Immediate Optimization for Compressed Transport Triggered Architecture Instructions

Jari Heikkinen, Tommi Rantanen, Andrea Cilio, Jarmo Takala, and Henk Corporaal*

Tampere University of Technology
P.O.Box 553, 33101 Tampere, Finland
jari.heikkinen@tut.fi

*Eindhoven University of Technology
P.O.Box 513, 5600 MB Eindhoven, The Netherlands
h.corporaal@tue.nl

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, long immediate encoding of compressed transport triggered architecture instructions is optimized to further improve the code density. Six applications taken from different application domains are used for benchmarking.

1. Introduction

Very long instruction word (VLIW) architectures have gained considerable popularity in embedded systems, especially in digital signal processing (DSP) domain, due to their modularity and scalability. Concurrently operating functional units are controlled by a long instruction word that contains dedicated fields for each of the functional units. This kind of instruction encoding leads to poor code density [3]. This in turn increases the system cost.

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 [3], no-operations (NOP) were eliminated from the instructions. A "mask" identifier preceding the instruction specified the fields presented. In [10], 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 has also been applied in [7], 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. Most frequent symbols are given shorter codes and vice versa. Entropy encoding has been applied on VLIWs by means of arithmetic coding, e.g., in [11], and Huffman encoding, e.g., in [8].

Transport triggered architecture (TTA) is a class of statically programmed instruction-level parallel (ILP) architectures that reminds VLIW architectures [4]. In the TTA programming model, only the data transports (moves) to be performed by an interconnection network are specified. Operations occur as a "side-effect". A TTA processor consists of a set of functional units and register files that 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 user-defined functional units.

TTA processors can be designed with a MOVE framework, a toolset that provides a semi-automatic design process [5]. Processor configurations yielding the best cost/performance ratio are searched with a design space explorer and the hardware subsystem is used to generate the hardware description of the chosen configuration. The MOVE software subsystem generates the ILP code for this target processor. TTA processors designed with the MOVE framework are called MOVE processors.

The most serious drawback of TTA is its poor code density, which is mostly due to the long instruction word that contain 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 a data transport on the bus is executed or not. The destination ID field specifies the address of the socket that reads data from the bus, and the source ID field the address of the socket that writes data on the bus. In addition, the instruction word may also contain dedicated fields to define long immediate values.

In our previous work [6], instructions of MOVE processors designed for six applications from DSP and multimedia domains were compressed using an instruction template-based compression method. In this paper, the template-based method is supplemented with a long immediate optimization method to further improve the code density.

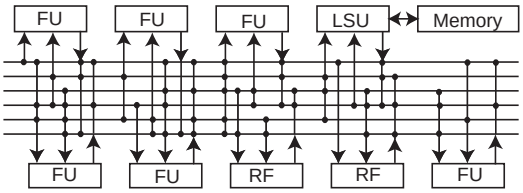

Figure 1. TTA processor organization.

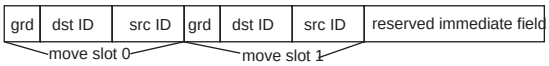

Figure 2. MOVE instruction format.

2. Compression method

Most programs contain parallelism for the VLIW compiler to exploit. However, programs contain also parts where data dependencies limit the parallelism resulting in sequences of instructions that contain only few operations. As most VLIW architectures are tailored for the highly parallel parts, the less parallel parts result in large number of NOPs and waste of instruction bits.

A method using multiple instruction formats, denoted as instruction templates, to avoid explicit specification of NOPs for VLIW and EPIC architectures has been proposed in [2]. An instruction template provides operation slots only for a subset of the functional units. The rest of the functional units receive NOPs implicitly. Each instruction contains a template selection field to specify the used template. From its value the instruction decoder obtains the number of fields in the template, their widths, and their bit positions. This compression scheme results in variable-width instructions and, consequently, more complex instruction fetch and decode.

The cost of the instruction fetch and decode logic becomes significant when the number of templates gets large. Thus, it is not sensible to have instruction templates for all operation slot combinations of the program [1]. A set of templates capable of covering all the combinations found in the program needs to be chosen. To do this, the target program is profiled to find the probabilities of all operation slot combinations. The most beneficial combinations are chosen as templates according to the selection algorithm presented in [2]. For the combinations that do not have a template, a template specifying a superset of the operation slots to be coded must be chosen. On the unused operation slots of the superset template NOPs are explicitly specified.

In [6], the instruction template compression method was applied to instructions of MOVE processors designed for six different applications from DSP and multimedia domains. Move slots and long immediate fields were considered as elements of the templates, i.e., the instruction templates specified fields only for a subset of all the buses and long immediate fields. The results indicated code size reduction to about 30 percent of the largest uncompressed program size. Thus, the template-based compression method efficiently avoids specification of empty data transports and unused long immediate values.

