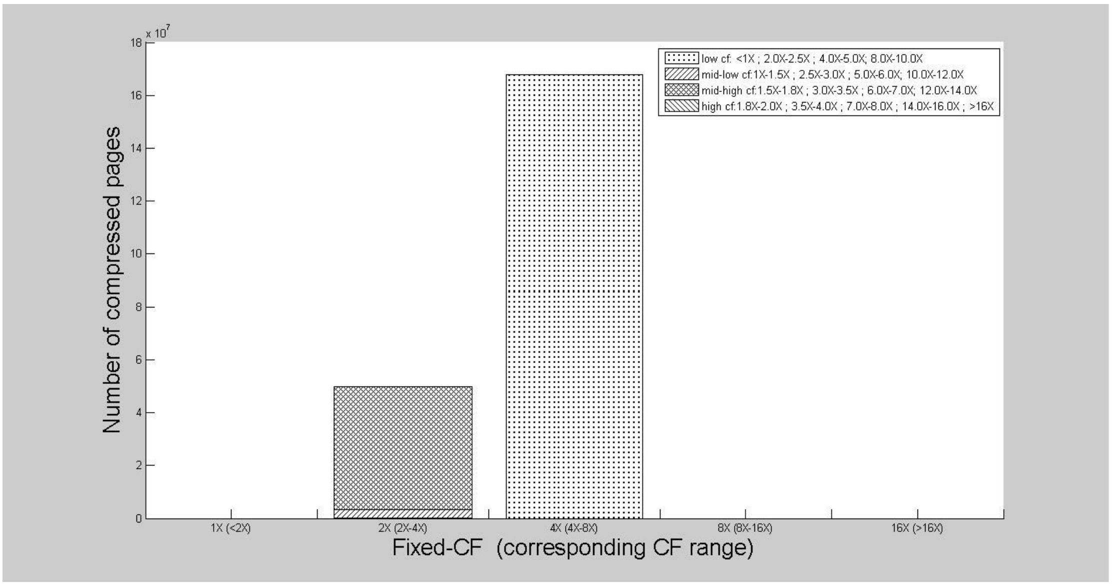
Figure 4.19: *CF distribution of pages by fixed-size compression for astar phase 2.*

More details about the fixed-size compression in *astar* phase 2 are shown in Table 4.11. As shown is Figure 4.19(d), the compression for G4-VFT-16K-mfv introduces many pages to be compressed in 4X which leads a significant improvement of $CF_{fixed\text{-}size}$(CF by applying fixed-size compression).

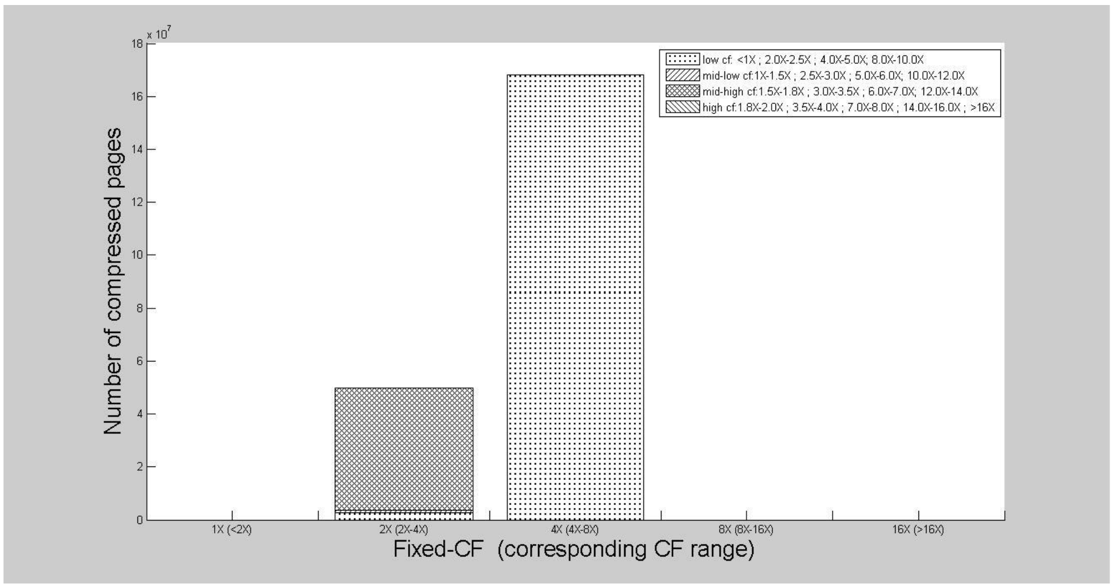
Table 4.11 *CF distribution of all compressed pages for astar phase 2.*

Number of compressed pages	Granularity -VFT size	Fixed-size CF option	Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high	$CF_{fixed\text{-}size}$	$CF_{1.5+3X}$
106378293	G2-128	1X	527(0.01%), 26520(0.65%), 3981115(97.51%), 74499(1.82%)	2.3826	3.2124
		2X	523352(0.79%), 4432833(6.72%), 3044101(4.61%), 58008535(87.88%)		
		4X	8334061(34.44%), 1648752(6.81%), 3720749(15.37%), 10498173(43.38%)		
		8X	12077441(99.94%), 7635(0.06%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	0(0%), 6362(0.17%), 72782(1.94%), 3669499(97.89%)	2.4982	3.2124
		2X	390485(0.61%), 1987844(3.10%), 4196406(6.54%), 57560889(89.75%)		
		4X	9475372(60.62%), 1702893(10.89%), 1316601(8.42%), 3136203(20.06%)		
		8X	22655281(99.09%), 207676(0.91%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	1114(0.03%), 133244(3.22%), 45292(1.10%), 3954565(95.65%)	2.4403	3.3105
		2X	226585(0.35%), 4431674(6.78%), 3006870(4.60%), 57711048(88.28%)		
		4X	8476574(49.03%), 1470704(8.51%), 1702715(9.85%), 5637972(32.61%)		
		8X	9709058(49.59%), 9758700(49.84%), 106744(0.55%), 5434(0.03%)		
		16X	0(0%)		

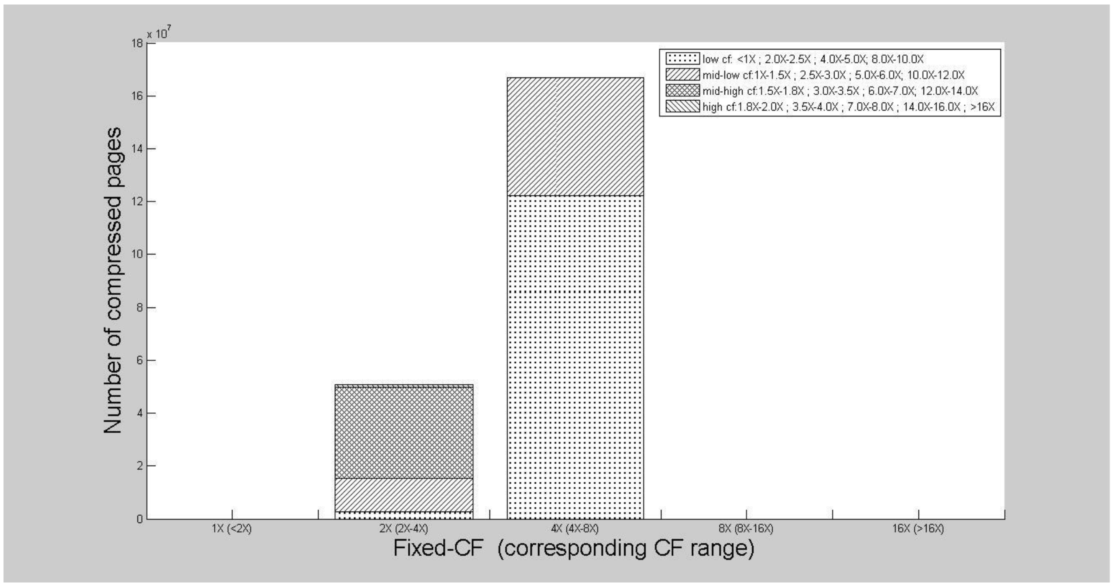
	G4-16384	1X	2200(0.05%), 122461(2.99%), 31438(0.77%), 3939973(96.19%)	3.7596	4.1049
		2X	40287(0.52%), 1326624(17.24%), 3915045(50.86%), 2415304(31.38%)		
		4X	61074666(89.52%), 5170931(7.58%), 1062511(1.56%), 912925(1.34%)		
		8X	1601783(6.08%), 17746550(67.32%), 6774867(25.70%), 236942(0.90%)		
		16X	3786(100%)		



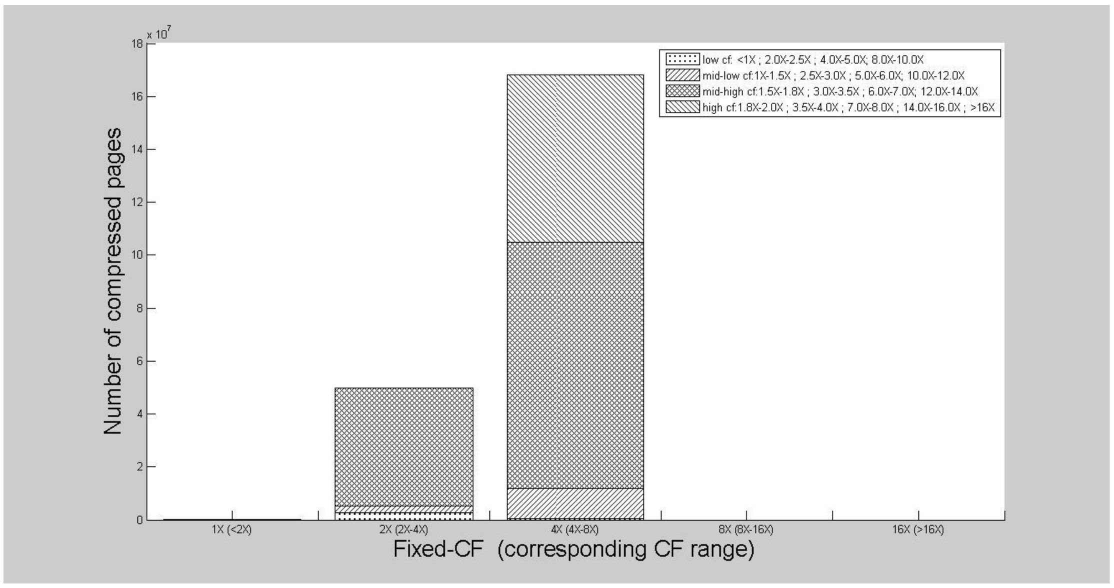
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

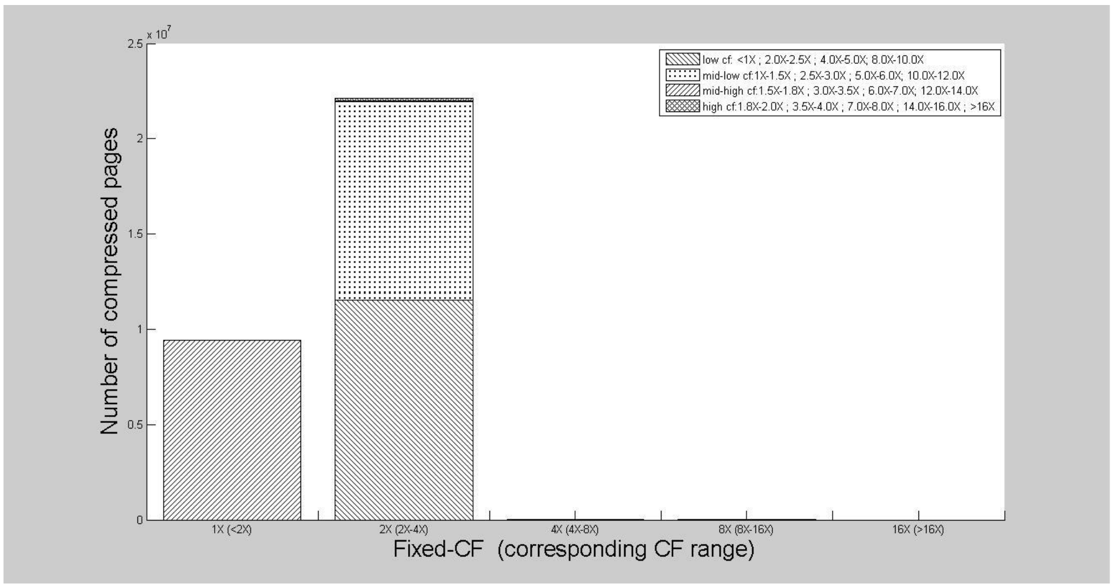
Figure 4.20: *CF distribution of pages by fixed-size compression for mcf phase 1.*

More details about the fixed-size compression in *mcf* phase 1 are shown in Table 4.12. The difference between $CF_{\text{fixed-size}}$ and $CF_{1.5+3X}$ is small due to the lack of extra fixed-CF option in the region of 4-8X.

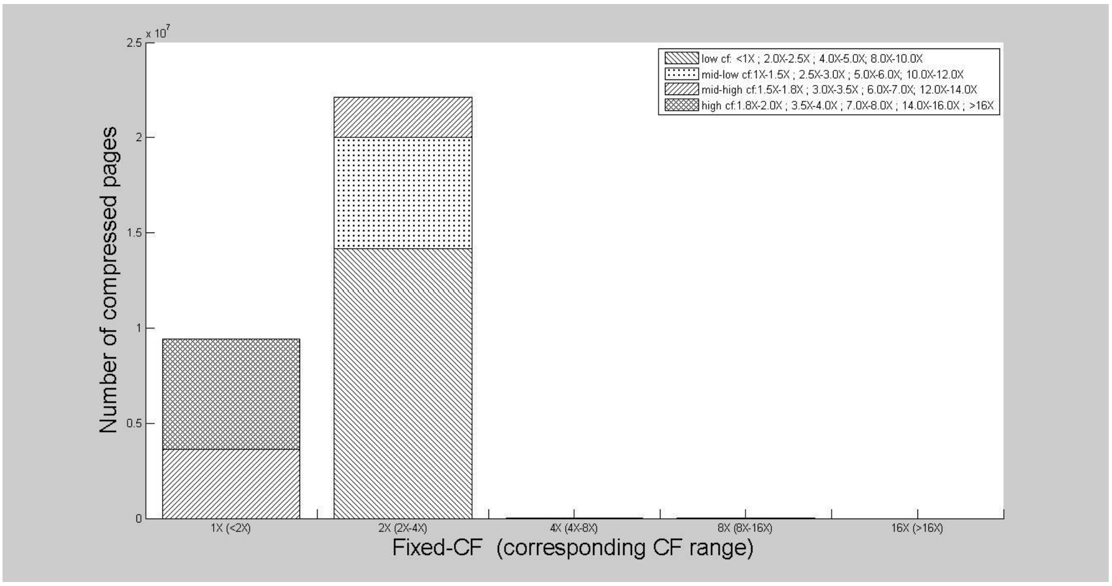
Table 4.12 *CF distribution of all compressed pages for mcf phase 1.*

Number of compressed pages	**Granularity -VFT size**	**Fixed-size CF option**	**Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high**	**$CF_{\text{fixed-size}}$**	**$CF_{1.5+3X}$**
217752746	G2-128	1X	0(0%), 0(0%), 0(0%), 0(0%)	3.2539	3.6810
		2X	697(0.00%), 3367587(6.75%), 46391263(92.93%), 163011(0.33%)		
		4X	167830188(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	0(0%), 0(0%), 0(0%), 0(0%)	3.2558	3.6795
		2X	2788123(5.60%), 802617(1.61%), 46168807(92.77%), 7451(0.01%)		
		4X	167984419(100.00%), 1329(0.00%), 0(0%),		

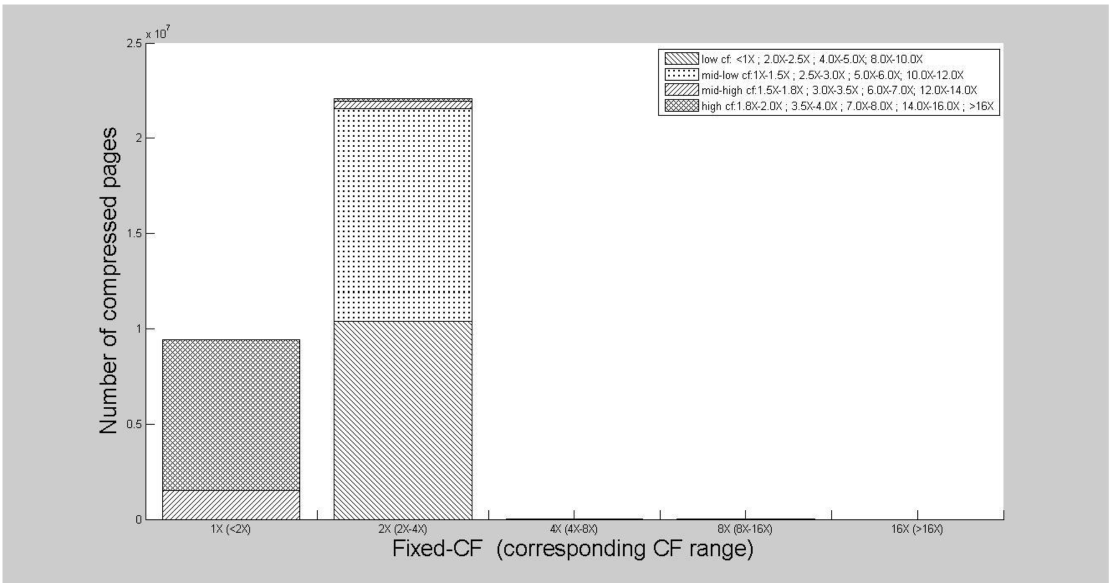
			0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	0(0%), 0(0%), 0(0%), 0(0%)	3.2426	3.5571
		2X	2788123(5.48%), 12468650(24.52%), 34502741(67.85%), 1095524(2.15%)		
		4X	122274820(73.26%), 44549276(26.69%), 73612(0.04%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-16384	1X	0(0%), 0(0%), 0(0%), 20(100.00%)	3.2559	3.6628
		2X	2788119(5.60%), 2422052(4.87%), 44549323(89.52%), 4719(0.01%)		
		4X	510783(0.30%), 11286933(6.72%), 93040667(55.39%), 63150130(37.59%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		



