Dual Selective Code Compression

Eduardo B. Wanderley Netto
CEFET-RN, Dept. Informatics,
Natal, Brazil, 59015-000
braulio@cefetrn.br

Eduardo A. Billo
UNICAMP, Institute of Computing,
Campinas, Brazil, 13084-971
eduardo.billo@ic.unicamp.br

and

Rodolfo J. Azevedo
UNICAMP, Institute of Computing,
Campinas, Brazil, 13084-971
rodolfo@ic.unicamp.br

Abstract

Code compression has been shown to be efficient in code size reduction and, recently, performance improvement and energy savings. In this paper we use a compression method, the ComPacket, which has a very fast decompressor in hardware, to compress selectively regions of the code to improve performance and complementary regions to sustain the code size reduction both at the same time. Using the leon (SPARC v8) platform and benchmarks from Mediabench and MiBench suites we reached, on average, 25% of code memory area reduction, and a speed-up of 1.5 simultaneously.

Keywords: Computer Architecture, Code Compression, Compression, Performance.

1. INTRODUCTION

As the embedded systems market grows up, the software applications become more and more sophisticated requiring huge amounts of memory. On the other hand many embedded devices require strict energy consumptions and performance. RISC processors are commonly found integrating such devices and the code density of such processors is not their best feature.
In this scenario, several code compression approaches have been used to cope with the RISC density code problem and sometimes they also achieve energy consumption savings and performance improvements.
Techniques from data compression field have been considered as the base to derive methods that can be applied to code. However, special requirements like random access and on-the-fly (thus, fast) decompression discard some outstanding data compression methods.
The first approach to compress code for RISC processors is due to Wolfe and Chanin [1]. The CCRP used a relative slow decompressor in hardware located between the Cache and Main Memory, thus hiding the decompressor latency. It is usual that a compression method that produces a high density code implies in a slow decompression. That is why Wolfe and Chanin, and many other researches after them, have chosen to hide this latency behind the cache [2,3,4].
Nevertheless, this decompression overhead is usually large enough to negatively interfere in the system performance. Selectively choosing instructions or regions to be compressed can help in minimizing the decompressor impact by selecting rarely used instructions to be compressed, favoring hot-spots of code execution to remain intact and performing as good as in an uncompressed execution code system.
Performance improvements can be achieved by relocating the decompressor engine to between the processor and the cache, so that, the code is stored compressed in cache increasing its hit ratio and avoiding many accesses to the main memory. The same reasoning can be used to justify energy savings.
This last model, called PDC (Processor-Decompressor-Cache), requires a very fast decompressor hardware as it is located on the processor critical path and nothing else can hide its latency. The main implication in this approach is that the code compression method has to be simple enough to require a fast decompressor.
In this work we introduce selectivity to our code compression method called ComPacket[5], which was specially designed to be a PDC, and has a fully functional implementation in FPGA[6]. The ComPacket uses an incomplete dictionary to hold instructions selected by any criteria. These instructions are changed in the original code for packets of indexes into the dictionary.
Unlike the selective goals aforementioned, we wanted to focus in the code execution hot spots for a better compression thus improving cache hit ratio and so, the system performance.
We have realized that, when using an incomplete dictionary based code compression method, the dictionary formation drives the results for performance and area. Using a dictionary built upon the count of the instructions in the code benefits code size reduction and using a dictionary which is composed with instruction that is executed the most, benefits performance. These dictionaries we call Static Dictionary (SD) and Dynamic Dictionary (DD) respectively.
As far as SD and DD use to hold different instructions, we use both of them in our selective approach. The DD is used for inner-loops and the SD for the remaining code. The assumption is that, for loop-intensive applications, a specialized dictionary for inner-loop will enhance the performance improvement and the remaining regions, which use to be rarely executed (following the 90/10 rule of the thumb) but represent the biggest part in the code, will be compressed with a SD specially designed for them which will promote better compression.
This approach reduces the code up to 60% of its original size and produces speed-ups as big as 2 for loop-intensive applications from Mediabench[7] and MiBench[8] benchmark suites running in a SPARC v8 architecture (Leon Processor).
This paper is organized as follows: Section 2 discusses the related work. Section 3 describes the compression method used. Section 4 explains the selectivity in our approach. In Section 5 we describe the experimental setup followed by the results we obtained in Section 6. Finally, Section 7 presents our conclusions.

2. RELATED WORK

In this section we present the PDC and Selective code compression methods that are closest to our work. Moreover, we define the Compression Ratio as the ratio between the compressed code size (including the dictionaries sizes when apply) over the original code size.
Benini et al [9] used a dictionary based compression method formed by using dynamic profiling information. A small 256 entries dictionary with one instruction per entry is used to keep the most executed instructions. They compress instructions that belong to the dictionary if they can be accommodated in a cache line size package. A 75% compression ratio is produced for the DLX. Also, a 30% energy reduction was obtained. Unfortunately, cache access time is increased in 32% as the decompressor is docked to the cache.

