Virtual Hardware Byte Code as a Design Platform for Reconfigurable Embedded Systems

Sebastian Lange, Udo Kebschull
{lange, kebschull}@ti-leipzig.de
Department of Technical Computer Science
University of Leipzig
Augustusplatz 11, 04103 Leipzig, Germany

Abstract

Reconfigurable hardware will be used in many future embedded applications. Since most of these embedded systems will be temporarily or permanently connected to a network, the possibility to reload parts of the application at run time arises. In the 90ies it was recognized, that the huge variety of processors would lead to a tremendous amount of binaries for the same piece of software. For the hardware parts of an embedded system, the situation today is even worse. The java approach based on a java virtual machine (JVM) was invented to solve the problem for software. In this paper, we show how the hardware parts of an embedded system can be implemented in a hardware byte code, which can be interpreted using a virtual hardware machine running on an arbitrary FPGA. Our results show that this approach is feasible and that it leads to fast, portable and reconfigurable designs, which run on any programmable target architecture.

1. Introduction

With a widespread use of reconfigurable hardware such as FPGAs and PLDs in devices as well as the interconnection between them through networks new possibilities and demands arise. It is now possible to not only change software components at runtime, but also the hardware logic itself. This can be done at the customer site without the need of physical access to the device, possibly even without an explicit interaction with the user. In order to allow for a multitude of devices to interact with each other, hardware components should be interchangeable in a way that is abstracting from the specific design of the underlying hardware architecture. However, due to the lack of a standardized interface to the FPGAs and other common programmable devices, it is problematic to define logic in a general and portable fashion.

Reconfiguration in hardware is synonymous with the incorporation of FPGAs in the design. Unfortunately, the configuration file formats of different FPGAs, which represent the very essence of reconfigurable hardware, differ greatly and prove incompatible. The incompatibilities are not only vendor induced, but moreover root in the different physical layouts of FPGAs. It is therefore not foreseeable that a standardized, compatible format will be available in the future, leaving reconfigurable hardware in the same dilemma, software was in a decade ago. At that time Java [1] was invented to eliminate the compatibility issues caused by the variety of incompatible processor platforms. Ever since its introduction the Java concept has been inarguably successful. Reconfigurable hardware faces the same principle problem as software did. In consequence the Java approach of abstracting software from the underlying processor platform should be applied to reconfigurable hardware. One such solution will be presented in the remainder of this paper.

The approach of the Virtual Hardware Byte Code (VHBC) as described in this paper provides a means to combine the virtues of hardware design with the flexibility inherent in software. Algorithms are designed as hardware but mapped to a special hardware architecture called the Virtual Hardware Machine (VHM) by a dedicated hardware byte code compiler. The VHM is designed to be easily implemented on virtually any underlying hardware architecture and acts as a mediator between an abstract algorithmic description and the hardware itself.

Another important aspect of a design platform for embedded systems is the driving need for low power devices. With the advent of wireless communication devices and hand-held computers (PDA's, cellular phones) a new door has been opened towards computing power in the light of information whenever and wherever it is needed. Fast execution is paramount to keep up with new ever more involved algorithms. However, this when solved entirely in software on state of the art all-purpose processing elements requires ever higher clock frequencies which puts a heavy burden on the energy resources of any hand-held device or embedded system. Using dedicated hardware resources effectively increases the computational power as well as lowers the power consumption, but this comes at the price of very limited flexibility. The VHBC is specifically designed to repre-

1530-1591/03 $17.00

sent hardware. Designs mapped to the VHM will thus run more effectively than corresponding solutions in pure software on general purpose processors, yielding higher performance and lower power consumption.

1.1. State of the Art

At present, the design of a new embedded system usually leads to the deployment of two coexisting design platforms. On one hand, functionality is cast into hardware by means of ASIC design. ASICs provide the highest performance and the lowest power consumption. However, they do not account for later changes in functionality demands making reconfigurability impossible. Moreover, the cost of designing ASICs is skyrocketing. The production cost for a single mask set alone can easily consume well over $1 million. Furthermore manufacturing techniques have ventured deep into the fields of sub-micron physics, introducing effects unaccounted for in years passed, such as signal integrity, power leakage or electromigration, which in itself makes it harder to design working chips and thus extends the amount of money as well as time spent on designing ASICs. A detailed discussion of these effects, however, is far beyond the scope of this paper and is discussed in detail in the literature [2]. The other design platform mentioned is software. Software, being an abstract description of functionality is inherently exchangeable. It allows not only to account for changing demands, which greatly shortens design cycles, but provides also means to transfer functionality among systems. Furthermore, processors are applied "off the shelf", thus being cost-effective and thoroughly tested. Yet a lot of different processor platforms exist, leading towards compatibility problems and a multitude of equivalent implementations of the same application. Far worse, processors prove to be very power hungry and very complex devices wasting a lot of available computing power because they are designed to be general purpose, but usually only perform very specific tasks when employed in embedded systems.

