FBT: Filled Buffer Technique to Reduce Code Size for VLIW Processors

Talal Bonny and Jörg Henkel
University of Karlsruhe, Chair for Embedded Systems, Karlsruhe, Germany
{bonny, henkel}@informatik.uni-karlsruhe.de

Abstract*— VLIW processors provide higher performance and better efficiency etc. than RISC processors in specific domains like multimedia applications etc. A disadvantage is the bloated code size of the compiled application code. Therefore, reducing the application code size is a design key issue for VLIW processors. In this paper we adapt a hardware-supported approach called "Deflate" [12] which has been used before in data compression. It can significantly reduce the code size compared to state-of-the-art approaches for VLIW processors as we will show within this work. In fact, we enhance the "Deflate" algorithm by using a new technique called Filled Buffer Technique which can be applied to any Lempel-Ziv family algorithms to improve compression ratio in average by more than 13% compared to the sole "Deflate" algorithm. Using our Filled Buffer Technique in conjunction with "V2F" [15] improves the compression ratio by 10%. We have conducted evaluations using a representative set of benchmarks (from Mediabench and Mibench) and have applied our scheme to two VLIW processors, namely TMS320C62x and TMS320C64x. We achieved allover compression ratios as low as 44% using the "Deflate" algorithm (61% and 56% in average for TMS320C62x and TMS320C64x, respectively).

I. Introduction and Related Work

An important issue in embedded system design is the size of the program memory of an embedded application since it may occupy up to 3-7 times of the silicon footprint of an embedded processor [6]. VLIW processors provide higher performance than RISC processors for a broad range of applications because of their ability to exploit fine-grain, instruction-level parallelism. However, the drawback of an VLIW architecture is the bloated code size of the compiled applications in comparison to RISC processors. Fig. 1 shows the relative average code size of several benchmarks (from Mibench) on different processor architectures. It is obvious that code compression is of paramount importance when it comes to VLIW architectures.

Code compression differs from data compression in the size of the *text* that is to be compressed/decompressed. In data compression, the data is compressed and decompressed as a whole (i.e. as one block) while in code compression small segments or blocks of code (called compression blocks) are compressed and decompressed individually/independently to ensure *Random Access* in decompression. For that, data compression typically results in higher compression ratio than code compression. Data compression algorithms from the Lempel-Ziv family use the "sliding window" method to match a series of bits in a so-called look-ahead buffer to bits already in the search buffer. At the beginning of compressing each compression block, the search buffer is empty. Hence, the first series of bits in the look-ahead buffer will find no match in the search buffer and remain without any compression. This will negatively impact the final compression ratio when it comes to small *text* size.

We therefore *explicitly* reduce the size of compressed

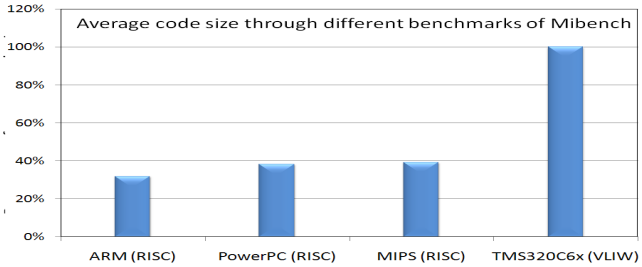

Fig. 1. Relative average code size through different benchmarks of Mibench on different processor architectures

instructions by using our novel technique (called *FBT: Filled Buffer Technique*) which can be applied in conjunction with any algorithm from the Lempel-Ziv family. As we will show later on in detail, our *Filled Buffer Technique* fills in the search buffer with a series of bit patterns at the beginning of compressing of any compression block. This technique improves the compression ratio (in average) by more than 13% compared to algorithms that do not use our technique (discussed later in the experimental results). To show the orthogonality of our *Filled Buffer Technique (FBT)*, we apply it to another algorithm from the Lempel-Ziv family which is called "LZMA". We also compare our compression results with the results of state-of-the-art previous work like"V2F" [15] (as this method achieves very high decoding throughput) and show the optained improvements.

In **Related Work**, various compression techniques have been studied for VLIW processors. In [15] and [16], the authors use variable to fixed code compression schemes based on range encoding for VLIW processors. The most important advantage of their scheme is the ability of parallel decoding. In [9] the authors divide instructions into 11 groups. Within a group, they partitioned the instruction into two 2-byte segments and created tables for each group leading to an average compression ratio of 70% with 3 bytes decompression each clock cycle. Ros and Sutton ([10] and [11]) developed a dictionary compression technique based on victor Hamming distances for the complete 32-bit instructions. The authors achieve compression ratios of 72.1% to 80.3% while targeting TMS320C6x VLIW processor. Lin et al. [8] proposed a variable-size code compression method based on LZW for VLIW processors. Their method generates adaptive coding tables on-the-fly during compression and decompression to avoid storing. They achieved an average compression ratio of 75.2% targeting the TMS320C6x VLIW processor.

