# Code Compression on Transport Triggered Architectures

Jari Heikkinen, Jarmo Takala, and Jaakko Sertamo
*Institute of Digital and Computer Systems*
*Tampere University of Technology*
*P.O.B. 553, 33101 Tampere, Finland*

**Abstract**: Entropy coding is an efficient method for compressing information and it can be used to improve code density of processor architectures. However, optimal compression requires information of the probability of instructions, which, in general, is not available when processor is designed. However, when designing customizable processors, which are tailored for a given application, such information can be obtained at design time. In this paper, entropy coding is used to improve code density of application-specific programmable processors based on transport triggering paradigm. Three DSP applications are used as benchmark applications and Huffman coding is applied to compress the instructions of a processor architecture tailored for each benchmark.

**Key words**: code compression, Huffman coding, customizable processor architecture, transport triggered architecture, discrete cosine transform, Viterbi decoding

## 1. INTRODUCTION

The current trend in software development for embedded systems in the field of digital signal processing (DSP) is to move towards high-level language (HLL) programming and customizable architectures [1]. The reason behind this is the gap between the productivity of designers and increased complexity of DSP applications. A popular means to improve performance in DSP applications is to exploit the instruction-level parallelism and especially very long instruction word (VLIW) architectures have currently gained considerable popularity. In such architectures, a long instruction word is used to control the operation of parallel function units (FU). The instruction word consists of dedicated fields for each of the FUs specifying the operations and operands for the FUs.

Due to the several control fields, the code density of VLIW architectures has been low [2]. However, the code density can by improved by utilizing code compression methods to the instructions. Programs are stored into the memory in compressed format, then decompressed in the instruction decoding phase, and finally executed in the processor. Such systems are called compressed-code architectures [3].

In [4], Huffman coding [5] is used to compress general-purpose programs taken from SPEC benchmark suite for six different architectures including MIPS R4000, Sun 68020, and Sun SPARC. Results indicate compression ratios between 0.77 and 0.87 for these architectures. In the previous approach, the coding parameters are constant for each application, thus optimal compression rate cannot be obtained. Better compression ratios can be achieved when the coding is optimized for each application, which implies that an application-specific instruction decoding hardware is needed. Such an approach can be applied when designing customizable processor architectures, where the hardware resources are tailored according to the application. In such a case, the probability of each instruction can be measured during the architecture design and optimal entropy coding can be applied resulting in more efficient compression rate.

In this paper, entropy coding is used to improve the code density of customizable processor architectures. Three DSP applications, for which tailored processor configurations are designed, are used as benchmark applications. Huffman coding is then applied to compress the instructions of these three processor designs to improve code density.

## 2. TRANSPORT TRIGGERED ARCHITECTURE

VLIW architectures lend themselves for computationally intensive DSP applications. They are modular; the number of function units (FU) can be easily increased. There are even VLIW architectures that support customized, application-specific function units. This may, however, restrict the flexibility, e.g., in TriMedia [6], support for multi-operand instructions, i.e., multi-operand FU, reserves several instruction fields from the VLIW instruction. VLIW architectures have been criticized for their requirements on read/write ports in the register file [7]. In order to alleviate this problem, clustered approach has been suggested, i.e., the register file is partitioned as illustrated in Fig. 1. In addition, the complexity of the bypassing network and the register file is high since the bypassing network must support bypassing of operands from all the FU outputs to all the FU inputs.

The bypass complexity of VLIW architecture can be reduced by making the bypass visible at the architectural level. Furthermore, the bypass complexity can be reduced by reducing the number of read and write connections and by reducing the number of bypass buses. This implies that besides the operations also the operand transfers (transports) need to be scheduled at compile-time. The bypass transports become visible at the architectural level implying that operations can be hidden. In this model, the data transports trigger the operations implicitly, thus the traditional

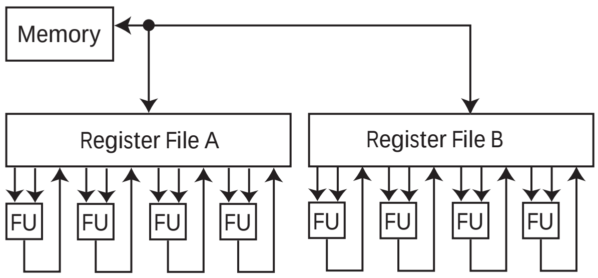


Figure 1. Principal block diagram of clustered VLIW. FU: Function unit.

operation triggered programming paradigm is mirrored, hence the name transport triggered architecture (TTA) [8]. In TTA programming model, program specifies only the data transports to be performed by the interconnection network. Therefore, only one type of operation is supported: move operation, which performs a data transport from a source to a destination. The number of move operations per instruction is equal to the number of simultaneous transports supported by the interconnection network.

