Java Driven Codesign and Prototyping of Networked Embedded Systems

Josef Fleischmann
Technical University of Munich
Inst. of Electronic Design Automation
D-80290 Munich, Germany
Josef.Fleischmann@ei.tum.de

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

Abstract

While the number of embedded systems in consumer electronics is growing dramatically, several trends can be observed which challenge traditional codesign practice: An increasing share of functionality of such systems is implemented in software; flexibility or reconfigurability is added to the list of non-functional requirements. Moreover, networked embedded systems are equipped with communication capabilities and can be controlled over networks. In this paper, we present a suitable methodology and a set of tools targeting these novel requirements. JACOP is a codesign environment based on Java and supports specification, co-synthesis and prototyping of networked embedded systems.

1. Introduction

The rapidly growing market for web-enabled consumer electronic devices introduces a paradigm shift in embedded system design. Traditionally, embedded systems have been designed to perform a fixed set of previously specified functions within a well-known operating environment. After shipment, the functionality of the embedded system remains unchanged during product lifetime. However, with shorter time-to-market windows and increasing product functionality this design philosophy has exhibited its shortcomings. Hardware/software codesign tools are increasingly used to alleviate some of the problems in the design of complex heterogeneous systems. But the key features of next-generation embedded devices will be the capability to communicate over networks and to adapt to different operating environments. There is an emerging class of systems which concurrently execute multiple applications, such as processing audio streams, capturing video data and web browsing. This systems need to be adaptive to changing operating conditions. For instance, in multimedia applications the video frame rate has to be adjusted depending on network congestion. Likewise, for audio streams different compression techniques are applied depending on network load. Besides this class of multi-function systems there are multi-mode systems, i.e. systems which know several alternative modes of operation, for example a mobile phone which is able to switch between different communication protocols or a transmitter which can toggle between different encryption standards. This paradigm shift in functional and non-functional requirements of embedded appliances not only holds for consumer devices. In industrial automation there is a growing demand for sensor and actuator devices which can be remotely controlled and maintained via Internet.

Several system-level design languages and codesign frameworks have been proposed by researchers and are gaining acceptance in industry. But there is a lack of methods and tools to investigate issues which are raised when designing runtime-reconfigurable hardware/software systems. Our goal is to develop a complete design environment for embedded systems which include dynamically reconfigurable hardware components. JACOP (Java driven codesign and prototyping environment) is based on Java which is used for specification and initial profiling as well as for the final implementation of system software. In this paper, we give an overview of the implemented codesign flow, we present a tool for managing the interaction of hardware and software components and briefly outline our integrated concept for component based reuse.

Existing work in the area of networked embedded systems concentrates on analytical models for performance prediction. Kalavade et. al. developed a tool for early performance prediction of adaptive systems [6]. With respect to the underlying hardware architecture for prototyping, an overview of related work can be found in [12]. Recently, also other researchers have proposed Java for specification of embedded systems [4][13]. In [11] extensions to Java for specification of systems with hard real-time constraints are presented. A case study on how to use Java for building circuit components on FPGAs has been published in [8]. Other recent approaches propose Java for simulation and emulation of digital VLSI circuits [1][7]. To the best of our knowledge, our paper is the first to introduce a Java based design flow for networked reconfigurable systems including rapid prototyping. In contrast to previous approaches, we have developed and implemented mechanisms for integrating reconfigurable hardware components into the Java virtual machine. JACOP also provides means for managing different threads in hardware and a concept for reuse of implemented hardware and software components.

2. Java Based Codesign

Designing the digital hardware part of a system has become increasingly similar to software design. With widespread use of hardware description languages and synthesis tools, circuit design has moved to higher levels of abstraction. For managing complexity of future designs, abstract specification and reuse of Intellectual Property (IP, previously developed HW & SW components) is essential. Therefore, a specification language should provide comfortable means for integrating reuse libraries (i.e. packages of previously developed IP components). Furthermore, object-oriented programming has proven to be a very efficient paradigm in the design of complex software systems. Object orientation may infer performance loss, but its benefits weigh much higher: First, it provides a better means for managing complexity and for reusing existing modules. Secondly, it reduces problems and costs associated with code maintenance. In embedded system design, one major trend is to increasingly implement functionality in software. The dominant reasons for this are faster implementation, more flexibility and easier upgradability. Consequently, the costs of software maintenance are an issue of growing importance. For this reasons, we chose Java as a specification language. It is a clean, object-oriented, versatile language of moderate complexity. As Java has built-in support for handling multiple threads; expressing concurrency and managing different flows of control is well supported.