The rest of the paper is organized as follows. In Section II we give an overview of the VLIW processors. Our compression

978-1-4244-2820-5/08/$25.00 ©2008 IEEE

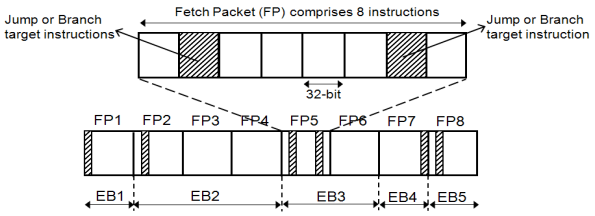

Fig. 2. Example for exploring the Extended Blocks (EB) from "Fetch Packets" (FP) for compression

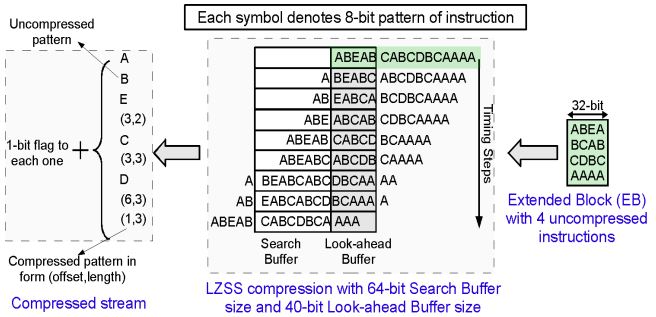

Fig. 3. Example for compressing an *Extended Block* using LZSS

scheme is presented in Section III. In Section VI, the hardware implementation for our scheme is presented whereas experimental results are presented in Section V. We conclude this paper with Section VI.

II. VLIW Processors

We apply our compression technique targeting two VLIW DSPs from Texas Instruments, namely the TMS320C62x [13] and TMS320C64x [14] (we refer to them in this work as C62x and C64x), though our technique can be applied to any other VLIW processor as well.

The instruction word in the C62x comprises eight 32-bit instructions and is called a "Fetch Packet". The instructions in the same "Fetch Packet" may be executed in one or more clock cycles (due to data dependencies). The "Execution Packet" contains only the instructions which may be executed in parallel (in one clock cycle). Each instruction has a bit which indicates whether the next instruction is part of the same "Execution Packet". The "Fetch Packet" is not restricted with the boundary of the *Basic Blocks*, i.e. the branch target instruction may emerge in the middle of the "Fetch Packet".

The C64x is a developed version of the C62x and is a part of the DaVinci multimedia processor by Texas instruments. The number of 32-bit registers is extended to 64 and the "Execution Packet" is developed to include instructions from another "Fetch Packet" (i.e. cross the "Fetch Packet"). These changes decreases the number of NOPs and consequently reduces the size of compiled program code in comparison to the C62x (See Fig. 7).

III. Our Code Compression Approach

In code compression, the compression algorithm is applied to small segments or code blocks (called "compression blocks") to ensure random access. The first step in any code compression approach is to determine the size of these compression blocks. Selecting *Basic Blocks* as compression blocks diminishes the possible compression due to their small *text* size. Extending the size of a compression block to include more instructions than in the *Basic Blocks* will improve the final compression ratio as explained later. In a VLIW processor, if a branch target instruction appears in the middle of the "Fetch Packet", only the instructions which follow the branch target will be executed. Hence, we may extend the compression blocks to include more instructions than the *Basic Blocks* and call them *Extended Blocks* which are defined as follows:

(1) The *Extended Block* may contain one or more complete "Fetch Packets". **(2)** The branch target instruction should only exist in the first "Fetch Packet" of the *Extended Block*.

Fig. 2 shows an example for exploring the *Extended Blocks*. In this figure there are 8 "Fetch Packets" (FP1 - FP8). The dashed area refers to a branch target instruction. The number of *Basic Blocks* (BB) is 6 (because of the 6 branch target instructions). The first *Extended Block* (EB1) contains only one "Fetch Packet" (FP1) because the next branch target instruction appears directly in the next "Fetch Packet" (FP2). The second *Extended Block* (EB2) contains 3 "Fetch Packets" (FP2 to FP4) and so on. The number of *Extended Blocks* in this example is reduced to 5.

A. Deflate Compression Algorithm

'Deflate" [12] is a data compression algorithm that has originally been used in the Zip and Gzip software to compress data files. It is based on an optimized version of LZ77 (which is called LZSS) combined with Huffman codes.

