SYNERGISTIC INTEGRATION OF CODE COMPRESSION AND ENCRYPTION IN EMBEDDED SYSTEMS

By

KARTIK SHRIVASTAVA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2010

To my family and friends

ACKNOWLEDGMENTS

First, I would like to thank my thesis supervisor Dr. Prabhat Mishra for providing me an opportunity to solve many interesting and complex problems, assisting me to learn new technologies and recognizing the potential in me to positively contribute in ongoing research in Embedded Systems Lab. My sincere thanks to Dr. My Thai and Dr. Alin Dobra for being my thesis committee members and providing me valuable feedback and constructive comments on my thesis. I would also like to thank all Computer and Information Science and Engineering Department faculty for offering advanced courses to augment my knowledge and inspiring me to apply those concepts to solve numerous complex issues. I would like to extend my profound gratitude to research members in Embedded System Lab who were there at all times providing me a joyous environment, listening to all my problems and for assisting me in solving them with great ease. Last but by no means the least, I would like to convey my heartfelt thanks to my family and friends who, at all times have been a great source of positive spirit, inspiration, encouraging me to take bold decisions and accomplish them successfully.

This work is partially supported by the National Science Foundation under Grant No. CNS-0915376.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

SYNERGISTIC INTEGRATION OF CODE COMPRESSION AND ENCRYPTION IN EMBEDDED SYSTEMS

By

Kartik Shrivastava

August 2010

Chair: Prabhat Kumar Mishra
Major: Computer Engineering

Embedded systems are used in a wide variety of places today, from cell phones to automobiles. Architects aim to make embedded systems more powerful and space efficient as well as secure. Code compression techniques are promising for reducing the memory requirements, whereas existing encryption techniques are widely used for application security. Code compression is traditionally used to reduce the code size by compressing the instructions with higher static frequency. However, it may produce a decompression overhead. Performance aware compression strategies try to improve performance through reduction of cache misses by utilizing the dynamic instruction frequency, but it sacrifices code size. Code compression and encryption can be integrated to make embedded system efficient (in terms of area, power and performance) as well as secure. This thesis studies a promising direction of compression followed by encryption to reduce the decryption overhead while maintaining the individual advantages of both code compression and encryption. This thesis also proposes a dual compression scheme that aims to simultaneously optimize code size reduction and performance improvement. Experimental results show that dual compression can achieve both compression ratios of up to 60% and an average performance improvement of 50%. Moreover, compression followed by encryption reduces the execution time of the encrypted binary by 40% on an average.

CHAPTER 1
INTRODUCTION

Embedded systems have a wide variety of applications today, from multipurpose handheld PDAs to dedicated real-time control systems. Embedded systems are resource constrained i.e., they generally have limited memory and computational capabilities and there is a driving need to extract as much space efficiency and performance from the available resources as possible. There is also a need of securing proprietary programs from espionage and sabotage, while minimizing the effect on performance. Code compression addresses the memory requirements, whereas encryption provides security for application programs. This chapter is organized as follows. Section 1.1 describes code compression techniques. Section 1.2 motivates the need for combining code compression with encryption. Finally, Section 1.3 describes the thesis' contributions and organization.

1.1 Code Compression

General data compression techniques like Huffman [1], LZW [2] etc. are used to reduce the size of the targeted data to better utilize storage space. Compressing the application binary and decompressing it at runtime helps us better utilize the limited memory space in embedded systems. Figure 1-1 shows an overview of code compression in embedded systems. The compressed code is placed in the main memory and/or in the instruction cache, thus increasing their effective sizes by enabling them to hold more number of instructions. During runtime, compressed code is fetched, decompressed and sent to the next memory level or to the processor. Decompression introduces a certain overhead which increases the number of cycles for each fetch, which may reduce the program's execution rate. However, a reduced binary size of a compressed application has some features which can improve its performance. If the compressed code is stored in the main memory, filling up a cache line on a cache miss will require fewer number of cycles on the average, in effect reducing the average

latency to fetch an instruction block from the memory. Moreover, placing the compressed code in the cache means that it holds more instructions, hence increasing the effective cache size and causing a reduction in the miss rate.

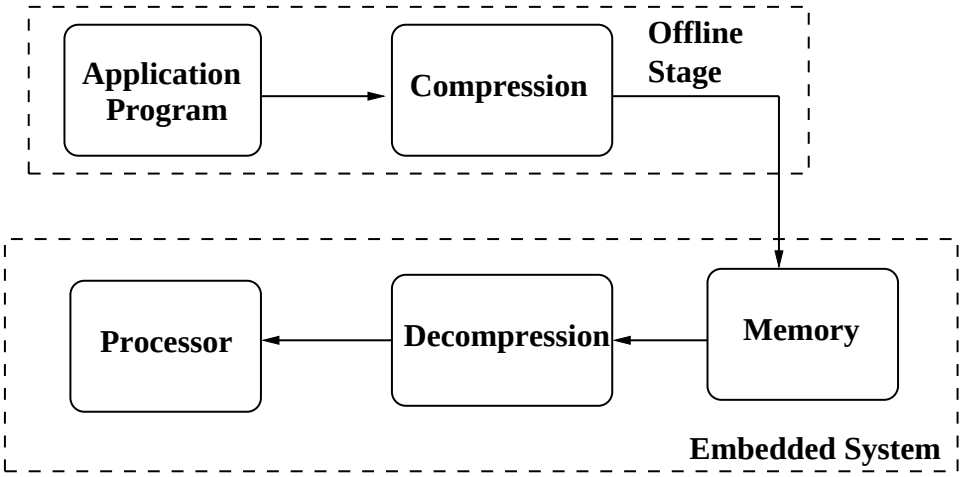

Figure 1-1. Overview of Code Compression

Code compression has been employed to exploit both code size reduction and performance increase in embedded systems. Compression Ratio is widely accepted as the metric for measuring the efficiency of compression algorithms and is defined as:

$$Compression\ Ratio = \frac{Compressed\ Code\ Size}{Original\ Code\ Size}$$

Good compression ratios can be achieved by compressing the instructions that occur most frequently in the code, whereas, a speedup is achieved by compressing the instructions that are fetched most often. Most frequent instructions in static code may not be the most executed ones and vice versa. Hence, a binary compressed to maximize one benefit may not provide the best results in the other scenario.

There are some mixed profile based compression schemes which attempt to achieve both code size reduction and performance improvement. In mixed profiling, the dictionary consists of instructions from both sets of instructions by selectively combining both static and dynamic frequencies. This approach can lead to a trade-off but cannot

achieve the best of both worlds. Chapter 3 describes how to simultaneously achieve both code size reduction and performance improvement.

1.2 Integration of Code Compression and Encryption

For more than two decades now, software industry has risen remarkably in size and importance, and like all other industries it is susceptible to espionage and sabotage. There are numerous ways devised to recover the logic of the code or even to alter the instructions in the code from the binaries of poorly protected software. As our dependency on software increases so does the necessity of better and efficient protection schemes from these threats i.e., making the software more secure without highly compromising its execution time and throughput. This thesis proposes a way of achieving it using encryption over code compression. Encryption has always served as a dependable way of protecting information. Protecting critical data for storage and transmission is one of the most commonly used applications of cryptography. For example, encryption of messages while sending them out in common media is a common practice by the wireless companies. Ciphering files wile storing them on the hard disk protects them from being read in case the hardware itself is lost or stolen. Software or rather binary code is fundamentally different from other static data. Code encryption itself is a relatively new and open field of research. Encrypting static data is mainly concerned with the encryption algorithm and the mode of operation. Code encryption requires that instructions need to be decrypted during execution, and therefore can introduce significant overhead. Such intricacies of using binary code as an active process make encryption/decryption more complex. Chapter 4 integrates code compression with code encryption, attempting to make code execution secure and efficient at the same time.

1.3 Thesis Contributions and Organization

This thesis has two major contributions: i) a novel dual compression scheme which aims to simultaneously maximize the reduction in the overall execution cycles and

the binary size, and ii) synergistically and efficiently combine encryption with code compression. In dual compression scheme, first the code is compressed on the basis of its execution profile and then second compression is done to reduce the binary size, based on the static occurrences of the instructions after the first compression. During execution, decompression is first done between the cache and the memory and then between the processor and cache. I present a detailed description of compression algorithm and decompression system with performance results and analysis. While combining encryption and code compression, I present an analysis of the most feasible and efficient architecture followed by analysis of how various parameters such as compression ratio, decryption and decompression latency, cache size etc. affect the application's performance. The framework is implemented on the SimpleScalar simulator and validated using MediaBench and MiBench benchmarks.

Rest of the thesis is organized as follows. Chapter 2 surveys related work on code compression and encryption. Chapter 3 describes the dual compression scheme. Integration of compression and encryption is discussed in Chapter 4. Finally, Chapter 5 concludes the thesis.

CHAPTER 2
RELATED WORK

The existing approaches can be divided into three related categories: code compression, encryption and a combination of code compression and encryption. Section 2.1 lists the existing code compression techniques that target code size reduction, performance improvement and the attempts to combine these two. Sections 2.2 and 2.3 give a listing of some of the encryption techniques and attempts to combine them with code compression.

2.1 Code Compression

Code compression techniques were first developed for embedded systems by Wolfe and Channin [3]. They developed a Huffman coding based compression technique in which the compressed program is stored in the main memory. A Line Address Table (LAT) is used to map the instructions in the original code to the compressed code. Lekatsas and Wolfe used Arithmetic coding for code compression in embedded systems [4]. Nam et al. [5] used dictionary based compression to compress VLIW instructions.

Larin and Conte devised a Huffman based compression on embedded systems in [6]. Tunstall coding was used by Xie et al. [7] to perform variable to fixed length compression. Usage of variable sized block was further exploited by Lin et al. [8], when they proposed LZW compression scheme for code compression of embedded processors. Code compression techniques were applied on variable length instruction set processors by Das et al. [9]. Several new techniques have been proposed to improve the standard dictionary based compression by remembering as many mismatches as possible. Although different approaches have been proposed to accomplish this, recent work by Seong et al. [10] has given promising results. They remember the mismatch positions using bitmasks, which is advantageous since a number of mismatches can be remembered using a single bitmask. The other major advantage of this method is that the compressed code can be decompressed in one cycle and therefore, it has a minimal decompression overhead and does not hamper

the processor performance. All these works emphasize on reducing the code size of the application at the cost of potential performance degradation.

There has also been some work on code compression based on dynamic frequency profiling to increase performance efficiency. Benini et al. [11] proposed a technique of selective compression to reduce the energy required by the program to execute on embedded systems. They compressed the most commonly fetched instructions to reduce the energy dissipated in memory accesses. Their profiling results show that 256 most frequently fetched instructions in their benchmark took up a large portion of the program execution time. Therefore, they only compressed these 256 instructions. The advantage of their method is the simplicity of the decompression logic. However, they targeted energy dissipation rather than system performance or the code size. Lekatsas et al. [12] proposed a dictionary based compression technique for code compression, which exclusively dealt with taking advantage of compressing words with higher frequencies. They developed a compression scheme and a decompressor which takes one clock cycle to extract instructions from compressed code through which a performance increase was achieved. They used fixed and variable-length code words in their experiments. Their results show an average performance improvement of 25%. However code size reduction is not discussed.

Netto et al. described [13] a multi-profile based compression technique where they proposed an approach to mix static and dynamic instruction profiling to effectively exploit size-performance trade-off. Like our approach, they too used a word-sized sets of indices, removing any compressed word misalignments, giving a faster decompression. Their results show a 35% reduction in code size and a 50% reduction in instruction cache accesses. However, their work uses a single compression scheme, with a single decompressor. So, for any combination of instructions from their dynamic and static profiles, both size and performance cannot be optimal at the same time.

