HW/SW Partitioning of an Embedded Instruction Memory Decompressor

Shlomo Weiss and Shay Beren
EE-Systems, Tel Aviv University
Tel Aviv 69978, ISRAEL

ABSTRACT

We introduce a new PLA-based decoder architecture for random-access run-time decompression of compressed instruction memory in embedded systems. The compression method employs class-based coding. We show that this new decoder architecture can be extended to provide high throughput decompression. The design of the decompressor is based on the following HW/SW tradeoff: decoding is done in hardware to provide high throughput, yet the codebook used for decompression is fully programmable.

Keywords: embedded systems, compressed instruction memory

1. INTRODUCTION

The program memory of a system-on-chip (SOC) design, usually ROM or flash memory, may take a substantial portion and sometimes more than half [9] of the chip's area. By compressing the programs that reside in the instruction memory, we can reduce the size, and therefore the cost of the embedded system, or alternatively we can store in the program memory larger and more sophisticated applications. Thus object code compression in a SOC offers the following tradeoff: investment in hardware (decompressor unit) helps to reduce the size of the software (application programs, real-time operating system), without reducing the functionality of the software. There is also another hardware/software tradeoff that has to do with the design of the decompressor itself: the codebook used for decompression may be fixed in hardware, or stored in on-chip memory. In the latter case, the compression method may be adapted to the symbol frequency statistics of a given set of programs. In the rest of this section we briefly review relevant work on the design of decompressors.

Sun and Lei [10, 7] describe the design of a constant-output-rate decoder for compression systems in advanced television applications. Chang and Messerschmitt [2] and Lin and Messerschmitt [8] present VLSI architectures and parallel decoding methods for variable-length-code decoders. While the primary application they envision is high throughput video compression systems, their work is generally applicable to compression systems that use a prefix code, such as the Huffman code [3].

In this paper we focus on class-based coding (see Section 3). Building on the work of Sun and Lei [10, 7] and Chang and Messerschmitt [2], in Section 4 we describe a new decoder architecture designed to decode class-based codewords. The new architecture uses a PLA to decode class codes and, as in IBM's CodePack [5], a ROM to decode symbols in each class. From the published literature [5, 4, 6], it appears however that the IBM design is not a PLA-based design. Also, it is not clear to what extent the CodePack implementation may be scaled to provide higher throughput. The scalability of our PLA-based design is investigated in section 6.

In the Sun and Lei design, any modification in the codebook requires changes in the PLA specification; this implies that PLA optimization software must be run again, with results that are difficult to predict in terms of the PLA area. Such changes are normally not done after a custom-designed chip becomes a product. This is also a limitation of the Chang and Messerschmitt variable I/O rate decoder, which is discussed in the Conclusions section of their paper [2]. (Actually, programmability of the codebook is mentioned in [2] as a possible, but "not a simple extension" of their work). Our architecture offers codebook programmability. The PLA specification and its implementation are fixed; the compression algorithm may be adapted to the statistics of various object programs by modifying the symbol codebook, which is stored in a ROM and is fully programmable.

After a discussion on class-based code (Section 3), we introduce a class-based decoder architecture in Section 4. In Section 5 we show how this new decoder architecture can be extended to provide higher throughput. In section 6 we apply our decoder architecture to a specific class-based compression algorithm.

2. BACKGROUND

In this section we briefly review the Huffman code and a decoder architecture for it. Huffman coding [3] assigns variable-length codes to the symbols of an alphabet based on the frequency of occurrence of a symbol in the text or object file. As shown in the example below, frequent symbols are assigned short codes.

Example 1

Table 1 illustrates the code assignment for an eight-symbol alphabet.

Alphabet	Frequency	Codeword
A	0.50	0
B	0.15	110
C	0.11	100
D	0.09	101
E	0.07	1110
F	0.05	11110
G	0.02	111110
H	0.01	111111

Table 1: An example of Huffman code. The average code length is 2.26 bits.