The basic idea behind the LZSS method (Algorithm 1) is to use part of the previously-seen input stream as the dictionary. The encoder then shifts the input in that window from right to left as strings of bits (i.e. patterns) are being encoded (line 2). Thus, the method is based on a *sliding window*. The window (see Fig. 3) is divided into two parts. The part on the left is the *search buffer*. This buffer includes patterns of consecutive bits that have recently been input and encoded (each symbol denotes 8-bit pattern of instruction). The part on the right is the *look-ahead buffer*, containing patterns yet to be encoded. The algorithm 1 adds the flag '0' to the "length" (line 6) and the flag '1' to "literal" to distinguish between the uncompressed and compressed ones. Finally the algorithm returns the compressed stream (line 10) which contains the uncompressed patterns ("literals") and the compressed ones (pair of("offsets","lengths")).

Fig. 3 shows an example for compressing an *Extended Block* which contains 4 instructions using the LZSS algorithm. These instructions compose the string of symbols "ABEABCABCD-BCAAAA" (each symbol denotes a pattern which correspond to 8 bit of the instruction). The encoder in this example outputs the compressed stream "ABE(3,2)C(3,3)D(6,3)(1,3)". The encoder adds a 1-bit flag at the beginning of each compressed- uncompressed pattern.

In the next step of "Deflate", we encode the compressed stream using three different models of *Huffman Coding*:

Static-Static Huffman Tables: This is the standard model in the"Deflate" algorithm in which two static code tables are prepared for encoding: one to encode the "literals" and "lengths" and the other to encode the "offsets". The encoder replaces the codes that are written on the compressed stream along with the new codes of the tables.

Dynamic-Static Huffman Tables: In this model, we create a Huffman table for the "literals" and "lengths" only. For the

Algorithm 1 : LZSS Compression Algorithm

/* pat: pattern which is consecutive bits of instruction (pat is 8-bit long) */
/* LB: Look-ahead Buffer */
/* SB: Search Buffer */

1: **Function** LZSS (pat) {
2: $pat >>$ LB {shift patterns in the Look-ahead Buffer}
3: **if** pat in SB match pat in LB **then** {pattern is compressed}
4: find $offset$ and $length$
5: $pat \Rightarrow (offset,length)$ {pat is compressed as ($offset$,$length$)}
6: $length$ + '0' {add '0' as flag for uncompressed}
7: **else** {pattern is not compressed}
8: pat + '1' $\Rightarrow literal$ {uncompressed pat and flag is called $literal$}
9: **end if**
10: **return** ($literal$,$offset$,$length$) {return compressed stream}
11: }

"offsets", we use the prepared static table.

Dynamic-Dynamic Huffman Tables: In this model, we create two Huffman tables, one for "literals" and "lengths" and the other for the "offsets". Depending on their repetition frequency, shorter code words are assigned to the most frequent patterns and vice versa.

B. Our Filled Buffer Technique (FBT)

Our main goal here is to adapt the "Deflate" algorithm such that it works efficiently in code compression. As the *search buffer* is empty at the beginning of each encoding of an *Extended Block*, the first patterns in the *look-ahead buffer* will naturally not find any match in the *search buffer* and therefore they remain without any compression. In data compression, these non-matching patterns occur *only one time* (in the beginning of compressing the whole data stream). However, in code compression that occurs for every *Extended Block*. For that, the number of non-matching patterns (i.e. uncompressed patterns) in code compression will be more than data compression and negatively impact the compression ratio as each uncompressed pattern will have one extra bit flag to be distinguished from the compressed one.

To overcome this deficiency, we use a novel and efficient technique (that we call *Filled Buffer Technique (FBT)*) before we apply the actual compression algorithm. Hence, the encoder fills in the empty *search buffer* (off-line) with selected patterns in the beginning of encoding each *Extended Block*. Therefore, the patterns in the *look-ahead buffer* may match those filled patterns in the *search buffer* and then they may be compressed with a pair of (offset, length).

To find those patterns, we use Algorithm 2 which consists of three steps. In the first step, we compress the whole application as one block (i.e. in a data compression manner) using the LZSS algorithm (lines 1-4) and compute the compression ratio for each pattern of all instructions (lines 5-7). In the second step, we compress each *Extended Block* of the application separately (i.e. in code compression manner) using the LZSS algorithm (lines 1-5) and compute the compression ratio of each pattern in the *Extended Block* (line 7). In the third step, we compute the difference in compression ratios of patterns (lines 1-3). Then, we sort the patterns by their difference in compression ratios in descending order (line 4). As the size of the search buffer is 1024 Byte, we just need to select the first 1024 sorted patterns to fill in the search buffer at the beginning of compressing each *Extended Block*.