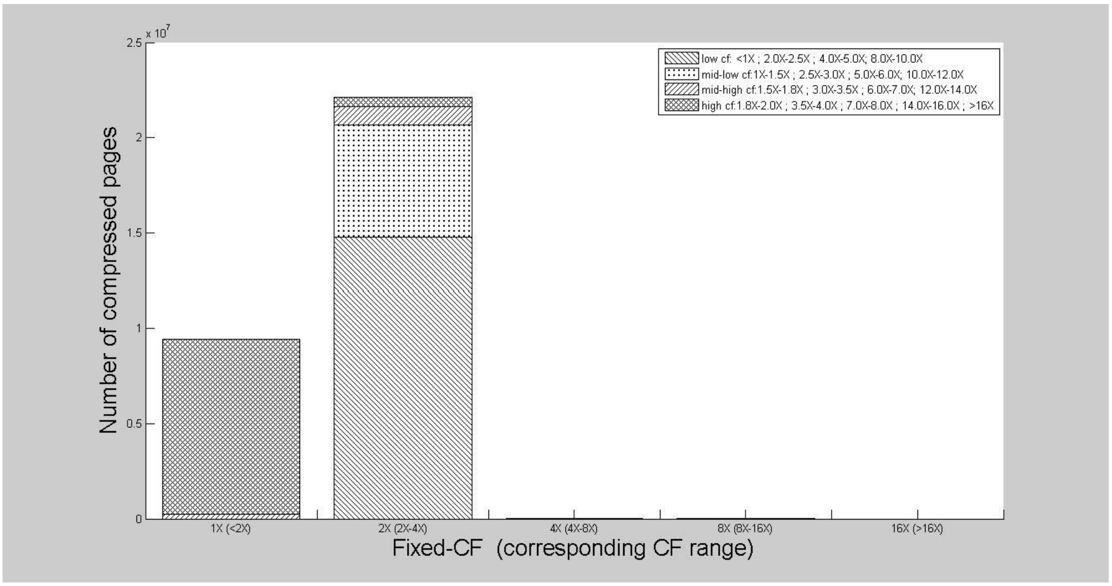
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

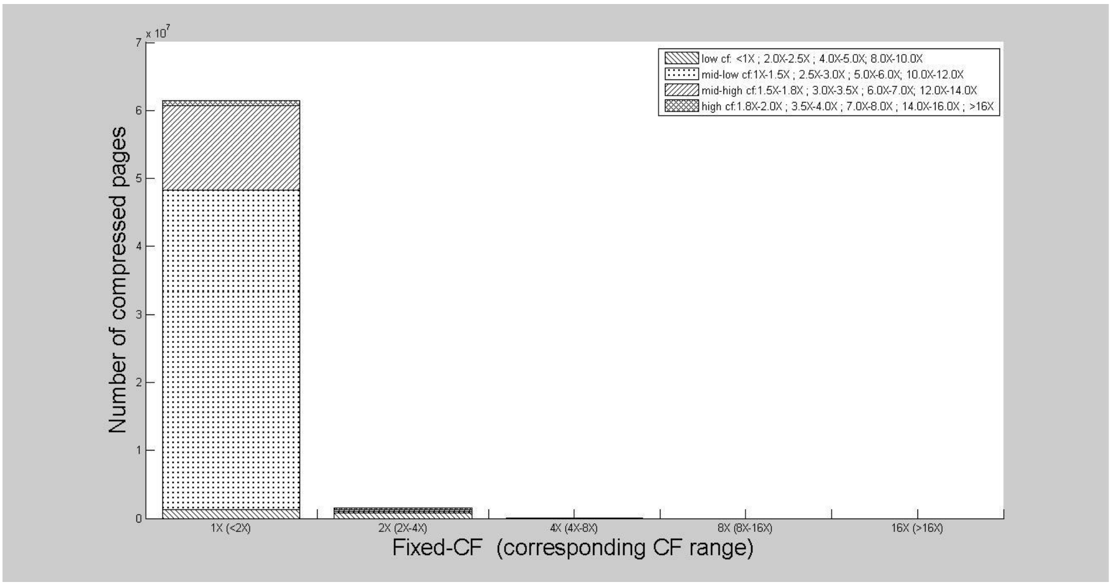
Figure 4.21: *CF distribution of pages by fixed-size compression for mcf phase2.*

More details about the fixed-size compression in *mcf* phase 2 are shown in Table 4.13.

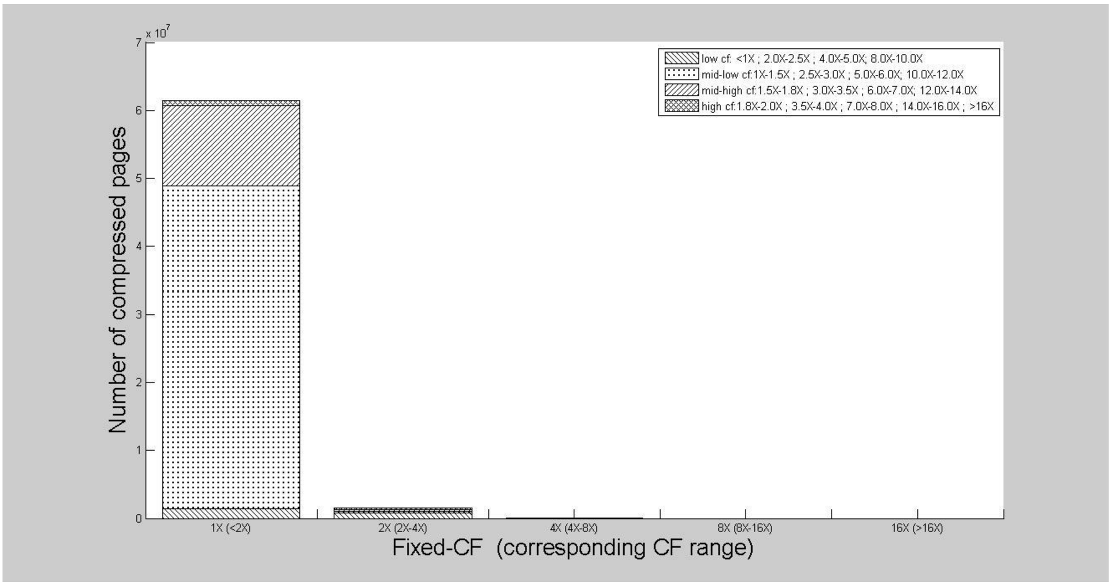
Table 4.13 *CF distribution of all compressed pages for mcf phase 2.*

Number of compressed pages	Granularity -VFT size	Fixed-size CF option	Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high	$CF_{fixed-size}$	$CF_{1.5+3X}$
31570405	G2-128	1X	0(0%), 0(0%), 9424528(100.00%), 123(0.00%)	1.5412	1.8232
		2X	11552447(52.26%), 10408187(47.08%), 27219(0.12%), 118353(0.54%)		
		4X	4781(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	34767(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	0(0%), 0(0%), 3645072(38.68%), 5779579(61.32%)	1.5413	1.8584
		2X	14147628(64.01%), 5849002(26.46%), 2107131(9.53%), 0(0%)		

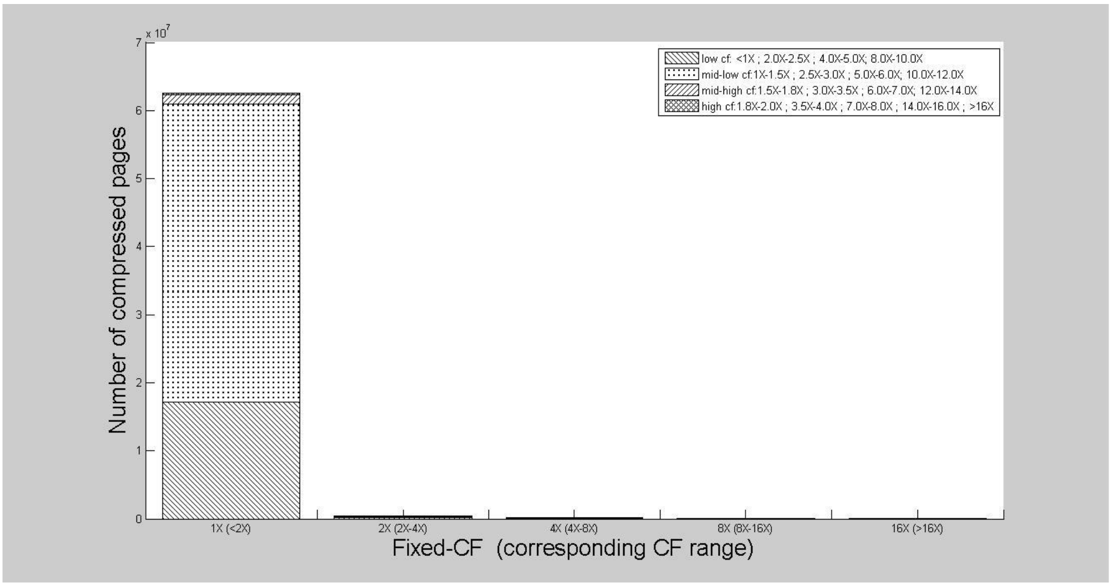
		4X	7226(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	34767(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	0(0%), 0(0%), 1521421(16.10%), 7929796(83.90%)	1.5401	1.8295
		2X	10412098(47.15%), 11133935(50.42%), 410113(1.86%), 128217(0.58%)		
		4X	58(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	34767(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-16384	1X	0(0%), 0(0%), 229024(2.43%), 9199415(97.57%)	1.5410	1.8459
		2X	14779204(66.85%), 5893444(26.66%), 963721(4.36%), 470265(2.13%)		
		4X	565(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	34767(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		



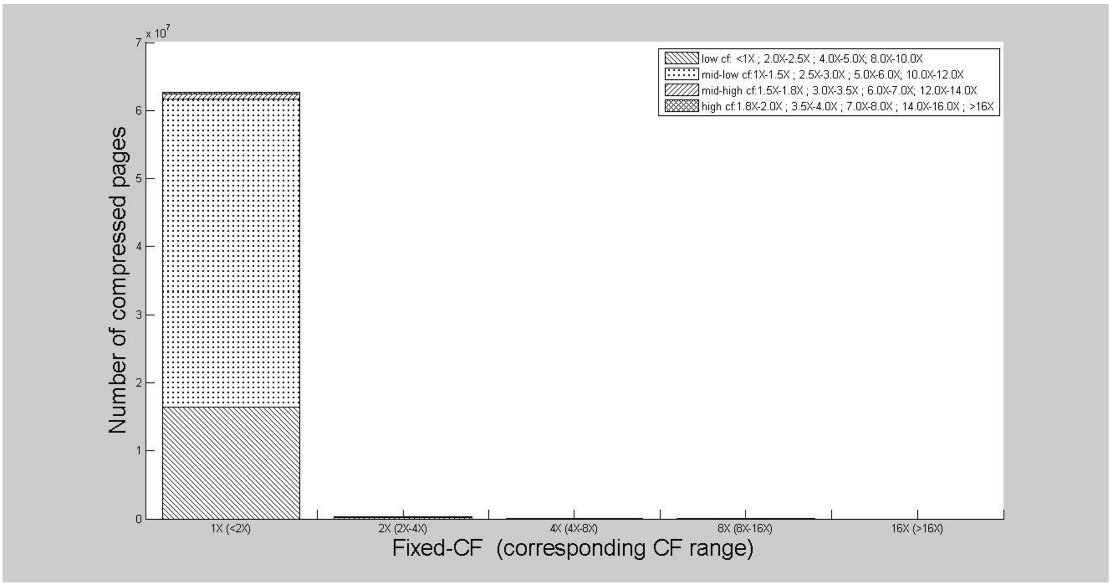
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

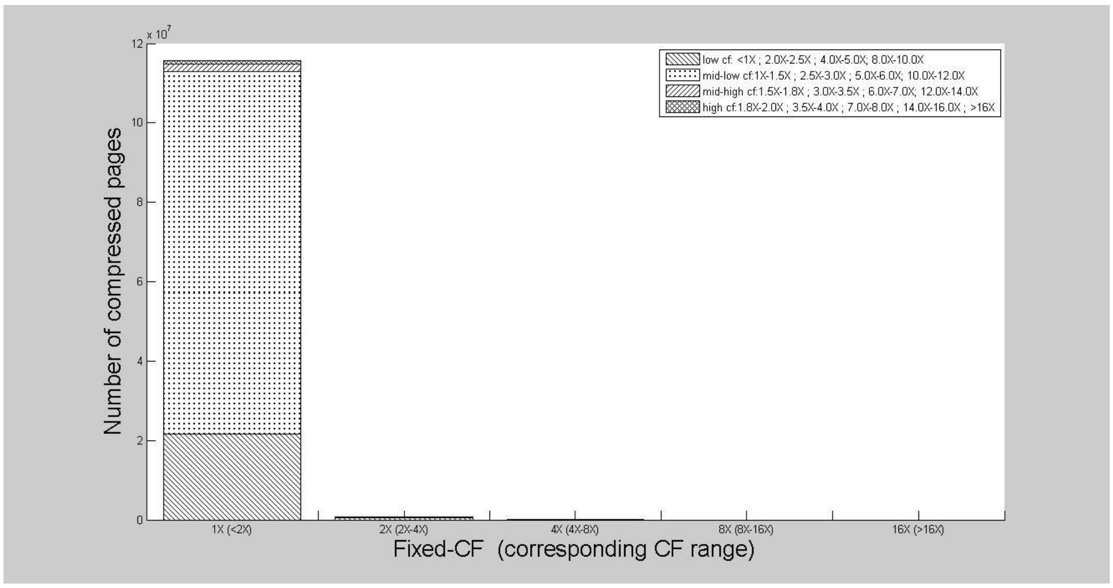
Figure 4.22: *CF distribution of pages by fixed-size compression for bzip2 phase 1.*

More details about the fixed-size compression in *bzip2* phase 1 are shown in Table 4.14. By applying fixed-size compression, most pages keep as uncompressed due to their low actual compressibility (<2X).

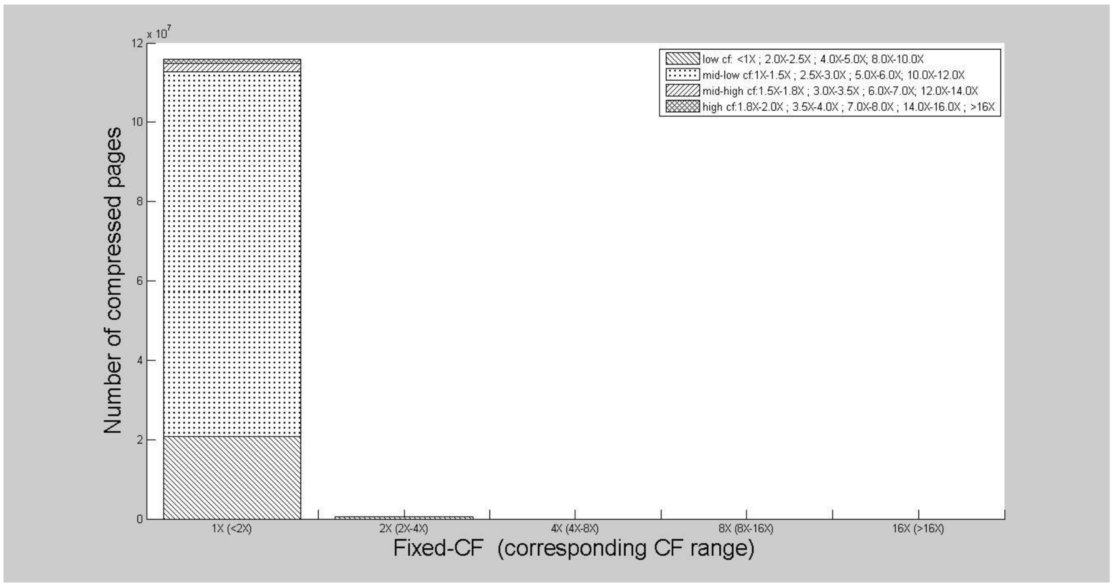
Table 4.14 *CF distribution of all compressed pages for bzip2 phase 1.*

Number of compressed pages	Granularity -VFT size	Fixed-size CF option	Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high	$CF_{fixed-size}$	$CF_{1.5+3X}$
63068975	G2-128	1X	1352662(2.20%), 46900132(76.29%), 12408273(20.18%), 812255(1.32%)	1.0131	1.0917
		2X	782624(51.02%), 285540(18.61%), 212862(13.88%), 252991(16.49%)		
		4X	61636(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	1419104(2.31%), 47439364(77.16%), 11825550(19.23%), 801602(1.30%)	1.0127	1.0878
		2X	785196(49.75%), 282553(17.90%),		

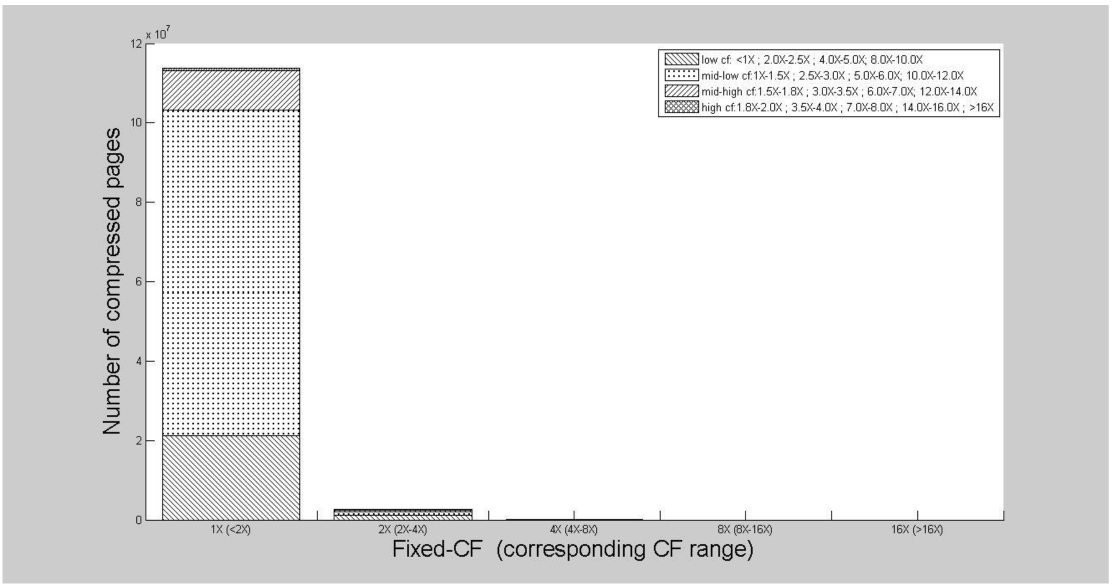
			220855(13.99%), 289726(18.36%)		
		4X	5025(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	17206594(27.53%), 43683582(69.89%), 1365993(2.19%), 243216(0.39%)	1.0053	1.0143
		2X	199267(51.44%), 85347(22.03%), 58511(15.11%), 44219(11.42%)		
		4X	72959(45.19%), 38874(24.08%), 28372(17.57%), 21229(13.15%)		
		8X	10484(50.38%), 4629(22.25%), 3251(15.62%), 2445(11.75%)		
		16X	3(100.00%)		
	G4-16384	1X	16498136(26.34%), 45235371(72.21%), 715908(1.14%), 190806(0.30%)	1.0035	1.0089
		2X	143762(39.57%), 93759(25.80%), 77629(21.37%), 48188(13.26%)		
		4X	48613(80.14%), 5506(9.08%), 3731(6.15%), 2812(4.64%)		
		8X	3825(80.46%), 929(19.54%), 0(0%), 0(0%)		
		16X	0(0%)		