Sun and Lei [10] designed a decoder architecture for advanced television applications. It decodes variable-length code at a constant output rate of one symbol per clock cycle. As shown in Figure 1, the core of the decoder is a PLA. Assuming an alphabet size of 2^n symbols and the use of bounded Huffman code [11] such that the longest codeword does not exceed w bits, then the PLA implements a truth table with 2^n product terms, w-bit wide input, and two outputs: the n-bit decoded symbol, and the codeword length encoded in $\log_2 w$ bits.

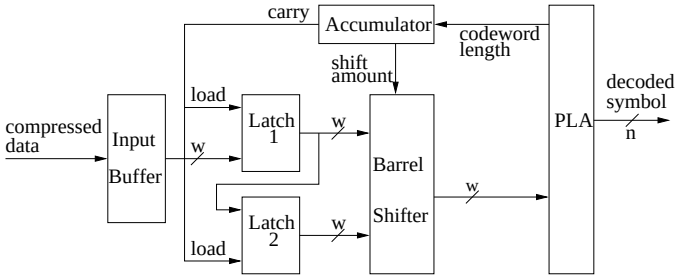

Figure 1: The one-symbol-per-cycle output decoder designed by Sun and Lei.

Still referring to Figure 1, the accumulator adds up the codeword length for each decoded symbol, and controls the barrel shifter. When the accumulator exceeds the maximum codeword length w, it produces a carry that transfers the contents of Latch1 to Latch2, and loads w bits from the Input Buffer into Latch1.

3. CLASS-BASED CODE

An obvious problem with Huffman codes is that the decoding performance is limited by the recursive nature of the decoding process. We do not know where the code for a symbol begins until we decode the previous symbol.

Another problem is that practical implementation of Huffman codes is limited by the size of the alphabet, which determines the size of the Huffman tree. If we consider an object file as a sequence of bytes, the alphabet consists of $2^8 = 256$ symbols. We may prefer to look at the same object file as a sequence of 16-bit symbols, in which case the alphabet size is $2^{16} = 65,536$. Although the choice of 16-bit symbols might give better compression performance, especially if the object file consists of fixed-length 32-bit RISC instructions [5], maintaining a full Huffman tree with 2^{16} leaf nodes is prohibitively expensive both in terms of storage space and decoding speed.

Both of these problems may be addressed by using classes. A *class* is a group of symbols that are assigned codes with the same length. Every symbol in the alphabet belongs to a single class. Every class is identified by a unique *class-code*. If a class consists of 2^q symbols, a q-bit *symbol-code* is appended to the class-code to identify each symbol that belongs to that class. A *codeword* consists of a *class-code* followed by a *symbol-code*.

Example 2

This example illustrates class-based coding for an alphabet with eight symbols. Table 2 shows the class structure. The class-based code, shown in Table 3, is suboptimal compared with the Huffman code (Example 1).

Number of symbols in the class	Class structure
1	0
2	1 0 b
5	1 1 b b b

Table 2: Class structure with three classes. The alphabet consists of eight symbols. Following the class-code, in each class there is a sequence of zero or more bits b, which are used to encode the symbols in that class.

Alphabet	Frequency	Class	Codeword
A= 000	0.50	0	0
B= 001	0.15	10b	100
C= 010	0.11	10b	101
D= 011	0.09	11bbb	11011
E= 100	0.07	11bbb	11100
F= 101	0.05	11bbb	11101
G= 110	0.02	11bbb	11110
H= 111	0.01	11bbb	11111

Table 3: An example of class-based code, using the classes defined in Table 2. The average code length is 2.48 bits, longer than the average Huffman code length shown in Table 1. Decoding, however, is simplified because it is sufficient to look at the first two bits to determine the code length.

By using classes we split the decoding process into two phases: (1) determining the code length by decoding the class-code, and (2) decoding the symbol-code by accessing a lookup table. This simplifies decoding, and helps with the first problem mentioned above (the recursive nature of decoding) because class codes are typically short, and it takes less effort to determine the code length in comparison with the Huffman code with the same size alphabet.

We define a class of *literals* – symbols whose contents is not changed by the coding process. Literals are coded by simply attaching to them the class-code. The class of literals contains symbols that have the lowest frequencies.