To compress the *Extended Blocks* using "Deflate" in conjunction with our *Filled Buffer Technique*, we use Algorithm 3. In this algorithm, and before we compress each *Extended Block*,

Algorithm 2 FBT: Filled Buffer Technique (three steps)

/* i: 32-bit instruction */
/* EB: Extended Block */
/* CR_1: Compression ratio in case data compression*/
/* CR_2: Compression ratio in case code compression*/
/* diff: Difference in Compression ratio between CR_1 and CR_2 */
/* Compress whole application as one Block (data compression) */

1: **for all** instructions i in the application **do** {partitioning}
2: partition i into 4 pat {each pattern is 8-bit long}
3: **end for**
4: **Call** LZSS (pat) {call function LZSS in Algorithm 1}
5: **for all** patterns pat in application **do** {compute CR_1 for each pat}
6: CR_1 = compressed pattern size(in bits)/8
7: **end for**

/* Compress each EB in application separately (code compression)*/

1: **for all** Extended Blocks EB in the application **do**
2: **for all** instructions i in the EB **do** {partitioning}
3: partition i into 4 pat
4: **end for**
5: **Call** LZSS (pat) {call function LZSS in Algorithm 1}
6: **for all** patterns pat in EB **do** {compute CR_2 for each pat}
7: CR_2 = compressed pattern size(in bits)/8
8: **end for**
9: **end for**

/* Compute the difference in compression ratios for each pattern */

1: **for all** patterns pat in application **do** {compute diff}
2: diff= $CR_2 - CR_1$ {Compute the difference for each pattern}
3: **end for**
4: sort patterns pat by diff descendingly
5: select the first 1024 sorted patterns to fill in the SB

Algorithm 3 Deflate with Filled Buffer Technique

1: **for all** Extended Blocks EB in the application **do**
2: SB $\Leftarrow$ sorted pat {fill in Search Buffer with first 1024 sorted patterns}
3: **Call** LZSS (pat) {call function LZSS in Algorithm 1}
4: **end for**
5: Compress the compressed stream using Huffman Coding

we fill in the *search buffer* with the first 1024 sorted patterns (from Alg. 2). Then, we compress the patterns using LZSS (Alg. 1) in line 3 and Huffman coding (line 5).

Fig. 4 shows an example for the *Filled Buffer Technique (FBT)*. Assuming that we have three *Extended Blocks*, we want to find the patterns which are required to fill in the *search buffer* before compressing each *Extended Block*. According to Alg. 2, we first compress the three blocks together using the LZSS as data compression technique. As the *search buffer* is empty at the beginning of the compression, the patterns "ABCDEF" in EB1 will be left without compression but the patterns "ABCD" in EB2 and 'ABCDEF" in EB3 will find a match in the *search buffer* and will be compressed using the pair (offset, length). Then, we compress each *Extended Block* using LZSS, separately. As the *search buffer* is empty in the beginning of compressing each *Extended Block*, the patterns in each *Extended Block* will find no match and will be left without compression. Each uncompressed pattern will have one extra flag bit to indicate that it is not compressed. Considering that each pattern has 8 bit and that the "offset" is encoded in 10 bit, the compression ratio for any uncompressed pattern will be 112% (which is bad). However, for each pattern in EB2 and in EB3 is 59% and 39%, respectively. The differences in compression ratios between each pattern in EB1, EB2 and EB3 will be 0%, 53% and 73%, respectively. As the patterns in the EB3 achieve the highest difference in compression ratios, we select its patterns to fill in the *search buffer* at the beginning of compressing each *Extended Block*.