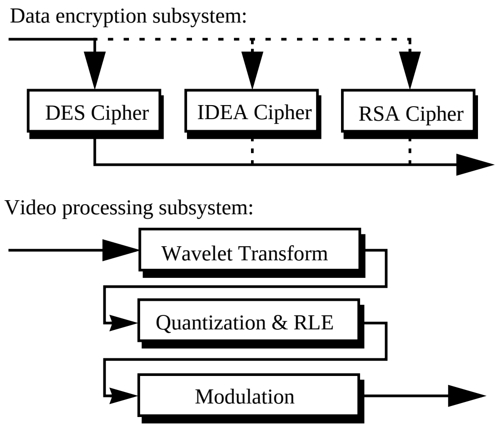

Figure 1. Reconfigurable systems: set-top box

The JavaBeans specification [5] also provides a standard concept for reuse of software components. With respect to networked embedded system design, features like internet mobility and network programming (e.g. *function shipping* for maintenance and feature updates), security and synchronization are of great importance. However, Java has not been designed for specification of real-time systems. Therefore, several researchers have proposed extensions (and restrictions) to the language for specifying such systems. The need for communicating with embedded systems over the Internet pushes more designers towards Java, e.g. PersonalJava has recently been adopted as the premier design platform for implementing applications for set-top boxes.

As already mentioned, our framework for specification and design exploration of mixed hardware/software implementations is targeted to reconfigurable embedded systems. In figure 1, two examples for application of runtime reconfiguration are given. The first task graph shows an encryption system which is able to switch between different operations modes, i.e. different encryption algorithms. Typically, only one application at a time is active and reconfiguration of the system is not time-critical. The second task graph represents a part of a video communication system. In this case, the task of processing a video stream is decomposed into a series of subtasks which are executed on the same piece of silicon. The chip is reconfigured periodically. In order to meet soft deadlines, hardware components which are rapidly reconfigurable are needed.

Our proposed design flow can be summarized as follows: Starting from an initial Java specification, profiling data is gathered while executing the program with typical input data. This profiling data is then analyzed and animated to guide the designer in the partitioning process. Partitioning is done at the method level of granularity using a graphical user interface. Functions which are to be implemented in hardware are synthesized using high-level and logic synthesis tools. Previously designed hardware components are integrated by using a database of parameterizable VHDL components. More information on the individual synthesis steps and the automatic generation of the hardware/software interface can be found in [3]. After co-synthesis, Java bytecode for all methods of the initial specification is stored in the pool of software methods. For all methods, which are candidates for implementation in reconfigurable hardware, the FPGA configuration data as well as interface information is stored in the pool of hardware methods. The target hardware platform consists of dynamically reconfigurable FPGAs (DPGAs). These new FPGA architectures may be partially reconfigured at run-time, i.e. a portion of the chip can be reprogrammed while other sections are operating without interruption. The target software platform for system prototyping is currently a Linux Pentium PC.

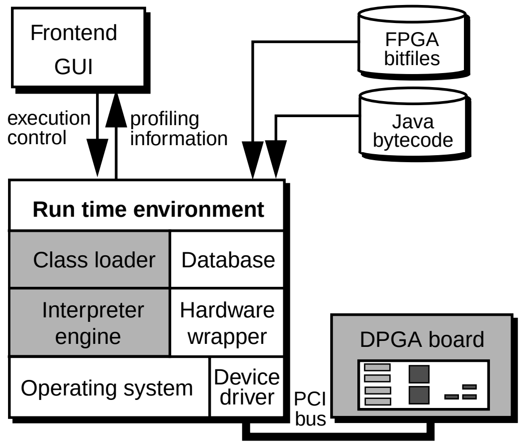

Figure 2. Design exploration platform

3. Runtime Management

The interactions between the hardware and software parts of the system, as well as the reconfiguration process is managed by the runtime environment (figure 2). The run-time manager schedules methods for execution either as software on the Java virtual machine of the host processor or as hardware on the reconfigurable DPGA hardware. The scheduling depends on the dynamic behavior of the application and on the current partitioning table chosen by the designer. In contrast to traditional FPGA based prototyping systems, execution on this platform is a highly dynamic process. The execution flow of the hardware/software system is dominated by the software part. Software methods are executed on the Java virtual machine. Whenever the control flow reaches a hardware method, the run-time system determines whether the appropriate configuration file has already been downloaded. If not, then the manager chooses an available DPGA and starts configuration. If there is already a DPGA configured with the desired functionality, or if only partial reconfiguration is necessary, the address and parameters of the communication channel to the target DPGA are loaded.