Example 3

In Table 3, the five symbols that belong to the third class (class-code "11") are literals. The symbol itself is contained in the codeword following the class-code.

Literals help with the second problem mentioned above (the size of the Huffman tree) because literals are used to reduce the size of the lookup table; in fact, if we use 16-bit symbols we are likely to encode most of the 65K symbols as literals, and maintain a relatively small lookup table that stores only the most frequently used symbols. To build a full Huffman tree we have to limit ourselves to a small alphabet; 8-bit symbols are typically used, 16-bit symbols would require a huge 65K Huffman tree. We can construct, however, class-based code using 16-bit symbols.

4. CLASS-BASED DECODER ARCHITECTURE

The Sun and Lei decoder (Section 2) uses 8-bit symbols, and a maximum codeword length of 16 bits. It requires a reasonable sized PLA: 16-bit input, 12-bit output (8-bit symbol and 4-bit codeword length), and 256 product terms. As it is, this design cannot be used for an alphabet size of 2^{16} symbols because the PLA would require 65,536 product terms. In this section we adapt the Sun and Lei design to class-based codes by making the following changes.

1. The alphabet size is large, its size might be 2^{16} symbols or even larger.
2. The PLA is used to decode the class-code, but not the symbols. This results in a small PLA, because the number of product terms is equal to the number of classes, not to the number of symbols in the alphabet.
3. A relatively small ROM stores the most frequently used symbols (typically 512 or 1024 symbols). The remaining symbols are encoded as literals.

As shown in Figure 2, the PLA decodes the class-code and generates the following output:

1. *Codeword length.*
2. *Mask control* (the number of 1's in the mask).
3. ROM *address* of the block of symbols for the corresponding class, if the symbol is not a literal.
4. A *select literal* signal if the symbol is a literal, as determined by the class-code.

The codeword length is used as the rotate left control; the result is a right-aligned codeword. The mask is used to clear the class-code and the extra (unused) bits in the input word. The mask output is a word that contains a single right-aligned symbol-code (that is, a codeword stripped of its class-code), padded with zeros up to the length of the longest symbol-code.

If the "select literal" signal is active, the mask output is a literal, which is selected as the decoder output. Otherwise, the mask output is interpreted as the low-order ROM address bits, which are ORed with the high-order address bits produced by the PLA (the block address of the block of the symbols that belong to this class), to yield the ROM address of the symbol.

Example 4

Table 4 illustrates the PLA specification for the code of Example 2, and Table 5 shows the contents of the ROM used to decode the eight-symbol alphabet of the example.

PLA Input (contains at least one class-code)	PLA Output			
	Select Literal	Rotate Control (codeword length)	Mask Control (symbol length)	High-Order ROM Address
0x	0	000	00	00
10	0	010	01	10
11	1	100	11	xx

Table 4: PLA specification for the code of Example 2. Since the shortest codeword is 1-bit long, codeword length 000 denotes a 1-bit (the shortest) codeword, and codeword length k is interpreted as a length of $k+1$ bits. When "Select literal" is 1, the symbol itself is contained in the codeword following the class-code, and the ROM address is xx since, whatever the ROM output is, it will not be selected as the decoder output.

Class	Codeword	ROM Address			ROM Contents (Decoded Symbol)
		High-Order (PLA Output)	Low-Order (Rotate Output)	Full Address (OR output)	
0	0	00	00	00	A
10b	100	10	00	10	B
10b	101	10	01	11	C

Table 5: Address generation and contents of the ROM used to decode the first three symbols of the eight-symbol alphabet from Example 2. The remaining five symbols are literals.

5. MULTIPLE-SYMBOL-PER-CYCLE CLASS-BASED DECODER

The decoder described in the previous section produces one symbol per cycle. If the alphabet consists of 16-bit instruction halves of a typical RISC architecture with 32-bit long instructions, then decoding a 64-byte cache line of 16 instructions requires 32 decoding cycles. Even if the processor clock cycle is long enough to fit two decoding cycles, the decompression penalty still adds substantial delay to the memory path.

