Code Compression Based on Operand-Factorization for VLIW Processors

Montserrat Ros, Peter Sutton
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia QLD 4072
{ros,p.sutton}@itee.uq.edu.au

Abstract

As programs become more complex for both embedded systems and large-scale applications, bloated code size continues to be an ever increasing problem. The size of object code greatly effects the space used and ultimately contributes greatly to cost. Code compression techniques have been devised as a way of battling large code. Code compression algorithms usually require specific techniques to maintain the integrity of the program and ensure its functionality.

This paper presents three code compression algorithms based on dictionary methods for entire instructions, instructions factorized into op-code/operand pairs and operand factorization applied to instruction words. MediaBench[1] benchmarks are compiled for maximum code optimization on the TI TMS320C6x then compressed. Compression Ratios (defined as the ratio of compressed code to uncompressed code) of 81.5%, 68.3% and 84.7% are reported for the three compression schemes.

The instruction-based factorization scheme outperforms the other schemes in relation to code size, but requires sequential decompression of unit instructions. Operand factorization across instruction words allows for decompression of instructions in parallel at a cost to the compression ratio.

1. Introduction

The main goal of any compression algorithm is to reduce redundancy and increase the information content in a given block of information. Code compression is a special case of data compression where the information to be compressed is a series of instructions making up a program. This brings with it some aspects specific to code compression, and as a result, some data compression techniques cannot be used. In particular, the majority of data compression techniques view the information to be compressed as a block of data (such as a file) that needs to be compressed in size. When compressing a series of instructions, however, certain information needs to be retrieved at will. For example, branch targets and function entry points must be able to be decompressed on demand. Furthermore, where decompression of the instructions is done in hardware, restrictions exist on the level of design complication of the decompression unit. If the decompression unit is too large, then the reduction in the size of the program memory is outweighed by the large new decompression unit. For hardware reasons also, byte-aligned and nibble-aligned restrictions are considered when designing code compression schemes and relevant hardware decompressors.

Information within program instructions is also organized in a very specific way, adhering to a rigid set of rules governed by the Instruction Set Architecture. This information can be fairly repetitive and sometimes redundant. Most code compression algorithms attempt to provide both the on-demand access to the necessary memory locations, as well as exploiting the repetition and redundancy present in object code.

RISC processors have been the main focus for code compression techniques but VLIW (Very Long Instruction Word) processors are now being considered in this area as a result of their increased appeal to not only larger applications, but also the embedded field.

VLIW processors are a classification of multiple-issue processors that execute multiple instructions simultaneously by issuing them to separate execution units. The VLIW compiler is responsible for the scheduling of instructions without dependencies, etc. The instructions in VLIW processors are sometimes referred to as operations and are grouped, in the code, into groups of *x* instructions where *x* is the maximum number of instructions that can be executed at once.

The TI TIMS320C6x is an 8-issue processor which has the ability to determine which of those 8 instructions will be executed in parallel with the others. This is known as Various Length Execution Set.

This paper compares compression algorithms based on entire instructions, factorization of instructions into op-code/operand fields and operand factorization applied to instruction words and discusses the tradeoffs with regard to sequential decompression time penalties vs compression ratio. Section 2 outlines the relevant related work and Section 3 explains the three compression algorithms used. Section 4 reports experimental results of applying those compression schemes on the benchmarks and Section 5 finishes with conclusions.

2. Related Work

The idea of using code compression as a tool for chip size reduction in microprocessors has mostly incited interest in the area of single instruction issue (usually RISC) processors. These compression schemes can be categorized as dictionary methods such as CodePack™ in [7] and SADC in [12], or as statistical such as Arithmetic Coding [8, 17] and Markov models [11].

2.1 Code Compression for RISC

Code compression for RISC processors first emerged in a paper by Wolfe and Channin [16]. This paper proposed a new RISC system architecture based on existing architectures called a CCRP (compressed Code RISC Processor). Due to RISC programs tending to be larger, a CCRP was suggested to compress the code and use a 'code-expanding instruction cache', such that the decompression could be transparent to the processor. Various Huffman-based encoding schemes were used. By using a compression technique that did not give consideration to branch targets and function beginnings, extra hardware was required to fetch addresses.

