A Hardware/Software Prototyping Environment for Dynamically Reconfigurable Embedded Systems

Josef Fleischmann
Technical University of Munich
Inst. of Electronic Design Automation
D-80290 Munich, Germany
Josef.Fleischmann@e-technik.tu-muenchen.de

Klaus Buchenrieder, Rainer Kress
Siemens AG
Corporate Technology
D-81730 Munich, Germany
{Klaus.Buchenrieder|Rainer.Kress}@mchp.siemens.de

Abstract

Next generation embedded systems place new demands on an efficient methodology for their design and verification. These systems have to support interaction over a network, multiple concurrent applications and changing operating conditions. Therefore, besides existing requirements like low cost and high performance, new demands like adaptivity and reconfigurability arise. Traditional design methodologies do not support exploration and implementation of this flavor of networked embedded systems. In this paper, we present a suitable methodology and a flexible experimental environment which supports design exploration and prototyping of dynamically reconfigurable embedded systems based on Java specifications.

1. Introduction

Today, electronic products face a heterogeneous, rapidly changing market in which the key factors for success are efficient product development and the ability to quickly respond to customer demands. This leads to the requirement that a product should not only be adaptive via software updates, but it should also be flexible within the hardware part via field programmable hardware. The situation becomes even worse for embedded systems because in many cases their behavior has to be adaptive to the working environment. More specifically, we consider networked embedded systems which are able to run multiple concurrent real-time media applications such as audio and video. For instance the frame rate of a video stream is changed by the video application manager if the network is congested. Thus a trade-off between picture quality and bandwidth of the network becomes possible. In audio applications, completely different compression algorithms are employed depending on network load [11].

Combining programmable hardware, such as field-programmable gate arrays (FPGAs) with microcontrollers or even digital signal processors (DSPs) gives an entire adaptive system. However, a flexible, configurable hardware solution is also associated with costs in terms of area and timing performance when compared to ASIC designs. As a trade-off, hybrid embedded systems with partially fixed and partially adaptable hardware combined with a processor are employed.

Our target architecture basically consists of a set of processing elements (processors, FPGAs) which execute concurrently. Therefore, the specification language must support constructs for expressing concurrency. Several languages have been proposed for describing embedded systems [4], [5]. We decided to use Java which was originally designed for the use in embedded electronic applications to overcome the major weaknesses of C and C++ [3]. Java is a multi-threaded language and supports system descriptions as sets of concurrent behaviors. As Java has certain built-in multi-threading primitives, expressing concurrency and management of different flows of control is greatly simplified. Threads also provide an efficient way for task distribution and multiprocessing. Java has all benefits of object-oriented languages. Because of its class, package and interface concept, it encourages modular and small programs. Especially for designing embedded systems, features like multi-threading, exception handling, synchronization, code reuse (e.g. *Java Beans*), security and network programming (e.g. *function shipping* for maintenance and software updates) are of great importance. The prospect of communicating with an embedded system from anywhere on the Internet is pushing more and more designers towards Java [12]. For modeling real-time systems, language extensions have been proposed [9], [10]. However, the major issue which currently hinders the wide-spread adoption of Java as a development language is its execution performance. But as soon as just-in-time compilation tools and native code compilers mature, the performance gap will be closed and Java will increasingly be adopted in the domain of embedded hardware/software systems.

Related work concerning networked embedded systems concentrates on developing stochastic models for performance verification [6]. Our goal is to provide a semi-automated co-synthesis and co-simulation framework which supports fast prototyping of hardware/software systems specified in the Java language. Within this co-design environment, it is possible to evaluate different design alternatives and gather profiling information during execution of the alternative prototype designs within a simulation testbench. The most appropriate mapping of functionality to hardware and software parts is determined in the partitioning step. Also, portions of the design which are implemented in reconfigurable hardware and portions which remain unchanged during run-time can be identified. Back-annotation of data gathered during the co-synthesis, co-emulation, and profiling steps provides the necessary information to guide the partitioning and optimization process.

