Automatic Selection of Application-Specific Reconfigurable Processor Extensions

Christophe Wolinski, Krzysztof Kuchcinski

Automatic Selection of Application-Specific Reconfigurable Processor Extensions

Christophe Wolinski
University of Rennes I / IRISA, France
wolinski@irisa.fr

Krzysztof Kuchcinski
Dept. of Computer Science
Lund University, Sweden
krzysztof.kuchcinski@cs.lth.se

Abstract

This paper presents a new method for automatic selection of application-specific processor extensions and shows how applications are scheduled on these new reconfigurable architectures. The extensions are implemented as specialized sequential or parallel instructions. They correspond to identified most frequently occurring computational patterns or other interesting patterns and are finally selected during mapping and scheduling. Our methods can handle both time-constrained and resource-constrained scheduling. Experimental results show that the presented method provides high coverage of application graphs with small number of patterns and ensures high application execution speed-up both for sequential and parallel application execution with processor extensions implementing selected patterns.

1 Introduction

Embedded systems are built around a processor that provides flexibility for executing different applications at the cost of lower performance. Instruction sets provide general purpose instructions that can implement any functionality but cannot explore particular specialized parts of an algorithm that might contain very specific computational patterns. These application specific patterns need to be identified and some of them can later be implemented in order to speed up application's execution. These tasks are implemented in our system, called *UPaK* (Unified Pattern Based Synthesis Kernel) [15] and executed in two consecutive steps. In the first step we explore typical computational patterns and identify most useful ones for a given application. The identified computational patterns are then used in mapping and scheduling step where a subset of patterns is selected for implementation. This selection step follows timing and resource constraints in such a way that the necessary patterns are only selected. In this way we select different patterns for different applications and implement them as special statically or dynamically reconfigurable units connected to the main processor.

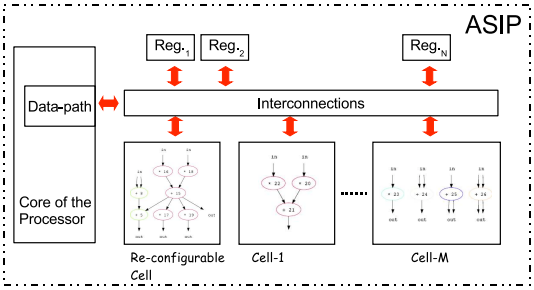

Figure 1. Generalized ASIP processor model.

Systematic methods for identification of frequent computational patterns (or other interesting patterns) and search for maximal coverage of a particular application graph with found patterns were already presented in [13, 14]. In this paper we focus on concurrent pattern selection from previously identified ones, mapping and application scheduling on a newly created processor. We define and solve both time- and resource-constrained scheduling.

Different architecture models can be explored within our framework. However, in this paper we consider an architecture model of an ASIP processor with extended instruction sets implemented as reconfigurable units. An instruction can be implemented to make it possible sequential or parallel execution of computational patterns in respect to ASIP core processor instructions (see section 5.1). This provides ways to trade-off execution time against hardware cost. Our generic simplified architecture is depicted in Figure 1. It is composed of heterogeneous cells and registers connected by an interconnection structure with the processor's datapath. The number of registers and the structure of interconnections are application-dependent. Each cell implements one or more patterns selected by *UPaK* system. The registers store intermediate results that reduces data transfers between our architecture extension and a processor register file.

978-3-9810801-3-1/DATE08 © 2008 EDAA

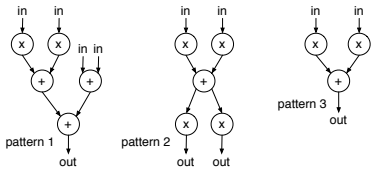

Figure 2. Set of patterns corresponding to new instructions.

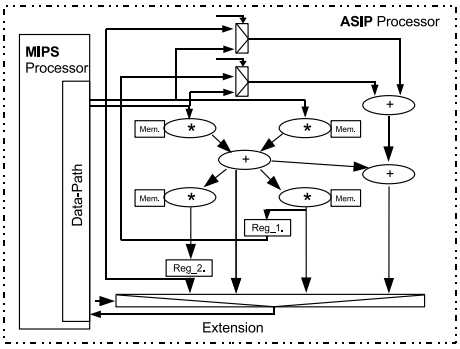

Figure 3. An example of ASIP processor build around MIPS processor core for ARF application.