A TTA processor consists of a set of FUs and register files containing general-purpose registers connected by interconnection buses as illustrated in Fig. 2. Connections to buses are established through sockets; an input socket feeds operands from the buses into the FUs and an output socket places the FU results into the correct bus. The TTA concept provides flexibility in form of modularity; functional units with standard interface are used as basic building blocks.

MOVE framework is a design environment containing a set of software tools for designing application-specific instruction set processors using the TTA paradigm [9]. The design flow consists of three principal components as illustrated in Fig. 3. The design space explorer searches for a processor configuration, which yields the best cost/performance ratio for a given application. The explorer optimizes both the resources and connectivity. The hardware subsystem generates structural hardware description of the selected processor configuration and produces statistics of timing, area, and power consumption. The software subsystem generates instruction-level parallel code for the selected processor configuration. It provides statistics, e.g., on cycle count, instruction count, and hardware resource utilization. Processors designed with the MOVE tools are called MOVE architectures.

MOVE design environment supports all the trends in DSP processors mentioned previously: HLL programming, customization, and instruction-level parallelism. Designer can tailor and optimize the resources of the programmable architecture and the HLL compiler adapts to these modifications. This approach allows processor core to be customized according to the requirements of the application in question.

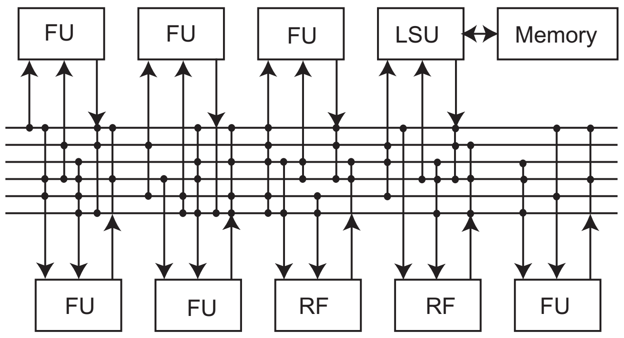


Figure 2. Principal block diagram of TTA. FU: function unit. RF: Register file. LSU: Load-store unit. Dots represent socket connections.

One drawback of MOVE architecture is its poor code density. A long instruction word, containing a dedicated field for each bus, together with a large number of instructions results in a large program code. However, MOVE architecture offers a possibility for efficient code compression. Since the processor is tailored according to the requirements of the application, application-specific compression methods can be utilized.

## 3. MOVE INSTRUCTION FORMAT

The instruction format of a MOVE architecture consists of specific fields, slots, for all the data transports, namely moves. Since one bus can perform a single move per cycle, each instruction contains as many move slots as there are buses. The instruction shown in Fig. 4 contains two move slots and a reserved field for an optional long immediate.

Each move slot contains identification code (ID) for a guard, destination, and source identification. The guard ID is used by a guard unit to control the execution or to squash a move operation. This provides means to realize conditional execution. The source and destination IDs are used to select the source and destination sockets of the data transport, respectively. Both the destination and source IDs contain a socket address and an optional operation code, opcode. This is used to select the operation to be performed at the FU. When the opcode is not used, destination and source IDs contain only the socket addresses.

MOVE architectures support also immediate extensions. Long immediate operands can be specified with dedicated fields at the end of instruction word. Short immediates are supported by storing the data into source ID field. The most significant bit of the source ID field is used as a flag to determine whether the bits of the source ID field are used to specify a source address and an opcode or a short immediate value.

