Bitwise and Dictionary Modeling for Code Compression on Transport Triggered Architectures

J. Heikkinen, P. Kuukkanen, and J. Takala
Institute of Digital and Computer Systems
Tampere University of Technology
P.O.Box 553, 33101 Tampere, FINLAND
jari.heikkinen@tut.fi

ABSTRACT
Program code size has become a critical design constraint of embedded systems. Code compression is one of the approaches to reduce the program code size; it results in smaller memories and reduced cost of the chip. In this paper, bitwise and dictionary modeling schemes for code compression are evaluated on transport triggered architecture processors, designed for four applications taken from different application domains. Results of the bitwise modeling scheme are promising, whereas the dictionary modeling scheme needs further improvements to be profitable.

KEY WORDS
transport triggered architecture, customizable processor architecture, digital signal processing, program compression, bitwise modeling, dictionary modeling

1 Introduction

Very long instruction word (VLIW) architectures are widely used in embedded systems, especially in digital signal processing (DSP) domain, due to their modularity and scalability. VLIW is an instruction-set philosophy which exploits the instruction level parallelism by executing several operations in parallel in concurrently operating functional units. The concurrently operating functional units are controlled by a long instruction word that contains dedicated fields for each functional unit. Each field contains a RISC-type operation that controls a single functional unit in the architecture. Unfortunately, this kind of an instruction encoding approach usually leads to poor density [1], and implies a need for large memories. In embedded systems, large memories are expensive and raise the power consumption of the chip.

Straightforward way to increase the code density, and thus decrease the size of the memories on the chip, is to compress the instruction words of the program code and store them into the memory in compressed form. During execution, each instruction word is individually fetched, decompressed and executed. Systems where code is interpreted from compressed form are called compressed-code architectures [2]. Variety of compression methods have been considered for VLIW architectures. In one of the earliest approaches, no-operations (NOP) are eliminated from the instructions [1]. A “mask” identifier preceding each instruction specifies which fields are present in the instruction word. In [3], NOPs in the instruction words are avoided by assembling the original VLIW instructions from a limited set of operation slot pairs, denoted as functional unit instruction words (FIW), which control the associated functional units regardless of the remaining FIWs. In [4], a dictionary-based compression method is applied to VLIWs to improve the code density. Instructions are separated to operand and opcode streams to find more redundancy. Dictionary compression is then applied to both streams by storing unique bit patterns into a dictionary and replacing these bit patterns in the actual program with codewords. Dictionary-based compression has also been applied in [5], where the non-time-critical part of the program is compressed using *superinstructions* that correspond to frequently used instruction patterns. Entropy encoding exploits the fact that some symbols are used more frequently than others. Therefore, the shortest codes are allocated to the most frequent symbols and vice versa. Entropy encoding has been applied on VLIWs by means of arithmetic coding, e.g., in [6], and Huffman encoding, e.g., in [7].

Transport triggered architecture (TTA) is a class of statically programmed instruction-level parallel (ILP) architectures that reminds VLIW architectures [8]. In the TTA programming model, the program specifies only the data transports (moves) to be performed by an interconnection network. Operations occur as a “side-effect”. Thus, TTA has also a flavour of dataflow machines. A TTA processor consists of a set of functional units and register files. These structures are connected to an interconnection network consisting of buses through input and output sockets as illustrated in Fig. 1. The architecture is extremely flexible and modular and it allows easy inclusion of custom user-defined functional units.

TTA processors can be designed with MOVE framework, a toolset that provides a semi-automatic design process for designing application-specific instruction set processors [9]. MOVE framework exploits the scalability, flexibility and simplicity of the TTA in designing a processor for a given application. The design flow consists of

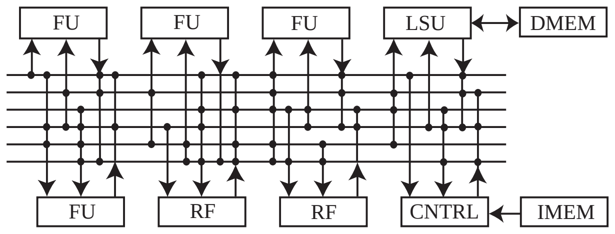

Figure 1. TTA processor organization. FU: functional unit. RF: register file. LSU: load-store unit. CNTRL: control unit. DMEM: data memory. IMEM: program memory. Dots represent connections between buses and sockets.

