Code Compression and Decompression for Instruction Cell Based Reconfigurable Systems

Nazish Aslam[1], Mark Milward[2], Ioannis Nousias[2], Tughrul Arslan[1, 2], Ahmet Erdogan[1, 2]

[1] Institute *for* System Level Integration, Alba Campus, Livingston, UK EH54 7EG
[2] University of Edinburgh, School of Engineering and Electronics, Edinburgh, UK EH9 3JL

Abstract

Code compression has been applied to embedded systems to minimize the silicon area utilized for program memories, and lower the power consumption. More recently, it has become a necessity for multiple-issue architectures, such as VLIW and TTA, to permit a viable realization of these designs. In this paper, a code compression and decompression scheme suitable for newly emerging reconfigurable technologies is presented, which pose further challenges by having an order of magnitude higher memory requirement due to much wider instruction words than typical VLIW/TTA architectures. Two dictionary-based lossless compression schemes are implemented and compared for an example reconfigurable system. This paper looks at several conflicting design parameters, such as the compression ratio, silicon area and speed. Test programs for a 2D DCT, minimum error, wimax and H.264 have been evaluated with compression ratios in the range of 41% to 62% recorded with the best scheme.

1. Introduction

The newly emerging technology of reconfigurable computing [1-6] aims to combine the flexibility of FPGAs with the programmability found in General Purpose Processors/Digital Signal Processing Processors in a unified and easy programming environment. Reconfigurable computing cores tend to be developed in terms of flexibility and performance, which correspond to a high amount of parallel processing units with its associated interconnect.

Similarly, other multiple-issue architectures such as VLIW and TTA also exploit instruction level parallelism in order to provide higher throughput for computationally intensive algorithms. However controlling several parallel processing units demand excessively large instruction memories, and an associated very wide instruction fetch bandwidth. A wide bandwidth instruction fetch mechanism is required to supply multiple instructions per cycle to the several processing units of the architecture. This increases the silicon cost, making the higher throughput offered by these architectures less attractive.

Code compression plays a crucial role in tackling these issues by reducing the amount of information needed to represent the code. A lot of research has gone into code compression for embedded systems, particularly for RISC-based architectures [7-9], to reduce silicon area occupied by the memory and consequently reduce power consumption. The asymmetric nature of code compression allows the compressor to be made as complex and computationally intensive task as required as it is performed once at compile time. However, since the decompressor has a direct effect on the targeted processors performance, the decompression hardware should be kept as small and simple as possible to minimize area overhead and latency.

Code compression in multiple-issue architectures faces extra challenges than single-issue due to the need of decompressing a very large instruction word quickly enough so not to compromise the speed of the processors. Yet a reduced instruction bandwidth is also desired to minimize wiring congestion and power consumption. By applying code compression, there is consequent inevitable delay between the main program memory and the processor, hence the aim is to minimize this enough so that it does not become the speed bottleneck for the actual core. Furthermore, it is desirable to minimize the overall area taken up by the decompressor logic so that the benefits achieved by performing program compression, and the sacrifice made in terms of extra latency, are not lost by having a large decompressor hardware.

It is well recognized that higher compressions may be achieved if a compressor and decompressor are made application specific, however this is not feasible for our targeted reconfigurable architecture as well as conventional processors since they should be capable of running several different programs and those programs may change in

1-4244-0910-1/07/$20.00 ©2007 IEEE

future via software downloadable upgrades. Thus a generic design is needed which would give a reasonable compression across most programs, and would be easily scalable as the number of functional units increase.

This paper presents code compression and decompression schemes for an example coarse-grain reconfigurable architecture. The reconfigurable architecture introduced in [6] offers a very high number of parallel processing units and thus has a very wide instruction width. It is dynamically reconfigurable, thus it has to have the ability to store many configuration codes in memory which program the processing units for a particular moment in time.

