LZW-Based Code Compression for VLIW Embedded Systems

Chang Hong Lin
Princeton University
Princeton, NJ 08544, USA
chlin@ee.princeton.edu

Yuan Xie
Pennsylvania State University
University Park, PA 16802, USA
yuanxie@cse.psu.edu

Wayne Wolf
Princeton University
Princeton, NJ 08544, USA
wolf@ee.princeton.edu

Abstract

We propose a new variable-sized-block method for VLIW code compression. Code compression traditionally works on fixed-sized blocks and its efficiency is limited by the small block size. Branch blocks – instructions between two consecutive possible branch targets – provide larger blocks for code compression. We propose LZW-based algorithms to compress branch blocks. Our approach is fully adaptive and generates coding table on-the-fly during compression and decompression. When encountering a branch target, the coding table is cleared to ensure correctness. Decompression requires only a simple lookup and update when necessary. Our method provides 8 bytes peak decompression bandwidth and 1.82 bytes in average. Compared to Huffman's 1 byte and V2F's 13-bit peak performance, our methods have higher decoding bandwidth and comparable compression ratio. Parallel decompression could also be applied to our methods, which is more suitable for VLIW architecture.

1. Introduction

Embedded systems are cost and space sensitive, and memory is a large component of system cost. Code compression is used to reduce code size in embedded systems. It refers to compress the program off-line and decompress it on-the-fly during execution. The idea was first proposed by Wolfe and Chanin in the early 90's [1], and many researches have been done to reduce the code size for RISC machines [2, 3, 4, 5]. As instruction level parallelism (ILP) becomes the trend, a high-bandwidth instruction fetch mechanism is required to supply multiple instructions per cycle. Under these circumstances, reducing the code size and providing fast decompression speed are both critical challenges we face when applying code compression on VLIW machines.

This paper introduces branch-block based code compression. To ensure random accesses, previous work uses small, equally-sized blocks as compression units; each block could be decompressed independently with little or without information from others. When execution flow changes, decompression could restart at new position without or with little penalty. Not all instructions could be the destination of jump or branch, and the possible targets are determined once the program is compiled. We define **branch blocks** as the instructions between two consecutive possible branch targets, and use them as basic compression blocks. Our benchmarks contain only 80.1 branch blocks in average (454 bytes in size). Compiler methods can be used to increase the distance between branch targets. Since the size is much larger than the blocks used in previous work, we have more freedom in choosing the compression algorithms.

Our method uses LZW-style compression to create adaptive self-generating tables to avoid storing the decoding table, and would work on all embedded architectures. Moreover, this method has the advantages of fast and parallel decompression, which is suitable for VLIW architecture.

This paper is organized as follows. Section 2 reviews previous related work. Section 3 describes the general idea of our approach. We introduce the LZW-based code compression in section 4, and the selective code compression in section 5. Experimental results on benchmarks for Texas Instruments' TMS320C6x VLIW processors are presented in Section 6.

2. Previous Work

Wolfe and Chanin proposed the first code compression scheme [1], which used Huffman coding to compress MIPS programs. They use a Line Access Table (LAT) to map compressed block addresses, and this method is inherited by most of later studies. Based on the same concept, IBM built a decompression core, called CodePack, for PowerPC 400 series [4]. As shown in Figure 1, compressed code is stored in the external memory, and CodePack is placed between memory and cache. Liao [2] and Lefurgy [3] replaced frequently used instruction groups into dictionary entries, which make compressed code easy to be decoded. Lekatsas and Wolf [5] proposed SAMC, a statistical scheme based on

1530-1591/04 $20.00 (c) 2004 IEEE

arithmetic coding and Markov model. All of these methods targeted RISC architecture.

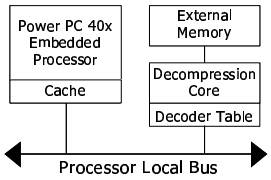

Figure 1. IBM CodePack for PowerPC.