A virtual machine has been developed where a Java class and native methods are used for interfacing with the hardware part of the system. The main focus is on implementing all necessary functionality for hardware interfacing and reconfiguration in Java. Therefore, the platform specific API of the reconfigurable hardware board can be kept very small. In this case the board API basically consists of native functions to write a value to and read a value from a certain address of the board. These functions are implemented via the *Java Native Interface (JNI)*. That means, all methods for managing the reconfiguration process and execution are implemented in Java and all communication to the hardware board is based on the native implementations of the read and write functions. For communicating with the external board via the PCI bus, a dedicated Linux device driver has been implemented.

The benefits of this approach are clear. The Java VM does not have to be modified and the hardware interface is clearly defined within the Java language. This means the designer has complete control over all methods for accessing and managing the reconfigurable hardware. The drawback is that the applications source code has to be modified. The interface class `DPGA_circuit` has to be included in the application source and designer has to call the appropriate functions for using the DPGA. However, this methodology can be used with any virtual machine. Therefore, it is relatively simple to integrate and test different commercial implementations of the JVM. To support a more comfortable interface for the user of the JACOP system, a special class (`HW_base`) for managing hardware designs and for controlling the reconfiguration process has been implemented as shown in figure 3. This class encapsulates all functionality which is specific to the underlying hardware resource and also hides the details of the hardware/software interfaces from the designer. Communication to the DPGA board is done by method invocations of the board API (`DPGA_circuit`) class. For exploiting the dynamic reconfiguration capabilities of the hardware, it is desirable to implement several individual hardware designs on the chip. Each of these designs is accessed by a corresponding Java thread. With the JaCoP native implementation, a mechanism is provided that multiple threads can make concurrent use of the external hardware resource. Therefore the class `HW_job` has been developed. An object of this type (figure 3) represents a thread which makes use of hardware methods. Again, these objects use a single instance of type `HW_base` when accessing the DPGA. In order to avoid racing conditions or invalid hardware configuration, the operations which access the DPGA are defined as critical sections. That means, that methods for reconfiguration and for register reads and writes are synchronized. The currently active `HW_job` cannot be preempted by other threads while executing such critical sections. In our applications, threads are used in two different scenarios: One typical example is that a system task operates on certain blocks of data, which consumes a significant amount of time. After processing this data, the thread terminates. The second case is, that a task is periodically activated. In order to save the costs of repeated reconfiguration, the corresponding hardware design is kept on the FPGA. Whenever data is available for processing, the corresponding thread becomes active and returns to a wait state afterwards. To some extent this way of using a reconfigurable hardware resource can be compared to the swapping of processes in a multi-tasking operating system.

3.1 Reuse of Hardware Beans

Well established techniques from software engineering are increasingly used to leverage the concurrent design of hardware and software of embedded systems based on a single system-level description. When designing a set of systems within a certain application domain (e.g. set-top boxes), reuse of standard components (modulators, video/audio processing tasks, ciphering algorithms, data compression, error control) is especially attractive. In order to reduce development costs and time-to-market, we are currently integrating a tool and a methodology for reuse into JACOP. The methodology is based on the JavaBeans mechanism [5] for reusing software components. Therefore, the concept had to be extended to also allow for hardware components (hardware beans). Because of space limitations, only a short discussion of the main aspects of beans and how they are related to the special requirements for hardware components is given.

Introspection and Serialization: Typically, beans are used for composing applications by help of a visual builder tool. In this design environment, the different parameters of a component are exhibited (introspection) and interactively customized. After customization, the chosen configuration has to be saved (serialization)

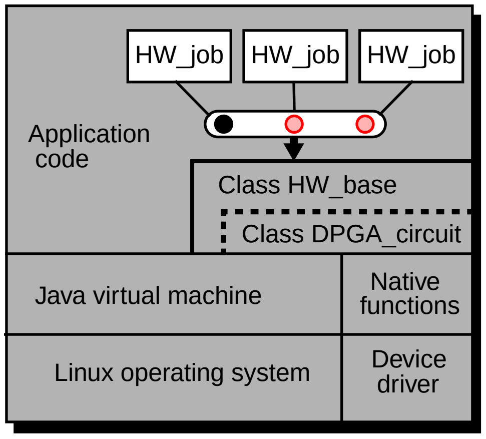

Figure 3. Thread access to hardware

for use within the run-time environment. This mechanisms are also used for hardware designs. For example when composing a system in the JACOP environment, the weights of a FIR-filter bean can be chosen by the system designer without the need for an additional synthesis step.