The remainder of this paper is organized as follows. In section 2 we introduce the new concept for co-synthesis and co-simulation of reconfigurable hardware/software systems based on the Java specification. This includes code generation for software and hardware parts, generation of the hardware/software interface and managing the execution of the generated prototype. The current implementation of the experimental hardware platform and results are presented in section 3. Finally, some concluding remarks and an overview of our future work are given in section 4.

2. Co-synthesis environment

Our framework supports a software oriented strategy for co-synthesis as we start from a Java specification and identify parts which are to be implemented in hardware. In contrast to previous approaches we integrate co-verification of the system prototype by co-emulation of the hardware/software architecture. Verifying the correct interaction between software and hardware is a key problem in the design of a combined system. Thus, we propose a design flow which includes a complete synthesis flow and accommodates fast prototyping. After identifying a suitable partitioning of the system, the appropriate hardware/software interface is generated by an interface generator. The software part is instrumented and bytecode is generated for execution on the Java virtual machine. The hardware part is synthesized and mapped to the reconfigurable FPGA target platform. The interface is partially realized in software and in hardware. The complete design flow of the prototyping environment is shown in figure 1. A description of the individual steps is given in the following.

2.1 Partitioning

During the specification phase, only the software branch (left path in figure 1) is used for functional validation and profiling. After that, the initial specification is partitioned into a part for the execution on the host PC (software methods) and a part which is executed onto the FPGA hardware platform (hardware methods). Currently, the granularity of the partitioning is the level of Java methods. For future environments, a partitioning based on loops or on basic block level will be examined. All methods chosen for hardware can also run on the host PC, but not vice versa. This means that during code generation bytecode is generated for all methods whereas FPGA configuration files are generated for the individual hardware methods only. The runtime system (RTS) reads information from the partitioning step and decides according to the partitioning whether a method is scheduled onto the host PC or onto the FPGA hardware. The RTS also manages dynamic reconfiguration of the hardware at run-time.

For each hardware method, an interface description is automatically generated. It consists of a RT-level VHDL frame for inclusion into the hardware building block, and a hardware method call for inclusion into the software code.

Code generation for the hardware as well as for the software part is based on the Java compiler *guavac* [1]. Guavac consists of a scanner, lexical analyzer, and a part for the evaluation of the internal tree representation. The translation of a Java method into bytecode or VHDL is sketched in figure 2. Up to the internal tree representation, code generation for hardware and for software is both the same. The input is divided by the scanner into a token stream. Next, the tokens are mapped to an internal tree representation (expression trees) according to the semantic rules. Of course, also manually designed VHDL components can be used for the hardware part.

2.2 Hardware code generation

Code generation for the hardware part consists of three major steps:

- High-level synthesis (HLS) of the selected Java methods, which are to be implemented in hardware