In the dual compression scheme described in Chapter 3, compression for speed and size are done separately. To increase speed, we have improved the code compression technique in [13] with a more compact compression format and a faster decompression method that uses an auxiliary table. The compression technique targeting a reduction in size is similar to Seong et al. [10], as it gives the best compression ratio while faces minor decompression overhead.

2.2 Encryption

Encryption techniques have been used since historic times. These techniques were basically of two types: substitution ciphers and transposition ciphers. In the former, substitution rules are defined to substitute one character with another. On the other hand, transposition ciphers change the order of characters in the code. However, all these ciphers were easily broken using statistical attacks.

Private key cryptography has been used since early 20th century in which both parties operating on the data had the same key to encrypt and decrypt. This type of shared key cryptography is of two types: block and stream. A block cipher operates on a block of data while a stream cipher works by combining the data with a stream of pseudo-random bits. Example of block ciphers include AES and DES. RC4 is an example of stream cipher. However, the problem of sharing the private key forced people to change to public key cryptography. In this system, there are two sets of keys, public key and private keys, with the encryptor and the decryptor respectively. These keys are different and one cannot be produced from the other. The encryptor encrypts the code using its public key while the decryptor decrypts it using its own private key. Therefore, the need of key sharing is avoided. RSA is an example of public key cryptography.

2.3 Combination of Code Compression and Encryption

There are few efforts to combine both encryption and compression together. Johnson et al. [14] proposed a method to compress encrypted data using Low Density Parity Check codes (LDPC) and they have shown their performance on OTP encrypted

data. However, their method is not suitable since LDPC compression is NP hard. Also, they have used their algorithm only on OTP encrypted data, which is not considered a good encryption scheme. Ruan et al. [15] improved the Shannon-Fano-Elias technique of encrypting compressed data by improving the code length. However, the intensive decryption/decompression of these codes are not applicable in embedded systems. Although IBM Codepack uses keys for decompression [16], there has been no indication of encryption in them.

Shaw et al. [17], developed a method in similar lines with ours on combination of compression and encryption. They work mainly on image and video files and not for embedded systems. The compression schemes used by them, which comprises of codebooks is lossy in nature. This may be suitable for data, but certainly not applicable for the code, since a lossy code can lead to inaccuracy and inadequate functionality. Cypress, developed by Lekatsas et al. [18] has integrated compression and encryption. They deal with both code and data sequences for multimedia embedded systems. In their system, they use a compression/encryption technique which works on both code and data. The problem arises when operating on data. Data can be written back to the memory by the processor. They try to counter this by changing the system including changes in page table and placement of instruction and data caches. This data has to be encrypted and compressed again before being written. Since, in this case, encryption and compression have to be performed during runtime, it will significantly affect processor performance and is not applicable in many systems with real-time constraints. Moreover, their approach is inherently intrusive, since there is a significant change in the actual hardware of the system. In chapter 4, I have proposed a method in which the compression and encryption of the code can be done with minimal modification in the hardware of the system, while it retains all the advantages of compression as well as encryption.

CHAPTER 3
DUAL CODE COMPRESSION

Dual code compression targets to optimize both system performance and code size reduction, which is not possible in any singular code compression scheme as it will target either size reduction or performance improvement. At a high level, dual code compression scheme is similar to other code compression methods, i.e., first a compressed binary is created offline, then decompression is done dynamically for each block of code when it is fetched during execution. The difference of course lies in the fact that compression and decompression are done twice, first for performance improvement and then for size reduction, based on frequencies of dynamic and static instructions respectively. Therefore, there should be a synergy between the two steps. The output of the first compression step should be a valid input for the second. Moreover, dynamic decompression for the two steps should be done in such a way that the overhead is minimal.

To achieve a speedup we must reduce the cache miss ratio which is possible by placing compressed code in the cache. Holding the most frequently executed instructions in compressed form will greatly enhance cache usage and correspondingly improve system performance as the cache miss-rate will reduce. The main memory utilization is enhanced by holding statically compressed code with minimal code size. The ordering of the two compression and decompression steps are shown in Figure 3-1.

The rest of the chapter is organized as follows, Section 3.1 presents an overview of dual compression and decompression. Section 3.2 describes the first half of dual code compression and its corresponding decompression, i.e., compression to target performance enhancement. Section 3.3 presents a description for the second half of dual compression which reduces the code size. Finally, section 3.4 presents the experimental results and analysis.

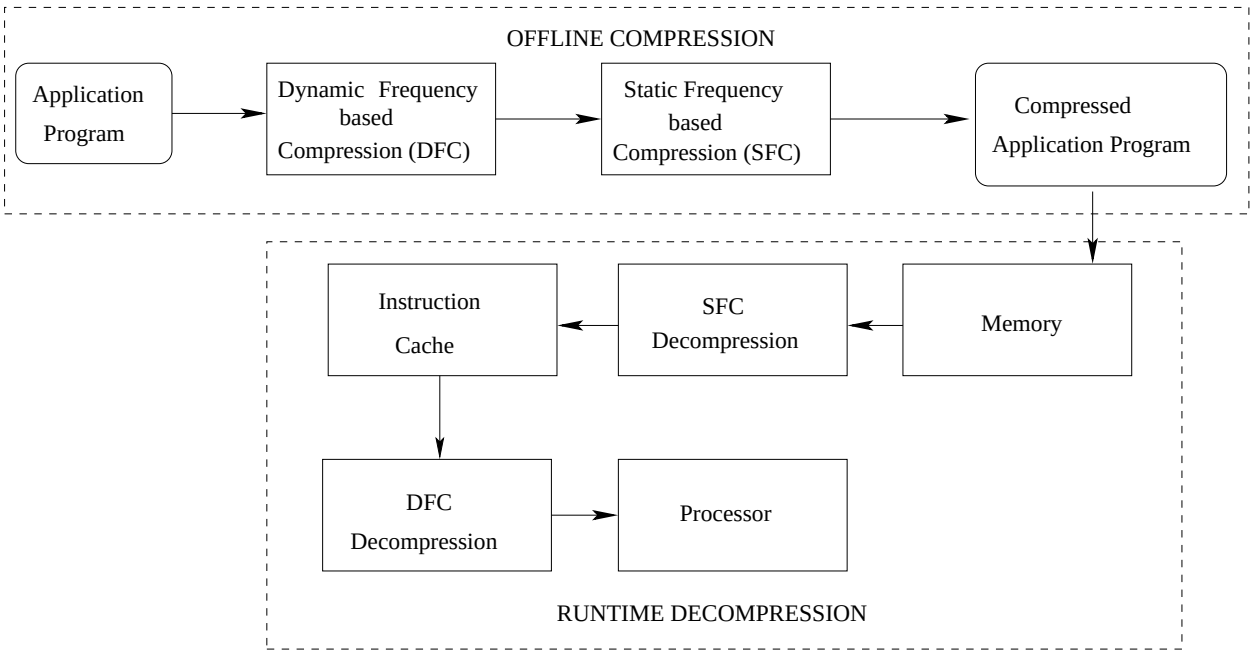

Figure 3-1. Overview of Dual Code Compression

3.1 Overview

3.1.1 Offline Dual Compression

Algorithms 1 and 2 show an overview of steps involved in generating a compressed binary. First, code compression is done to improve performance by selecting the most frequently fetched instructions. We refer to this step as Dynamic Frequency based Compression (DFC). In the second step, code size reduction is aimed and the most frequently occurring static instructions are selected for compression which we will call Static Frequency based Compression (SFC).

Algorithm 1 Dynamic Frequency based Compression

1: Create profile P of most executed basic blocks
2: Create a 256 entry dictionary D1 based on P.
3: Compress each 32-bit vector using D1 to produce C1.
4: Generate Basic Block Mapping Table BBM
5: **return** C1, D1, BBM

In DFC the most frequently called basic blocks are tightly compressed in such a way that compressed and uncompressed word lengths are fixed at the original instruction

Algorithm 2 Static Frequency based Compression

1: Create Dictionary D2 using the most frequent words in C1
2: Compress C1 using D2 to produce C2
3: Readjust Jump targets in C2
4: **return** C2, D1, BBM, D2

word length. SFC compresses the output of DFC where the most frequently occurring static words are compressed. The word boundaries are maintained in DFC, which facilitates compression in SFC. Bit-mask based code compression algorithm is used in SFC.

3.1.2 Decompression Architecture

Decompression for DFC is done between cache and memory to enable increase in cache utilization. The cache holds the most frequently executed instructions in compressed form, thus, the effective size of the cache increases and the total number of cache misses gets reduced. As decompressor is invoked for each instruction fetch, it has to be fast enough to decompress a compressed word and provide it to the processor's fetch unit in a single clock cycle.

Decompression for SFC is done between cache and memory to make decompression distributed and to reduce its overhead. When a cache line needs to be refilled, compressed words are fetched from the main memory, which are decompressed and then sent to the cache. As decompression is done only when there is a cache-miss, the decompressor's invocation is less frequent. Therefore, we can use high efficient compression techniques, yielding the best possible compression ratios which may have a reasonable decompression overhead.

There are various other combinations of placement of DFC and SFC decompression possible but they are less efficient. Post-cache decompression for both DFC and SFC will cause a heavy latency for each instruction fetch. Decompressing them together before the cache would mean that the cache would hold uncompressed instructions.

In the decompression architecture, whenever there is a cache miss, compressed blocks are fetched from the main memory, which would be enough to fill up the cache line on decompression. This way the cache holds code that is compressed with DFC. If the instruction present in the cache is in uncompressed form it is directly sent to the processor. If it is compressed, the decompressor fetches and decompresses it and stores it in its buffer, and passes on the required instruction to the processor. The details of DFC and SFC are discussed in the following sections.

3.2 Dynamic Frequency based Compression

The DFC scheme is split into three steps. The first step is profile creation, which involves identifying all the basic blocks of code in the program and the relative frequencies with which they are fetched and then creating a dictionary based on the most frequently fetched blocks. The second step efficiently compresses the code in a manner which best exploits the locality of the most frequently fetched instructions in the basic block. The third step performs a fast runtime decompression of the compressed code.

3.2.1 Profile creation

The first step in profile creation is the identification of the basic blocks and their relative access frequencies of being fetched. A basic block is a code with one entry point, one exit point and no jump instructions contained in it. It is a sequence of instructions which are all executed if the first one in the sequence is executed. The starting instruction of the block may be jumped to from any location, but none of the other instructions can be branch targets.

The method used to identify the basic blocks and their respective frequencies is as follows. We generate an execution trace of the program and calculate the frequency with which each instruction is fetched. We also identify the targets for the jump instructions. Here basic blocks are those sequence of instructions which have the same frequency of execution and no instruction as the jump target except the first one.

The next step in profile creation is selecting the basic blocks which are most frequently fetched. We compress the most frequently fetched basic blocks using a couple of intuitions. Firstly, keeping the most frequently executed instructions in the cache in compressed form will help us better utilize its space and reduces the number of cache misses. If a basic block is compressed it will take less number of fetches to bring it from the memory, therefore it saves a certain number of cycles for each fetch. Moreover, higher the frequency of that block being fetched, more the cycles we save cumulatively over the entire execution. We have to decide exactly how many instructions should be marked for compression. For this we rely on the 90-10 rule which states that 90% of program execution time is spent on 10% of the code. A dictionary is created consisting of the most frequently fetched instructions. Figure 3-2 shows percentage of fetches to the instruction contained in dictionaries for different dictionary sizes. For example, in epic benchmark, 256 most frequently fetched instructions makeup for 96.9% of the total number of fetches.