Lekatsas et al [10] used the Xtensa 1040 as the underline processor to support a small dictionary compression method based on static instruction count. The authors used variable-length codewords of 8 and 16 bits to compress the original 24 bit instructions. Their primary goal was to guarantee that the decompressor engine requires no more than one cycle to decompress one or two codewords. Some decompression overhead comes from the fact that the engine is supposed to keep fractions of misaligned instructions or codewords that come from the cache. Moreover, the dictionary was doubled (2x256-entry) to support two codewords decompressions per cycle. A 35% code size reduction was achieved and a 25% performance improvement (cycles count reduction) was reported.
Since the first code compression work [1] selectivity has been considered to minimize the execution time overhead due to the decompressor. Nevertheless, two works in special focus the selective code compression methods: the first by Debray and Evans [11] used a threshold value (% of execution time related to the total execution) to limit the number of functions that are compressed. Only the functions that contribute less with execution time are compressed. This method has a decompressor in software thus no hardware support is needed.
The second specialized selective code compression method is the work by Xie et al [12]. They use an arithmetic coding which requires a long latency to the decompression. The decompressor in hardware is avoided for traces that are often used. Both selective work presented here are very sensible to the threshold used to limit the code regions to be compressed. If just a small portion of frequently executed regions is compressed, the performance is strongly affected. Compression ratios vary from 72% to 88% depending on the threshold used. Both used dynamic profile information to setup the threshold.
Our compression method differs from previous work by using selectivity not to avoid performance penalties but to enforce the execution of compressed instructions in inner-loops and, at the same time, using two dictionaries with different goals and formations to guarantee good compression ratios and performance simultaneously. In fact, our selective approach can be applied to any incomplete dictionary based code compression method. The ComPacket method was used as a case study, and it was chosen because we authored it, so that it is on the shelf.

3. THE COMPACKET

The ComPacket compression method is named after the main unit used for compression: the Compressed Packet (ComPacket). This ComPacket holds a set of indexes into an instruction dictionary. We use an invalid opcode of the underling architecture (SPARC v8) to signalize that a ComPacket has been fetched. The encoding of the ComPackets favors regularity and can be easily identified by the decompressor. All the ComPackets are sized 32 bits and hold from 2 to 4 indexes. Only 4 ComPacket formats are supported, presented in Figure 1. The escape sequence uses 4 bits to identify an invalid instruction; 2 bits to inform which instruction is supposed to be executed first in the case of the code stream execution reaches the ComPacket by a branch instruction; 1 bit to inform the size of the indexes in the ComPacket (6 or 8 bits); and 1 bit to signalize the presence of a branch instruction. This escape sequence is uniformly located in all ComPackets. The method allows misaligned branch targets by using the TT pair bits to signalize that the decompression should begin from any Index inside the ComPacket. Currently, only one target is allowed inside a ComPacket. It is also allowed to hold one branch instruction index inside the ComPacket restricted to an offset of 8 or 6 bits depending on the format used.

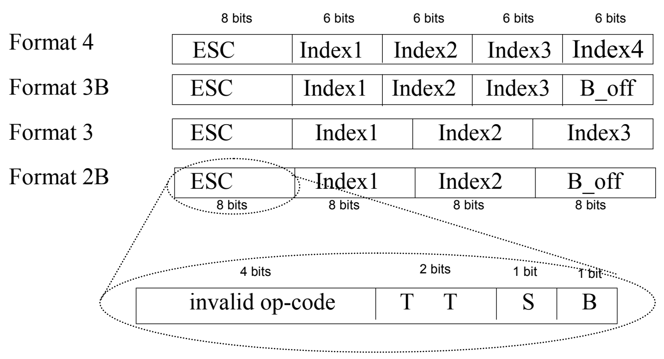

Figure 1: ComPackets formats

When a ComPacket is fetched into the decompressor it is recognized as a ComPacket and the correspondent instructions are fetched from the dictionary and delivered to the processor in order.
The ComPacket can use dictionaries of up to 256 entries, but some formats (Format 4 and Format 3B) uses only the 64 first entries.
The process for building the dictionary is completely independent of the encoding, but searching instructions that better suits the formats, like code patterns with 4, 3 or 2 instructions, will produce better compression results.
Some restrictions apply to this encoding: at most one instruction is supposed to be a branch in the ComPacket and this branch has a branch offset that can be represented with 6 or 8 bits depending on the format chosen. Another restriction is that only one instruction in the ComPacket can be the target of a branch.