Ishiura et al. split VLIW instructions into fields such that each field could be compressed optimally by using dictionary lookup scheme [6]. Nam et al. proposed a dictionary-based method by using isomorphism among VLIW instruction words [7]. These two schemes targeted traditional VLIW architectures, with rigid instruction word formats and lots of redundancy.

Larin and Conte were the first to apply code compression schemes on modern VLIW with flexible instruction formats [8]. They applied Huffman coding on an architecture similar to Intel/HP IA-64. Based on Tunstall coding, Xie and Wolf proposed variable-to-fixed (V2F) compression, which used fixed-length codeword to represent variable-length data [10]. Prakash et al. constructed a table of frequently appeared code strings and use the index along with the difference to compress the programs [12]. Xie also proposed the concept of profile-driven code compression [11], which used program profiles as one of the compression constrains.

3. Our Approach

To explain our VLIW code compression method, we will use the TI's TMS320C6x VLIW DSP [13], though our method is applicable to other VLIW processors as well. TMS processor gets a fetch packet (32 bytes) from cache, and separate the eight instructions into several execution packets. The instructions in the same execution packet are parallel executable.

We use branch blocks as our compression unit. Programs in the memory are compressed, and would be decompressed on-the-fly when the branch blocks are needed. The coding table used is self-generated during run-timet. As illustrated in Figure 2, The decompression engine could be put in two possible positions, pre-cache or post-cache. In the pre-cache structure, the timing overhead for decompression could be hidden behind cache miss penalty; while post-cache has more area and power saving. Although LZW-based methods could work on either case, post-cache structure would get more benefit due to our larger decompression bandwidth. Compressed blocks would not be at the same position as their original ones, so we borrow Wolfe and Chanin's idea of using a LAT to map the addresses [1] into original instruction addresses. Instead of storing the addresses of all cache lines, only those of branch targets are needed, which gives us a much smaller LAT.

Figure 3 shows the flow chart of our method. In both compression and decompression, the coding table is reset if the incoming address is a branch target; otherwise, we just keep on and update the table when necessary. Execution flow might change and the target address for branch or jump is computed during runtime; however, locations of possible targets are determined once the code compiled. We assume that we do note need random access to all instructions, but only ensure possible branch targets are accessible. Compression methods like the Lempel-Ziv (LZ) family give a good compression ratio with the requirement of long texts. **Compression ratio** (CR) is defined as compressed code size over its original size. Blocks used by traditional code compression schemes are too small for the LZ family; large, variable-sized branch blocks give us more freedom to choose the compression algorithms.

When programs are running, there is no problem to identify branch targets if execution flow changes. However, we need to find a way to distinguish branch targets from others when incrementing the PC causes execution to cross a block boundary. One way is to maintain a list of branch targets, but the entries have to be compared every time an instruction is executed. The other way is to use a codeword as branch target indicator. An indicator is sent to the output before branch targets during compression. When the decompression engine sees an indicator, it will know that the following instruction is a branch target.

4. LZW-Based Code Compression

Ziv-Lempel compression uses previously seen data to compress incoming one [14]. The coding table need not be stored with the compressed file, and can be recreated on-the-fly during decompression. The LZ family was not used for code compression before because it lacks random accessibility and has poor performance when deals with small

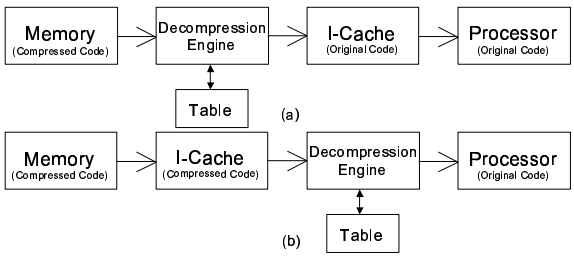

Figure 2. Two possible code decompression structures: (a) pre-cache; (b) post-cache.

