Dynamic Reconfigurable System Using Modular Design and JBits

Zoltan Baruch
Computer Science Department
Technical University of Cluj-Napoca
26-28 Gh. Baritiu St., 400027 Cluj-Napoca
Romania
Zoltan.Baruch@cs.utcluj.ro

Abstract – **This paper describes the design of a dynamic reconfigurable system in an FPGA (Field Programmable Gate Array) device. The system contains a reconfigurable part, which can be configured as a convolution filter or a FIR (Finite Impulse Response) filter, and a fixed part, which is used by both filters. The design method employed combines the use of mainstream synthesis and implementation tools and the JBits tool suite. Mainstream tools allow to design at a high level, but they lack the support required for dynamic reconfiguration. The JBits tool suite allows to generate and modify configuration bitstreams for Xilinx Virtex FPGA devices. Its ability to create partial configurations is combined with the modular design flow in order to exploit the advantages of both design methods.**

I. INTRODUCTION

Reconfigurable computing is a computing paradigm that has emerged in the last decade. This paradigm allows to define the computing resources required by each application and to configure these resources onto a programmable logic device, usually a Field Programmable Gate Array (FPGA). The reconfiguration of the target device is performed under software control. In this way, applications that are computationally demanding can be executed efficiently by allocating more hardware resources [1]. Research of FPGA-based reconfigurable systems has demonstrated their efficiency over general-purpose processors and software solutions in several applications [2].

Many of the systems designated as reconfigurable can only be statically configured [2]. When *static reconfiguration* is used, the target device is completely configured before system operation begins. If a new configuration is required, it is necessary to stop system operation and to reconfigure the device before operation can be resumed.

As opposed to static reconfiguration, *dynamic reconfiguration* or *run-time reconfiguration* (RTR) allows to modify only a part of the system while the rest of the system continues to operate. Dynamic reconfiguration has several important advantages. First, it allows custom synthesized logic to be generated and configured at run-time, which results in simpler and faster circuits [3]. Second, by only partially reconfiguring the device, the amount of configuration data required and the reconfiguration time can be drastically reduced.

Although significant progress has been made in the field of reconfigurable technology, this technology is not widely used yet. The main reason is the lack of high-level design tools that support partial reconfiguration. The various design solutions for partial reconfiguration cannot be combined easily with the high-performance mainstream design tools, and usually the design can only be performed at a low level. An example is the Xilinx Java-based class library JBits [4], which allows to create configuration bitstreams for Xilinx Virtex FPGA devices using structural descriptions of the system. Currently, JBits does not offer higher-level synthesis, optimization, or timing analysis capabilities, which seriously restricts its use for designing complex systems.

In this paper we illustrate a design method that combines the ability of the JBits tool to create and manipulate partial configuration bitstreams with the possibility of designing at a high level using mainstream synthesis and implementation tools for FPGA devices. This design method is based on the modular design adapted for partial reconfiguration. We illustrate this method by designing a dynamic reconfigurable system containing a reconfigurable part and a fixed part. The reconfigurable part can be configured as either a convolution filter which uses distributed arithmetic, or a Finite Impulse Response (FIR) filter, while the fixed part is used by both filters.

This paper is organized as follows. Section II introduces the reconfigurable computing paradigm. Section III details the main design flows that can be used for reconfigurable systems: the JBits tool suite, direct bitstream manipulation, and modular design. Section IV discusses the implementation of a simple dynamic reconfigurable system to illustrate the proposed design method. Finally, Section V concludes the paper.

II. RECONFIGURABLE COMPUTING

Reconfigurable computing combines the flexibility of general-purpose processors with the efficiency of custom hardware, bridging the gap between the performance of ASICs and microprocessors [5]. The effectiveness of reconfigurable computing has been shown in several areas, such as video image processing, microprocessor emulation, encryption/decryption, or digital signal processing. The performance achieved by several reconfigurable architectures is often with one or two orders of magnitude greater than that of programmable processors [6].

The reconfigurable computing paradigm allows to improve the performance of a computing machine by defining custom computing resources based on the specific application required. Currently, these resources are usually implemented in FPGA devices. An FPGA device is an array of logical blocks whose function and interconnection can be configured by the user. Most FPGA devices use small look-up tables as programmable computational elements. These tables are wired together with programmable interconnects.