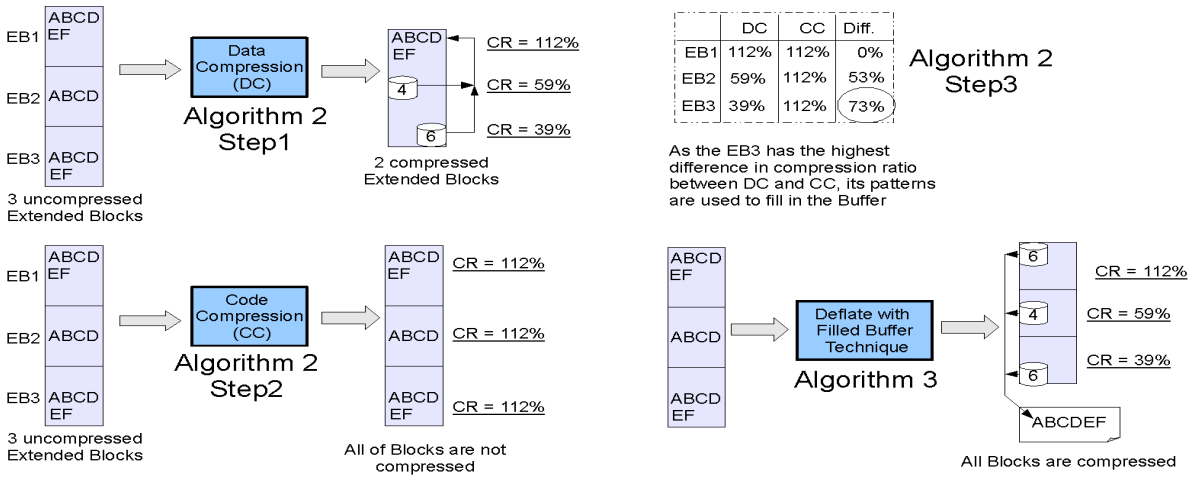

Fig. 4. Example to compress 3 *Extended Blocks* using the "Deflate" algorithm in conjunction to the *Filled Buffer Technique (FBT)*

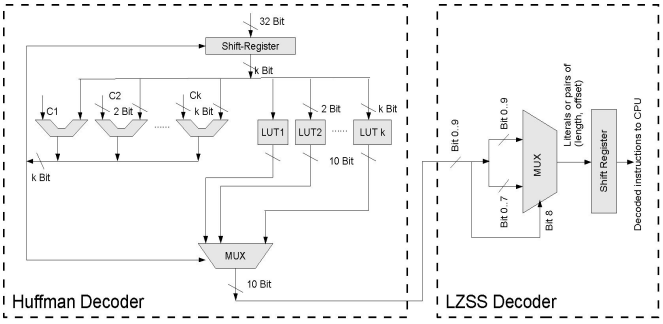

Fig. 5. Hardware Decoder

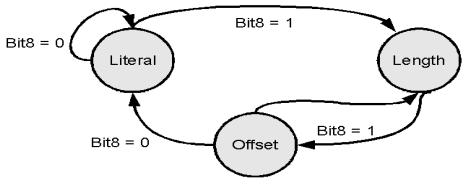

Fig. 6. State diagram of the LZSS decoder

Compressing the *Extended Blocks* after applying our FBT will improve the compression ratio for the *Extended Blocks* (EB2 and EB3) to be 59% an 39%, respectively.

IV. Decompression Architecture Design

To decode the compressed instructions using the "Deflate" algorithm, we use two sequential decoders (Fig. 5) which is a Huffman Decoder and an LZSS decoder

A. Huffman Hardware Decoder

The Huffman Coding Algorithm generates a large decoding Look-up Table which is used to decode the compressed instructions with a hardware decoder. An efficient way to store the Huffman Tables is to use Canonical Huffman Tables [7]. Each table stores the codes of the same length contiguously. To decode these tables we derived the hardware decoder from [2] and optimized it to improve its throughput. The hardware architecture is illustrated in Fig. 5. It consists of four components: shift register, comparators unit, Look-Up Table for each code length and multiplexer. We optimized the comparators unit and Look-Up Tables to be integrated in one pipeline stage. The optimized decoder decodes the Huffman codes in three phases (three pipeline stages). In the first phase, the shift register receives a 32-bit compressed Huffman code and shifts its contents by the amount of length of previous decompressed code (in bits). The shift register outputs k-bit equal to the longest Huffman code in the Look-up Tables. In the second phase, the k-bit output of the shift register is transferred to the comparator unit and to the Look-up Tables simultaneously. The number of comparators and Look-up Tables is equal to the number of different code lengths. The incoming k-bit is compared in each comparator to the maximum code of its length and the outputs of these comparators control the multiplexer output in the third phase. In parallel to the comparators unit, the incoming k-bit is also transferred as indices to the Look-up Tables (according to the length of each table). As Huffman codes are prefix free, the output of only one Look-up Table will be considered. In the third phase, the multiplexer chooses one output of the Look-up Tables according to the control signal which is received from the comparator unit. The Huffman decoder outputs a 10-bit code which may be a "literal" (9-bit), "length" (9-bit) or "offset" (10-bit). For that, the main task for the LZSS decoder is to decode the content of the 10-bit output of the Huffman decoder ("literal", "length" or "offset") and then to build the buffer to retrieve the original instructions.