In this paper, we present a couple of dictionary-based code compression schemes for reconfigurable architectures which aim to satisfy the above requirements and a comprehensive comparison is performed for them. The first scheme is based on existing methods where one dictionary is assigned to a single functional unit. The second scheme is based on a novel 'unit-grouping' technique. It is noteworthy that many of the previously published work on code compression for multiple-issue processors fail to account for silicon area or decoding speed of algorithms, which gives an unrealistically optimistic view of some compression schemes.

This paper is organized as follows. Section 2 reviews related work. Section 3 gives a brief overview of the targeted reconfigurable architecture and discusses the features of a typical program code which may be run on it. The first compression scheme implementation is presented in Section 4, followed by the second compression scheme implementation in Section 5. Section 6 gives the experimental results for both compression schemes and performs a comparison. Finally, Section 7 summaries the findings.

2. Related Work on Code Compression

Different lossless compression schemes can be found in abundance in literature. However it is important to note that code compression has different requirements to other forms of lossless data compression, thus the same compression schemes cannot be applied to both, though some ideas may be borrowed. Many well known data compression schemes provide very good compression ratios but they typically decompress files from beginning to end in a very sequential manner. This is not feasible for embedded systems which require either decompression to be performed on small blocks or on an instruction-by-instruction basis due to the change of flow in the program.

Code compression normally requires each compressed instruction to be encoded as such that its decompression and execution can be done immediately, without waiting for the subsequent instructions to be decoded, as otherwise an unacceptable time delay will be introduced. Also programs require the ability to make conditional jumps to new locations within the code. Whether or not a jump is taken directly depends upon how the condition is evaluated at execution, which in return mandates the previous requirement of decompressing and executing individual instructions immediately without waiting for subsequent instructions decode, as that effort may be wasted if a jump is required.

The two common categories of lossless compression are statistical and dictionary-based. Statistical compression extracts statistical information from the data and uses that information to perform the compression. Many statistical methods result in codewords that are not fixed in length, thus it becomes necessary to first establish the range of bits for the next instruction, and only then the extraction and decompression can start for that particular instruction. Thus, this becomes a very serial operation and following instructions cannot be decoded until the prior ones have already been decoded, increasing the overall processing latency. However, for dense program codes of multiple-issue architectures, statistical methods perform better than dictionary based [10]. It is noteworthy that the targeted reconfigurable architecture in this paper is more similar to the traditional VLIWs with rigid instruction formats rather than modern VLIWs with flexible instructions; i.e. the individual instruction positions within a wide instruction correspond to specific functional units, thus their code is less dense due to many inactive units, whereas in flexible instructions, the individual instructions within a wide instruction can be processed by any functional units.

Statistical compressions for modern VLIWs include work done by Larin and Conte [11] who uses Huffman coding to compress instructions in 3 different ways allowing varying degrees of trade-off to be made between the compressed program size and the decompressor size. Further work is done by Xie et al [12] who have used arithmetic coding with a Markov model and report compressions of 67.3% to 69.7%. They also present a Tunstall based variable to fixed encoding scheme [13] and a fixed to variable encoding scheme [10] with compressions in the range of 65% and 70% to 82% respectively.

Dictionary-based schemes compile dictionaries of frequent instructions found in a program and replace those instructions with the corresponding dictionary index. Dictionary-based algorithms normally result in poorer compression than statistical methods but tend to result in faster and simpler decompression logic. Nam et al [14] propose a dictionary based compression scheme for traditional VLIWs, and use isomorphism to create two dictionaries, one for storing operations, and the other for operands. Only frequent instruction words are compressed. The point to note is that this compression scheme becomes worse when the number of functional units increases from

4 to 12. Compression ratios from 63% to 71% are reported although no discussion on silicon area or latency is found.

Further dictionary based compression schemes are provided by Ros and Sutton [15] who investigates compression at 3 levels of instruction granularity. The most efficient compression scheme gave an average compression ratio of 68.3%; however it is a sequential design, thus quite slow. Other design parameters such as area or latency are not shown, and furthermore dictionary initialization bits are not added to the compression ratio calculation. Ishiura and Yamaguchi [16] also use dictionaries where they apply automatic field partitioning to partition instructions into smaller bit-sets to keep the corresponding dictionaries small. Compression ratios of between 46% and 60% are reported.