Further developments in RISC code compression developed code compression methods that looked at compiler techniques [5, 6], expression trees and operand factorization [3,

4], enhanced dictionary schemes and statistical schemes based on Markov models and arithmetic coding.

The CodePack™ encoding algorithm [7] encompasses a similar idea, as the most common half-instructions are replaced by the indexes to the smallest dictionary, the next set of half-instructions (in order of frequency) are replaced by an index into the second-smallest dictionary, etc. This introduces some overhead to determine which dictionary is used to decompress the half-instruction, but ensures that very few bits are required for the most common half-instructions. CodePack™ is said to achieve compression ratios of 35-40%, not including the dictionaries themselves.

Araujo et al [4], presented a compression algorithm called operand factorization, for program compression in RISC systems. The authors used to their advantage, the high repetition of opcode tree patterns (blocks of code containing both op-code and operand information), and separately, operand stream patterns. They find that a small number of distinct opcode tree patterns exist, due to the reduced instruction set in a RISC processor and also due to the nature of program instructions (where structures - such as for and while loops - are repeated often). After investigation of seven SPECInt95 programs, they found that less than 10% of the total number of expression trees are distinct. That is, the other > 90% of expression trees contain repeated occurrences. Likewise, operand streams also display a high factor of repetition, with an average of 23% of all operand sequences being distinct.

The authors investigated further to conclude that, as an average taken over the programs studied, 20% of distinct opcode tree patterns accounted for almost all opcode trees in a program, and that 20% of distinct operand streams contribute to 80% of the streams in a program. Once tree and operand patterns were separated, the authors compressed them with four different encoding schemes including the Huffman, Fixed-Length, Bounded Huffman and the MPEG2 VLC encoding schemes. As expected, the results showed the Huffman encoding scheme had the best compression ratio, with the compressed program being 35% of the original program size. Average compression ratios of 43% and 48% are reported for Huffman and MPEG2 VLC Encoding schemes respectively, taking into account the size of the decompression engine.

2.2 Code Compression for VLIW

The code compression techniques applied to date on multiple-issue processors (particularly the more original rigid VLIW processors, but also recently targeting variable execution set architectures) are limited. The compression schemes applied to VLIW processors to date is only a subset of the techniques available for both data compression and single-issue code compression.

Nam et al. [13] achieved average compression ratios of 63%-71% on SPEC95 benchmarks for varying VLIW architectures using a dictionary compression method. Two methods of investigating common instruction words are compared (identical – whole instructions words; and isomorphic – split into opcode/operand fields) in varying VLIW architectures. The isomorphic instruction words method is similar to the operand factorization presented in [4], except that expression trees are replaced with execution packets. Their results show that using the isomorphic instruction words method out-

performed the identical instruction words method by a compression ratio difference of at least 17%.

Ishiura and Yamaguchi [9] investigated code compression for VLIW processors, this time based on a statistical method called Automatic Field Partitioning. Their paper reduces the problem of compressing code to the problem of finding the field partitioning that yields the smallest compression ratio. Each field partition is then encoded with a dictionary scheme. Ishiura and Yamaguchi [9] achieve compression ratios of 46-60%.

Prakash et al [14] present a dictionary based encoding scheme that divides instructions into 2 16-bit halves. Compression ratios of 80% are recorded. Xie et al. [17] use a reduced-precision arithmetic coding technique combined with a Markov model (statistical method) and applies it to similar systems with different sized sub-blocks. Xie et al. also present a Tunstall-based memory-less variable-to-fixed encoding scheme as well as an improved Markov variable-to-fixed algorithm with varying model depths and widths in [18].

3. Compression Algorithms

Three compression algorithms were implemented for the purpose of comparison and investigation into the compression ratios obtain by applying each.

