Application Analysis with Integrated Identification of Complex Instructions for Configurable Processors

Nikolaos Kavvadias and Spiridon Nikolaidis

Section of Electronics and Computers, Department of Physics,
Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece
nkavv@skiathos.physics.auth.gr

Abstract. An extensible and configurable processor is a programmable platform offering the possibility to customize the instruction set and/or underlying microarchitecture. Efficient application analysis can identify the application parameters and instruction extensions that would influence processor performance. An application characterization flow is presented and demonstrated on the Wavelet/Scalar Quantization image compression application. In this context, novel application metrics are identified as the percentage cover, maximum cycle gain for each basic block and candidate-induced application speedup due to possible complex instructions. Furthermore, evaluating the instruction candidates during application analysis is proposed in order to establish a link with subsequent design space exploration steps.

1 Introduction

Embedded processors suitable for consumer applications, present interesting architectural refinements, in order to support power-hungry algorithms e.g. for high bandwidth wireless communications or video compression and decompression [1]. The portability of these systems makes energy consumption a critical design concern. For successfully implementing software applications on domain-specific processors under tight time-to-market constraints, requirements of high flexibility and programmability have also to be met.

The challenge of delivering the optimum balance between efficiency and flexibility can be met with the utilization of customizable processors. Most commercial offerings fall in the category of configurable and extensible processors [2],[3]. *Configurability* lies in either a) setting the configuration record for the core (regarding different cache sizes, multiplier throughput and technology specific module generation) or b) allowing modifications on the original microarchitecture template. In the first case, the end user selects the synthesis-time values for certain parameters of the processor core [4]. The second case requires that the basic architecture of the core is modifiable. For instance, the flexible pipeline stage model employing local control in [5] enables altering the pipeline depth of the processor. *Extensibility* of a processor comes in modifying the instruction set architecture by adding single-, multi-cycle or pipelined versions of complex instructions. This may require the introduction of custom units to the execution stage of the processor pipeline and this should be accounted in the architecture template of the processor. The instruction extensions are generated either automatically or manually from a self-contained representation of the application code, assuming a structural and instruction set model of the processor [6].

Characterizing the application workload is a fundamental step in microprocessor design, since based on this analysis, the processor designer can decide the appropriate configuration and the required instruction extensions of a customizable core for achieving an advantageous performance-flexibility tradeoff. In this paper, an approach to application analysis is presented for extracting application parameters. The framework is based on the freely available SUIF/Machine SUIF (MachSUIF) compiler infrastructure [7]. Opposed to previous approaches, complex instruction candidates are identified at the stage of application analysis, since such information can be used for pruning the design space of possible instructions in an Application-Specific Instruction set Processor (ASIP) design flow. Static and dynamic characteristics of the application are also extracted and their impact on candidate identification is investigated. The metrics of percentage cover, maximum basic block cycle gain and candidate-induced application speedup that quantify the impact of including specific complex instructions are given. Overall, it is argued that generating an initial set of instruction candidates should be an integrated step of the application characterization flow to guide subsequent design space exploration steps.

The rest of this paper is organized as follows. The related work in application analysis and candidate instruction (template) identification is summarized in Section 2. Each step of the application characterization flow is described in Section 3 along with the use of existing and the associated in-house tools we have developed. Section 4 discusses the application of the proposed approach on the Wavelet/Scalar Quantization (WSQ) image compression algorithm and the corresponding results. Finally, Section 5 summarizes the paper.

2 Related Work

An important issue in domain-specific processor design is the task of application analysis extracting both static and dynamic metrics for the examined applications. Although important in ASIP synthesis, the effect of introducing candidate instructions to accelerate processor performance on a given application set is not adequately examined in context of application analysis in the vast majority of related work.

In [8] both the application and a specification of the processor are input to an estimation framework based on SUIF. A number of parameters characterizing the application are extracted: the average basic block size, number of multiply-accumulate operations, ratio of address to data computation instructions, ratio of I/O to total instructions, register liveness, and degree of instruction-level parallelism. Compared to [8], our approach searches for all candidate instructions by identifying fully-connected subgraphs in the DFG of each basic block, instead of restricting the search to a specific complex instruction type. Also, their tool has been designed for processor selection and not to assist ASIP synthesis, which explains the fact of using coarse parameters extracted from the instruction mix. These are intended as thresholds for selecting or rejecting a specific processor while our method performs the analysis in a much finer level.