3. Target Reconfigurable Multiple-Issue Architecture

3.1 Instruction cell based architecture

The recently developed industrial distributed reconfigurable instruction cell based architecture [6] was targeted for applying the proposed code compression techniques. The architecture is able to provide dynamic hardware reconfigurability and a high throughput. It uses a complex scheduler to effectively extract instruction level parallelism from general-purpose high level language codes [17].

The architecture consists of an array of heterogeneous instruction cells, where the number and type of these function units are parameterizable upon application. For this paper, a 64 function units architecture was chosen, although the actual reconfigurable system is capable of having several fold more units if desired. This implies significantly more processing units than other existing multiple-issue architectures; e.g. VLIWs usually have up to 12 processing units. Furthermore, for VLIW, each processing unit can perform different functions like an ALU, whereas the units of the target reconfigurable architecture are more specific purpose, such as multiplier, divider, shifter, adder, logic, registers, etc. Understanding the exact nature of each of the processing element is irrelevant for this paper, however it is suffice to say that each unit, depending upon its type, can have a varying number of configuration bits associated with it. The units of our target system contain cells with configuration bits varying from 3 up to 32 bits each. Each processing unit performs a specific subset of primitive operations, and the associated configuration bits are used to configure that unit's operation appropriately at any moment in time. These configuration bits are accessed from the instruction memory, whereas the operands required by the functional units are obtained separately from the data memory.

3.2 Program code

Each wide instruction for the targeted 64 unit architecture is 474 bits; these wide instructions will now be referred to as steps. Examining a typical program code, poor code density is obvious. Most steps contain 'No Operation' (nop) instructions for inactive units, similar to traditional VLIW. Units are inactive if they are not utilized in a given step and occur frequently due to inter-instruction dependencies. They can be identified by their all zero configuration bits. This is labeled spatial redundancy. The second type of redundancy is the repetition of configuration settings for a given unit several times throughout the lifetime of a program code. This is labeled temporal redundancy. And finally, performing some code profiling revealed a very frequent usage of units with large number of configuration bits, thus any compression achievable on these would be quite beneficial.

4. Compression Scheme 1 (CS1)

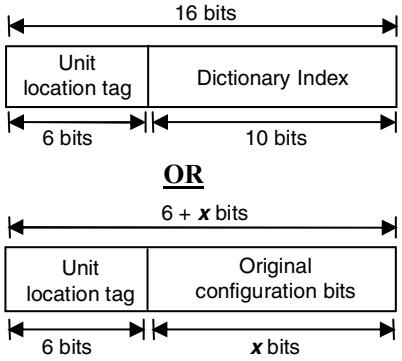
The first design was intentionally made simplistic in an attempt to minimize the latency associated with a single step decompression, while achieving a reasonable memory size reduction, and is based on similar compression techniques already applied for VLIW/TTA processors.

Spatial redundancy removal eliminates all nop instructions for inactive units in a given step. This results in varying step sizes. Consequently, the decompressor would neither know when a complete single step has ended nor which functional units the configuration settings are intended for. Adding especially reserved end-of-step tags to the end of each step can signify a step completion. Some tags are also needed before each configuration bit-set within a step to provide information on which unit they are allocated. Given there are 64 units in the targeted reconfigurable architecture, 6bits can identify all the units uniquely, and would precede the corresponding configuration bit-sets. A reserved end-of-program tag is needed to indicate that the entire compressed program code has been decoded. For this, a codeword which can be guaranteed to never occur in any compressed program code can be used.