3.1 The compression algorithm

To use the ComPackets we outlined a compression algorithm, shown in Figure 2, that begins by finding the executed inner-loops and build a DD for them, based on the dynamic information retrieved from the profile. Similarly, the remaining code is scanned to form another dictionary, the SD. Once the dictionaries are built (step1), the compressor scans the code to identify branches/calls/jumps and mark them and their targets (step 2). Then, it scans again the code trying to match a ComPacket 4 for the first addresses. If it is possible, it marks the instructions to a ComPacket4. If it is not possible for a ComPacket4, try a ComPacket3, 3B and a ComPacket2B in this sequence. Then, continue scanning the code until all instructions are investigated (step 3).

```
Compress()
  1. Build the Dictionaries
  2. Code Marking
  3. Scan the Code
        a. Try to mark Format 4 ComPacket
        b. Try to mark Format 3/3B ComPacket
        c. Try to mark Format 2B ComPacket
  4. Assembly ComPacket formats marked in 3.
  5. Replace ComPackets in the code
  6. Allocate change dictionary instructions
  7. Patch addresses
```

Figure 2: Compression method algorithm

The next step (step 4) assembles the ComPackets marked. Then, they are inserted in the code replacing the original instructions (step 5). Next, it allocates room for a new *change dictionary* (`ChgDict`) instruction. This instruction is allocated in each inner-loops pre-header and after the post-dominator, so that the decompressor understands which dictionary must be used at a given time (step 6). Finally, all branch/call/jumps offsets are adjusted to the new targets positions (step 7).
The adaptation to the original ComPacket compression algorithm is only the inclusion of step 6, which deals with the new ChgDict instruction. This instruction is supposed to affect only the decompressor hardware switching the dictionaries. It is not recognized by the processor itself. Every time a ChgDict instruction is found by the decompressor it issues a bubble to the processor pipeline. These instructions are rarely found by the decompressor as they are outside the inner-loops and the applications spend most of their execution time inside the inner-loops.
Notice that not all the instructions that belong to the dictionary are compressed in the code. To be worth, at least two neighbor instructions that belong to the dictionary are changed by one ComPacket.

4. SELECTIVE CODE COMPRESSION

The goal of our selectivity is to find regions of the code that executes the most and constitute a specialized dictionary for those regions. For the remaining regions we find another dictionary also specialized for them. For loop-intensive applications, which is commonly found in real systems, the inner-loops often dominate the execution time of the code.
Table 1 presents the contribution of the inner-loops in the code execution for our set of benchmarks. It is also depicted the number of distinct instructions that belongs to this inner-loops. We can see that, for some applications, a dictionary of 64 entries is enough to accommodate all the inner-loops instructions. Nevertheless, the encoding restrictions do not allow all the instructions to be compressed.

Table 1: Inner-loops

	Instructions Executed	Time in Inner-loop. (%)	Distinct instructions in inner-loops
Search	8,070,065	51	63
Pegwit	32,976,116	63	315
Djpeg	3,707,977	84	558
Cjpeg	15,070,171	61	668
Apcm_enc	9,527,331	78	65
Apcm_dec	7,091,219	91	57

The selection of a dictionary for inner-loops is done by having performance in mind; so that the most compressed instructions in the inner-loops, the most is the benefit with performance. If we have to select instructions to be in this dictionary, it should be done by their execution contribution avoiding traces that executes rarely inside the inner-loops. This is the reasoning behind our choice of using a dynamic dictionary for inner-loops.

The dictionary for the remaining code should be built upon the compression goal. In this case a static dictionary performs better.

Figure 3 justifies why we are not using only one dictionary. The problem is that SD and DD are very different. On average, only 16% of the instructions in SD belong also to DD for 256 entries dictionaries. This figure presents the dictionaries similarities for small dictionaries sized up to 256 entries. Note that most of the instructions that belong to DD do not belong to SD so that having two dictionaries will benefit performance and compression at the same time.