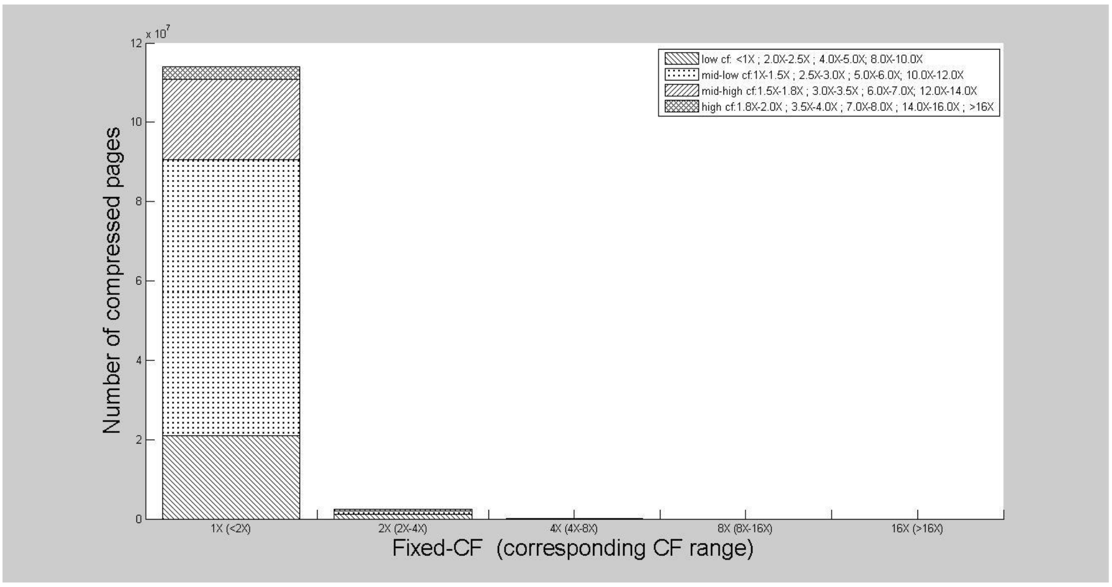
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

Figure 4.23: *CF distribution of pages by fixed-size compression for bzip2 phase 2.*

More details about the fixed-size compression in *bzip2* phase 2 are shown in Table 4.15. Similarly as *bzip2* phase 1, most pages keep uncompressed. As most pages have actual compressibility lower than 1.5X, the extra fixed-CF options of 1.5X and 3X don't make sense in this case.

Table 4.15 *CF distribution of all compressed pages for bzip2 phase 2.*

Number of compressed pages	Granularity -VFT size	Fixed-size CF option	Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high	$CF_{fixed-size}$	$CF_{1.5+3X}$
116452964	G2-128	1X	21596227(18.67%), 91366413(78.99%), 1866813(1.61%), 842060(0.73%)	1.0026	1.0114
		2X	627838(87.49%), 45343(6.32%), 26792(3.73%), 17654(2.46%)		
		4X	23801(37.29%), 15829(24.80%), 11326(17.75%), 12868(20.16%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	20900261(18.04%), 91785769(79.23%), 2098873(1.81%), 1067657(0.92%)	1.0114	1.0119
		2X	501743(83.57%), 47591(7.93%), 26471(4.41%), 24599(4.10%)		
		4X	0(0%), 0(0%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	21226846(18.64%), 81943159(71.97%), 10042689(8.82%), 640907(0.56%)	1.0113	1.0446
		2X	1175940(46.22%), 773681(30.41%), 562716(22.12%), 31752(1.25%)		
		4X	28651(51.83%), 16407(29.68%), 10216(18.48%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-16384	1X	20931274(18.38%), 69640159(61.15%), 20331642(17.85%), 2974556(2.61%)	1.0035	1.0854
		2X	1173584(46.62%), 794589(31.56%),		

			524164(20.82%), 25232(1.00%)		
		4X	30535(52.86%), 16969(29.38%), 10260(17.76%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		

The reason of the negligible impact of granularity and VFT size on the compressibility can be found from Figure 4.15 to Figure 4.23. That is, increasing VFT size or applying larger granularity indeed improves the potential compressibility of each page. The significant variation of compressibility distribution of pages can be easily seen when comparing the sub-figures of each picture. The problem is the compressibility improvement normally is not enough for the page to reach the next fixed-CF level. For example, even if the actual CF of one page is improved from 2.1X to 3.9X, the page is still compressed with fixed-CF of 2X. That is why the granularity and VFT size have negligible impact on CF in this experiment.

In conclusion, the improvement due to bigger VFT and granularity is unfortunately skewed by using coarser fixed-CF options. If we can define more fixed compression size options, as we did in this experiment, allowing the page to be compressed by the factor of 1.5X and 3X, the average CF could be improved by 15% on average. Thus, the trade-off between data placement and compressibility should be concerned.

The similar compression experiments are implemented for footprint at the end of each phase. The results are presented in Table 4.16.

Table 4.16 *Comparison between $CF_{with\ VFT}$ (from Section 4.2) and $CF_{fixed\text{-}size}$ by applying fixed-size compression on footprint, the CF distribution of pages for fixed-size compression.*

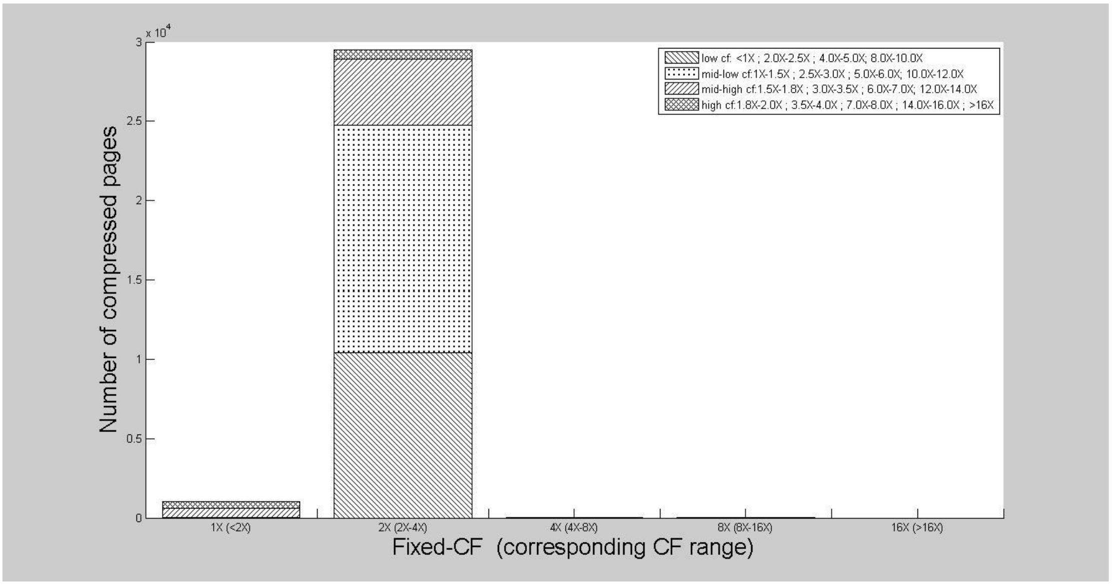
Benchmark-phase -granularity-VFT size	**$CF_{with\ VFT}$**	**$CF_{fixed\text{-}size}$**	**Total pages**	**Number of pages(percentage) compressed with fixed-CF of 1X, 2X, 4X, 8X, 16X**
omnetpp-p2-g2-128	2.7832	2.0004	36360	107(0.2943%), 36023(99.0732%), 228(0.6271%), 2(0.0055%), 0(0%)
omnetpp-p2-g2-8192	3.1036	2.0155		78(0.2145%), 35520(97.6898%), 759(2.0875%), 3(0.0083%), 0(0%)

omnetpp-p2-g4-1024	3.6145	2.1999		107(0.2943%), 29426(80.9296%), 6824(18.7679%), 1(0.0028%), 2(0.0055%)
omnetpp-p2-g4-16384	3.7629	2.3368		75(0.2063%), 25528(70.2090%), 10754(29.5765%), 1(0.0028%), 2(0.0055%)
xalan-p1-g2-128	2.5877	1.9371	30529	1009(3.3051%), 29487(96.5869%), 25(0.0819%), 8(0.0262%), 0(0%)
xalan-p1-g2-8192	2.7000	1.9549		705(2.3093%), 29779(97.5433%), 37(0.1212%), 8(0.0262%), 0(0%)
xalan-p1-g4-1024	3.1786	2.0864		562(1.8409%), 26312(86.1869%), 3644(11.9362%), 6(0.0197%), 5(0.0164%)
xalan-p1-g4-16384	3.2274	2.1015		430(1.4085%), 26168(85.7152%), 3920(12.8403%), 6(0.0197%), 5(0.0164%)
xalan-p2-g2-128	2.5939	1.9575	43283	942(2.1764%), 42334(97.8075%), 7(0.0162%), 0(0%), 0(0%)
xalan-p2-g2-8192	2.6717	1.9781		467(1.0789%), 42792(98.8656%), 24(0.0554%), 0(0%), 0(0%)
xalan-p2-g4-1024	3.0518	1.9884		314(0.7255%), 42839(98.9742%), 125(0.2888%), 5(0.0116%), 0(0%)
xalan-p2-g4-16384	3.1000	1.9868		292(0.6746%), 42855(99.0112%), 130(0.3003%), 6(0.0139%), 0(0%)
mcf-p1-g2-128	4.3082	3.9872	381032	12(0.0031%), 1186(0.3113%), 379833(99.6853%), 1(0.0003%), 0(0%)
mcf-p1-g2-8192	4.6142	3.9868		14(0.0037%), 1176(0.3086%), 379842(99.6877%), 0(0%), 0(0%)
mcf-p1-g4-1024	4.6279	3.9850		1(0.0003%), 1424(0.3737%), 379606(99.6258%), 1(0.0003%), 0(0%)

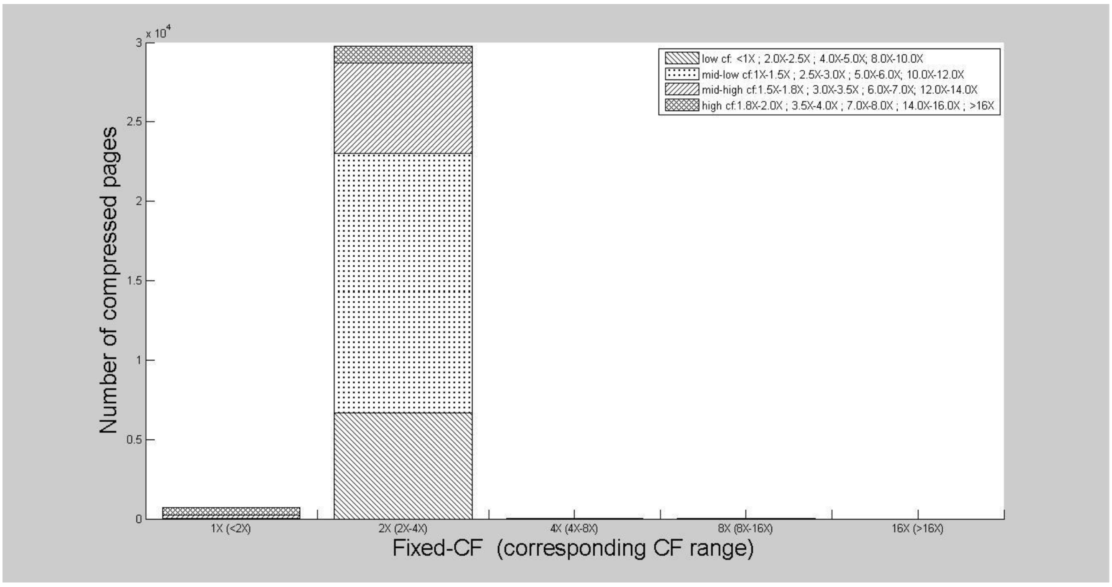
mcf -p1-g4-16384	6.7911	3.9833		165(0.0433%), 976(0.2561%), 379890(99.7003%), 1(0.0003%), 0(0%)
mcf -p2-g2-128	2.9761	2.0000	428292	2(0.0005%), 428289(99.9993%), 0(0%), 1(0.0002%), 0(0%)
mcf -p2-g2-8192	3.1811	1.9999		2(0.0005%), 428287(99.9988%), 2(0.0005%), 1(0.0002%), 0(0%)
mcf -p2-g4-1024	3.2348	1.9999		20(0.0047%), 428259(99.9925%), 11(0.0026%), 1(0.0002%), 0(0%)
mcf -p2-g4-16384	3.6797	2.0004		7(0.0016%), 427984(99.9283%), 299(0.0698%), 1(0.0002%), 0(0%)
astar-p1-g2-128	1.7661	1.0967	5734	4724(82.3858%), 1006(17.5445%), 4(0.0698%), 0(0%), 0(0%)
astar-p1-g2-8192	1.8644	1.1018		4690(81.7928%), 964(16.8120%), 80(1.3952%), 0(0%), 0(0%)
astar-p1-g4-1024	1.2560	1.1514		4677(81.5661%), 147(2.5637%), 910(15.8702%), 0(0%), 0(0%)
astar-p1-g4-16384	2.0749	1.4850		2381(41.5242%), 2440(42.5532%), 913(15.9226%), 0(0%), 0(0%)
astar-p2-g2-128	5.7936	3.7588	29966	261(0.8710%), 4205(14.0326%), 19368(64.6333%), 6132(20.4632%), 0(0%)
astar-p2-g2-8192	6.4816	4.8961		124(0.4138%), 3591(11.9836%), 7260(24.2275%), 18991(63.3752%), 0(0%)
astar-p2-g4-1024	6.2595	4.4217		270(0.9010%), 3907(13.0381%), 10624(35.4535%), 15163(50.6007%), 2(0.0067%)
astar-p2-g4-16384	7.7442	5.2985		249(0.8309%), 3103(10.3551%), 3976(13.2684%), 22627(75.5089%), 11(0.0367%)
bzip2-p1-g2-128	1.3834	1.0054	1276	1262(98.9028%), 14(1.0972%),

				0(0%), 0(0%), 0(0%)
bzip2-p1-g2-8192	1.3564	0.9961		1262(98.9028%), 14(1.0972%), 0(0%), 0(0%), 0(0%)
bzip2-p1-g4-1024	0.9856	1.0097		1257(98.5110%), 4(0.3135%), 7(0.5486%), 8(0.6270%), 0(0%)
bzip2-p1-g4-16384	0.9875	0.9853		1257(98.5110%), 5(0.3918%), 14(1.0972%), 0(0%), 0(0%)
bzip2-p2-g2-128	0.9568	1.0029	2317	2304(99.4389%), 11(0.4748%), 2(0.0863%), 0(0%), 0(0%)
bzip2-p2-g2-8192	1.0035	0.9959		2312(99.7842%), 5(0.2158%), 0(0%), 0(0%), 0(0%)
bzip2-p2-g4-1024	0.9811	1.0024		2305(99.4821%), 7(0.3021%), 3(0.1295%), 2(0.0863%), 0(0%)
bzip2-p2-g4-16384	0.9820	0.9897		2303(99.3958%), 10(0.4316%), 4(0.1726%), 0(0%), 0(0%)
average	3.1018	2.2134		

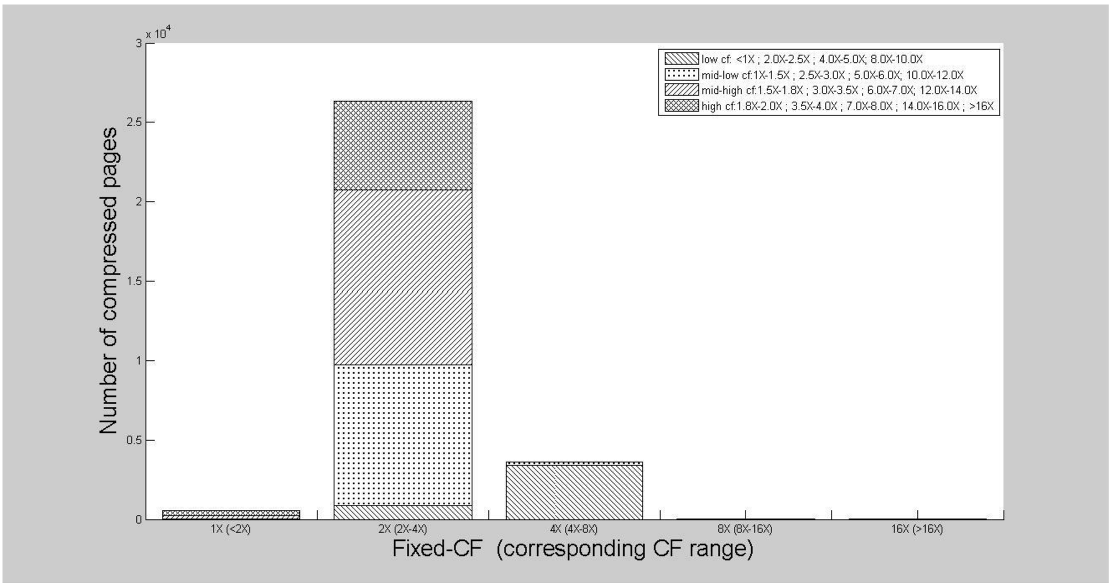
Figure 4.24 to Figure 4.32 and Table 4.17 to Table 4.25 show the variation of page compressibility due to different VFT sizes and granularities for footprint compression of 5 applications.