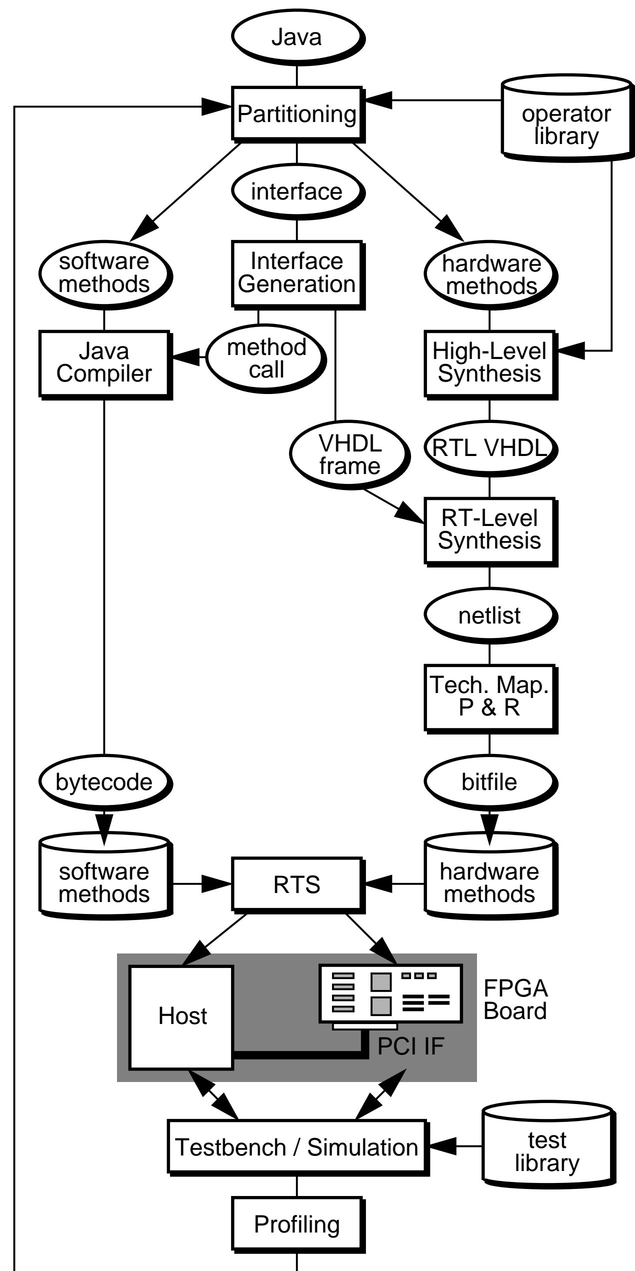

Fig. 1. Prototyping environment

- Register-transfer-level synthesis of the VHDL description which has been generated by the HLS Tool in the previous step and logic optimization of the resulting netlist
- Layout synthesis and generation of the configuration data for the target FPGA hardware platform

High-level synthesis. In the high-level synthesis step, main emphasis was laid on producing synthesizable VHDL code. Semi-automatic synthesis will be an essential feature of our co-design framework, as we want to encourage iterations during the design exploration phase. Therefore, a synthesis tool has been developed, which is based on an operator library. For all selected Java methods which comply with the restrictions of our current library synthesizable VHDL is generated. Based on the internal tree representation, high-level synthesis step is performed using standard techniques. The design representation is split into a datapath part and a controller part. Expressions and statements are converted into corresponding VHDL constructs for a datapath description. The sequence of execution and loop control constructs as well as conditional constructs are transformed into a finite state machine (FSM) representation. The HLS tool also produces corresponding synthesis scripts which are used afterwards for logic synthesis.

RT-level synthesis. The output of the HLS step is a register-transfer level description containing combinational building blocks for the datapath and a finite state machine for the controller part. Before logic synthesis of the RT-level description, an automatically generated VHDL frame for interfacing the hardware method to the software partition is added to the hardware description produced in the previous step. The VHDL frame consists of a reconfiguration controller and an interface for passing arguments and results to the Java hardware method. This interface is described in more detail in section 2.4. The complete RT-level description is synthesized with Synopsys Design Compiler.

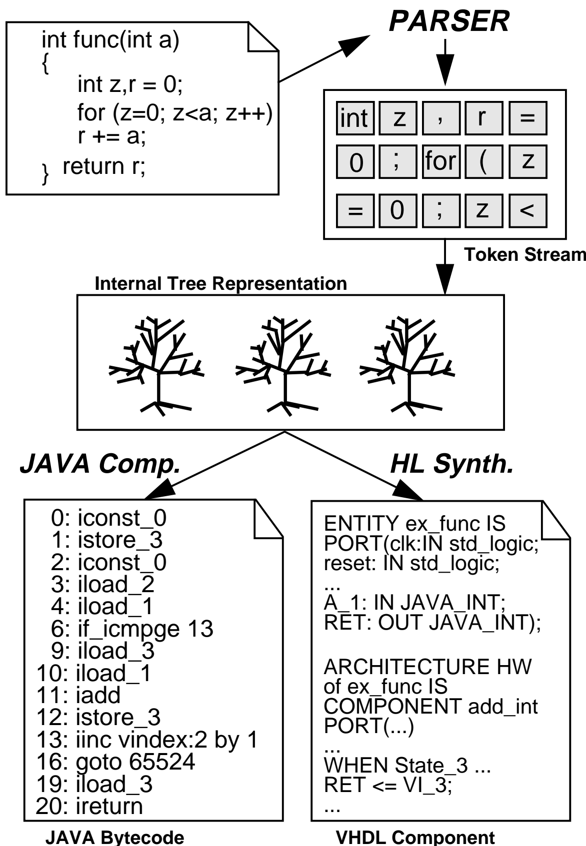

Fig. 2. Transformation of the Java specification into bytecode and RT-level VHDL

Technology mapping, placement and routing. The resulting EDIF netlist of the RTL synthesis step is further processed with commercial tools. After technology mapping, placement and routing has succeeded, a valid configuration file for the hardware method is written. This file is added to a hardware method library. The run-time system takes the necessary configuration files from this library.