For removal of temporal redundancy, the common technique of index dictionaries is applied. One dictionary is associated per unit, where each dictionary holds a single copy of all the unique configuration values for its associated unit. Thereafter the original program code is scanned and all the configuration values are replaced by their corresponding dictionary index. Compression can only be achieved with dictionaries if the number of bits being replaced is more than the dictionary index bits. This can only happen if we can guarantee that only a subset of all the possible unique configuration values can occur during the lifetime of any program code. This can easily be

the case for a 32bit functional unit, where the 32 configuration bits can have up to 2^{32} unique values, yet in practical terms, only a small fraction of these are ever utilized. However, for some of the other units that have very few configuration bits, such as 3 or 4 bits each, using a dictionary is not feasible, since the chances of at least half of 2^3 or 2^4 unique combinations occurring in a given program are very high. Hence such unit configuration bits are best kept in their original form. After performing some code profiling, it was found that a dictionary with 2^{10} words would be more than sufficient for units with large configuration bits without the risk of overflowing, even for quite large programs like H.264. The large configuration bits will thus get replaced by 10bit indices. Note that even though the dictionary lengths are the same for large configuration units, the word widths will vary for each unit dictionary to match its corresponding unit's number of configuration bits.

The above mentioned redundancy removal techniques result in variable length compressed steps as well as variable sized payloads within each step. A payload represents one compressed instruction for an active unit inside a step and for CS1, it is made up of:



Thus, during decompression, the first 6bits for unit location tag have to be extracted and decoded to deduce the number of bits to read in for either the dictionary index or configuration bits. Once established, the next bits are read and decoded. Then, the following 6bits are read and decoded to know how many bits are now needed, and so on. A good compression ratio may be achieved using variable length payloads, however due to the sequential nature of the entire decoding process, the time required to decompress a complete step is unacceptable as the decompressor will become the speed bottleneck for the reconfigurable processor.

Nonetheless, if all the payloads are made the same fixed length, then several payloads may be extracted and decompressed in parallel, speeding up the time it takes to decompress a complete step. A payload size of 16 bits is needed as a minimum to be able to correctly compress the larger unit instructions. Thus all the other payloads for the smaller units can also be fixed to this, which will result in an expansion of the code in some areas rather than compression, however this trade-off is necessary for speeding up the decompression process. The resulting decompressor design is shown in Fig. 1.

This decompressor assumes that it is able to process up to 8 payloads in parallel from the compressed program memory; hence it expects 128bits bandwidth. This is clearly smaller than the initial 474bits bandwidth requirement, and the number of payloads fetched concurrently can easily be reduced or increased as desired. More payloads mean faster decode per step. The design has a one stage pipeline in order to increase its throughput.

The compression is performed on a per basic block basis in order to ease how conditional jumps can be taken. When the reconfigurable core executes a step, and identifies a branch to a new location, it simply sets the target address on the 'sram_start_address' line and toggles the 'initialize_sram_address' signal of the decompressor. This causes the decompressor to entirely flush the pipeline and start decompressing steps from the new specified address. The target addresses for each branch get updated during program compression according to their new locations.

Performing code compression undoubtedly introduces some extra delay in the path of code fetch and execution. Given that the performance of the reconfigurable core is of utmost importance, then it is quite desirable to be able to run smaller program codes directly in an uncompressed form. This is possible by using the index dictionaries for each of the units as a cache to directly store all the steps of an uncompressed program if the program is no larger than 1024 steps. A simple state-machine may be implemented to record all the steps into the dictionaries and steps may be extracted in an incremental manner.

With this scheme the best compression achievable for a single step is 474bits reduced down to 16bits. This happens if only one unit is active. However, the worst case step

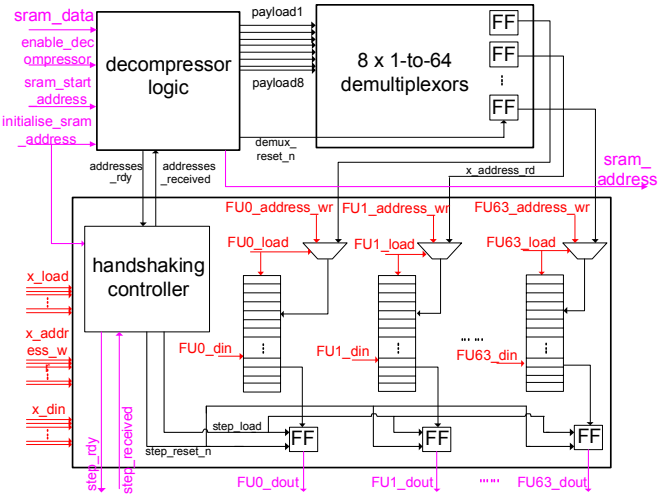


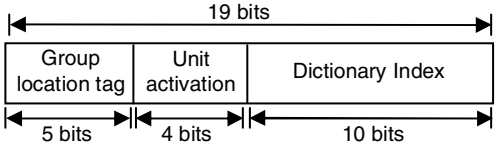
Figure 1. Decompressor design for CS1.