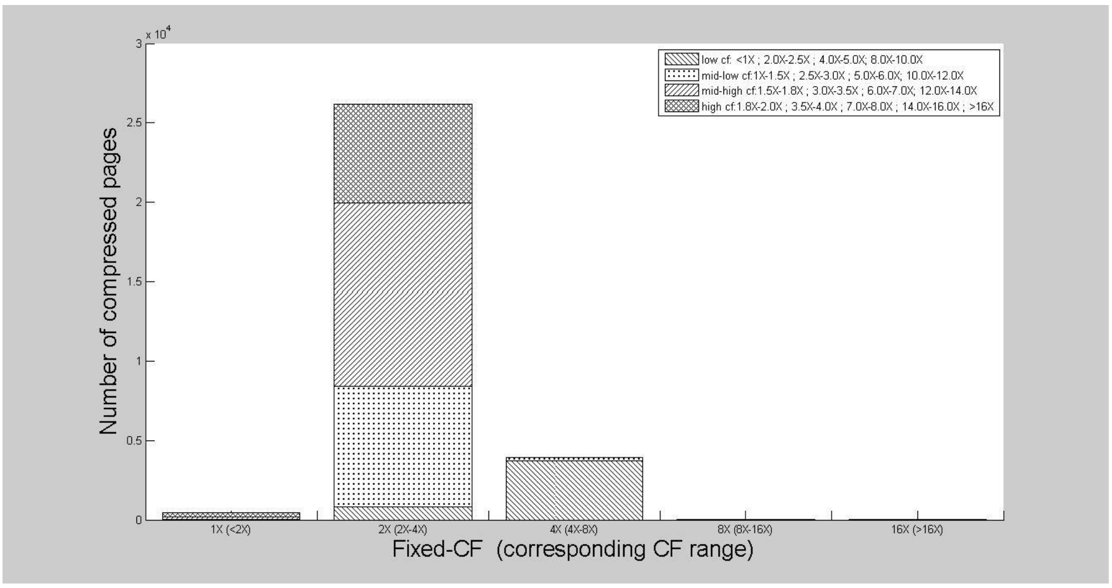
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

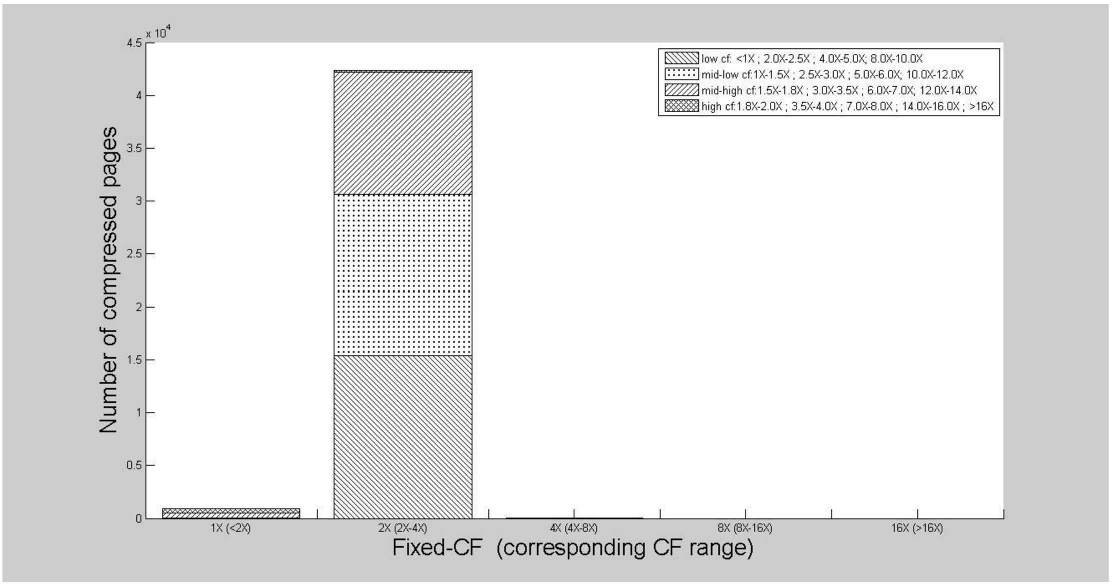
Figure 4.24: *Compressibility distribution of pages in xalan phase 1.*

More details about the fixed-size compression in *xalan* phase 1 are shown in Table 4.17.

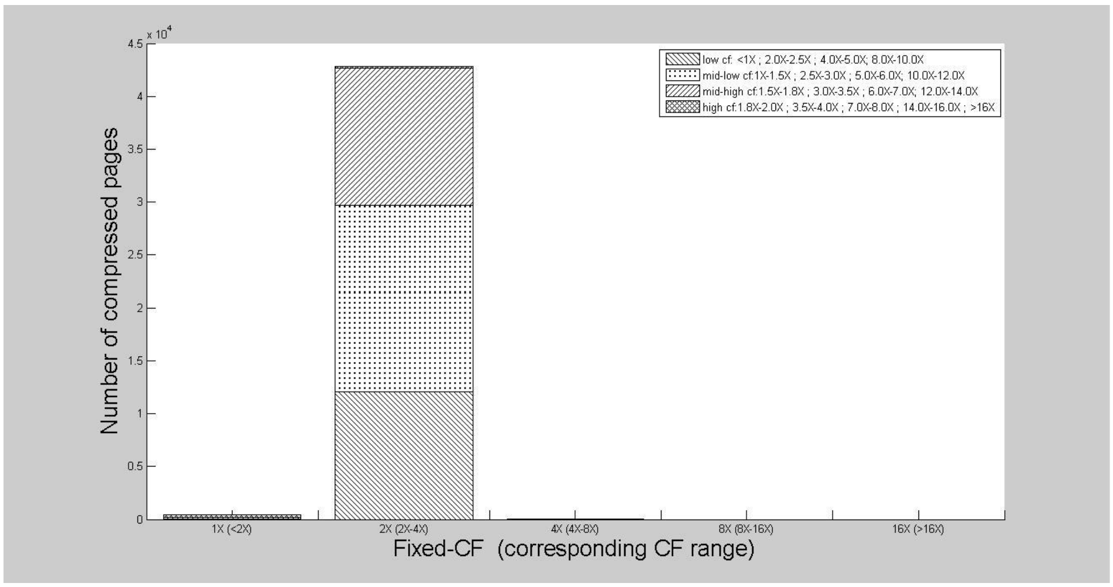
Table 4.17 *CF distribution of all compressed pages for xalan phase 1.*

Number of compressed pages	**Granularity -VFT size**	**Fixed-size CF option**	**Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high**	**CF$_{fixed\text{-}size}$**	**CF$_{1.5+3X}$**
30529	G2-128	1X	19(1.88%), 12(1.19%), 601(59.56%), 377(37.36%)	1.9371	2.0845
		2X	10414(35.32%), 14345(48.65%), 4173(14.15%), 555(1.88%)		
		4X	17(68.00%), 4(16.00%), 2(8.00%), 2(8.00%)		
		8X	2(25.00%), 2(25.00%), 3(37.50%), 1(12.50%)		
		16X	0(0%)		
	G2-8192	1X	19(2.70%), 11(1.56%), 234(33.19%), 441(62.55%)	1.9549	2.1404
		2X	6676(22.42%), 16336(54.86%), 5675(19.06%), 1092(3.67%)		

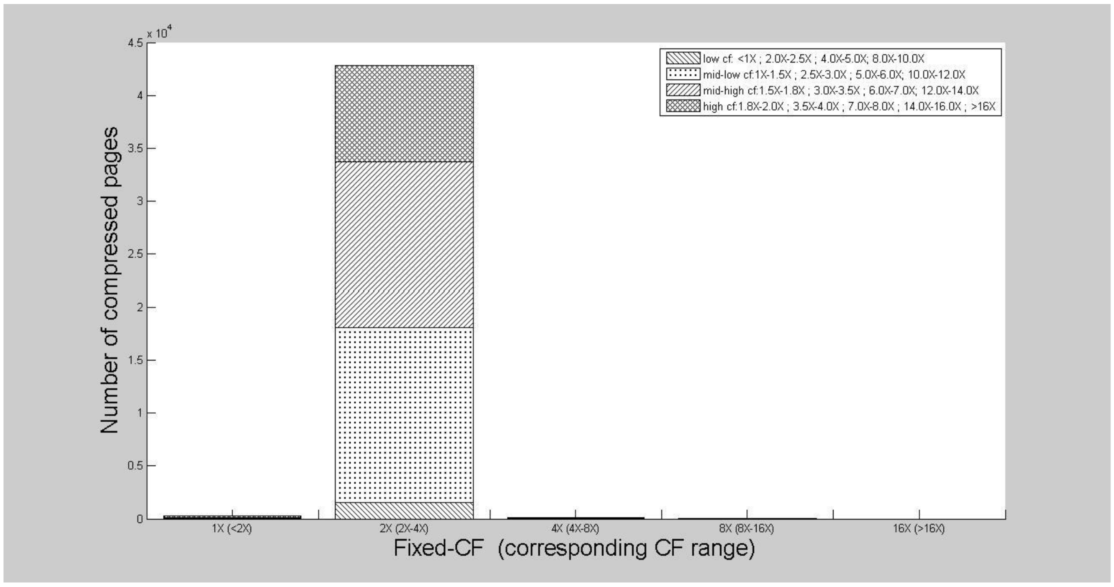
		4X	25(67.57%), 8(21.62%), 2(5.41%), 2(5.41%)		
		8X	2(25.00%), 2(25.00%), 2(25.00%), 2(25.00%)		
		16X	0(0%)		
	G4-1024	1X	19(3.38%), 8(1.42%), 226(40.21%), 309(54.98%)	2.0864	2.6114
		2X	863(3.28%), 8879(33.75%), 10970(41.69%), 5600(21.28%)		
		4X	3429(94.10%), 209(5.74%), 4(0.11%), 2(0.05%)		
		8X	3(50.00%), 0(0%), 3(50.00%), 0(0%)		
		16X	5(100.00%)		
	G4-16384	1X	2(0.47%), 21(4.88%), 161(37.44%), 246(57.21%)	2.1015	2.6713
		2X	801(3.06%), 7599(29.04%), 11525(44.04%), 6243(23.86%)		
		4X	3709(94.62%), 202(5.15%), 8(0.20%), 1(0.03%)		
		8X	3(50.00%), 0(0%), 2(33.33%), 1(16.67%)		
		16X	5(100.00%)		



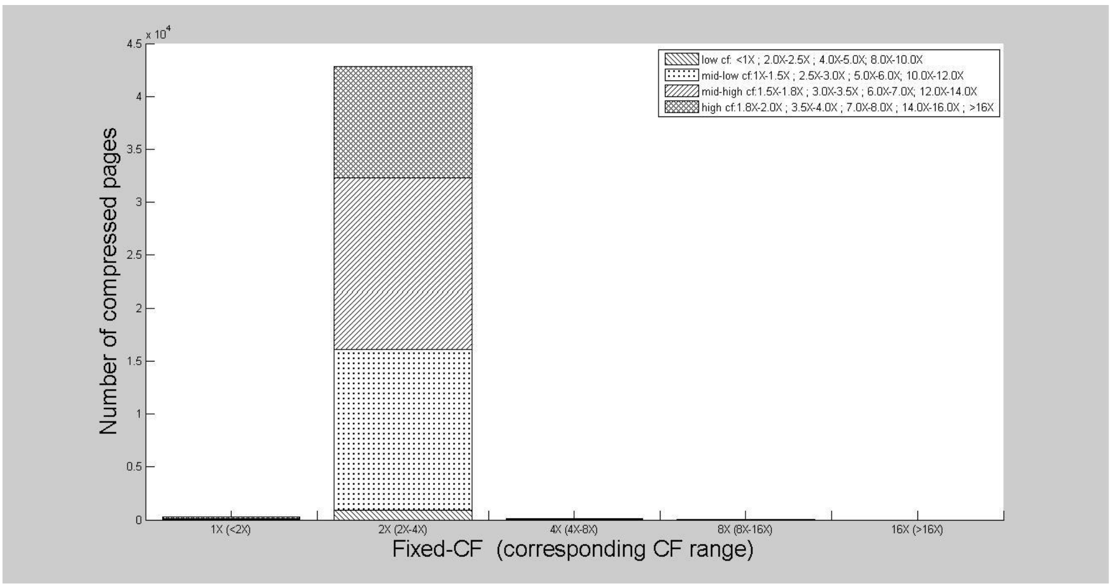
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

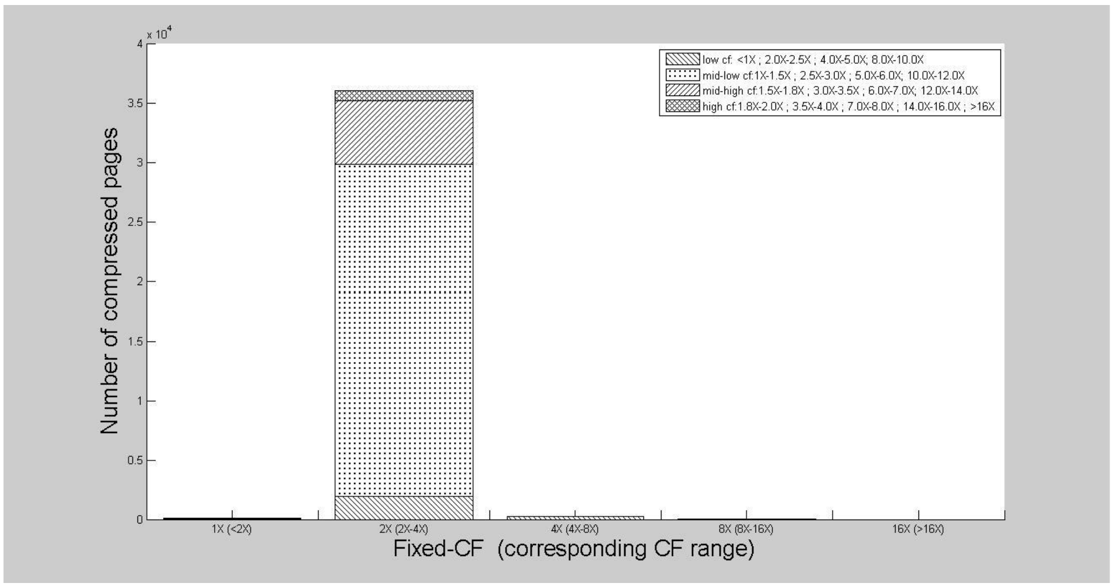
Figure 4.25: *Compressibility distribution of pages in xalan phase2.*

More details about the fixed-size compression in *xalan* phase 2 are shown in Table 4.18.

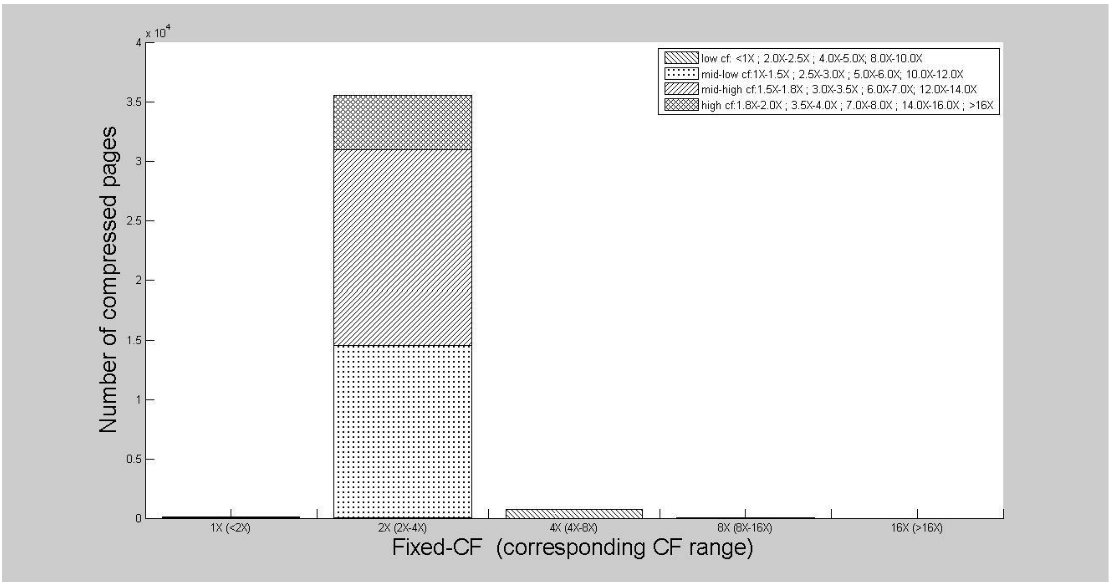
Table 4.18 *CF distribution of all compressed pages for xalan phase 2.*

Number of compressed pages	**Granularity -VFT size**	**Fixed-size CF option**	**Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high**	**$CF_{fixed-size}$**	**$CF_{1.5+3X}$**
43283	G2-128	1X	9(0.96%), 12(1.27%), 475(50.42%), 446(47.35%)	1.9575	2.1800
		2X	15390(36.35%), 15252(36.03%), 11573(27.34%), 119(0.28%)		
		4X	1(14.29%), 3(42.86%), 2(28.57%), 1(14.29%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	10(2.14%), 9(1.93%), 186(39.83%), 262(56.10%)	1.9781	2.2143
		2X	12068(28.20%), 17623(41.18%), 12996(30.37%), 105(0.25%)		

