Dictionary-Based Program Compression on TTAs: Effects on Area and Power Consumption

Jari Heikkinen 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 result in large memories, which increase the size and cost of the chip. Poor code density is a problem especially in VLIW architectures, where a long instruction word is used to control the concurrently operating hardware resources. In addition, wide instructions increase the memory bandwidth, which may result in increased power consumption. Dictionary compression is one of the most often used compression methods to improve the code density due to its simplicity. In this paper, dictionary-based program compression is applied on transport triggered architecture, a customizable processor architecture that is particularly suitable for tailoring the hardware resources according to the requirements of the application. The effects on area and power consumption were measured. We observed that at best, the area of the instruction memory and the fetch and decode logic could be reduced by 87%, and power consumption by 80%, correspondingly.**

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. Furthermore, fetching wide instructions from the instruction memory consumes significant amount of power.

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. A similar approach was presented in [2], where NOPs were eliminated by using multiple instruction formats, or instruction templates, that provided operation slots for only a subset of all the functional units. In [3], 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 [4], where the non-time-critical part of the program was compressed using *superinstructions* that correspond to frequently used instruction patterns. In [5], a dictionary-based compression method was presented that minimized the number of dictionary entries. The dictionary entries were chosen so that all the instructions of the program code were at most a specified maximum Hamming distance from a dictionary entry. Bit toggling information was used to restore the original instruction. Entropy encoding is another fairly commonly used approach to improve the code density. It 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 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 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 [9]. 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 is used to simplify decoding. Minimal instruction encoding leads to long instruction words. 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

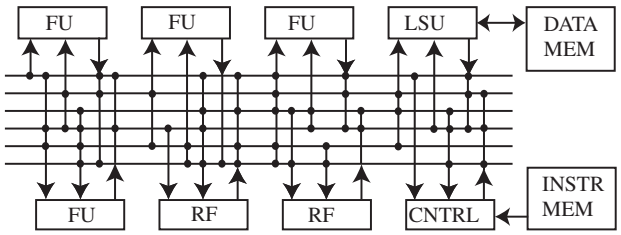


Fig. 1. Example of TTA processor organization. FU: functional unit. RF: register file. LSU: load-store unit. CNTRL: control unit. Dots represent connections between buses and sockets.

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 that waste instruction bits.

In this paper, dictionary-based program compression is applied on two TTA processors that have been designed for four benchmark applications from the digital signal processing application domain. Both processors have been designed to be capable of executing all four benchmark applications. Dictionary compression is applied at three different levels of granularity: at full instruction level, at move slot level, and at ID field level. The TTA processors are implemented in hardware to obtain statistics on the effects of dictionary-based program compression on area and power consumption.

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 [10]. 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.

Dictionary-based program compression is based on the fact that instructions in a program code are typically highly repetitive [11]. 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 processor's pipeline.

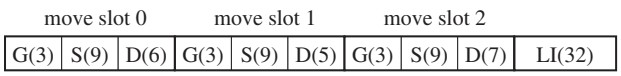


Fig. 2. Structure of instruction word. G: Guard field. S: Source ID field. D: Destination ID field. LI: Long immediate field. (x): x-bit field.

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. Hence, 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 usually reduces 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 instruction memory and the processor is reduced, resulting in smaller power consumption.

III. Evaluation Methodology

Dictionary-based program compression was evaluated on TTA by applying it to four digital signal processing benchmark applications that were compiled on two TTA processor configurations. The benchmarks realized two versions of the discrete cosine transform (DCT), two-dimensional (2-D) 8×8 DCT and 1-D 32-point DCT, 1024-point Radix-4 fast Fourier transform, and Viterbi decoding. The two TTA processors, both capable of executing all four benchmark applications, were designed with the design space explorer of the MOVE framework [9]. Two processor configurations, a cost-efficient configuration and a configuration being a compromize between cost and performance were chosen. The resources, i.e., buses, functional units, and registers of these configurations are described in Table I. The statistics of the uncompressed benchmarks compiled on the two TTA processors are illustrated in Table II.