Even though most of the unused long immediate values can be removed by using the instruction templates, the used long immediate values still consume quite a lot of instruction bits, because the width of the long immediate field of the current MOVE toolset is restricted to 32 bits, which is rarely required. The code density can be further improved by getting totally rid of these fields. Instead of using dedicated fields to define long immediate values, all the long immediate values used in the program are stored into a look-up table (LUT). The LUT is accessed by and index that is specified in the opcode part of the source ID field. In typical MOVE designs a 7-bit opcode can be used to specify the LUT index, i.e., 128 locations of the LUT can be indexed. If more long immediate values are used in the program, an additional field needs to be added to the instruction word. Bits of this field are concatenated with the bits of the opcode field to form the index to the LUT. The value obtained from the LUT is then stored to the immediate unit, which puts the value on the desired bus, as depicted in Fig. 3.

3. Experimental Results

Six different applications from DSP and multimedia domain were used for benchmarking. The DSP benchmarks realized two versions of the discrete cosine transform (DCT), two-dimensional (2-D) 8x8 DCT and 1-D 32-point DCT, Viterbi decoding, and edge detection. The multimedia applications, taken from the MediaBench benchmark set [9], implemented MPEG2 decompression and JPEG compression.

First, the design space explorer of the MOVE framework was used to perform design space exploration for each benchmark [5]. From the results three processor 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 template selection and evaluation were then performed. In order to evaluate the long immediate optimization approach, all the long immediate fields were ignored in the template selection. The number of different long immediate values used in the program was also evaluated to determine if an additional LUT index field would be required. For the MPEG2 decompression and JPEG compression benchmarks an additional 3-bit field had to be included.

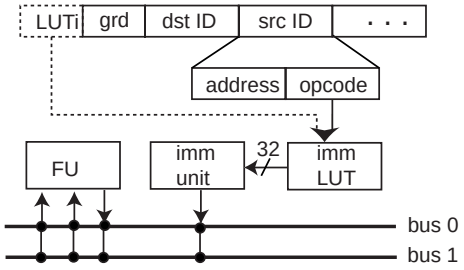

Figure 3. The principal method of obtaining long immediate values from the look-up table.

The instruction widths, number of instructions, and program code sizes for each benchmark on each processor configuration are listed in Table 1. The relative code sizes for each benchmark on different processor configurations, in relation to the code size on the largest configuration, are depicted in Fig. 4. The relative code sizes of our previous work [6], where only the template-based approach was applied, are depicted at the background using white bars. The target processors are classified according to the number of buses. All other resources are scaled correspondingly. The number of templates is varied between 2 and 32. In addition, the case of having a template for every possible combination is evaluated. Furthermore, the lower bound code size (effective code size) is illustrated. This is the theoretical upper bound of the code size reduction, because it does not include bits needed to specify empty data transports nor variable-width encoding support information.

Compared to the results obtained using only the template-based method, code size decreases on all benchmarks and on all processor configurations, except for the smallest configuration of JPEG compression, when also the long immediate optimization is applied. On an average the immediate optimization results in 7.1 percent reduction in code size compared to the reference code sizes. The results indicate that the code size reduction does not depend on the processor configuration. This is quite expected as the number of data transports and long immediates remains quite unchanged even though the processor configuration, for which the program is compiled, changes. Basically, only the number of empty data transports varies between processor configurations. As the empty data transports can be removed quite efficiently by using instruction templates, the immediate optimization results in code size reduction of the same magnitude on different processor configurations, because the same number of long immediates will be removed.

The code size reduction obtained with the immediate optimization decreases when more templates are used to encode the program. This results from the fact that when only the template-based method is applied, the long immediate fields are taken into account as elements of the templates. As the width of the long immediate field is restricted to 32 bits, the template-based method results in better reduction in code size when a long immediate field rather than a move slot can be left out from the instructions. When the immediate optimization is applied, the long immediate fields are not taken into account in the combinations. As a result, the relative improvement achieved using the template-based method is smaller when more templates are used.

Table 1. Statistics of all benchmarks compiled for different processor configurations.

Application	Buses	Instr. width [bits]	Instr. count	Code size [bytes]
8x8 2-D DCT	13	261	127	4144
8x8 2-D DCT	8	179	138	3088
8x8 2-D DCT	3	86	208	2236
32-point DCT	10	196	477	11687
32-point DCT	8	160	502	10040
32-point DCT	4	96	1038	12456
Viterbi	9	200	253	6325
Viterbi	6	146	262	4782
Viterbi	3	89	385	4283
edge	9	207	565	14619
edge	5	126	605	9529
edge	3	89	644	7165
MPEG2dec	8	184	7652	175996
MPEG2dec	6	145	8018	145326
MPEG2dec	5	127	8550	135731
cJPEG	13	272	12198	414732
cJPEG	7	160	12248	244960
cJPEG	3	91	12909	146840

The code size reduction varies also between benchmarks. Best code size reduction is obtained with 32-point DCT, 2-D 8x8 DCT and MPEG2 decode applications. These applications use long immediates more frequently, so the template-based compression method cannot optimize the long immediate fields away from the templates. As all the long immediates are removed when the immediate optimization is applied, the benchmarks that use long immediates more frequently obtain a better code size reduction.