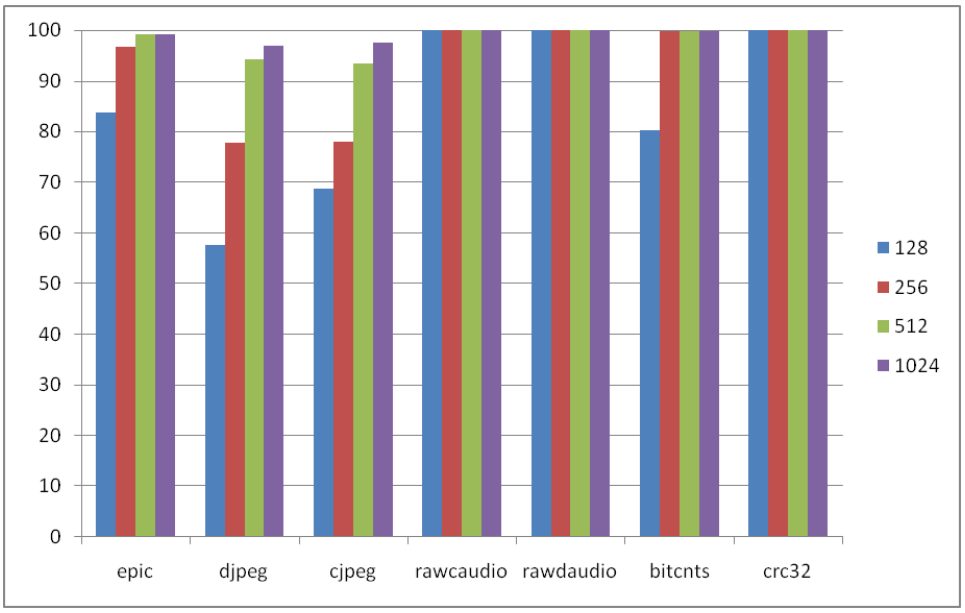

Figure 3-2. Percent of coverage of dynamic instructions for various dictionary sizes in the selected benchmarks.

3.2.2 Compression Mechanism

First we have to decide on the dictionary size. Figure 3-2 suggests that a dictionary size of 256 is reasonable since it can accommodate around 70 to 99 percent of the total instructions executed in these benchmarks. To compress the code we replace the instructions with their respective indices. As the target instruction set architecture here is Alpha, the instruction size is 32 bit i.e., 4 bytes. By selecting a dictionary size of 256, the index size would be one byte. Unlike bit-masking or dictionary based compression, a fixed block encoding is used to better facilitate compression and decompression of the basic blocks. Groups of words belonging to a basic block are compressed together to form a single word. The main advantage of this approach is that the compressed code does not get misaligned, so only a single fetch is required to obtain an instruction. Moreover, fetching a compressed word aligned to the word boundary is faster and can enable parallel decompression.

Figure 3-3 illustrates the compression mechanism. Instructions $\{1, 2, 3, 4, 5\}$ and $\{8, 9, 10\}$ form basic blocks in the program and each instruction is of size 32 bits. Due to the chosen dictionary size of 256, the index size will be 8 bits. Instructions 1, 2, 3 and 4 are replaced to form one word consisting of their respective dictionary indices. Instruction 5 is put as an index in the next word and the remaining space is filled up with padding. Similarly, instructions 8, 9 and 10 are put as indices and the remaining space is left padded. The idea behind such compression is that whenever the first instruction of a basic block is called, the next few instructions are fetched along with it.

As the words do not contain any information as to whether they are compressed or not, a Basic Block Mapping (BBM) table is required to indicate if a word is in the compressed format or not. Each entry in the table consists of information about a basic block, such as the address of the first instruction of the block, the address of the last instruction and its address mapped to the compressed form. BBM table eliminates the necessity of flag bits/bytes that indicate the whether the instruction is compressed or

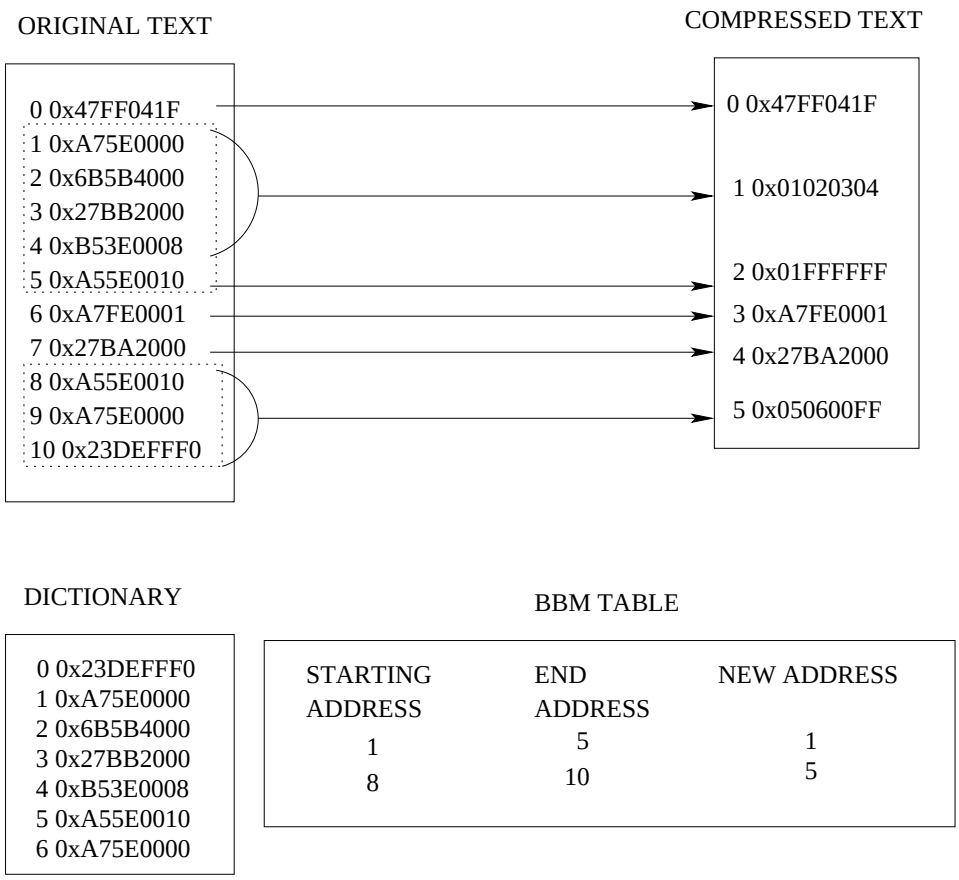

Figure 3-3. Dynamic Frequency based Compression Mechanism

not. This extra information (used in existing methods) spanned over the entire binary adds on to the size of the compressed binary. The size of the BBM table itself is very small as it only contains information about the most frequently fetched basic blocks. It also eliminates the requirement of Line Address Tables (LAT) which map the jump targets in the compressed code in existing methods. It is easy to map the jump target using the BBM table due to fixed encoding.

The dictionary size is important in this type of compression format. One compressed word fully consists of dictionary indices. Thus, smaller the dictionary, more instructions could be fit in to one compressed word. In Alpha ISA, an instruction word is 32 bits, if we choose a dictionary size of 256, i.e., an index of size 8, we would be able to fit in a maximum of four instructions into one word, as shown in Figure 3-3. A choice has to be made for the index size; a large index would mean more number blocks to be

compressed but they will be loosely compressed, whereas a smaller index will have the opposite effect.

3.2.3 Runtime Decompression

Here I describe the details of the system that is used to perform the runtime decompression. The decompressor is placed between the cache and the processor for DFC that has two advantages. Firstly, compressed code is placed in the cache and secondly fetching individual words from the cache (compressed or uncompressed) is straight forward.

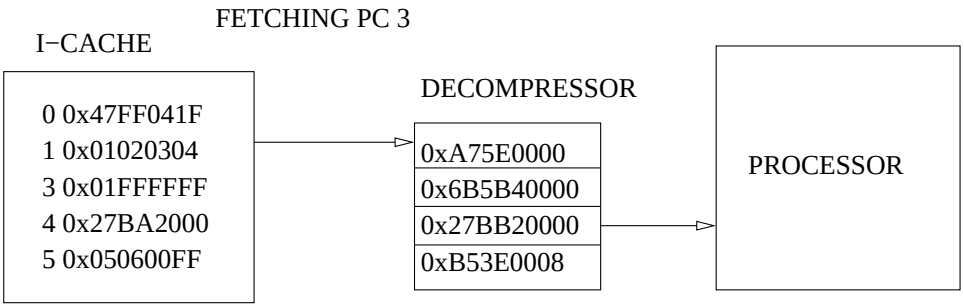

Figure 3-4. Decompression and execution of DFC compressed code

The runtime decompression unit uses the BBM table to see which instructions are compressed. When the decoder fetches a compressed word, it decompresses it using the dictionary and sends back the required instruction to the processor and stores the rest in its buffer. The number of instructions contained in a compressed word depends on the dictionary size as discussed earlier. As the word boundaries are maintained even after compression, fetching a compressed word from the cache is fast and simple. If the instruction to be fetched is uncompressed, we only have to map it to the right location and fetch the whole word. If the instruction is compressed, we fetch the compressed word, obtain the index and return the required instruction after a dictionary lookup. Figure 3-4 shows how an instruction is extracted from a compressed word and executed. Here we execute the instruction with the original PC 3. By looking at the BBM table,

PC 3 is shown to be in the basic block $\{$1,2,3,4,5$\}$ which starts from address 1 in the compressed code which will contain instructions $\{$1,2,3,4$\}$.

Consider another example where PC is not part of the basic block, example PC 6. In this case the basic block which is before 6 is $\{$1,2,3,4,5$\}$. We need to divide the basic block size by 4 (right shift by 2) to obtain the number of compressed words and add it to the offset from the last word of the compressed block, i.e., new address for current PC = new address for the block above+((block size-1) $>>$ 2)+(current PC- last address of the block). In this case, new address for PC 6 will be 1+(5-1) $>>$ 2)+(6-5)=3.

After mapping PC, the compressed word at address one is fetched by the decompressor, the instructions are extracted from it and kept in the decompressor's buffer and the required instruction is sent to the processor for execution. The decompressor's fetches to the cache is pipelined thus fetching any instruction from the cache only takes one cycle except for instructions that are jump targets. The additional advantage of a BBM table is that it enables code compression without the use of an additional compressed/uncompressed flag with each word. This saves significant space since the BBM table itself is very small.

Placing the decompressor after the cache also means that the cache holds compressed code, thereby the effective size of the cache increases. If the cache only holds the most frequently executed code, i.e., the compressed basic blocks, the effective size of the cache increases by inverse of the compression ratio of the basic blocks. In this system where each compressed word holds four instructions, the cache size effectively increases four times. This increase in effective cache size is the reason of the expected speedup. A larger cache means less number of overall fetches from the main memory.

The decompression overhead should also be small in order to obtain a proper speedup. The decoder in this system uses one cycle to fetch an instruction from the cache, decompress it and stores the four uncompressed words in its buffer. The

processor fetches the instructions from the buffer in the next cycle. Thus, fetching four instructions from a compressed basic block takes five cycles. Fetching an instruction which is not compressed will take two cycles, one for the decompressor to fetch it from the cache and one for the processor to fetch it from the buffer. We can reduce the number of cycles further by pipelining the fetches by the decompressor. A fetch by the decoder takes two cycles only if the instruction to be fetched is a jump target, otherwise all the instructions will take just a single cycle. Cycle time to fetch an instruction from the decoder's buffer would be very small compared to that from an L1 cache.

