Ubiquitous Access to Reconfigurable Hardware: Application Scenarios and Implementation Issues

Leandro Soares Indrusiak [1,2], Florian Lubitz [2], Ricardo Reis [1], Manfred Glesner [2]
[1] *Instituto de Informática, UFRGS, Porto Alegre, Brazil*
[2] *Microelectronic Systems Institute, TU Darmstadt, Darmstadt, Germany*
E-mail: <lsi, flubitz, glesner>@mes.tu-darmstadt.de, reis@inf.ufrgs.br

Abstract

This paper presents an approach for the integration of reconfigurable hardware and computer applications based on the concept of ubiquitous computing. The goal is to allow a network of reconfigurable hardware modules to be transparently accessible by client applications. The communication between them is done at the API level, and a Jini-based infrastructure is used to provide an interface for the client applications to find available reconfigurable hardware modules over the network. A DES-based cryptography system was implemented as a case study.

1 Introduction

The concept of ubiquity, initially introduced within the computer systems world by Weiser in the early 1990s [1], comprehends the ability to exist everywhere at the same time. Weiser's interpretation of this concept advocated that computer systems should disappear into the background, requiring minimum - if any - attention overhead from its users. In this paper, we explore such concept to create a higher level of abstraction for the usage of reconfigurable hardware platforms. Our goal is, quoting Weiser, to allow reconfigurable hardware "disappear in [a] way [we are] freed to use them without thinking and so to focus beyond them on new goals". We are aware that it is an ambitious statement - as it was Weiser's goal when stated a decade ago - so we focus first on a few application scenarios, described in Section 3. The underlying approach, however, is general enough to be extended into other application domains. In Section 4, some implementation issues are covered, followed by a case study where we use a cryptography system to validate our approach on two of the proposed application scenarios. The paper is closed with the envisioned extensions for this work and with our conclusions on the benefits of applying the concept of ubiquity in the reconfigurable hardware arena.

2 Ubiquitous reconfigurable computing

The possibilities granted by the integration of reconfigurable hardware modules to computer systems are well known [2]. Applications domains range from consumer electronic devices which can be upgraded after deployment, to emulation platforms supporting the design and verification of electronic systems. Such integration, however, requires specific expertise from the designers and the integration procedure itself can be very time-consuming.

Our work aims to reduce the integration overhead of reconfigurable hardware modules and computer systems. We propose to reduce such overhead by raising the level of abstraction of the integration architecture, allowing the communication to be done via message passing, as proposed in the object-oriented paradigm. By using this approach, each reconfigurable hardware module can be seen by the rest of the system as an object[1]. Thus, it should be reconfigured and used through method calls. This can make a significant difference for the system designer, because he/she can abstract the internal details of the reconfigurable module - a typical result of the encapsulation feature of object-oriented systems - and can design the whole system communication at the API level. In such approach, all the subsystems depending on the reconfigurable hardware module can call a configuration method to set up the desired functionality, and then call methods to pass the data to be processed and receive the results. Figure 1 depicts such possibility.

In order to cope with the demands of the current application scenarios - where the computation is performed by several interconnected appliances - we also have to support the integration of reconfigurable hardware

[1] an entity with a well defined boundary and identity that encapsulates state and behavior [3]

1530-1591/03 $17.00 © 2003 IEEE

modules into distributed computer systems. We can expect that each subsystem could be in a different location, connected to the others by a heterogeneous network. So, our approach should allow those subsystems to interact with any number of encapsulated reconfigurable hardware modules. The minimum infrastructure to do so comprehends distributed resource localization and remote method invocation. The first technique provides means for the distributed objects to locate other objects according to their needs, while the second one is responsible for the common dialect used by the objects to exchange messages once they have establish communication. Section 4 will cover those techniques in more detail.