3.1 Dictionary Compression on Instructions

The first encoding scheme used in this paper is a dictionary compression method that analyses the instructions in a program, builds a dictionary with every unique instruction and compresses the original program by replacing every instruction with its corresponding reference into the dictionary. This is slightly different to our previous work on dictionary instruction compression in [15] where only those instructions appearing more than once were compressed. Figure 1 shows an example of the dictionary compression and Figure 2 shows a block diagram of the necessary hardware to decompress the code on the fly.

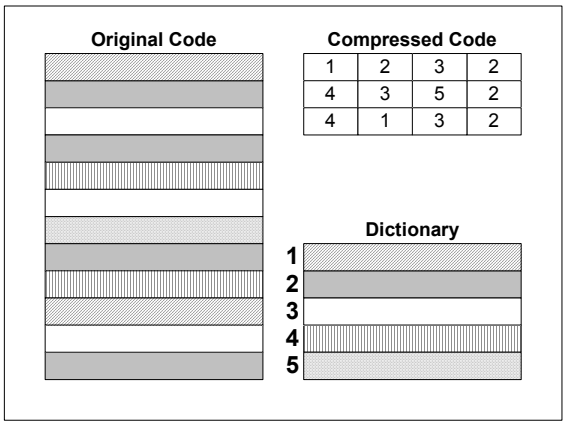

Figure 1 – Instruction Dictionary Compression

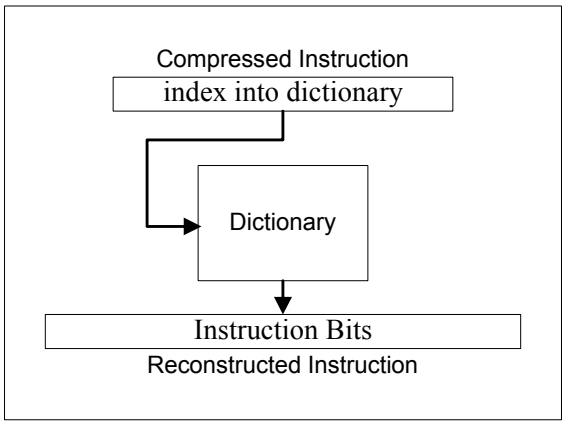

Figure 2 – Decompression Unit

Unfortunately, instructions appearing only once in the original code will contribute to code expansion as a result of them being stored in the dictionary and then having a codeword in place of them in the code. Although this comes at a cost to the compression ratio, the hardware needed to decode the original instructions is simplified. As a result,

no escape sequence is needed to determine whether an instruction is compressed or uncompressed in memory (they are all compressed!) and the latency associated with retrieving the original code is the same for all instructions.

This unfortunately gives rise to the possibility (especially in smaller programs) of having an increase in overall code size after compression, as a result of having many instructions that are scheduled to be executed only once. This usually isn't a problem for large programs, with many repetitive blocks, as is the case for most VLIW programs.

3.2 Compression using Instruction Factorization

The second compression algorithm exploits the facts that 1) the entire op-code space is usually not used and 2) operand patterns are often common between different instructions. A simplistic overview of the suggested algorithm is that each instruction is separated into two fields – the instruction-bits field: those bits that determine the instruction (op-code, escape sequences, conditional bits, hardware-specific bits, etc) and the operand-bits field: those bits that determine the operand sequence. These two fields are then compressed independently, with separate dictionaries, meaning that each full instruction in the original code is replaced by a pair of codewords that index into the two separate dictionaries. Figure 3 represents this idea.