For example, consider an extension of a MIPS processor for an auto-regression-filter (ARF) application. Our method selected three computational patterns, depicted in Figure 2, for processor instruction extension. These patterns are then merged and synthesized to create a specialized computational dynamically reconfigurable cell architecture. This highly optimized architecture is depicted in Figure 3 where MIPS processor is extended with a specialized data-path. Patterns are executed sequentially in this case.

The rest of this paper is organized as follows. In section 2 we discuss related work on pattern identification and scheduling. Constraint programming basics and the graph matching constraint that are used in our approach are discussed in sections 3. Sections 4 and 5 present pattern generation process as well as scheduling and match selection. Finally experimental results are presented in section 6 and conclusions in section 7.

2 Related Work

Previous research on pattern extraction, such as [7, 1, 4], is characterized by combined pattern matching and pattern generation for ASIPs. In [7], this is achieved with clustering that uses information on frequency of node type successions. Authors of [1] and [4] use an incremental clustering that uses different heuristic approaches with the common aim of identifying frequently occurring patterns.

Another method is presented in [2] where the pattern searching algorithm identifies a big pattern using convexity and input/output constraints. Some improvements of this method were proposed in [3]. Pattern searching under input/output constraints is also used in [11]. The basic algorithm starts from each exit node of the basic block and constructs a sub-graph by recursively trying to include parent nodes. The assembled sub-graph is considered as a potential new instruction. The quality of this instruction is then determined by their system. In [5], a set of Multiple Input Single Output sub-graphs (MaxMISO) is identified first. Each MaxMISO sub-graph is not contained in any other MISO sub-graph. In the next step a candidate set composed of two-input/one-output MISOs found inside the MaxMISO set is selected. Finally, using the selected candidates the application graph is partitioned by a nearly-exhaustive search method using the branch-and-bound algorithm. Recently, in [10], a complete processor customization flow was presented where patterns are clustered, one after the other, making some local decisions.

Our approach resembles the method presented in [6]. Patterns are incrementally assembled by adding the neighbor nodes to existing matches corresponding to non isomorphic patterns formed in the previous iteration. The difference is the selection of neighbor nodes and new potential patterns. In particular, our approach applies smart filtering of patterns. It decides which new potential pattern is "useful" and can be later extended. The smart filtering uses information derived by a special method that is based on sub-graph isomorphism constraints and constraints programming that is also radically different from the approach proposed in [6].

Pattern selection, binding and scheduling are computationally difficult problems and therefore most researchers use heuristic approaches, such as greedy algorithms, simulated annealing, genetic algorithms and tabu search. Recently several interesting approaches have been proposed. Wang et.al. uses *ACO* (Ant Colony Optimization) algorithm [12] and Guo et.al. [6] a heuristic algorithm based on maximum independent set of a conflict graph. Our approach is different. The pattern selection, binding and scheduling problem is completely defined using a constraint model. We use our graph matching constraint together with other binding and scheduling constraints.Then the problem can be solved using either complete or heuristics methods.

3 Background

In our work we use extensively constraint satisfaction methods implemented in constraint programming environment JaCoP [8].

A *constraint satisfaction problem* is defined as a 3-tuple $\mathcal{S} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$ is a *set of variables*, $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_n\}$ is a set of *finite domains* (FD), and $\mathcal{C}$ is a set of *constraints*. Finite domain variables (FDV) are defined by their domains, i.e. the values that are possible for them. A finite domain is usually expressed using integers, for example $x :: 1..7$. A constraint $c(x_1, x_2, \ldots, x_n) \in \mathcal{C}$ among variables of $\mathcal{V}$ is a subset of $D_1 \times D_2 \times \ldots \times D_n$ that restricts which combinations of values the variables can simultaneously take. Equations, inequalities and even programs can define a constraint.

In this paper we use intensively two combinatorial constraints `Diff` and `GraphMatch`. `Diff` constraint takes as an argument a list of 2-dimensional rectangles and assures that for each pair of i, j $(i \neq j)$ of 2-dimensional rectangles, there exist at least one dimension k where i is after j or j is after i. The 2-dimensional rectangle is defined by a tuple $[O_1, O_2, L_1, L_2]$, where O_i and L_i are respectively called the origin and the length of the 2-dimensional rectangle in *i-th* dimension. The `Diff` constraint is used in this paper for defining constraints for scheduling and resource binding. Graph matching constraint (`GraphMatch`) defines conditions for (sub-)graph isomorphism between target and pattern graphs (the pattern graph can be defined as a set of separate sub-graphs). It has been implemented using a pruning algorithm developed for this special purpose [14].