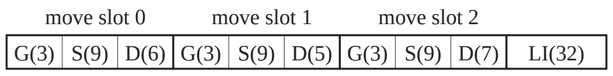

Figure 2. Structure of instruction word. G: Guard field. S: Source ID field. D: Destination ID field. LI: Long immediate field. (x): x-bit field.

three main components: design space explorer, hardware subsystem, and software subsystem. Processor configurations yielding the best cost/performance ratio are searched with the design space explorer. The design space explorer evaluates several different TTA processor configurations in terms of performance, area and power consumption. The hardware subsystem is responsible for generating a structural hardware description (VHDL) of the chosen processor configuration. Commercial tools can then be used to perform logic synthesis, placement, and routing to obtain the layout of the processor. The software subsystem generates the ILP code for the chosen processor configuration.

As VLIW processors, also TTA processors suffer from poor code density. The poor code density is mostly due to minimal instruction encoding, which leads to long instruction words. The bits of the instruction word are sent directly to the data path without being decoded first in a decoder unit. The long instruction word contains dedicated fields, called move slots, for each bus to define data transports on the buses. Each move slot contains three fields, as illustrated in Fig. 2. The guard field specifies the guard value that controls whether the data transport on the bus is executed or not. The destination ID field contains the address of the socket that reads data from the bus. The source ID field contains the address of the socket that writes data on the bus. In addition to move slots, instruction words may contain dedicated long immediate fields to define long immediate values, which are mainly used to specify jump addresses and large constant values. Another reason for poor code density is that the hardware resources are typically tailored for the highly parallel sections of the program. The less parallel parts result in large number of null data transports, which increase the size of the program code.

In this paper, the performance of two different compression methods is evaluated on TTA processors. Instructions of TTA processors, designed for four applications taken from different application domains, are statistically analyzed to approximate the compressibility of typical TTA code. The first method, bitwise modeling, builds probability distributions of the individual bits of the instruction words. These distributions are then stored as a model and used to compress the instruction words with an entropy coder. The second method, dictionary modeling, stores the instruction words into a separate dictionary and replaces the original code by indices pointing to the dictionary.

2 Bitwise Modeling

Bitwise modeling is based on context modeling, where a preceding event is used to model the next one. In bitwise modeling, this means that the next bit to occur is predicted using the preceding bits. The next bit is said to occur within the context of a set of preceding bits. The length of the context is denoted as the order of the model. The probabilities of zeroes and ones, $p(0)$ and $p(1)$, are determined in the different bit positions of the instruction word. Since $p(1) = 1 - p(0)$, only $p(0)$ is stored. Each probability is stored using one byte. These probabilities are then used to generate a model for predicting the next bit to occur. The probabilities can be coded by Huffman encoding [10] (combining several consecutive bits into one symbol) or quasi-arithmetic coding [6].

Three orders of the bitwise model are considered. In order 0 modeling, each position is handled independently. The probabilities are independent: they have no context. The model contains as many bytes as there are positions in the instruction word. In order 1 modeling, each probability has one previously occurred bit as its context. The context position is chosen optimally in such a way that 1) the entropy is as small as possible and 2) several parts of the instruction word can be decompressed simultaneously, in parallel. The easiest way to achieve this is to keep the positions relatively close to the position where the probabilities are collected. For each position, the model contains three bytes: 1) probability of zero when the previous position contains zero, $p(0 \mid 0)$; 2) probability of zero when the previous position contains one, $p(0 \mid 1)$; and 3) the index of the affecting position. In order 2 modeling, each probability has two previously occurred bits as its context. The positions of the context bits are again chosen optimally. For each position, the model contains 6 bytes: $p(0 \mid 00)$, $p(0 \mid 01)$, $p(0 \mid 10)$, $p(0 \mid 11)$, and the indices of the two positions. The model size increases exponentially when the order grows. Due to this, order 2 was considered to be the highest order worth evaluating.