If the placement and routing fails for the target FPGA device, no configuration file is generated and of course none can be added to the hardware method library. Thus, this method has to be executed on the host PC. Currently, no support for partitioning a single hardware method onto several FPGA devices has been implemented. Besides this semi-automatic code generation for the hardware, manually designed configuration files for the FPGAs can be integrated as native methods into the Java code.

2.3 Software code generation

Code generation for software part is done using conventional compilation methods. As mentioned earlier the Java Parser transforms the specification into a token stream (figure 2). From the token stream the expression trees are built according to grammatical rules. Possible optimization of the code can occur at this stage. A list of all expression trees is processed to get a list of bytecodes which are saved as *.class*-files. These object files are stored in the software method database (refer to figure 1).

2.4 Interfacing hardware and software

In order to facilitate the exchange of data between the hardware and the software part of the system a special interface concept and an automated interface generator have been developed.

The software part. Since the chosen level of granularity is the Java method level, only little extensions were necessary on the software side. As previously mentioned, in the current implementation of our co-design environment software methods are executed by the interpreter. For interpreting Java bytecode, we use the Gnu Interpreter Kaffe, as the sources are in the public domain. For using it in our co-design framework several extensions to the interpreter were necessary. First, the interpreter must be able to read in the partitioning information, i.e. which function is to be executed in hardware and which method in software. If the control flow during bytecode interpretation reaches a hardware method, then mechanisms for synchronization and communication become necessary. Method arguments have to be delivered to the hardware driver and the corresponding results must be read back. During execution of a method in hardware, the calling thread in software is suspended. The calling thread resumes normal operation after

the results of the hardware method have been sent back. In the current implementation only one thread at a time is allowed to access the emulation hardware, therefore a suitable synchronization mechanism has been implemented. For evaluating system performance, the interpreter had to be equipped with functions for gathering profiling data. Therefore, routines for measuring times for data processing and for communication between hardware and software had to be integrated in the Java run time system.

The hardware part. The same way the software part had to be extended for communication, the hardware design also needs extensions for exchanging information whenever a request for processing data occurs. Thus, a concept for a flexible, parameterizable communication interface for the hardware design has been developed. Depending on the number and size of arguments that the hardware method has to process, an appropriate interface frame is generated for each Java hardware method. The interface generator uses information provided by the high-level synthesis tool and automatically generates a register-transfer level description of the VHDL frame with the correct size. The structure of this interface frame is depicted in figure 3. The frame consists of a set of registers for storing input and output data of the corresponding Java method. Furthermore, it contains a small logic block for handling control signals which are necessary for interaction with the run-time system. This includes signals for starting, resetting the hardware process and a ready signal which indicates that the computation in hardware has finished and that the results can be read back.

The VHDL component for the Java hardware method which has been generated during high-level synthesis is embedded into the VHDL frame produced by the interface generator. Then the complete design is synthesized for the FPGA platform as described in section 2.2.

2.5 Run-time system

The run-time system (RTS) is responsible for managing execution and interaction of hardware and software methods. Therefore, it relies on a database which contains the set of methods which are executable in either hardware or software and a number of methods for which both hardware and software implementations are available. The software executables (Java classes) are stored in Java bytecode format and the corresponding hardware blocks which are emulated on the FPGA board are stored as configuration bitfiles.