In this section we look at a method for increasing the decoding throughput by increasing the PLA complexity. The PLA can be used to decode multiple codewords by specifying it as a table of all possible combinations of the class-codes. To decode n codewords concurrently we slightly change the compression procedure. Every sequence of n codewords in the object file is encoded as $P_1P_2\ldots P_nS_1S_2\ldots S_n$ (rather than the normal $P_1S_1P_2S_2\ldots P_nS_n$ sequence), where P_i and S_i are the class-code and symbol-code of codeword i. If the

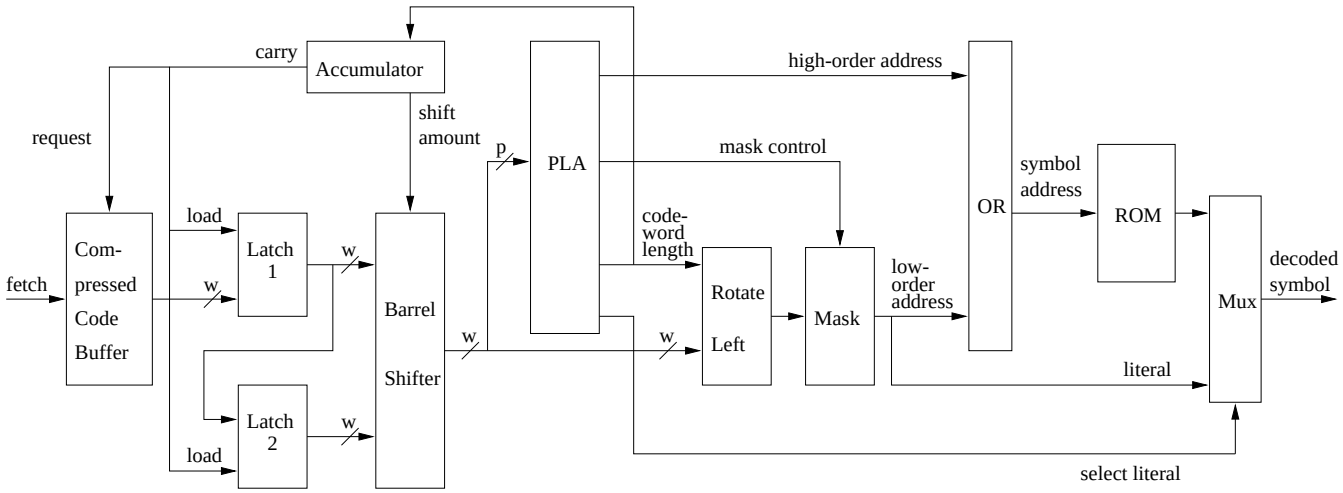

Figure 2: Class-based decoder architecture.

class-code is at most p bits long, the $n \cdot p$ input bits of the PLA contain n class-codees and possibly some extra bits.

As shown in Table 6, the PLA generates $n + 1$ output fields:

PLA Input (no. bits)	PLA Output (no. bits)		No. of Product Terms
	Class-codes $P_1P_2 \ldots P_n$ Combination no.	Symbol class no. (n symbols)	
$n \cdot p$	$\log_2$(no. combinations)	$n \cdot \log_2 c$	c^n

Table 6: The PLA size used to simultaneously decode n codewords, assuming the longest class-code is p bits and c classes.

1. One field that enumerates the combinations of class-codes $P_1P_2 \ldots P_n$. Combinations that give the same length of class-codes are considered identical.

2. n fields that contain the class number of each of the n symbols. The c classes are numbered $0 \ldots c-1$.

Figure 3 illustrates the structure of a four-symbol-per-cycle decoder. As shown in the figure, the symbol class number is decoded to produce the following:

1. The *select literal* signal, which is asserted if the class number identifies the class of literals.

2. The *high-order address* (block address) of the block of symbols that belong to this class.

3. The *symbol-code length.*

Conceptually, the four-symbol-per-cycle decoder shown in Figure 3 may be scaled to n symbols per cycle. Parallel decoding of n symbols is limited by the PLA complexity. In the next section we look at the PLA complexity for a specific class-based compression algorithm.