compression would actually increase the 474bits step to 1024bits, and this can occur if all 64 units are active within a step.

5. Compression Scheme 2 (CS2)

CS2 is similar to CS1. It performs compression on per basic block basis and handles branches in the same manner. Spatial and temporal redundancies are removed as before and the decompressor has a one-stage pipeline implemented like in the previous scheme. However, the main difference lies in how the temporal redundancy is removed. Instead of assigning one 1K dictionary per functional unit, a *group of functional units* get assigned a 1K dictionary. One group may contain from 1 up to 4 different units. This idea came after encountering the undesirable expansion of code for smaller units in the previous scheme. Grouping some of the smaller units together as such that their accumulated number of bits is either at least equal to or more than 10bits, ensures that the compressed payload is not worse than what it would have been in its uncompressed form. It has to be ensured that the accumulated number of bits do not become too much greater than the 10bits, as the 1K dictionaries will become insufficient to hold all the possible unique combinations occurring in a given program code. For example, grouping the four 3bit adder units would result in a total of 12bits. Such a grouping means that up to a maximum of 2^{12} unique values can occur in the program code if we assume that all of the four add units are active together in any given step and go through each of their possible values. However, this scenario actually occurring in any practical program code is highly unlikely. So, instead of having a 2^{12} words dictionary, a 2^{10} dictionary is quite sufficient as it allows for up to 25% of the total unique configurations to occur in any given single program code.

25 groups were created after performing the unit groupings for the target 64 functional unit reconfigurable core. A 5bits group location tag is now sufficient to uniquely identify all the group dictionaries. However, each payload now requires the addition of unit activation bits that show which units within a given group are active in a given step. As a group was chosen to contain from 1 up to 4 units, 4 bits are needed for units' activation information. Thus, a single payload now becomes 19bits in length:



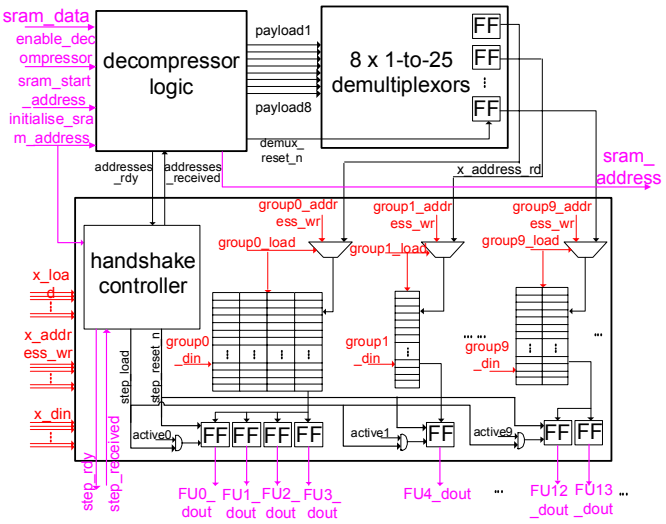


Figure 2. Decompressor design for CS2.

Another advantage of unit groupings is that fewer demultiplexers are now needed in the design, which can lower the logic area overhead. As before, the decompressor (see Fig. 2) assumes it is able to decode 8 payloads in parallel, therefore expects 152bits instruction fetch bandwidth. The dictionaries are still usable as an instruction cache to directly store short (<1025 steps) uncompressed programs and bypass the decompressor completely. With this scheme, the best compression achievable for a single step is 474bits reduced down to 19bits. This happens if only one functional unit is active. If all the functional units within a step are active then 474bits expand to only 475bits and this represents the worst case compression for this scheme.