Dictionary compression was then applied to the program codes. Compression was applied at three different levels of granularity; at instruction level, at move slot level, and at ID field level. At instruction level, the program codes were searched for unique instructions that were stored into a dictionary and replaced with indices pointing to the entries in the dictionary. At move slot level, instructions were divided into parallel streams according to move slot boundaries and the search for unique bit patterns to be stored into dictionaries was

TABLE I
Hardware resources of the two TTA processor configurations

Conf.	Buses	Functional units	Registers	Instr. width [bits]
A	5	1 multiplier, 1 load-store, 1 ALU, 1 compare, 1 shifter, 1 logic, 1 sign extend	19	128
B	8	1 multiplier, 2 load-stores, 2 ALUs, 1 compare, 3 shifters, 1 logic, 2 sign extend	52	192

TABLE II

CODE STATISTICS OF THE BENCHMARKS ON THE TWO TTA PROCESSORS

Application	*(x)*	Conf.	Instr. count	Code size [bytes]	Clock cycles
32-point DCT	1	A	484	7744	466
	2	B	441	10584	423
2-D 8×8 DCT	3	A	163	2608	22959
	4	B	137	3288	19455
1024-point Radix-4 FFT	5	A	315	5040	282547
	6	B	149	3576	123667
Viterbi decoding	7	A	367	5872	2710738
	8	B	253	6072	1568227

made inside these streams. Each move slot was then replaced by an index to a corresponding dictionary. At ID field level, instructions were divided to even smaller streams according to guard, source ID, and destination ID field boundaries. The search for unique bit patterns to be stored into dictionaries and replaced with indices was then made inside these streams.

The structural hardware descriptions (VHDL) of the two TTA processor cores were then created using the hardware subsystem of the MOVE framework. To support execution of compressed code, the control unit, which is responsible for executing the first two stages of the three-stage TTA pipeline [8], had to be modified. The data width of the instruction fetch stage had to be reduced down to the width of the compressed instructions. In addition, a decompressor had to be added to decompress compressed instructions back to the original format using the dictionary. The decompressor was implemented as an additional pipeline stage in between the instruction fetch and decode stages. The decode stage did not need any modifications. Figure 3 illustrates the organization of the modified control unit.

As the decompressor is added as an extra pipeline stage, the branch latency is increased due to the increased pipeline depth. This results in increased cycle count. For the two TTA processor configurations running the four benchmark applications, the cycle count was increased on average 7.3% on configuration A and 3.9% on configuration B. Another possibility would have been to include the decompressor in the instruction decode stage, but that would have increased the clock cycle time, which was not desired.

The decompressor was implemented separately for each of the benchmark cases on both processor configurations. Two alternatives of implementing the dictionary of the decompressor were evaluated. In the first case, the dictionary was implemented using RAM. By implementing the dictionary using RAM, the programmability is maintained. The dictionary is loaded before the execution starts. In the second case, the dictionary was implemented using standard cells. This limits the programmability. The program can be modified only if the bit patterns, i.e., instructions, move slots, or ID fields, depending on the granularity on which the compression is applied, can be found from the original, fixed dictionary. The advantage of using standard cells to implement the dictionary is its smaller area and power consumption.

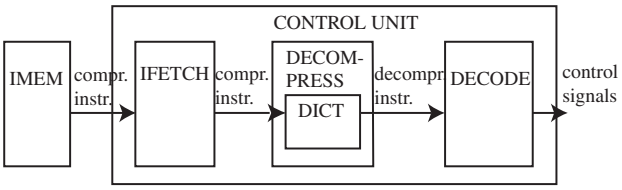


Fig. 3. Structure of the modified control unit. IMEM: instruction memory, IFETCH: instruction fetch stage: DECOMPRESS: decompress stage, DECODE: decode stage, DICT: dictionary.

Data and program memories were chosen so that they could be used for all the four benchmark applications. A single-ported, 32kB data memory (with 32-bit data width) was chosen for the processor configuration A. Configuration B required a 32kB dual-ported data memory as it had two load-store units that can access the data memory simultaneously. For the instruction memory, a 512 word, 128-bit wide memory (8kB) was chosen for the configuration A. Configuration B required a 512 word, 192-bit wide memory (12kB), which was constructed out of six 32-bit wide, 512 word memory blocks. For the compressed cases, the word width of the memory was adapted to the width of the compressed instruction word but the number of memory words remained the same.

