Development of a Design Framework for Platform-Independent Networked Reconfiguration of Software and Hardware

Yajun Ha[1*2], Bingfeng Mei[1*2], Patrick Schaumont[1],
Serge Vernalde[1], Rudy Lauwereins[1], and Hugo De Man[1*2]

[1] IMEC, Kapeldreef 75, Leuven 3001, Belgium,
yjha@imec.be,
WWW home page: http://www.imec.be
[2] Katholieke University Leuven, Department of Electrical Engineering,
Kasteelpark Arenberg 10, Leuven 3001, Belgium

Abstract. The rapid development of the Internet opens wide opportunities for various types of network services. Development of new network services need the support of a powerful design framework. This paper describes such a design framework that can help service providers to build platform independent hardware-software co-designed services. Those new services consist of both software and hardware components, which can be reconfigured through the network. The new design framework can be considered as a Java framework with a hardware extension. Part of the measurement results and an application demonstrator are given.

1 Introduction

Many of us are familiar with software networked reconfiguration, of which Java technology is a good example. But in some applications, especially quality of service (QoS) oriented ones, it is required to extend networked reconfiguration to hardware [5].

Enabling hardware to be networked reconfigurable brings new challenges to the network service designers and EDA tool vendors. Of those challenges, support for platform independent hardware design becomes a very important and difficult requirement.

A lot of industry and academic efforts have already been put in the development design tools for network-oriented reonfiguration. On the one hand for the industry category, design frameworks like Java [1] can support platform independent designs, but only with software components. Xilinx Online [8] is devoted to enable any network connected Xilinx programmable system that can be modified after the system has been deployed in the field. But basically, it is a vendor specific approach, not a platform-independent one. On the other hand for the academic category, virtual hardware and circlets concepts had been raised by Gordon[2][3]. The goal of circlets is to find a virtual description of circuits and integrated circlets with applets, which is quite similar to us. But the abstraction level that circlets work at is too low, and it seems far away from practical realization from the engineering point of view. JHDL [4] proposes to use general-purpose programming languages Java for FPGA design. Their work is focused

on using Java to describe hardware, but not to solve platform-independent issues for networked hardware reconfiguraion. JBits [12] is currently developed in Xilinx, and it is a set of Java classes that provide an API to access Xilinx FPGA bitstreams. It is expected that networked hardware reconfiguration will be benefited from JBits.

This paper describes an extended Java design framework. In the design framework, service providers can build platform independent hardware-software co-designed services. Section 2 introduces the challenges and the proposed solutions for the networked reconfiguration. An overview of the platform independent design framework is then described. Next we introduce two of the main components of the framework: the hardware virtual machine and the virtual SW/HW interface. In that two sections, experimental results of hardware bitstream and bytecode sizes are compared, and performance measurement of the virtual SW/HW interface is given. Finally, a web MPEG player is introduced to demonstrate the design flow of this framework.

2 Problem Definition

In networked reconfiguration, the reconfiguration task is done in two steps and in two geographic sites: (1) a design is described on the server side. (2) clients pick up the design description from the server and implement it on their local reconfigurable platforms. Different clients use different reconfigurable platforms, that include CPU, FPGA and interface.

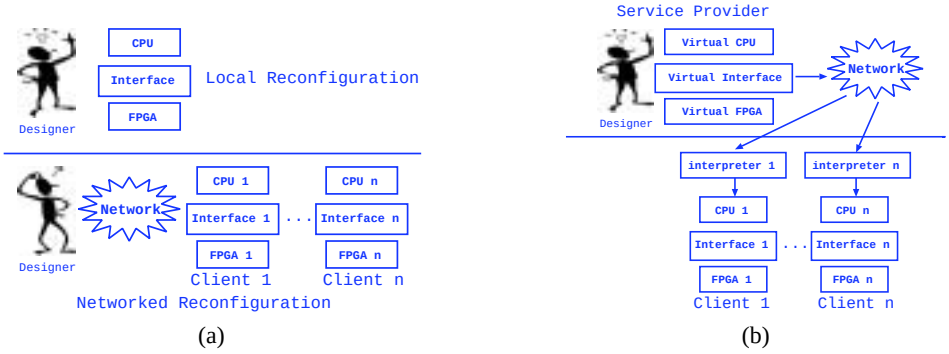


Fig. 1. Challenges and solutions for the networked reconfiguration. (a) Challenges (b) Solutions.