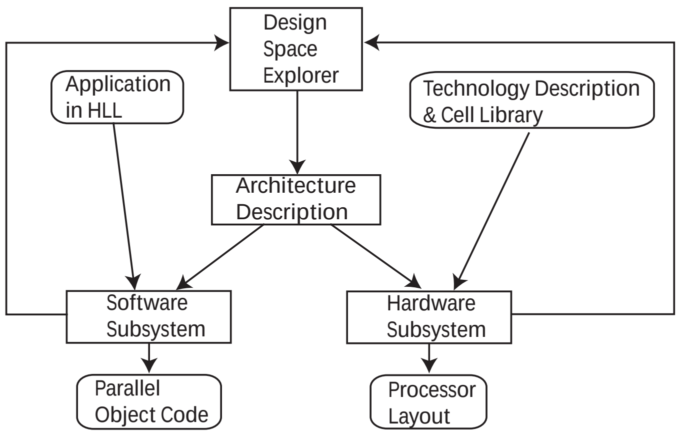


Figure 3. Principal design flow in MOVE framework.

In general, 20 bits are allocated for each move slot. The width of the guard ID field is determined by the number of Boolean registers and compare units, since the outputs of compare units with Boolean registers can be used as guard definitions. This allows compare operations to be used immediately and there is no need to transfer them to Boolean register file. The width of guard ID field is typically less than four bits. The width of the destination ID field is determined by the number of destination connections on the bus. In general, destination ID field is up to seven bits. The width of the source ID field is determined by the number of source connections or the size of the short immediate. In most cases, the size of the short immediate, typically eight bits, determines the source ID width. Taking the immediate flag bit into account, the width of the source ID field is in general nine bits.

The width of the entire MOVE instruction depends on the number of move slots and dedicated fields for long immediates. In general, MOVE architectures contain five to 10 buses and one or two long immediate fields. Therefore, the instruction width varies between 130 and 260 bits.

## 4. ENTROPY CODING ON MOVE INSTRUCTIONS

Entropy coding is based on the fact that if *b*-bit symbols used to describe information are not uniformly distributed, there exists a code, which uses less than *b* bits per symbol. According to this principle, compression can be achieved when small-length codes are assigned to the most probable symbols. This implies that the coding results in a variable-length code.

One of the most popular entropy coding techniques is Huffman coding where symbols are arranged in decreasing order according to their

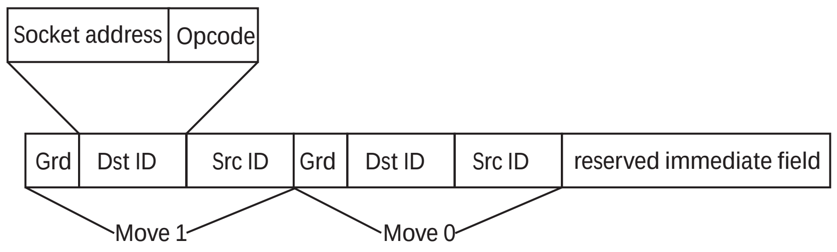


Figure 4. Principal organization of MOVE instruction.

probabilities. After the symbols are arranged, two nodes with smallest probabilities are merged forming a new node with a probability of the sum of the merged nodes. '0' and '1' are assigned to the branches merging into these two nodes. This process is repeated as long as there is only one node left. The compressed code for each symbol is then obtained by reading from the root node to each leaf node and obtaining the binary code from the branches along the path.[10]

In order to improve code density of MOVE architectures, entropy coding can be applied by interpreting the move slots in the instruction word as symbols. However, this would result in symbol explosion and, therefore, we have considered the fields inside move slot as symbols. Source and destination IDs are coded separately since, on each bus, data can be moved from each source to all connected destinations. The guard ID is not coded since, in most cases, its width is small. The guard belongs also to the critical path since guard value determines whether the operand transfer on bus should be squashed or not. Therefore, the guard must be detected early in the instruction cycle. Coding of the guard ID field would require decoding of the field, which would lengthen the critical path even more resulting in lower clock frequency.

The probabilities of each source and destination on each bus are obtained from the parallel code, which is generated by the MOVE compiler. Huffman coding is then applied and new variable-length codes for each source ID and destination ID symbol on each bus are obtained.

## 5. BENCHMARKS

In order to evaluate the code density, we have used two applications as examples: discrete cosine transform (DCT) and Viterbi algorithm. DCT is widely used in video, image, and audio coding. Viterbi algorithm is widely used in many decoding and estimation applications in communications and signal processing since it is computationally efficient.