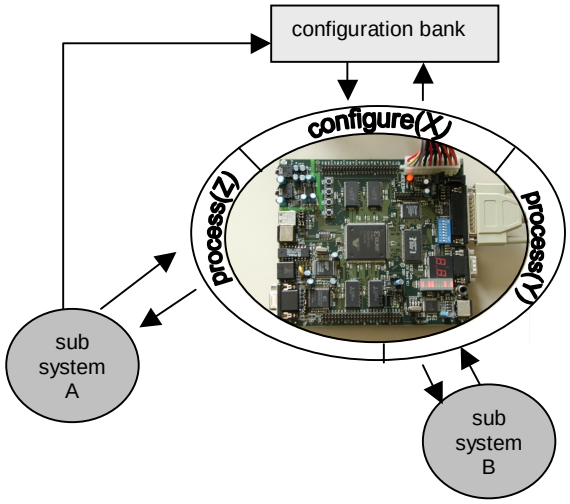

Figure 1. Reconfigurable hardware encapsulation

3 Application scenarios

Many of the applications of reconfigurable hardware can benefit from the proposed approach. For instance, devices which were already deployed - such as an ad-hoc network of sensors - could be located and upgraded by method calls if they have encapsulated reconfigurable hardware supported by an infrastructure for localization and remote method invocation. Another application scenario would be the use of reconfigurable hardware modules as accelerators for specific computational tasks. For instance, a mobile device which needs to decode a stream of data and does not have the computational power to do so could use the resource localization feature to search for a reconfigurable hardware module which is able to decode the data stream.

A third application could be found on the use of reconfigurable hardware as a prototyping platform. Let's imagine a system design specification done in the functional level. Once that specification fulfills the functional requirements, it should be submit to successive synthesis steps in order to be implemented as a physical entity. Reconfigurable platforms are being used as an intermediate stage within such process, allowing system designers to verify the correctness of their designs prior to the final implementation. Our approach could provide a simpler way to integrate the functional specification with the prototyping platform, in such a way that they can inter-operate. This would allow a mixture of simulation and emulation in the functional level, because one could synthesize and implement part of the functional specification in the reconfigurable hardware and still be able to perform the functional simulation, as the rest of the specification would communicate with the prototype in the same way it did before with the functional description.

Similarly to the prototyping platform scenario, a distributed IP core validation system could benefit from the proposed approach. In simulation systems such as those presented in [4,5], a designer access remotely an IP core so it can be simulated together with the rest of the design. Our approach could provide a layer between the IP core repository and the client, so the cores could be accessed seamlessly, without a previous connection to a predefined server. Another advantage would be the possibility of simulating an actual core implemented in a reconfigurable module, instead of the simulation models proposed in [4,5].

Finally, we can envision also the benefits of ubiquitous access to the application reported in [6], which uses FPGA boards as an educational resource for distance learning and on-the-job training.

4 Implementation issues

Several design decisions were taken in order to implement the proposed concepts. The first of them regarded the underlying network infrastructure over which the objects should communicate. As TCP/IP networks are nowadays the *de facto* standard for the intercommunication of computer systems, this was not much of a choice. By using TCP/IP as underlying infrastructure, the deployment of reconfigurable modules can be done over any Internet-like network.

The following decision regarded the implementation of the infrastructure allowing ubiquitous access to the reconfigurable hardware modules. We consider a module to be ubiquitously accessible if a client application is able to use its reconfigurable hardware services no matter where it is located or which kind of reconfigurable devices it has. It is also desirable that the infrastructure allows the reconfigurable hardware module to be included, moved or and excluded dynamically within the

distributed system, aiming to more flexibility and fault tolerance.

There are some middleware solutions which can fulfill those needs, such as CORBA [7], Jini [8] and UPnP [9]. All of them work over TCP/IP networks, and share the common architecture showed on Figure 2. In principle, any of them could be used to support our approach.

Our implementation uses Jini, mainly because of the freely available development tools and because most of the facilities for service lookup, discovery and join are already implemented and freely available. Jini also includes a programming model - built over the Java language framework - covering leases, events and transactions. The remote method invocation infrastructure also rely on Java language features, specifically on the JavaRMI package. Such programming model uses local proxies to reference remote objects, so in the application domain all method calls seem to be local, reducing the overhead usually associated to dealing with remote subsystems.

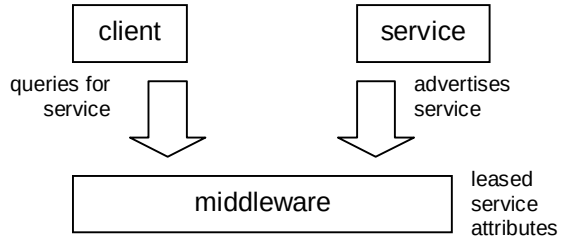

Figure 2. Middleware architecture

The procedure a client goes through in order to access the reconfigurable module is described in the UML sequence diagram in Figure 3. Initially, the client searches for a lookup service. If the network address of the lookup server is known, a direct connection is established. Otherwise, an UDP multicast request can be sent, which would be replied by the lookup servers reached by the request.