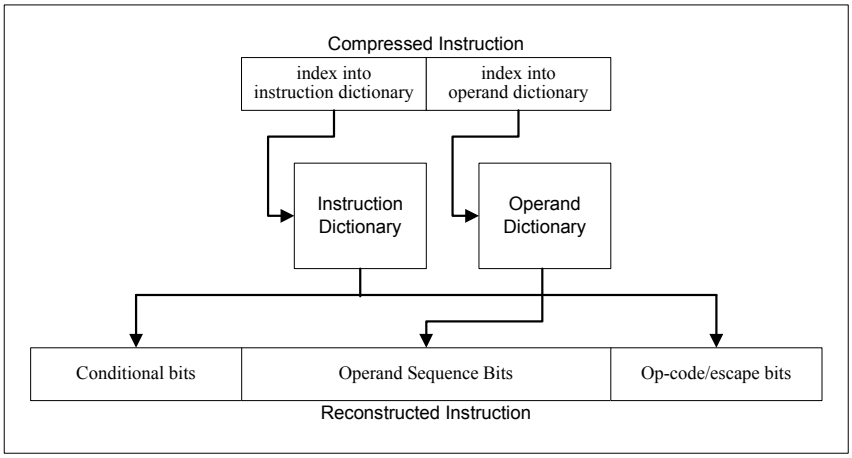

Figure 3 – Simplified Decompression Unit

The main problem encountered with the simplistic view of this compression algorithm is that in VLIW processors, instructions are executed on one of many execution units. Hence, the instruction formats are not fixed for all instructions and the above-mentioned field sizes differ from one instruction type to the next.

The proposed modification to the method of compressing these instructions is to first classify instructions into one of several instruction classes. Each of these classes contain instructions with similar op-code/operand field distributions. Then, two dictionaries are generated for each instruction class and the instructions are compressed as per the simplistic method above. Extra information (classification selection bits) is required to determine which class the instruction fell into, so that decompression is possible.

A further consideration is that some bits may not fall directly into the category of either instruction-determining bits or operand-sequence-determining bits. One example of this is the 'p' bit in the TMS320C6x family [2]. This lone bit at the end of each instruction

determines whether an instruction will be executed in parallel with the following instruction or not. If there were multiple bits that didn't fit into either category, this information could be treated as a third field, applying the same algorithm as per the instruction-bits- and operand-bits- fields. However, the TMS320C6x Instruction Set Architecture only has one such bit and as a dictionary would be pointless, that bit is carried through to the compressed instruction.

This gives us the format for the compressed instructions as given in Figure 4. Figure 4 also shows the block diagram of the decompression unit required.