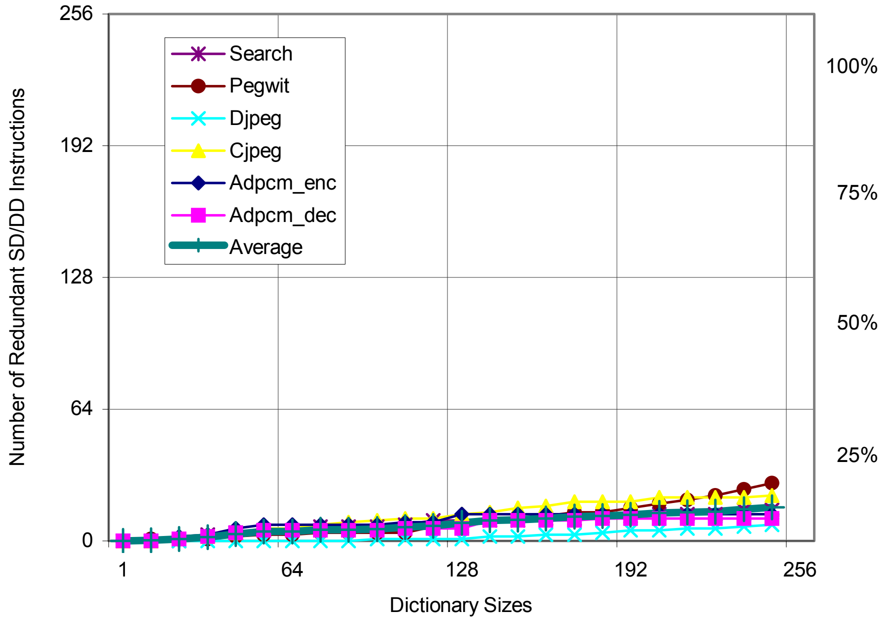

Figure 3: SD and DD similarities

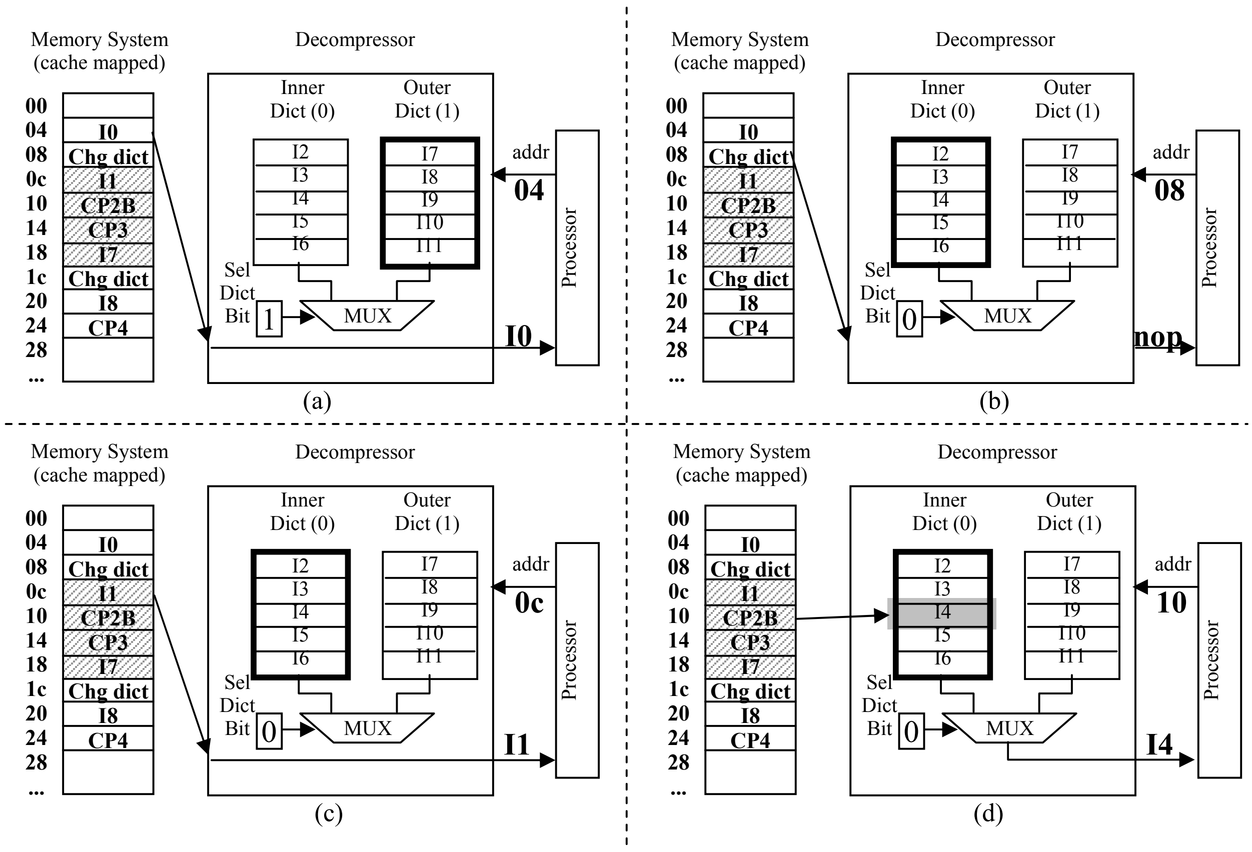

Figure 4: Decompressor Scheme

The scheme of using two dictionaries is depicted in Figure 4. In fact, it is an abstraction of the real (much more complex) implementation. The inner-loop is hachured in the memory system. Initially the bit Sel Dict, inside the decompressor is always set up to 1, which means that the execution stream is not inside an inner-loop. The processor asks for the instruction at address 0x04 (Fig. 04(a)). The decompressor fetches the instruction from main memory and realizes that it is not a ComPacket. Instruction I0 is delivered to the processor as is. Then, the processor asks for the instruction at address 0x08. This is a ChgDict instruction, which means that an inner-loop will follow. The decompressor understands the instruction, changes the Sel Dict Bit to 0, and inserts a bubble in the processor pipeline by issuing a nop instruction (Fig. 04(b)). Then, the processor asks for the next instruction. The decompressor realizes that it is an uncompressed instruction and delivers it to the processor (Fig. 04(c)). Finally, the processor asks for the instruction at address 0x10. In this address a ComPacket holds indexes to the dictionaries. In this situation, the decompressor will fetch from both dictionaries the first index inside the ComPacket, say index into entry 2. As far as the dictionary 0 (inner-dict) is selected in the MUX the instruction that will be delivered to the processor is I4 (Fig. 04(d)). The next instruction asked by the processor will be already inside the decompressor (in ComPacket 2B (CP2B)), so that no fetch from main memory is required.

When the processor reaches the end of the inner-loop at address 0x1c the decompressor switches the MUX selection and the outer dictionary is activated again. If the ComPacket at address 0x24, for example, holds an index into entry 2, the instruction that is supposed to be delivered to the processor is I9, not I4.

In our approach, instead of using just one region of the code, like the other selective approaches in the literature, we use all the regions, but we select which dictionary is supposed to be used at a certain time in execution. We named this selectivity *dual*.

5. Experimental Setup

We used a simulator of the Leon processor (SPARC v8) [13] developed in our lab to support our experiments. The benchmarks are extracted from MiBench[8] and Mediabench[7]. They are a string search algorithm, Search, commonly used in office suites; Djpeg and Cjpeg, used for compressing and decompressing images from and to JPEG formats; Adpcm encodes or decodes audio; and Pegwit, an encryption tool.
We used LECCS, a GCC based cross compiler for the Leon processor, with –O2 option in all the benchmarks, so that we avoid typical optimizations that increase object code size, like function in-lining and loop unrolling.
A binary rewriting tool was developed to read an original code, perform the compression and write the new binary compressed code. It is fed with profile information and a Control Flow Graph generated by the simulator.