But how do the challenges relate to networked reconfiguration? Let us first make a comparison between local reconfiguration and networked reconfiguration (see fig. 1(a)). In local reconfiguration, designers know exactly what CPU, interface and FPGA their platforms use. But in networked reconfiguration, this is not the case. Designers and implementation platforms are isolated in two geographic sites, and designers do not know the implementation platform details of their clients.

To cope with this problem, a design framework is being built to provide the service designer a simpler design environment. In the framework, the service designers only need to design towards to a single virtual platform. To build that single virtual platform, three abstractions are necessary for each of the three main platform components (CPU, Interface, FPGA) as shown in fig. 1(b). This means that we need a virtual representation of the CPU, interface, and FPGA. Since virtual CPUs like Java virtual machines [1] have already been commercially available, our current work is focused on developing a virtual FPGA and a virtual interface.

3 Design Framework Overview

An overview of the design framework will be given in this section by going through a complete design flow. The design flow consists of server and client components.

3.1 Design Flow on the Service Provider Side

On the service provider side, as the first stage, the service is represented by a functional model. This functional model will be partitioned into three sub-models in a hardware/software codesign environment as CoWare [9]. One sub-model describes the application software part. Another describes the application hardware part. The third describes the interface between the partitioned hardware and software sub-models. This interface sub-model contains both a software part and a hardware part.

The software sub-model and the software part of interface go through various phases of software development, which generates the detailed source code to implement the two sub-models. The detailed source code is then precompiled into software bytecode by a software virtual machine precompiler.

The hardware sub-model and the hardware part of interface are fed into a hardware design environment. After behavioural synthesis, the sub model for the hardware will be transformed into a structural register transfer level (RTL) hardware description. The hardware bytecode can be obtained by floorplanning this netlist on the abstract FPGA model.

Both hardware and software bytecodes are sent to the service bytecode binder, which produces the combined service bytefiles.

3.2 Design Flow on the Client Side

On the client side, the received service bytefile is first demultiplexed to software and hardware bytecode respectively. Software bytecode is interpreted by the software virtual machine and turned into *native application software code* that runs on the native CPU. On the other hand, the hardware bytecode is interpreted by the hardware virtual machine, and turned into *native FPGA bitstreams* that will configure the native FPGA. A HW/SW interface will first be defined via the virtual interface API calls. Through this defined interface, native FPGA bitstreams will be sent to the FPGA for reconfiguration. The reconfigured FPGA can then be used as a hardware accelerator. The native application software code interacts with the FPGA accelerator through the virtual interface.