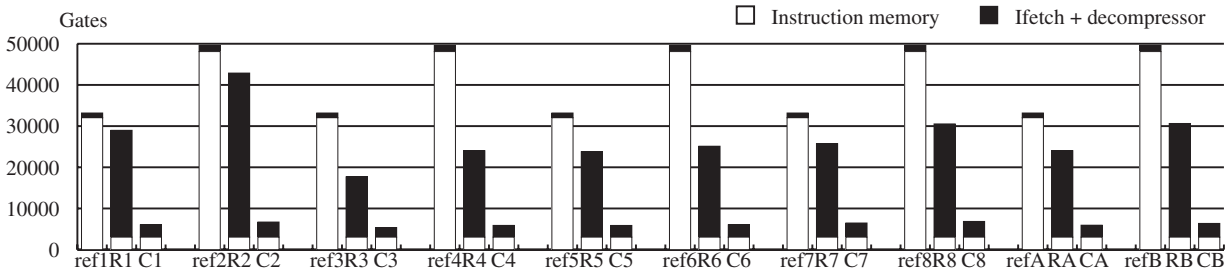
The designed processors were then synthesized using a 0.13 μm CMOS standard cell technology using the Synopsys Design Vision version 2003.06 for Linux platforms. The processors were synthesized using a timing constraint of 200MHz. Data and program memories and RAM dictionaries were included as presynthesized macro cells. The switching activities for power analysis were obtained from the gate-level simulation run on ModelSim. A VHDL VITAL [12] timing model was used to obtain the switching activities for the memory macro cells. Statistics were obtained in terms of area and power consumption.

Table III shows the areas of the processors. The size of the instruction memory is comparable to the area of the processor core. In configuration A they are equal, in configuration B the area of the instruction memory is 60% of the area of the processor core. Data memory turned out to consume most of the area. This was due to storing all the input and output data into the data memory as there was no I/O support. The average power consumptions, when running the four benchmarks, are on average 41 mW for the configuration A and 71 mW for the configuration B. Instruction memory consumes on average 16.1 mW on configuration A and 25.7 mW on configuration B, i.e., close to 40% of the total power consumption.

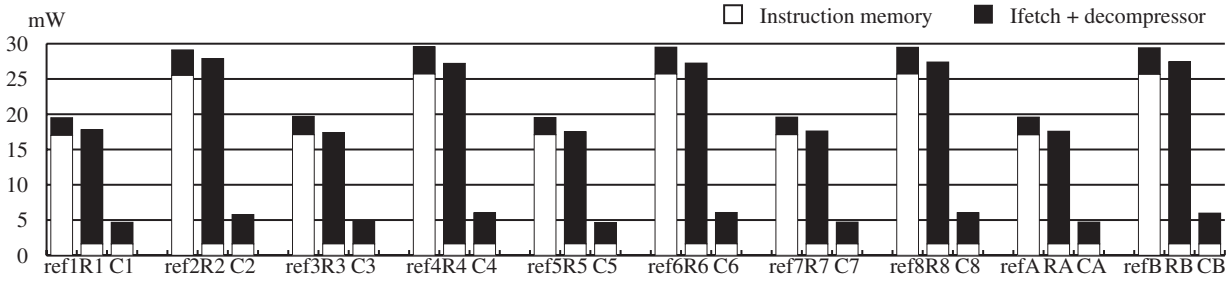
TABLE III

AREAS OF THE REFERENCE DESIGNS

Conf.	Data mem. [kgates]	Instr. mem. [kgates]	Proc. core [kgates]	Total [kgates]
A	104	32	31	167
B	410	48	80	538



(a) Areas of the instruction memory and the instruction fetch and decompress stages of the control unit



(b) Power consumptions of the instruction memory and the instruction fetch and decompress stages of the control unit

Fig. 4. Results of applying dictionary compression at instruction level for the two TTA processors. The graph illustrates the areas and power consumptions of the uncompressed case (ref*(x)*), compressed case with the dictionary implemented using RAM (R*(x)*), and compressed case with the dictionary implemented using standard cells (C*(x)*). *(x)* refers to a benchmark application compiled on either configuration A or B, as listed in Table II. The last two *(x)*'s, A and B, refer to the averages of all the benchmark applications on the two processor configurations.