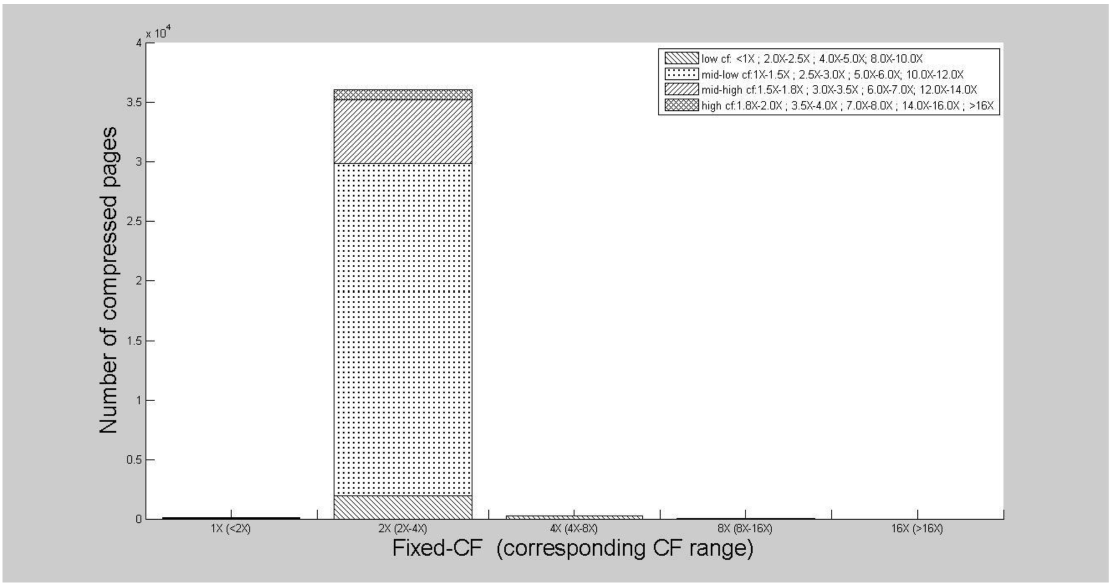
		4X	17(70.83%), 4(16.67%), 1(4.17%), 2(8.33%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	9(2.87%), 10(3.18%), 99(31.53%), 196(62.42%)	1.9884	2.4663
		2X	1571(3.67%), 16566(38.67%), 15585(36.38%), 9117(21.28%)		
		4X	12(9.60%), 1(0.80%), 111(88.80%), 1(0.80%)		
		8X	3(60.00%), 1(20.00%), 1(20.00%), 0(0%)		
		16X	0(0%)		
	G4-16384	1X	8(2.74%), 5(1.71%), 70(23.97%), 209(71.58%)	1.9868	2.5123
		2X	926(2.16%), 15161(35.38%), 16175(37.74%), 10593(24.72%)		
		4X	18(13.85%), 1(0.77%), 111(85.38%), 0(0%)		
		8X	4(66.67%), 0(0%), 2(33.33%), 0(0%)		
		16X	0(0%)		



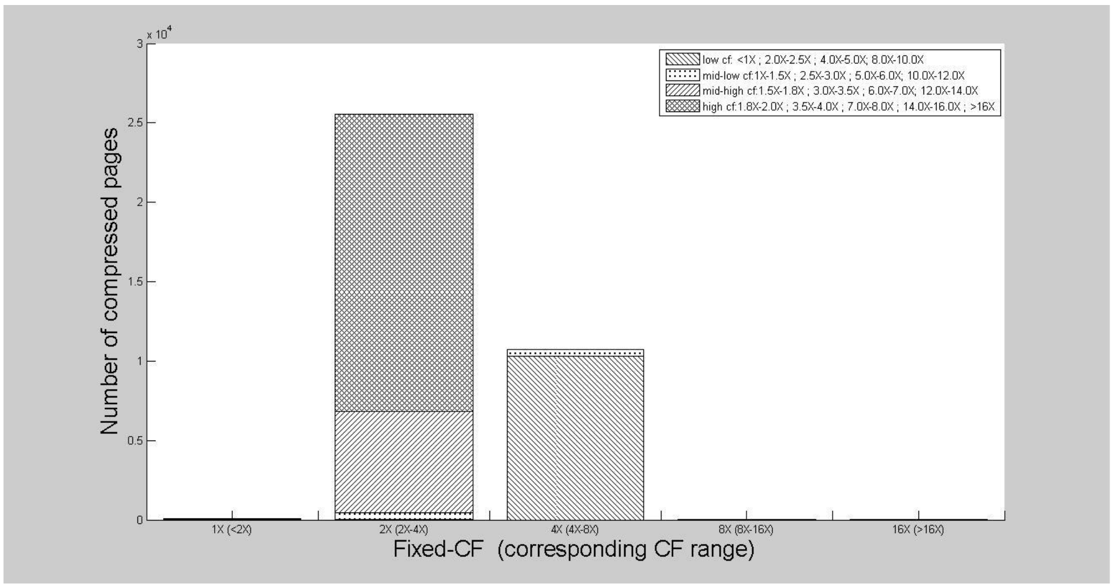
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

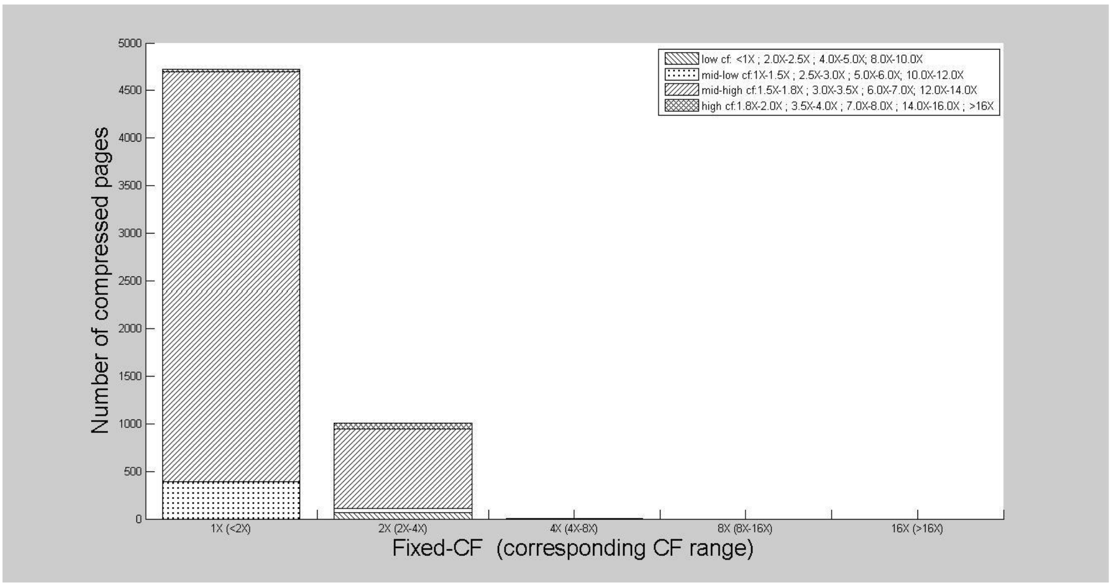
Figure 4.26: *Compressibility distribution of pages in omnetpp phase2.*

More details about the fixed-size compression in *omnetpp* phase 2 are shown in Table 4.19.

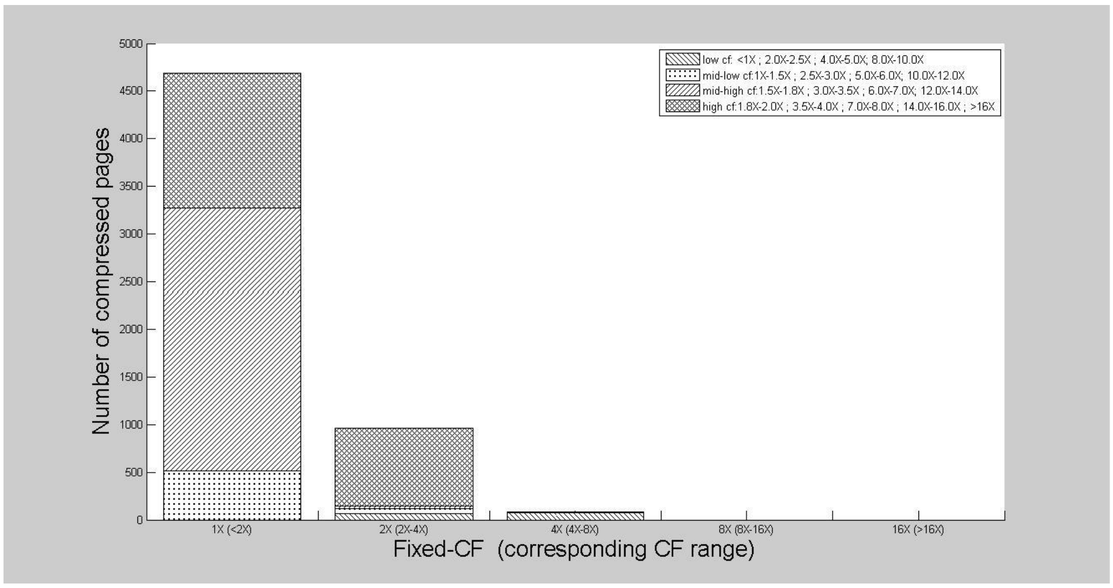
Table 4.19 *CF distribution of all compressed pages for omnetpp phase 2.*

Number of compressed pages	Granularity -VFT size	Fixed-size CF option	Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high	$CF_{fixed-size}$	$CF_{1.5+3X}$
36360	G2-128	1X	7(6.54%), 14(13.08%), 86(80.37%), 0(0%)	2.0004	2.1245
		2X	1926(5.35%), 27905(77.46%), 5327(14.79%), 865(2.40%)		
		4X	226(99.12%), 1(0.44%), 0(0%), 1(0.44%)		
		8X	0(0%), 1(50.00%), 1(50.00%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	9(11.54%), 10(12.82%), 3(3.85%), 56(71.79%)	2.0155	2.5037
		2X	32(0.09%), 14510(40.85%), 16388(46.14%), 4590(12.92%)		
		4X	758(99.87%), 1(0.13%), 0(0%), 0(0%)		

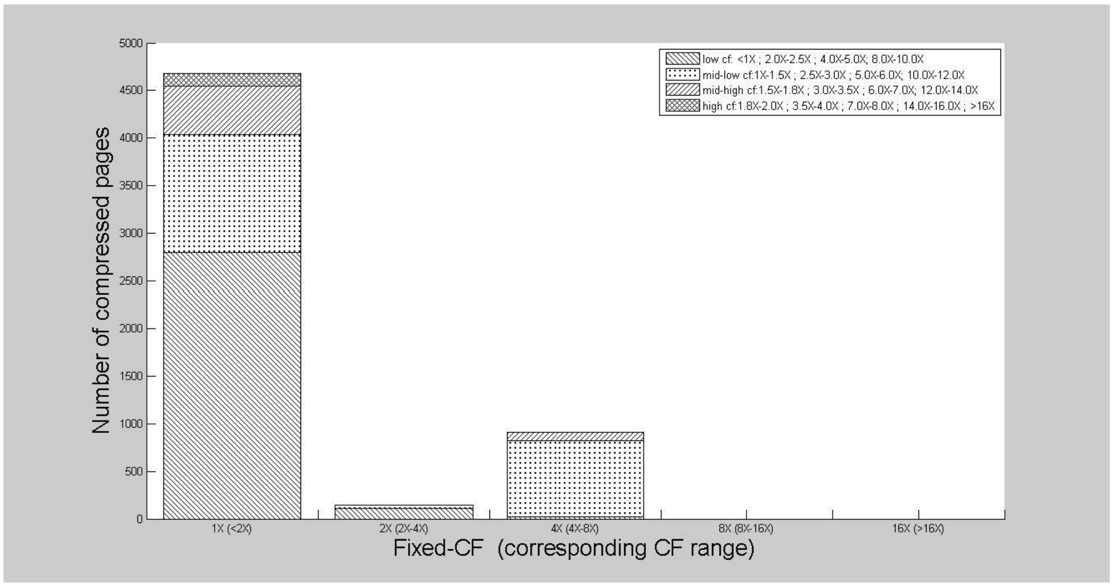
		8X	1(33.33%), 1(33.33%), 1(33.33%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	7(6.54%), 15(14.02%), 1(0.93%), 84(78.50%)	2.1999	3.0526
		2X	3(0.01%), 1904(6.47%), 8996(30.57%), 18523(62.95%)		
		4X	6470(94.81%), 352(5.16%), 2(0.03%), 0(0%)		
		8X	1(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	2(100.00%)		
	G4-16384	1X	7(9.33%), 15(20.00%), 36(48.00%), 17(22.67%)	2.3368	3.2019
		2X	12(0.05%), 411(1.61%), 6435(25.21%), 18670(73.14%)		
		4X	10334(96.09%), 418(3.89%), 2(0.02%), 0(0%)		
		8X	1(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	2(100.00%)		



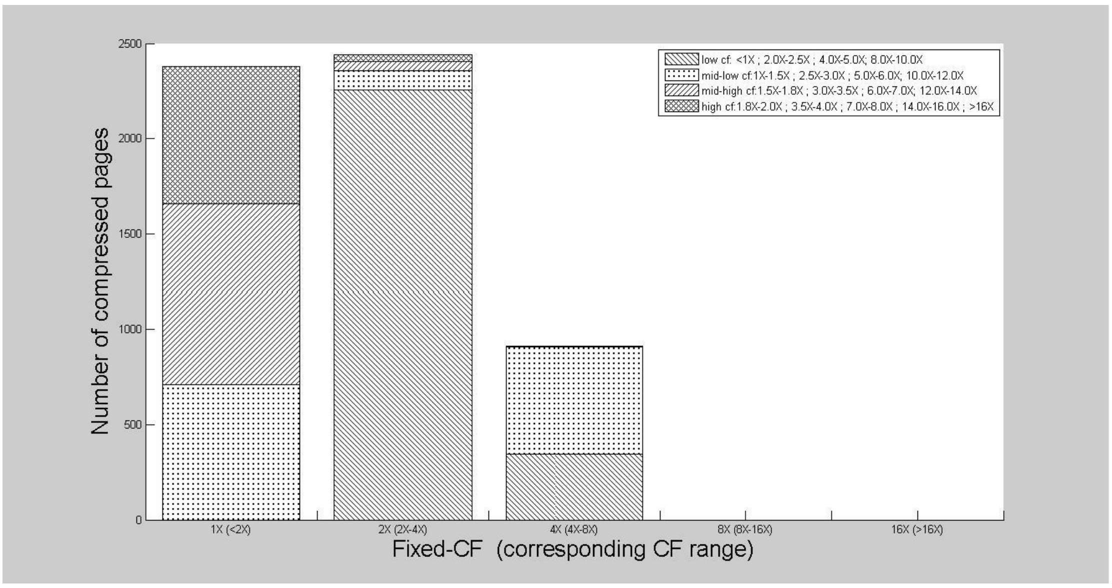
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

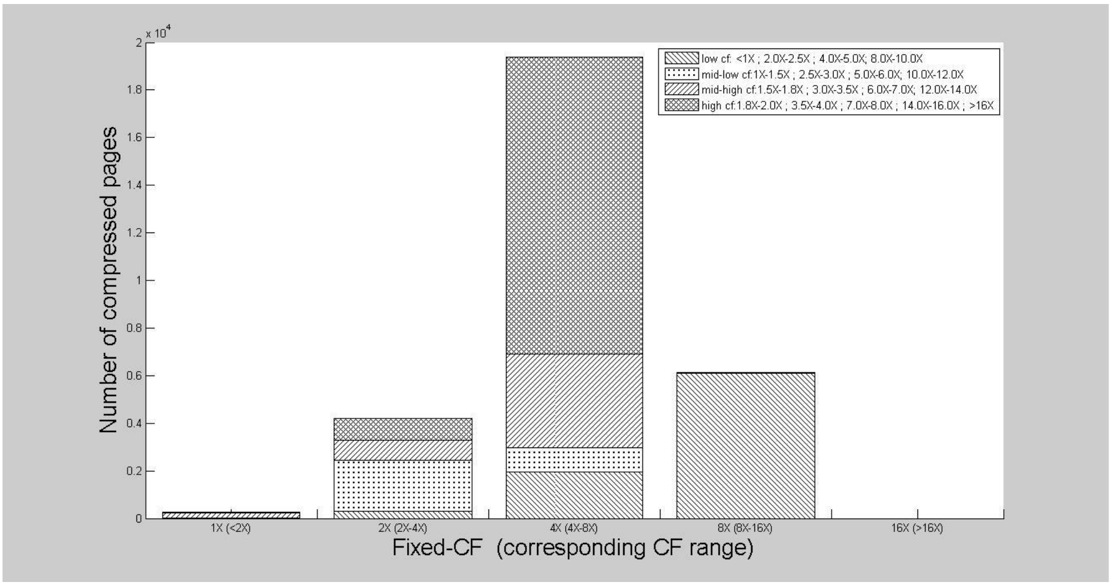
Figure 4.27: *Compressibility distribution of pages in astar phase1.*

More details about the fixed-size compression in *xalan* phase 1 are shown in Table 4.20.