During co-simulation the run-time system schedules methods for execution according to the current partitioning

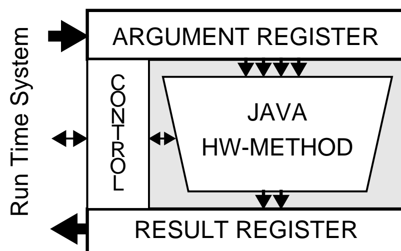

Fig. 3. VHDL frame

table. Software methods are executed on the Java Virtual Machine (JVM) on the host workstation as shown in figure 1. As we chose a software-oriented approach, the execution flow of the combined system is dominated by the software part of the system.

The run-time system also manages execution of tasks on the emulation platform. The set of currently available FPGAs and communication channels is specified in a configuration file. During co-simulation when control flow in one of the threads reaches a hardware method, the run-time system determines whether the corresponding configuration bitfile has already been downloaded to an FPGA. At the beginning of the co-simulation procedure none of the FPGAs is configured with a bitstream. In this case, the run-time system determines an available FPGA and triggers the download mechanism. In case the bitfile has already been downloaded to an FPGA, the RTS determines the address of the FPGA containing the currently requested hardware method, and starts transfer of data which has to be processed by this method. After the emulated method has finished processing, it sends an interrupt signal to the RTS and the results from the computation can be read from the result register of the FPGA design. As previously mentioned, only one thread at a time is allowed to access the emulation hardware in our current implementation. During emulation of a hardware method on the FPGA, the calling thread is suspended but any other software thread can be executed on the Java virtual machine at the same time.

2.6 Simulation and Refinement

Executing simulation with the actual hardware gives feedback for the initial partitioning process via the profiling step. The prototype is executed in a testbench environment where several test cases exist in a test library. These test cases consist of data sets for the execution of the Java specification. A profiling step measures execution times for the hardware and the software part, as well as the communication time via the interface. These results are directly fed back to the partitioning process. The goal is to improve the partitioning especially when the granularity of the partitioned parts changes or if methods are clustered. This feedback loop is currently under development.

3. Experimental results

For verifying the feasibility and studying different concepts of our new approach, an implementation of the proposed design flow and the corresponding hardware/ software co-emulation engine has been done. Our initial implementation was based on an dedicated prototyping board containing four SRAM based FPGAs [2]. This solution was found to be inefficient for dynamically reconfigurable embedded systems. Chip reconfiguration requires too much time in traditional FPGA architecture, and partial reconfiguration during run-time is impossible. Furthermore, a rather sophisticated mechanism for interfacing between processor and FPGA hardware had to be developed. Because of these experiences, we decided to use a new FPGA architecture in our new implementation which overcomes this limitations. An overview of the hardware/ software prototyping platform is given in figure 4.

Hardware. The workstation is used to enter the Java specification, compiling Java to bytecode and executing the bytecode for the software part of the system. Furthermore, the different synthesis tasks necessary for moving dedicated Java methods to the FPGA emulation board are carried out on the host workstation. The run-time system is also implemented in software on the host. For implementing the hardware part of a prototype, a dedicated FPGA board is connected to the workstation over the PCI bus. We used the XC6200DS board which basically provides a PCI interface, and a reconfigurable processing unit XC6216. The card also includes two banks of memory, which may be accessed from either the FPGA or the PCI bus. For a more detailed description see [8].

Performance. For evaluating the performance of our platform and identifying bottlenecks, we used some small data-oriented examples specified in Java. The XC6216 reconfigurable processing unit that is currently used has a very limited gate capacity (approx. 16000 gate equiv.). Larger design examples will be tested as soon as a larger chip of the XC6200 family or a board with more chips is available. The most prominent feature of this FPGA is its integrated microprocessor interface. As this interface supports direct register reads and writes over an address and a data bus. Thus transfer of arguments and results between hardware and software parts of the prototype is greatly simplified ('wireless I/O'). Similar to a memory access, all configurable cells and registers on the chip can be read from and written to at run-time. The second important feature for our target application area is the high-speed reconfiguration capability of the chip, which is mandatory in a system which has to adapt to different operating conditions. For our sample designs the times needed for reconfiguration of the complete hardware part varied between 18 and 300 ms depending on the size of the design. After additional optimization of reconfiguration process over the PCI interface of our board, these times have been cut down to 4.5 to 29 ms. When testing the speed of the hardware/software interface, we found that in one second about 84000 register writes (32bit) or 63000 register reads are possible. However, the PCI interface implementation on the XC6200DS board has been identified as a bottleneck. For reading and writing to the on-board SRAM, only data transfer rates of about 4 and 7 MB/s respectively can be obtained (depending on board frequency) and this is significantly lower than in typical PCI applications. Related research [7] showed that with the XC6216 significant speedups for image processing applications can be obtained. For estimation of worst case behavior, we used an application consisting mostly of multiply and add operations, which are ideally suited for a standard microprocessor like a cached Pentium 133 as used in our host PC. For our dynamic Java application (including 3 hardware designs; 82 dynamic reconfigurations) execution times of 4.1s for the hardware/software prototype compared to 2.8s for the pure software execution were measured. These are very promising results concerning performance of the execution of a hardware/software system prototype, as 80-90% of the execution time are due to communication over the PCI interface. Therefore, we will benefit greatly from an improved version of the PCI interface which will become available in the near future.