6. Performance Comparison

A set of DSP applications are used to evaluate the performances of each compression scheme; namely 2D DCT, minimum error, wimax and H.264 test programs. The decompressor designs are implemented using Verilog HDL and synthesized onto 0.13µm CMOS technology using Synplicity ASIC. The results are given in Table 1.

The total decompressor area for CS2 is 31.6% lower than CS1. In both designs, the dictionaries make up 96% of the total area. These dictionaries are automatically generated using Virtual Silicon's Memory Compiler. Note that even though the capacities of dictionaries for both designs are equivalent, their areas differ significantly. This is because more SRAM read and write circuitry is needed in CS1 as it has 64 1K dictionaries, whereas CS2 only has 25 1K dictionaries with wider words. Furthermore, by reducing the number of demultiplexers in CS2 decompressor, and the easing of the wiring congestion as a result, the decompressor logic area is also reduced.

		CS1	CS2
Decompressor logic		0.16mm^2	0.11mm^2
Dictionaries area		3.79mm^2 (485376bits)	2.59mm^2 (485376bits)
Total decompressor area		***3.95mm^2***	***2.7mm^2***
Propagation delay		1.914ns	1.918ns
Best case step decode		3.828ns	3.836ns
Worst case step decode		19.14ns	9.59ns
Power consumption		1253µW/MHz	764µW/MHz
Compression ratio	**2D DCT**	73.75%	62.1%
	Min error	40.2%	42%
	Wimax	41.51%	40.66%
	H.264	46.1%	41%

Table 1. Performance results for CS1 and CS2

The propagation delays for both designs are equivalent, allowing just over 500MHz clock frequency. Since both decompressors have a single stage pipeline, the shortest time to decode a complete step is 2xpropagation delay. However, in the worst case where all the units within a complete step are active, the CS1 takes almost twice as long as CS2 to decode a complete step. As the dictionaries contribute to the bulk of the area overhead, the power consumption for each design was estimated using datasheets of the individual SRAMs. CS2 is almost 40% less power hungry than CS1. The compressions are recorded in terms of compression ratio, which is defined as follows:

$$\text{Compression ratio} = \frac{\text{Compressed program size} + \text{Dictionary initialization bits}}{\text{Original program size}} \times 100$$

Many other published works ignore the bits required for initializing the dictionaries and treat them as being pre-initialized in ROM. For our examples, reductions in the range of 26.25% to 59.8% were achieved with CS1, whereas CS2 gave reductions in the range of 37.9% to 59%. To better judge whether either of these program compressions is worthwhile, given that the decompressor introduces its own area overhead and delays, the results can be analyzed in terms of effective area savings achieved. To do this, the size of the original uncompressed program is found in terms of equivalent SRAM area. Then the size of the compressed program, including any dictionary initialization bits, is measured in terms of SRAM area too and the fixed size of the total decompressor area added to it. See Table 2.

It is apparent from the results that short programs actually worsen the area overhead; however larger programs, like the H.264, provide significant savings. For the compressions to be worthwhile, a total program size reduction of approximately 122Kbytes is required for CS1 and 49Kbytes for CS2.

		CS1	CS2
No. of bits for original program & equivalent SRAM area	**2D DCT**	16590bits; 0.193 mm^2	
	Min error	11850bits; 0.178 mm^2	
	Wimax	204768bits; 0.814 mm^2	
	H.264	9442554bits; 36.9 mm^2	
No. of bits for compressed program & equivalent SRAM area	**2D DCT**	12235bits; 0.178 mm^2	10295bits; 0.175 mm^2
	Min error	4758bits; 0.098 mm^2	4967bits; 0.101 mm^2
	Wimax	84996bits; 0.418 mm^2	83253bits; 0.411 mm^2
	H.264	4350841bits; 17.4 mm^2	3836164bits; 15.4 mm^2
Decompressor h/w area		3.95mm^2	2.7mm^2
Total area required after performing compression	**2D DCT**	4.13 mm^2	2.88 mm^2
	Min error	4.05 mm^2	2.8 mm^2
	Wimax	4.37 mm^2	3.11 mm^2
	H.264	21.4 mm^2	18.1 mm^2
Area savings (negative value if worse area)	**2D DCT**	-2040%	-1392%
	Min error	-2175%	-1473%
	Wimax	-437%	-282
	H.264	42%	51%