Like processors, FPGA devices are "programmed" (configured) after fabrication to solve a particular task. In traditional processors, operations are temporally composed by sequencing them in time, using registers or memory to

store intermediate results. In contrast, in reconfigurable devices tasks are implemented by spatially composing primitive operators [7]. Because computations are performed using spatial pipelines composed of a large number of active computing elements, rather than sequentially reusing a small number of computing elements, high performance can be achieved.

Due to the limited configurable hardware available in a device, there is a need to change the configuration of the device upon demand and in real-time in order to perform multiple functions using a minimal configuration. FPGA devices require a relatively long reconfiguration time. To achieve run-time reconfiguration, a very high reconfiguration data rate is needed if the configuration has to be changed at a high frequency.

The attempts to reduce the reconfiguration data rates led to different reconfigurable architectures, such as multiple-context and partially reconfigurable [8]. A *multiple-context* architecture stores multiple layers of configuration information, referred to as contexts. Only one context is active at a time, but a very fast context switch is possible. Each layer of the configuration memory can be independently written, so that the circuit defined by the active layer may continue its operation. A *partially reconfigurable* architecture allows a selective reconfiguration of the target device. In a *dynamic* reconfigurable (or RTR) architecture, the parts of the architecture which are not being configured continue execution.

At this time, only a few FPGA vendors support partial and dynamic reconfiguration. One of them, Xilinx, offers the Virtex FPGA family. With this family, partial reconfiguration is possible because internal configuration elements of a device can be individually addressed [9]. Another vendor, Atmel, produces the FPSLIC (*Field Programmable System Level Integrated Circuit*) device, which includes a general-purpose processor, memory, and programmable logic [10]. This *Configurable System on a Chip* (CSoC) supports partial and dynamic reconfiguration through context switching.

III. DESIGN FLOWS FOR PARTIAL RECONFIGURATION

Although there are several types of FPGA devices that support partial reconfiguration, so far there is no integrated design flow that allows to develop complex partially reconfigurable systems. Currently, designers can choose between the following basic design flows: bitstream generation and manipulation with the JBits tool suite, direct bitstream manipulation based on standard FPGA implementation tools, and modular design. These design flows are introduced next.

A. The JBits SDK

The JBits System Development Kit (SDK) has its origin in the *Java Environment for Reconfigurable Computing* (JERC) developed at Xilinx, targeting the XC6200 family of devices, now discontinued. This software environment allowed logic and routing to be configured and macros to be constructed at run time [11]. The XC6200 family allowed run-time reconfiguration and featured an open architecture, with all circuit configuration data available to the users. The next step was the development of SPODE circuit specification library, which allowed full access to all configurable resources within the XC6200 series. The SPODE library functions were called from a C program, which generated a configuration file. Finally, JERC and SPODE were combined to form JERCng (JERC next generation), a Java environment for the XC6200 series.

At Xilinx, the JERC project was transferred to the older family XC4000 and renamed the *Xilinx Bitstream Interface* (XBI) [12]. This environment was later renamed JBits. Since the XC4000 family does not support partial reconfiguration, any changes to the circuit configuration requires to reload the entire configuration bitstream. The slow reconfiguration time is unacceptable for most applications. More recently, the JBits software has been ported to the Xilinx Virtex family of devices, which has architectural support for partial reconfiguration. However, the software support for partial reconfiguration only arrived with the addition of the JRTR API to the Virtex version of the JBits tool suite. JRTR uses combined hardware and software techniques which allow to make small changes to the device configuration data quickly and without interruption of operation [13].

The main components of the JBits SDK are the JBits Application Program Interface (API), the *Run-Time Parameterizable Core* (*RTPCore*) library, the *JRoute* API, the *Java Run-Time Reconfiguration* (JRTR) API, the *Xilinx Hardware Interface* (XHWIF), the BoardScope debugger, and the *VirtexDS* simulator.

The JBits API is a set of Java classes which allow to generate and modify configuration bitstreams for the Xilinx Virtex devices. This API operates either on configuration bitstreams generated by Xilinx synthesis tools, or on bitstreams read back from the actual device [14]. Using the JBits API, all configurable resources of the device can be individually set under software control. Therefore, a dynamic and partial reconfiguration of the Xilinx Virtex devices is possible from a Java application.