A *solution to a CSP* is an assignment of a value from variable's domain to every variable, in such a way that all constraints are satisfied. The specific problem to be modeled will determine whether we need *just one solution*, *all solutions* or an *optimal solution* given some cost function defined in terms of the variables.

The solver is built using constraints own consistency methods and systematic search procedures. *Consistency methods* try to remove inconsistent values from the domains in order to reach a set of pruned domains such that their combinations are valid solutions. Each time a value is removed from a FD, all the constraints that contain that variable are revised. Most consistency techniques are not complete and the solver needs to explore the remaining domains for a solution using search.

Solutions to a CSP are usually found by systematically assigning values form variables domains to the variables. It is implemented as depth-first-search. The consistency method is called as soon as the domains of the variables for a given constraint are pruned. If a partial solution violates any of the constraints, backtracking will take place, reducing the size of the remaining search space.

4 Pattern Generation Process

Pattern generation is an iterative process (for details see [13]) where in each iteration larger patterns (incremented by one node each iteration) are explored and candidates for inclusion in a set of patterns are identified. The pattern generation flow is depicted in Figure 4. Each iteration of this algorithm results in the generation of a *Next Pattern Set* (*NPS*). The algorithm stops when the *end condition* is reached. In our experiments the algorithm stops when the pattern size becomes K but other stop criteria are possible.

The inputs to the pattern searching algorithm are: Definitely Identified Pattern Set (*DIPS*), Current Pattern Set (*CPS*) and Architecture. Initially, *DIPS* and *CPS* sets contain the same set of one-node patterns [13].

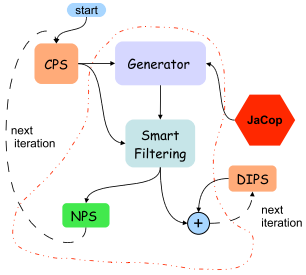

Figure 4. Pattern Generation Flow.

The pattern searching algorithm searches for new patterns that can be synthesized from all possible matches in application graph *AG* corresponding to each pattern in *CPS*. A new temporary pattern tp is created from current match $m_j^{p_i}$ of the pattern $p_i \in CPS$ and the selected node $n_i \in N$ where N is a set of the nodes directly connected to $m_j^{p_i}$ in *AG* . The search is organized both upward and downward to best explore new patterns. A new pattern tp is accepted if it is not isomorphic to any of already found patterns.

A smart filtering then decides whether pattern tp should be saved in set *NPS* or not. It also determines inclusion of current pattern p_i to the *DIPS*. This filtering process uses information about the number of matches in *AG* corresponding to patterns p_i and tp. When all patterns from the *CPS* set has been processed the pattern generation algorithm stops the current iteration by copying the *NPS* content to the *CPS*.

5 Scheduling and Pattern Selection

The match selection, component binding and scheduling problems are solved for the first time concurrently using constraints programming methods. The input to the system consists of an application graph *AG*, a target architecture model and a selected pattern set *DIPS* obtained during the pattern searching process. Graph *AG* is used to generate operation precedence constraints and dependence requirements while the architecture model and pattern set *DIPS* are used to find actual matchings. These constraints are defined as inequalities and specialized constraints as specified in

section 5.2. The constraint solving technique is then used to find an optimal or suboptimal solution which satisfies the given constraints and optimizes a given cost function. We use a branch-and-bound (*B&B*) algorithm to find the pattern selection and the schedule. Our prototype system has been implemented using the JaCoP solver [8]. We model both time-constrained and resource-constrained scheduling.

5.1 Architecture Model

Our system supports a very general abstract architecture composed of interconnected cells where each cell can execute a pattern or a set of patterns. When a cell contains a set of patterns then the patterns are merged to form a run-time reconfigurable data-path as depicted in Figure 3 where the data-path is synthesized from the three patterns as shown in Figure 2. In general, all of the abstract architecture's cells can work concurrently with each other but they can use only one pattern at a time. In the case of an *ASIP* processor, each identified computational pattern can be assigned to a separate instruction and executed sequentially with all other patterns. The corresponding abstract architecture is reduced to one cell containing the entire *DIPS* set of patterns. The resulting synthesized architecture has a structure similar to one depicted in Figure 3. One can also combine a number of patterns with similar delays and execute them in parallel using a single instruction. In this case an abstract architecture is composed of one cell containing all one-node patterns corresponding to the original instruction set and a set of other cells containing different patterns from the remaining patterns in the *DIPS* set. The resulting synthesized architecture is similar to the one depicted in Figure 1. Our framework can also manage many different cells containing the same pattern in order to speed-up application execution. In this paper, we consider both sequential and parallel cases and report the related experimental results.