IV. Experimental Results

A. Compression at Instruction Level

The results of applying dictionary compression at instruction level on the two TTA processor configurations are illustrated in Fig. 4. Figure 4(a) illustrates the areas of the uncompressed case (ref*(x)*) and of the two decompressor implementation approaches, first using RAM (R*(x)*) and second using standard cells (C*(x)*) to implement the dictionary. Figure 4(b) illustrates the power consumptions of these three cases. Results are presented for the instruction memory and for the instruction fetch and decompress stages of the control unit, as these are the only parts of the design whose areas and power consumptions are affected by the dictionary compression.

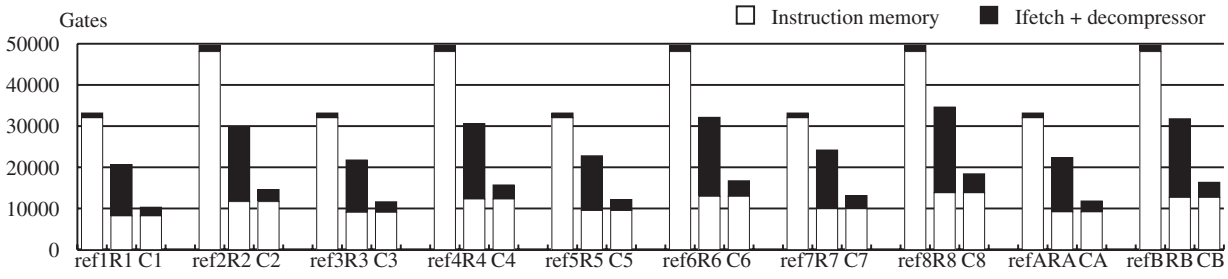
When the dictionary is implemented using RAM, the combined area of the instruction memory and control unit's instruction fetch and decompress stages is reduced on average 28 % on configuration A and 38 % on configuration B. The results indicate that few repetitive instructions are found in the program codes, i.e., most of the instructions are stored into the dictionary, which becomes large and increases the size of the decompressor. The size of the dictionary can be reduced by implementing it using standard cells and letting the synthesis tool to minimize the logic. By implementing the dictionary using standard cells, the area is reduced on average 82% on configuration A, and 87% on configuration B. However, this approach limits the programmability of the processor as the dictionary becomes fixed. In order to modify the program, all the instructions of the modified version of the code need to be found from the original dictionary. As TTA instructions are long and are composed of several smaller fields, the number of possible combinations is huge and the probability that all the instructions of the modified code can be found from the original dictionary is small.

The power consumption behaves similarly to area. When the dictionary is implemented using RAM, the combined power consumption of the instruction memory and the instruction fetch and decompress stages is reduced on average only 10% on configuration A and 7% on configuration B. This is due to having a large RAM dictionary in addition to the RAM instruction memory. With the dictionary implemented using standard cells, the power consumption is reduced on average 76% on configuration A and 80% on configuration B.

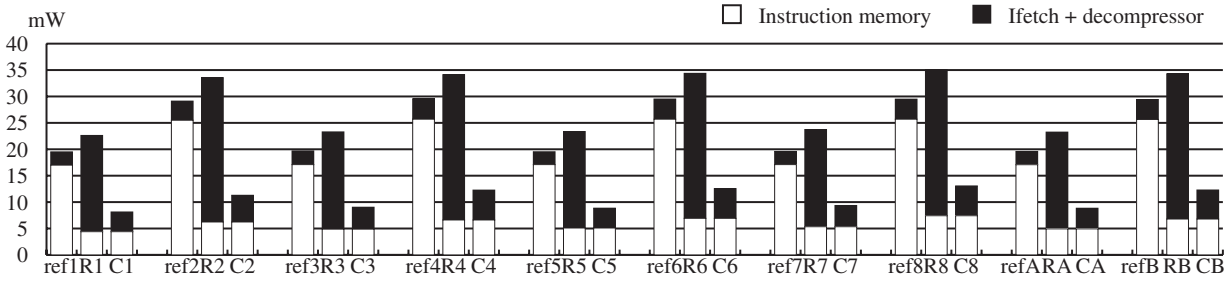
Thus, there is a trade-off between the programmability and the reduction in area and power consumption. If the programmability needs to be maintained, the dictionary needs to be implemented using RAM. This results in poor reduction in area and power consumption. The area and power consumption can be reduced significantly by implementing the dictionary using standard cells at the cost of losing the programmability.