In conjunction, neither software nor ASIC design can sufficiently solve the problems of modern design challenges. What is needed, is a way to combine the virtues of both design platforms, namely reconfigurability and short design cycles coupled with fast execution and low power consumption, thus making hardware "virtual" [3]. FPGAs were a very important step towards Virtual Hardware, because they offer close resemblance of the performance of custom-built circuits, yet still provide for changing functionality through reconfiguration. However, FPGAs show several disadvantages which prevent them from being the ideal technology for Virtual Hardware. Most predominantly, FPGAs do not share a common, standardized way of describing hardware, but rather differ greatly in the layout and format of their bit file descriptions. Furthermore, FPGAs impose harsh limitations towards the size of the Virtual Hardware designs. Although the number of logical gates that can be fit on a FPGA is increasing, it is still too small for a lot of real world designs to be implemented entirely on an FPGA. As another aspect, the time needed to reconfigure a whole FPGA, well lies in the range of several milliseconds to seconds, proving too long for applications which change dynamically during execution. The introduction of partial reconfiguration has helped alleviate the problem, but in consequence leads to a coarse grain place and route - process to bind the partial designs to available resources within the FPGA, which has to be done at runtime on the chip. This however, adds complexity to the surrounding systems, because they have to accommodate for the place and route logic.

Several proposals have been made addressing different shortcomings of current FPGAs. The "Hardware Virtual Machine" [4] [5] project, lead by Hugo de Man at the K.U. Leuven, gives attention to the problem of incompatible bit files, proposing the definition of an abstract FPGA which provides the essence of FPGAs. Designs should be mapped onto an abstract FPGA and placed and routed into small fractions. The so mapped and routed fragments pose an abstract yet portable representation of the design. In order to allow specific host FPGAs to make use of it, a Hardware Virtual Machine converts the abstract representation to bit files specific to the FPGA and reconfigures it. This approach has the advantage of running the actual design on FPGAs, thus providing high performance. The conversion itself, however, involves a placing of the hardware fragments as well as a routing of signals between them on the FPGA. This requires considerable computational effort and has to be done at design loading time. Furthermore, FPGA architectures differ greatly and their common ground might be too small to account for an efficient representation of Virtual Hardware and allow for great efficiency in the resulting specific bit files.

Another project, "PipeRench" [6] at CMU in Pittsburgh, addresses the spacial limitation imposed by FPGA design. The basic idea is to identify pipeline stages within a given design. Each stage is then treated as a single block of functionality, that is swapped in and out of the FPGA as needed. The control over the reconfiguration process is given to a special purpose processor that transfers the stages from an external memory to the FPGA and vice versa. The project assumes that the reconfiguration of a single pipeline stage can be done with great speed, allowing for stages to execute while others are reconfiguring. The results of this project present a possibility to design Virtual Hardware without the need to consider the spatial extend of the underlying reconfigurable hardware substrate, while also enabling designers to incorporate dynamic reconfiguration code changes into the design, thus clearing the way towards devices that change the functionality provided within the device ad hoc at user request. The pitfalls of the PipeRench approach clearly lie within its requirement of special FPGAs with extremely small reconfiguration delays, as well as the use of a control processor which turns out to be very complex [7]. Furthermore the architecture provides only limited support for interstage feed-back data flow, thus restricting the domain of applications.

1.2. Our approach

Our approach is different in that it defines means of implementing the hardware part of an embedded system as an intermediate hardware byte code. This byte code simply describes the hardware net list on register transfer or logic level. It is compacted and contains all the parallelism of the underlying hardware, which makes it run much faster than a description of the same functionality in software. A Virtual Hardware Machine (VHM) interprets the byte code. The VHM is implemented in VHDL and can therefore be mapped to any target FPGA or even into a CMOS implementation.