A performance estimator using a parameterized architecture model has been developed in [9]. While the work presented is significant, the method has been constructed with a specific processor type in mind. E.g. the assumed addressing modes are specific to DSP processors. Our method can identify non-DSP specific complex addressing schemes, as shifter-based addressing modes similar to those of the ARM7 processor.

Multimedia benchmark suites have been presented in [10],[11] along with their characterization profile. In [11] the popular MediaBench suite is introduced, characterized with metrics suitable for general-purpose processors. The benchmark suite in [10] is comprehensive with a thorough study, however also assuming a GPP template. Again, guidelines to finding the appropriate extension instructions suitable to multimedia-enhanced GPPs are not provided.

3 The Proposed Approach for Application Analysis and Characterization

It is often at early stages in processor design, that the compilers and simulators for the entire range of applicable processor architectures one needs to consider, are not available [8]. In order for the application characterization results to be useful to the spectrum of evaluated microarchitectures, a common estimation platform is required. We propose using the MachSUIF intermediate representation (IR) for this purpose. In MachSUIF, the IR description uses the SUIF virtual machine instruction set (SUIFvm), which assumes that the underlying machine is a generic RISC, not biased towards any existing architecture.

In this case, the application is decomposed into its IR consisting of operations with minimal complexity, best known as primitive or atomic instructions. It is possible to organize the IR description of the application into Control Data Flow Graphs (CDFGs) with primitive instructions as nodes and edges denoting control and data dependencies. In MachSUIF, the executability of the original C program is retained at the level of the SUIFvm instruction set. The corresponding SUIFvm code consists the executable intermediate representation [12] of the benchmarked program. Dynamic characterization can be performed on the host machine by executing the resulting C program, generated by translating the SUIFvm back to C code.

The proposed application characterization flow is shown in Figure 1. Shaded blocks on the diagram distinguish our in-house tools from the available passes of the infrastructure. In the remaining paragraphs of this section, we detail the steps of the application characterization procedure.

In Step 1, the input C code for the application is passed through the *c2s* pass of the SUIF frontend. In this stage, the application is preprocessed and its SUIF representation is emitted. In the second step, *do_lower*, several machine-independent transformations are performed as dismantling of loop and conditional statements to low-level operations. The resulting description is termed as lower SUIF in contrast to higher

SUIF generated by *c2s*. Step 3 is needed to translate the lower SUIF code into the SUIFvm representation. For this task, the *s2m* compiler pass is used.

The IR code has not been scheduled and has not passed through register allocation, which is important so that false dependencies within the data flow graphs of each basic block are not created [13]. Step 4 performs architecture-independent optimizations on the IR, such as a) peephole optimization, b) constant propagation, c) dead code elimination, and if decided, d) common subexpression elimination (CSE) to construct the optimized SUIFvm description.