Properties and Events: The implemented functionality of a hardware/software basic component is reflected by the methods of a bean. Events are used for inter-component communication during run-time. Properties are basically named attributes of a component, which can be accessed at design-time and at run-time. According to the strict naming convention, property X can be accessed by so-called getX and setX methods. With respect to hardware components, properties and events are both used for transferring data between components. For example by setting a property of a decoder bean, a block of input data can be transferred to the decoder. After processing of the data block, an event is fired and the result can be obtained by retrieving the corresponding bean property. As previously mentioned, multiple threads can access both hardware and software beans at run-time. Therefore, aspects of synchronization need to be taken into account when developing a bean.

Reuse methodologies typically require additional efforts during the design of a specific component. These costs easily pay off when implementing a variety of systems within a certain application domain. One of the benefits of Java is that it encourages design for reuse by providing the JavaBeans mechanism and also by providing a built-in concept and a tool for hypertext documentation of Java classes.

4. Experiments

The JACOP design flow has been implemented and tested on a given prototyping platform. The performance of alternative approaches for run-time management has been analyzed. With respect to our target DPGA architecture, efficient optimizations of the hardware/software interface have been developed.

4.1 Target Platform

The target architecture of our system consists of a standard microprocessor tightly interfaced with a dynamically reconfigurable FPGA. Both are also connected to static random access memory. This card is a prototype of a single chip solution of a reconfigurable system. Such systems will be available in the near future, e.g. the Siemens Tricore [2] or the National Napa1000 Reconfigurable Processor [9]. The Napa1000 is a single-chip implementation for signal processing which provides both fixed logic (a 32 bit RISC core) and a reconfigurable logic part (a 50k gate Adaptive Logic Processor). For developing our co-design methodology and the corresponding co-synthesis flow, we used a Linux PC as development platform and host processor and the XC6200DS board as a reconfigurable hardware resource. The main components of this board are a PCI interface and a reconfigurable processing unit XC6216. Furthermore, there are two banks of memory included, which can be accessed from both the DPGA and the PCI bus. For a more detailed discussion of this hardware extension see [10]. The most prominent feature of this DPGA is its microprocessor interface. To every logic cell or register on the chip direct write and read operations over an address and data bus are supported. This provides a comfortable mechanism for transferring data between hardware and software components of a design, because the hardware wrapper can read from and write to every register of the hardware design at run-time.

4.2 Optimization and Results

In this section, the first alternative implementation for the run-time system (as presented in [3]) will be referred to as the ‘JACOP interpreter’ and the second implementation (as discussed above) as the ‘JACOP native interface’. For evaluating the performance of the system, our initial experiments with the JaCoP interpreter were focussed on the costs of DPGA reconfiguration and communication between hardware and software. First measurements indi-

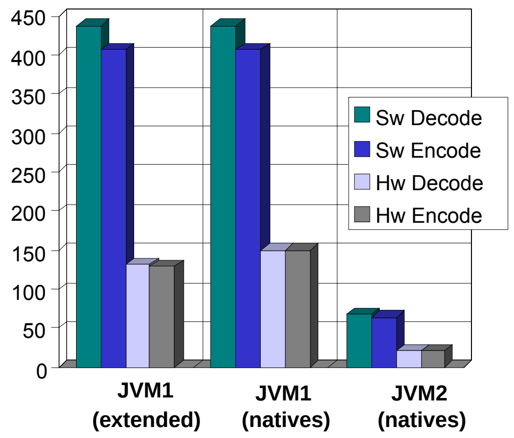

Figure 4. Error codec application

cated, that complete reconfiguration of the chip would be too slow for dynamic applications with soft deadlines. Typical times for standard reconfiguration ranged between 30ms and 400ms depending on the size of the circuit. In order to reduce this overhead for reconfiguration, a mechanism for compression and improved transmission of the DPGA configuration files has been developed. Basically, all redundant address/data pairs are omitted and the necessary configuration data is transmitted in a binary format. With this optimization, reconfiguration time is reduced to about 4ms to 31ms. Besides reconfiguration, the second important factor which introduces overhead is the hardware/software communication. An efficient implementation of the hardware/software interface is the most important factor for overall performance of the combined system. In our target architecture communication is done by writing and reading internal registers on the DPGA. For the XC6216, a single write or read operation can only access one specific column of logic cells. In order to minimize communication costs, a layout optimization is introduced such that individual registers are placed in individual columns whenever possible. Furthermore, for the used DPGA, a so-called map register has to be configured before accessing an individual register to mask out the corresponding rows of logic cells. Profiling shows that this is a very time consuming process, therefore we introduced a second optimization for the layout of our hardware designs. Whenever possible, all I/O registers of a design have to be placed in corresponding rows. With this layout constraint, configuration of the map register before read or write accesses can be avoided. Consequently, register access is improved drastically from about 70 µs down to 8µs.