The JBits API provides access to all the resources of a Virtex device, including the look-up tables (LUTs) inside each *Configurable Logic Block* (CLB) and the routing resources adjacent to the CLBs. The device architecture is represented as a two-dimensional array of CLBs, and each CLB is referenced by a row and column. This API allows to develop RTR systems in a high-level language.

There are four main functions in the JBits API. The *read()* and *write()* functions allow configuration bitstreams to be read or written. The *get()* function allows to query the state of a programmable logic resource, and the *set()* function allows to set the state of a programmable logic resource to a specified value. The JBits API also contains a series of constants which define each of the programmable resources of the device and the values they can be set to.

Fig. 1 illustrates the JBits design flow. The user-written Java application configures the FPGA device by communicating with the board containing the device. The bitstream input to the Java application can be a null bitstream or a bitstream for an existing design. The application may use the bit-level interface provided by the JBits API, which allows to set or clear a single bit or a group of bits in the bitstream. This is a low-level interface responsible for knowing the bit location in the bitstream of a given configuration data for the devices supported in the Virtex

FPGA family. The bit-level interface interacts with the *Bitstream* class, which manages the device bitstream and provides support for reading and writing bitstreams from and to files. This class can also read back the existing configuration data from the operating device, which is necessary for dynamic reconfiguration.

The user application may also use the *RTPCore* library provided by the JBits SDK. This library is a collection of Java classes defining macrocells or cores that can be dynamically parameterized and relocated within a device. Examples of cores are registers, counters, adders, multipliers and other standard Xilinx Unified Library logic and computation functions. In addition to these primitive cores, other non-primitive RTP cores can be used, which are created by instantiating primitive or non-primitive subcores connected with nets and busses [14].

The JBits tool suite includes the *JRoute* API, which is an automatic router with the ability to dynamically route and unroute connections. However, currently the capabilities of the automatic router are limited and it cannot manipulate some routing resources, such as long lines.

The *Java Run-Time Reconfiguration* (JRTR) API allows small changes to be made directly to the Virtex device. Such changes can be done much faster than with usual methods. The JRTR API keeps track of the changes done in the configuration and only the necessary data is rewritten to the device [13].

The *Xilinx Hardware Interface* (XHWIF) is a standard interface for communicating with FPGA-based boards [15]. It can also be used to communicate with the VirtexDS device simulator. XHWIF contains methods for describing the type and number of FPGA devices on the board, for configuring the devices, for reading back the configuration memory of the devices, for incrementing the on-board clock, and for reading and writing from/to on-board memories, if they are available.

XHWIF provides a portable layer to connect JBits applications to reconfigurable hardware. By using this layer, JBits applications can communicate with a variety of boards. All the hardware specific information is hidden inside of a class that implements the XHWIF interface. Therefore, this interface enables applications to communicate with boards connected through any bus or communication link. Using the *Java Native Interface* (JNI), which allows Java programs to interface with C programs, calls to the interface in the Java language are converted into calls to the board's drivers, usually written in C.

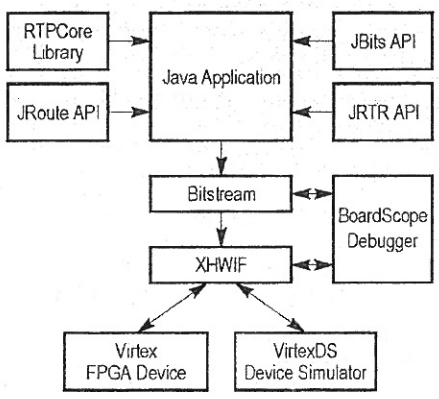

Fig. 1. Design flow using the JBits SDK

The XHWIF interface is implemented as a server application. This server allows other applications to communicate with reconfigurable computing boards located anywhere across the Internet. This capability allows multiple users to access a board and to debug designs using tools such as *BoardScope*, without having direct access to the hardware.

BoardScope is an interactive debug tool for Xilinx FPGA-based hardware [15]. It features a graphical interface for viewing the state of FPGA circuits during operation. The main display of *BoardScope* shows the complete state of any CLB, including flip-flop configuration and look-up table (LUT) values. The waveform display allows to view signals and busses in a way similar to that used by circuit simulators. BoardScope uses the XHWIF interface to communicate with the FPGA-based hardware.

