Dual Code Compression for Embedded Systems*

Kartik Shrivastava and Prabhat Mishra
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, Florida, 32611-6120, USA
kshrivas@cise.ufl.edu, prabhat@cise.ufl.edu

Abstract—**Computer architects aim to make embedded systems more powerful and space efficient. Code compression is traditionally used to reduce the code size by compressing the instructions with higher static frequency. However, it may introduce decompression overhead. Performance-aware compression techniques try to improve performance through reduction of cache misses by utilizing the dynamic instruction frequency, but it sacrifices code size. We propose a dual compression scheme that aims to simultaneously optimize both code size reduction and performance improvement. Experimental results show that our approach can simultaneously achieve best of both scenarios - achieves up to 40% compression efficiency and an average performance improvement of 50%.**

I. 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. Code compression addresses both of these requirements.

Compressing the application binary and decompressing it at runtime helps us better utilize the limited memory space in embedded systems. Figure 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 certain overhead which may increase the number of cycles for each fetch, which in turn 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 cycles on 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 can hold more instructions, hence increasing the effective cache size and causing a reduction in the miss rate.

Compression Ratio is widely accepted as the metric for measuring the efficiency of compression algorithms and is defined as: *Compression Ratio* = (*Compressed Code Size* / *Original Code Size*). Good compression ratio 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 static instructions may not be the most executed dynamic instructions 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 [1] which attempt to achieve both code size reduction and performance improvement. In these schemes, 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.

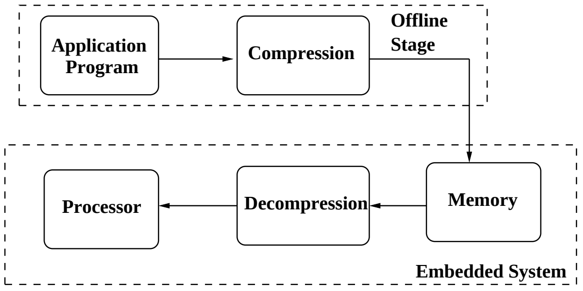

Fig. 1. Overview of Code Compression

In this paper, we describe a novel dual compression scheme which aims to simultaneously maximize the reduction in both the overall execution cycles and the binary size. 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. We present a detailed description of compression algorithm and decompression system with performance results and analysis. The framework is implemented in the SimpleScalar simulator and validated using MediaBench and MiBench benchmarks.

Rest of the paper is organized as follows. Section II surveys related work on code compression for both size reduction and performance improvement. Section III presents our dual code compression scheme, followed by experimental results in Section IV. Finally, Section V concludes this paper.

II. Related Work

Code compression techniques were first developed for embedded systems by Wolfe and Channin [2]. Lekatsas and Wolf

*This work was partially supported by NSF grant CNS-0915376.

used Arithmetic coding for code compression [3], whereas, Larin and Conte devised a Huffman based compression on embedded systems in [5]. Tunstall coding was used by Xie et al. [6] to perform variable to fixed length compression. Usage of variable sized block was further exploited by Lin et al. [7], 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. [8]. Seong et al. [9] improves dictionary-based compression by remembering mismatches using bitmasks. Bitmask based compression [9] has been successfully applied in various domains including manufacturing test compression [13] and FPGA bitstream compression [12]. All these works emphasize on reducing the 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. [10] 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. Lekatsas et al. [11] proposed a dictionary based technique for code compression, which takes advantage of compressing words with higher frequencies. However code size reduction is not targeted by these methods.

Netto et al. described [1] 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 word-sized sets of indices, removing any compressed word misalignments, giving a faster decompression. However, their work uses a single compression scheme. Therefore, for any combination of instructions from their dynamic and static profiles, it cannot achieve both best possible code size and performance at the same time. Our dual code compression can simultaneously achieve best possible code size reduction as well as best possible performance improvement.

III. Dual Code Compression

Dual code compression targets to optimize both system performance and code size reduction. Figure 2 show the overview of our framework. The difference between our approach and existing compression methods is 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

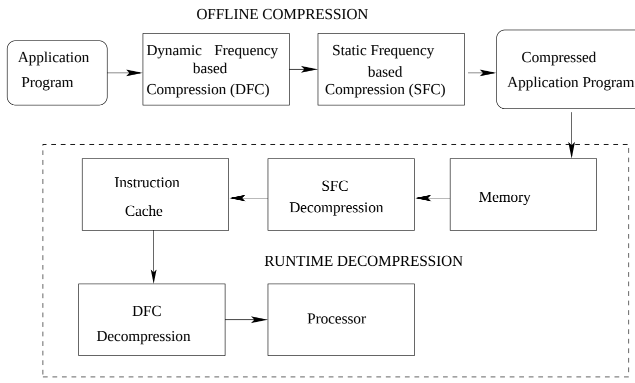

Fig. 2. Overview of Dual Code Compression

size. Figure 2 has four important steps: dynamic frequency based compression (DFC), static frequency based compression (SFC), SFC decompression and DFC decompression. The remainder of this section describes these steps in details.