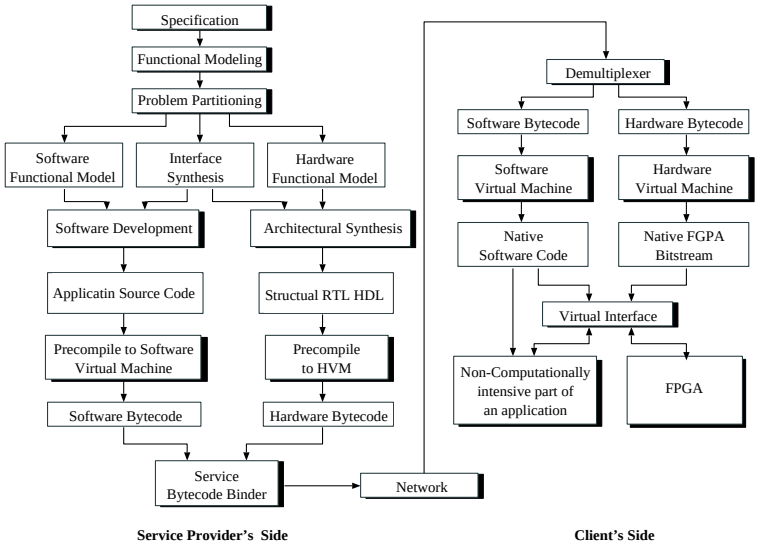


Fig. 2. Design flow in the framework

4 Hardware Virtual Machine

Initially, as for a choice of FPGA design flow for networked hardware reconfiguration, there are three options for the FPGA mapping tools [6]. Firstly, the mapping tools can be totally put on the service provider side. This is the strategy Xilinx Online [8] adopted. It is easy to implement in theory, but very troublesome in maintenance. As a second choice, the mapping tools can be totally put in the client's side. It is an easy way both for implementation and maintenance (from the point of view of the service provider), but too expensive for the terminals. Finally, as a third choice, mapping tools can be separated into two parts, partially on the service provider side (Map_S), and partially on the client side (Map_C). Let us define the Map_C block as *hardware virtual machine* (HVM), while Map_S block as *HVM-compiler*. As benefits of this approach, service providers only need to maintain a single or few FPGA CAD tools to distribute and update their new services, while at the same time, the client does a reasonable portion of the mapping task.

4.1 Logic Vs RTL HVM

Once we decide to use the HVM approach for the service deployment, the next question is at which level should the HVM separate the design flow. A traditional FPGA design flow is shown in fig. 3. By separating the design flow at different levels, different HVMs can be obtained.

The level that a HVM belongs to is mainly depended on the level of abstract FPGA model that the HVM used. On the server side, each design will first be mapped on a

abstract FPGA model, and then interpreted by its corresponding HVM on the client side. We will introduce two different level abstract FPGA models in the next two paragraphs.

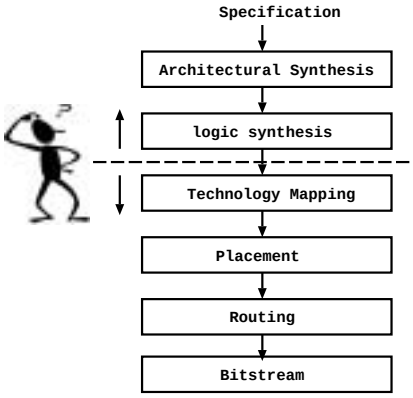


Fig. 3. Abstraction level consideration for hardware virtual machine.

Fig. 4(a) shows a logic level abstract FPGA model. Corresponding to commercial FPGAs, the abstract logic level FPGA model also contains three blocks. They are abstract logic block, abstract routing architecture, and abstract I/O pads [6]. Applications are first mapped and pre-placed and routed onto this abstract FPGA architecture. Then on the client side, the pre-placed and routed bytecode will be converted into local bitstream.

Fig. 4(b) shows a register transfer level abstract FPGA model. It consists of two parts: datapath and controller. The datapath is composed of different RTL level cores, e.g. ALU, ACU, multiplier, adder. The controller is based on microcode. Architectural synthesis tools like Frontier A|RT Designer [15] will be used to generate the RTL level structural VHDL netlist from the high level specification. In this structural VHDL netlist, the datapath is described by a netlist of datapath cores, while the controller is described by microcode, allowing for a flexible RAM implementation.

By using the RTL-level abstract FPGA model instead of the logic level one, RTL-level HVM is better than logic level HVM from several different aspects. From the bytecode size point of view, the RTL HVM bytecode size is much smaller than that of the logic level HVM, since less design information is contained at the RTL-level. The logic level bytecodes size normally are in the magnitude of MBits, wheras the RTL-level bytecodes are of the order of KBits (as shown in the table. 1). Besides, the RTL HVM enables those architectural specific features like carry chain to be used in its library cores, while logic level HVM does not. Moreover, because there are less net connections in the RTL-level than in the logic level, the run time interpretation of RTL HVM is much faster than that of the logic level HVM.