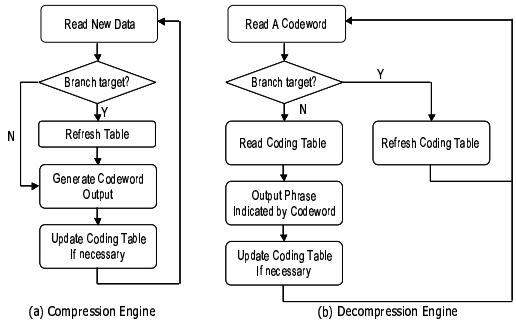

Figure 3. Flow chart of our method: (a) compression; (b) decompression.

Index	Phrase	Derivation
0	a	Initial
1	b	Initial
2	aa	0 + a
3	ab	0 + b
4	ba	1 + a
5	aba	3 + a
6	abaa	5 + a

Table 1. Coding table for LZW example.

blocks of data. The use of larger branch blocks makes it possible to take advantage of the well-compressed and fast-decompressing LZW method, and apply it on code compression.

4.1. LZW Data Compression

Lempel-Ziv-Welch (LZW) compression was modified from Ziv-Lempel 78 by Welch [15]. Initially, the LZW coding table has all the possible elements and phrases (series of elements) would be added during runtime. The compressor would search for the longest phrase in the table, output the index, and add the phrase with the next element as a new table entry. Suppose we have 2 elements a, b, and the input sequence is ”aababababaaa”. At the beginning, the longest phrase is ”a”, so ”0” would be generated, and ”aa” would be added in entry 2. As compression goes on, the output would be ”001352”, and the table would look like Table 1. When decompressor got the first ”0”, it would decode it as ”a”. However, no entry would be generated, since it has no idea what the next codeword is. ”aa” could be added only after the next codeword arrives. If the newly generated phrase were used, decompressor would not have that codeword in its table at that moment; however, this codeword could still be decoded as the previous phrase plus its own first element.

4.2. Code Compression and Decompression

To apply LZW to code compression we use the byte as our basic element. We have found that an initial table with 256 entries is adequate. We generate a new entry per iteration. During compression phase, the compressor would find the longest phrase in the table, send the codeword to the output, and add the phrase with the next byte as a new entry. Once the table is full, the compressor would keep on using the existing table to compress upcoming data. When a codeword is read from the memory, we first check if it is a branch target. If yes, the engine would shift out the padding from buffer, reset coding table, and restart decompression at a byte-aligned position. Otherwise, the decompression core would get a codeword, look it up, output the content, and add the old phrase with the first element of next phrase as a new entry.

We propose two ways to handle the appearance of branch targets. The first is to keep a separate list containing the address of branch targets. Both the compressor and decompressor have to compare current PC with the list. If a branch target is met, the process will restart with a initialized table. Since the list has to be compared each iteration, parallel comparators are needed, and would cost little degradation on decompression. The second method uses an exception index to indicate branch targets. We could simply use the all *1* codeword to indicate the existence of branch targets. The memory system in embedded system is often byte addressed, so each branch block has to start at a byte-aligned position. If the compressed branch blocks is not byte-aligned, several padding bits are necessary.

Codeword length and decoding bandwidth are two important parameters in our methods. LZW-based compression is a variable-to-fixed method. Fixed-length codeword is used to represent variable-length phrases. The codeword has to be at least 9 bits long, and would determine the table size exponentially. The larger the table, the more phrases could be represented, which yield better compression result. However, we also use a longer codeword to represent 8-bit basic elements. In the worst case, if none or only few repeated phrases occur, we might have to use one-element codewords most of the time. This often happens when the source file is not big enough. The other adjustable feature is the decompression bandwidth. From our simulation results, the compression ratio differs within 1% for widths from 8 to 20 bytes. Since the size of the decoding table is linearly dependent on the bandwidth, 8-byte wide decoding table would be the desired choice.