The *Virtex Device Simulator* (*VirtexDS*) is part of the JBits tool suite and provides a software model of the entire Virtex family of FPGA devices. The main advantage of this simulator is that it includes support for RTR [16]. VirtexDS operates at the device level, simulating the actual FPGA device, and therefore provides a high level of simulation accuracy. It also allows to identify illegal configurations that would damage the actual hardware, which is important for debugging RTR applications. The device simulator interface is identical to that of the actual hardware, which permits existing applications, including the *BoardScope* debug tool, to interface directly to the simulator with no modifications.

The main disadvantage of the JBits tool suite is that, up to now, it supports structural design only. Currently, it does not provide support for state machine design, higher-level combinational and sequential synthesis, timing-driven placement and advanced routing. Therefore, a pure JBits design flow is only applicable for simpler or data-flow oriented applications.

B. Direct Bitstream Manipulation

Standard design implementation tools, such as the Bitgen software of the Xilinx ISE design package, can generate full configuration bitstreams, as well as custom bitstreams for small sections of the device. Switching the configuration of a module from one implementation to another can be performed rapidly, as the bitstream differences are small compared to the entire device bitstream. With an appropriate software support, these bitstreams can be loaded quickly into the device.

In [17], examples are given of how to make small changes to a design using the FPGA Editor software included in the Xilinx ISE package. Simple modifications, such as changing LUT equations, changing block RAM contents, or changing I/O standards, are relatively easy. Also, there are other properties of slices, input/output blocks (IOBs), and block RAMs which can be changed by dynamic reconfiguration. However, only those properties or values can be changed that would not impact routing, since otherwise there is a risk of internal contention. This technique can be used, for example, to change the coefficients of a digital filter during run-time.

The main advantage of direct bitstream manipulation is that it can modify full configuration bitstreams generated by standard synthesis and implementation tools [18]. These

tools allow to design at a high level, using hardware description languages (HDLs), and can generate optimized implementations of a design. However, the direct bitstream manipulation design flow has important limitations. For complex designs, the low-level manipulation of bitstreams becomes very complex. In addition, the routing cannot be changed, so that different reconfigurable parts must occupy exactly the same area of the FPGA device, and the interface between the reconfigurable parts and the fixed part must be bound to a fixed location. Moreover, the designer must ensure that the routes for the fixed part do not run through the reconfigurable parts [18]. Because current implementation tools do not allow to specify location constraints on routing resources, constraining the routing usually involves manual intervention.

Several tools are described in the literature for direct bitstream manipulation. James-Roxby et al. describe a tool called *JBitsDiff*, which uses the JBits tool suite to extract circuit information from existing cores [19]. These cores can be created by any available design method. The user defines the bounding box of the core with a floorplanning tool, and *JBitsDiff* produces a JBits core as output. This core can then be inserted into an existing configuration bitstream.

Horta et al. describe in [20] the PARBIT tool, which is also based on direct bitstream manipulation. This tool can be used to transform and restructure bitstreams in order to implement dynamically loadable hardware modules. The PARBIT tool uses the original bitstream, a target bitstream, and parameters specified by the user. These parameters include the block coordinates of the logic implemented on a source FPGA, the coordinates of the area for a partially programmed target FPGA, and the programming options. PARBIT reads the configuration frames from the original bitstream and copies to the partial bitstream only the configuration bits related to the area defined by the user. It can also reallocate the partial reconfigurable area according to the new coordinates specified by the user.

In [21], Dyer et al. present a design tool flow that allows to generate an initial full configuration and a number of subsequent partial configurations. The routing between static and reconfigurable parts is defined using a structure called *virtual socket*. This is a component that provides fixed locations for a set of predefined signals. It is manually placed and routed to guarantee correct connections between cores.