When the probabilities of each position are collected, the entropy can be calculated for each individual bit position. Entropy, denoting the average amount of information per symbol s over the whole alphabet, is given by

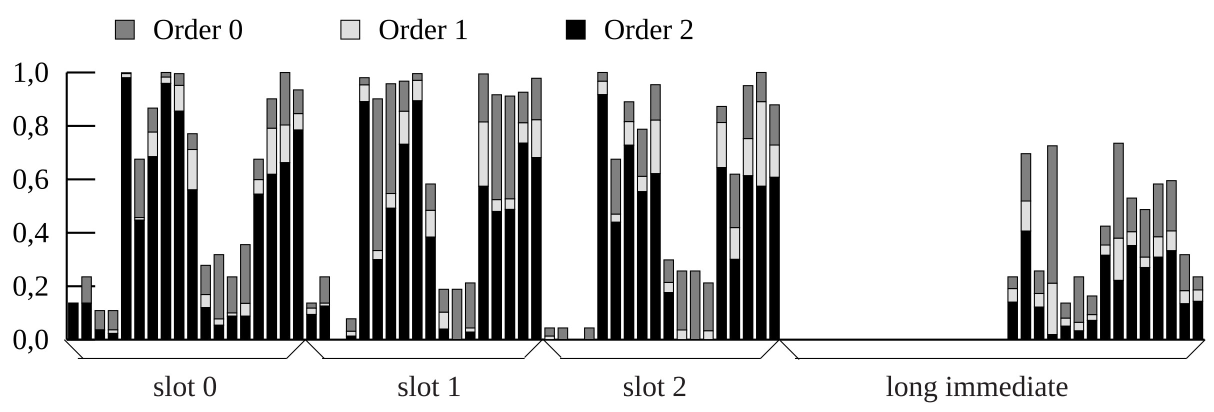

Figure 3. Optimal compression ratios of individual bits of an instruction word of a 3-bus TTA processor designed for 8×8 2-D DCT application

$$H = \sum_{s} -p(s)log_2(p(s)) \qquad (1)$$

where $p(s)$ is the probability of a symbol. If the symbols appear independently and with assumed probabilities, entropy can be considered as lower bound of compression, i.e., optimal compression ratio. This can be achieved only by an optimal entropy coder. However, arithmetic coding and Huffman encoding perform well enough so that the entropy can be considered as a good estimation of the compression ratio.

Entropy, given by equation 1, can be applied to calculate the entropies, i.e., compression ratios, of each bit in the instruction word. In order 0 modeling, the entropy of a bit is given by

$$H_{bit} = -p(0)log_2(p(0)) - p(1)log_2[p(1)] \qquad (2)$$

where $p(0)$ is the probability of a bit being zero and $p(1)$ the probability of a bit being one. In orders 1 and 2 the context needs to be taken into account. The entropy for the orders 1 and 2 is given by

$$H_{bit} = \sum_{i} c_i[-p_i(0)log_2(p_i(0)) \\ -p_i(1)log_2(p_i(1))] \quad i = \begin{cases} 0..1 & \text{order 1} \\ 0..3 & \text{order 2} \end{cases} \qquad (3)$$

where c_i is the probability of the context, and $p(0)$ and $p(1)$ are the probabilities of bit being zero or one. For order 1 there are two possible contexts, the preceding bit being '0' or '1'. Correspondingly, for order 2 there are four possible contexts, preceding two bits being '00', '01', '10', or '11'.

Figure 3 shows optimal compression ratios of the bit positions of an instruction word of a TTA processor designed for discrete cosine transform (DCT) application. Some of the bits have the same value in each instruction word and are thus compressed down to 0 percent, meaning that they can be removed. Such bits may be, for example, the most significant bits of long immediate fields. The width of the long immediate field is restricted to 32 bits in the current version of the MOVE framework toolset. Because the DCT application requires only 16 bit long immediates, 16 leftmost bits of the long immediate field are unused and they can be removed.