3.3 Static Frequency based Compression

Compression schemes used in optimizing code size can be complex and their dynamic decompression can have significant decompression latency. Dynamic decompression for SFC is done before the cache, thus decompression is invoked only when there is a cache miss. The fact that the decompressor is not in the critical path of execution, i.e., the decompressor is not invoked for each fetch by the processor gives us the freedom to use efficient compression mechanisms, such as, Huffman or arithmetic compression that provide excellent compression ratio but have a high decompression latency.

The compression mechanism used for SFC is based on the work done by Seong et al [10], which uses a bit-mask based compression scheme which gives a high compression efficiency and has a single cycle decompression penalty. Compression is performed on the DFC compressed code. As mentioned earlier, the word boundaries in DFC are maintained, hence, direct application of bitmask based compression is possible to perform SFC.

Unlike [10], we have placed the decompression engine for SFC before the cache. Thus, decompression is invoked at each cache miss to fill a cache line. As the code in the main memory is in compressed form, intuitively it will require less number of fetches to the main memory on the average to fill a cache line. Furthermore, as we are using

a fast decompression engine, we should see a further speed increase in the system because of SFC.

I have used bitmask based compression where a two-bit bitmask is used. The dictionary consists of the most frequently occurring static instructions and the bitmask is selected by XORing the variation in the instruction from the dictionary index. Other than these variations the compression mechanism remains the same as [10] and is outlined in Algorithm 3. Figure 3-5 shows the encoding used for compression.

Algorithm 3 Bitmask-Based Compression

1: Create the frequency distribution of instructions.
2: Create the dictionary based on frequency as well as bit-mask based savings.
3: Compress each 32 bit vector.
4: Handle and adjust branch targets
5: **return** Compressed code and dictionary

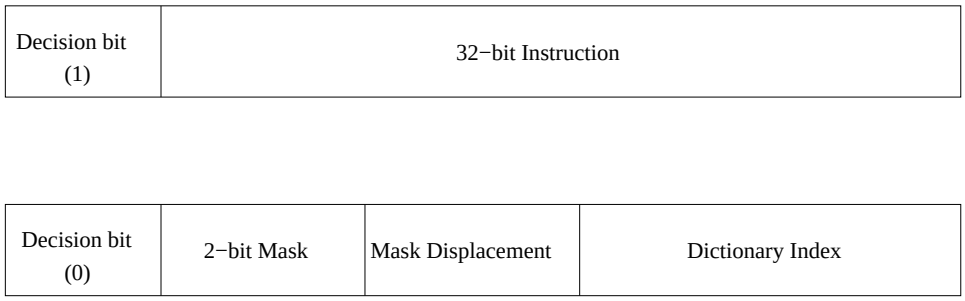

Figure 3-5. Compression encoding used in bit-mask based encoding

It is useful to consider larger dictionary sizes when the current dictionary size cannot accommodate all the vectors with frequency value above certain threshold. (e.g., above 5 is profitable). However, there are certain disadvantages of increasing the dictionary size. The cost of using larger dictionary is more since the dictionary index becomes bigger. The cost increase is balanced only if most of the dictionary is full with high frequency vectors. Most importantly, a bigger dictionary increases access time and thereby reduces decompression efficiency. A standard dictionary size of 2048 is used.

During execution, each time there is a cache miss, compressed blocks are fetched from the memory which are then decompressed and placed in the cache. The number

Table 3-1. A summary of the number of static and dynamic instructions in the selected benchmarks where each instruction is of 4 bytes.

Benchmark	Dynamic Instructions	Static Instructions
epic	59494631	47124
cjpeg	19025567	49896
djpeg	5887958	53852
rawcaudio	7610111	27256
rawdaudio	6309300	27248
bitcnts	5276065	23284
crc32	5108304	28392

of blocks fetched from the memory should be sufficient to fill up the cache line after decompression. The rest is stored in decompressor's buffer. As the number of blocks fetched from the main memory to fill up the cache line would be less compared to regular execution of uncompressed code, a speedup is expected.

3.4 Experiments

3.4.1 Experimental Setup

Experiments were performed in SimpleScalar performance simulator for MIPS uniprocessor architecture using a selection of benchmarks from MediaBench and MiBench compiled for Alpha ISA. The benchmark programs employed were epic, cjpeg and djpeg image compression utility, adpcm-encode and decode voice compression program, bitcnt from MiBench's automotive suite and crc32 from telcom suite. The simulation system consisted of a Super Scalar MIPS Processor, a decompressor each for DFC and SFC, a single instruction direct cache with a line size fixed at 16 bytes, and fetching a cache line from the main memory which takes 64 cycles. Table 3-1 shows a description of the number of static and dynamic instructions for each benchmark used.

3.4.2 Code Size Reduction

Figure 3-6 shows the code size reduction achieved in the code by SFC. The implementation of bit-mask based compression for a dictionary size of 2048 entries give compression ratios from 0.60 to 0.65. These numbers are similar to the results in

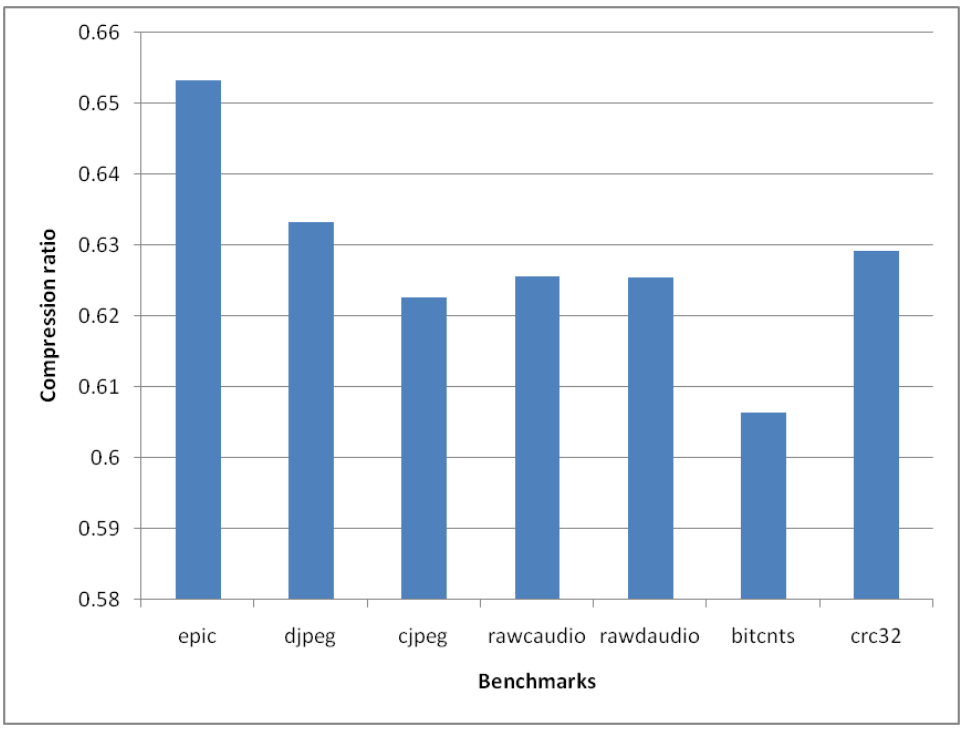

Figure 3-6. Compression ratios for the benchmarks, using SFC

[10]. As expected, there is almost no size reduction in the DFC stage, therefore, SFC is exclusively responsible for code size reduction.

3.4.3 Performance Increase

Table 3-2 shows performance of the uncompressed and compressed binary in terms of the number of clock cycles taken to execute using a range of cache sizes and the corresponding misses for each benchmark. For each benchmark there is a trend in performance improvement as the size of the cache decreases. The reason for this trend lies in the fact that the difference in the cache misses between uncompressed and compressed code decreases with an increase in cache size. Therefore, the ratio of reduction in cycles will reduce with cache size increase. The greatest performance improvement is observed for cache size of 128 bytes. Embedded systems generally use small caches and our techniques can be beneficial in such environments.

Performance improvement is more for benchmarks whose critical code (the most frequently executed instructions) is much larger than the cache. This could be seen

Table 3-2. Number of clock cycles for the uncompressed and compressed benchmarks for various cache sizes. The cache sizes are in bytes

Benchmarks	Cache(bytes)	Clock Cycles		Cache misses	
		Original	Compressed	Original	Compressed
epic	128	291288151	125664394	4790372	1701837
	256	144591613	94542719	1915429	178483
	512	94687156	81393578	986832	662288
	1024	80951000	67085506	660391	317468
	2048	63696412	51322191	298124	39456
djpeg	128	86863373	25427322	1474658	540177
	256	46942406	19158549	766816	392742
	512	24810547	14050627	567226	284815
	1024	17415100	11420931	562740	153818
	2048	9579240	7054849	114398	95077
cjpeg	128	237649522	104696109	4144970	2056011
	256	113684344	90706713	1834525	1710372
	512	77921692	67695685	1216562	1159591
	1024	59294358	56290566	732964	855303
	2048	40666875	29184203	436131	265894
rawcaudio	128	83493514	4454026	436131	312296
	256	4584825	4429282	302220	4660
	512	4479397	4395664	6771	3551
	1024	4360331	4344856	4233	2313
	2048	4334726	4339501	1200	1074
rawdaudio	128	83493514	4454026	1480939	4660
	256	4584825	4429282	6771	3551
	512	4479397	4395664	4233	2313
	1024	4360331	4344856	1200	1074
	2048	4334726	4339501	702	925
bitcnt	128	55195281	16122581	1480939	265894
	256	30083364	9716254	509068	88666
	512	16747278	9728918	228571	88377
	1024	10459757	5248356	108273	8113
	2048	5338534	5220950	7748	7404
crc32	128	3697698	3574271	5215	3529
	256	3697698	3574271	5215	3529
	512	3662466	3601606	4654	4662
	1024	3576362	3538366	2853	2458
	2048	3535711	3513130	1958	1936

for djpeg which has fairly large critical code size. There is a huge increase in the percentage reduction in the number of cycles, which decreases steadily with increase in cache size. For benchmarks whose critical code fits easily in the cache, the difference between the performance of compressed and uncompressed code is negligible. For example, there is minor change in the number of cache misses for rawcaudio and crc32 if we increase the cache size after 256 bytes, which implies that the cache easily accommodates the entire critical code, even in uncompressd form. Therefore, compressing the code in that cache configuration would not decrease the number of cache misses, hence no performance improvement is seen. In the case of bitcnts, no performance improvement for cache size 2K was observed, a 2K cache holds the whole critical code. Moreover, that performance is equal to that seen for compressed code in a 1K cache. This is because compressed version of the critical code fits entirely in a 1K cache but not the uncompressed form which results in a performance improvement of two times in this case.

Figure 3-7 summarizes the trends in reduction in cache-misses with increase in cache size for various benchmarks. The decrease in the cache miss-ratio for the benchmarks for different cache sizes and reduction in the number of cycles due to compression follows a similar trend as shown in Figure 3-8.