The following sections describe the general concept of the Virtual Hardware Byte Code and the Virtual Hardware Machine. To this point a first implementation of the VHM is available, which, however, supports only a subset of all possible features. The byte code reflects this simplification by providing instructions only on the logic level. These restrictions were imposed to allow for a first evaluation of the concept. Nevertheless, it should be evident that the concept is not limited to the current implementation and will in fact be easily adaptable to descriptions on register transfer level, yielding even higher performance.

2. The Virtual Hardware Byte Code

The definition of a coding scheme for virtual hardware reveals several important aspects. The most predominant ones are generality and high performance, which unfortunately proof to be quite orthogonal in this matter. Generality for one demands that the vast majority of circuits may be feasible within the execution model of the code while also calling for the highest level of abstraction of the code from the underlying hardware substrate. Performance, on the other hand, requests the code to be as close to the hardware as possible in order to minimize the overhead induced by emulation on a hardware implementation. Another important aspect is that hardware reveals a very high level of parallelism. This inherent parallelism should map to instruction level parallelism in the code in order to increase performance.

With these goals in mind the Virtual Hardware Byte Code has been defined with a number of features. In order to support a majority of circuits the available instructions mimic basic logic operations. This also yields a close resemblance of the code to traditional hardware concepts and thus allows for great efficiency in the actual implementation of an interpreting hardware processor (Virtual Hardware Machine). To account for high performance and alleviate the runtime system from the involved task of extracting instruction level parallelism at runtime the virtual byte code image is pre-scheduled into blocks of instructions which are independent of each other. Hence, all instructions belonging to the same block can be executed in parallel without side effects resulting from data dependencies.

The Virtual Hardware Byte Code uses a register transfer model to represent functionality. Instructions have a fixed format and consist of an operation code, two source registers and a destination register. The size of all registers is defined as one bit. The instructions, which are available to manipulate register contents represent basic logic operations as well as data movement instructions. A short overview of the available instruction set is given in table 1.

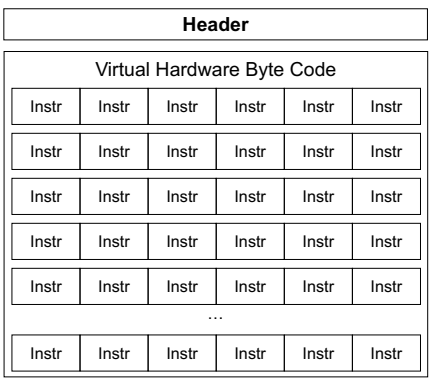

Figure 1. Structure of VHBC image

The byte code is structured hierarchically. At the topmost level it consists of two major parts, the header section, which provides general information about the circuit embedded in the image as well as information necessary to adapt it to the specific implementation of the Virtual Hardware Machine. The Virtual Hardware itself is encoded in the second part and consists of an arbitrary number of instructions grouped into code blocks. In order to illustrate the rationale behind the available instruction set the `CMOV` instruction will be described in detail.

OpCode	Mnemonic	Mathematical Expression
0000_2	EOB	N/A
0001_2	NOP	$c = c$
0010_2	MOV	$c = a$
0011_2	NOT	$c = \overline{a}$
0100_2	AND	$c = a \wedge b$
0101_2	NAND	$c = \overline{a \wedge b}$
0110_2	OR	$c = a \vee b$
0111_2	NOR	$c = \overline{a \vee b}$
1000_2	XOR	$c = (\overline{a} \wedge b) \vee (a \wedge \overline{b})$
1001_2	EQ	$c = (a \wedge b) \vee (\overline{a} \wedge \overline{b})$
1010_2	IMP	$c = (a \wedge b) \vee \overline{a}$
1010_2	NIMP	$c = a \wedge \overline{b}$
1100_2	CMOV	$c = \begin{cases} b & \text{if } a \text{ true} \\ c & \text{otherwise} \end{cases}$

Table 1. Instruction set of the VHBC

The `CMOV` operation (short for Conditional MOVe) is, as the name suggests, a conditional operation. It moves the content of the second source register to the output register

if and only if the value contained in the first source register is not zero. It thus sets itself apart from all other instructions in the instruction set, because the condition introduces a certain amount of control flow to the code and opens the possibility to incorporate a dynamic optimization scheme similar to branch prediction. Conceptually, the `CMOV` operation takes two data flows, the one which led to the content already stored in the output register and the one which contributes to the second input, and chooses among them according to the condition in the first input register, thereby discarding the result of one data flow. The instructions contained in the data flow which was discarded were thus superfluously computed. Given that the same condition holds in the next cycle it is possible to speculatively prune that data path.

