Virtual Java/FPGA Interface for Networked Reconfiguration

*†Yajun Ha, *Geert Vanmeerbeeck, *Patrick Schaumont
*Serge Vernalde, *Marc Engels, †Rudy Lauwereins, *†Hugo De Man
*IMEC, Kapeldreef 75, Heverlee 3001, Belgium.
†EE Department, K. U. Leuven, Kard. Mercierlaan 94, Heverlee 3001, Belgium.

Abstract— A virtual interface between Java and FPGA for networked reconfiguration is presented. Through the Java/FPGA interface, Java applications can exploit hardware accelerators with FPGAs for both functional flexibility and performance acceleration. At the same time, the interface is platform independent. It enables the networked application developers to design their applications with only one interface in mind when considering the interfacing issues. The virtual interface is part of our work to build a platform-independent deployment framework for the networked services. In the framework, both the software and hardware components of services can be platform independently described and deployed.

I. INTRODUCTION

With the platform independent feature provided by the Java technology [1], applications written in Java can "write once, and run everywhere". Java technology is thus considered to be an ideal way to do networked software reconfiguration. But since the extra interpretation work that Java virtual machine (JVM) should do, Java programs normally run slowly compared to their function-equivalent C programs. Currently, two different ways have been proposed to improve the performance of Java applications. One way is to implement the JVM in hardware [2] [3], the other way is to use FPGAs to accelerate the computation-intensive part of the Java applications. The topic discussed in this paper falls in the second domain.

To give you an idea of one of the possible hardware accelerated Java applications, we will use the service of "Web Movie" (see fig. 1) as an example. Suppose there is a client who wants to watch web movies. But he has neither the "Web Movie Player" nor the movie files installed on his side. He goes to the web page of the web movie service provider, where he finds a list of movies that are available. The client then points his mouse to one of the items and chooses his favorite movie. Upon receiving this request from the client, the service provider first detects that the client does not have a movie player. As a result, it first automatically sends the movie player *service bytefile* to the client. After the service bytefile has been successfully installed, the client can use it to playback the movie files provided in the same or another service provider's site.

The web movie service bytefile is the combination of software and hardware bytecodes that realizes the web movie player. Normally, the software bytecodes describe the user interface, hardware/software interface, and all the computationally non-intensive functions of the player. The hardware bytecodes describe the FPGA reconfiguration information that implements the computationally intensive functions like video decoding. By preparing the reconfiguration information in an abstract bytecode format which is abstract enough to be implemented on a wide variety of fixed hardware (processors) and reconfigurable hardware (FPGA platforms), the "Web Movie" service provider only needs to write the service bytefile once, and it can run on any client platform.

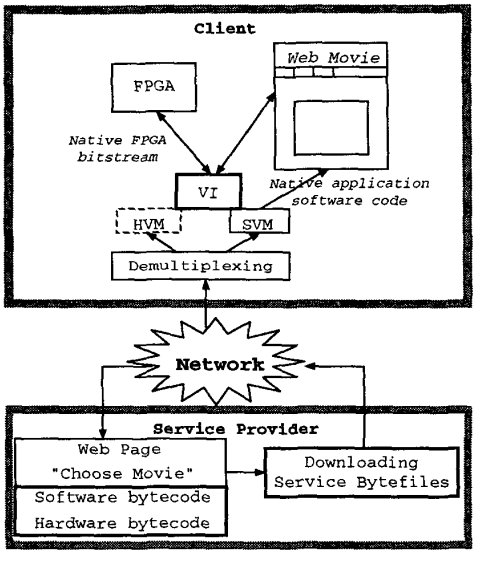

Fig. 1. Deployment of the web movie service in a virtual framework.

To develop such a platform-independent and SW/HW combined service as "Web Movie", we need to build a virtual framework. In the framework, both the software and hardware components of a service can be abstractly described and platform independently deployed. Two issues arise when we are building the framework [5] [6]. First, the hardware design should be described and deployed in an abstract way.