The first benchmark is an 8x8 DCT realized with row-column approach, i.e., the entire two-dimensional (2-D) transform is computed with the aid of 1-D transforms. Here the constant geometry algorithm proposed in [11] has been used. Constant geometry algorithms being regular and modular allow better exploitation of the inherent parallelism.

The second benchmark is a 32-point DCT, where DCT algorithm described in [12] is used. The created C-code contains five functions, one for each processing column of the signal flow graph of the algorithm. Each processing column is written totally unrolled, i.e., no iterations are used. This way the MOVE compiler was able to detect and exploit the inherent parallelism of the algorithm. On the other hand, this sort of code results in larger program code.

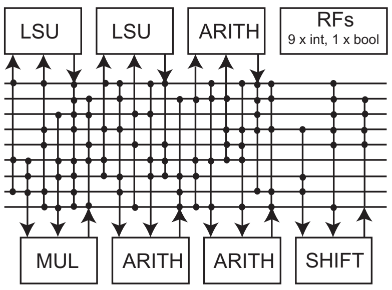
Both the DCT applications were described in C-language using fractional data representation, i.e., fixed-point representation where the number range is normalized. Fractional representation is often used in DSP realization and it represents challenges for C compilers because ANSI C does not contain predefined data type for fractional representation.

The third benchmark task was Viterbi decoding, which, as the DCT application, was written in C-language. This algorithm contains more complex control flow; conditional statements are also needed.

These three benchmarks represent different kinds of DSP applications. The 32-point DCT code has been tuned for fast execution and, therefore, it requires large program code, while the 2-D DCT code is a compromise between performance and code size. Viterbi application is control dominated, more complex application resulting in a long execution time and large program code.

## 6. RESULTS

The design space explorer of the MOVE framework was first used to find an area-efficient architecture for each of the three applications. This was done by first performing resource optimization for an oversized and fully connected architecture configuration to find optimal resources for each of the applications. Next, the unnecessary connections of the obtained architecture configurations were removed by running connectivity optimization. Optimized architecture configurations for 32-point DCT and Viterbi algorithm are shown in Fig. 5. After obtaining the customized architecture configurations, benchmark applications were compiled and the symbol probabilities were calculated. Finally, Huffman coding was applied and variable-length codes were assigned for each symbol.



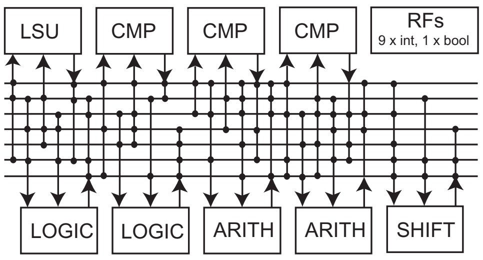


Figure 5. Optimized architectures for a) 32-point DCT and b) Viterbi decoding. LSU: load-store unit. ARITH: arithmetic unit. MUL: multiplier. CMP: compare unit. SHIFT: shifter. LOGIC: logic unit. RFs: register files.

Uncompressed code sizes were calculated by multiplying the number of instructions with the instruction size, which are presented in Table 1. Compressed code size was obtained by first determining the number of bits each new source ID and destination ID code saves compared to the original fixed-length code. Then the total number of bits saved was obtained by multiplying the number of bits saved on each source and destination IDs with the number of their occurrences. Finally, the compressed code size was obtained by subtracting the number of bits saved from the uncompressed code size.

Table 1. Instruction widths for benchmark applications.

| Application | Number of instructions | Instruction size [bits] |
|---|---|---|
| 8x8 2-D DCT | 237 | 155 |
| 32-point DCT | 552 | 181 |
| Viterbi | 1076 | 172 |

By utilizing Huffman coding on MOVE instructions, notable reduction in code size is obtained in all benchmark tasks as can be seen in Fig. 6. Compression ratios for the three benchmarks are given in Table 2. Compression ratios of the two DCT tasks are the same. The compression ratio of Viterbi decoding is slightly worse. This was caused by the problems in MOVE compiler, since it was not able to use the least connected architecture configuration obtained from the design space explorer. Thus, an architecture configuration with more connections had to be chosen. This resulted in larger number of symbols to be coded and thus lower code compression ratio. If the MOVE compiler had been able to use the most optimal architecture configuration for Viterbi, the compression ratio would have been the same as for both DCT benchmarks. Compared to the compression ratios of fixed-resource processor architectures in [4], results indicate a notable improvement in code density.