As discussed earlier, SFC may produce a slight speedup as the total number of fetches made to the memory is expected to decrease due to reduced binary size. As a corollary to this, combining DFC with SFC should give a better speedup than DFC alone. The following figures show the number of cycles for four cases, namely running an uncompressed binary, a binary compressed using SFC only, compressed using DFC only and compressed using both DFC and SFC . Running uncompressed code requires in the most number of cycles, followed by SFC only code, DFC only code, and SFC and DFC combined. The improvement due to SFC is more apparent in smaller caches. A smaller cache means a greater miss rate, which results in more number of accesses to

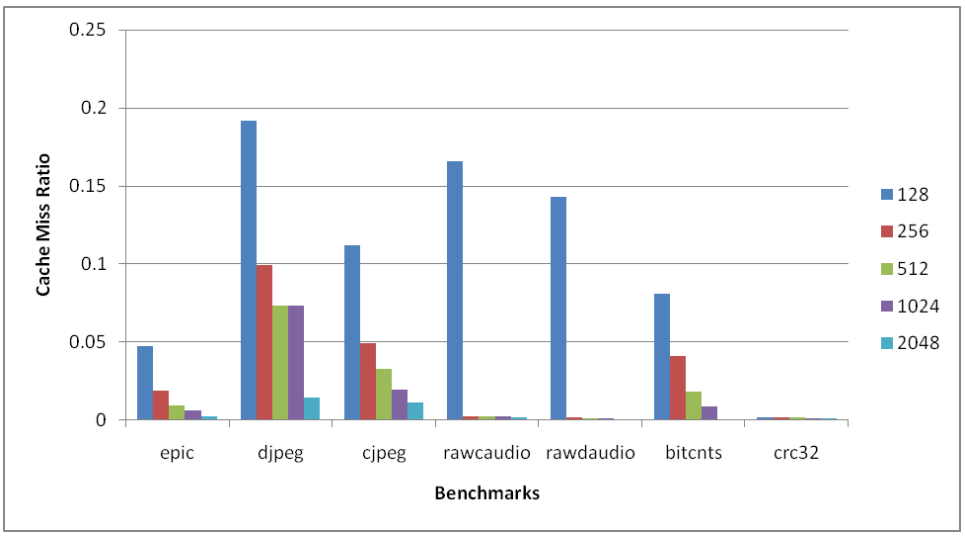

Figure 3-7. The miss ratios for the benchmarks for various cache sizes

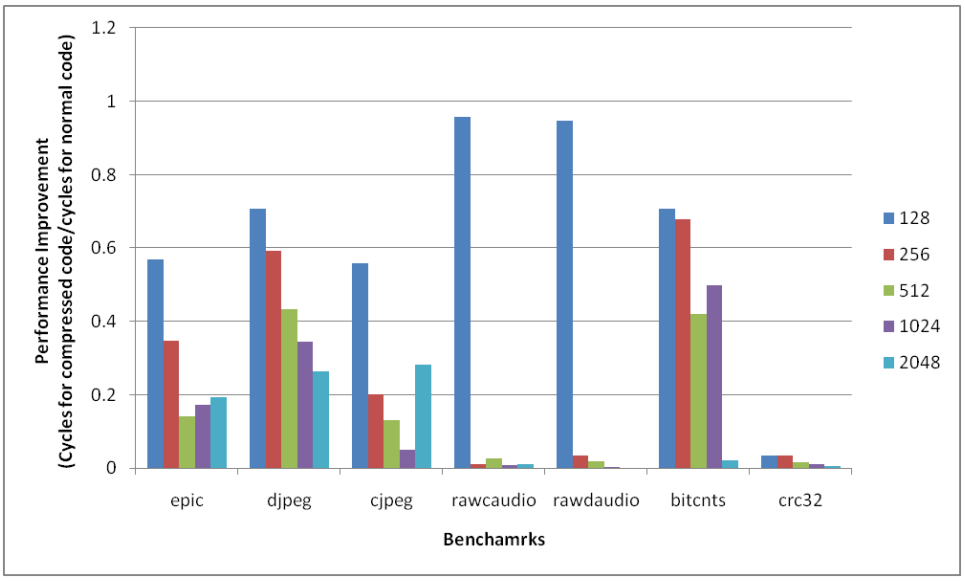

Figure 3-8. Ratio of the reduction in the number cycles due to compression for various cache sizes.

the main memory. If the main memory holds compressed code each memory access will effectively bring in more instructions. Thus, less number of memory accesses is required during the entire execution. This difference in the number of memory accesses is the reason for the attributed speedup.

Figure 3-9, Figure 3-10 and Figure 3-11 shows this trend for the larger benchmarks epic, djpeg and cjpeg respectively. Figure 3-12 shows the same for rawcaudio. The

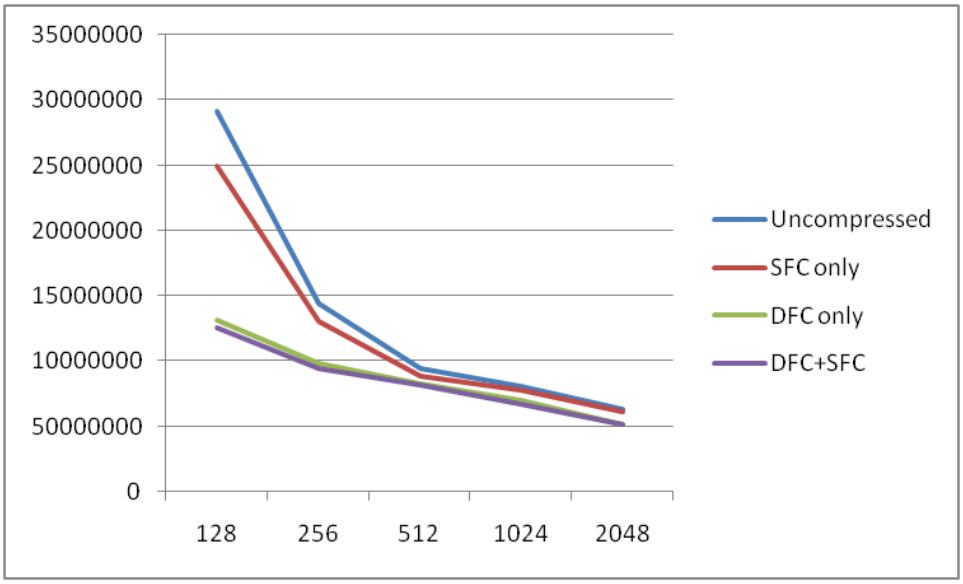

Figure 3-9. Cycles for epic

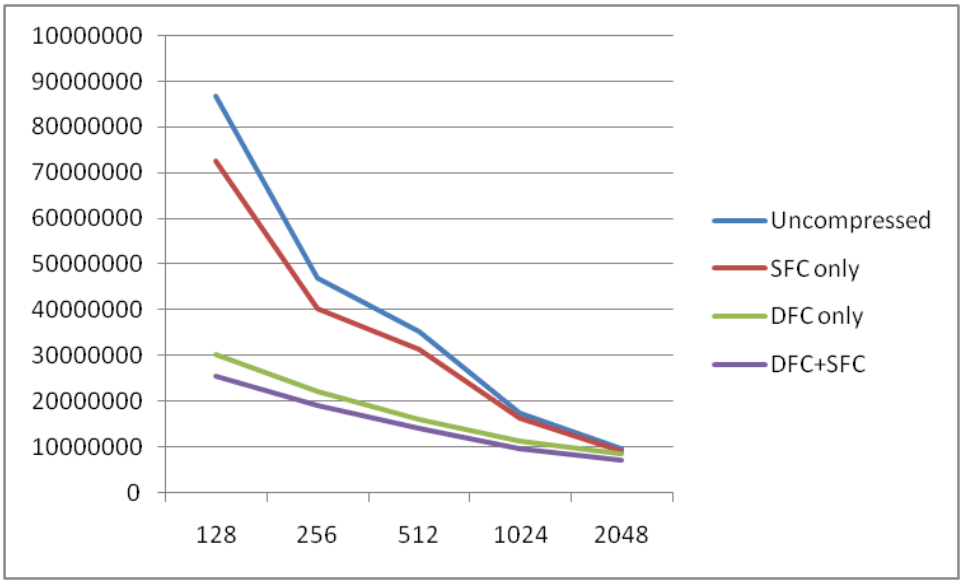

Figure 3-10. Cycles for djpeg

performance improvement is more apparent in the larger benchmarks because their critical code is large hence have more cache misses. Critical code is fairly small in the case of rawcaudio which easily fits in a cache of size 256 bytes if the binary is not compressed using DFC. When compressed, critical code of rawcaudio also fits in a cache of 128 bytes. Therefore, no improvement is apparent for all cache sizes after 128 bytes for rawcaudio.

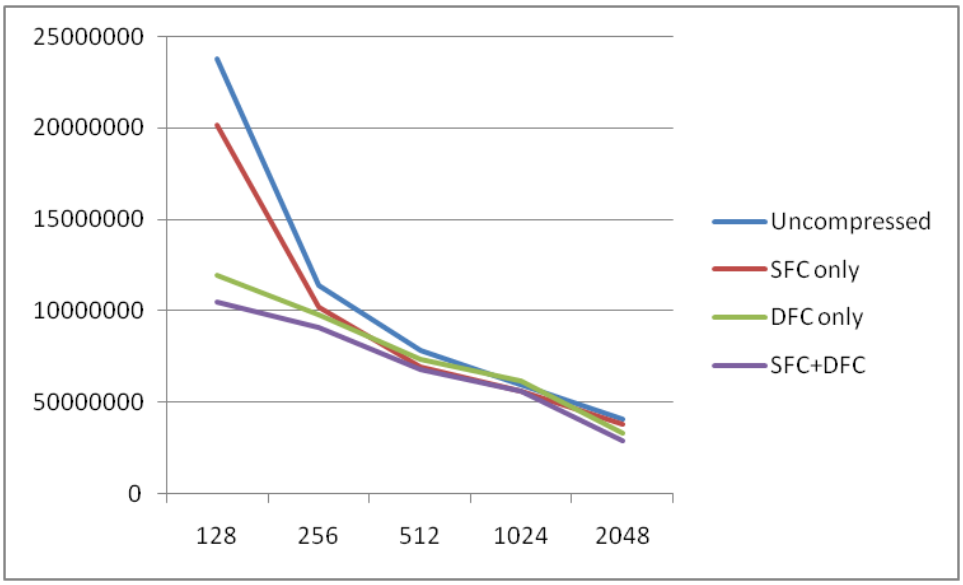

Figure 3-11. Cycles for cjpeg

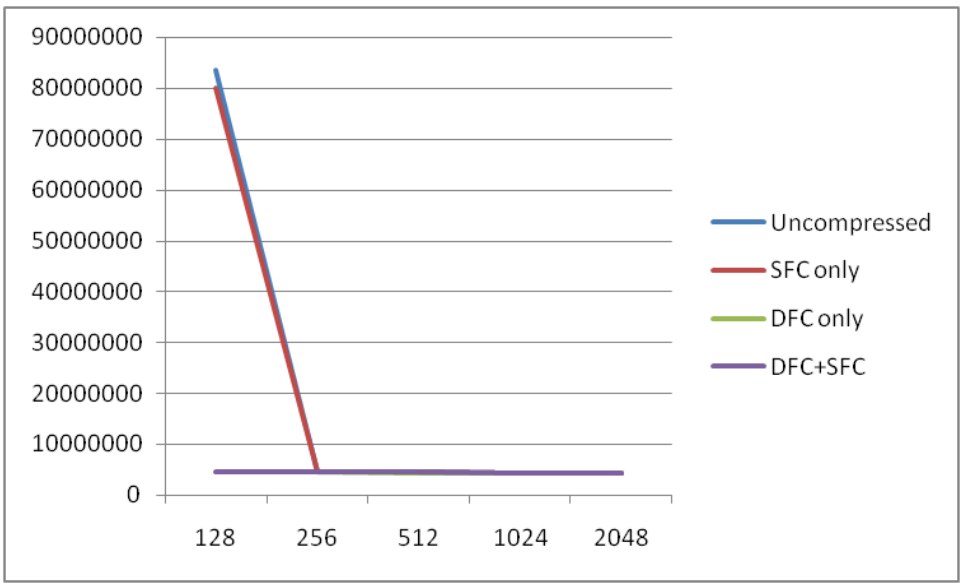

Figure 3-12. Cycles for rawcaudio

CHAPTER 4
INTEGRATION OF CODE COMPRESSION AND ENCRYPTION

This chapter presents a description and analysis on integration of encryption and code compression. In Section 4.1 a basic architectural decision regarding the sequence of encryption and compression is discussed. Section 4.2 presents a performance model of compression and encryption with an analysis of the placement of caches. Section 4.3 presents the experimental setup and results.