We established a memory hierarchy with the main memory sized 1Mbytes and each access costs two clock cycles (8 cycles to delivery 4 instructions). In fact, this main memory setup can be considered a very fast one. Our intention is to demonstrate that even if we experience a low miss penalty, in terms of clock cycles, code compression can produce better performance results than the original execution.
The caches are chosen by the original code execution behavior. Commonly, the cache size is decided at design time based on the point in which a doubled sized cache does not improve the hit ratio considerably. In our case, we selected the hit ratio threshold for 0.9, so that every cache must miss at a maximum of 10%. Also, if we doubled the cache size the difference for cache hit ratio must be lower than 5%. We relaxed the first threshold for the adpcm codes because even with a very small cache most of the misses are compulsory, so that once the code is read by the cache for the first time it will produce hits for practically all the execution time. Choosing the cache size by a percentage of its performance allows us to infer the results for larger applications (and larger caches).

6. Results

The first measurements show the Compression Ratio for the benchmarks used. We experienced with many combinations of dictionary sizes from 0 to 256 (0, 32, 64, 128, 256 entries) for inner-loops and outside inner-loops regions. Table 2 presents the best results for compression (obtained by the adpcm decoder benchmark) and table 3 presents the average results.
These tables present the compression ratios with and without the dictionary size included in the compressed code size. This allows us to measure the overhead implied by the dictionary on the compressed code. For example: using a inner-loop dictionary with 64 instructions and the outside dictionary with 128 instructions we reach a compression ratio of 52% for the adpcm without including the dictionaries sizes and 60% if we include them. Thus, the dictionaries represent 8% in the compression ratio.
Note that, when using only one dictionary for outside the inner-loops (inner size = 0, e.g. first data row) the compression ratio is as good as 47% (58%) for the adpcm. As the dictionary grows, the compressed code becomes smaller, but the dictionary itself influences more in the compression ratio. On average, the compression ratio for this set of benchmarks is 70% (or 74% if the dictionary is counted). On average, the dictionaries represent up to 5% of the compressed code, depending on their sizes and combination used.
When we use only the dictionary for inner-loops (first data column, e.g. Outside = 0) the compression is simply unexpressive. Note that, as the inner-loop dictionary increases, the compressed code (without counting the dictionary size) is not influenced. This revels that increasing the inner-loops dictionaries sizes do not imply in more compression and then it is not suitable to use a big dictionary here.