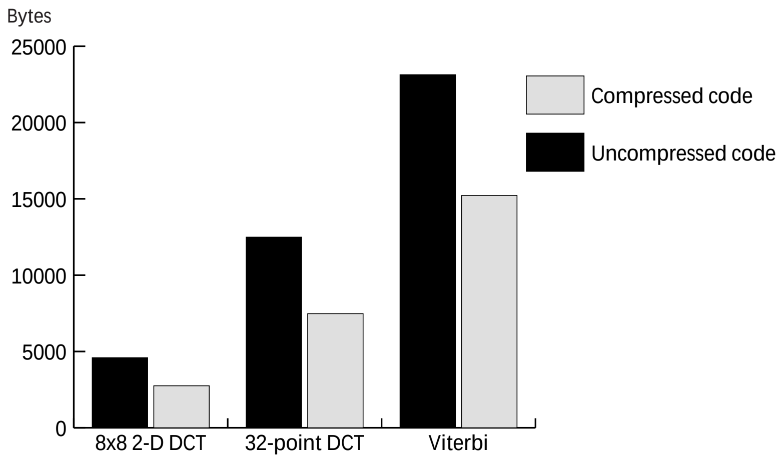


Figure 6. Uncompressed and compressed code sizes.

MOVE instructions could be further compressed by compressing the dedicated long immediate fields. When the field does not contain a long immediate value, the bits dedicated for it are wasted. By compressing the long immediate field in the case it is not used, further improvement in code density could be obtained. Furthermore, the guard ID field could be coded, when its width is larger than two bits.

Table 2. Compression ratios for three benchmark applications.

| Application | Compression ratio |
|---|---|
| 8x8 2-D DCT | 0.600 |
| 32-point DCT | 0.599 |
| Viterbi | 0.658 |

## 7. CONCLUSIONS

In this paper, entropy coding has been applied to improve the code density of application-specific processors based on transport triggering paradigm. Huffman coding was used as entropy coding method to compress the instructions. It was noted that entropy coding provides a notable improvement in code density. The code density could even be further improved by compressing also the dedicated long immediate fields when they do not specify a long immediate value.

In the future research, long immediate fields will also be compressed. Implementation of the decompression circuitry will also be examined to provide fast enough instruction decoding for the real-time requirements of DSP applications.

## 8. REFERENCES

[1] J. M. Rabaey, W. Gass, R. Brodersen, T. Nishitani, and T. Chen, “VLSI design and implementation fuels the signal-processing revolution,” *IEEE Signal Processing Mag.*, vol. 15, no. 1, pp. 22–37, Jan. 1998.

[2] H. Suzuki, H. Makino, and Y Matsuda, “Novel VLIW code compaction method for 3D geometry processor,” in Proc. *IEEE Custom Integrated Circuits Conf.*, Orlando, FL, U.S.A., May 21-24 2000, pp. 555–558.

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

[4] M. Kozuch and A. Wolfe, “Compression of embedded system programs,” in Proc. *IEEE Int. Conf. Computer Design*, Cambridge, MA, U.S.A., Oct. 10–12 1994, pp. 270–277.

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

[6] J. Hoogerbrugge and L. Augusteijn, “Instruction scheduling for TriMedia,” *Journal on Instruction-Level Parallelism*, vol. 1, pp. 1–21, Feb. 1998.

[7] 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.

[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 Engineering*, vol. 5, no. 1, pp. 19–38, 1998.

[10] A. K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall Int., Inc., Englewood Cliffs, NJ, U.S.A., 1989.

[11] J. Kwak and J. You, “One- and two-dimensional constant geometry fast cosine transform algorithms and architectures,” *IEEE Trans. Signal Processing*, vol. 47, no. 7, pp. 2023–2034, July 1999.

[12] J. Takala, D. Akopian, J. Astola, and J. Saarinen, ”Constant geometry algorithm for discrete cosine transform,” *IEEE Trans. Signal Processing*, vol. 48, no. 6, pp. 1840–1843, June 2000.