4.1 Combining Compression and Encryption

There can be two ways in which compression and encryption can be combined: encryption followed by compression, or compression followed by encryption. Combining both encryption and compression may lead to a number of problems. The first and major problem being that both decompression and decryption are slow and hence may prevent the full utilization of the processor performance. The decompression engine should be such that the rate at which instructions are produced from it is equal to the rate at which the instructions are executed by the processor. In the next two subsections, we will discuss the challenges associated with the two possible combinations of encryption and compression.

4.1.1 Encryption followed by compression

The first scenario is shown in Figure 4-1. Most compression algorithms take advantage of the matching patterns in the uncompressed data set. Encrypted data generally has high entropy and therefore, has less similarity in patterns. As a result, it is difficult to compress those data.

4.1.2 Compression followed by encryption

This is the most useful sequence when one thinks of combining compression and encryption. It is easier to compress the unencrypted code as the regularity patterns present in the instructions are pretty high. Moreover, this compressed data can be easily encrypted and sent across the insecure channel to the receiving end. The decryptor and

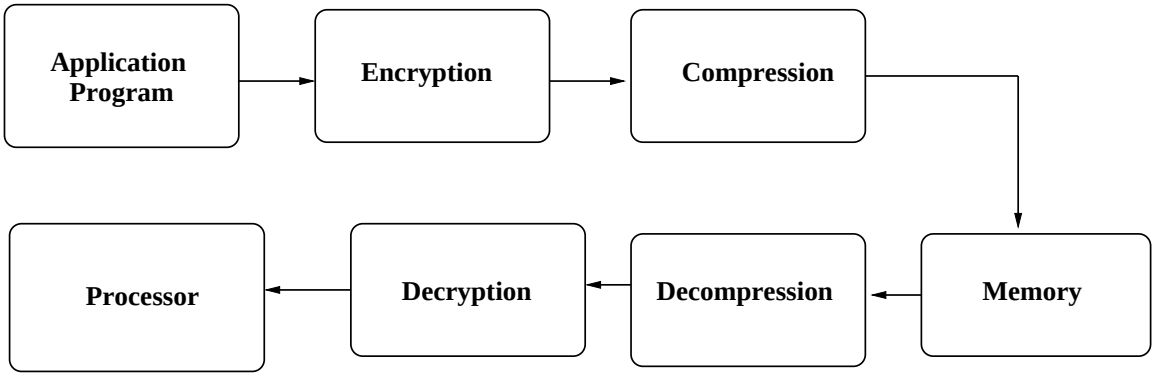

Figure 4-1. Encryption followed by Compression

the decompressor can do the rest of the work. The whole scenario is shown in Figure 4-2.

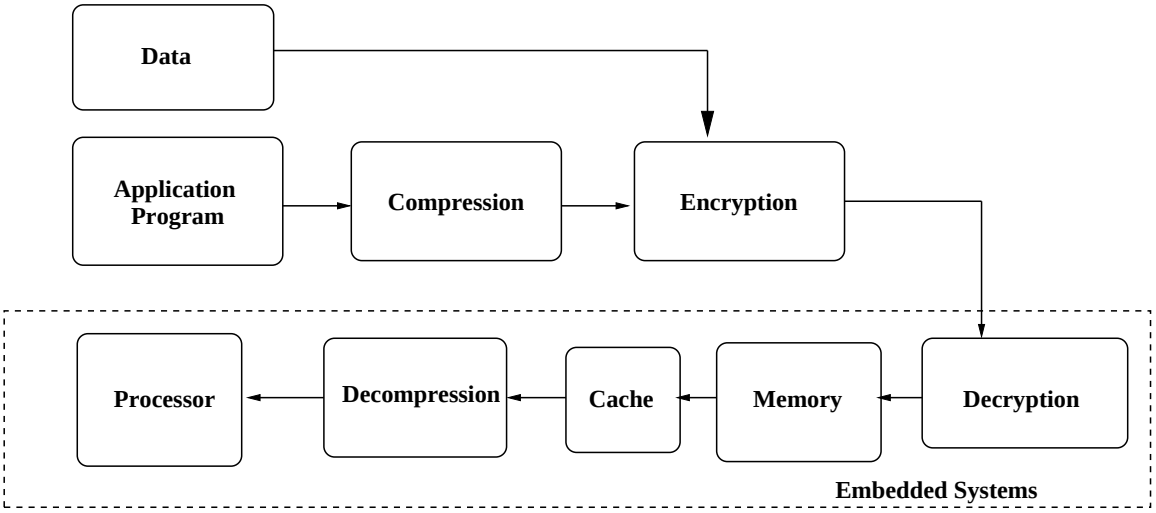

Figure 4-2. Compression followed by Encryption

4.2 Dynamic Code Encryption and Compression

The discussion in the previous section concludes that compression followed by encryption is suitable for embedded systems. This section discusses various implementation mechanisms that are possible and their impact on performance.

4.2.1 Compressed Binary Creation

Algorithm 4 outlines the basic steps in creating a compressed-encrypted text segment in a binary.

Algorithm 4 Basic Compression-Encryption

1: Compress the text segment of the program.
2: Retarget the jumps where possible.
3: Create a mapping table for the rest of the jumps.
4: Encrypt the compressed text segment.

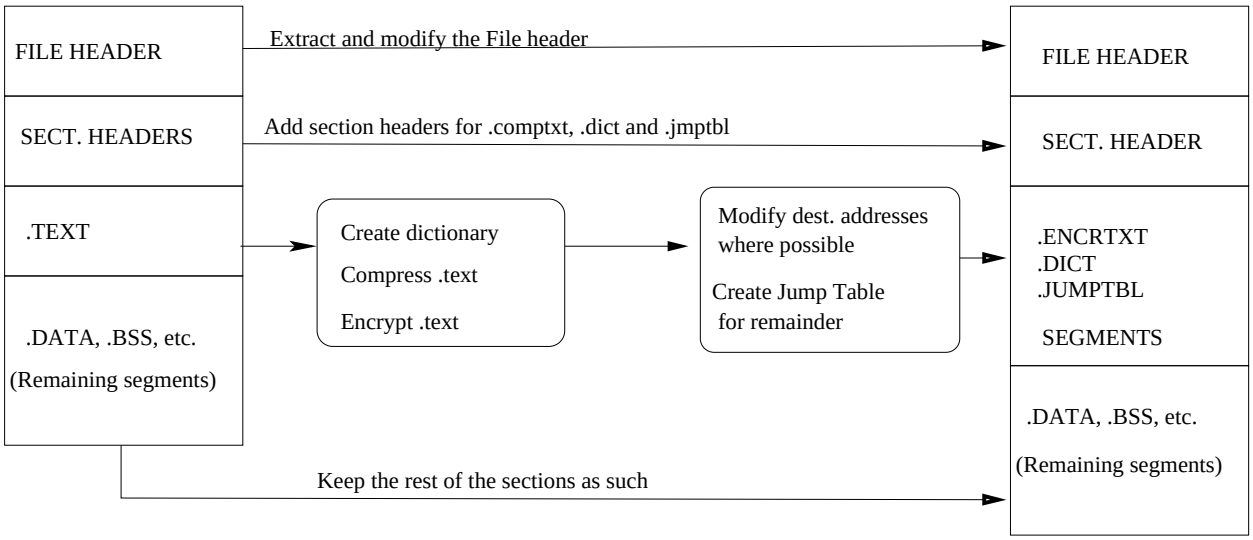

Figure 4-3. Procedure used to compress and encrypt an ECOFF binary

Figure 4-3 illustrates the algorithm for an ECOFF binary[1] . The text segment is extracted from the binary and compression is performed on it using bitmasking [10], which produces an auxiliary jump-mapping table, a dictionary and a compressed text segment. This compressed text is then encrypted and a new binary file is created using the encrypted text, the dictionary, the jump-mapping table and the rest of the segments from the original file. The creation of the encrypted binary is static, i.e., it is done offline. Execution of this binary however will be dynamic. The encrypted text segment is kept in the memory and during fetch of each instruction it has to be decrypted and decompressed. Dynamic decoding (decryption and decompression) involves dedicated decoder which fetches instruction blocks from the memory, decodes them and sends

[1] ECOFF and EFL are widely used formats for binary representation of application programs

back the decoded instruction to the cache or the fetch unit and stores the rest in its buffer. There is always a decompression overhead associated with the decoder unit which can be minimized by pipelining.

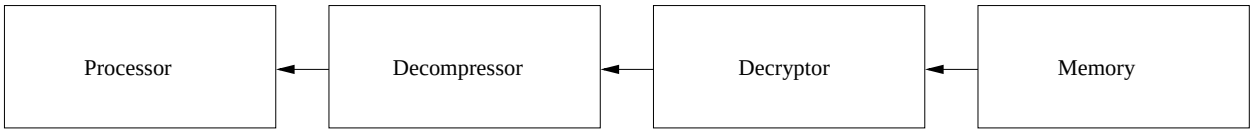

Figure 4-4. Basic compression followed by encryption model

4.2.2 Performance Analysis

Figure 4-4 shows a basic system on which dynamic decoding is performed. The decoder sits between the processor and the memory, individual instructions are fetched from the memory.

As the code is compressed, more instructions would be brought in with a single fetch of a block on the average. This results in lessening the total number of fetches to the main memory as a single fetch brings in more instructions in compressed from than in uncompressed from. Reducing the number of fetches should reduce the total number of cycles of execution. However, decoding a block of compressed instruction will take up some cycles depending on the complexity of the compression and encryption algorithm. Now, if the total cycles taken up in decoding the instructions is less than the total cycles required to fetch them, we will see a speedup in the execution.

The following equations give a basic mathematical model for the above analysis. We take the basic system shown in figure 4-4 and assume there is a uniform compression ratio throughout the text segment. We consider a simple unpipelined model which can be improved when pipelining is introduced

Let,

C=Compression ratio of the text segment

M=Cycles taken to fetch a word from memory

E=Cycles taken to decrypt a word of encoded text

R=Cycles taken to decompress a word of encoded text

N=Total number of instruction words fetched during execution

Then,

Total cycles taken to fetch the code for an unencrypted and uncompressed binary would be

$$T_n = N.M \tag{4–1}$$

Total cycles taken to fetch and decrypt the code for a binary that is only encrypted would be

$$T_e = N.(M + E) \tag{4–2}$$

Total cycles taken to fetch, decrypt and decompress the code that is encrypted as well as compressed

$$T_{er} = C.N.(M + E + R) \tag{4–3}$$

Note that T_n, T_e and T_{er} do not constitute the cycles taken by the processor to execute the code, but only those used in fetching the code to the processor. Now Equation 4–4 gives the ratio of non-executing cycles between encrypted-compressed text and regular text and Equation 4–5 gives that ratio for encrypted-compressed text and only encrypted text.

$$\begin{aligned} C_N &= \frac{T_{er}}{T_n} \\ C_N &= \frac{C.(M + E + R)}{M} \\ C_N &= C\left(1 + \frac{E + R}{M}\right) \end{aligned} \tag{4–4}$$

$$\begin{aligned} C_E &= \frac{T_{er}}{T_e} \\ C_E &= \frac{C.(M + E + R)}{M + E} \end{aligned}$$

$$C_E = C\left(1 + \frac{R}{M+E}\right) \tag{4–5}$$

C_E gives the effect of compression alone on the performance of an executed binary while C_N gives the absolute effect on performance that encrypting and compressing a binary would have. The goal is to make C_N and C_E as low as possible. The obvious way to do it is to have a lower compression ratio C (better compression efficiency) and a low decompression latency R. We usually have to make tradeoffs when considering these two potentially conflicting requirements. For example, Huffman and arithmetic encoding give the best compression ratio but its decompression proves to be very slow. Dictionary based compression and selective bit-masking give a decent compression ratio with fast decompression.