The bitwise modeling was applied to the instructions of TTA processors designed with the MOVE framework for four benchmark applications from DSP and multimedia domains. Three configurations, a high-performance configuration, a cost-efficient configuration, and a configuration being a compromise between cost and performance were chosen for each benchmark. The DSP benchmarks realized two versions of the discrete cosine transform (DCT), two-dimensional (2-D) 8×8 DCT and 1-D 32-point DCT, and Viterbi decoding. The multimedia application, taken from the MediaBench benchmark set [11], implemented MPEG2 decompression. The applications and their code statistics are illustrated in Table 1. Target processors are classified according to the number of buses in the configuration. Other resources scale correspondingly.

Table 1. Code statistics of benchmark applications

Application	Buses	Instr. width [bits]	Instr. count	Code size [bytes]
8×8 2-D DCT	3	88	208	2288
	8	184	138	3174
	13	272	127	4318
32-point DCT	4	96	1038	12456
	8	160	502	10040
	10	200	477	11925
Viterbi decoding	3	96	385	4620
	6	152	262	4978
	9	208	253	6578
MPEG2 decoding	5	128	8550	136800
	6	152	8018	152342
	8	184	7652	175996

Table 2. Results of order 0 bitwise modeling

Application	Buses	Model size [bytes]	Compr. instr. width [bits]	Compr. code size + model [bytes]	Compr. ratio [%]
8 × 8 2-D DCT	3	67	40.0	1106	48.3
	8	142	76.4	1460	46.0
	13	189	88.2	1590	36.8
32-point DCT	4	92	65.9	8641	69.4
	8	149	97.3	6255	62.3
	10	179	116.0	7090	59.5
Viterbi decoding	3	88	43.8	2197	47.5
	6	146	77.4	2676	53.8
	9	195	93.2	3143	47.8
MPEG2 decoding	5	127	56.2	60159	44.0
	6	134	64.7	64964	42.6
	8	184	73.9	70825	40.2

Table 3. Results of order 1 bitwise modeling

Application	Buses	Model size [bytes]	Compr. instr. width [bits]	Compr. code size + model [bytes]	Compr. ratio [%]
8 × 8 2-D DCT	3	192	28.3	928	40.5
	8	401	41.2	1113	35.1
	13	526	42.6	1202	27.8
32-point DCT	4	205	33.6	4567	36.7
	8	373	59.6	4111	40.9
	10	444	61.1	4089	34.3
Viterbi decoding	3	222	31.9	1758	38.0
	6	393	50.7	2054	41.3
	9	528	51.0	2141	32.5
MPEG2 decoding	5	339	34.1	36761	26.9
	6	352	37.8	38241	25.1
	8	508	42.2	40848	23.2

Table 2 illustrates the results of the bitwise modeling of order 0. Tables 3 and 4 illustrate the results of order 1 and order 2, respectively. The tables depict the size of the probability distribution model, the width of the compressed instruction, the total code size (including the size of the model), and the compression ratio. The width of the compressed instruction word is an average of all the instruction word widths of the compressed code. The compression ratio is the theoretical limit when using the bitwise modeling scheme. It can be achieved only by an optimal entropy coder.

Already the order 0 model shows a significant reduction in code size. On average, a compression ratio of 49.9 percent is obtained. As higher orders are used, the compression ratio decreases even further, though the improvement achieved with the order 2 compared to the order 1 is negligible. With the order 1 modeling, the compression ratio is on average 33.5 percent and with the order 2 modeling 33.0 percent. The average width of the compressed instruction shortens as higher modeling orders are used, but on the other hand, the size of the model increases. With lower orders, order 0 and order 1, the size of model does not have a big effect on the total code size, but with the order 2, the size of the model has a notable effect on the total code size in most of the applications. Because of this, the compression ratio decreases only slightly when the order 2 is used instead of the order 1. Taking into account the increased complexity of the decompression hardware with higher modeling orders, order 1 seems to be the most reasonable modeling order to be used.