5.2 Scheduling and Pattern Selection Modeling

The inputs of our framework are the application graph $AG = (V, E)$ (where V is a set of vertexes and E is a set of edges), the *DIPS* set and the abstract architecture model . During the selection and scheduling processes, the system is looking for a sub-graphs of the *AG* corresponding to the pattern graphs from the *DIPS* set. These sub-graphs are called *matches* of pattern graphs because they match these patterns. Some of the patterns from the *DIPS* set whose matches have been selected during the optimization process are later implemented by computational cells in order to build a target architecture. In the case where a cell contains many implemented patterns, each pattern must be used exclusively. Moreover, the final system implementation can use different numbers of copies of the selected cells. The defined constraints must follow this requirements.

We first define FDVs which are used in our scheduling problem. We use variable T to denote the start time of a given node. The subscripts are used to identify related nodes. For example, T_i denotes a start time of node $i \in V$.

Precedence constraints: The node dependencies in *AG* are defined by the inequalities as follows.

$$\forall_{(i,j)\in E}\ T_i + D_i \leq T_j \wedge D_i \in DS \tag{1}$$

where *DS* is a set of delay times corresponding to all cells used in the system. A delay of node i can vary since it can be a part of different matches. This is handled by other constraints defining mapping between selected cell and its delay.

Graph Matching: We simply use our graph matching constraint to identify matches in graph *AG* corresponding to the patterns from the *DIPS* set.

$$\texttt{GraphMatch}(AG, DIPS) \tag{2}$$

This constraint implies that only a limited number of matches can coexist in *AG*. The existing matches, represented by FDVs $m_0, \ldots, m_n$, are used in the following constraints for imposing timing and resource constraints.

Match timing modeling: In our architecture model it is required that all input signals of a match are available before its cell starts execution. To model this we divide nodes of each match of a pattern into a set of input nodes V_{in} and a set of non-input nodes V_{int}. We also introduce a new set of dummy nodes V_d. The input node has no input edges connected to other nodes in the same match. Non-input nodes have inputs from other nodes in the match but they can have a number of non-connected input edges as well. Dummy nodes are added, in such cases, to assure the right timing for input signals. The following constraints are defined for nodes identified for match m_k.

$$\begin{aligned}
&\forall_{i\in V_{in}, j\in V_d}\ T_i = T_j = T_{m_k} \\
&\forall_{i\in V_{in}, j\in V_d}\ D_i = D_j = D_{m_k} = D_{Cell_k} \\
&\forall_{i\in V_{int}}\ D_i = 0
\end{aligned} \tag{3}$$

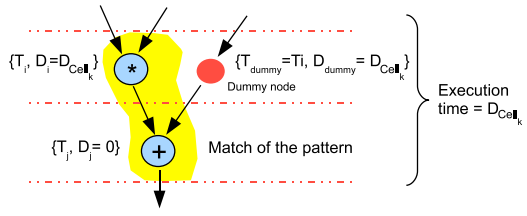

Figure 5. Match timing modeling.

Delay D_{Cell_k} is the time needed by the $Cell_k$ to execute a selected match. Thanks to the constraints described by

equations (3) the match is considered by the remaining *AG* nodes as a separate unit executed at time T_{m_k} with delay D_{m_k}. The delay from any of the inputs or dummy nodes to any of the outputs is constant and is therefore equal to D_{m_k}. The constraints rule-out also all non-convex matches.