Table 2: Compression Ratio for adpcm_dec for various dictionaries sizes

Outer	**0**		**32**		**64**		**128**		**256**	
Inner	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w
0	100	100	61	62	58	61	53	59	47	58
32	99	100	60	63	57	61	53	60	46	58
64	98	101	59	63	56	62	**52**	**60**	45	59
128	98	104	59	66	56	65	52	63	45	62
256	98	109	59	72	56	70	52	69	45	62

Table 3: Average Compression Ratio

Outer	0		32		64		128		256	
Inner	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w
0	100	100	80	81	78	79	75	77	70	74
32	99	100	80	81	78	79	74	77	69	74
64	99	100	80	81	77	80	74	77	69	75
128	99	101	79	82	77	81	74	78	69	76
256	99	103	79	85	77	83	74	81	69	76

The next experiments show how performance is affected by code compression. In fact, the most compressed is the code, the less instructions are fetched from the main memory (due to cache hit ratio improvement). The memory latency is also a factor to be considered: the slower the main memory, the better is the benefit of compression, as we do not need to wait for the original amount of fetches, but just the compressed one.

We have also selected the performance measurements for a set of dictionaries sizes combinations. As some applications require an inner-loop dictionary smaller than 66 instructions (see Table 1) we used a 64 entry dictionary for the inner-loops to observe the influence of the outside dictionary on performance. Figure 5 shows the speed-ups obtained.

Note that performance is rarely affected by the outside inner-loops dictionary. Then, the inner-loop dictionary is responsible for the performance benefits of the compressed code.

This figure depicts also the magnitude of the performance benefits we can obtain with code compression. On average, 1.5 is the speed-up for a 64/256 (inner/outside) dictionaries sizes combination.

To realize how good this dual selective approach is comparing it with others, we measured the performance and compressibility of the same benchmarks without selectivity. We compressed all the code with only one SD and also with only one DD. These dictionaries have 256 entries and are strongly optimized for area and performance benefits, respectively.