5. Selective Code Compression

In this section we present a modified algorithm, selective compression, with better compression ratio and even higher

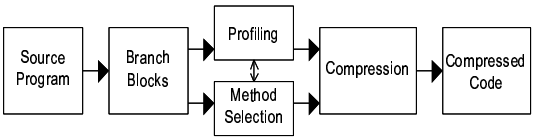

Figure 4. Block diagram of selective compression.

decompression bandwidth. The coding tables generated by different codeword length for the same branch block share exact the same entries in the front part. If the block is too small to fill up the smallest 9-bit table, there would be no benefit to use more bits to encode this block. On the other hand, longer codeword compress better in the larger blocks. We studied our benchmarks and found out that only 12.8% of the branch blocks could use up all the entries in 9-bit table, and only 1% could fill up the 12-bit table. This gives us the inspiration to apply different compression methods on different branch blocks.

We propose two selective compression schemes based on LZW compression. As shown in Figure 4, we collect the size, instruction and execution frequency, and other information for the branch blocks. The compression method is then determined based on the profile. Depends on the branch target identification policy used, we have different freedom in choosing the code compression methods.

We first try to minimize the size of the coding table used for each block. The shortest codeword is selected that all the phrases generated by the branch block could fit in the table. For example, when the block generates less than 256 phrases, only 9-bit LZW is needed. As longer codeword is used, the size of the table grows exponentially. Our experiments show that only few branch blocks really need larger table. And the larger the table, the fewer the branch blocks. 12-bit is chosen as the maximum codeword length for our methods.

If the branch target list is used, we use two bits in the table entry to indicate the method for the certain branch block (9-12 bit LZW). On the other hand, if the indicator is used, two bits are needed before each branch blocks to notify the method used. The average compression ratio using *minimum table-usage selective compression* (MTUSC) is 79.2%, which is about 3.5% better than 9-bit LZW.

The drawback of using MTUSC is there are only few repeated phrases in small blocks. Some of those compressed blocks use more bytes than original ones. *Minimum code-size selective compression* (MCSSC) is proposed to ease this problem. We would first compress each branch block using different codeword length LZW, and choose the smallest one (including the original code). By doing so, we ensure that each block has the minimum code size.

When the block is left uncompressed, there is no codeword to indicate the branch targets. Experimental results show that only blocks with size 32, 64 or 96 bytes could be left uncompressed. We could encode these situations along with the methods into 3 bits. These three bits could also be used for branch target list. The average compression ratio we could get by using MCSSC is 76.8%, which is about 6.3% better than 9-bit LZW.

We propose *Dynamic LZW* to ease the penalty caused by longer codewords. Since only one codeword is generated per iteration, in the first 256 iterations, only the first 512 codes might be used; and only 1024 codes are used in the following 512 iterations. We could use 9 bits to represent the codes in the first 256 iterations, and so on. We apply dynamic LZW on both MTUSC and MCSSC, and the compression ratio we get is 75.8% and 75.2% respectively, which is almost 8% better than 9-bit LZW.

Although we use four different codeword lengths in selective compression, only one 12-bit LZW decompression core with dispatching logic is enough for all of them. When branch indicator is met, the coding table would be initialized, and the dispatching logic would reconfigure to the upcoming method. Otherwise, the decompression engine just performs its normal operations. For codewords shorter than 12 bits, zeros would be padded in the front. The decompression core would use those padded codewords to address the coding table. It also works the same way when dynamic LZW is used. The only difference is the dispatching logic has to count the number of incoming codewords, and change the padding when necessary. On the other hand, the dispatching logic would bypass the instructions when the branch block only contains uncompressed instructions.

6. Simulation Results

In this section, we present experimental results on benchmarks for TI's TMS320C6x VLIW processor. The benchmarks are collected from TI and Mediabench (*http://www.cs.ucla.edu/leec/mediabench*), which are general embedded system applications with strong DSP component. The benchmarks are compiled using *Code Composer Studio IDE* from TI. The compression ratio, bandwidth, and overhead of our methods are described, and the comparison with some previous work is summarized in Table 2.