Resource Constraints: If two or more matches in *AG* graph can be implemented using the same cell type they must either use different physical cells or their executions must not overlap. We model execution of a match on a cell through the rectangle as it is often done in scheduling (Figure 6) and use `Diff` constraints. This constraint assures that two dimensional rectangles do not overlap that fits our purpose. For each set of matches executed on a given cell q we impose `Diff` constraint. Below a formulation for three matches m_i, m_j and m_k executed on cell type q is presented. Figure 6 illustrates this constraint.

$$\begin{aligned}&\texttt{Diff}([T_{m_i}, Cell_{qi}, D_{m_i}, 1],\\ &\quad [T_{m_j}, Cell_{qj}, D_{m_j}, 1], [T_{m_k}, Cell_{qk}, D_{m_k}, 1])\end{aligned} \tag{4}$$

Domains for variables $Cell_{qi}$,$Cell_{qj}$ and $Cell_{qk}$ are defined as $\{1 \ldots |Cell_l|\}$ indicating possible selection of a cell numbered from 1 to $|Cell_l|$ for execution of each match.

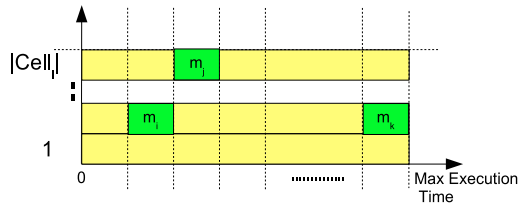

Figure 6. Modeling of resource constraints.

Cost for resource-constrained scheduling: In resource-constrained scheduling the solver minimizes the schedule length under given resource constraints. Therefore we defined our cost function for minimization as the latest completion time (LCT) for nodes of graph *AG*. The constraint can be simplified by defining it for nodes that are last to execute. Minimization of LCT will provide the shortest schedule. Our cost function is defined below.

$$LCT = max(T_0 + D_0, \ldots, T_n + D_n) \tag{5}$$

where T_i and D_i are respectively the start time and delay of node i.

Cost for time-constrained scheduling: In this scheduling we minimize the number of different cells used in an implementation and therefore we defined our cost function as follows.

$$Resources = \sum_k \begin{cases} 1 & \text{if exists } m_i \text{ that uses } Cell_k \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Time-constrained scheduling is done under execution time constraint defined as follows.

$$LCT \leq ExecTimeLimit \tag{7}$$

6 Experimental Results

We have carried out extensive experiments to evaluate the quality of generated patterns as well as possible speed-up when implementing them as specialized instructions. We have used DSP applications form the MediaBench test suite [9]. All experiments have been run on 2GHz Intel Core Duo under Mac OSX operating system.

The quality of the graph coverage with identified patterns and its comparison to other results is discussed later. Up to our knowledge the direct comparison of our results on obtained speed-up was not possible due to the lack of existing published results. However, in many cases our framework can prove the optimality of the obtained results.

Table 1 presents the results obtained for the benchmark set. We present the number of patterns identified by our algorithm for the original graph (step 1) as well as for a graph obtained from this graph after removing all found matches of patterns (step 2). We also present the number of patterns that are actually selected for maximum coverage of the graph as well as the graph coverage for step 1 and the total coverage.

To explore possible speed-ups we consider two scheduling choices for a selected set of computational patterns. The first scheduling method schedules all matches and nodes assuming sequential execution of all operations while parallel scheduling method assumes most parallel execution. Both methods are compared to a sequential execution of the original graph. Speed-up is presented for both steps. We consider that each match (up to 7 nodes) can be executed during one clock cycle by the corresponding cell. The clock cycle is determined for the MIPS processor used as the ASIP's processor core (Figure 1) and implemented on Altera Cyclon II FPGA devices running at 25 MHz.

The total average coverage obtained with our method is 89.1% and the average coverage after the first step 74.1% while the number of patterns identified by our algorithm is not very large. It is on average 3.4 for step 1 and 5.1 for both steps. The authors of [7], for example, obtained similar coverage for these graphs but used larger number of patterns (on average 21.9 patterns to reach a coverage of 83% of nodes).The speed-up obtained after scheduling applications with patterns found in both steps is high and it is on average 3.17 for sequential execution and 7.26 for parallel one. Obviously different trade-offs between execution time and hardware cost are further possible.

Table 1. Results for patterns of 7 nodes limit.