When the connection with the lookup server is established, the client should perform the service lookup. Such lookup is usually done based on a key which provides unique identification of the service. In our implementation, such key would identify the desired configuration for the reconfigurable module. By using Jini, we also have the possibility of using an object as a key, so a more sophisticated lookup can be done. It allows a scenario where the client is able to provide a specific configuration as a key. In such case, the lookup server can use the attributes of this configuration (target device, logical capacity, etc.) to find an available reconfigurable module which matches the request. In Figure 3, however, the general case is depicted, when the configurations are all pre-stored in a configuration bank (in our implementation, a special Jini service called JavaSpaces [10] is being used to store the configurations).

By relying in such approach, the client can access a reconfigurable hardware module in a completely transparent way. Its location - as well as the location of the lookup server - are dynamically obtained during runtime and are transparent for the client developer. The internal functionality of the reconfigurable hardware module is also hidden, since the client access it through a proxy object. The interface of such proxy object - the API calls it support - are the only information the client requires in development time.

In order to map the API calls into specific functions - which are usually particular for each type of reconfigurable hardware platform - we use a backend module in the host computer in which the reconfigurable hardware module is connected. Such backend is developed specifically for a single type of reconfigurable platform and handles the platform-dependent functionality - e.g. setting up for configuration, accessing memory modules, etc. Our implementation was done in a XESS XCV-800 Virtex Board, so we implemented a specific backend API for it, encapsulating on it the functionality to program the FPGA and access the SRAM memory banks.

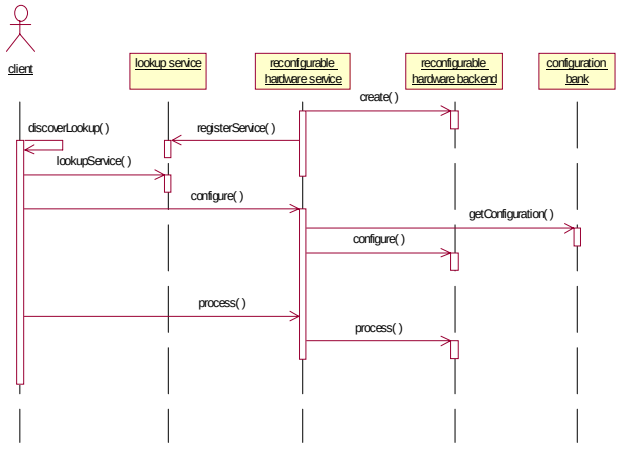

Figure 3. Accessing reconfigurable hardware

Another problem which must be solved in the backend module is the data abstraction support. While in the object-oriented domain all the computation is based in complex data types, when it comes to hardware such data types should be processed at the bit level. In order to make the bit level computation invisible from the object-oriented domain, we provide a data conversion interface API, which can be used with small updates in a wide

range of reconfigurable platforms. Such interface organize the data to be processed by the reconfigurable module in a bit array, including in such array the information about the data types. The array is uploaded to the reconfigurable module's memory, where the reconfigurable processor reads, it, processes it and write the results back so the backend can do the inverse operation and send the results back to the object domain

In order to facilitate the development of hardware configurations supporting our approach, we provide a HDL description of a module which is able to read the data array assembled by the backend module and reconstruct it as data types which are tractable within the HDL domain. The inverse path is of course implemented, so that the results of the data processing done by the reconfigurable processor can be written into the memory in such a way the backend module can rebuild into data types.

This type abstraction module should be built on top of the memory interface module (in our implementation we use the XESS memory interface developed by [11]). By using such module, the designers of the configurations for the reconfigurable processor can focus on its functionality, without particular care to the fact that the processor will be encapsulated and integrated into a distributed objects environment. Figure 4 depicts the layered architecture we provided to grant the complete separation between domains.

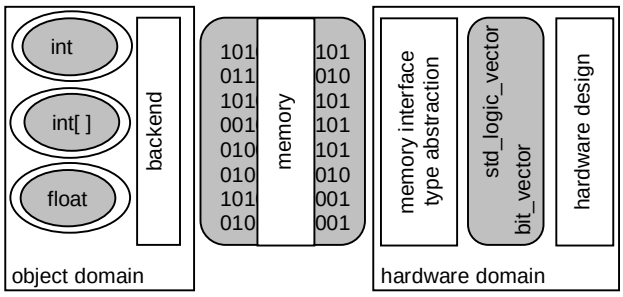

Figure 4. Abstraction layers to application development and hardware design

5 Case study: cryptography system

In order to validate the proposed approach, a cryptography system was implemented. It was designed over the proposed infrastructure, so the system could take advantage on reconfigurable hardware modules which were available in the network.