4. Conclusions

In this paper, a template-based compression method supplemented with a long immediate optimization was applied

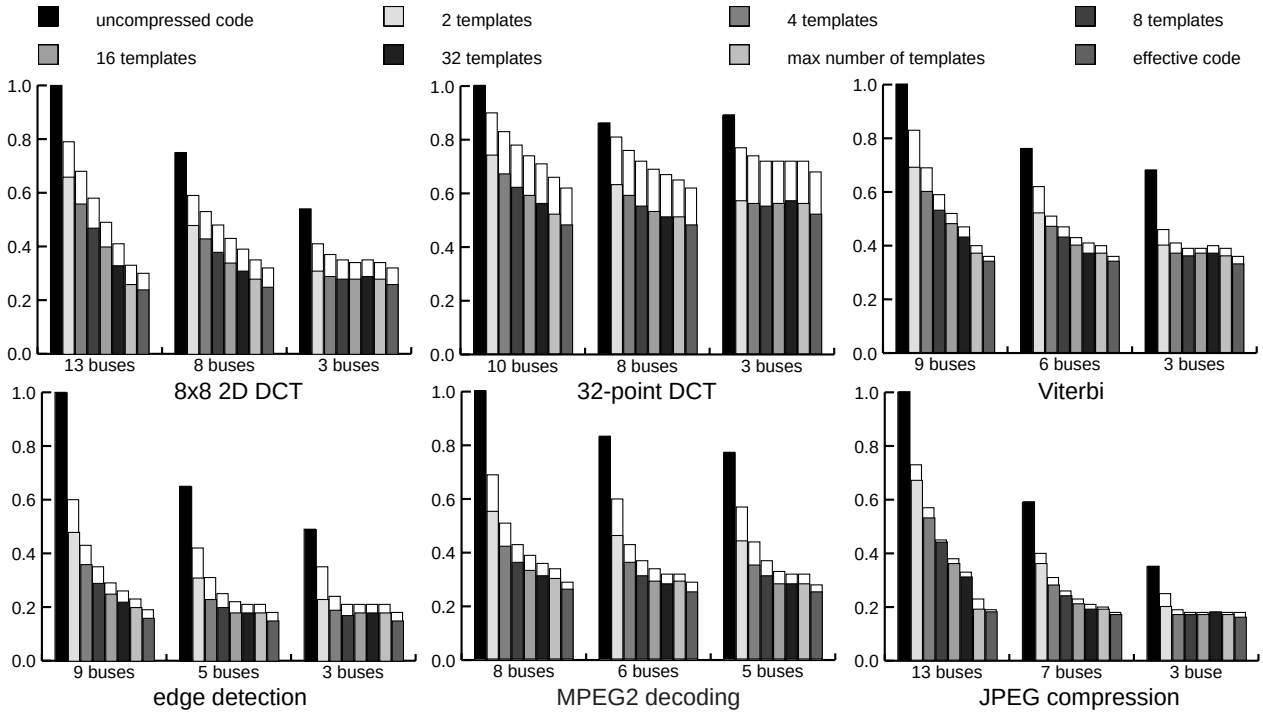

Figure 4. Relative code sizes with different number of templates.

to MOVE processors designed for six benchmarks from multimedia and DSP application domains to improve the code density. It was shown that the immediate optimization reduces the code size on an average by 7.1 percents. Thus, the code wastage caused by the 32-bit wide long immediate fields of the MOVE instructions can be reduced efficiently. The code size reduction does not depend on the processor configuration for which the processor is compiled, but as the number of templates increases, the code size reduction decreases. Together with the template-based compression scheme the immediate optimization results in significant code size reduction.

References

[1] S. Aditya, S. A. Mahlke, and B. R. Rau. Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats. *ACM Trans. Design Automation of Electron. Syst.*, 5(4):752–773, Oct. 2000.

[2] S. Aditya, B. R. Rau, and R. C. Johnson. Automatic design of VLIW and EPIC instruction formats. Technical Report HPL-1999-94, Hewlett-Packard Laboratories, 2000.

[3] 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.*, 37(8):967–679, Aug. 1988.

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

[5] H. Corporaal and M. Arnold. Using transport triggered architectures for embedded processor design. *Integrated Computer-Aided Eng.*, 5(1):19–38, 1998.

[6] J. Heikkinen, T. Rantanen, A. Cilio, J. Takala, and H. Corporaal. Evaluating template-based instruction compression on transport triggered architectures. In *Proc. 3rd IEEE Int. Workshop System-on-Chip for Real-Time Applications*, pages 192–195, Calgary, AB, Canada, June 30 – July 2 2003.

[7] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software - Practice and Experience*, 29(11):1005–1023, 1999.

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

[9] 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*, pages 330–335, Research Triangle Park, NC, U.S.A., Dec. 1–3 1997.

[10] 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*, E82-A(11):2318–2124, Nov. 1999.

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