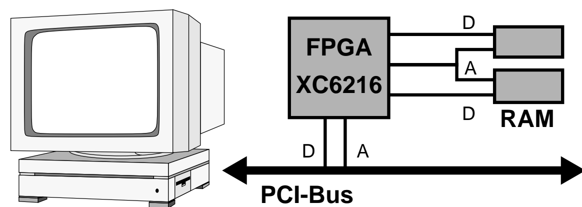

Fig. 4. Experimental platform

4. Conclusions

When developing complex systems, intelligent trade-offs between hardware and software components are necessary to deliver the design best satisfying performance and cost constraints. Therefore, designers need a complete development framework that facilitates such trade-offs. A new concept for tool-assisted design exploration and fast prototyping of hardware/software systems has been proposed in this paper. Starting from a system specification in Java, a novel design flow has been presented which is targeted to next generation embedded systems including reconfigurable hardware. It supports trade-off evaluation, interface generation and verification of the design by co-emulation. The focus of current research is on the implementation of different signal processing applications within the prototyping framework. Future work will concentrate on integrating a seamless VHDL to FPGA synthesis design flow. For instance, currently available tools for this new FPGA architecture still require manual intervention of the designer during layout synthesis.

5. References

[1] D. Engberg: GUAVAC home page; Internet URL: http://http.cs.berkeley.edu/ ~engberg/guavac/, 1997.

[2] J. Fleischmann, K. Buchenrieder, R. Kress: Co-Design of Hardware/Software Systems based on Java Specifications. Tech. Report TUM-LRE-97-4, Tech. Univ. of Munich, 1997.

[3] J. Gosling, B. Joy, G. Steele: The Java^{TM} Language Specification; Addison-Wesley, Reading Massachusetts, 1996.

[4] D. Gajski, F. Vahid, S. Narayan, J. Gong: Specification and Design of Embedded Systems. Prentice Hall, 1994.

[5] R. Helaihel, K. Olukotun: Java as a Specification Language for Hardware-Software Systems. In Int. Conf. on Computer-Aided Design (ICCAD), 1997.

[6] A. Kalavade and P. Moghe: On the Performance Verification of Embedded Systems with Concurrent Dynamic Applications. In Asilomar Conf, Nov. 1996.

[7] T. Kean, A. Duncan: A 800Mpixel/sec Reconfigurable Image Correlator on XC6216. In Int. Workshop on Field-Programmable Logic and Applications, 1997.

[8] S. Nisbet, S. A. Guccione: The XC6200DS Development System. In Int. Workshop on Field-Programmable Logic and Applications, 1997.

[9] K. Nilsen: Embedded Real-Time Development in the Java Language. In Embedded Systems Conf. West, 1996.

[10] R. Passerone, et al.: Modeling Reactive Systems in Java. In Int. High Level Design Validation and Test Workshop, Nov. 1997.

[11] H. Schulzrinne, et al.: RTP: A Transport Protocol for Real-Time Applications. RFC 1889, Audio-Video Transport WG, Jan. 1996.

[12] T. Williams: Java goes to work controlling networked embedded systems.Computer Design 35, No. 9, Aug. 1996.