B. Compression at Move Slot Level

Figure 5 illustrates the results of applying dictionary compression at move slot level, i.e., dividing instruction to parallel streams according to move slot boundaries and searching for repetitive bit patterns inside these fields. Figure 5(a) illustrates the areas of the three evaluated cases. With RAM dictionary, the area is reduced on average 33% on configuration A and 36% on configuration B. Plenty of repetitive bit patterns are found inside the parallel move slot streams resulting in



(a) Areas of the instruction memory and the instruction fetch and decompress stages of the control unit



(b) Power consumptions of the instruction memory and the instruction fetch and decompress stages of the control unit

Fig. 5. Results of applying dictionary compression at move slot level for the two TTA processors. The graph illustrates the areas and power consumptions of the uncompressed case (ref*(x)*), compressed case with the dictionary implemented using RAM (R*(x)*), and compressed case with the dictionary implemented using standard cells (C*(x)*). *(x)* refers to a benchmark application compiled on either configuration A or B, as listed in Table II. The last two *(x)*'s, A and B, refer to the averages of all the benchmark applications on the two processor configurations.

fairly small dictionary. However, as the compressed instruction is composed of several dictionary indices, it becomes wide and results in larger instruction memory. With the dictionary implemented using standard cells, the area is reduced on average 64% on configuration A and 67% on configuration B. The dictionary is again fixed and the programmability is limited. However, there are better possibilities to modify the program as the bit patterns stored into the dictionaries are smaller. The recompiled code can be executed on the processor with the fixed dictionary if all the move slots of the recompiled code can be found from the corresponding dictionaries.

In terms of power consumption, depicted in Fig. 5(b), the approach, where the dictionary is implemented using RAM, results in increased power consumption compared to the reference case without any compression. This is due to having several small RAM dictionaries inside the processor core that consume power. On average, the combined power consumption of the program memory and the instruction fetch and decompress stages of the control unit is increased 19% on configuration A and 17% on configuration B. However, when implementing the dictionary using standard cells, the power consumptions is reduced on average 55% on configuration A and 58% on configuration B.

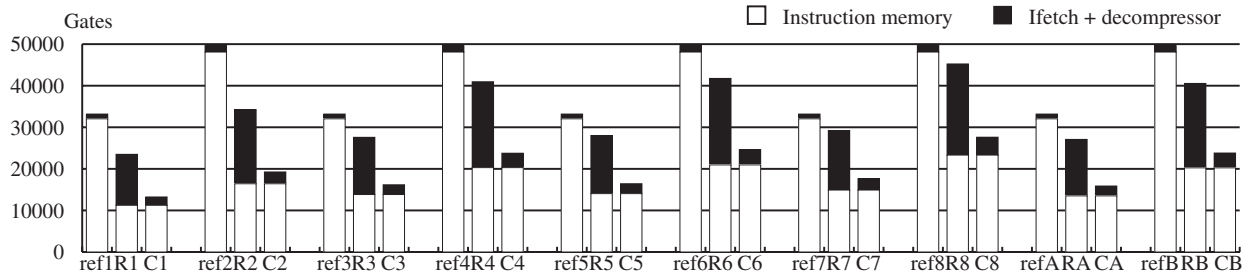
Hence, when standard cells are used to implement the dictionaries, the area and power consumption are reduced efficiently. Furthermore, the programmability is maintained better than at instruction level as smaller bit patterns are stored into the dictionaries, making compression at move slot level a feasible compromize between the reduction in area and power consumption and the programmability.