Here we have assumed that the compression is uniform throughout the text segment. In a real scenario, uniformity or irregularity in compression would also affect C_N and C_E. There will be some parts of the code that may be more tightly compressed than others and different parts of the code are fetched at different frequencies. If the parts that are fetched the most number of times are compressed more tightly the performance will improve further.

Similarly, encryption algorithm should be chosen in relation to C, M, and R so as to keep C_N small. However, lesser the computational complexity of the encryption algorithm the less secure it would be. So the designer has to make a choice between security and speed. For example, AES will have a larger decryption latency than DES. Hence C_N would be smaller for DES and C_E would be smaller for AES, i.e., execution of an encrypted and compressed code will be slower for AES compared to DES but the effect of compression would be more significant for AES.

For both Equations 4–4 and 4–5, M gives the cache miss latency, and as Equations 4–4 and 4–5 suggest, a large M would reduce the effect of the other latencies. That means, if M is much larger than both E and R the decompression and decryption latencies would be negligible.

4.2.3 Placement of Cache

Till now we have discussed a generic system involving only a processor, decoder and memory. A more realistic system would also involve caches. This gives us the opportunity to explore the different configurations that the system caches can have and their relative advantages and disadvantages.

First of all, decryptor should always be placed before the cache. Placing it after the cache would mean that each instruction fetch invokes the decryptor, which would make the system extremely slow. Placing it before the cache will cause its invocation only on cache miss. As decompression could be fast, the placement of the decompressor is more flexible.

Figure 4-5 shows a configuration where the decoder is put between the cache and the main memory. Here the job of the decoder is to both decrypt and decompress a block of code from the memory and provide the cache with a block of regular code.

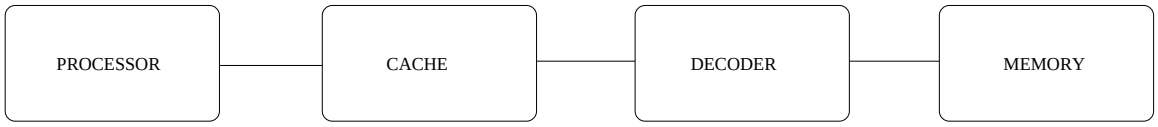

Figure 4-5. Processor-Cache-Decoder (PCD) architecture

The decoded instructions are sent back to the cache. In this scheme the granularity of decompression is changed from one instruction to an instruction block of the cache. The larger the cache block size, more the number of instructions that are sent back to the cache per fetch. Also, more the number of compressed instructions fetched for a single round of decompression, better the performance improvement. A larger block size would not necessarily mean a reduction in the total number of fetches from the cache. That would depend on the actual binary and the size of the basic blocks in the code.

The PCD architecture is similar to the simplistic model in figure Figure 4-6 except that the granularity has changed, instead of individual words, processing is done on cache blocks. A larger cache size would mean more hits and a lower frequency of

fetches. As that frequency drops with a larger cache size, so does the effect of the decoder unit, i.e., total number of fetches to the memory and the total number of blocks decoded will drop. So the difference between the number of fetches in compressed code and uncompressed code is smaller. In other words a large cache reduces the significance of a reduced code size. However as the unencrypted code sits in the cache, the total number of times blocks are decrypted would reduce with a large cache. That means a lower decryption overhead. So when compared with the performance of a regular program (unencrypted and uncompressed), with the same cache size, there should be little difference in the performance ratio.

Figure 4-6 shows a processor-decompressor-cache-decryptor (PDCD) architecture. In this scheme the encrypted text is fetched as blocks from the memory by the decryptor, which are then decrypted and sent back to the cache.

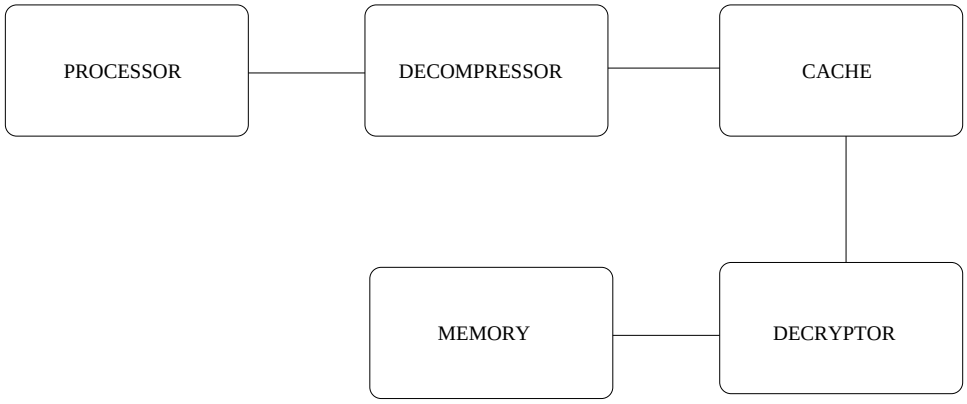

Figure 4-6. Processor-Decompressor-Cache-Decryptor (PDCD) architecture

The decompressor fetches the compressed text from the cache and sends back instruction words to the processor. The decryptor and the decompressor cannot be placed together between the processor and the cache as it would give a large decryption overhead for each fetch from the processor. In the PCD architecture, the uncompressed instructions are kept in the cache from where the processor fetches them. Cache size and the cache replacement method determines miss rate of the cache system.

The advantage of PDCD over PCD is that as the compressed text is kept in the cache, more instructions are effectively placed in the cache, i.e., the effective cache size increases which reduces the miss ratio which in turn reduces the number of fetches to the main memory. However as the processor fetches instructions directly from the decompressor, there is a decompression latency for each instruction fetch, and it would be essential for the decompressor to be extremely fast.

4.3 Experiments

4.3.1 Experimental Setup

Our experiments were performed using SimpleScalar performance simulator for MIPS uniprocessor architecture. A selection of benchmarks from MediaBench and MiBench compiled for Alpha ISA is used to perform our experiments. The benchmark programs used were epic, cjpeg and djpeg image compression utility, adpcm-encode and decode voice compression programs.

I added the decompressor and decryptor modules in SimpleScalar's sim-outorder. I kept a single instruction direct cache with a line size fixed at 16 bytes, and fetching a cache line from the main memory takes 64 cycles.

I have used three compression techniques: bitmaks-based compression, dictionary-based compression and dual compression described in Chapter 3. Dictionary and bitmask based compression use a single-cycle decompressor. Block ciphers DES and AES are used for encryption using ECB encryption mode. DES uses a block size of 64 bits where as AES uses that of 128 bits and have a decryption latency of 32 and 128 cycles, respectively.

As bitmask- and dictionary-based compression algorithms have a single cycle decompression rate, we have used the PDCD architecture to integrate them with AES and DES. For dual compression we have also used PDCD, however, SFC and decryption are done at the same stage.

Table 4-1. Average ratio of the number of cycles for a combination of the used encryption and compression methods

	AES	DES
Uncompressed	4.35	2.29
Bitmask	2.79	1.69
Dictionary	3.47	1.77
Dual	2.50	1.36

4.3.2 Results

Figure 4-7 shows the compression ratios for dual, bit-mask and dictionary based compression schemes. Bit-masking and dual compression have similar compression ratios, and are better than the dictionary based compression scheme.

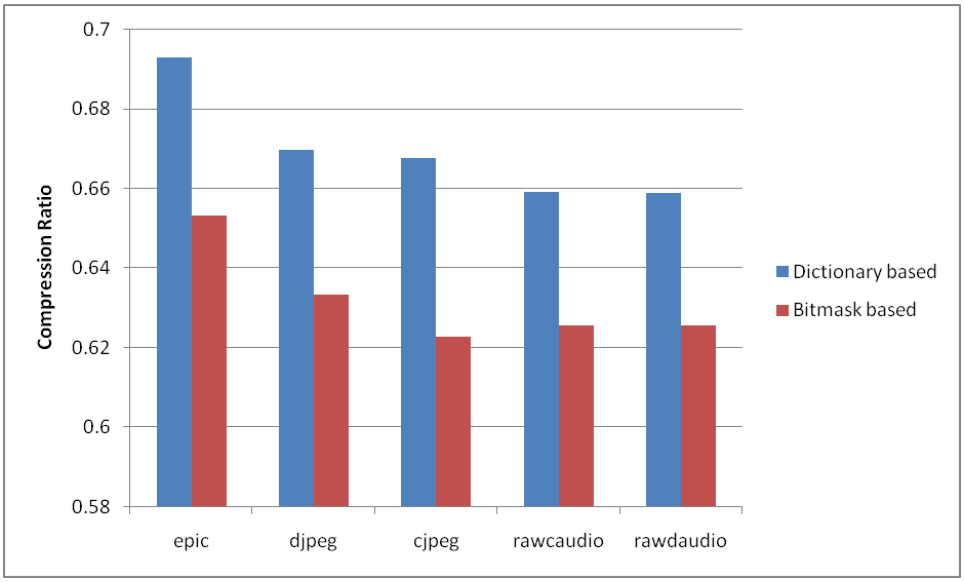

Figure 4-7. Compression ratio for the various benchmarks

The benchmarks show an average increase in performance when comparing binaries that are only encrypted and those that are encrypted as well as compressed. Improvement is large when the cache size is small as the number of as a greater number of cache misses is reduced. Table 4-1 shows the average ratio of cycles when compared to regular execution with the various combinations of AES and DES with Dual, Bitmask and Dictionary based compression schemes as well as when they are uncompressed.

Programs that are only encrypted take the most number of cycles, followed by those that are compressed using dictionary based compression scheme, bitmask based scheme and dual compression respectively. This result complies with our hypothesis as bitmasking gives a better compression ratio than dictionary based compression. Also, cache utilization is the best in dual compression as the most frequently fetched instructions are most tightly compressed. Programs encrypted with AES take more number of cycles compared to DES as it has a higher decryption latency.

Figure 4-9 and Figure 4-8 show the effect of caches on execution performance of only encrypted and encrypted and compressed code for DES and AES respectively. As the size of the cache increases, the ratio of performance of regular code and encrypted code decreases for both encrypted as well as encrypted and compressed code. This is because larger cache sizes means a lower miss ratio. A lower miss ratio means less number of fetches to the main memory and less number of blocks to decode.

