Dictionary-Based Program Compression on Transport Triggered Architectures

Jari Heikkinen, Andrea Cilio, and Jarmo Takala
Tampere University of Technology
P.O.Box 553, 33101 Tampere, Finland
Email: jari.heikkinen@tut.fi

Henk Corporaal
Eindhoven University of Technology
P.O.Box 516, 5600 MB Eindhoven, The Netherlands
Email: h.corporaal@tue.nl

***Abstract*— Program code size has become a critical design constraint of embedded systems. Large program codes require large memories, which increase the size and cost of the chip. Poor code density is a problem especially in parallel architectures where a long instruction word controls the concurrently operating hardware resources. Dictionary compression is one of the most often used compression methods to improve the code density due to its simplicity. In dictionary compression, unique bit patterns, e.g., instructions are stored into a dictionary and replaced in the program code by indices pointing to the dictionary. In this paper, dictionary compression is evaluated on transport triggered architecture, a customizable processor architecture that is particularly suitable for tailoring the hardware resources according to the requirements of the application. Obtained results indicate significant improvements in code density.**

I. Introduction

Very long instruction word (VLIW) architectures have gained considerable popularity in embedded systems, especially in digital signal processing (DSP), due to their modularity and scalability. VLIW architectures exploit the instruction level parallelism (ILP) by executing operations in parallel in concurrently operating functional units (FU). These FUs are controlled by a long instruction word that contains dedicated fields for each FU. This kind of an instruction encoding leads to poor code density [1] and, implies a need for larger memories. This, in turn, increases the total cost of the system.

Poor code density can be improved by compressing the instructions. Several compression methods have been proposed for VLIW architectures. In one of the earliest approaches, no-operations (NOP) were eliminated from the instructions [1]. A "mask" identifier preceding each instruction specified which fields were present in the instruction word. In [2], a dictionary-based compression method was applied to VLIWs. Frequently used instruction words were stored into a dictionary and occurrences of these instructions in the actual program were replaced by codewords. Dictionary-based compression was also applied in [3], where the non-time-critical part of the program was 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 [4], and Huffman encoding, e.g., in [5].

Transport triggered architecture (TTA) is a class of statically programmed instruction-level parallel architectures that reminds VLIW architectures [6]. 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 flavor 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 a MOVE framework, a toolset that provides a semi-automatic design process [7]. Processor configurations yielding the best cost/performance ratio are searched with a design space explorer, which evaluates several different processor configurations in terms of performance, area, and power consumption. The hardware subsystem is used to generate the hardware description of the chosen processor configuration. The software subsystem generates the ILP code for the chosen target processor.

Like VLIW processors, 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, dictionary-based program compression is evaluated on TTA processors, designed for six different bench-

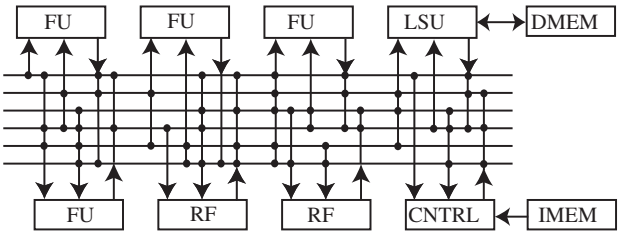


Fig. 1. TTA processor organization. FU: functional unit. RF: register file. LSU: load-store unit. CNTRL: control unit. DMEM: data memory. IMEM: program memory. Dots represent connections between buses and sockets.

marks from multimedia and digital signal processing application domains. The compression is made at three different levels of granularity. First, unique instructions are considered as bit patterns to be stored into a dictionary and replaced by indices. Secondly, the search for unique bit patterns is made inside each move slot field. Finally, unique bit patterns are searched inside source and destination ID fields.

II. Dictionary Compression

Dictionary-based compression methods use the principle of replacing substrings, e.g., words in a text, with a codeword that identifies that substring in a dictionary [8]. The dictionary contains a list of substrings and a codeword for each substring. As the codeword is smaller than the original substring, compression is achieved. During decompression the codeword is used to fetch the original substring from the dictionary.