Figure 5(a) shows the compression ratio for all the benchmarks using 9 to 12-bit LZW. Longer codeword performs worse in most of the benchmarks. However, for the huge files, 12-bit LZW gives better compression ratio. The front part of the tables contain the same phrases for different codeword length. For benchmarks that couldn't fill up a small table, longer codeword won't help. The effect of large table could only be seen on huge benchmarks. In average, the compression ratio for 9 to 12-bit LZW is 83, 83,

Reference	Target	Method	Compression Ratio	Hardware Overhead	Decompression Bandwidth	Parallel Decompression
Wolfe and Chanin [1]	MIPS	Huffman	73%	Under 1 mm^2	1 byte per iteration, 1303M/sec asynchronous logic [16]	No
IBM [4]	PowerPC	CodePack	60%	Under 1 mm^2	1 byte per iteration	No
Lekatsas et al. [5]	MIPS	SAMC	57%	4k table + control logic	Not available	No
Xie et al [9, 10]	TMS 320 C6x	F2V	65%	0.01-0.02 mm^2	Average 4.9 bits per iteration	No
		V2F	70-82%	6-48k table + control logic	up to 13 bits per iteration	Only *iid* could
		LZW	83-87%	2-30k table, $<0.005mm^2$ control logic	Up to 8 bytes per iteration, Average 1.36-1.72 bytes	Yes
Our methods		MCSSC	75%	30k table, $< 0.005mm^2$ control logic, gated VDD could be applied	Up to 8 bytes per iteration, Average 1.82 bytes	Yes

Table 2. Overall comparison with previous work on code compression.

84 and 87% respectively. Figure 5(b) summarizes the compression ratio using MTUSC and MCSSC (both with and without dynamic LZW). It is clear that MTUSC is always the worst, and dynamic MCSSC is always the best among all four schemes.

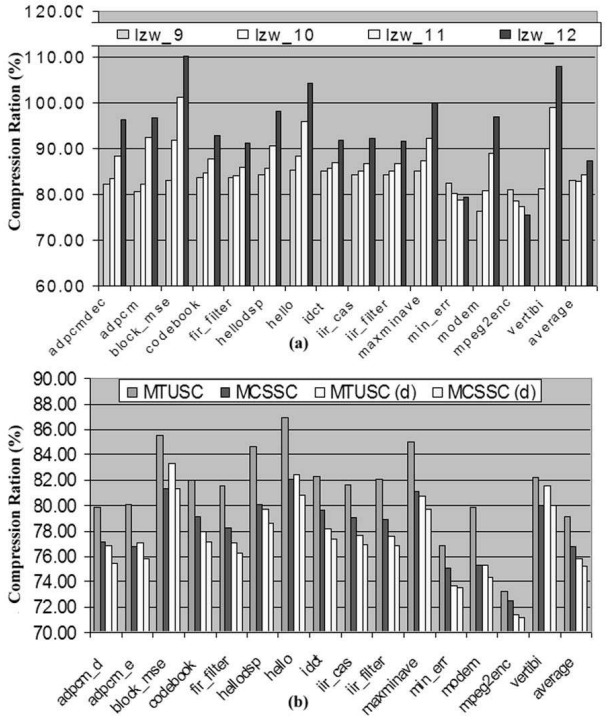

Figure 5. Compression ratio for (a) 9 to 12-bit LZW and (b) selective compression.