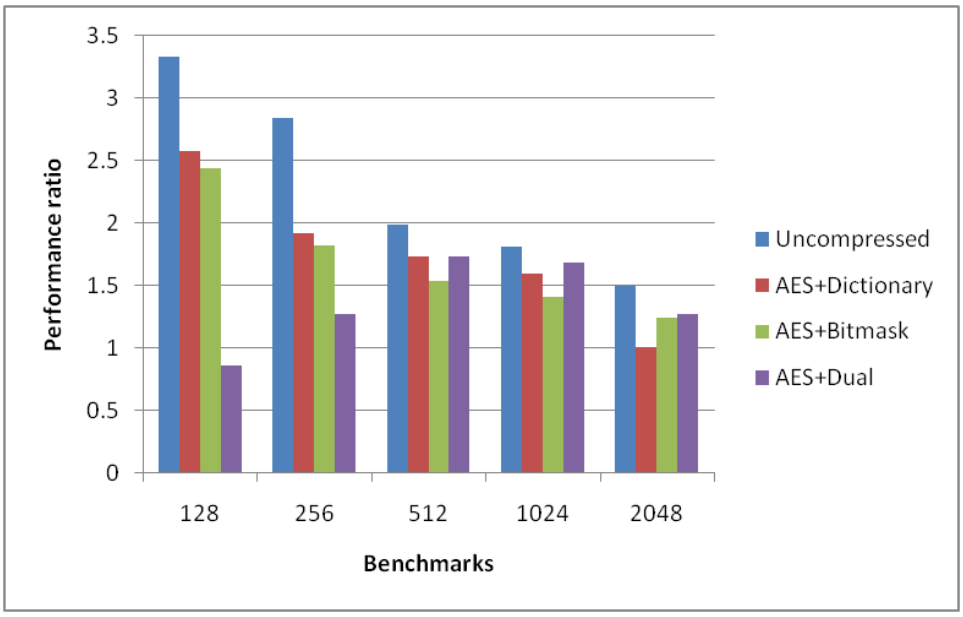

Figure 4-8. Performance ratios for DES for various cache sizes

Figure 4-10 shows the reduction in the number of cycles for various combinations of compression and encryption methods compared to the corresponding encryption method. Performance improvement is most noticeable for small caches. Small cache

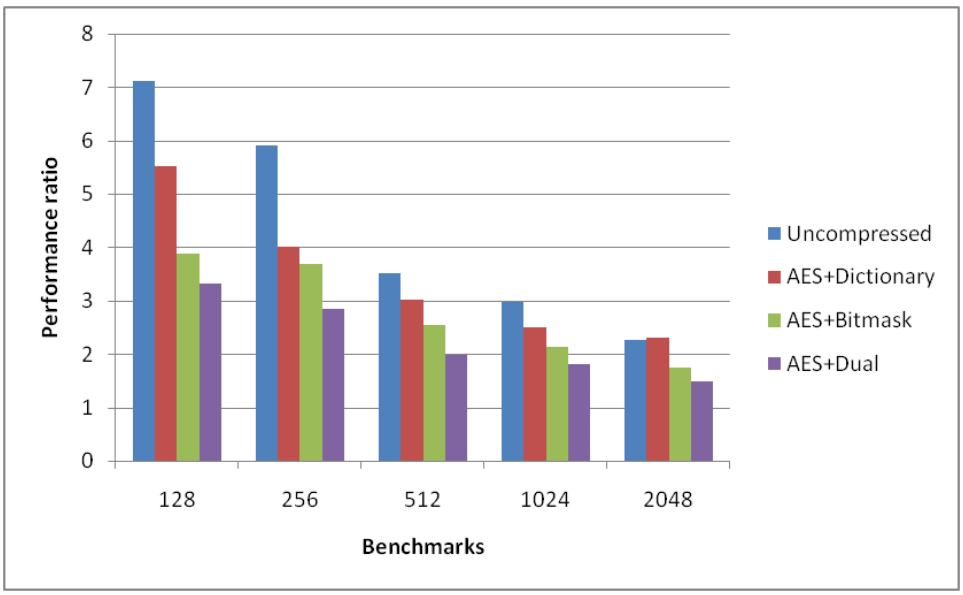

Figure 4-9. Performance ratios for AES for various cache sizes

means more number of fetches, hence the effect of compression would be more prominent.

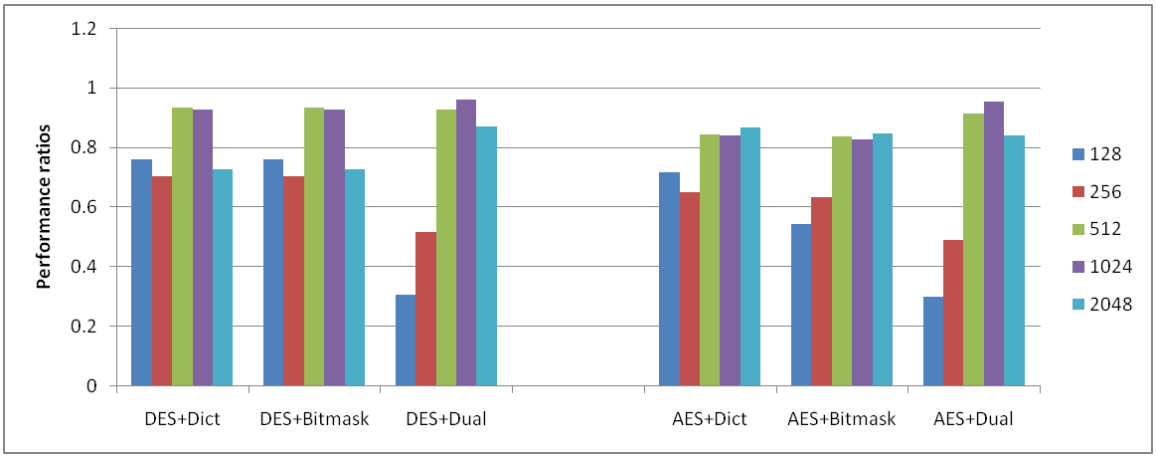

Figure 4-10. Ratio of execution cycles between compressed and uncompressed binaries

CHAPTER 5
CONCLUSION

Existing embedded systems are used everywhere; starting from day-to-day appliances to complex biomedical, military and other scientific equipment. Such systems need to be efficient (in terms of area, power and performance) as well as secure. This thesis described a novel dual code compression scheme. Code compression techniques can be used in embedded systems to either improve code size or performance. Through my proposed scheme of dual compression, we can simultaneously optimize code size and performance. Dual compression is split into two parts. Dynamic Frequency based Compression (DFC) improves performance by compressing the most frequently executing basic blocks. Static Frequency based Compression (SFC) exploits the most frequent static instructions and uses bit-mask based compression to reduce the code size. DFC compresses the original binary and provides a valid input for SFC as the word boundaries are maintained. DFC may cause a minor size reduction. The dynamic decompression for DFC is done between the cache and processor and that for SFC is done between cache and main memory. This way decompression is distributed and cache and memory space is efficiently utilized. SFC itself causes a minor speedup as fetching compressed code from the main memory would comparatively take less number of cycles. Experimental results demonstrate that dual compression reduces cache misses significantly for small caches and produces an average speed up of 50%, and achieved compression ratios from 60-65%.

The second part of this thesis presented a synergistic scheme of combining encryption and code compression. While the former provides security to application programs from reverse engineering and malicious manipulation, the latter is used to minimize the code size and thus reduce the memory requirements. This thesis analyzed the sequence in which compression and encryption should be done and showed that it is useful to first compress the code and then encrypt it, as then it will

have reduced code for encryption and decryption. The thesis also analyzed the effect of various parameters on the performance of such a system and the effect of cache. A complex encryption algorithm makes the effect of compression more prominent and the performance increase is proportional to the compression ratio. Large memory access latencies diminish the latencies of decryption and decompression and a smaller access latency, as is the case with embedded systems, makes their effect more visible. With the introduction of caches, it is always more profitable to place the decryptor before the cache as that would reduce its invocation rate. A large cache translates in to a lower miss ratio which means less number of accesses to the main memory and less number of invocations of decryptor, thus, decryption latency would be less prominent. Finally, a Processor-Decompressor-Cache-Decryptor (PDCD) architecture would give better results compared to a Processor-Cache-Decoder (PCD) architecture for a fast decompressor as in that case the cache would hold compressed instructions, thus, effectively increasing its size. Experimental results demonstrated that the execution time required is indeed less if encryption is combined with compression rather than if encryption had been done alone. Compression algorithms which give a better compression ratio, like bitmask-based over dictionary-based compression, gave better performance results. For example, bitmask-based compression gave a 17% reduction whereas that for dictionary based compression was 13% when combined with DES . Furthermore dual compression makes the best utilization of both the memory as well as cache and therefore gave the best performance result of 40% reduction in execution cycles with DES. The performance improvement due to compression was more apparent for small caches as more cache misses invoked the decryptor more frequently.

REFERENCES

[1] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Proceedings of the Institute of Radio Engineers (IRE)*, vol. 40, no. 9, pp. 1098–1101, September 1952.

[2] Welch, T.A., “Piparazzi: A test program generator for micro-architecture flow verification,” *Computer*, vol. 17, no. 6, pp. 8–19, June 1984.

[3] A. Wolfe and A. Chanin, “Executing compressed programs on an embedded RISC architecture,” in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 1992, pp. 81–91.

[4] H. Lekatsas and W. Wolf, “SAMC: A code compression algorithm for embedded processors,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 18, no. 12, pp. 1689–1701, December 1999.

[5] S. Nam, I. Park and C. Kyung, “Improving dictionary-based code compression in VLIW architectures,” *IEICE Trans. Fundamentals*, vol. E82-A, no. 11, pp. 2318–2324, November 1999.

[6] S. Larin and T. Conte, “Compiler-driven cached code compression schemes for embedded ilp processors,” in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 1999, pp. 82–91.

[7] Y. Xie, W. Wolf and H. Lekatsas, “Code compression for VLIW processors using variable-to-fixed coding,” in *Proceedings of International Symposium on System Synthesis (ISSS)*, 2002, pp. 138–143.

[8] C. Lin, Y. Xie and W. Wolf, “LZW-based code compression for VLIW embedded systems,” in *Proceedings of Design Automation and Test in Europe (DATE)*, 2004, pp. 76–81.

[9] D. Das and R. Kumar and P.P. Chakrabarti, “Dictionary based code compression for variable length instruction encodings,” in *Proceedings of VLSI Design*, 2005, pp. 545–550.

[10] S. Seong and P. Mishra, “Bitmask-based code compression for embedded systems,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27(4), pp. 673–685, April 2008.

[11] L. Benini, D. Bruni, A. Macii and E. Macii, “Hardware-assisted data compression for energy minimization in systems with embedded processors,” in *Proceedings of Design Automation and Test in Europe (DATE)*, 2002, pp. 449–453.

[12] H. Lekatsas and J. Henkel and V. Jakkula, “Design of an one-cycle decompression hardware for performance increase in embedded systems,” in *Proceedings of Design Automation Conference (DAC)*, 2002, pp. 34–39.

[13] E. Wanderley Netto, R. Azevedo, P. Centoducatte, G. Araujo, “Multi-profile based code compression,” in *41st Design Automation Conference*, 2004, pp. 244–249.

[14] M. Johnson, “On compressing encrypted data,” *IEEE Transactions on Signal Processing*, vol. 52, pp. 2992–3006, 2004.

[15] X. Ruan, “Using improved shannon-fano-elias codes for data encryption,” *Information Theory, 2006 IEEE International Symposium on*, pp. 1249–1252, 2006.

[16] A. Orpaz and S. Weiss, “A study of codepack: optimizing embedded code space,” in *CODES ’02: Proceedings of the tenth international symposium on Hardware/software codesign*. New York, NY, USA: ACM, 2002, pp. 103–108.

[17] C. Shaw, D. Chatterji, P. Maji S. Sen, B. Roy, P. P. Chaudhuri, “A pipeline architecture for encompression (encryption + compression) technology.” in *Proceedings of International Conference on VLSI Design*, 2003, p. 277.

[18] H. Lekatsas, J. Henkel, S. T. Chakradhar, and V. Jakkula, “Cypress: compression and encryption of data and code for embedded multimedia systems,” *IEEE Design & Test of Computers*, vol. 21, no. 5, pp. 406–415, Sep–Oct 2004.

BIOGRAPHICAL SKETCH

Kartik Shrivastava received his Bachelor of Technology in information technology from Malviya National Institute of Technology, India in 2008. He completed his Master of Science in computer engineering from University of Florida in 2010. Since summer of 2009, he has been working on code compression techniques for embedded systems at Embedded Systems Laboratory, University of Florida.