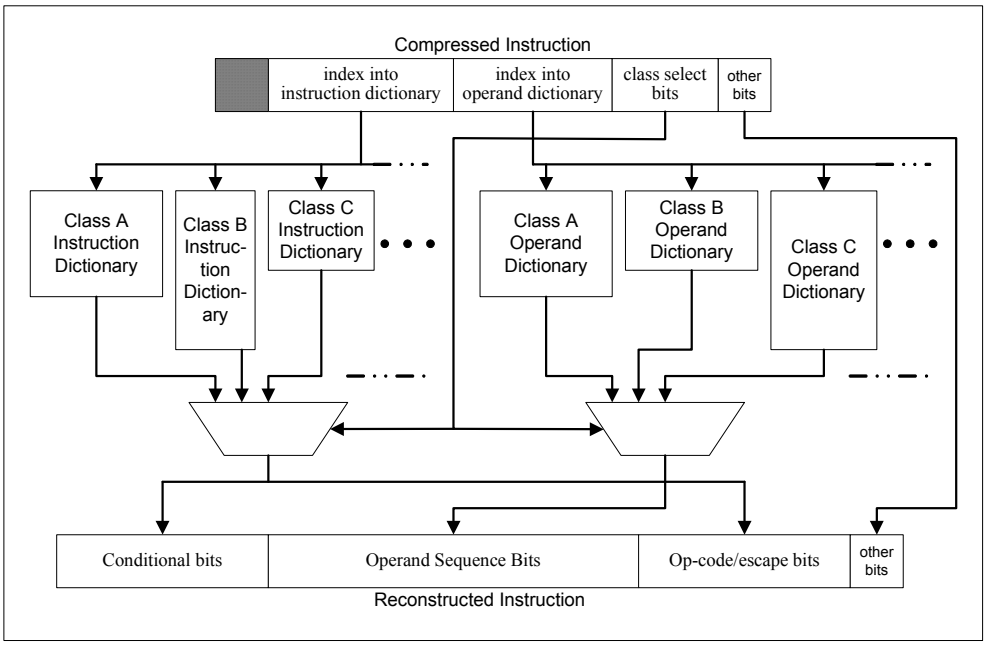

Figure 4 – Decompression Unit for Instruction Factorization

The codewords used in this compression scheme were byte-aligned. This meant that in some cases, entire bytes were not used and some padding was required (the shaded section in Figure 4). This resulted in some wasted bits throughout the programs, however this was viewed as a better trade-off than implementing the hardware required to decode variable length instructions wrapping from one byte to the next.

3.3 Compression using Operand Factorization on Instruction Words.

One of the issues with hardware decompression units is the time taken to decompress instructions on the fly. Where a fixed to variable compression scheme is applied, it is often necessary to sequentially decompress one instruction at a time, to find where the next instruction starts. When there are multiple instructions scheduled to be executed at the same time, this may take a considerable toll.

In the previous compression scheme, the bits for the instruction and operand fields are variable length (dependent on the instruction classification). As a result, instructions must be decompressed sequentially. This means that all instructions in the execution packet would need to be decompressed one by one before instructions could be executed.

In an attempt to parallelize this compression scheme, instructions were grouped into instruction words and the corresponding instruction/operand bits from all 8 instructions

were grouped together to form dictionary entries. This meant we were once again faced with multiple dictionaries to choose a dictionary word from. The class selector bits for each of the 8 instructions as well as the 'p' bits for each, were added to the codeword. This way, all 8 selector bits would be known initially.

This notion of separating sequences of instructions isn't a new one – operand factorization was used in [4] concentrating on expression trees and checking for isomorphic instructions involved separating instruction-words into the instructions' opcode and operand portions.

3.4 Benchmarks and Implementation

Mediabench [1] were chosen as an appropriate set of benchmark programs to investigate. These programs were compiled for both fixed point and floating point targets and an average was taken. The benchmarks used included:

- adpcm (rawc- and rawd-audio)
- g721 (encode and decode)
- epic (and unepic)
- mpeg2 (mpeg2enc and mpeg2dec)
- jpeg (cjpeg and djpeg)
- pgp

After each benchmark program was compiled with the TI TMS320C6x compiler (with varying optimization parameters), the smallest code size build was used in all subsequent compression calculations. This was in keeping with the findings in our previous work [15].

3.5 Branch Targets and Function Entry Points

As mentioned earlier, one of the important facets of code compression is the importance of access to branch targets and function entry points. In compression schemes where the compressed instruction size is fixed (or dependent on the instruction type which is known), and all instructions are compressed, it is possible to re-calculate the branch target references and calls to functions to correspond to the new location of a given instruction. This is similar to the technique applied in [10] where branch and calling instructions are 'patched' with the updated address of the memory location.

4. Results

The compression algorithms were applied to the benchmarks and the results obtained for the compressed sizes for each algorithm are shown in Figure 5. Compression Ratios are shown in Figure 6. As expected, the dictionary instruction based and the instruction factorization yielded good compression ratios (81.5% and 68.3% respectively).

However, when applied to instruction-words consisting of 8 operations, the operand factorization method produced some varying results. In some cases, operand factorization on 8-instruction words resulted in one of the better compression ratios, however in other cases it not only proved to be the worst compression method, but in the case of ***mpeg2*** and ***pgp*** benchmarks, it resulted in a code expansion. This was as a result of compressing all instruction words in the program, including those that only appear