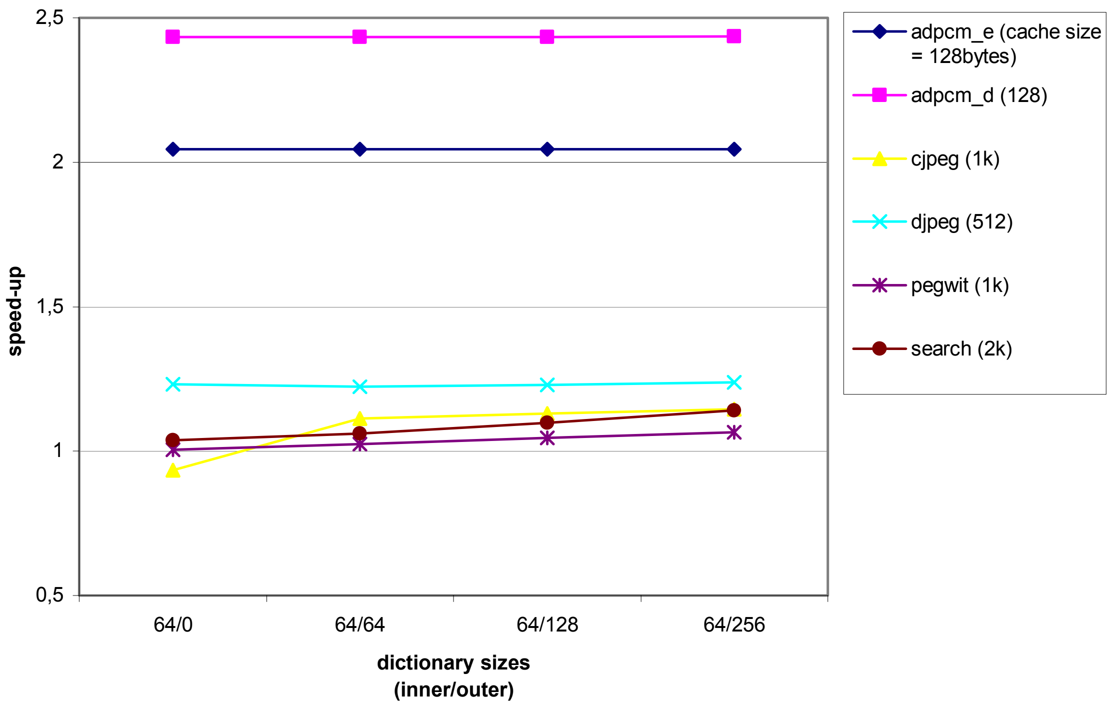

Figure 5: Performance speed-ups

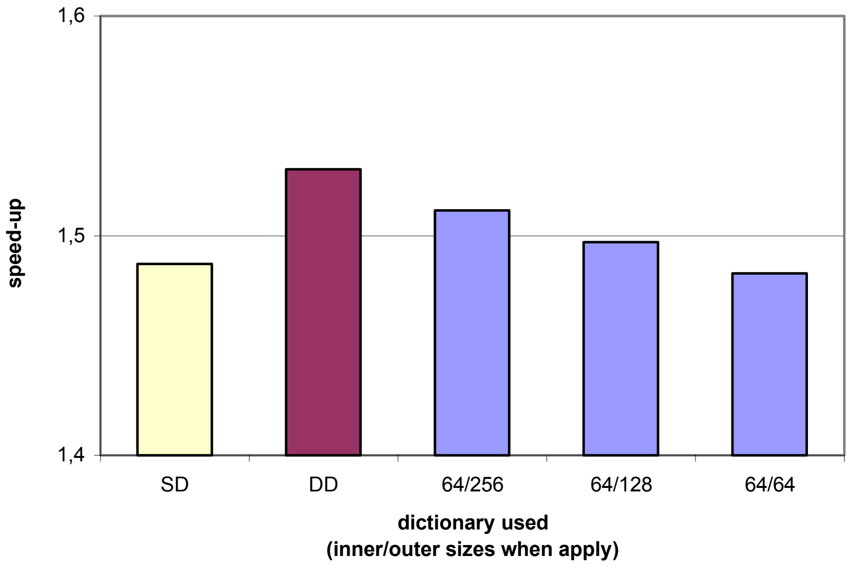

Figure 6: Speed-ups comparison

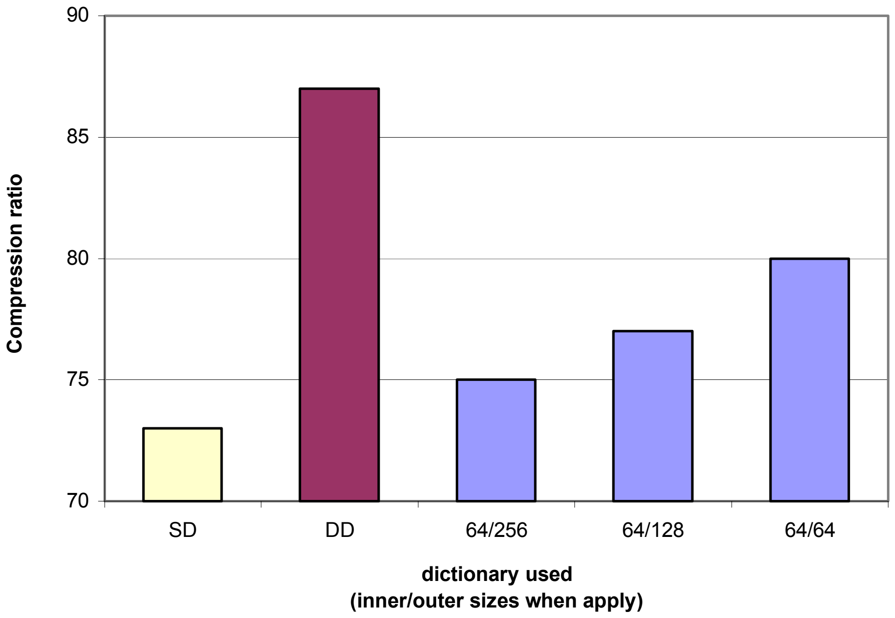

Figure 7: Compression ratio comparison

Figure 6 shows the comparative results for performance for a set of dictionaries sizes combinations. Observe that with selectivity (64/256 combination) we outperform the results obtained for SD. Even with a smaller area used by the dictionaries (64/128 case, compared with the 256-entry SD) we still outperform the SD approach and, in a case where we halve the number of dictionaries entries with selectivity (64/64 compared to 256 for SD), we still have the same performance of the SD.
For the DD, which is a specialized dictionary for performance, this approach is not as good. But unfortunately, the penalty we have to pay for performance using the DD is a small compression as we can see in Figure 7. The DD do not have a good compression ratio compared to the SD. The dual selective approach, on the other hand, is almost as good as SD for compression.

Thus, we conclude that, using dual selectivity, we can have good performance and at the same time we save memory area for the code, approaching the best results simultaneously.
Another conclusion is that using smaller dictionaries than the usual 256-entry is not so prohibitive, neither for performance nor for chip area savings.
This is the first approach with selectivity (using inner-loops) to promote performance improvement and it is independent of the compression method utilized. In this paper we are not comparing the performance of the ComPacket method in terms of compression or performance to other PDC approaches, as it is not introduced here. In fact, we used ComPacket just to compare the pure static and pure dynamic approaches with this mixed one. Comparing with our former method [5], which builds only one dictionary based on static and dynamic measurements, the dual selective approach, introduced here, reached similar result for compression and performance with smaller dictionaries.