A. Dynamic Frequency based Compression (DFC)

Algorithm 1 outlines the steps in DFC. 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.

Algorithm 1 Dynamic Frequency based Compression
```
1: Create profile P of most executed basic blocks
2: Create a 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
```

1) Profile creation: The first step in profile creation is the identification of basic blocks[1] and their relative access frequencies of being fetched. To identify the basic blocks and their respective frequencies we generate an execution trace of the program and calculate the frequency with which each instruction is fetched along with all the jump targets. The basic blocks are those sequences of instructions which have the same frequency of execution and no instruction as a jump target except the first one.

The next step in profile creation is selecting the most frequently fetched basic blocks for compression and creating a dictionary from them. Compressing the most frequently fetched basic blocks has the following advantages. Firstly, keeping the most frequently executed instructions in the cache in compressed form will help us better utilize its space and reduce the number of cache misses. If a basic block is

[1]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.

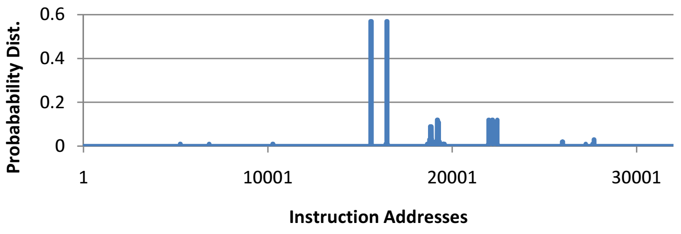

Fig. 3. Probability distribution of instruction fetches (cjpeg)

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. By analyzing the profiles of our benchmarks, we see that they follow the 90-10 rule, i.e., 90% of program execution time is spent on 10% of the code. Figure 3 shows a distribution of the instructions executed for *cjpeg*. Therefore, a small dictionary is sufficient. Figure 4 shows the percentage of fetches to the instructions in the dictionary for different dictionary sizes. For example, in epic benchmark, 256 most frequently fetched instructions makeup for 96.9% of the total number of fetches.

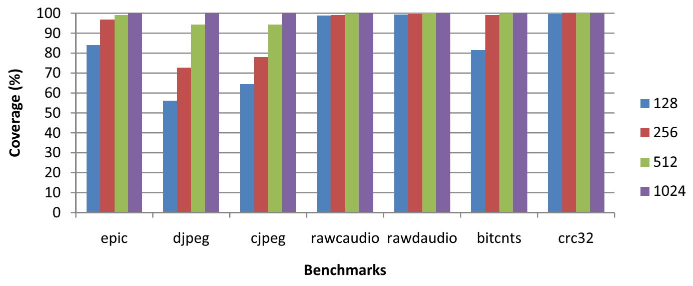

Fig. 4. Execution frequency of dictionary entries for various dictionary sizes.

2) Compression Mechanism: To compress the code we replace the instructions with their respective indices in the dictionary. As the target instruction set architecture here is Alpha, the instruction size is 32 bits 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 [14].

Figure 5 illustrates the compression mechanism. Instructions {1, 2, 3, 4, 5} and {8, 9, 10} are the basic blocks to be compressed 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 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.