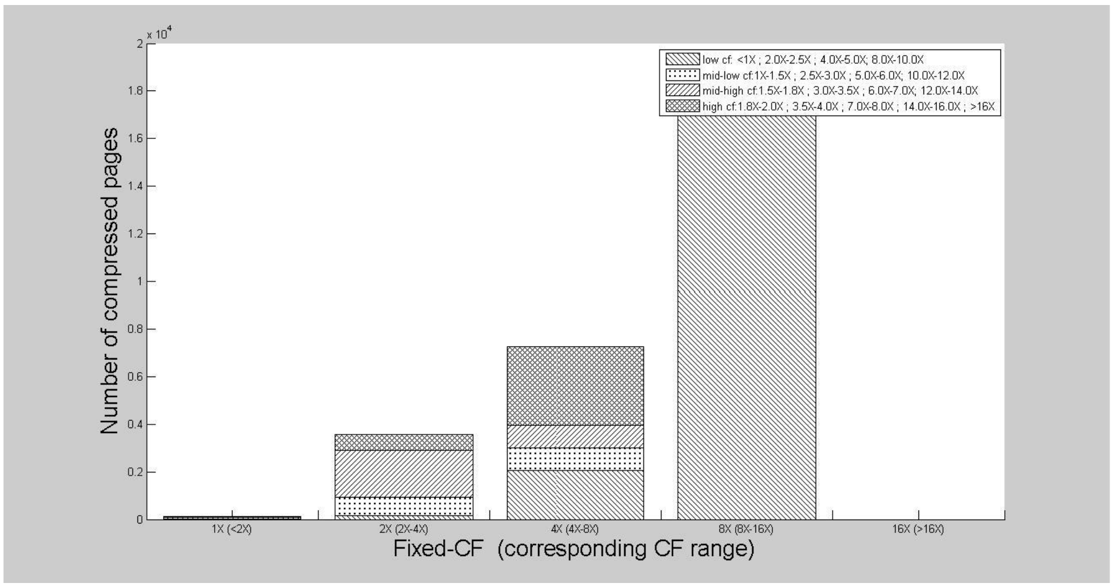
Table 4.20 *CF distribution of all compressed pages for astar phase 1.*

Number of compressed pages	Granularity -VFT size	Fixed-size CF option	Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high	$CF_{fixed\text{-}size}$	$CF_{1.5+3X}$
5734	G2-128	1X	0(0%), 388(8.21%), 4309(91.22%), 27(0.57%)	1.0967	1.5783
		2X	63(6.26%), 49(4.87%), 838(83.30%), 56(5.57%)		
		4X	1(25.00%), 3(75.00%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	0(0%), 519(11.07%), 2754(58.72%), 1417(30.21%)	1.1018	1.5616
		2X	67(6.95%), 51(5.29%), 27(2.80%), 819(84.96%)		

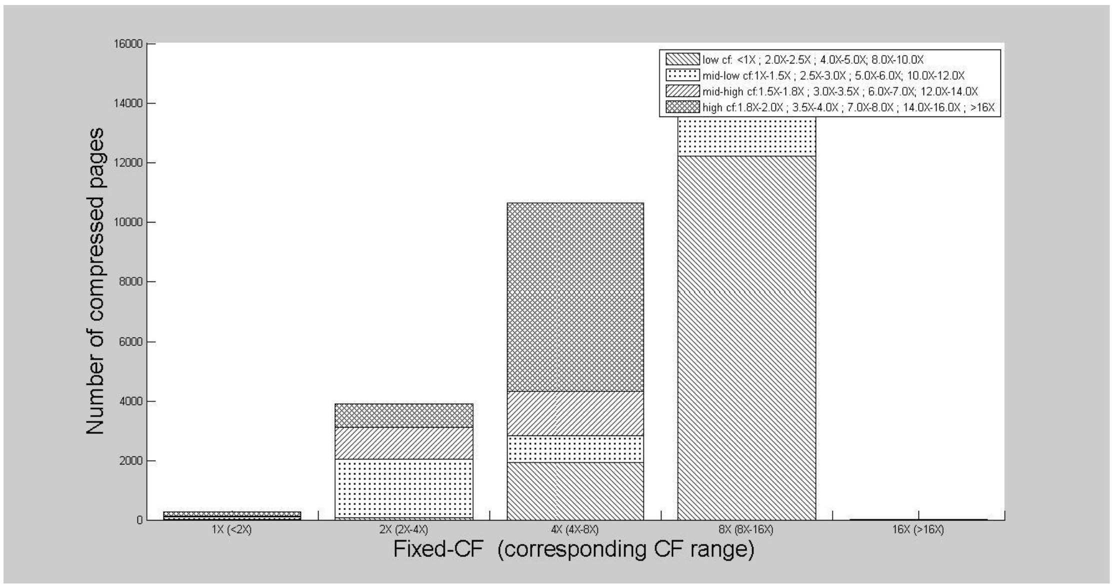
		4X	77(96.25%), 3(3.75%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	2796(59.78%), 1241(26.53%), 509(10.88%), 131(2.80%)	1.1514	1.2031
		2X	108(73.47%), 36(24.49%), 3(2.04%), 0(0%)		
		4X	22(2.42%), 805(88.46%), 80(8.79%), 3(0.33%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-16384	1X	0(0%), 709(29.78%), 947(39.77%), 725(30.45%)	1.4850	1.7430
		2X	2257(92.50%), 100(4.10%), 49(2.01%), 34(1.39%)		
		4X	344(37.68%), 566(61.99%), 3(0.33%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		



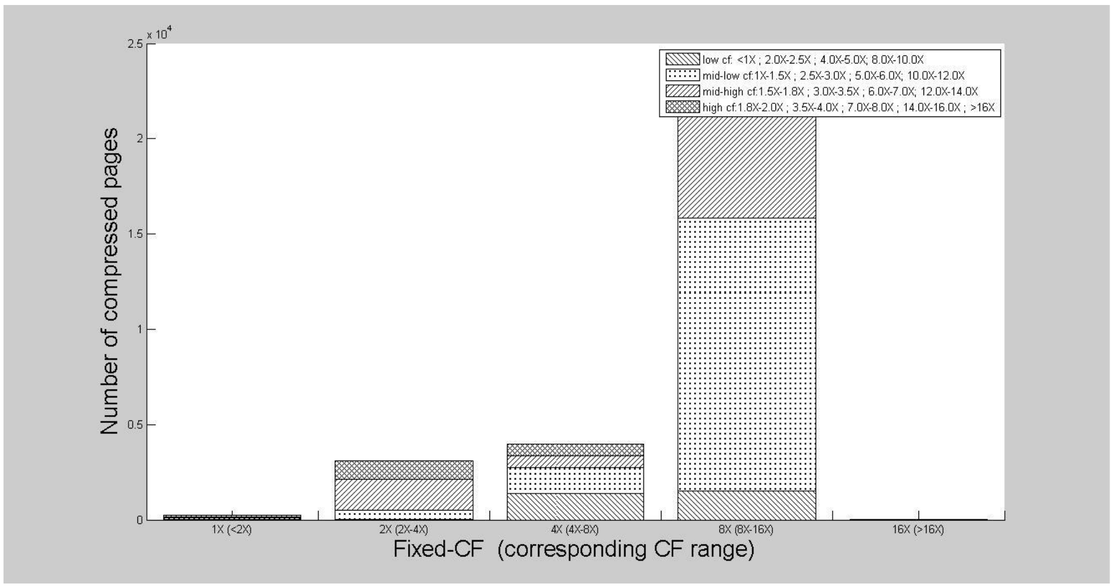
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

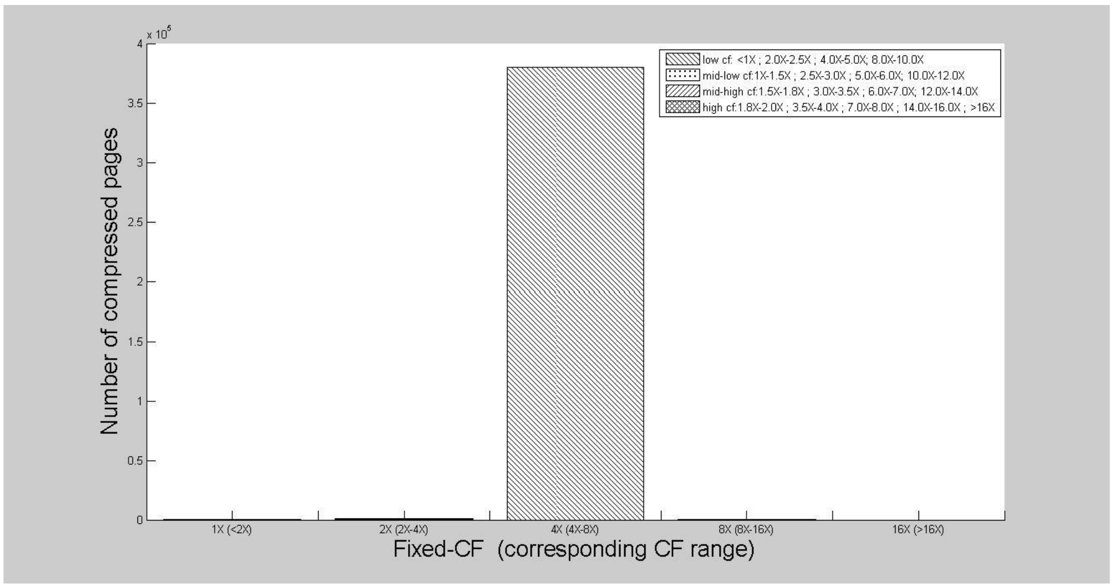
Figure 4.28: *Compressibility distribution of pages in astar phase2.*

More details about the fixed-size compression in *astar* phase 2 are shown in Table 4.21.

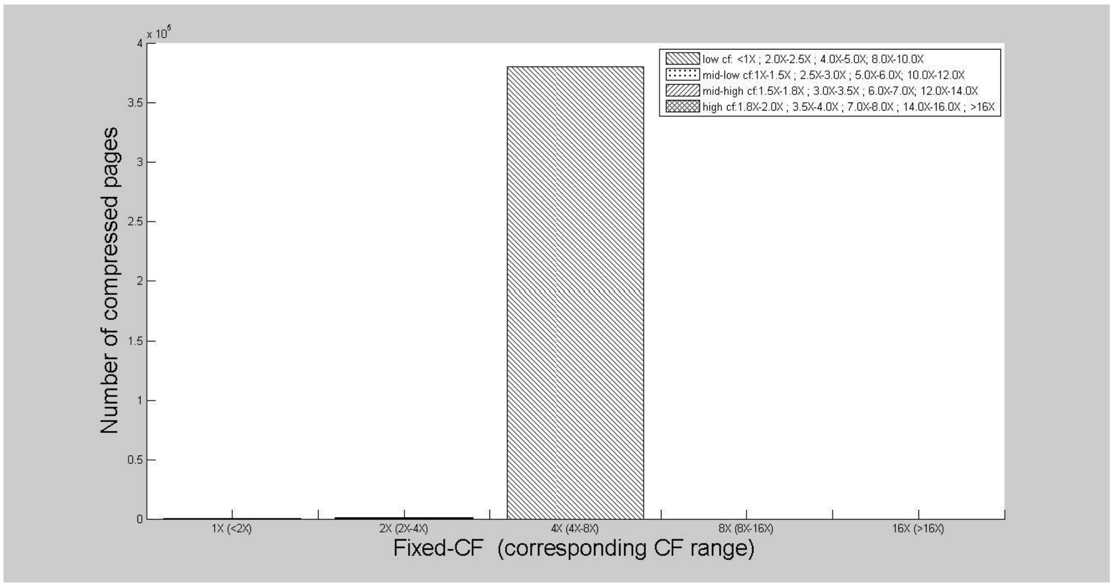
Table 4.21 *CF distribution of all compressed pages for astar phase 2.*

Number of compressed pages	Granularity -VFT size	Fixed-size CF option	Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high	$CF_{fixed\text{-}size}$	$CF_{1.5+3X}$
29966	G2-128	1X	3(1.15%), 23(8.81%), 198(75.86%), 37(14.18%)	3.7588	3.9421
		2X	308(7.32%), 2144(50.99%), 830(19.74%), 923(21.95%)		
		4X	1937(10.00%), 1031(5.32%), 3968(20.49%), 12432(64.19%)		
		8X	6115(99.72%), 15(0.24%), 1(0.02%), 1(0.02%)		
		16X	0(0%)		
	G2-8192	1X	1(0.81%), 13(10.48%), 74(59.68%), 36(29.03%)	4.8961	5.3106

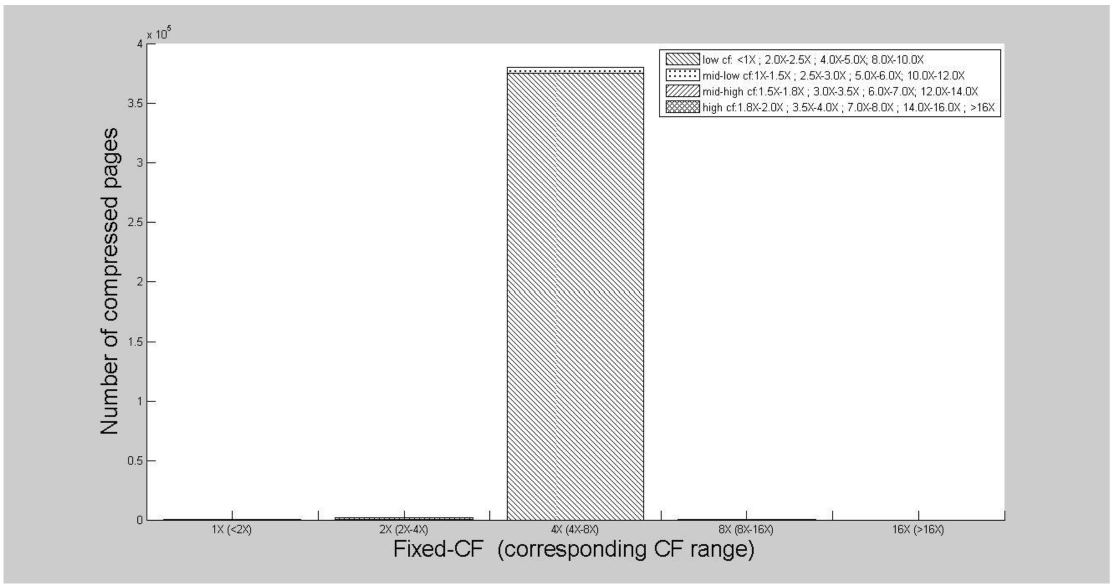
		2X	151(4.20%), 795(22.14%), 1957(54.50%), 688(19.16%)		
		4X	2052(28.26%), 974(13.42%), 924(12.73%), 3310(45.59%)		
		8X	18613(98.01%), 376(1.98%), 1(0.01%), 1(0.01%)		
		16X	0(0%)		
	G4-1024	1X	4(1.48%), 108(40.00%), 13(4.81%), 145(53.70%)	4.4217	4.6716
		2X	78(2.00%), 1971(50.45%), 1050(26.87%), 808(20.68%)		
		4X	1925(18.12%), 898(8.45%), 1496(14.08%), 6305(59.35%)		
		8X	12204(80.49%), 2754(18.16%), 194(1.28%), 11(0.07%)		
		16X	2(100.00%)		
	G4-16384	1X	5(2.01%), 98(39.36%), 61(24.50%), 85(34.14%)	5.2985	5.7958
		2X	24(0.77%), 461(14.86%), 1635(52.69%), 983(31.68%)		
		4X	1378(34.66%), 1366(34.36%), 624(15.69%), 608(15.29%)		
		8X	1498(6.62%), 14310(63.24%), 6382(28.21%), 437(1.93%)		
		16X	11(100.00%)		



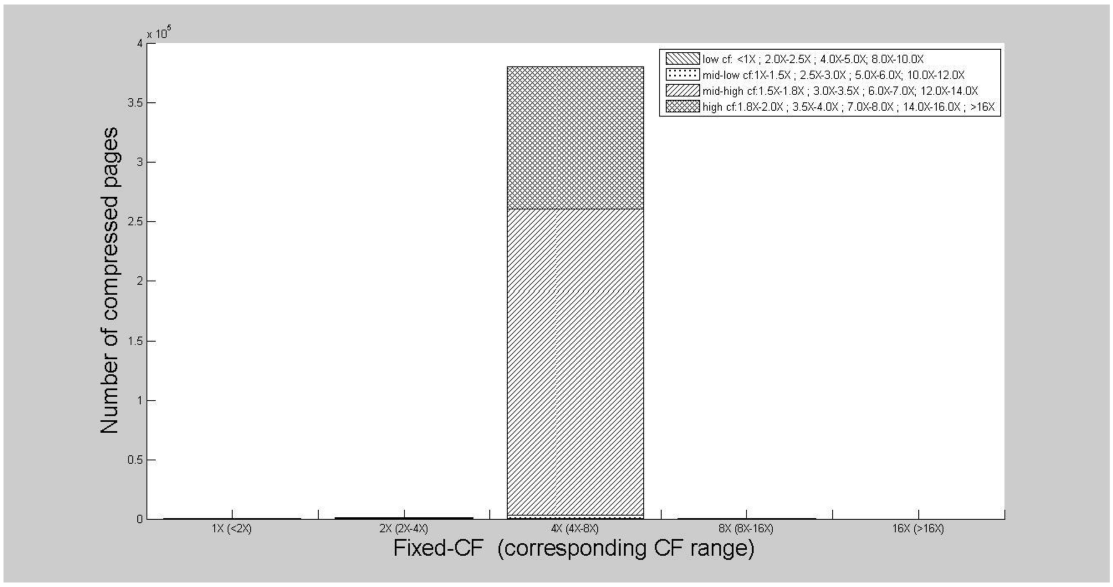
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

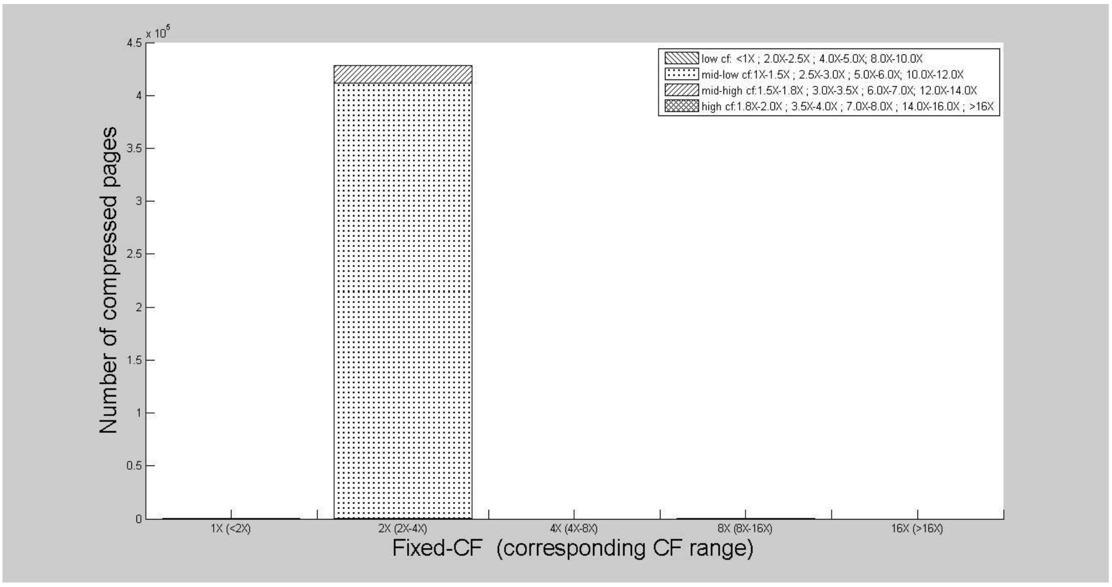
Figure 4.29: *Compressibility distribution of pages in mcf phase1.*

More details about the fixed-size compression in *mcf* phase 1 are shown in Table 4.22.