0-7803-6633-6/01/$10.00 ©2001 IEEE.

That could be solved by the introduction of a hardware virtual machine[4]. Second, the interface between the Java and FPGA should also be virtual. That enables the networked application developers to design their application with only one interface in mind when considering the interfacing issues. The second issue is the focused topic of this paper.

In the next section, we will first introduce the functionality and components of the virtual Java FPGA interface. In the section 3, design flow of the virtual interface is described. Finally, the experimental results for a driving example - web MPEG player are discussed in the section 4.

II. Virtual Java/FPGA Interface

A software hardware interface generally sits between the application software and hardware, and makes the data communication between them possible. An example of a normal software hardware interface is shown in the fig. 2(a), where the user bus (UBUS) is platform dependent. It may vary from client to client. For instance. the UBUS may be PCI bus, or be ISA bus. This uncertainty brings difficulty for networked application designers. Because when they begin a design, they don't know what kind of interfaces that their application may encounter on the client side.

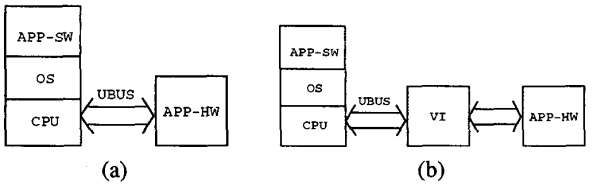

Fig. 2. Overview of normal and virtual software hardware interface (a) normal interface, (b) virtual interface.

To enable networked application developers design their applications with only one interface in mind when considering the interfacing issues, a virtual interface is necessary. The virtual interface sits between the actual physical interface and the application hardware block. As you could see from fig. 2(b), the application hardware block only communicates with the virtual interface. That liberates the application hardware developer from knowing different bus details of different clients.

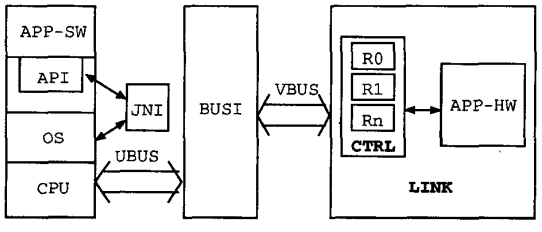

Fig. 3. The proposed virtual Java FPGA interface.

To practically use the virtual interface, it is not enough to only provide the interface hardware. Interface software like device drivers used to access the local FPGA boards should also be provided. At the same time, they should be platform independent. Thanks to the Java technology, we can achieve it by writing FPGA API. The Java written FPGA API contains native C methods that can actually access the FPGA board through the native board device drivers. In other words, the outlook of the FPGA API is uniform like the other Java APIs (image, sound), which are platform independent, but the implementation of those APIs are platform dependent.

Our proposed implementation of virtual interface is shown in fig. 3. Since in our case the software is Java, and the hardware block is implemented in FPGAs, we also call the virtual interface as virtual Java FPGA interface (VJF). The proposed VJF interface consists of two main parts. The virtual interface hardware and Java FPGA API software. The virtual interface hardware consists of the virtual port access, the virtual internal bus (VBUS) and its interface (BUSI). The Java FPGA API software consists of the Java native driver interface. We will discuss their functions in the following subsections respectively.

A. Virtual Port Access

When locally implementing application hardware in FPGAs, the I/O ports of the hardware block usually are bound to specific FPGA pins. The pin scheduling is made based on the system physical constraints (e.g. PCB board) and available FPGA pins. The hardware blocks communicate with the outside through those bound pins. But in networked reconfiguration, it is very difficult for application designers to know beforehand the availability of specific FPGA pins. A virtual way to access the application ports is necessary.

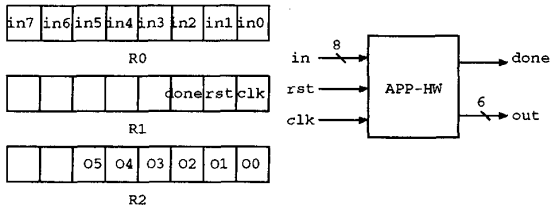

Fig. 4. Register allocation for ports of the application hardware block.