Program compression using the dictionary approach is based on the fact that instructions in a program code are typically highly repetitive [9]. Furthermore, often only a small part of the instruction set provided by the processor is used. This indicates that the program can be executed with a set of much shorter instructions. This can be achieved by storing all the unique instructions of a program into a dictionary and by replacing the instructions in the program code with indices to the dictionary. Given a program with N unique instructions, the length of the codeword is $\lceil log_2|N| \rceil$ bits. Decompression is fairly straightforward. During program execution the indices, fetched from the program memory, are used to fetch the corresponding instructions from the dictionary for decoding. As the dictionary access introduces an additional delay, an additional decompression stage is typically added to the instruction pipeline.

As the dictionary indices are smaller than original instructions, the size of the program code decreases. On the other hand, the dictionary, typically implemented using ROM inside the processor core, introduces an additional hardware cost. Thus, the compression is profitable only if the dictionary is smaller than the code size reduction achieved in the program code. In addition to reduced code size, dictionary compression may reduce the power consumption of the system. 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.

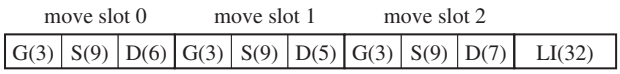


Fig. 2. Structure of instruction word. G: Guard field. S: Source ID field. D: Destination ID field. LI: Long immediate field. (x): x-bit field.

III. Experimental Results

To evaluate the dictionary compression on TTA, it was applied to TTA processors that were designed for six benchmark applications from DSP and multimedia application domains. 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, Viterbi decoding, and edge detection. The multimedia applications, taken from the MediaBench benchmark set [10], implemented MPEG2 decompression and JPEG compression.

Processor configurations for each of the benchmarks were searched with the design space explorer of the MOVE framework [7]. Three configurations, a high-performance, a cost-efficient, and a configuration being a compromise between cost and performance were chosen for each benchmark. The compiler of the MOVE framework was then used to compile the benchmarks for the chosen processor configurations. Table I illustrates the code statistics of the uncompressed benchmark programs on different processor configurations. The configurations are classified according to the number of move buses. Other resources scale correspondingly.

Dictionary compression was applied at three different levels of granularity; at full instruction level, at move slot level, and at ID field level. Memory implementation was taken into account by aligning the compressed instructions, consisting of one or more fixed-width index fields on byte boundaries. The memory was assumed to be word addressable. This way the jump addresses could be left unchanged in the long immediate fields as the order of instructions did not change.

A. Compression at Instruction Level

Dictionary compression was first applied at the level of full instructions. The compiled programs were searched for unique instructions that were stored into a dictionary and replaced with indices to the dictionary. Figure 3 illustrates the obtained compression ratios as proportions of the compressed program code and the dictionary. The compression ratios are on average 0.77. The size of the dictionary is fairly large indicating that not many repetitive instructions were found in the program codes. This is quite expected as TTA instructions are fairly long and are composed of several move slots that specify independent data transports. As there are several sources and destinations for each data transport, the probability of finding identical instructions becomes small and results in large dictionary. Edge detection, MPEG2 decoding and JPEG compression achieved slightly better compression ratios due to the complexity of these applications. The compiler could not fully utilize all the move slots, which resulted in large number of totally empty instructions that were efficiently compressed.

TABLE I

CODE STATISTICS OF BENCHMARK APPLICATIONS

Application	Conf.	Buses	Instr. width [bits]	Number of instr.	Code size [bytes]
DCT8x8	a	13	248	127	3937
	b	8	176	138	3036
	c	3	88	208	2288
DCT32	d	10	200	477	11925
	e	8	160	502	10040
	f	4	104	843	10959
VITERBI	g	9	200	253	6325
	h	6	152	262	4978
	i	3	96	385	4620
EDGE	j	9	200	565	14125
	k	5	128	605	9680
	l	3	96	644	7728
MPEG2DEC	m	8	184	7652	175996
	n	6	152	7925	150575
	o	5	128	8550	136800
CJPEG	p	13	272	12198	414732
	q	6	152	12308	233852
	r	3	96	12909	154908

Dictionary compression limits the programmability of the processor. The modified code can be run only if all of its instructions can be found from the dictionary that is generated for the first version of the code. As TTA instructions are composed of several move slots that all can specify data transports from several sources to several destinations, the number of possible instruction becomes huge. Thus, it is unlikely that all the instructions of the modified code would be found from the dictionary.