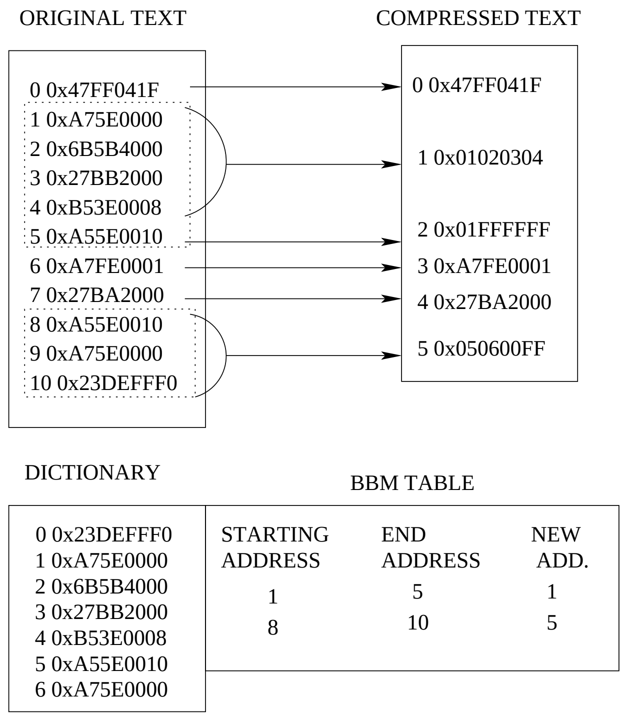

Fig. 5. Dynamic Frequency based Compression Mechanism

As the words do not contain any information as to whether it is compressed or not, a Basic Block Mapping (BBM) table is required. 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 whether the instruction is compressed or not. The flag bits/bytes (used in existing methods) spanned over the entire binary adds on to the size of the compressed binary. Moreover, the processing speed is also negatively impacted since even uncompressed instructions need to be flagged in existing methods. 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) as it is easy to map the jump target using the BBM table due to fixed encoding.

B. Static Frequency based Compression (SFC)

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. The compression mechanism used for SFC is based on the work done by Seong et al. [9] that uses a bit-mask based compression scheme which gives a high compression efficiency and enables single cycle decompression. Compression is performed on the DFC compressed code. As mentioned earlier, the word boundaries in DFC are maintained, hence, direct application of bit-mask based compression is possible to perform SFC. The compression mechanism is outlined in Algorithm 2.

It is useful to consider larger dictionary sizes when the current dictionary size cannot accommodate all the instructions 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 of high frequency vectors. Most importantly, a bigger dictionary increases access time and thereby reduces decompression efficiency. A standard dictionary size of 2048 is used in our implementation.

Algorithm 2 Static Frequency based Compression

```
1: Create Dictionary D2 using the most frequent words in C1 (produced by Algorithm 1) as well as bit-mask based savings.
2: Compress C1 using D2 to produce C2.
3: Handle and adjust branch targets in C2.
4: return  Compressed code and dictionary.
```

C. Decompression Architecture

As shown in Figure 2, DFC needs to be completed before SFC during compression (offline). Therefore, SFC decompression needs to be done before DFC decompression during runtime. Figure 2 shows the most beneficial placement of decompression units. There are various other combinations of placement of DFC and SFC decompression possible but they are less efficient. Placing both DFC and SFC decompression between cache and processor will negatively impact instruction fetch latency. Similarly placing them together between cache and memory would mean that the cache would hold uncompressed instructions, and therefore loose opportunities for improved cache hits, which is possible if cache contains compressed instructions. The details of SFC and DFC decompression mechanisms are discussed in the following sections.

1) SFC Decompression: We use the decompression architecture for bitmask-based compression developed by Seong et al. [9]. Unlike [9], we have placed the decompression engine for SFC before between cache and main memory. 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. Compressed blocks are fetched from the memory on cache misses, which are then decompressed and placed in the cache. The number 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.

2) DFC Decompression: Decompression for DFC is done between cache and processor to enable increase in cache utilization by making it hold the most frequently executed instructions in compressed form. This way, the effective size of the cache increases and the total number of cache misses reduces. 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. In this case, 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.

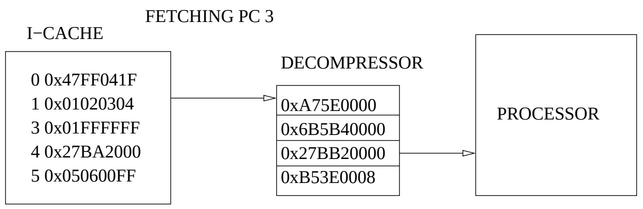

Fig. 6. Decompression of DFC compressed code for PC = 3

The runtime decompression unit uses the BBM table to see which instructions are compressed and to map the instructions to their correct addresses. When the decoder fetches a compressed word, it gets decompressed using the dictionary and sends back the required instruction to the processor and stores the rest in its buffer. As the word boundaries are maintained even after compression, fetching words from the cache is fast and simple. Figure 6 shows an example. Here we are fetching the instruction with the original PC 3. By looking at the BBM table, address 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}. 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.

Uncompressed instructions are also mapped to the correct address and fetched. Consider another example where PC is not part of the BBM table, for 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.

The simple mapping and decompression mechanism enables the decoder to fetch a compressed word, decompress it and store 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, i.e., by the prefetching instructions. 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.