C. Modular Design

The modular design flow has been defined by Xilinx [17]. This design flow is based on the modular design methodology supported by the latest versions of Xilinx ISE package. The modular design flow represents a guideline which should be followed in order to design, implement and dynamically reconfigure portions of Virtex and Virtex-II series of FPGA devices. A series of restrictions are placed on the reconfigurable modules. For example, the height of such a module is always the full height of the device. All logic resources included within the width of a module, including all routing resources, are considered part of the module's bitstream frame. Clocking logic is always separate from the reconfigurable module, since clocks have separate bitstream frames. The implementation should ensure proper operation of the design during the reconfiguration process, using explicit handshaking logic, for instance. It is not possible to use the global set/reset logic available in the Xilinx FPGA devices to independently initialize the state of the reconfigurable module. Instead, user-defined set/reset signals should be defined in the HDL code.

For reconfigurable modules that communicate with each other, a special bus macro is provided by Xilinx, which grants a fixed bus communication between two adjacent modules. Each bus macro provides four bits of inter-module communication. This macro is a pre-synthesized bitstream, which uses fixed routing resources. These resources will not change from compilation to compilation. The communication is done using tri-state buffers. Each time partial reconfiguration is performed, the bus macro should be used to establish unchanging routing channels between modules. Without this bus macro, signals could not cross over a partial reconfiguration boundary, since it would be impossible to guarantee fixed routing between modules. Therefore, in the HDL code the designer should ensure that any reconfigurable module signal that is used to communicate with another module passes through a bus macro.

Modular design allows to use standard design implementation tools, but it also has several limitations. Each bus macro must be physically locked so that its center will be placed on the boundary line between the two reconfigurable modules, and it must be locked in exactly the same position for all compilations. The placement of bus macros can be specified by inserting location constraints in the user constraints file. Another limitation is that currently the bus macro signals cannot be bidirectional or reconfigurable.

IV. DESIGN OF AN EXAMPLE RECONFIGURABLE SYSTEM

In this section we illustrate a design method that combines the bitstream manipulation capabilities of the JBits tool suite with the modular design flow. The advantage of this combined design flow is that the system can be designed at high level using mainstream synthesis and implementation tools, while the bitstream manipulation capabilities of JBits allow to simplify the design compared to direct bitstream manipulation. We use this method to design a dynamic reconfigurable system containing a reconfigurable part and a fix part. The reconfigurable part can be configured as either a convolution filter which uses distributed arithmetic (DA) or a FIR filter. The fix part is used by both filters.

The main steps of the design process are described next.

A. Designing the Individual Modules

The design consists of the following modules:

- The shift register module represents the fix part of the system and it is designed using the JBits tool suite;
- The DA filter is the first reconfigurable module, designed in VHDL;
- The FIR filter is the second reconfigurable module and it is designed using the JBits tool suite.

The shift register is designed in Java as a core derived from the *RTPCore* class. The constructor of the *ShiftRegister* class calculates the dimensions of the core and sets the height and width of the core with the *setHeight()* and *setWidth()* methods. The granularity of the core is at the CLB level, and this granularity (*Gran.CLB*) is set with the *setHeightGran()* and *SetWidthGran()* methods. The ports of the core are created with the *newInputPort()* and *NewOutputPort()* methods. The hardware implementation of the core is performed with the *implement()* method. This method defines the signals and busses with the *Net()* method, connects the ports to signals with the *PortObject.setIntSig()* method, and connects the nets to busses with the *Bitstream.connect()* method.

After designing the shift register core, the actual implementation of the shift register and the generation of its bitstream is performed with the *TestShiftRegister* class. The *main()* method of this class parses the command line to determine the name of the target device, the input bitstream file and the output bitstream file. Then this method instantiates a JBits object based on the command line parameters, calls the *run()* method and writes the configuration into the output bitstream file. The *run()* method defines the global parameters of the core, including the position of the core in the device, creates the nets for the input and output signals, creates the output bitstream file, instantiates the system clock and the shift register core, sets the position of the core, calls the *implement()* method for the clock and the core, and connects the signals.

The DA filter uses distributed arithmetic in order to replace the multiply operations required for the convolution with shift and add operations. Therefore, the hardware resources required for the implementation are significantly reduced. The basics of distributed arithmetic are described in many publications, including [1]. The DA filter was designed in VHDL. The inputs of the filter are the constants representing the filter taps, and the serial data streams of eight bits each. The filter module contains a 2D-shifter to which the input data are applied, a LUT to perform the multiply operations and an adder to sum the partial products. After designing and implementing the DA filter, the JBits tool was used to extract a partial configuration bitstream from the full configuration bitstream generated by the synthesis and implementation tools.