The maximum decompression bandwidth is determined by the width of coding table. As the width grows, the maximum phrase length also increases. However, as we could see from Figure 6(a), the average phrase length is 1.72 bytes. The throughput for MTUSC is the same as 12-bit LZW since they have the same codewords. On the other hand, the throughput for MCSSC is a little worse than MTUSC, since shorter codeword might be used. However,

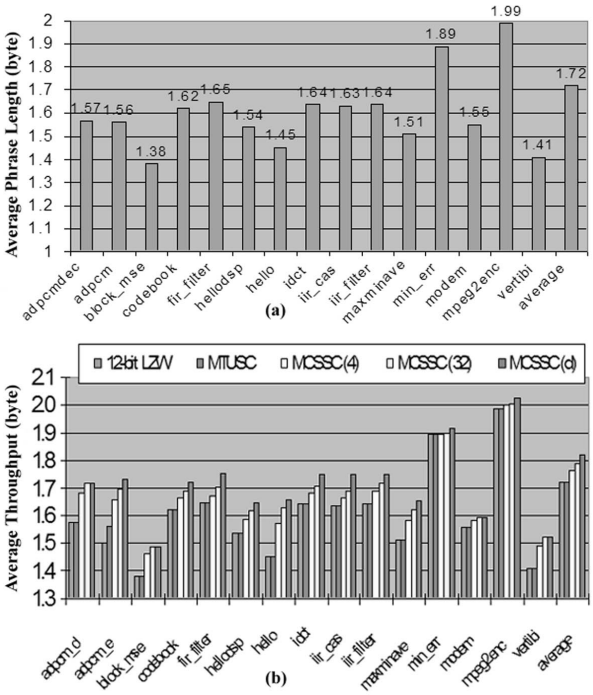

Figure 6. Average throughput for (a) 12-bit LZW and (b) selective compression.

some blocks in MCSSC are left uncompressed, and we can then bypass the uncompressed block by instruction (4 bytes) or by fetch packet (32 bytes). Then the average throughput would become 1.76 and 1.79 bytes. The use of dynamic LZW would also help in getting larger throughput in MCSSC, and the average throughput would become 1.82 bytes. Figure 6(b) shows the average throughput for all the benchmarks. We analyzed the decompression trace of selected branch blocks from our benchmarks. The throughput is only 1 or 2 bytes for small blocks due to only few repeated phrases. However, as the block size grows, the throughput also increases.

The LZW method is a variable-to-fixed code compres-

sion, and is suitable for parallel decompression by using a look-ahead scheme – parallel decompress the codewords when the newly generated codeword is not used. By using this scheme, the execution time to decompress the whole program becomes 0.51x,0.27x,0.14x for decompressing 2,4,8 codewords in parallel. The average throughput becomes 3.31, 6.37 and 12.29 bytes respectively. The throughput is almost doubled and the decompression time is halved by using parallel decompression. Although V2F also used a fixed-length codeword, the Markov model they used makes it impossible for parallel decompression [10]. V2F could only be parallel decompressed when independent and identical distribution is used, which has 83% compression ratio on TMS. Compare to V2F, our methods have much wider decompression bandwidth, and is more suitable for VLIW architecture.

For LZW-based code compression, the coding table used for both compression and decompression engine is determined by the codeword length and the decompression bandwidth. Suppose 9-bit LZW is used and the bandwidth is set as 8-byte wide, the table would be 4k bytes; and 8k,16k and 32k for 10,11,12-bit LZW. Since the first 256 entries are the basic elements, only some combinational logic is needed instead of storing them in the table. So, 2k bytes could be saved. Compare to previous schemes, Lekatsas' SAMC [5] uses 4k and Xie's V2F [10] uses 6-48k tables.

We use Verilog to implement both a full LZW-based decompression engine (except the coding table) and a decompression core witch could be used for selective and parallel decompression. The decompression engine we built uses a four-state FSM, and occupies 1129 4-input LUT in Xilinx' FPGA. This engine could even be pipelined into three pipeline-stages, and operates at 47.3 MHz. The core uses 314 LUT in the FPGA. We also synthesize the modules using TSMC .25μm model, and the area is 4417 and 1270 μm^2 respectively. If three-stage MCSSC is used, it would take 5508 cycles to decompress 9344-byte ADPCM decoder, and 90k cycles to decompress 182k mpeg2 encoder.