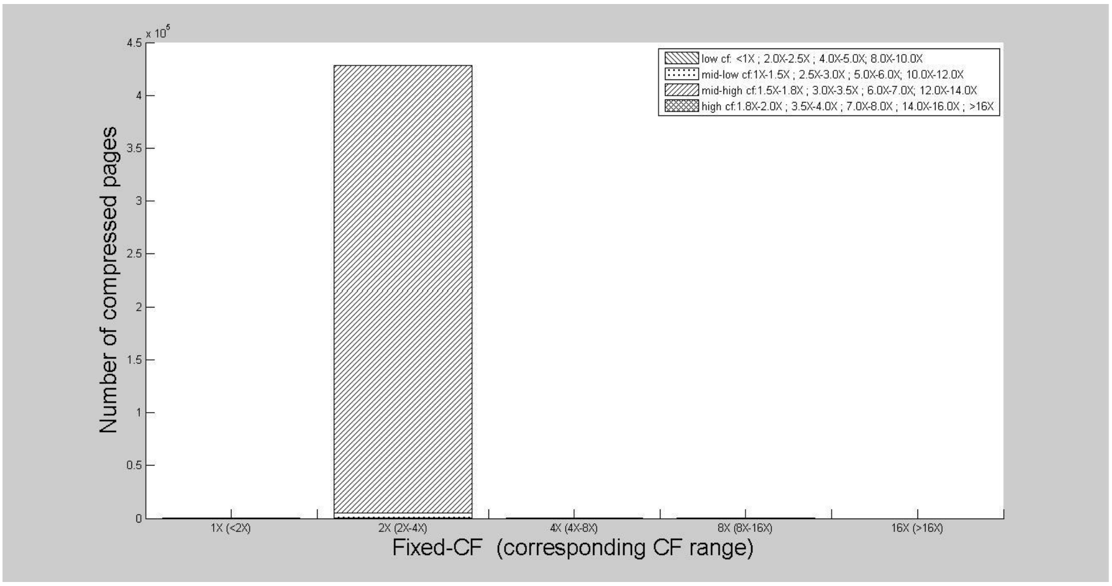
Table 4.22 *CF distribution of all compressed pages for mcf phase 1.*

Number of compressed pages	**Granularity -VFT size**	**Fixed-size CF option**	**Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high**	**CF$_{fixed\text{-}size}$**	**CF$_{1.5+3X}$**
381032	G2-128	1X	0(0%), 0(0%), 0(0%), 12(100.00%)	3.9872	3.9874
		2X	547(46.12%), 629(53.04%), 0(0%), 10(0.84%)		
		4X	379833(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	1(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	0(0%), 0(0%), 0(0%), 14(100.00%)	3.9868	3.9874
		2X	397(33.76%), 716(60.88%), 61(5.19%), 2(0.17%)		
		4X	379841(100.00%), 0(0%), 0(0%), 1(0.00%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	0(0%), 0(0%), 0(0%), 1(100.00%)	3.9850	3.9919
		2X	410(28.79%), 21(1.47%), 0(0%), 993(69.73%)		
		4X	374872(98.75%), 4734(1.25%), 0(0%), 0(0%)		
		8X	0(0%), 1(100.00%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-16384	1X	0(0%), 0(0%), 0(0%), 165(100.00%)	3.9833	3.9905
		2X	266(27.25%), 1(0.10%), 30(3.07%), 679(69.57%)		
		4X	52(0.01%), 3239(0.85%), 256592(67.54%), 120007(31.59%)		
		8X	0(0%), 1(100.00%), 0(0%), 0(0%)		

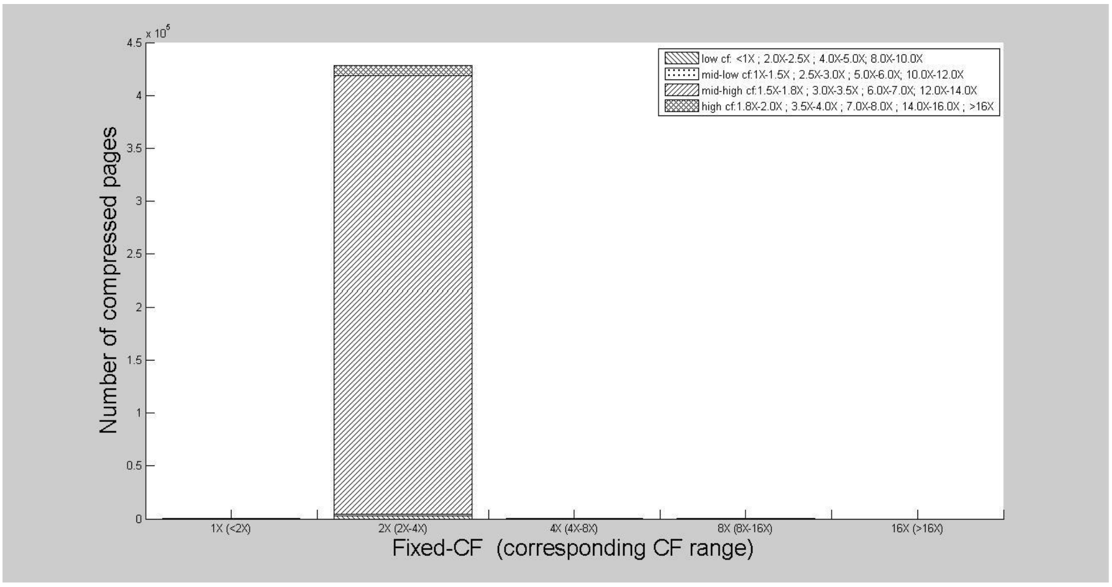
		16X	0(0%)		



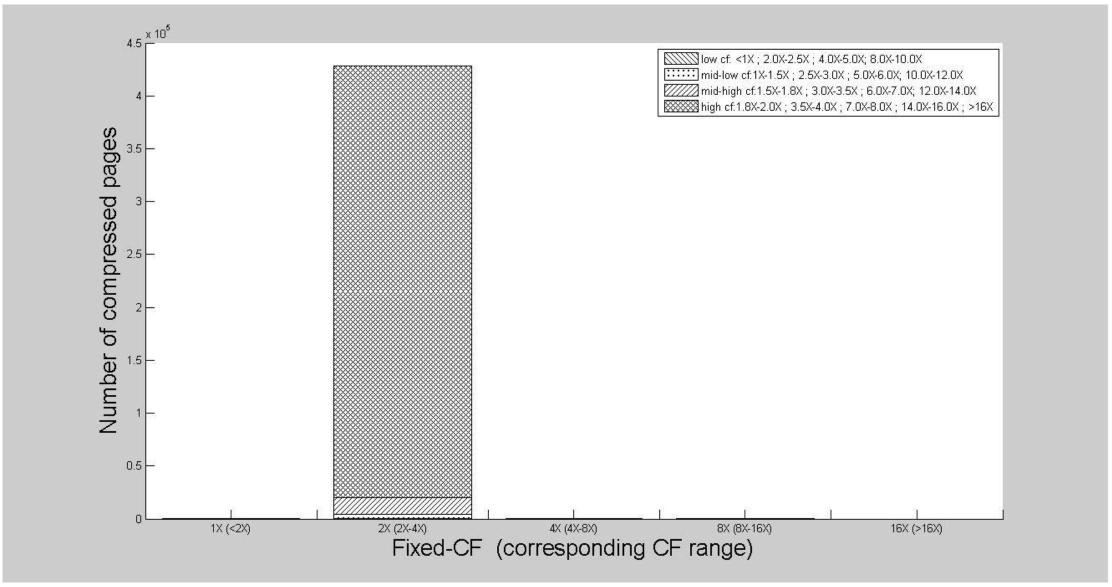
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



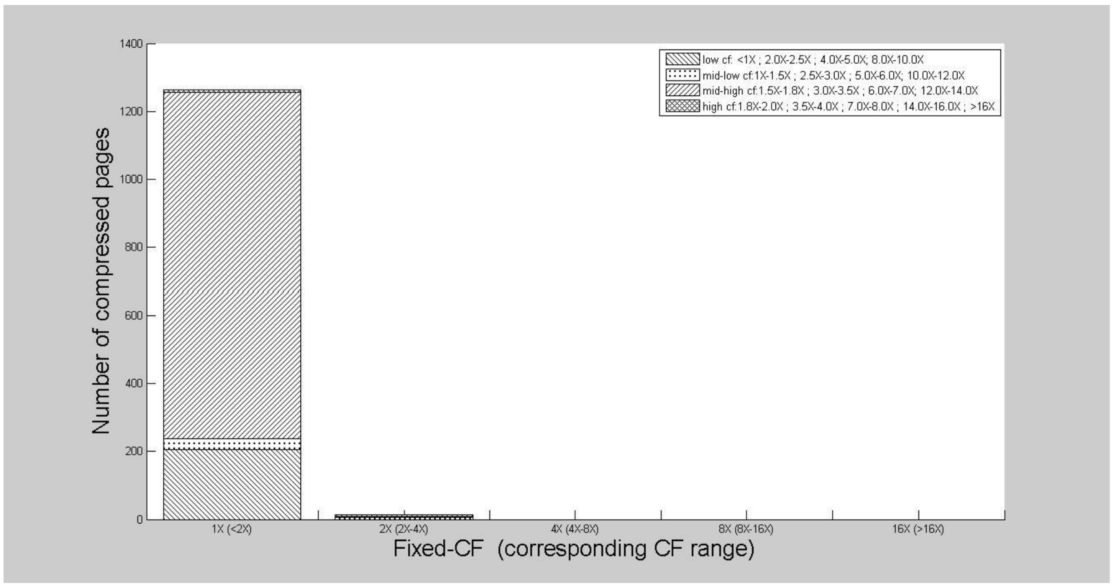
(d) 4-byte granularity, VFT-16384-mfv

Figure 4.30: *Compressibility distribution of pages in mcf phase2.*

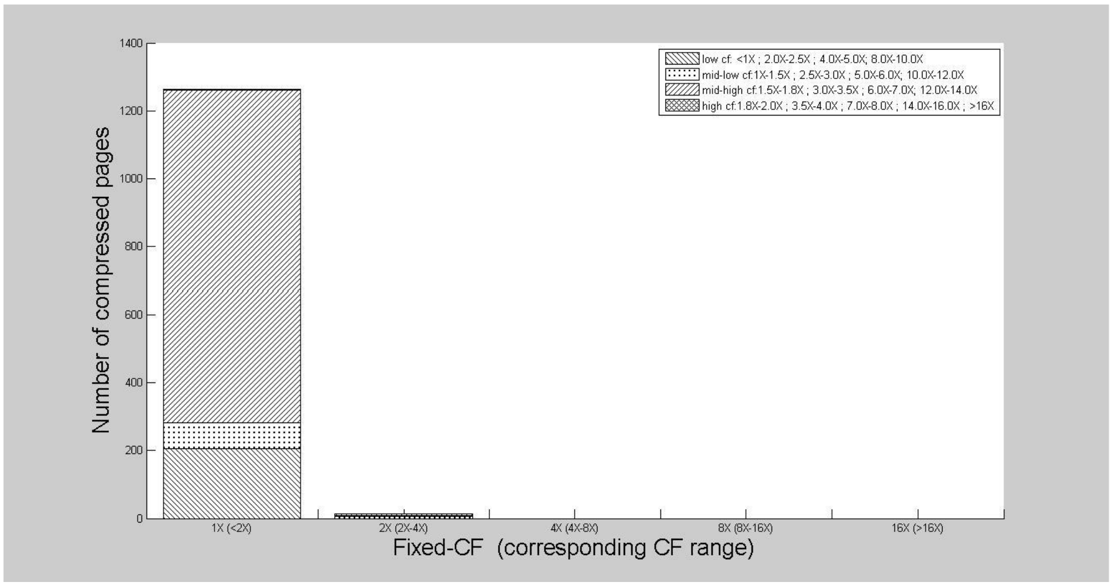
More details about the fixed-size compression in *mcf* phase 2 are shown in Table 4.23.

Table 4.23 *CF distribution of all compressed pages for mcf phase 2.*

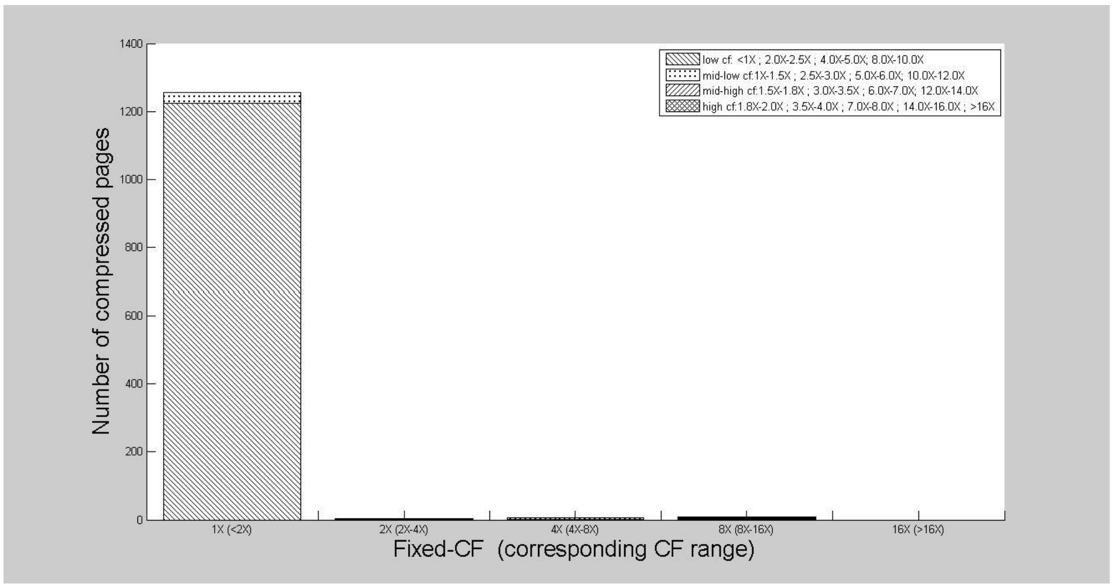
Number of compressed pages	Granularity -VFT size	Fixed-size CF option	Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high	$CF_{fixed\text{-}size}$	$CF_{1.5+3X}$
428292	G2-128	1X	0(0%), 0(0%), 2(100.00%), 0(0%)	2.0000	2.0266
		2X	710(0.17%), 410752(95.91%), 16825(3.93%), 2(0.00%)		
		4X	0(0%), 0(0%), 0(0%), 0(0%)		
		8X	1(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	0(0%), 0(0%), 2(100.00%), 0(0%)	1.9999	2.9820
		2X	576(0.13%), 4714(1.10%), 422997(98.76%), 0(0%)		
		4X	2(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	1(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	0(0%), 0(0%), 2(10.00%), 18(90.00%)	1.9999	2.9861
		2X	2725(0.64%), 1429(0.33%), 414204(96.72%), 9901(2.31%)		
		4X	11(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	1(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-16384	1X	0(0%), 0(0%), 2(28.57%), 5(71.43%)	2.0004	2.9862
		2X	676(0.16%), 3426(0.80%), 15820(3.70%), 408062(95.35%)		
		4X	299(100.00%), 0(0%), 0(0%), 0(0%)		
		8X	1(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		



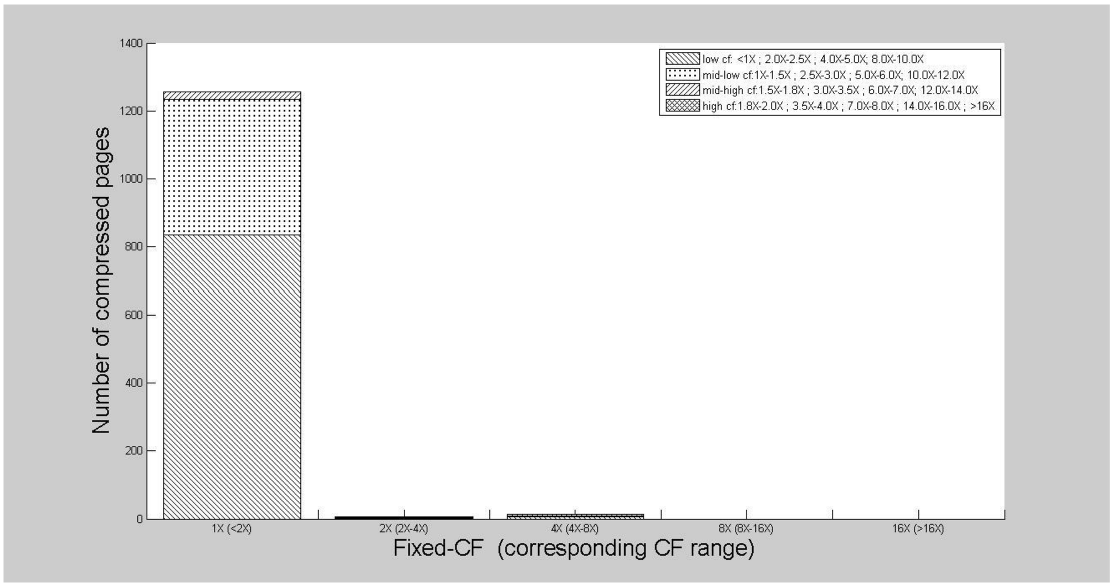
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



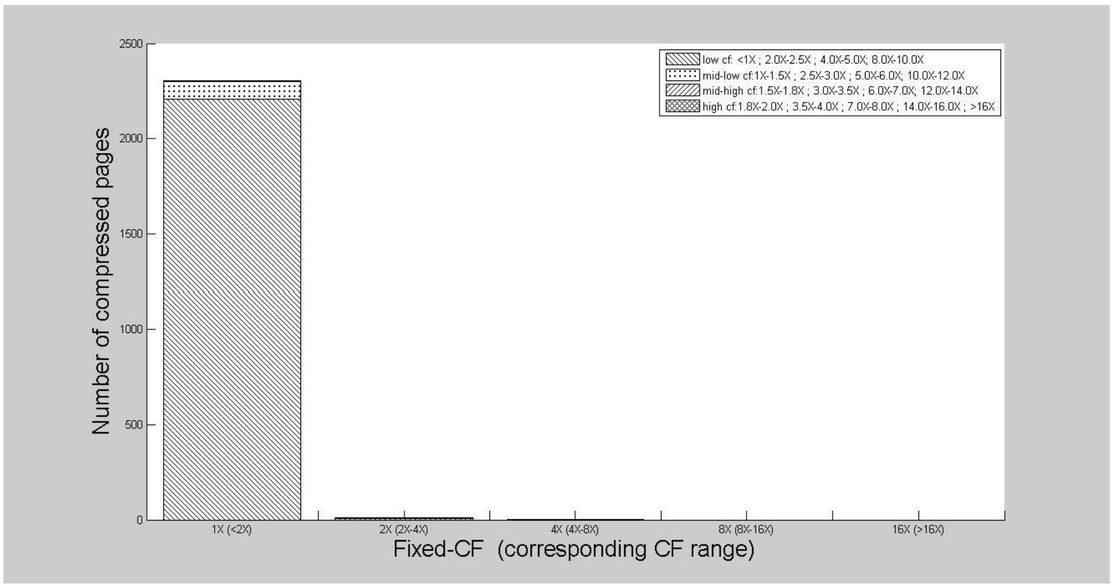
(d) 4-byte granularity, VFT-16384-mfv

Figure 4.31: *Compressibility distribution of pages in bzip2 phase1.*

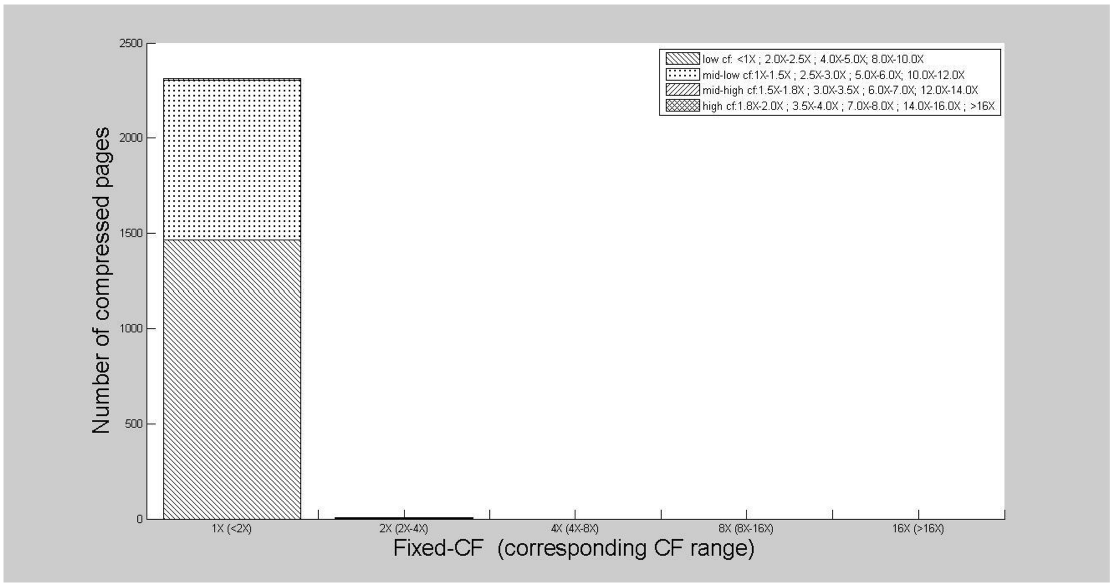
More details about the fixed-size compression in *bzip2* phase 1 are shown in Table 4.24.