6. AN APPLICATION: IBM'S CODEPACK

In this section we apply the multiple-symbol-per-cycle decoder design presented above to a specific compression algorithm: IBM's CodePack [5]. CodePack is a class-based object-code compression method introduced in IBM's 405 PowerPC core. As in most other RISCs, PowerPC object code consists of fixed-length 32-bit instructions. CodePack compression (Figure 7) is done by using different class structures for the left 16-bit halves and for the right 16-bit halves of the instructions.

No. of symbols	Class structure	No. of symbols	Class structure
8	00 *bbb*	1	00
32	01 *bbbbb*	16	01 *bbbb*
64	100 *bbbbbb*	32	100 *bbbbb*
128	101 *bbbbbbb*	128	101 *bbbbbbb*
256	110 *bbbbbbbb*	256	110 *bbbbbbbb*
	111 *16-bit literal*		111 *16-bit literal*

Table 7: IBM's CodePack class structure. Coding is done differently for the left and the right halves of the instruction. In immediate-format instructions the right instruction-half is used for constants. The zero constant occurs frequently enough to justify its own code, it is the only symbol encoded in the first class of the right instruction halves.

There are $c = 6$ classes in CodePack, which we number $0 \ldots 5$. Three bits are required to encode the class number, and $3n$ bits to encode the class numbers of n instruction halves. Class-codes are either 2- or 3-bit long. Therefore, as shown in Table 8, the number of class-code combinations that give distinct lengths for the field of n class-codes is quite small and may be encoded in 1-4 bits, depending