The first experiment covered the application scenario where a developer can incrementally prototype the target system described in the functional level. The chosen example is a messaging system that sends and receives encrypted messages using the DES encryption algorithm. We implemented the whole system functionality using Java language, so that the potential users can evaluate if it fulfill their functional needs. In the next step, we started the incremental prototyping of the system by implementing some system modules in reconfigurable hardware. So, the DES encrypt and decrypt modules had to be converted to HDL in order to be synthesized and implemented in our FPGA prototyping board. We used an HDL core for the DES algorithm [12] in this implementation, but when such possibility is not available the conversion can be done with design automation tools such as Forge [13] or even by hand if the design isn't too complex. We used the type abstraction and the memory access interfaces depicted in Figure 4, so the HDL core could be integrated easily.

After the configuration was generated, it was stored in the JavaSpaces repository. A Jini service federation was initialized, and a service for the FPGA board was registered on it.

In the application side, we replaced the software objects which were performing the DES encrypting and decrypting by proxy objects with the same external interface. The rest of the application objects were not changed, because the API they used to communicate with the DES objects was kept. The internal implementation of the DES objects was changed into proxies, relying on our implementation of a Jini client, so they contact the FPGA service every time a DES encryption or decryption was requested by the application. The FPGA service downloads the configuration from the JavaSpaces, program the FPGA, receives the data from the proxy objects, maps it into the FPGA memory, starts the FPGA execution, reads the processed results and returns them to the proxy object.

We successfully implemented such scenario, and the use of such prototyping strategy was found very convenient. It made possible the functional validation of the prototyped design block, as it was tested together with the rest of the functional description. The use of the proposed abstraction layers between the object domain and the hardware domain allowed a clear separation of concerns, making easier the development on each of the sides.

After the experimentation of the proposed approach as a support for incremental prototyping, we decided to test its suitability on the support of distributed processing. We envisioned a scenario where a particular computational task could be used remotely by a device with small computational power, like a mobile phone or PDA. In such case, the device would not be interested on simulating the communication between parts of a system, but actually request a particular data processing task

which would be too costly for it to implement alone. In such cases, the ubiquitous nature of our approach would be critical, because the small device could be mobile - perhaps the processing reconfigurable units too - and a greater variety of tasks to be performed could be available. We used the same DES implementation described in the previous scenario, but this time its implementation could be seen by the application system as an ubiquitous reconfigurable co-processor.

The implementation was successfully achieved, and the application used our infrastructure to request to the Jini service federation for a co-processor for computationally intensive tasks. As expected, the actual execution time of the DES algorithm in the FPGA implementation is much faster than the implementation is software, as shown in Table 1. The first two lines show the maximum data rates for the encryption and decryption in the application without the use of the reconfigurable co-processor. The third line shows the data rate obtained by the co-processor.

However, such gain could not be delivered to the client applications. The overhead of the FPGA configuration and of the data transfer from the host computer to the reconfigurable hardware module were too big, hindering the major goal of providing a significant acceleration compared to the computation that would be done locally by the client device. Such overhead can be reduced if there is no need for frequent reconfiguration of the modules - e.g. when there are many reconfigurable hardware modules in the Jini federation so the client can query for a module already configured with the needed functionality.

Processor	Clock (MHz)	Data rate (Mbits/s)
Sun JVM over Intel Pentium MMX	133	0.162
Sun JVM over AMD Athlon XP-1700	1467	1.295
Xilinx XCV-800	27.23	108.93

Table 1. DES implemetation comparison

The data transfer overhead can be also minimized, if the host computer and the reconfigurable hardware module are connected through a faster interface. In our prototype, the communication between the host computer and the FPGA board was done via a parallel interface, and the access to the SRAM banks was very slow. Much better results could be obtained by using a reconfigurable hardware module connected to the backend through a high speed bus, such as in the Pilchard platform [14].

The current implementation of the DES co-processor can be accessed over Internet. The lookup service listens to the port 4160 on the IP address 130.83.30.181. In the same machine, you can download using HTTP from the port 80 the source code to integrate a client into your application, as well as the API documentation for the DES proxy objects. If there are prototyping boards available, the lookup service will assign one to do DES encryption and decryption for your application.

6 Conclusions and future work

The integration of reconfigurable hardware into computer applications is not trivial, usually requiring from the designer a set of skills and expertise, including hardware design, communication between reconfigurable hardware and host computer and interface with software modules. In the case of a distributed application, more expertise is required in order to implement resource distribution and communication.