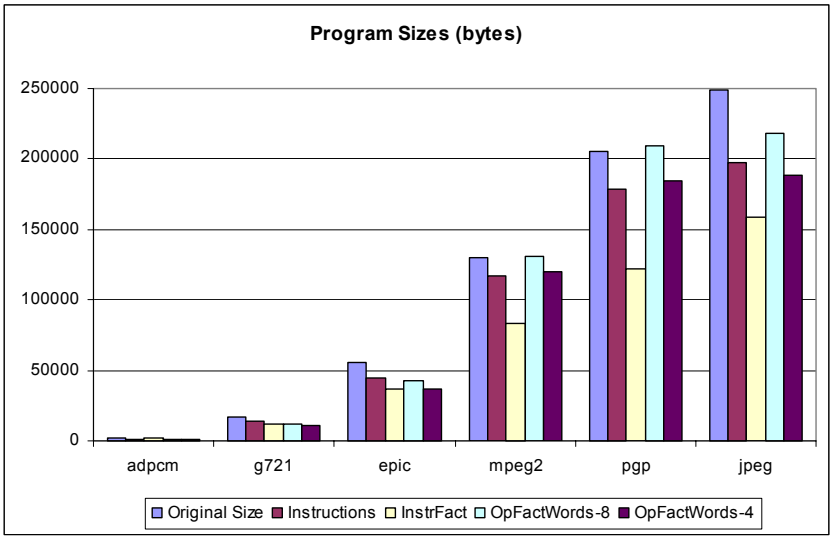

Figure 5 – Original and Compression Sizes of Benchmarks

once. If the distribution of such instruction words is too high, then we can get code expansion.

As the results for 8-instruction operand factorization were not as efficient as hoped, the same compression algorithm was applied to half-instruction-words, or sequences of 4 instructions, which gave much more favorable results – this time with no code expansion. This means that the two instruction-word halves consisting of 4 instructions each could be decompressed separately. This shows the trade-off between aggressive compression of code with sequential decompressing; against the not so compressive scheme allowing parallelization of decompression of instructions.

The best performing compression scheme, in particular in the larger benchmarks was the instruction-based factorization with an average compression ratio of 68.3%.

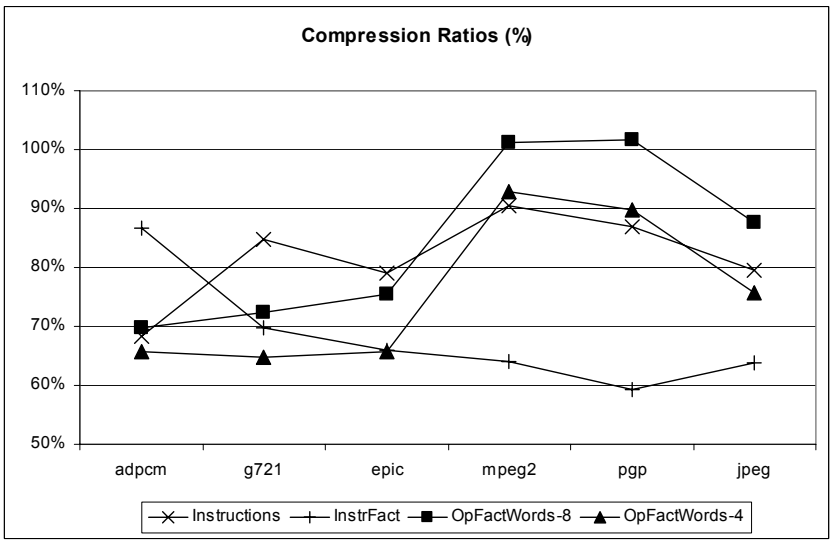

Figure 6 – Compression Ratios of Benchmarks

5. Conclusions and Further Work

This paper has presented the results of applying three different dictionary compression methods using whole instructions, instruction factorization and operand factorization on instruction words. Compression ratios of 81.5%, 68.3% and 84.7% were reported respectively. Instruction factorization was found to be the most efficient compression scheme, though decompression is done sequentially. Although this technique may be very advantageous for single-issue processors, it is detrimental to the time cost of the decompression of instruction words for VLIW processor, particularly when many instructions are scheduled for simultaneous execution. Operand factorization across instruction-words allows decompression to be parallelized for instructions in the same instruction word, however this is at a cost to compression ratio.