The size of the processor configuration has a notable effect on the results. Bigger processor configurations result in bigger models, but on the other hand, smaller compression ratio. Bigger processor configurations have more buses, i.e., there are more move slots that increase the width of the instruction. As there are more bits in the instruction word, the size of the model increases. However, the compiler is rarely capable of fully utilizing all the buses of the architecture. This results in large number of empty data transports in the program code, which are optimized away with the bitwise modeling approach, resulting in better compression ratio. Although the compression ratios are

Table 4. Results of order 2 bitwise modeling

Application	Buses	Model size [bytes]	Compr. instr. width [bits]	Compr. code size + model [bytes]	Compr. ratio [%]
8×8 2-D DCT	3	360	23.9	982	42.9
	8	726	30.7	1255	39.5
	13	907	29.3	1373	31.8
32-point DCT	4	397	28.9	4142	33.3
	8	715	51.3	3935	39.2
	10	684	51.1	3912	32.8
Viterbi decoding	3	414	26.6	1695	36.7
	6	738	42.1	2117	42.5
	9	945	39.9	2208	33.6
MPEG2 decoding	5	672	28.6	31231	22.8
	6	700	31.6	32332	21.2
	8	1012	35.0	34522	19.6

Table 5. Results of dictionary modeling of whole instructions

Application	Buses	Dict. entries	Dict. size [bytes]	Compr. instr. width [bits]	Compr. code size + dict. [bytes]	Compr. ratio [%]
8×8 2-D DCT	3	189	2079	8	2287	100.0
	8	120	2760	7	2881	90.8
	13	101	3434	7	3545	82.1
32-point DCT	4	900	10800	10	12098	97.1
	8	452	9040	9	9605	95.7
	10	434	10850	9	11387	95.5
Viterbi decoding	3	345	4140	9	4573	99.0
	6	224	4256	8	4518	90.8
	9	205	5330	8	5583	84.9
MPEG2 decoding	5	4353	69648	13	83542	61.1
	6	4005	76095	12	88122	57.8
	8	3630	83490	12	94968	54.0

quite good in this approach, the instruction fetch and decoding become more complicated. Compressed instruction become variable-width, which complicates the instruction fetching. As the instructions are compressed, they need to be decompressed before decoding. For decompression, a state machine is needed to find the codewords from the incoming bit stream as the codewords are of variable width.

3 Dictionary Modeling

In dictionary modeling, all the unique bit strings that occur in the program to be compressed are collected into a dictionary. Each bit string in the original program is then replaced with an index pointing to the dictionary [12]. The indices could be stored with variable-width codewords (such as Huffman codes) but, for simplicity, only constant length codewords were used in this paper. The indices become $\lceil log_2|D|\rceil$ bits wide, where $|D|$ is the number of entries in the dictionary. During program execution fixed-width codewords simplify instruction fetching, decompression, and decoding. Fixed-width indices are fetched from the memory. The indices are then used to fetch the original instructions from the dictionary to be decoded. In addition to reduced code size, dictionary modeling approach may also reduce the power consumption. As the indices are smaller than original instructions, fewer bits are fetched from the memory. Thus, the memory I/O bandwidth between the external program memory and the processor is reduced, resulting in smaller power consumption.

Dictionary modeling was evaluated with the same set of benchmarks and processor configurations as the bitwise modeling scheme. In the evaluations, the program codes were searched for unique instruction words, that were all stored into a dictionary. The instruction words in the program code were then replaced with codewords that point to the dictionary. The results are shown in Table 5, which depicts the number of entries in the dictionary, the size of the dictionary, width of the codewords, total code size (including the dictionary), and the compression ratio.

The results indicate that the achieved compression ratios are poor, except for the MPEG2 decoding application that has compression ratios between 54.0 and 61.1 percent depending on processor configuration. The poor results of this method were quite expected, because TTA instructions are fairly long and they are composed of several move slots that all specify independent data transports. As there are several sources and destinations for each data transport, the probability that two or more instructions would specify exactly the same data transports in exactly the same move slots is fairly small. The most probable case for finding identical instructions is when all move slots specify empty data transports. Actually, most of the repeated instructions found in the benchmarks were this kind of totally empty instructions. Still, only for MPEG2 decoding application there were enough of these totally empty instructions so that a good compression ratio was achieved. MPEG2 decoding application was the biggest and most complex of the benchmark applications. Due to this, the compiler was incapable of scheduling data transports efficiently on the available transport buses, which resulted in large number of totally empty instructions. These instructions were optimized away with the dictionary modeling method.