A flexible solution is adopted to solve this problem. It is a layered approach to access the application ports. The application ports are not directly bound to FPGA pins anymore. Instead, they are first mapped to FPGA internal registers, and then accessed from those mapped registers. The FPGA internal registers can be addressed by the virtual bus which will be introduced later. Fig. 4 shows an example of register allocation for ports of the application hardware block.

In the layered approach, a control entity CTRL and a glue entity LINK are created (as in fig.3). The CTRL entity does

virtual bus communications and maps the incoming and outgoing data to the right application virtual I/O ports. The LINK then glues the CTRL's virtual ports to the user's application port. In fig. 5 and fig. 6, the VHDL files for CTRL and LINK are given for the example hardware block. The hardware block is the one that had been used as an example in fig. 4.

In the LINK code, CTRL and APP-HW are hooked up by instantiating a component of each and mapping them in such a way that the ports of APPL are connected to the corresponding ports on CTRL.

B. Bus Interface/FPGA Internal Bus

Different clients may use different kinds of buses to communicate between host PCs and peripheral boards. To abstract the bus concept and hide their platform dependent details, a virtual internal bus (VBUS) is built. The VBUS defines its own address and data buses, read and write commands. Through it, arbitrary register in the LINK entity can be addressed, so as to access the I/O ports of the application hardware block. The bus interface (BUSI) translates the user bus signals into the VBUS signals. The VBUS and and BUSI are shown in fig. 7.

C. Java Native Driver Interface

Device drivers are pieces of software that act as the entry points for the communication between application's software and hardware. Since they are easily influenced by the changes of either software or hardware platforms, device drivers are normally very platform dependent. In this work, we write a portable device driver using the Java Native Interface (JNI)[8].

As shown in fig. 8, JNI is a glue to link the C and Java language. Java programmer can use it to write Java native methods that are implemented by C, so as to utilize some platform dependent features.

Deployed together with the Java Virtual Machine, a standard Java class library is enclosed in the Java development Kit (JDK) package. Example Java application programming interfaces (API) in the standard library include Sound API, Image API. In those APIs, some of the methods are written in C (through JNI) instead of pure Java language, because they need to access the local hardware resources.

Similarly, a Java FPGA APIs had been developed for the Java FPGA interfaces (as shown in fig. 9). The declaration of the Java FPGA API is platform independent, and with the same outlook across the platforms. But the real implementations of those FPGA APIs are different from platform to platform. They are written in C language with the help of JNI.