7. DISCUSSION

This method is as good as the application is loop-intensive. One of its drawbacks is for applications that do not use inner-loops very often, like susan, and dijkstra, others benchmarks from the suites utilized. We still have to cope with the problem of selecting instructions from the inner-loops when they do not fit in a small dictionary like those used by the ComPacket. By this time we simply used a DD approach, but maybe some traces inside the inner-loops could be priority.
As far as this selective approach is used in other compression method, the 256-entry dictionary limit can be relaxed. In fact, ComPacket indexes codification limits the maximum dictionaries sizes to be used. This small dictionary size, on the other hand, is important not to impact the total compressibility of the code and to make it fast to fetch instructions from the dictionary.
The dictionaries are supposed to be loaded from each application memory footprint. This will happen in the beginning of the execution which means that the performance penalty over millions of executed instructions is just inexpressive. This is reasonable in embedded systems, as far as the applications are well known at design time and trend not to change over time.
Performance improvements come from the compressed instruction in cache, avoiding main memory fetches, usually slow. In fact, compression will improve spatial locality in cache, which implies that we can reach better performance with compression or we can use smaller caches and still reach the same performance as before.
A key aspect in performance comes from the fact that introducing a decompressor between cache and processor will probably consume clock cycles. Nevertheless, the implementation of the ComPacket method in FPGA [6], integrated the decompressor with the processor pipeline so that no extra cycle is necessary to decompress an instruction. In this scenario the decompressor itself does not implies in performance penalties.
Energy consumption is another key aspect in embedded systems and uses to follow the behavior of the performance (as far as main memory accesses are reduced, and energy per accesses is still the same) when using code compression, so that we believe that this method will help in energy savings as well.

8. CONCLUSIONS

So far we have presented a selectivity approach to code compression to enhance the performance of the system without paying the also restrictive price of area in embedded systems. Our dual selectivity favors performance by allocating a specialized dictionary for the inner-loops of an application, thus providing better hit ratios for the cache system. Another specialized dictionary is used for the other regions of the code to guarantee a good compressibility.
Using the leon (SPARC v8) platform and benchmarks from Mediabench and MiBench suites we reached, on average, 25% of code memory area reduction, and a speed-up of 1.5 simultaneously. We believe that energy consumption will also benefit much by the code compression usage, and this is one of future works.

References

[1] A. Wolfe and A. Chanin, Executing Compressed Programs on an Embedded RISC Architecture. In Proc. of ACM/IEEE Annual International Symposium on Microarchitecture, pp. 81-91, Nov. 1992

[2] D. Kirovski, J. Kin and W. Mangione-Smith. Procedure Based Program Compression. In Proc. of ACM/IEEE Annual International Symposium on Microarchitecture, pp. 194-203, Dec. 1997

[3] T. Kemp, R. Montoye, D. Auerbach, J. Harper, J. Palmer, A Decompression Core for PowerPC, IBM Journal of Research and Development 42(6):807-812, Sep. 1998.

[4] G. Araujo, P. Centoducatte, R. Azevedo, and R. Pannain. Expression tree based algorithms for code compression on embedded RISC architectures. IEEE Transactions on VLSI Systems 8(5):530-533, Oct. 2000

[5] E. W. Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Multi-profile based code compression. In Design Automation Conference, DAC04, 2004 pp 244-249

[6] E. Billo, E. W. Netto, R. Azevedo. Design of a Decompressor Engine on a SPARC Processor. Submitted to SBCCI 05

[7] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia communication systems. Pp. 330–337, Dec. 1997.

[8] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, and T. Mudge. Mibench: a free, commercially representative mbedded benchmark suite. In Proc. of the IEEE 4th Annual Workshop on Workload Characterization, pp. 3–14, Dec. 2001.

[9] L. Benini, A. Macci and A. Nannarelli, Cached-code compression for energy minimization in embedded processor. Proc. of ISPLED'01. pp. 322-327, Aug 2001

[10] H. Lekatsas, J. Henkel and V. Jakkula. Design of one-cycle decompression hardware for performance increase in embedded systems. Proc. of DAC'02. pp. 34-39, Jun 2002.

[11]S. Debray and W. Evans, Profile-guided Code Compression, In Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, 37(5):95-105 Jun 2002.

[12]Y. Xie, W. Wolf and H. Lekatsas, Profile-driven Selective Code Compression, In Proc. of the Design, Automation and Test in Europe Conference and Exhibition, pp. 462-467, Mar 2003

[13] G. Gaisler. Leon. [OnLine], Oct. 2003. Available: http://www.gaisler.com.