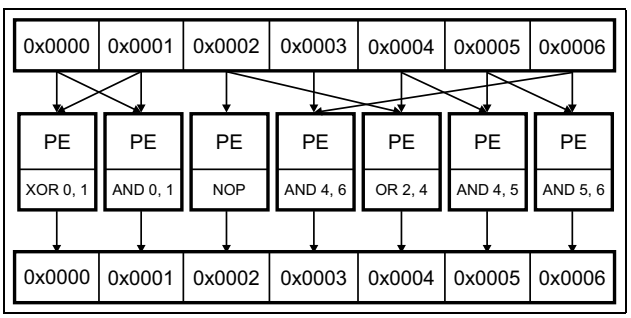

Figure 2. Instruction Mapping

The Virtual Hardware Byte Code defines every position in a code block to be available for every possible type of instruction of the instruction set. Therefore no special positions or slots exist for certain types of instructions. This leaves a degree of freedom in the placement of instructions, which is used to implicitly encode the address of the output register and thereby saves the space otherwise used to explicitly declare it. The address of the destination register is given by the position of the instruction in its code block starting to count at 0. If for example an instruction appears on the third position in the code block its output register address will be 2. Figure 2 illustrates how instructions are mapped to output register addresses.

3. The Byte Code Compiler

The Byte Code Compiler is a very important feature of the VHBC approach, because it provides the means to compile working hardware designs, coded as a VHDL description, into a portable and efficient VHBC representation, thus removing the need for redesigning working hardware projects.The tool flow within the VHDL compiler can basically be divided into three main stages, the hardware synthesis, the net list to byte code conversion and the byte code optimization and scheduling.

In the first stage, the VHDL description is compiled into a net list and standard logic optimization is performed upon it, resulting in an optimized net list. The net list is then mapped to the components contained within the SimPrim library from Xilinx. The resulting output of the first stage is converted to structural VHDL and passed on to the second stage. Most standard industry VHDL compilers with a support for Xilinx FPGAs readily provide the functionality needed for this step and can therefore be applied. Current implementations of the VHDL compiler make use of the FPGAExpress tool from Synopsis.

In the second stage, VHBC fragments substitute the components of the SimPrim library and form a VHBC instruction stream. Before, however, the components are mapped to a VHBC representation, the net list is analyzed and optimized for VHBC. The optimization is necessary because commercial compilers targeting FPGAs usually output designs which contain large amounts of buffers to enhance signal integrity otherwise impaired by the routing of the signals. Furthermore, compilers show a tendency towards employing logic representations based on NAND or NOR gates, which are more efficient when cast into silicon. However, the resulting logic structure is more complex, revealing higher levels of logic. The code fragments used for substituting the logic components are based on predefined, general implementations of the latter in VHBC and are adjusted according to the data flow found in the structural description from the first phase, thus registers are allocated and the instructions are sequenced according to the data dependencies inherent.

In the third stage, the byte code sequence is optimized and scheduled into blocks of independent instructions. First of all, the data flow graph of the entire design is constructed, which is possible due to the lack of control flow instructions such as jumps. The code fragments introduced in the second stage are very general, so the resulting code gives a lot of room to code optimization techniques. One such technique is dead code elimination, which removes unnecessary instructions. The code is further optimized by applying predefined code substitution rules along the data paths, such as XOR extraction or negation removal, to reduce the number of instructions and compact the code.

The thus optimized code is scheduled using a list based scheduling scheme [15]. The objective of the scheduling is to group the instructions into code blocks such that the number of code blocks is minimal and the number of instructions per code block is evenly distributed among all code blocks. Furthermore, the time of data not being used, i.e. the number of clock cycles between the calculation of a datum and its use in another operation, should be minimal. The scheduled code is then converted to the VHBC image format and the compiler flow concludes.

4. The Virtual Hardware Machine

Our approach assumes that hardware descriptions can be translated into a portable byte code which can efficiently be interpreted by a special hardware processor called the Virtual Hardware Machine. The design of the VHM is greatly

influenced by the properties of the byte code, namely simple gate level operations and a high level of available instruction level parallelism, which suppose a VLIW-like architecture with a very high number of functional units which possess only very small footprints.