Further work can be done on the application of arithmetic compression algorithms to the separated instruction/operand streams, as well as applying operand factorization across the sequences of varying numbers of simultaneously-scheduled instructions with in instruction-words. Further investigation can also be done into the efficiency of these schemes on other platforms, particularly other VLIW processors with different sized instruction words.

References

[1] *Mediabench Benchmarks*, accessed 2003, http://www.cs.ucla.edu/~leec/mediabench/

[2] *TMS320C6000 CPU and Instruction Set Reference Guide*: Texas-Instruments, 2000.

[3] G. Araujo, P. Centoducatte, R. Azevedo, and R. Pannain, "Expression-tree-based algorithms for code compression on embedded RISC architectures," in *IEEE Transactions on Very Large Scale Integration VLSI Systems. Oct. 2000; 8(5)*: IEEE, 2000, pp. 530-3.

[4] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain, "Code compression based on operand factorization," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture. 1998*: IEEE Comput. Soc, Los Alamitos, CA, USA, 1998, pp. 194-201.

[5] K. D. Cooper and N. McIntosh, "Enhanced code compression for embedded RISC processors," in *SIGPLAN Notices. May 1999; 34(5)*: ACM, 1999, pp. 139-49.

[6] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting, "Code compression," in *SIGPLAN Notices. May 1997; 32(5)*: ACM, 1997, pp. 358-65.

[7] M. B. Game, A, "CodePack: Code Compression for PowerPC processors (version 1.0)," PowerPC Embedded Processor Solutions, IBM, North Carolina 2000.

[8] P. G. Howard and J. S. Vitter, "Practical implementations of arithmetic coding," in *Image and text compression. 1992*, J. A. Storer, Ed.: Kluwer Academic Publishers, Dordrecht, Netherlands, 1992, pp. 85-112.

[9] N. Ishiura and M. Yamaguchi, "Instruction Code Compression for Application Specific VLIW Processors BAsed on utomatic Field Partitioning," 1997.

[10] C. Lefurgy, P. Bird, I. C. Chen, and T. Mudge, "Improving code density using compression techniques," in *Proceedings. Thirtieth Annual IEEE/ACM*

International Symposium on Microarchitecture Cat. No.97TB100184. 1997: IEEE Comput. Soc, Los Alamitos, CA, USA, 1997, pp. 194-203.

[11] H. Lekatsas and W. Wolf, "SAMC: a code compression algorithm for embedded processors," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 1689-701, 1999.

[12] H. A. Lekatsas, "Code compression for embedded systems," Princeton University, 2000, pp. 171.

[13] S. J. Nam, In Cheol Park, and Chong Min Kyung, "Improving dictionary-based code compression in VLIW architectures," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E82-A, pp. 2318-24, 1999.

[14] J. S. Prakash, C.; Shankar, P.; Srikant, Y.N., "A Simple and Fast Scheme for Code Compression for VLIW processors," presented at Data Compression Conference, 2003.

[15] M. Ros and P. Sutton, "Compiler optimization and ordering effects on VLIW code compression," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. San Jose, CA, 2003, pp. 95-103.

[16] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in *SIGMICRO Newsletter. Dec. 1992; 23(1 2)*, 1992, pp. 81-91.

[17] Y. Xie, H. Lekatsas, and W. Wolf, "Code compression for VLIW processors," in *Proceedings DCC 2001. Data Compression Conference. 2001*, J. A. Storer and M. Cohn, Eds.: IEEE Comput. Soc, Los Alamitos, CA, USA, 2001, pp. 525.

[18] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression using variable-to-fixed coding based on arithmetic coding," in *Proceedings DCC 2003. Data Compression Conference. 2003*, J. A. Storer and M. Cohn, Eds.: IEEE Comput. Soc, Los Alamitos, CA, USA, 2003, pp. 382-91.