B. LZSS Hardware Decoder

The LZSS decoder (Fig. 5) consists of a special multiplexer and shift register. The input of the multiplexer is connected to the output of the Huffman decoder. LZSS Decoder decodes the instructions in three phases. In the first phase, The multiplexer receives the 10-bit code from the hardware decoder. Its main task is to analyze the incoming code to decode its content, i.e. "literal", "length" or "offset". The "literal" and "length" are 8-bit long each (from bit 0 to bit 7). In each of them, the bit 8 is a flag bit and the bit 9 is not used. The "offset" is 10-bit long. Fig. 6 shows the state diagram of the LZSS decoder to decode the incoming 10-bit code from the Huffman decoder. When a new 10-bit code arrives, the Bit number 8 is checked. If it is '0', that means the code is a "literal" and the multiplexer

Benchmark	C62x			C64x		
	# total instructions	# Basic Blocks	# Extended Blocks	# total instructions	# Basic Blocks	# Extended Blocks
fft	11208	935	447	9296	679	284
basicmath	13184	1107	467	10752	828	299
unepic	27728	2293	1260	23600	1336	527
mpeg2enc	47168	3828	1952	39936	2064	606
rdjpgcom	68528	4613	2640	59912	1731	599
wrjpgcom	82632	5445	3154	72464	1915	654
djpeg	83920	5564	3223	73608	1968	663
jpegtran	90832	5982	3491	79792	2063	686
ansi2kr	91776	6074	3563	80560	2079	688
Average	57442	3982	2244	49991	1629	556

Fig. 7. Number of original instructions, Basic and Extended Blocks for benchmarks compiled for C62x and C64x processors

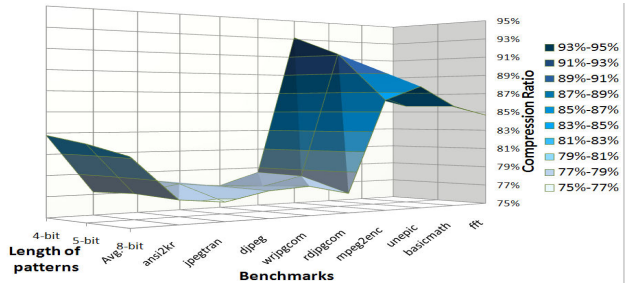

Fig. 8. Compression Ratios for different Benchmarks compressed using LZSS and for 4-bit, 5-bit and 8-bit pattern length

transfers the 8-bit directly to the shift register (in the second phase). If bit 8 is '1', then the code is "length" and the next 10-bit code will be "offset". In this case the multiplexer keeps the 8-bit "length" and waits for the next 10-bit "offset". When it receives the "offset", it transfer the pair (offset, length) to the shift register. In the third phase, the shift register builds the look-ahead buffer and generates the original instructions. We designed both of the decoders in VHDL and implemented them using Xilinx ISE9.2 on the FPGA platform *"Platinum"* from *Pro-Design*. An average access time of 3ns was achieved and around 800 slices were need for the decoder [1].

V. Experiments and Results

We conducted experiments for two VLIW processors (Texas Instruments), namely C62x and C64x. For both architectures, all benchmarks from *MediaBench* [4] and *MiBench* [5] served as a representative set of applications. We compiled and linked the applications using the Code-Composer-Studio (CCS) from Texas Instruments and used the simulator "c6xsim" [3] to obtain performance results. Experimental results are shown in figures 7 - 12 and are explained in the following sections. The bar labeled *"Average"* shows the average across all benchmarks in each chart.

A. Statistics of the Benchmarks

Fig. 7 shows the number of total code instructions, Basic and *Extended Blocks* for different benchmarks compiled for C62x and C64x processors. This figure shows that the average number of instructions for the processor C64x is 16% less compared to the C62x. This is because the C64x processor is a further developed version of the C62x. The Fig. 7 shows also that the average number of *Extended Blocks* is 43% and 65% less than the number of *Basic Blocks* for C62x and C64x processors, respectively. This shows the importance of applying the "Deflate" compression technique on *Extended Blocks* other than *Basic Blocks*.

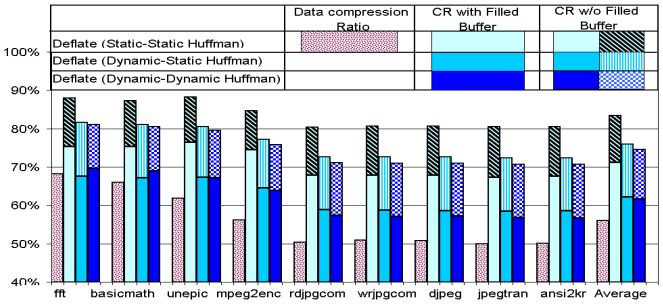

Fig. 9. Compression ratios using three different models of the "Deflate" for C62x VLIW processor