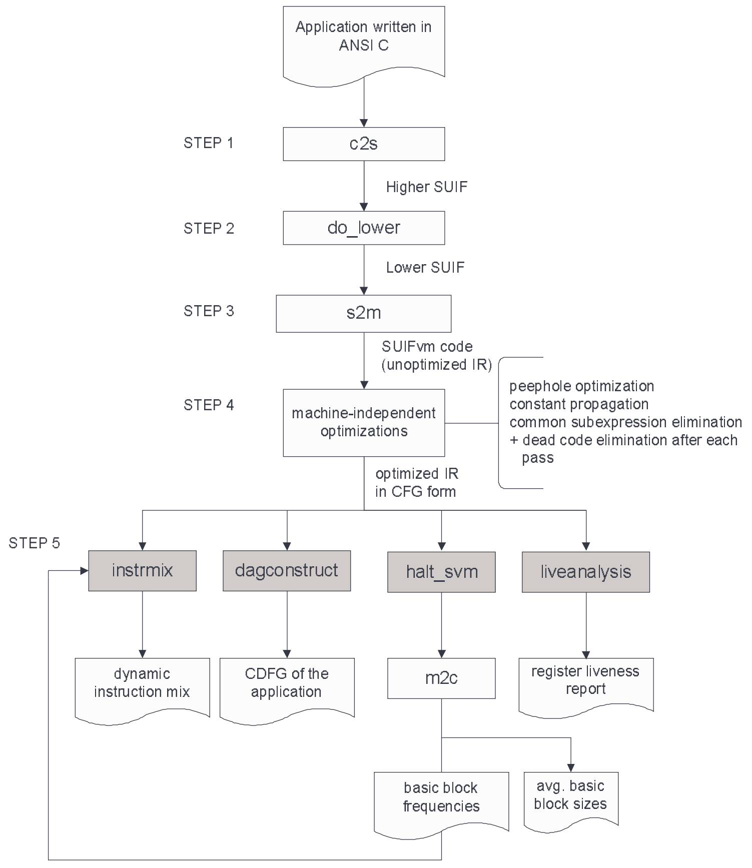

Fig. 1. The proposed application analysis and characterization flow

Peephole optimization suppresses redundant move operations and is used to remove unnecessary type casting (CVT) operations that MachSUIF has the tendency to produce after the application of an active pass. The usefulness of CSE depends on the algorithm applied for complex instruction generation. Specifically, if overlapped templates are permitted during instruction generation, CSE will not prohibit the identification of any beneficial candidate. However, if a faster algorithm is used that only allows orthogonal covers, some opportunities will be missed. Assume a basic block with two instances of the same subexpression, e.g. a subgraph comprised of two primitive operations, placed in the core of two different single-cycle complex instructions consisting of three and four primitives respectively. If the second subexpression is eliminated, then the second instruction candidate could only consist of two primitives.

Finally, during Step 5, specific static and dynamic metrics are gathered. The corresponding analysis passes accept SUIFvm in CFG form.

The *dagconstruct* pass parses each node in the CFG and constructs the corresponding CDFG. Note that instructions involving memory operands (as in CISC-like machines) require additions to some libraries of the infrastructure. In this case, the *dagconstruct* pass should be updated to reflect these changes introduced to the *suifvm* library.

A pass for generating the static instruction mix, *instrmix*, has also been developed. By using the execution frequencies for the basic blocks of the application, the dynamic instruction mix can be easily calculated. For calculating the execution frequencies, the SUIFvm code is translated to single-assignment style C using the *m2c* pass. Pass *halt_svm* is used to instrument the C code by adding counters at the start of each basic block.

The *liveanalysis* pass is based on the *cfa* library and calculates the number and names of registers that are alive at basic block boundaries. The corresponding results help the designer decide the register file size.

4 Application analysis for the Wavelet/Scalar Quantization image compression algorithm

The case study application is based on a wavelet image compression algorithm [14] and is part of the Adaptive Computing Benchmarks [15], which are used to evaluate specific characteristics of reconfigurable architectures. Reportedly, the selected benchmark is used to stress reconfigurability by splitting execution time among several kernels. A compliant implementation of the WSQ algorithm is required to perform four standard steps: wavelet transform, quantization, run-length encoding and entropy coding (for the encoder part). The entropy encoding stage is realized with a Huffman encoder. In our paper, the application analysis framework is used to extract characteristics for both the encoding and decoding algorithms.

4.1 Instruction mix

The dynamic instruction mix provides a classification of the application instructions into types based on the functional units used. Instructions are divided into integer and floating-point, while each of those has distinct subtypes: load and store, arithmetic, logical, shift, multiply, division, unconditional and conditional branch, call/return and remaining instructions. Figure 2 shows the instruction mix statistics for the *compress* and *decompress* applications, which correspond to the WSQ encoder and decoder, respectively. Note that WSQ is a pure integer application.

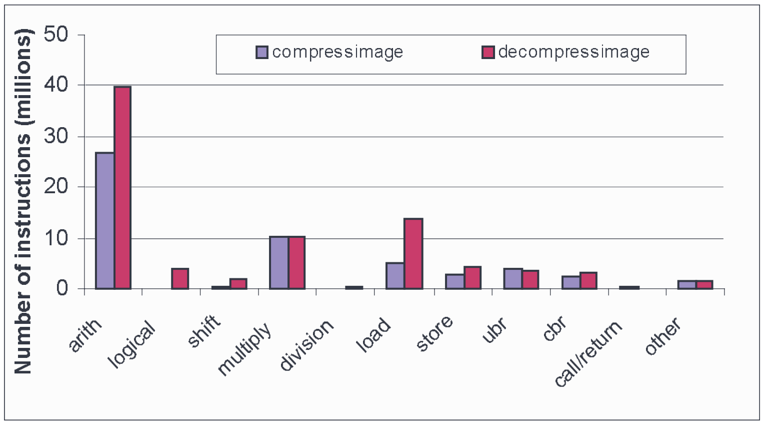

Fig. 2. Instruction mix statistics for the WSQ algorithm

It is clear that arithmetic operations dominate the instruction mix of the applications. Also, *decompress* has higher computational complexity than *compress* since it requires higher amount of arithmetic and load instructions. The ratio of branches to the total instructions is very small (9.8%) which means that higher execution frequencies are encountered for relatively large basic blocks. This conclusion is supported by the results of Section 4.4.