Application	$\|V\|$	time	step 1			speed-up		step 2		total	speed-up	
		(s)	id.	sel.	cov.	Seq.	Par.	id.	sel.	cov.	Seq.	Par.
JPEG Write BMP Header	106	4.99	6	4	90%	2.83	10.60	3	2	96%	3.18	17.60
JPEG Smooth Downsample	51	3.36	8	4	74%	1.96	3.70	3	1	88%	2.55	6.30
JPEG IDCT	134	20.90	7	5	69%	2.03	2.48	3	2	81%	2.44	5.10
MPEG IDCT	114	5.30	3	2	54%	1.78	2.03	6	3	71%	2.32	3.25
MPEG Motion Vector	32	0.9	8	3	93%	4.57	8.00	1	1	100%	4.65	10.60
EPIC Collapse	56	2.09	7	4	69%	2.24	2.64	3	3	85%	2.94	3.07
MESA Smooth Triangle	197	120.00	7	3	73%	1.68	3.20	2	1	87%	2.11	4.58
MESA Horner Bezier	18	0.36	9	3	83%	3.00	6.00	1	1	94%	3.60	6.00
MESA Interpolate Aux	108	22.8	3	2	85%	3.37	6.30	3	1	100%	5.40	21.00
MESA Matrix Multiplication	109	28.4	7	4	56%	1.57	2.80	4	2	86%	2.42	3.89
MESA Feedback Points	53	1.70	4	2	80%	2.73	5.00	3	3	94%	3.46	8.30
FIR	44	12.5	7	4	72%	2.44	7.30	2	2	90%	3.66	7.30
Elliptic Wave Filter	34	2.20	9	4	67%	2.01	2.60	2	2	94%	3.09	3.40
Auto Regression Filter	28	3.30	4	3	96%	3.50	5.60	-	-	-	-	-
Cosine	66	7.05	8	3	50%	1.57	1.70	7	3	74%	2.27	3.00

7 Conclusions

In this paper we have presented a radically new approach to automatic selection of application-dependent processor extensions and we have shown how applications are mapped and scheduled on these new architectures. The important novelty is that the pattern selection, mapping and scheduling are executed concurrently. In the other approaches these steps are usually done after pattern selection. This has been possible since we have been developed a method that uses sub-graph matching constraints combined with scheduling constraints in one constraint programming formulation.

Our experiments show the good quality of the generated patterns. We achieve high coverage for application graphs with a small number of patterns compared to other approaches making possible better hardware optimization for pattern implementation. We have also obtained significant speed-ups for MediaBench applications with identified computational patterns.

References

[1] M. Arnold and H. Corporaal. Designing domain specific processors. In *Proceedings of the 9th International Workshop on Hardware/-Software CoDesign*, pages 61–66, Copenhagen, April 2001.

[2] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instructionset extensions under microarchitectural constraints. In *40th Design Automation Conference (DAC)*, 2003.

[3] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne. ISEGEN: Generation of high-quality instruction set extensions by iterative improvement. In *42nd Design Automation Conference (DAC)*, 2005.

[4] N. Clark, H. Zong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *36th Annual International Symposium on Microarchitecture*, 2003.

[5] Y. Guo. Mapping applications to a coarse-grained reconfigurable architecture. In *PhD Thesis, University of Twent, Eindhoven, Netherlad*, Sept. 8, 2006.

[6] Y. Guo, G. Smit, H. Broersma, and P. Heysters. A graph covering algorithm for a coarse grain reconfigurable system. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, San Diego, California, June 11-13, 2003.

[7] R. Kastner, A. Kaplan, S. O. Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Trans. Des. Autom. Electron. Syst.*, 7(4), 2002.

[8] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8(3):355–383, July 2003.

[9] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[10] R. Leupers, K. Karuri, S. Kraemer, and M. Pandey. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In *DATE*, 2006.

[11] A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli. Automatic instruction set extension and utilization for embedded processors. In *ASAP*, 2003.

[12] G. Wang, W. Gong, and R. Kastner. System level partitioning for programmable platforms using the ant colony optimization. In *International Workshop on Logic & Synthesis (IWLS'04)*, Temecula, California, June 2-4, 2004.

[13] C. Wolinski and K. Kuchcinski. Computation patterns identification for instruction set extensions implemented as reconfigurable hardware. In *ERSA'2007, The International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, Nevada, USA, June 25-28, 2007.

[14] C. Wolinski and K. Kuchcinski. Identification of application specific instructions based on sub-graph isomorphism constraints. In *IEEE 18th Intl. Conference on Application-specific Systems, Architectures and Processors*, Montréal, Canada, July 8-11, 2007.

[15] C. Wolinski, K. Kuchcinski, and A. Postula. UPaK: Abstract unified pattern based synthesis kernel for hardware and software systems. In *Materials of the University Booth at DATE 2007*, Nice, France, Apr.16-20, 2007.