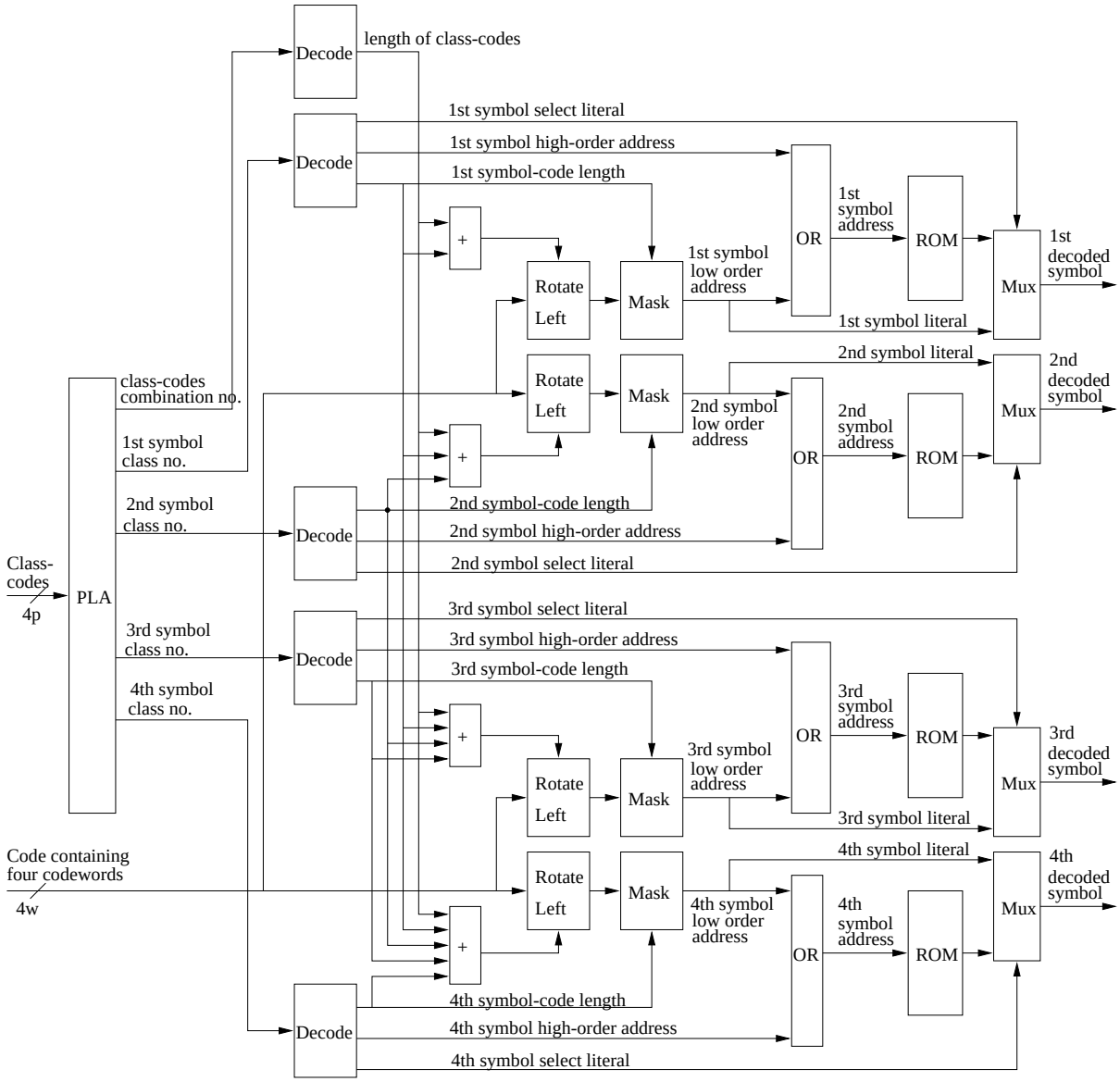

Figure 3: Class-based decoder, concurrent architecture that produces four symbols.

n symbols (Instruction Halves)	PLA Input (bits)	PLA Output (bits)			No. Product Terms	No. Minimized Product Terms Unified PLA	No. Minimized Product Terms Decomposed PLA
		Class-codes Combination no.	Symbol Class no.	Total Output			
1	1*3=3	1	1*3=3	4	6	4	5
2	2*3=6	2	2*3=6	8	36	14	15
3	3*3=9	2	3*3=9	11	216	30	35
4	4*3=12	3	4*3=12	15	1,296	92	75
5	5*3=15	3	5*3=15	18	7,776	238	155
6	6*3=18	3	6*3=18	21	46,656	554	315
7	7*3=21	3	7*3=21	24	279,936	—	—
8	8*3=24	4	8*3=24	28	1,679,616	—	—

Table 8: The number of PLA input and output bits and the number of product terms for an n-symbol-per-cycle decoder that decodes CodePack compressed object-code. The logic minimization software was unable to perform the PLA product term reduction for the last two rows of the table due to the huge size of the PLA.

on the number of symbols (instruction halves) decoded per cycle. The number of PLA terms is c^n for an n-symbol-per-cycle decoder with c classes. To reduce the number of PLA terms, we have used Espresso [1] to minimize two PLA configurations: (1) a *unified* PLA that produces all the output bits, and (2) a *decomposed* PLA which consists of two smaller PLAs, one that produces the "Class-codes Combination No." output, and a second PLA that generates the "Symbol Class No." for each of the n classes. Although the number of product terms before minimization is identical in both PLA configurations, as shown in Table 8 the minimized decomposed PLA has fewer product terms for $n > 3$.

For the PLA specified in Table 8, we were able to do the product term reduction up to $n = 6$. For larger n the software ran for a few days and eventually crashed due to the huge size of the PLA. For a four-symbol-per-cycle ($n = 4$) decoder, Verilog simulation of the design illustrated in Figure 3 shows, as expected, that the critical path of the design is the path that passes through the five-input adder (bottom of Figure 3). Timing simulation of the critical path shows that with 0.35-micron technology the decoding cycle can be done in 10 ns. This corresponds to a decompression throughput of eight bytes (four symbols) per clock cycle at a clock rate of 100 MHz.