Table 4.24 *CF distribution of all compressed pages for bzip2 phase 1.*

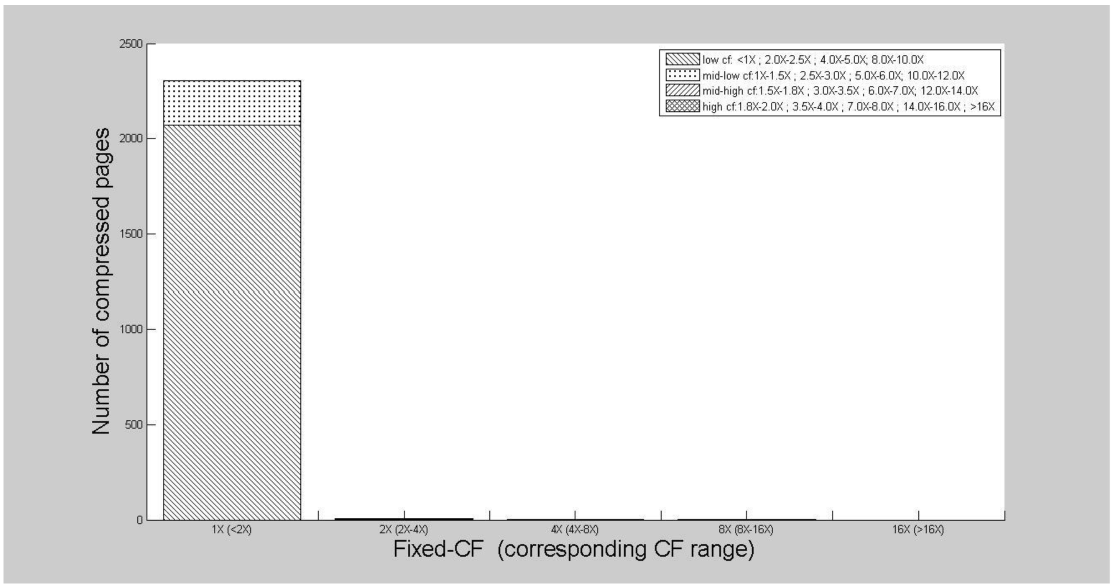
Number of compressed pages	Granularity -VFT size	Fixed-size CF option	Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high	$CF_{fixed\text{-}size}$	$C_{F1.5+3X}$
1276	G2-128	1X	207(16.40%), 30(2.38%), 1020(80.82%), 5(0.40%)	1.0054	1.3782
		2X	0(0%), 5(35.71%), 3(21.43%), 6(42.86%)		
		4X	0(0%), 0(0%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	207(16.40%), 75(5.94%), 978(77.50%), 2(0.16%)	0.9961	1.3394
		2X	0(0%), 5(35.71%), 3(21.43%), 6(42.86%)		
		4X	0(0%), 0(0%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	1224(97.37%), 31(2.47%), 1(0.08%), 1(0.08%)	1.0097	1.0106
		2X	1(25.00%), 0(0%), 3(75.00%), 0(0%)		
		4X	1(14.29%), 4(57.14%), 2(28.57%), 0(0%)		
		8X	2(25.00%), 1(12.50%), 2(25.00%), 3(37.50%)		
		16X	0(0%)		
	G4-16384	1X	835(66.43%), 399(31.74%), 22(1.75%), 1(0.08%)	0.9853	0.9915
		2X	1(20.00%), 2(40.00%), 1(20.00%), 1(20.00%)		
		4X	5(35.71%), 3(21.43%), 1(7.14%), 5(35.71%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		



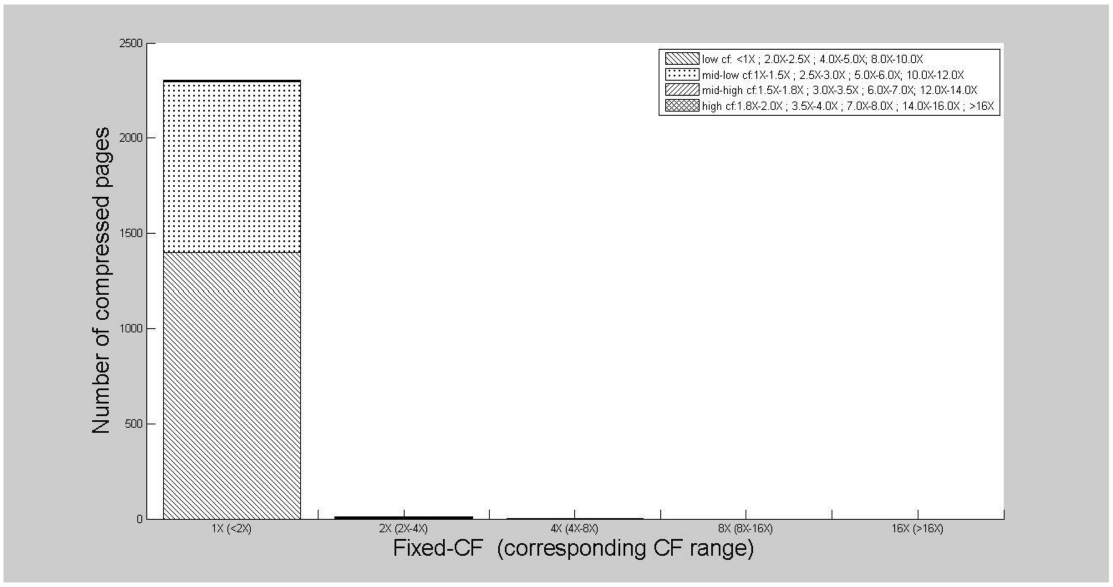
(a) 2-byte granularity, VFT-128-mfv



(b) 2-byte granularity, VFT-8192-mfv



(c) 4-byte granularity, VFT-1024-mfv



(d) 4-byte granularity, VFT-16384-mfv

Figure 4.32: *Compressibility distribution of pages in bzip2 phase2.*

More details about the fixed-size compression in *bzip2* phase 2 are shown in Table 4.25.

Table 4.25 *CF distribution of all compressed pages for bzip2 phase 2.*

Number of compressed pages	Granularity -VFT size	Fixed-size CF option	Number of pages (percentage) with actual CF in the region of low, mid-low, mid-high, high	$CF_{fixed\text{-}size}$	$CF_{1.5+3X}$
2317	G2-128	1X	2209(95.88%), 90(3.91%), 3(0.13%), 2(0.09%)	1.0029	1.0038
		2X	6(54.55%), 3(27.27%), 1(9.09%), 1(9.09%)		
		4X	0(0%), 2(100.00%), 0(0%), 0(0.00%		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G2-8192	1X	1462(63.24%), 844(36.51%), 5(0.22%), 1(0.04%)	0.9959	0.9968
		2X	3(60.00%), 2(40.00%), 0(0%), 0(0%)		
		4X	0(0%), 0(0%), 0(0%), 0(0%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-1024	1X	2071(89.85%), 233(10.11%), 1(0.04%), 0(0%)	1.0024	1.0026
		2X	4(57.14%), 2(28.57%), 1(14.29%), 0(0%)		
		4X	2(66.67%), 1(33.33%), 0(0%), 0(0%)		
		8X	2(100.00%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		
	G4-16384	1X	1396(60.62%), 899(39.04%), 6(0.26%), 2(0.09%)	0.9897	0.9910
		2X	3(30.00%), 5(50.00%), 1(10.00%), 1(10.00%)		
		4X	2(50.00%), 0(0%), 0(0%), 2(50.00%)		
		8X	0(0%), 0(0%), 0(0%), 0(0%)		
		16X	0(0%)		

The experimental result of fixed-size compression for footprint is similar as the result of fixed-size compression based on each access. On average, the compression factor by applying fixed-CF is 2.2134X while the $\mathrm{CF}_{\mathrm{with\ VFT}}$ of footprint is 3.1018X. In other words, 40.1% compressibility (i.e.,$\frac{CF_{with\,VFT}}{CF_{for\,fixed-size\,compression}} - 1$) is subtracted due to the fixed-size compression approach. By allowing extra fixed-CF options of 1.5X and 3X, the improvement of average CF of fixed-size compression is 14.48%. One interesting finding is the application *bzip2* which has a poor compression potential can benefit by applying fixed-CF for compression. That is, some pages with actual CF lower than 1X can be kept uncompressed. Thus the average CF of fixed-size compression is even higher than the $\mathrm{CF}_{\mathrm{with\ VFT}}$ for *bzip2*.

In conclusion, fixed-size compression scheme leads to more than 35% compressibility reduction. Besides, increasing VFT size or applying larger granularity contributes negligibly to the compressibility improvement due to the setting of the fixed-size compression options. Thus, the trade-off between data placement and compressibility is an important design consideration.

4.6 Conclusions

In this chapter we study and analyze the impact of different design parameters on the compressibility of Huffman-based compression scheme. We first illustrate the high compression potential of the scheme by applying Huffman coding on the footprint of seven benchmark applications. We study different ways targeting for low overhead due to metadata and the suitable granularity and VFT size options are proposed. This Chapter also discusses the impact of data variation on compressibility. We prove that the static Huffman codewords are efficient for compression over a long execution time of millions of instructions since the value distribution varies little during the execution in most cases. Finally, we evaluate the performance reduction by applying fixed-size compression scheme to better handle the memory fragmentation.

5. Conclusion

This thesis studies and discusses statistical compression algorithms, such as Huffman coding in memory compression. Several design parameters like compression granularity, VFT overhead, value distribution variation are concerned and explored in this thesis. Our experimental results show that the Huffman-based scheme has a promising compressibility and high potential of practicality in main memory compression.

5.1 Contributions and Findings

The first contribution of this thesis comes from the observation that Huffman-based compression can highly improve the utilization of memory resources. For seven applications from SPEC2006 benchmark suite, the memory footprint can be compressed by a factor of 3.86X, on average, and up to 8.92X when the metadata of Value Frequency Table (VFT) is not accounted for. This result is promising as most previous works report an ideal compression factor of less than 2X.

The thesis studies the impact of compression granularity on the compression performance. By investigating three granularity options of 1 byte (G1), 2 bytes (G2) and 4 bytes (G4), we demonstrate that G4 has the highest potential compressibility at the expense of big overhead for storing the VFT. G2 is competitive due to its small metadata and is suitable for compressing such applications with relatively low compression potentials in which the value distribution for G2 is much more concentrated than G4. G1 is found as an unsuitable compression granularity option due to its poor compression potential. That is, the maximum compression ratio for G1 is 8X which is 4 times smaller than G4 (32X).

To make the evaluation of Huffman-based compression scheme to be more practical, the metadata of VFT must be concerned. Our results show that the compressibility degrades seriously due to metadata. To address this problem, the thesis studies different VFT sizes trading high compressibility for reduced overhead. The exciting observation shows that by applying smaller VFT, most applications can achieve better compression factor than capturing all the unique values (VFT-all). In particular, the difference between $CF_{\text{with VFT}}$ and $CF_{\text{w/o VFT}}$ (considered as ideal potential compressibility) is narrowed by 32.11% for G4 compression by applying smaller VFT. Finally, the $CF_{\text{with VFT}}$ for G2 and G4 are 2.6087X and 3.1150X, respectively. Another encouraging result shows that by using really small VFT, a relatively high compression ratio is possible which implies a great potential of practicality.

Another contribution of the thesis is based on the analysis of the impact of value distribution variation on the compressibility. Our finding shows that in most cases the value distribution in memory varies little during a large execution phase (10^9 committed instructions or few hundreds of milliseconds) of application. This means that static Huffman coding can work sufficiently over a long execution time of millions of instructions without update. Besides, we present that sampling the value frequency distributions only for short phase (7M write accesses, approximately 50M committed instructions) is enough to establish representative enough and efficient Huffman coding.

At the last part of the thesis, we investigate the fixed-size compression scheme to address the fragmentation and location problems in compression. Our results show that the compressibility degrades seriously by 30%, on average, when only allowing fixed-size compressed pages. Moreover, the compressibility improvement by applying larger VFT size and granularity may be reduced by the fixed-size CF options. One promising solution is subdividing the fixed-size CF into more intervals to allow the page to be compressed to more different sizes. However, it may introduce more complexity for memory management.

In conclusion, Huffman-based compression scheme has a high potential for more efficient utilizations of main memory resources. By modifications like choosing proper granularity and VFT size, it is competitive in memory compression.

5.2 Future Work

Further interesting directions in this study are summarized as future work:

1. Impact of data variation to compression for longer execution phases.
2. More fixed-size CF options to improve the compression performance.
3. Collection of less information to build Huffman codewords (3M write accesses or even less).
4. Monitoring compressibility using different interval sizes to study the impact of sampling on monitoring.
5. Use of semi-adaptive Huffman coding for higher compressibility.

Reference

1. Wulf, W.A. and S.A. McKee, *Hitting the memory wall: implications of the obvious.* ACM SIGARCH computer architecture news, 1995. **23**(1): p. 20-24.

2. Patterson, D.A. and J.L. Hennessy, *Computer Architecture - A Quantitative Approach*. 1990.

3. Ekman, M. and P. Stenstrom. *A robust main-memory compression scheme*. in *ACM SIGARCH Computer Architecture News*. 2005. IEEE Computer Society.

4. Pekhimenko, G., T.C. Mowry, and O. Mutlu. *Linearly compressed pages: a main memory compression framework with low complexity and low latency*. in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. 2012. ACM.

5. Arelakis, A. and Stenstrom P., *A case for a value-aware cache, IEEE Computer Architecture Letters, vol 99, no. RapidPost, p.1,.* 2013.

6. Arelakis, A., *Design Considerations of Value-aware Caches.* 2013.

7. Denning, P.J., *The Locality Principle. In Communication Networks and Computer Systems (J.Barria, Ed.).* 2006: p. 43-67.

8. Patterson, D.A. and J.L. Hennessy, *Computer organization and design: the hardware/software interface*. 2008: Morgan Kaufmann.

9. Hallnor, E.G. and S.K. Reinhardt. *A unified compressed memory hierarchy*. in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. 2005. IEEE.

10. Alameldeen, A.R. and D.A. Wood, *Frequent pattern compression: A significance-based compression scheme for L2 caches.* Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep, 2004. **1500**.

11. Zhang, Y., J. Yang, and R. Gupta. *Frequent value locality and value-centric data cache design*. in *ACM SIGOPS Operating Systems Review*. 2000. ACM.

12. Yang, J., Y. Zhang, and R. Gupta. *Frequent value compression in data caches*. in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. 2000. ACM.

13. Thuresson, M., L. Spracklen, and P. Stenstrom, *Memory-link compression schemes: A value locality perspective.* Computers, IEEE Transactions on, 2008. **57**(7): p. 916-927.

14. Pekhimenko, G., V. Seshadri, O. Mutlu, T.C. Mowry, P.B. Gibbons, and M.A. Kozuch, *Base-delta-immediate compression: A practical data compression mechanism for on-chip caches*. in *Proceedings of the 21st ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012.

15. Pekhimenko, G., V. Seshadri, Y. Kim, H. Xin, O. Mutlu, M.A. Kozuch, P.B. Gibbons and T.C. Mowry, *Linearly compressed pages: a low-complexity, low-latency main memory compression framework*. in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 2013. ACM.

16. Molina, C., C. Aliagas, M. García, A. González and J. Tubella, *Non redundant data cache*. in *Proceedings of the 2003 international symposium on Low power electronics and design*. 2003. ACM.

17. Ziv, J. and A. Lempel, *A universal algorithm for sequential data compression.* Information Theory, IEEE Transactions on, 1977. **23**(3): p. 337-343.

18. Tremaine, R.B., P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, T. B. Smith and T. B. Smith, *IBM memory expansion technology (MXT).* IBM Journal of Research and Development, 2001. **45**(2): p. 271-285.

19. Sayood, K., *Introduction to data compression*. 2012: Access Online via Elsevier.

20. http://software.intel.com/sites/landingpage/pintool/docs/62732/Pin/html/.

21. https://code.google.com/p/protobuf/.

22. Salomon, D., *Variable-length codes for data compression*. Vol. 140. 2007: Springer.