The concept of the VHM is a general one. It aims to be easily adaptable to a variety of underlying hardware platforms, ranging from standard hardware CMOS implementations to different reconfigurable hardware substrates such as FPGAs. Due to differing platform capabilities, VHM implementations differ in the number of available functional units and registers as well as the extend of the available external port capabilities. In principle, the virtual hardware machine consists of five components:

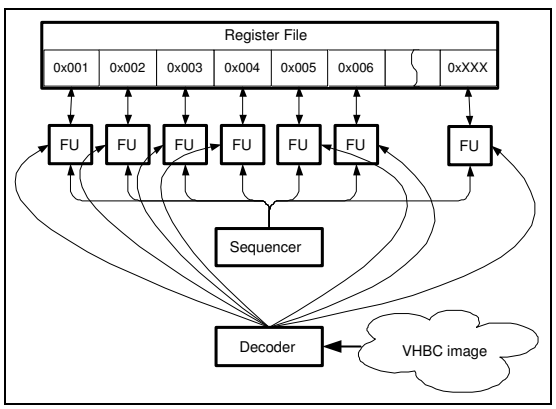

Figure 3. Principle components of the Virtual Hardware Machine

Decoder The decoder takes the byte aligned instruction input stream and extracts code blocks from it. The instructions embedded in the code blocks are adapted to the specific VHM implementation, thus register addresses might have to be recalculated or the register address sizes possibly need to be enlarged. The adapted instructions are then sent to the instruction caches of the functional units. Furthermore, the decoder is also responsible for resolving problems caused by oversized code blocks, meaning that more instructions are pre-scheduled into a code block than functional units are available. In this case, the scheduler tries to split the code blocks into smaller units.

Functional units Functional units execute the instructions of the VHBC. In order to allow for an efficient execution, each functional unit contains a processing kernel, sequencer and an instruction cache. The size of the instruction cache differs among implementations.

Register file The register file consists of single addressable memory cells. In the current implementation they possess the width of one bit. In later versions, when the VHM will work on register transfer level rather than logic, registers holding eight or more bit will be more appropriate.

Interconnect The interconnect between the functional units and the register file allows read access from every functional unit to every register. Write access to the registers is restricted to exactly one register per functional unit, thus every functional unit is associated with a hard-wired output register. The interconnect between the register file and the external ports is triggered by the sequencer. The values present at the input ports are read at the very beginning of each macro cycle, overwriting the corresponding registers, whereas the output port values are altered after all functional units have finished the execution of the instructions of the final code block.

Sequencer The global sequencer synchronizes the functional units and triggers the signal propagation to the external ports. Furthermore, it takes care of the reconfiguration of the functional units. Whenever new VHBC images or fragments arrive, the sequencer allows the decoder to extract the new instructions and distribute them to the functional units. This can be done, by either stopping the execution of the current VHBC image and fully reconfiguring all FUs with new code, or by inserting or replacing only certain hardware instructions in the instruction caches.

5. Results

In this paper, we have presented the concept of the Virtual Hardware Byte Code in a first preliminary version. To allow for a first feasibility study, the code only facilitates logic operations. Up to now, a rudimentary implementation of the Virtual Hardware Machine in VHDL, a cycle-precise simulation environment as well as a VHDL to VHBC compiler have been implemented to support first evaluations of the concept.

The VHM currently uses 32 functional units, with 16 instructions deep I-Caches and 32 registers in the register file. We implemented the VHM using a high level VHDL description and mapped it onto a Xilinx Virtex XCV800 FPGA. First tests show that the current implementation is capable of running with a core speed of at least 100 MHz.

Due to the simplicity of the current implementation three basic designs have been analyzed, a Fulladder (2Add), a 4 bit ripple carry adder (4Add) and a seven segment decoder (7Seg). Furthermore two more involved designs, a 16 bit counter (Count16) and a basic general purpose processor (GPP), were compiled and simulated using the VHM simulation environment. All five designs show that the claimed high levels of available instruction level parallelism were well grounded. Table 2 shows the obtained results. All designs were specified in VHDL and compiled using the VHDL to VHBC compiler. In the table, the column Blocks describes the number of code blocks found in the VHBC code and Parallelism the average number of instruction per block. With the number of code blocks n, a nominal delay d can be calculated for a given design as follows : $d = \frac{n}{100MHz} = n \times 10ns$. The delay values on the VHM were calculated using this formula. The delays on the Virtex were approximated using the timing information available from the Xilinx VHDL compiler.