C. Compression at ID Field Level

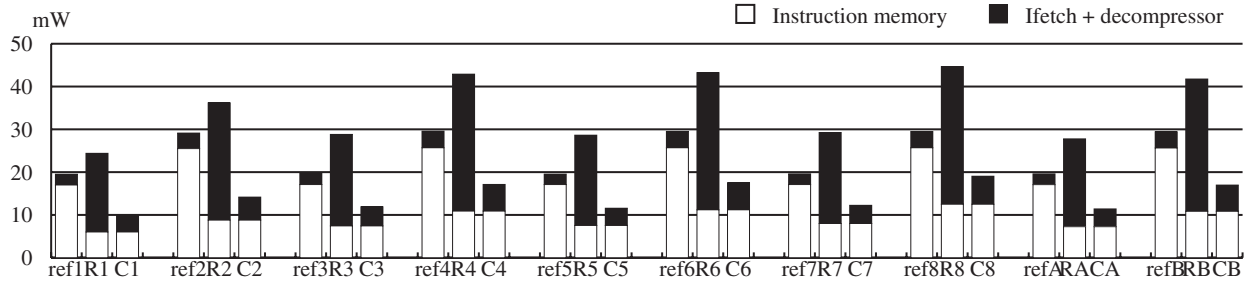
The results of applying dictionary compression at the level of ID fields, i.e., dividing all the move slots to fields according to guard and source and destination ID fields, is depicted in Fig. 6. Figure 6(a) shows the areas and Fig. 6(b) the power consumptions of the instruction memory and control unit's instruction fetch and decompress stages for the three evaluated cases. When implementing the dictionary using RAM, the area is reduced on average 18%on both configurations, and the power consumption is increased on average 42% on both configurations. The size of the dictionary is smaller than at move slot level, but due to several dictionaries, the compressed instruction becomes wider and increases the size of the compressed instruction memory. Similarly to applying dictionary compression at move slot level, the power consumption increases due to having several RAM dictionaries. When the dictionary is implemented using standard cells, the area is reduced on average 52% on both configurations, and the power consumption on average 42% on both configurations. The programmability is maintained even better than at move slot level as the bit patterns stored into the dictionaries are smaller and there are better possibilities to find all the bit patterns of the modified code from the original dictionary to execute the program correctly.

V. Conclusions

In this paper, the effects of dictionary-based program compression on the area and power consumption of transport triggered architecture processors were evaluated. Dictionary-based program compression was applied on two TTA processors that



(a) Areas of the instruction memory and the instruction fetch and decompress stages of the control unit



(b) Power consumptions of the instruction memory and the instruction fetch and decompress stages of the control unit

Fig. 6. Results of applying dictionary compression at ID field level for the two TTA processors. The graph illustrates the areas and power consumptions of the uncompressed case (ref*(x)*), compressed case with the dictionary implemented using RAM (R*(x)*), and compressed case with the dictionary implemented using standard cells (C*(x)*). *(x)* refers to a benchmark application compiled on either configuration A or B, as listed in Table II. The last two *(x)*'s, A and B, refer to the averages of all the benchmark applications on the two processor configurations.

were designed for benchmarks from digital signal processing application domain. Dictionary compression was applied at three different levels of granularity; at instruction, move slot, and ID field levels. Furthermore, two alternatives to implement the dictionary were experimented; first using RAM and second using standard cells to implement the dictionary.

Table IV summarizes the obtained results. The best reduction in area and power consumption can be obtained at instruction level using standard cells to implement the dictionary. However, the programmability of this approach is extremely limited. The programmability can be maintained if the dictionary is implemented using RAM, but the results are significantly worse compared to using standard cells to implement the dictionary. The best compromize between the reduction in area and power consumption and the programmability can be achieved by applying dictionary compression at move slot level and implementing the dictionary using standard cells. The bit patterns stored into the dictionary are fairly small, which increases the probability to find correct bit patterns also for the modified code to execute it correctly.

TABLE IV

AVERAGE RESULTS OF THE THREE EVALUATED APPROACHES

Compression granularity	RAM solution		Standard cell solution	
	Area	Power	Area	Power
Instruction level	-28/-38%	-10/-7%	-82/-87%	-76/-80%
Move slot level	-33/-36%	+19/+17%	-64/-67%	-55/-58%
ID field level	-18/-18%	+42/+42%	-52/-52%	-42/-42%

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. Aditya, B. R. Rau, and R. C. Johnson, "Automatic design of VLIW and EPIC instruction formats," Hewlett-Packard Laboratories, Tech. Rep. HPL-1999-94, 2000.

[3] 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.

[4] 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.

[5] M. Ros and P. Sutton, "A Hamming distance based VLIW/EPIC code compression technique," in *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, Washington, DC, U.S.A., Sept. 22–25 2004, pp. 132–139.

[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*. Chichester, UK: John Wiley & Sons, 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] 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.

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

[12] IEEE Standards Board, *IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification*, IEEE Std 1076.4-1995, 1996.