Bitstream, logic and RTL level bytecodes have been obtained for 5 large MCNC benchmark circuits [10]. The bitstream sizes are calculated according to XC4000 series FPGAs. The logic and RLT level bytecodes are obtained using VPR tools [11].

Circuit	#Gates	#FF	Bitstream (bits)	logic HVM Bytecode (bytes)	RTL HVM Bytecode (bits)
bigkey	4675	224	95,008	324,638	46,418
clma	22136	33	422,176	1916,162	189,931
elliptic	9475	1266	178,144	712,704	86,317
s38417	16911	1463	329,312	112,296	144,280
spla	10182	0	247,968	880,228	81,700
Average			254,521	789,205	109,729

Table 1. Bytecodes vs bitstreams comparison

Of which, logic bytecodes are based on their routed netlists, while RTL bytecodes are based on the placed netlist. Since the placed netlist of those benchmark circuits have already been at the gate level, therefore the bytecode sizes of their RTL level floorplanned netlists will be even smaller.

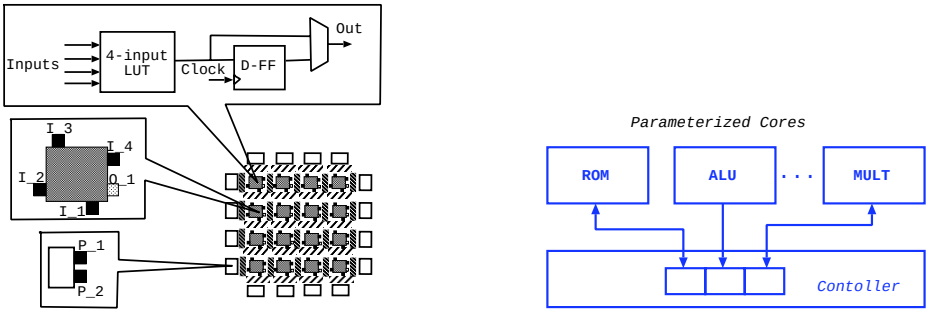


Fig. 4. Logic and register transfer level abstract FPGA. (a) logic level, (b) register transfer level.

4.2 Implementation Flow for RTL HVM

In the hardware virtual machine implementation, JBits software [12] will be highly relied on. JBits software is a set of Java classes which provide an Application Programming Interface (API) to access the Xilinx FPGA bitstream. The interface can be used to construct complete circuits and to modify existing circuits. In addition, the object-oriented support in the Java programming language has permitted a small library of parameterizable, object oriented macro circuits or Cores to be implemented [13]. There is also a run-time router JRoute [14] to provide an API for run-time routing of FPGA devices.

Fig. 5 shows the implementation flow for the register transfer level hardware virtual machine. The circuit to be implemented is first processed to be described by a

combination of datapaths and microcode-based controllers with architectural synthesis tools like Frontier A|RT Designer [15]. Both datapaths and controllers are considered as cores which can be abstractly described on the server side. The real implementations of those cores are provided in the core library on the client side.

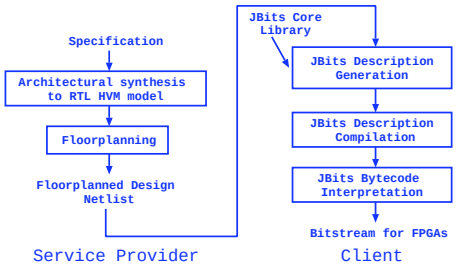


Fig. 5. Implementation flow for register transfer level hardware virtual machine.

On the client side, the hardware virtual machine maps each macro-netlist item into a JBits RTPCore module call. The floorplan and platform dependent information are used in the mapping process. After that, the platform independent macro-netlist is translated into a specific JBits Java program, which can finally create the FPGA specific bitstream.