The experiments demonstrated that entire TTA instructions are not suitable to be considered as elements for dictionary compression, because the probability of finding totally identical instructions is small. A better alternative would be to divide TTA instructions into smaller fields and then search for unique bit patterns inside these fields. The division could be made, e.g., according to move slot and long immediate field boundaries. As the bit patterns to be compared would be smaller, the probability of finding identical bit patterns would increase and, most probably, result in better compression ratio.

4 Conclusions

In this paper, two different compression methods, bitwise modeling and dictionary modeling, were evaluated on transport triggered architectures. The bitwise modeling scheme was found to be fairly efficient to improve the code density. The best results were achieved with order 2 modeling, but due to the complexity of the higher order models, it was concluded that the order 1 modeling would be preferable, as it achieves almost as good results as the order 2 modeling with less complex decompression hardware. Dictionary modeling approach was found to be inefficient when applied to whole TTA instructions. Only MPEG2 decoding application produced reasonable results. Better results can be obtained by splitting the instructions into smaller parts, e.g., according to move slot and long immediate field boundaries. This approach will be evaluated in the future.

References

[1] R. P. Colwell, R. P. Nix, J. J. O’Connel, D. B. Papworth, and P. K. Rodman, “A VLIW architecture for a trace scheduling compiler,” *IEEE Trans. Comput.*, vol. 37, no. 8, pp. 967–679, Aug. 1988.

[2] A. Wolfe and C. Chanin, “Executing compressed programs on an embedded RISC processor,” in *Proc. 25th Annual Symp. on Microarchitecture*, Portland, OR, U.S.A., Dec. 1–4 1992, pp. 81–91.

[3] M. H. Weiss and G. P. Fettweis, “Dynamic codewidth reduction for VLIW instruction set architectures in digital signal processing,” in *Proc. 3rd Int. Workshop on Image and Signal Processing on the Theme of Advances in Computational Intelligence*, Manchester, UK, Nov. 4–7 1996, pp. 517–520.

[4] S. J Nam, I. C. Park, and C. M. Kyung, “Improving dictionary-based code compression in VLIW architectures,” *IEICE Trans. Fundamentals of Electronics, Commun. and Comput. Sciences*, vol. E82-A, no. 11, pp. 2318–2124, Nov. 1999.

[5] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel, “A code compression system based on pipelined interpreters,” *Software - Practice and Experience*, vol. 29, no. 11, pp. 1005–1023, 1999.

[6] Y. Xie, W. Wolf, and H. Lekatsas, “A code decompression architecture for VLIW processors,” in *Proc. 34th Annual Symp. Microarchitecture*, Austin, TX, U.S.A., Dec. 1–5 2001, pp. 66–75.

[7] S. Y. Larin and T. M. Conte, “Compiler-driven cached code compression schemes for embedded ILP processors,” in *Proc. 32nd Annual Symp. Microarchitecture*, Haifa, Israel, Nov. 16–18 1999, pp. 82–92.

[8] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*, John Wiley & Sons, Chichester, UK, 1997.

[9] H. Corporaal and M. Arnold, “Using transport triggered architectures for embedded processor design,” *Integrated Computer-Aided Eng.*, vol. 5, no. 1, pp. 19–38, 1998.

[10] D. A. Huffman, “A method for the construction of minimum redundancy codes,” *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, Sept. 1952.

[11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, “MediaBench: A tool for evaluating and synthesizing multimedia communications systems,” in *Proc. 30th Ann. IEEE/ACM Int. Symp. Microarchitecture*, Research Triangle Park, NC, U.S.A., Dec. 1–3 1997, pp. 330–335.

[12] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufmann Publishers, San Francisco, CA, U.S.A., 1999.