7. CONCLUSIONS

We have presented an architecture for decoding class-based compressed object code in embedded processors. A central component of the architecture is a PLA that decodes the class-codes. The PLA is the only serial part of the decoder; the remaining operations of isolating the symbol-code and accessing the symbol codebook are done in parallel for all symbols. Thus the parallelism of the architecture is limited by the PLA complexity. By applying the proposed architecture to a specific class-based compression algorithm (IBM's CodePack [5]), we have shown that the proposed architecture scales to parallel decoding of four symbols. In a typical 32-bit RISC instruction set architecture, this corresponds to a decompression rate of eight bytes per decoding cycle.

Many embedded processors run at clock rates much slower than desktop systems. For example, a state-of-the-art automotive controller, the Motorola MPC 555 [9] runs at only 40 MHz. We expect that the next generation of similar automotive/industrial controllers will run at 100 MHz. Although the proposed decoder design has not been fabricated, we have implemented the design in Verilog. Based on simulation of the critical path of the design with 0.35-micron technology, we estimate that at our target rate of 100 MHz the decoder achieves a decoding rate of eight bytes per clock cycle.

In a system with a compressed main-memory and a decompressed instruction-cache, the decompression penalty is paid only when a miss occurs. Some embedded systems, however, do not have an instruction cache for a number of reasons, including the following:

1. Due to the relatively slow clock, the latency of the on-chip program memory is quite short and an instruction cache is simply not needed. For example, in the MPC 555 [9], the 448 KB on-chip flash memory provides burst access with an initial latency of only two clock cycles. The chip contains a burst buffer, but no cache.
2. Some embedded processors control real-time systems with critical safety requirements. In these systems the worst case execution time (WCET) must be determined for all programs. Caches, however, have unpredictable timing and make WCET analyses difficult.

In such cacheless, compressed-memory, embedded systems, high-throughput decompression is crucial.

A high-throughput decoder requires substantial hardware resources. Using again CodePack as a case study, parallel decoding of four half-instructions requires access to 4 ROMs, 1KB each, a total ROM size of 4 KB. This is a reasonable investment, however, considering that code compression saves on the average 40% [5] of the program memory. For example, 2 MB of object code could be stored in compressed form in only 1.2 MB of ROM or flash memory, in effect saving 800 KB of on-chip memory.

8. ACKNOWLEDGEMENT

We thank Gil Lidji for helpful discussions in the course of this research.

9. REFERENCES

[1] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Sythesis*. Kluwer Academic Publishers, 1984.

[2] S. Chang and D. G. Messerschmitt. Designing high-throughput VLC decoder Part I – concurrent VLSI architectures. *IEEE Transactions on Circuits and Systems for Video Technology*, 2(2), June 1992.

[3] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, 40(9):1098–1101, September 1952.

[4] IBM. *CodePack: PowerPC Code Compression Utility User's Manual. Version 3.0*. International Business Machines (IBM) Corporation, 1998.

[5] T.M. Kemp, R.M. Montoye, J.D. Harper, J.D. Palmer, and D.J. Auerbach. A decompression core for PowerPC. *IBM Journal of Research and Development*, 42(6), Nov 1998.

[6] C. Lefurgy, E. Piccininni, and T. Mudge. Analysis of a high-performance code compression method. In *Proc. Int'l Symp. on Microarchitecture*, Haifa, Israel, November 1999.

[7] S. M. Lei and M. T. Sun. An entropy coding system for digital HDTV applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 1(1), March 1991.

[8] H. Lin and D. G. Messerschmitt. Designing high-throughput VLC decoder Part II – parallel decoding methods. *IEEE Transactions on Circuits and Systems for Video Technology*, 2(2), June 1992.

[9] Motorola. *MPC 555 User's Manual. MPC555UM/AD*. Motorola Semiconductor Products Sector, 1999.

[10] M. T. Sun and S. M. Lei. A parallel variable-length-code decoder for advanced television applications. In *Proc. 3rd International Workshop on HDTV*, Torino, Italy, 1989.

[11] D. C. Van Voorhis. Constructing codes with bounded codeword lengths. *IEEE Transactions on Information Theory*, 20(3):288–290, March 1974.