	Blocks	Parallelism	Delay		Factor
			VHM	Virtex	
2Add	3	4	30ns	15.47ns	1.9
4Add	12	8	120ns	22.07ns	5.5
7Seg	8	31	80ns	12.7ns	6.2
Count16	15	16	150ns	18.4ns	8.2
GPP	37	358	370ns	58.3ns	6.3

Table 2. Results for different designs running on the VHM and the Xilinx Virtex

The results are quite encouraging to presume further work on the Byte Code, the Virtual Hardware Machine as well as the compiler. They clearly indicate that a design description employing the VHBC performs only factor 5 to 10 times slower than the same design compiled directly for a specific FPGA, while allowing for portability as well as easy run time reconfiguration without the need for placing and routing. On top of this, the early stage of implementation should be taken into consideration. Code optimizations as well as more sophisticated VHM implementations will definitely show even better performance results.

6. Conclusions and future work

We have defined a Virtual Hardware Byte Code (VHBC) representation for hardware components in embedded systems, which carries the concept and virtues of Java into the world of hardware design. As a result we received a portable and efficient way to transfer hardware designs via standard network environments. Consequently, we are working on a specially streamlined hardware processor, the Virtual Hardware Machine (VHM), as well as a host of software tools such as a VHDL compiler and a cycle accurate hardware simulator to support VHBC. The first version of the VHM has been implemented and vindicates the idea of implementing hardware components in VHBC and interpreting it to be viable and feasible.

The main focus of the current work is devoted to an optimized version of the VHM, which will be implemented on the Xilinx Virtex chip. It will be able to provide more functional units and a higher number of registers. In the future we will try to map the VHM design efficiently onto a variety of FPGAs of different vendors by using a pure VHDL description of the VHM similar to the C reference implementation of the Java Virtual Machine (C-Machine).

References

[1] T. Lindholm, F. Yellin, "The Java Virtual Machine Specification, Second Edition", *http://java.sun.com/docs/books/vmspec/*

[2] M. Mahadevan, R. M. Bradley, *Journal of Applied Physics*, Vol 79, 1996

[3] M. Budiu, "Application-Specific Hardware: Computing Without CPUs", *citeseer.nj.nec.com/497138.html*

[4] Y. Ha, P. Schaumont, M. Engels, S. Vernalde, F. Potargent, L. Rijnders, H. de Man, "A Hardware Virtual Machine for the Networked Reconfiguration", In *Proc. of 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, 2000

[5] Y. Ha, S. Vernalde, P. Schaumont, M. Engels, H. De Man, "Building a Virtual Framework for Networked Reconfigurable Hardware and Software Objects", In *Proc. of PDPTA '00*, 2000

[6] S. Goldstein et al., "PipeRench: A Coprocessor for Streaming Multimedia Acceleration", In *Proc. of 24th International Symposium on Computer Architecture*, 1999

[7] Y. Chou, P. Pillai, H. Schmit, and J. P. Shen, "PipeRench Implementation of the Instruction Path Coprocessor", In *Proc. of MICRO '00*, 2000

[8] B. Mei, P. Schaumont, S. Vernalde, "A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems"

[9] Y. Ha, B. Mei, P. Schaumont, S. Vernalde, R. Lauwereins, H. De Man, "Development of a Design Framework for Platform-Independent Networked Reconfiguration of Software and Hardware", In *Proc. of FLP*, 2001

[10] R. Kress, *A Fast Reconfigurable ALU for Xputers*, PhD thesis, Universitaet Kaiserslautern, 1996

[11] R. Hartenstein, M. Merz, T. Hoffmann, U. Nageldinger, "Mapping Applications onto reconfigurable KressArrays", In *Proc. of FLP*, 1999

[12] J. Becker, T. Pionteck, C. Habermann, M. Glesner, "Design and Implementation of a Coarse-Grained Dynamically Reconfigurable Hardware Architecture", In *Proc. of Workshop on VLSI (WVLSI)*, 2001

[13] C. Nitsch, U. Kebschull, "The Use of Runtime Configuration Capabilities for Networked Embedded Systems", In *Proc. of DATE'02*, Paris, 2002

[14] S. Guccione, D. Verkest, I. Bolsens, "Design Technology for Networked Reconfigurable FPGA Platforms", In *Proc. of DATE'02*, Paris, 2002

[15] G. De Micheli, "Synthesis and Optimization of Digital Circuits", McGraw-Hill Higher Education, 1994