IV. Experiments

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 programs (*rawcaudio*, *rawdaudio*), *bitcnts* from MiBench's automotive suite, and *crc32* from telecom suite. The simulation system consisted of a Super Scalar MIPS Processor, a decompressor each for DFC and SFC, an instruction cache with a line size of 16 bytes, and memory access time of 64 cycles. Table IV shows a description of the number of static and dynamic instructions for each benchmark used.

TABLE I
Profile of Selected Benchmarks.

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

A. Code Size Reduction

Figure 7 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 [9]. As expected, there is almost no size reduction in the DFC stage, therefore, SFC is exclusively responsible for overall code size reduction.

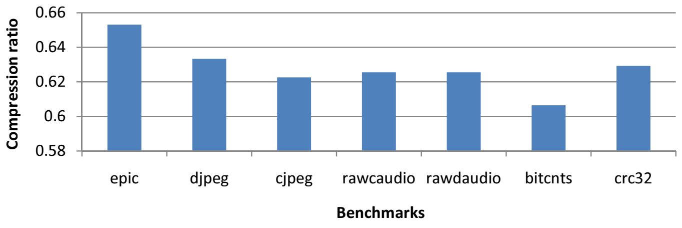

Fig. 7. Compression ratios for the benchmarks, using SFC

B. Performance Improvement

Table II shows the 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 an increasing 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 technique is beneficial in such environments.

TABLE II
Execution clock cycles for various cache sizes.

Benchmarks	Cache(bytes)	Clock Cycles	
		Original	Compressed
epic	128	291288151	125664394
	256	144591613	94542719
	512	94687156	81393578
	1024	80951000	67085506
	2048	63696412	51322191
djpeg	128	86863373	25427322
	256	46942406	19158549
	512	24810547	14050627
	1024	17415100	11420931
	2048	9579240	7054849
cjpeg	128	237649522	104696109
	256	113684344	90706713
	512	77921692	67695685
	1024	59294358	56290566
	2048	40666875	29184203
rawcaudio	128	83493514	4454026
	256	4584825	4429282
	512	4479397	4395664
	1024	4360331	4344856
	2048	4334726	4339501
rawdaudio	128	83493514	4454026
	256	4584825	4429282
	512	4479397	4395664
	1024	4360331	4344856
	2048	4334726	4339501
bitcnt	128	55195281	16122581
	256	30083364	9716254
	512	16747278	9728918
	1024	10459757	5248356
	2048	5338534	5220950
crc32	128	3697698	3574271
	256	3697698	3574271
	512	3662466	3601606
	1024	3576362	3538366
	2048	3535711	3513130

Performance improvement is more for benchmarks whose critical code (the most frequently executed instructions) is much larger than the cache. This could be seen 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 beyond 256 bytes, which implies that the cache easily accommodates the entire critical code, even in uncompressed form. Therefore, compressing the code in that cache configuration would not decrease the number of cache misses, hence no performance improvement is seen. In case of *bitcnts*, no performance improvement for cache size 2K was observed, as 2K cache holds the whole critical code. Moreover, its performance is equal to that seen for compressed code using 1K cache. This is because compressed version of the critical code fits entirely in a 1K cache but not in the uncompressed form which results in a performance improvement of two times in this case.

Figure 8 summarizes the trends in reduction in cache-

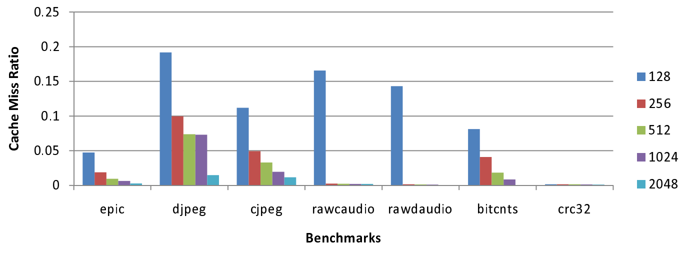

Fig. 8. Miss ratios for various cache sizes

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 9. In Figure 9, *performance improvement = (original execution time - execution after compression)/original execution time*.

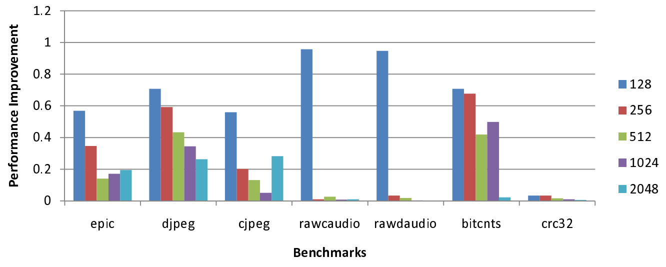

Fig. 9. Performance improvement for various cache sizes.