4.2 Average basic block size

The basic block sizes are easily calculated simultaneously to the static instruction mix. It is found that 3.98 and 4.48 instructions consist the average basic block for the *compress* and *decompress* applications, respectively. At a first glance, this result does not leave much room for performance benefits by exploiting complex instruction candidates within the same basic block. However, as it will be shown, heavily executed portions of the code comprise of rather large basic blocks.

4.3 Register liveness analysis

It is found that *decompress* has lower register pressure with a maximum of 8 saved registers while *compress* requires 11 saved registers. These results constitute a lower bound on the required local storage resources, more specifically the number of allocable registers of the architecture, for the WSQ algorithm.

4.4 Basic block frequencies

Figure 3 indicates the execution frequencies and sizes for the most heavily executed basic blocks for *compress* and *decompress*. Each basic block is assigned a unique name of the form: <*file_name*>.<*function_name*>.<*basic_block_number*>.

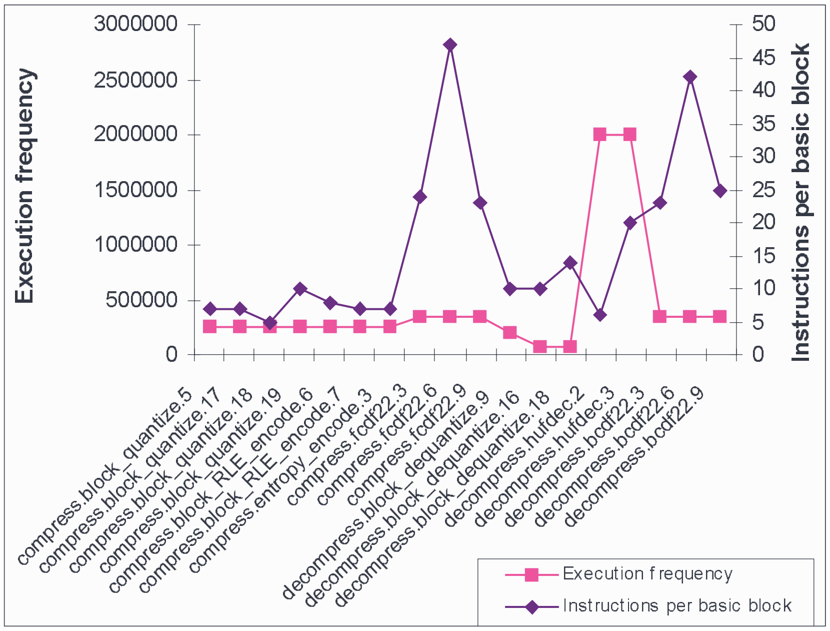

Fig. 3. Execution frequencies and sizes for the heavily executed basic blocks

It is evident from Figure 3 that there exists space for achieving speedup in the performance critical basic blocks since their size is significantly above average.

4.5 Data flow graph analysis for identifying candidate instruction extensions

The *dagconstruct* pass referred in the beginning of this section, generates DFGs for each basic block. Then, fully-connected subgraphs of these DFGs are identified as potential complex instructions. A measure of success for the selection of complex instructions using orthogonal covers is given by the *percentage cover factor* determined by the proportion of the number of instructions after selection to the number of instructions prior selection. A speedup factor, *maximum basic block cycle gain*, is also introduced and is calculated as the product of the maximum performance gain in cycles (assuming no data hazards and spills to memory) with the execution frequency of the specified basic block. An estimate on the performance impact of selecting a set of isomorphic patterns is given by the *candidate-induced application speedup* metric defined as the application speedup due to selecting the complex instruction. At this point, calculating the latter metric is not automated, and for this reason it is evaluated on the performance-critical basic blocks of the application. Since only 10 basic blocks incorporate the 85.3% of the instructions for *compress* and 8 basic blocks the 97.3% of the instructions for *decompress*, the extracted results are valid.

Table 1 shows the percentage cover factor and maximum cycle gain for the performance-critical basic blocks. In columns 2 and 3, the number of instructions prior and after complex instruction matching is given. The percentage cover and maximum gain values are given in columns 4 and 5 respectively. The average percentage cover is 83.2%.