The FIR filter was designed using the JBits tool suite. The filter implements an algorithm which is similar to that of a convolution filter. The tap values of the filter are computed based on the filter type (high-pass or low-pass) and the cutting frequency.

B. Creating the Floorplan

In this phase, we defined the location of the fix module and of the reconfigurable modules, taking into account the constraints specified in the modular design flow [17]. For the floorplanning of the DA filter designed in VHDL, we attached the area group properties that are specific to the partial reconfiguration. These properties were specified in the user constraints file (.ucf). For each bus macro, we entered a separate location constraint into the .ucf file, because the current version of the floorplanner software does not allow to create these constraints. The .ucf file was used during the implementation of the DA filter module. For the other modules created with JBits, the location constraints were specified in the Java code.

C. Implementing the Modules

Each module (fix and reconfigurable) has been implemented separately, and bitstreams were generated for each of them. For the modules designed with JBits, a null bitstream file has been used as input. For the DA filter designed in VHDL, the full configuration bitstream file generated by the implementation tool has been used to generate a partial configuration file with JBits.

D. Assembling the Modules

In this phase, we combined the fix module with each of the reconfigurable modules to create complete designs. This was performed because the partial reconfiguration design flow requires that the initial bitstream loaded into the FPGA device be a complete design. First, we combined the fixed module with the DA filter, preserving the placement and routing achieved during the implementation phase, and we created a full configuration bitstream. Then, we combined the fix module with the FIR filter, creating a second full configuration bitstream. These bitstreams have been used in the next phase for functional verification.

E. Functional Verification

We used the *BoardScope* debugger and the *VirtexDS* device simulator to verify the operation of both possible combinations of modules: the fix module with the DA filter, and the fix module with the FIR filter. After loading an initial full configuration bitstream, we tested the reconfiguration of the filter by loading the partial configuration bitstream of the other filter. Both filters operated correctly.

V. CONCLUSIONS

The first contribution of this paper is the overview and analysis of currently available design flows for reconfigurable systems: bitstream generation and manipulation with the JBits tool suite, direct bitstream manipulation, and modular design. The advantages and drawbacks of each design flow were highlighted. The JBits tool suite allows to create partial bitstreams for run-time reconfiguration, but currently it supports structural design only. Direct bitstream manipulation allows to modify full configuration bitstreams generated by standard synthesis and implementation tools, so that it is possible to design at a higher level. However, for complex systems the low-level manipulation of bitstreams becomes complex, and since the routing cannot be changed, the designer has to ensure that the routing between the fix part and the reconfigurable parts is correct. Modular design allows to use standard design entry and implementation tools, but it requires to use a special prerouted bus macro for communication between modules. The designer has to specify location constraints for the reconfigurable modules, as well as for the bus macros.

The second contribution of this paper is the demonstration of a design method that combines the ability of the JBits tool suite to create and manipulate partial configuration bitstreams with the modular design flow. Therefore, it

is possible to design at a high level, using HDLs and mainstream synthesis and implementation tools for FPGA devices. We illustrated this method for the design of a simple dynamic reconfigurable system containing a reconfigurable part and a fix part. The reconfigurable part can be configured as either a DA convolution filter or a FIR filter.

Reconfigurable architectures have several advantages, combining the flexibility of general-purpose processors with the efficiency of custom hardware. It is estimated that without reconfigurability, many future products will not be competitive. The main problem in this field is the lack of high-level design tools that support partial reconfiguration. A solution would be to combine the design methods which are specific to dynamic and partial reconfiguration with the high-performance mainstream design tools.

VI. REFERENCES

[1] Z. Baruch, "Run-Time Reconfigurable Implementation of DSP Algorithms Using Distributed Arithmetic", in *Proceedings of the 14th International Conference on Control Systems and Computer Science* (CSCS14), 2003, Bucureşti, Romania, pp. C1-C6.

[2] D. Mesquita, F. Moraes, J. Palma, L. Möller, and N. Calazans, "Remote and Partial Reconfiguration of FPGAs: Tools and Trends", in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)* CD-ROM, Reconfigurable Architectures Workshop, 2003, Nice, France.