D. Application Level Handshaking

Since the application software and hardware normally run with different clock speed, handshaking protocol is necessary when communicating data between a software processor and a hardware processor (see fig. 10). Handshaking protocol may vary from application to application. But since the handshaking protocol is developed together with the hardware block,

```
entity CTRL  is
   port(clk : in std_logic;
        reset : in std_logic;
        address : in std_logic_vector ( 7 downto 0 );
        data : inout std_logic_vector ( 7 downto 0 );
        chip_select : in std_logic;
        io_read : in std_logic;
        io_write : in std_logic;

appl_in : out std_logic_vector ( 7 downto 0 );
        appl_clk : out std_logic;
        appl_reset : out std_logic;
        appl_done : in std_logic;
appl_out : in std_logic_vector ( 5 downto 0 ));
end CTRL ;

architecture BEHAVIOR of CTRL  is

    CONSTANT REGISTER_0  : integer := 0;
    CONSTANT REGISTER_1  : integer := 1;
    CONSTANT REGISTER_2  : integer := 2;

    SIGNAL ReadEnable    : std_logic;
    SIGNAL WriteEnable   : std_logic;

    SIGNAL C_Buffer0     : std_logic_vector (7 downto 0);
    SIGNAL C_Buffer1     : std_logic_vector (2 downto 0);
    SIGNAL C_Buffer2     : std_logic_vector (5 downto 0);

begin

-------------------------------------------------------------
    ReadEnable  <= io_read  AND chip_select;
    WriteEnable <= io_write AND chip_select;
    ----- Assign signals ...
    appl_in    <= C_Buffer0;
    appl_clk    <= C_Buffer1(0);
    appl_reset    <= C_Buffer1(1);

    C_Buffer1(2)    <= appl_done;
    C_Buffer2    <= appl_out;

-------------------------------------------------------------
----- Writing to signals...
     writeaccess: PROCESS ( clk )
     begin
IF ( clk='1' AND clk'EVENT ) THEN
 IF ( WriteEnable = '1' ) then
    case ( conv_integer(address) ) is
when REGISTER_0 => C_Buffer0  <= data(7 downto 0);
when REGISTER_1 => C_Buffer1  <= data(2 downto 0);

when others     => null;
    end case;
 END if;
END if;
     END process writeaccess;
-------------------------------------------------------------
      ----- Reading from signals...
    readaccess: process ( clk )
    begin
       IF ( clk='1' AND clk'EVENT ) THEN
if ( ReadEnable = '1' ) then
    case conv_integer(address) is
    WHEN REGISTER_1 => data(2 downto 0) <= C_Buffer1;
    WHEN REGISTER_2 => data(7 downto 0) <= C_Buffer2;

when others     => data <= (7 downto 0 => 'Z');
    end case;
else
    data <= (7 downto 0 => 'Z');
        END if;
       END if;
     END process readaccess;

end BEHAVIOR;
```

Fig. 5. Example VHDL code of CTRL.

```
entity LINK is
   port(clk : in std_logic;
        reset : in std_logic;
        address : in std_logic_vector ( 7 downto 0 );
        data : inout std_logic_vector ( 7 downto 0 );
        chip_select : in std_logic;
        io_read : in std_logic;
        io_write : in std_logic;
       );
end LINK;

architecture STRUCTURE of LINK is
      ...
      signal APPL_in : std_logic_vector  ( 7 downto 0 );
      signal APPL_clk : std_logic;
      signal APPL_reset : std_logic;
      signal APPL_done : std_logic;
      signal APPL_out : std_logic_vector ( 5 downto 0 );

      ...

      CTRL  port map
      (
      ...
      appl_in => APPL_in,
      appl_clk => APPL_clk,
      appl_reset => APPL_reset,
      appl_done => APPL_done,
      appl_out => APPL_out
      );

      APPL port map
      (
      in => APPL_in,
      clk => APPL_clk,
      reset => APPL_reset,
      done => APPL_done,
      out => APPL_out
      );
end STRUCTURE;
```

Fig. 6. Example VHDL code of LINK.

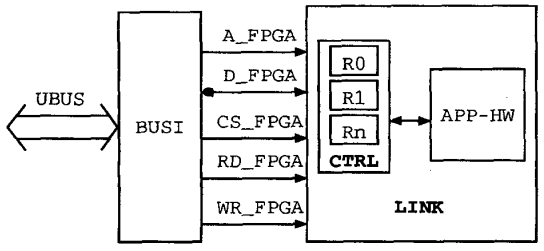

Fig. 7. Virtual internal bus between the bus interface and test FPGA.

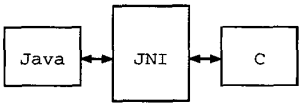

Fig. 8. function of the Java native interface.

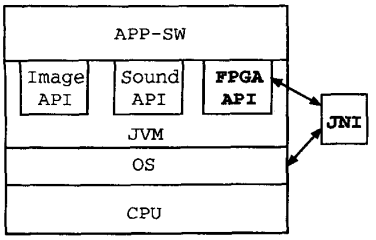

Fig. 9. JNI enabled FPGA API.

that application specific feature will not influence the platform independent implementation of the virtual interface.