B. Compression at Move Slot Level

To improve the compression ratio, the size of the dictionary should be reduced. This can be achieved by dividing instructions into smaller fields and searching for unique bit patterns inside these fields. As the bit patterns are smaller, the probability of finding repetitive bit patterns increases and results in smaller dictionary. One alternative to divide instructions to fields is according to move slot boundaries. Compression can then be performed either vertically or horizontally. In the vertical approach, instructions are divided into parallel streams according to move slot boundaries. The search for unique bit patterns is made inside each stream and a dictionary is generated for each of them. In the horizontal approach, unique bit patterns are searched inside all move slots. A single dictionary storing all the unique bit patterns is generated. In both cases, the long immediate field is considered as a separate stream for which an individual dictionary is generated.

The results of applying dictionary compression at move slot level are shown in Fig. 4. Figure 4(a) depicts the compression ratios of the vertical approach. The compression ratios are on average 0.53. The size of the dictionary has been reduced significantly, which indicates that many repetitive bit patterns were found inside move slots. The drawback is the increased size of the compressed program code due to dividing instruction to smaller fields. Fig. 4(b) shows the compression ratios

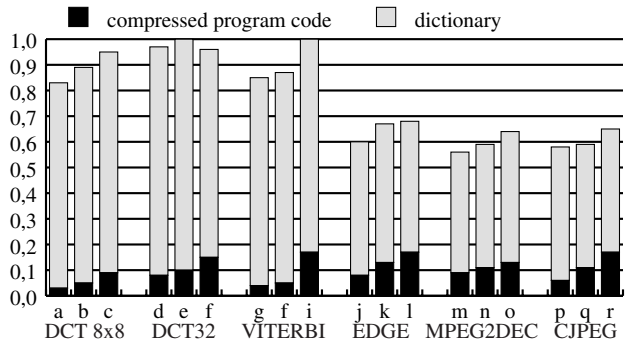


Fig. 3. Results of applying dictionary compression at instruction level

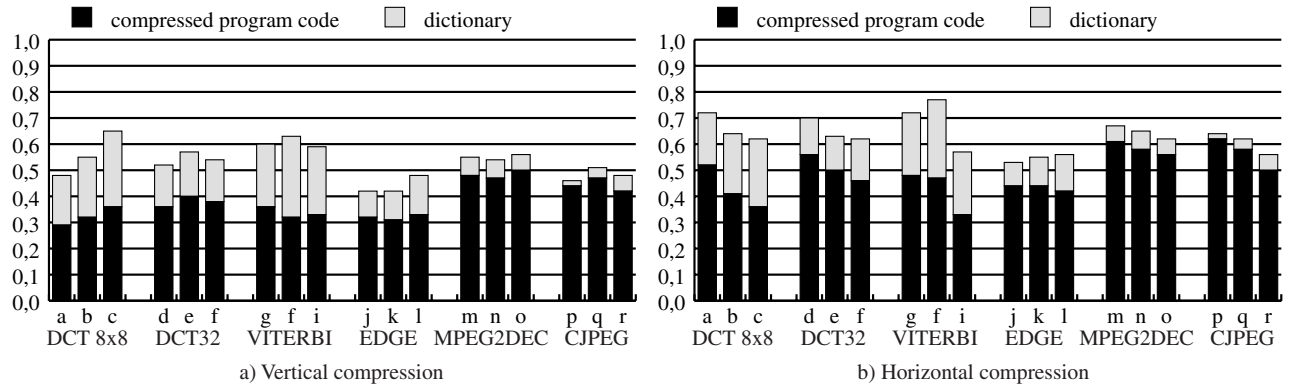
of the horizontal approach. The compression ratios are on average 0.62, which is slightly worse compared to the vertical approach. The dictionaries are almost the same size but the program codes are bigger in the horizontal approach. This is due to using only a single dictionary, which results in wide dictionary index that increases the width of the compressed instruction word. The vertical approach is also simpler and faster to decode as all the move slot dictionaries can be accessed simultaneously and the compressed instructions can be decompressed in parallel. In the horizontal approach, the dictionary needs to be accessed sequentially.

Dividing instructions to smaller fields improves also the programmability. As smaller bit patterns, move slots, are stored into the dictionaries, the number of possible bit patterns is smaller and the probability of finding all the move slot bit patterns of the modified code from the dictionary is better.