This paper presented an approach for the integration of reconfigurable hardware and computer applications based on the concept of ubiquitous computing. Based on the presented approach, a set of reconfigurable hardware modules can be plugged in a network and be transparently accessed by client applications. The client applications must not have any information about the network location or the internal implementation of the reconfigurable modules. The connection between client and reconfigurable hardware is based in a lookup mechanism. The reconfigurable hardware is encapsulated by a service interface, and all the communication with the client is done in the API level, through method calls.

We validated the proposed approach in two scenarios, one of them focusing on a system designer, which can use our approach as a support for incremental prototyping. For such scenario, we provide a library of code - both object-oriented software and HDL - to simplify the integration between the hardware domain and the distributed objects domain. In a second experiment, we used the infrastructure developed in the first scenario to implement a reconfigurable hardware co-processor service. Such application scenario was also successfully prototyped, but implementation-related limitations reduced the benefits expected from the proposed approach. So, further steps on this work should start by using faster interfaces for the communication between reconfigurable hardware and host computer, so that the performance gain shown in Table 1 could be actually delivered to the client application.

Further steps on the research can also be envisioned. One of the issues we intend to improve is on the supported API for the integration of reconfigurable hardware modules. Currently, there is a single API which should be implemented by all the modules. While this keeps it simple for the client application developer, it

reduces the potential of providing more specific services, assigning each of them to a different API call. We expect that the use of reflection mechanisms [15] could grant such possibility without sacrificing the simplicity for the developer. By using reflection, the simple interface can still be used, but mechanisms to issue requests during runtime for more detailed services would be available.

Another improvement that could increase the applicability of the proposed approach is the support of hardware originated method calls. In the current approach, all the data dependencies should be solved in advance by the backend, in order to start the computation in the reconfigurable hardware model. In the future, we intend to improve the communication between the reconfigurable module and the backend in such a way that the backend could call methods from other objects in the Jini federation under request of the reconfigurable hardware every time it faces an unresolved data dependency during its execution.

Acknowledgements

The authors wish to acknowledge and thank Dipl.-Ing. Alberto Garcia Ortiz and M.S.E.E. Abdulfattah Obeid for their valuable suggestions.

References

[1] M. Weiser. *The Computer for the 21st Century*. Scientific American, September 1991. Reprinted in IEEE Pervasive Computing, 1 (1), p. 18-25, 2002.

[2] K. Compton, S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. ACM Computing Surveys, 34 (2), pp. 171–210, 2002.

[3] J. Rumbaugh, I. Jacobson, G. Booch. The Unified Modeling Language Reference Manual. Reading: Addison-Wesley, 1998. 360 p.

[4] M. Dalpasso, A. Bogliolo and L. Benini. Specification and Validation of Distributed IP-based Designs with JavaCAD. Proceedings of Design Automation and Test in Europe, Munich, 1999. p. 684-688.

[5] M. J. Wirthlin and B. McMurtrey. IP Delivery for FPGAs Using Applets and JHDL. Proceedings of the 40th IEEE/ACM Design Automation Conference, New Orleans, 2002. p. 2-7.

[6] J. Becker, U. Mayer, M. Glesner, L. S. Indrusiak, R. Reis. Providing Flexible Internet-Infrastructure for FPGA-Based CAD Courses. Proceedings of the European Workshop on Microelectronics Education (EWME), Aix en Provence, France, 2000.

[7] Object Management Group. Common Object Request Broker Architecture (CORBA). v. 3.0. 2002. http://www.omg.org

[8] K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, J. Waldo. The Jini specification. Reading: Addison-Wesley, 1999.

[9] Microsoft Corporation. Universal Plug and Play Device Architecture. v. 1.0. 2000. http://www.upnp.org/download/UPnPDA10_20000613.htm

[10] E. Freeman, S. Hupfer, K. Arnold. JavaSpaces: principles, patterns, and pratice. Java Series. Reading: Addison Wesley, 1999.

[11] P. Sutton et. al. VHDL Interfaces and Example Designs for the XSV board. http://www.itee.uq.edu.au/~peters/xsvboard/

[12] The Free-IP Project. Free-DES. http://www.free-ip.com/DES

[13] D. Davis et. al. Forge-J: High Performance Hardware from Java. http://www.xilinx.com/forge/

[14] P.H.W. Leong et al. Pilchard - A Reconfigurable Computing Platform with Memory Slot Interface. Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2001.

[15] Sun Microsystems Inc. JavaTM Core Reflection - API and Specification, 1997. http://java.sun.com