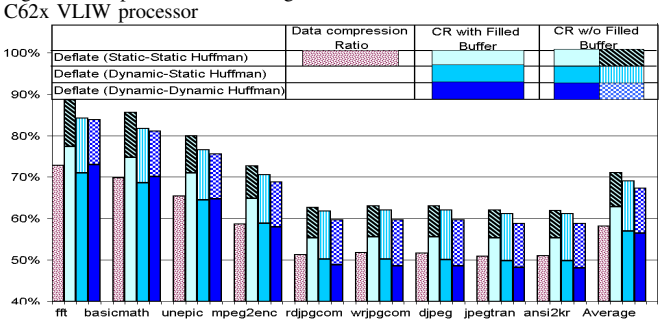

Fig. 10. Compression ratios using three different models of the "Deflate" for C64x VLIW processor

B. Compression ratio

Figures 8 shows the compression ratio using LZSS and for 4-bit, 5-bit and 8-bit pattern length. The pattern of 8-bit gives (in average) better compression ratio than other patterns length (82.78%, 82.11% and 81.61% for 4-bit, 5-bit and 8-bit, respectively). For that, we selected an 8-bit length pattern to be compressed as one symbol in "Deflate". Figures 9 and 10 show the compression ratios for different benchmarks using the three models of the "Deflate" algorithm for the C62x and C64x processors respectively. The first bar of each benchmark in these figures shows the compression ratio when the first model of "Deflate" algorithm (Static-Static Huffman Tables) is used as a data compression technique. The second, third and fourth bars show the compression ratios for the first model (Static-Static Huffman Tables), the second model (Dynamic-Static Huffman Tables) and the third model (Dynamic-Dynamic Huffman Tables) of "Deflate" algorithm, respectively. In each of theses bars, the bottom part shows the results by using our *Filled Buffer Technique* in that model. The whole bar presents the results without using our technique. We can observe that the data compression ratio is improved when the size of the benchmark is increased. The average data compression ratios for both processors are 56% and 58%. The difference between the first and second bars of each benchmark in figures 9 and 10 shows the overhead incurring when applying code compression in comparison to data compression. The overhead (in average) is 27% for C62x and 13% for C64x. The final compression ratios (using our *FBT*) for both processors are in Average 71% and 62% in the first model, 62% and 56% in the second model and 61% and 56% in the third model, respectively. We can conclude from the Figures 9 and 10 that the "Dynamic-Static Huffman Tables" model gives better compression ratio results for the small size benchmarks such that fft and Basicmath (which have less than 20000 instruction

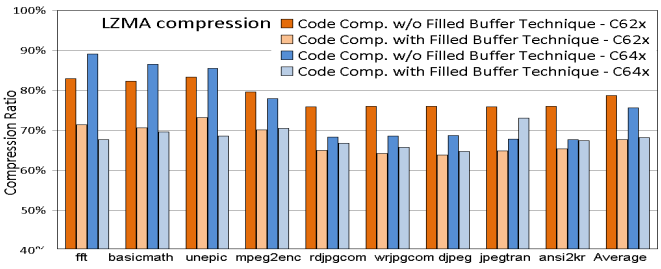

Fig. 11. Compression ratios using "LZMA" algorithm for C62x and C64x VLIW processors

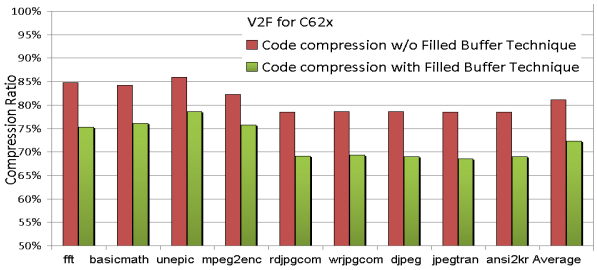

Fig. 12. Compression ratios using our *Filled Buffer technique* combined with the V2F [15] for the C62x VLIW processor

word). For the largest benchmarks (which have more than 20000 instruction word), the "Dynamic-Dynamic Huffman Tables" model is the best choice for compression from the perspective of compression ratio. To show the orthogonality of our *Filled Buffer Technique* we applied it to another data compression algorithm of the (Lempel-Ziv) family Which is called the "LZMA". "LZMA" (Lempel-Ziv-Markov chain-Algorithm) [12] is a data compression algorithm which is based on the LZSS algorithm combined with Range Coding. The *Extended Blocks* are compressed using the algorithm LZSS (as explained in Sec. III-A) and then the compressed stream is encoded using the Range Coding. Fig. 11 shows the compression ratio results both processors. Using the *Filled Buffer Technique* at the top of "LZMA" improved the compression ratios in average by 13% and 7% for both processors and final compression ratio of 67% and 68% were achieved. Comparing the results of "Deflate" and "LZMA" algorithms, the later one gives better data compression ratios, but the former one gives better compression ratios. As the hardware decoder of the work of Xie et al. [15] decodes one "Fetch Packet" in each clock cycle (parallel decoding), we selected their compression scheme (V2F) to apply our compression technique on it and to improve their final compression ratio (which was 82%). For that, we compressed the *Extended Blocks* using LZSS algorithm and then encoded the compressed stream with the V2F scheme (which is used in [15]). Fig. 12 shows the compression ratio results without using the *Filled Buffer Technique* (first bar) and after using it (second bar) for the C62x processor. Using the *Filled Buffer Technique* improved the average compression ratio from 82% to 72% (10% compression ratio improvment).