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. Figure 10 and 11 show the number of cycles (for *djpeg* and *rawcaudio*, respectively) 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 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 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.

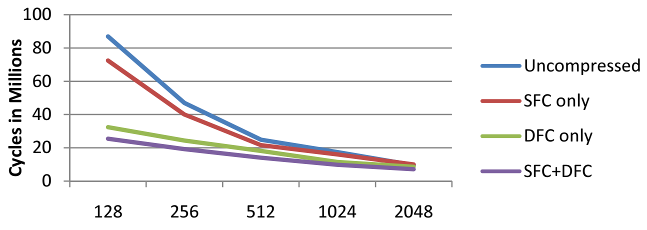

Fig. 10. Cycles for djpeg

Figure 10 shows the performance trend for *djpeg*. Figure 11 shows a similar trend for *rawcaudio*. The performance improvement is more apparent in case of *djpeg* because its 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 cache sizes larger than 128 bytes for rawcaudio.

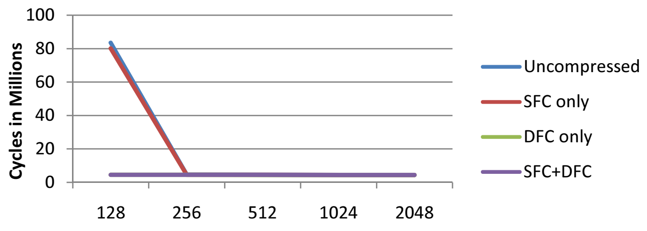

Fig. 11. Cycles for rawcaudio

V. Conclusions

Code compression is promising in embedded systems to improve code size and performance. Existing approaches can either target the best compression ratio or the best performance or make a trade-off. Our dual code compression method can simultaneously achieve both the benefits. Our results demonstrated that DFC reduces cache misses significantly for small caches which gives an average speed up of 50%. We have used a bit-mask based approach as SFC which gives compression ratio from 60-65%. DFC and SFC improve performance and code size respectively, they also complement each other. DFC itself reduces the code size slightly which gives an even better compression ratio than SFC alone. On the other hand reduced code size reduces the number of main memory accesses which further enhances the overall performance.

References

[1] E. Netto, R. Azevedo, P. Centoducatte and G. Araujo, “Multi-profile based code compression,” 244–249, *DAC* 2004.

[2] A. Wolfe and A. Chanin, “Executing compressed programs on an embedded RISC architecture,” *MICRO*, 1992, pp. 81–91.

[3] H. Lekatsas and W. Wolf, “SAMC: A code compression algorithm for embedded processors,” *IEEE Trans. on CAD*, 18(12), 1689–1701, 1999.

[4] Nam et al., “Improving dictionary-based code compression in VLIW architectures,” *IEICE Trans. Fundamentals*, E82-A(11), 2318–2324, 1999.

[5] S. Larin and T. Conte, “Compiler-driven cached code compression schemes for embedded ILP processors,” *MICRO*, 1999, pp. 82–91.

[6] Xie et al., “Code compression for VLIW processors using variable-to-fixed coding,” *ISSS*, 2002, pp. 138–143.

[7] Lin et al., “LZW-based code compression for VLIW embedded systems,” *DATE*, 2004, pp. 76–81.

[8] Das et al., “Dictionary based code compression for variable length instruction encodings,” *VLSI Design*, 2005, pp. 545–550.

[9] S. Seong and P. Mishra, “Bitmask-based code compression for embedded systems,” *IEEE Trans. on CAD*, 27(4), pp. 673–685, 2008.

[10] Benini et al., “Hardware-assisted data compression for energy minimization in systems with embedded processors,” *DATE*, 2002, pp. 449–453.

[11] Lekatsas et al., “Design of an one-cycle decompression hardware for performance increase in embedded systems,” *DAC*, 2002, pp. 34–39.

[12] X. Qin, C. Murthy and P. Mishra, “Decoding-aware Compression of FPGA Bitstreams,” *IEEE Transactions on VLSI*, 2010.

[13] K. Basu and P. Mishra, “Test Data Compression using Efficient Bitmask and Dictionary Selection Methods,” *IEEE Transactions on VLSI*, 18(9), 1277-1286, 2010.

[14] X. Qin and P. Mishra, “A Universal Placement Technique of Compressed Instructions for Efficient Parallel Decompression,” *IEEE Transactions on CAD*, 28(8), 1224-1236, 2010.