Table 1. Template selection results for the performance-critical basic blocks

Basic block ID	# Instr. (prior select.)	# Instr. (after select.)	% cover	Maximum cycle gain
compress.block_quantize.5	7	3	85.7	1048576
compress.block_quantize.17	7	3	85.7	1048576
compress.block_quantize.18	5	2	80.0	786432
compress.block_quantize.19	10	3	70.0	1806336
compress.block_RLE_encode.6	8	3	100.0	1290285
compress.block_RLE_encode.7	7	5	57.1	516096
compress.entropy_encode.3	7	4	57.1	786438
compress.fcdf22.3	24	9	87.5	5160960
compress.fcdf22.6	47	15	91.5	10895360
compress.fcdf22.9	23	9	82.6	4816896
decompress.block_dequantize.9	10	5	70.0	983040
decompress.block_dequantize.16	10	4	100.0	393216
decompress.block_dequantize.18	14	4	100.0	655360
decompress.hufdec.2	6	2	100.0	8000004
decompress.hufdec.3	20	11	70.0	18000000
decompress.bcdf22.3	23	9	82.6	4816896
decompress.bcdf22.6	42	13	92.9	9873920
decompress.bcdf22.9	25	10	84.0	5160960

In Table 2, candidate-induced application speedups are given in columns 2, 3 for the 23 unique (non-isomorphic) complex instructions that were identified. Estimates of the implementation details for these instructions are shown in column 4.

Table 2. Candidate-induced speedups for the compress and decompress applications

Application name	compress	decompress	
Candidate instruction	% candidate-induced speedup	% candidate-induced speedup	Possible implementation
mla	3.43	0.68	Multi-cycle/pipelined
lod_add_lsl	26.65	17.07	Single-cycle
stri_lsl	1.70	0.91	Single-cycle
bne_imm	0.57	0.08	Single-cycle
beq_inc	1.14	n.a.	Single-cycle
stri_add	1.73	n.a.	Single-cycle
mul_lsl	0.76	0.40	Multi-cycle/pipelined
str_lsl	15.85	5.72	Single-cycle
lsl_mla	1.51	n.a.	Multi-cycle/pipelined
add_sub	0.75	n.a.	Single-cycle
lod_lsl_inc	3.00	1.58	Single-cycle

lod_lsl_dec	2.25	n.a.	Single-cycle
add_asr_add	1.50	n.a.	Single-cycle
add_inc_mul	1.51	0.80	Multi-cycle/pipelined
bge_imm	n.a.	2.32	Multi-cycle/pipelined
ldc_and_sl	n.a.	2.32	Single-cycle
and_sli	n.a.	4.63	Single-cycle
lsl_inc	n.a.	2.32	Single-cycle
str_addi	n.a.	4.63	Single-cycle
add_asr_sub	n.a.	0.79	Single-cycle
add_add	n.a.	0.39	Single-cycle
mul_add_lsl	n.a.	1.18	Multi-cycle/pipelined
lod_lsrv	n.a.	6.95	Single-cycle

For both applications, load and store instructions as *lod_add_lsl*, *str_add_lsl* and *lod_lsrv* implementing shifter-based addressing modes provide the most significant speedup while not requiring any change to the memory access scheme. The generation of such specialized addressing modes although currently disallowed in [13], could be safely accounted in their DFG explorer.

Figure 4 shows a portion of the generated templates. A restriction of maximum 3 input and 1 output register operands has been applied to encompass for single register file limitations that apply to a generic RISC. The majority of these templates, for example (*i*), (*ii*), (*iv*), could be implemented as single-cycle instructions since 3 or more arithmetic (excluding multiplication and division), logical or shift-by-immediate operations would fit in a single cycle [13]. Complex instruction (*iii*) incorporates the multiply operation which almost certainly affects the processor cycle time. A multi-cycle or pipelined realization for this instruction is worthy investigating. Multi-cycle instructions may be acceptable even though the processing throughput against using their primitive instruction sequence may not be improved, since power consumption related to instruction fetch is significantly reduced.

It is possible that instruction templates can be merged into superset instructions that would be served on the same hardware. For example, templates (*i*), (*ii*), and (*iv*), make use of up to 2 adder/subtractors and 1 shifter, so that assuming appropriate control, a single instruction for these could be implemented. If a load/store operation is also part of the instruction, it must be performed on a different pipeline stage.