4.3 JBits description generation

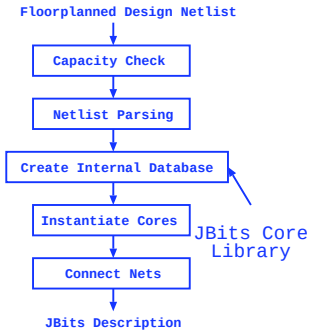
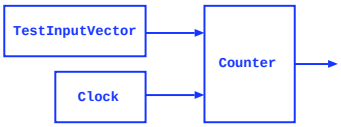


Fig. 6. JBits description generation.

The central task of the hardware virtual machine is to interprete the hardware bytecode and generate its corresponding JBits description. As shown in fig. 6, the JBits program generation starts from the capacity check of the floorplanned design netlist, which checks that whether the local platform is big enough to accommodate to the application to be downloaded. The second step is to create the internal database by parsing




```
.block sample
.outputs cout
.core Counter counter ce clk cout
Basic.Counter 2 4 counter cp
.core Clock clock clk
Basic.Clock -1 -1 clock clk
.core TestInputVector tv clk ce
TestGeneration.TestInputVector 2 2 tv 16 clk tvout
.end
```

(a) (b)

Fig. 7. Example for JBits description generation. (a) RTL level structural view of the counter to be implemented, (b) floorplanned netlist description obtained from its hardware bytecode.

the netlist. This database is based on the JBits core library, where the JBits description for each of the RTL-level cores is stored. From the information in the database, cores are instantiated, and nets between cores are specified. The outcome of those transformation steps is a full JBits description (program) that can be used to generate the final bitstream in seconds.

Fig. 7 gives an example of JBits program generation. To implement a small design as shown in fig. 7(a), a floorplanned netlist like in fig. 7 is first extracted from its bytecode. When this netlist is fed into hardware virtual machine, a JBits program will be generated. This program will be later compiled by the Java precompiler and executed by Java virtual machine to get the bitstream. In this counter example, it takes only 7 seconds to get the bitstream from its JBits bytecode.

5 Virtual Interface

To enable network application developers design their applications with only one interface in mind when considering the interfacing issues, a platform independent virtual interface as shown in fig. 8(a) has been previously worked out in [7]. The virtual interface has two functions: Firstly it provides platform independent API calls to prepare a configuration interface for the FPGAs to be configured. Secondly it provides platform independent API calls for communication (reading and writing) between the software implemented on CPU and the hardware implemented on FPGA.

The virtual interface consists of software and hardware parts. As you can see from fig.8(a), the local HW/SW interface has been isolated by the virtual software and hardware interface respectively. The software (Java program) only communicates with the virtual software interface, wheras the hardware (FPGA) only communicates with virtual hardware. Nevertheless the real communication is still done through the local SW/HW interface. This solution abstracts as much as possible platform specific details from the application designer, at the expense of some performance sacrifice for the interface. The measurments for reading and writing operations with and without virtual interface are given in table. 2. We saw around 10 times penality for the virtual interface with respect to specially designed interface. (The measurements are done by HPjmeter for reading and writing with virtual interface, and gprofiler for reading and writing without virtual interface).

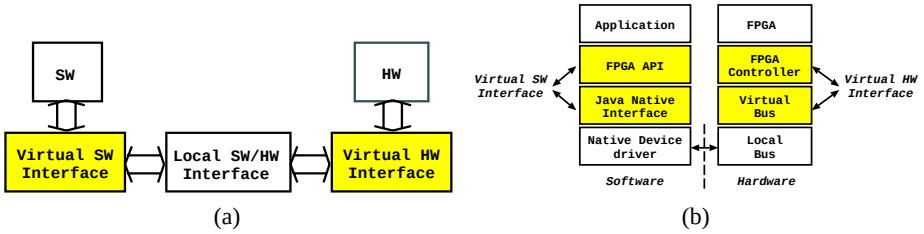


Fig. 8. (a) overview of the virtual software hardware interface, (b) detailed diagram of the virtual Java FPGA interface.

	time per write (us)	time per read (us)
without virtual interface	23	27
with virtual interface	214	381
penalty	191	354

Table 2. Performance measurement for virtual interface

A more detailed high-level overview of the FPGA API implementation is shown in fig. 8(b). On the software side, FPGA API and Java Native Interface constitute the virtual software interface. On the hardware side, virtual bus and FPGA controller constitute the virtual hardware interface.