C. Performance

Fig. 13 shows the performance of hardware decoder for C62x and C64x processors, respectively. In this figure, we can observe that the performance is better for the largest size of benchmarks like "jpegtran" and "ansi2kr" (4.27 Byte/Cycle

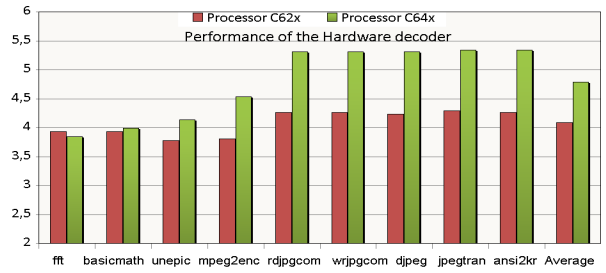

Fig. 13. Performance of the "Deflate" hardware decoder for C62x and C64x processors

for C62x and 5.34 Byte/Cycle for C64x). This is because the *Extended Blocks* in the large benchmarks contain more number of instructions than the small one and consequently, they have more compressed "Fetch Packets" which are decompressed with less number of clock cycles. In average, the performance of 4 and 4.8 Byte/Cycle was achieved for C62x and C64x, respectively. The performance may be improved by improving the decoder to decode the compressed instruction in parallel.

VI. Conclusion

We have presented a new approach for embedded system code compression. We have showed that the compression ratio of the sliding window compression algorithms may be improved by using our *Filled Buffer Technique*. Our approach can be used with any previous work which belongs to the Lempel-Ziv family algorithms to improve its compression results. We achieved an average compression ratios of 61% and 56% and performances of 4 and 4.8 Byte/Cycle for the TMS320C62x and TMS320C64x, respectively.

References

[1] Virtex-II platform FPGA user guide. http://www.xilinx.com/support/.
[2] T. Bonny and J. Henkel. Efficient Code Density Through Look-up Table Compression. *Design Automation and Test in Europe Conference (DATE07)*, pp. 809-814, 2007.
[3] V. Cuppu. Cycle Accurate Simulator for TMS320C62x, http://www.cs.cmu.edu/afs/cs/academic/class/15745-s07/www/c6xref.
[4] J. Fritts. MediaBench Project. *http://euler.slu.edu/ fritts/mediabench/.*
[5] M. Guthaus and J. R. et al. MiBench: a free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2002.
[6] T. M. Kemp, R. K. Montoye, D. J. Auerbach, J. D. Harper, and J. D. Palmer. A De-compression Core for PowerPC. *IBM Journal of Research and Development, VOL. 42, NO. 6,*, Sep., 1998.
[7] S. Klein. Space- and Time-Efficient Decoding with Canonical Huffman Trees. *Proceedings of the 8th Annual Sym. on Combinatorial Pattern Matching*, 1997.
[8] C. Lin, Y. Xie, and W. Wolf. LZW-Based Code Compression for VLIW Embedded Systems. *Proc. of the Design, Automation and Test in Europe conf.(DATE04)*, 2004.
[9] S. K. Menon and P. Shankar. Space/time Tradeoffs in Code Compression for the TMS320C62x Processor. *Technical Report IISc-CSA-TR-2004-4, Indian Institute of Science,*, 2004.
[10] M. Ros and P. Sutton. A hamming distance based VLIW/EPIC code compression technique. *Proc. of the 2004 inter. conf. on Compilers, architecture and synthesis for embedded systems,*, pp. 132-139, 2004.
[11] M. Ros and P. Sutton. A post-compilation register reassignment technique for improving hamming distance code compression. *Proc. of the 2005 inter. conf. on Compilers, architecture and synthesis for embedded systems,*, pp. 97-104, 2005.
[12] D. Salomon. Data compression: The complete reference. 2007.
[13] TMS320C62x DSP CPU and Instruction Set Reference Guide.
[14] TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide.
[15] Y. Xie, W. Wolf, and H. Lekatsas. Code Compression Using Variable-to-fixed Coding Based on Arithmetic Coding. *Proceedings of the Conference on Data Compression*, pp. 382-391, 2003.
[16] Y. Xie, W. Wolf, and H. Lekatsas. Code compression for VLIW processors using variable-to-fixed coding. *IEEE Trans. on Very Large Scale Integrated System,*, pp. 525-536, 2006.