7. Conclusions and Future Work

We propose LZW-based and selective code compression schemes that use branch blocks as the compression unit. The compression ratio is about 83% and 75% respectively. Low power is achieved by smaller memory required to store compressed source code. Compare to previous work, our schemes have less decompression overhead and larger decoding bandwidth with compatible compression ratio. Parallel decompression could also be applied to our methods to achieve faster decompression, which is suitable for VLIW architecture.

For our future work, compiler could be modified to generate source programs more suitable for code compression. We could also find other code compression schemes that could take advantage of the branch blocks, and could also apply the schemes on both instruction and data memories.

8. Acknowledgment

This work was supported by Semiconductor Research Corporation (SRC).

References

[1] A. Wolfe and A. Chanin. Executing Compressed Programs on an Embedded RISC Architecture. *Proc. of the Intl. Symposium on Microarchitecture*, p.81-91, Dec. 1992.

[2] S. Liao, S. Devadas, K. Keutzer. Code Density Optimization for Embedded DSP Processors Using Data Compression Techniques. *Proc. on Advanced Research in VLSI*, p. 393-399, 1995.

[3] C. Lefurgy, P. Bird, I. Chen and T. Mudge. Improving Code Density Using Compression Techniques. *Proc. of the Annual Symposium on Microarchitecture*, 1997.

[4] IBM. *PowerPC Code Compression Utility User's Manual*, Version 3.0, 1998.

[5] H. Lekatsas and W. Wolf. SAMC: A Code Compression Algorithm for Embedded Processors. *IEEE Trans. on Computer Aided Design*, Vol.18, p.1689-1701, Dec. 1999.

[6] N. Ishiura and M. Yamaguchi. Instruction Code Compression for Application Specific VLIW Processors Based of Automatic Field Partitioning. *Proc. of the Workshop on Synthesis and System Integration of Mixed Technologies*, p.105-109, Dec. 1997

[7] S. Nam, I. Park and C. Kyung. Improving Dictionary-Based Code Compression in VLIW Architectures. *IEICE Trans. Fundamentals*, p.2318-2324, Nov. 1999.

[8] S. Larin and T. Conte. Compiler-Driven Cached Code Compression Schemes for Embedded ILP Proceesors. *Proc. of the Annual Int. Symposium on Microarchitecture*, p.82-91, Nov. 1999.

[9] Y. Xie, W. Wolf and H. Lekatsas. Compression Ratio and Decompression Over-head Tradeoffs in Code Compression for VLIW Architectures. *Proc. of Intl. Conf. on ASIC*, Oct. 2001.

[10] Y. Xie, W. Wolf and H. Lekatsas. Code Compression for VLIW using Variable-to-fixed coding. *Proc. of ISSS*, 2002.

[11] Y. Xie and W. Wolf. Profile-driven Code Compression. *DATE*, March 2003.

[12] J. Prakash, C. Sandeep, P. Shankar and Y.N. Srikant. A Simple and Fast Scheme for Code compression for VLIW Architectures. *IEEE Data Compression Conf.*, Jan. 2003.

[13] Texas Instruments. *TMS320C62xx CPU and Instruction Set: Reference Guide*, January 1997

[14] T. Bell, J. Cleary and I. Witten. *Text Compression*, Prentice Hall, 1990.

[15] M. Nelson. LZW Data Compression. *Dr. Dobb's Journal*, Oct. 1989. (*http://dogma.net/markn/articles/lzw/lzw.htm*)

[16] M. Benes, S. Nowick and A. Wolfe. A fast Asynchronous Huffman Decoder for Compressed-Code Embedded Processors. *Proc. of Annual Int. Symposium on Advanced Research in Asynchronous Circuits and Systems-ASYNC*, p.43-56, 1998.