[3] S. A. Guccione and D. Levi, "The Advantages of Run-Time Reconfiguration", in John Schewel, et. al., editors, *Reconfigurable Technology: FPGAs for Computing and Applications, Proceedings of SPIE 3844*, Bellingham, WA, 1999, pp. 87-92.

[4] S. A. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-Based Interface for Reconfigurable Computing", in Richard Katz, editor, *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[5] R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective", in *Design, Automation and Test in Europe*, 2001, pp. 642-649.

[6] R. R. Vemuri and R. E Harr, "Configurable Computing: Technology and Applications", *Computer*, Vol. 33, No. 4, April 2000, pp. 39-40.

[7] A. DeHon, "The Density Advantage of Configurable Computing", *Computer*, Vol. 33, No. 4, April 2000, pp. 41-49.

[8] M. Sima, S. Vassiliadis, S. Cotofana, J. van Eijndhoven, and K. Vissers, "A Taxonomy of Custom Computing Machines", in *Proceedings of the First Workshop on Embedded Systems and Software (PROGRESS 2000)*, Utrecht, The Netherlands, 2000, STW Press, pp. 87-93.

[9] Xilinx Inc., "Virtex Series Configuration Architecture User Guide", Xilinx Application Note XAPP151, 2000, http://www.xilinx.com/xapp/xapp151.pdf.

[10] Atmel Corp., "Field Programmable System Level Integrated Circuits (FPSLIC)", 2002, http://www.atmel.com/atmel/products/prod39.htm.

[11] E. Lechner and S. A. Guccione, "The Java Environment for Reconfigurable Computing", in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications (FPL 1997)*, 1997, Springer-Verlag, pp. 284-293.

[12] S. A. Guccione and D. Levi, "XBI: A Java-Based Interface to FPGA Hardware", in John Schewel, editor, *Configurable Computing: Technology and Applications, Proceedings of SPIE 3526*, Bellingham, WA, 1998, pp. 97-102.

[13] S. McMillan and S. A. Guccione, "Partial Run-Time Reconfiguration Using JRTR", in R. W. Hartenstein and H. Grunbacher, editors, *Field-Programmable Logic and Applications*, Springer-Verlag, Berlin, 2000, pp. 352-360.

[14] Xilinx Inc., "JBits Tutorial", JBits SDK Version 2.8, 2001.

[15] D. Levi and S. A. Guccione, "BoardScope: A Debug Tool for Reconfigurable Systems", in John Schewel, editor, *Configurable Computing: Technology and Applications, Proceedings of SPIE 3526*, Bellingham, WA, 1998, pp. 239-246.

[16] S. P. McMillan, B. J. Blodget, and S. A. Guccione, "VirtexDS: A Device Simulator for Virtex", in John Schewel, et. al., editors, *Reconfigurable Technology: FPGAs for Computing and Applications II, Proceedings of SPIE 4212*, Bellingham, WA, 2000, pp. 50-56.

[17] D. Lim and M. Peattle, "Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations", Xilinx Application Note XAPP290 (v1.0), 2002, http://www.xilinx.com/xapp/xapp290.pdf.

[18] M. Dyer, C. Plessl, and M. Platzner, "Partially Reconfigurable Cores for Xilinx Virtex", in M. Glesner, P. Zipf, and M. Renovell, editors, *Field-Programmable Logic and Applications (FPL 2002)*, Montpellier, France, 2002, pp. 292-301.

[19] P. James-Roxby and S. A. Guccione, "Automated Extraction of Run-Time Parameterisable Cores from Programmable Device Configuration", in *Proceedings of 8th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2000)*, Napa, CA, 2000, pp. 153-161.

[20] E. L. Horta, J. W. Lockwood, and S. T. Kofuji, "Using PARBIT to Implement Partial Run-Time Reconfigurable Systems", in M. Glesner, P. Zipf, and M. Renovell, editors, *Field-Programmable Logic and Applications (FPL 2002)*, Montpellier, France, 2002, pp. 182-191.

[21] M Dyer, C. Plessl, M. Platzner, "Partially Reconfigurable Cores for Xilinx Virtex", in M. Glesner, P. Zipf, and M. Renovell, editors, *Field-Programmable Logic and Applications (FPL 2002)*, Montpellier, France, 2002, pp. 292-301.