6 Case Study

A web MPEG player demonstrator had been implemented on a APS-X208 board [16]. In the demonstrator, a co-designed MPEG player is designed and deployed in the framework. IDCT decoder of the player is partitioned to hardware, wheras all the other functions are partitioned to software. Bitstream of the IDCT and Java bytecodes for the rest of the MPEG player are put on the http server. On the client side, a client can visit the server page, and download the co-designed MPEG player and implement it on his own platform. On his platform, the virtual interface had been installed.

This web MPEG player is constructed to demonstrate the design flow of our framework. But in the current demonstrator, the HVM part of the design flow is bypassed, and only the bitstream for the IDCT block are transmitted with Java bytecode. Our final goal is to transmit hardware bytecode with Java bytecode, instead of bitstream in this demonstrator. For further details of the demonstrator, please refer to [7].

7 Current Status and Future Work

The framework structure as shown in fig.2 has been established and functions for each of the components in the framework have been identified. The virtual interface for the

framework has been designed and a MPEG movie player demonstrator has been constructed to show the framework design flow [7]. Logic level HVM has been investigated, and its bytecode format and sizes have been obtained and reported in [6]. As for the RTL-level HVM design flow, the floorplanned netlist for high level specifications can be generated, and we are currently working on the JBits description generator. In the future, the JBits core library will be built for the core library of architectural synthesis tool Frontier A|RT Designer, and the RTL HVM implementation flow will be evaluated by using it to design a complete application.

8 Conclusions

A framework that enables networked reconfiguration of both software and hardware is presented. In the framework, networked reconfiguration users only need to develop a single service description targeted on a single abstract software and hardware platform. The SW/HW co-designed services developed in the framework allow a write once run everywhere scheme. The framework is a new Java platform with a hardware extension.

References

1. The source for Java technology.(http://java.sun.com)
2. G. Brebner: Circlets: Circuits as applets. Proceedings of FCCM. (1998)
3. G. Brebner: A virtual hardware operating system for the Xilinx XC6200. Proceedings of 6th FPL. Springer LNCS 1142 (1996) 327-336
4. B. Hutchings, and B. Nelson: Using general-purpose programming languages for FPGA design. Proceedings of DAC. (2000)
5. 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 PDPTA. **6** (2000) 3046–3052
6. 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 RSP (2000) 194–199
7. Y. Ha, G. Vanmeerbeeck, P. Schaumont, S.Vernalde, M. Engels, R. Lauwereins, and H. De Man: Virtual Java/FPGA Interface for Networked Reconfiguration. Proceedings of ASP-DAC. (2001) 558–563
8. R. Sevcik, Internet Reconfigurable Logic, white papers, Xilinx Inc, 1999.
9. CoWare Software. (www.coware.com)
10. S. Yang: Logic Synthesis and Optimization Benchmarks, Version 3.0. Technical Report, Microelectronics Center of North Carolina. 1991
11. V. Betz and J. Rose: VPR: A New Packing, Placement and Routing Tool for FPGA Research. Int. Workshop on Field-Programmable Logic and Applications. 1997 213–222
12. S. Guccione, D. Levi and P. Sundararajan: JBits: Java based interface for reconfigurable computing. Proceedings of 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference
13. S. Guccione, D. Levi: Run-Time Parameterizable Cores. Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications. (1999) 215–222
14. Eric Keller: JRoute: A Run-Time Routing API for FPGA. Proceedings of the Reconfigurable Architecture Workshop. (2000) 874–881
15. Frontier Design. (www.frontierd.com)
16. APS-X208 FPGA test board user's guide. Associated Professional Systems Inc. Dec 1998.