C. Compression at ID Field Level

TTA instructions can be divided to even smaller fields to find more repetition in the program code. The move slots can be divided to fields according to guard, source ID, and destination ID field boundaries. Unique bit patterns to be stored into the dictionaries can then be searched inside these fields. Division to smaller fields will also improve the programmability. As the fields are fairly narrow, the number of possible bit patterns is small and most of the possible guards and source and destination IDs will be stored into the dictionaries. Thus, the guards and source and destinations IDs of the modified code will most likely be found from the corresponding dictionaries.

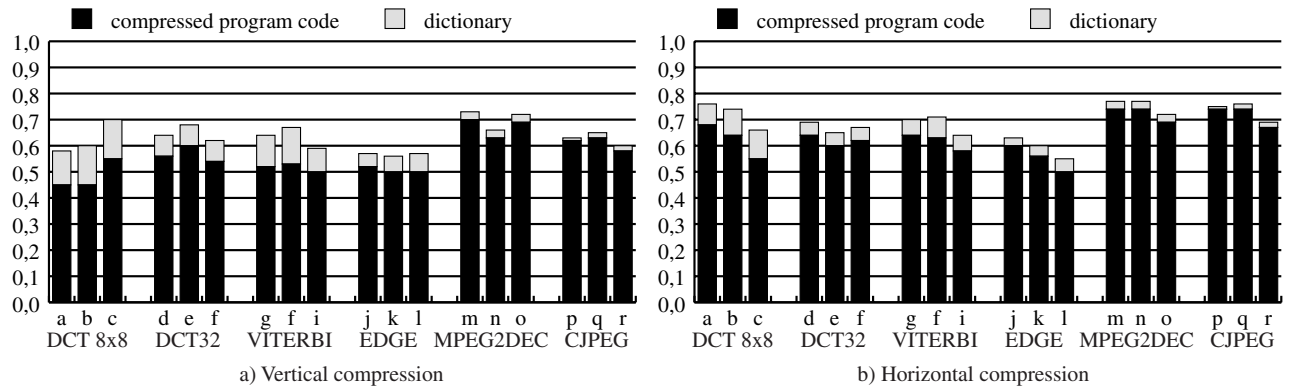
Figure 5 shows the results of applying dictionary compression at ID field level. The source and destination ID fields were compressed both vertically and horizontally. All the guard fields were combined into a single field as they were fairly small. The compression ratios of the vertical approach, depicted in Fig. 5(a), are on average 0.63. The compression ratios of the horizontal approach, illustrated in Fig. 5(b), are on average 0.69, which is again slightly worse compared to the vertical approach. Compared to the results of applying dictionary compression at move slot level, the size of the dictionary has been reduced even further, but the consequence of dividing instructions to even smaller fields is the increase in compressed program code size.



a) Vertical compression

b) Horizontal compression

Fig. 4. Results of applying dictionary compression at move slot level



a) Vertical compression

b) Horizontal compression

Fig. 5. Results of applying dictionary compression at ID field level

IV. Conclusions

In this paper, dictionary-based program compression was evaluated on transport triggered architecture processors that were designed for benchmarks from multimedia and digital signal processing application domains. Dictionary compression was applied to program codes at three different levels of granularity; at full instruction level, at move slot level, and at ID field level. For the last two levels, both vertical and horizontal compression approaches were evaluated.

Best compression ratios were achieved by applying dictionary compression at move slot level using vertical approach. An average compression ratio of 0.53 was achieved. At full instruction level, the program code size was reduced drastically but the dictionary grew large as most of the instructions had to be stored there. At ID field level, plenty of repetition was found resulting in small dictionary, but due to dividing instructions to several fields the compressed instruction word became wide and resulted in increased program code size.

The presented results do not take into account the actual implementation of the decompression circuitry. The size of the dictionary component would decrease as it could be implemented, e.g., using ROM, which is denser than RAM. In the future, the decompression circuitry will be implemented in hardware to evaluate the effect of dictionary compression on actual area, timing and energy consumption of the chip.

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] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

[9] C. Lefurgy and T. Mudge, "Code compression for DSP," EECS Department, University of Michigan, Technical Report CSE-TR-380-98, Nov. 1998.

[10] 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.