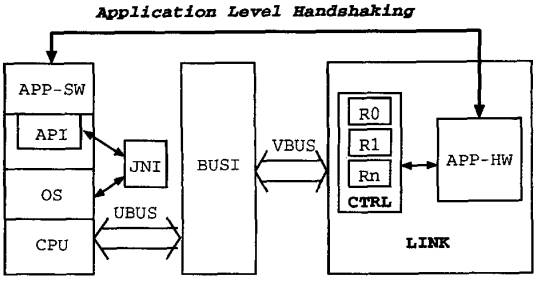

Fig. 10. Application level handshaking.

III. Interface Design Flow

To make the virtual Java FPGA interface work, both the service provider and client should do some work. But most of the work is application independent, they normally only need to be done for once. The interface design flow will be first introduced for the service provider side, and then the client side.

A. On the service provider side

There are four major steps (shown in fig. 11) on the service provider side to deploy an application using the virtual Java/FPGA interface. First, the application software and hardware blocks should be co-designed. The results of this phase are the software Java source code and hardware VHDL code. Then, ports of the hardware block will be allocated to their mapped register by running a program or doing manually (see descriptions in the section 2A). Each hardware port is assigned to one or more registers in the CTRL entity, based on how many bits it is wide. As a result, the port is then accessible through correct addressing of the virtual internal bus. When the register allocation results had been obtained, according to requirements of the hardware block, handshaking protocol can be written (see descriptions in the section 2D) to do the data communicate between the Java software and FPGA. Finally, a

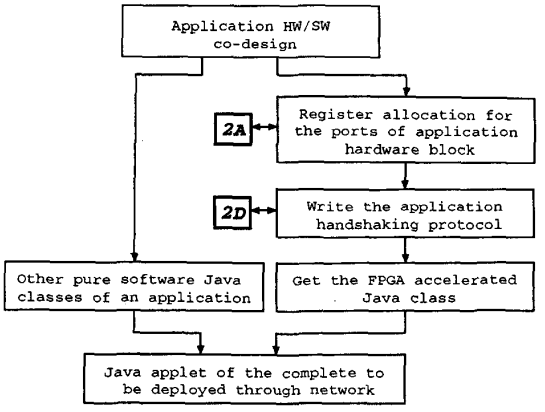

Fig. 11. Design flow on the service provider side.

FPGA accelerated Java class is ready, which works seamlessly with other software Java classes.

B. On the client side

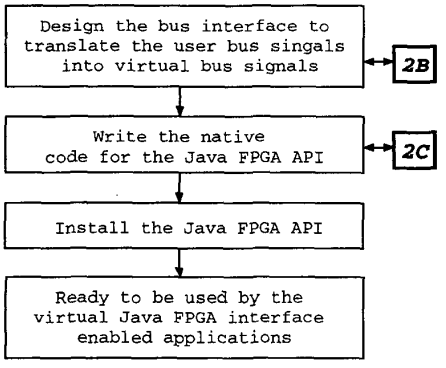

Fig. 12. Design flow on the client side.

The procedure to enable the virtual Java FPGA interface on the client side includes: first, design the bus interface to translate the user bus signal into virtual bus signals (see descriptions in the section 2B). The bitstream of bus interface design is then downloaded to the interface FPGA for configuration; second, write the native C code for the implementation of the Java FPGA API (see descriptions in the section 2C); third, install the Java FPGA API on your host PC. All the three steps are totally application independent. They only need to be done when your installing the FPGA board at the first time.

IV. Experimental Results

The virtual Java FPGA interface had been experimented for a Java MPEG video player on the APS-X208 board [7]. The APS-X208 board is a multipurpose FPGA development board, which can be plugged into the PC ISA bus. The board consists of two FPGAs, the interface FPGA and the test FPGA. The interface FPGA acts as the bus interface and handles interfacing between the PC ISA bus and virtual bus. The test FPGA is reconfigured to realize the custom functions of application hardware blocks. A standard Linux character device driver [9] is written as a loadable module for the board.