Table 2. Compression achieved in terms of effective area for individual applications

7. Summary

Two dictionary-based lossless code compression schemes are implemented for a multiple-issue reconfigurable architecture, with the aim to reduce the silicon area and bandwidth costs associated with the instruction memory. In a comprehensive comparison, our novel unit-grouping technique outperforms the standard technique used with dictionary-based compressions. Significant compression ratios in the range of 41% to 62% are achieved. The compression scheme is easily scalable to reconfigurable architectures with increased number of functional units.

References

[1] Mirsky, E.; DeHon, A.; "Matrix: A reconfigurable computing architecture with configurable instruction distribution and deployable resources," IEEE symposium on FPGAs for custom computing machines, pp.157-166, Apr. 1996

[2] Hauser, J.R.; "Augmenting a microprocessor with reconfigurable hardware," Thesis, University of California, Berkeley, 2000

[3] D-Fabrix processing array, Reconfigurable Signal Processor, www.elixent.com, 2004

[4] XPP, PACT, "OFDM decoder for wireless LAN–whitepaper," www.pactcorp.com, May 2002

[5] Reconfigurable Computing, Philips, Avispa, www.siliconhive.com, 2004

[6] Reconfigurable Instruction Cell Array, U.K. Patent Application Number 0508589.9.

[7] Lefurgy, C.; Bird, P.; Chen, I.-C.; Mudge, T., "Improving code density using compression techniques," Proc. of the 30th Intl. Symposium on Microarchitecture, pp:194 – 203, Dec. 1997

[8] Liao, S.; Devadas, S.; Keutzer, K., "Code density optimization for embedded DSP processors using data compression techniques," Proc. of the 16th Conference of Advanced Research in VLSI, pp272, 1995

[9] Wolfe, A.; Chanin, A., "Executing compressed programs on an embedded RISC architecture," Proc. of the Intl. Symposium on Microarchitecture, p.81-91, Dec. 1992

[10] Xie, Y.; Wolf, W.; Lekatsas, H.; "Code compression for embedded VLIW processors using variable-to-fixed coding," IEEE Trans. On Very Large Scale Integration Systems, Vol. 14, No. 5, pp.525-536, May 2006

[11] Larin, S.; Conte, T.; "Compiler-driven cached code compression schemes for embedded ILP processors," Proc. of 32nd Intl. Symposium on Microarchitectures, pp. 82–92, 1999

[12] Xie, Y.; Wolf, W.; Lekatsas, H.; "Code compression for VLIW processors," Proc. of Data Compression Conference, pp. 525, 2001

[13] Xie, Y.; Wolf, W.; Lekatsas, H.; "A code decompression architecture for VLIW processors," Proc. of the 34th Intl. Symposium on Microarchitectures, pp. 66–75, 2001

[14] Nam, S.; Park, I.; Kyung, C.; "Improving dictionary-based code compression in VLIW architectures," Trans. on Fundamentals of Electronics, Communications and Computer Sciences, Vol. E82-A, pp. 2318-24, Nov. 1999

[15] Ros, M.; Sutton, P.; "A Hamming distance based VLIW/EPIC code compression technique," Proc. of the Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems, pp. 132–139, Sept. 2004

[16] Ishiura, N.; Yamaguchi, M.; "Instruction code compression for application specific VLIW processors based on automatic field partitioning," Proc. of the Workshop on Synthesis and System Integration of Mixed Technologies, pp. 105-109, 1997

[17] Yi, Y.; Nousias, I.; Milward, M.; Khawam, S.; Arslan, T.; Lindsay, I.; "System-level scheduling on instruction cell based reconfigurable systems," Proc. of Design, Automation and Test in Europe, Mar. 2006