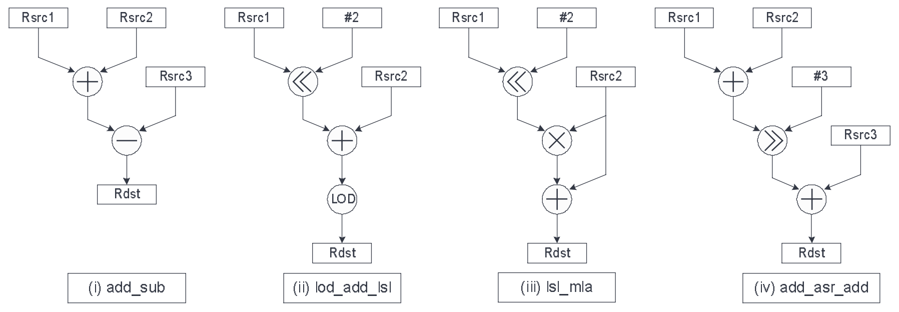

Fig. 4. Candidate instruction examples

5 Conclusions

In this paper, an application analysis flow is proposed for evaluating the characteristics of applications running on configurable processor platforms. For this reason, an open-source compiler infrastructure is utilized to develop our in-house tools. Novel application parameters are introduced in the scope of application characterization as the percentage cover, maximum basic block cycle and candidate-induced application speedup due to the introduction of instruction extensions. An initial set of complex instructions is also generated in order to be used as a starting point in design space exploration iterations. To show the potential of the presented approach, the Wavelet/Scalar Quantization image compression application is used as a case study.

References

1. Jacome, M.F., de Veciana, G.: Design challenges for new application-specific processors. IEEE Design and Test of Computers, Vol. 17, No. 2, (2000) 40-50
2. Gonzalez R.: Xtensa: A configurable and extensible processor. IEEE Micro, Vol. 20, No. 2, (2000) 60-70
3. Altera Nios: http://www.altera.com
4. Gaisler research: http://www.gaisler.com
5. Itoh, M., Higaki, S., Sato, J., Shiomi, A., Takeuchi, Y., Kitajima, A., Imai, M.: PEAS-III: An ASIP design environment. IEEE Int. Conf. on Computer Design, (2000) 430-436
6. A. Hoffmann et al.: A novel methodology for the design of application-specific instruction set processors (ASIPs) using a machine description language. IEEE Trans. on Computer-Aided Design, Vol. 20, No. 11, (2001) 1338-1354
7. Smith M.D., Holloway G.: An introduction to Machine SUIF and its portable libraries for analysis and optimization. Tech. Rpt., Division of Eng. and Applied Sciences, Harvard University, 2.02.07.15 edition, (2002)
8. Gupta T.V.K., Sharma P., Balakrishnan M., Malik S.: Processor evaluation in an embedded systems design environment., 13th Int. Conf. on VLSI Design, (2000) 98-103
9. Ghazal N., Newton R., Rabaey J.: Retargetable estimation scheme for DSP architecture selection. Asia and South Pacific Design Automation Conf., (2000) 485-489
10. Kim S., Somani A.K.: Characterization of an extended multimedia benchmark on a general purpose microprocessor architecture. Tech. Rpt DCNL-CA-2000-002, DCNL (2000)
11. Lee C., Potkonjak M., Mangione-Smith W.H.: MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. Proc. of the IEEE/ACM Symp. on Microarchitecture, (1997) 330-335
12. Leupers R., Wahlen O., Hohenauer M., Kogel T., Marwedel P.: An executable intermediate representation for retargetable compilation and high-level code optimization. Int. Wkshp. on Systems, Architectures, Modeling and Simulation (SAMOS), Samos, Greece (2003)
13. Clark N., Zhong H., Tang W., Mahlke S.: Automatic Design of Application Specific Instruction Set Extensions through Dataflow Graph Exploration. Int. Journal of Parallel Programming, Vol. 31, No. 6, (2003) 429-449
14. Hopper T., Brislawn C., Bradley J.: WSQ Greyscale Fingerprint Image Compression Specification, Version 2.0, Criminal Justice Information Services, FBI, Washington, DC (1993)
15. Kumar S., Pires L., Ponnuswamy S., Nanavati C., Golusky, Vojta M., Wadi S., Pandalai D., Spaanenburg H.: A benchmark suite for evaluating configurable computing systems – Status, reflections, and future directions. ACM Int. Symp. on FPGAs (2000)