A MPEG-1 video player [10] written entirely in Java is used as the starting point of our experiment. The Java source code of this player was first studied, and part of its inverse discrete cosine transform (IDCT) decoder was chosen to be implemented in the FPGA. Because IDCT is normally considered to be the most computation intensive part of the decoder. This part of Java source code of the IDCT was then rewritten in C and fed into a C++ hardware system design environment OCAPI [11]. In OCAPI, an untimed floating-point description is gradually refined into a cycle-true fixed-point description, which later is automatically translated into synthesizable RT-level VHDL codes. The data communication between Java MPEG player and FPGA IDCT block is gone through the virtual Java FPGA interface. A snap of the FPGA accelerated MPEG-1 video player is shown in fig. 13.

Fig. 13. FPGA accelerated Java MPEG-1 video player.

Since the limited capacity of the test FPGA (XC4028EX) on the APS-X208 board, it is unlikely to implement the whole IDCT algorithm into the test FPGA. Therefore, only part of the IDCT was implemented. Because we are experimenting the virtual Java FPGA interface, instead of designing high performance IDCT block, the current version of MPEG player is still a good reference example to show how a networked application can be deployed. Based on the obtained knowledge on APS-X208 board, we will migrate the same application to a prototyping board containing big FPGAs (like Xilinx Virtex series). Performance optimization of this interface will also be further explored.

V. Summary and Conclusions

In this paper, we presented a virtual Java/FPGA interface which is suitable for networked reconfiguration. Through the use of virual port access, virtual bus interface and JNI-enabled virtual device driver, the new interface helps the Java application designers to utilize FPGA in the networked environment. Creating the Java/FPGA interface is part of our efforts to build a platform independent deployment framework for networked services. In the framework, both the software and hardware components of services can be platform independently described and deployed.

Acknowledgements

The authors would like to thank Erik Watzeels for writing the Linux device driver for the APS-X208 board, Radim Cmar and Miro Cupak for supporting the OCAPI design environment. They all are from the DBATE group of DESICS division in IMEC, Belgium.

References

[1] B. Venners, " Inside the Java 2 virtual machine," *McGraw-Hill,* 1999.

[2] Sun Microsystems, "picoJava-II: Java processor core," *Sun Microsystems white paper*, April 1998.

[3] K. B. Kent, M. Serra, "Hardware/Software co-design of a Java virtual machine," *Proceedings of the 11th International Workshop on Rapid System Prototyping*, pp. 66-71, Paris, June 2000.

[4] Y. Ha, P. Schaumont, M. Engels, S.Vernalde, F. Potargent, L. Rijnders, and H. De Man, "A hardware virtual machine to support networked reconfiguration," *Proceedings of the 11th International Workshop on Rapid System Prototyping*, pp. 194-199, Paris, June 2000.

[5] Y. Ha, P. Schaumont, L. Rijnders, S.Vernalde, F. Potargent, M. Engels, and H. De Man, " A scalable architecture to support networked reconfiguration," *Proceedings of IEEE ProRISC*, pp. 677-683, The Netherlands, November 1999.

[6] Y. Ha, S.Vernalde, P. Schaumont, M. Engels, and H. De Man, "Building a virtual framework for networked reconfigurable hardware and software objects," *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, June 2000.

[7] " APS-X208 FPGA test board user's guide," Associated Professional Systems Inc., Dec 1998.

[8] S. Liang, " The Java native interface: programmer's guide and specification," *Addison-Wesley,* 1999.

[9] Alessandro Rubini, " Linux device drivers," *O'Reilly & Associates,* 1998.

[10] J. Anders, " Inline MPEG-1 player in Java (with MPEG layer I decoder)," *http:// rnvs.informatik.tu-chemnitz.de/ ja/MPEG/ MPEG_Play.html.*

[11] P. Schaumont, S.Vernalde, L. Rijnders, M. Engels, and I. Bolsens, "A programming environment for the design of complex high speed ASICs," *Proceedings of the 35th Design Automation Conference*, pp. 315-320, June 1998.