With this optimizations, we conducted some further experiments to investigate how the total overhead for integrating external hardware can be attributed to the different components involved in the DPGA execution process. We found that about 78% of the total time for hardware integration is spent in the virtual machine and the device driver. For executing the DPGA design and transfer of data over the PCI bus 22% of the time is spent. For this reason, speedup of a mixed hardware/software implementation can only be obtained when implementing a method of significant complexity in hardware. Moving only simple operations like additions or multiplications to DPGA hardware can not deliver speedup because of the overhead involved. Therefore, we implemented more complex examples, such as an algorithm for error detection and correction. Hamming codes are typically used in conjunction with other codes for detection and correction of single bit faults. Both the Hamming coder and decoder have been implemented on the DPGA. As this application includes more complex bitlevel operations, we consequently experienced significant speedup of the hardware/software implementation in comparison to the execution of the software prototype on the host CPU. The performance figures for the coding and decoding examples are illustrated in figure 4. The first column shows the execution time for an encoding and a decoding application with the JACOP interpreter. The second column uses the same Java virtual machine, but the JACOP native interface. The main advantage of the native interface is that it can be used with any Java VM; therefore we could use an optimized JVM2, which is only available in binary form. In all three cases, significant speedup can be obtained by integrating the reconfigurable hardware platform. If an application has computational complex components which are executed on the DPGA board, hardware accelerated execution is possible despite of the necessary overhead for reconfiguration and communication. The results also show that the implementation via native functions has a small performance drawback when compared to modifying the virtual machine. On the other hand, this implementation is especially attractive when using commercial JVMs. In any case, the hardware/software interface has to be designed and optimized with great care.

5. Conclusion

A new codesign environment for Java based design of reconfigurable networked embedded systems, called JACOP, has been presented. This framework includes tools which aid in specification and hardware/software partitioning, profiling, co-synthesis and prototype execution. We have developed two alternative methods for efficiently combining the Java execution mechanism and a reconfigurable hardware resource. This approach has been extended to provide means for implementation of hardware/software threads, i.e. multiple threads can concurrently exploit the features of the reconfigurable DPGA architecture. Based on the JavaBeans specification, a suitable methodology for design reuse has been integrated. The proposed design flow has been implemented and tested on a PC connected to a reconfigurable hardware board where performance figures with sample designs have been obtained.

References

[1] P. Bellows, B. Hutchings: JHDL - An HDL for Reconfigurable Systems. In IEEE Symposium on Field-Programmable Custom Computing Machines, 1998.

[2] Peter Clarke: Tricore to get flash FPGA integration. In EE Times, No. 1000; CMP Media, 1998.

[3] J. Fleischmann, et. al.: A Hardware/Software Prototyping Environment for Dynamically Reconfigurable Embedded Systems. In Int. Workshop on HW/SW Codesign (CODES), 1998.

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

[5] JavaBeans API specification, Sun Microsystems, http://java.sun.com/beans, 1998.

[6] A. Kalavade and P. Moghe: A Tool for Performance Estimation of Networked Embedded End-Systems. In Design Automation Conference (DAC), 1998.

[7] T. Kuhn, W. Rosenstiel: Java Based Modeling and Simulation of Digital Systems on Register Transfer Level. In Workshop on System Design Automation, 1998.

[8] D. E. Lechner and S. A. Guccione: The Java Environment for Reconfigurable Computing. In Int. Workshop on Field-Programmable Logic and Applications (FPL), 1997.

[9] National Semiconductor: Napa1000 Adaptive Processor, http://www.national.com/appinfo/milaero/napa1000, 1998.

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

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

[12] M. Vasilko: Dynamically Reconfigurable Hardware WWW Library, Bournemouth University, http://dec.bournemouth .ac.uk/drhw_lib/

[13] J. S. Young, et. al.: Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement. In Design Automation Conference (DAC), 1998.