A Dynamically Reconfigurable Virtual Instrument

Gerd Van den Branden [1], Geert Braeckman[1], Abdellah Touhafi [1], Erik Dirkx [2]

[1] Erasmushogeschool Brussel, Departement IWT, Nijverheidskaai 170, 1070 Brussel, Belgium
gerd.van.den.branden@ehb.be,
[2] Vrije Universiteit Brussel (VUB), Pleinlaan 1, 1000 Brussel, Belgium

Abstract -- **Hardware acceleration of software is a necessity in many computer engineering applications. The utilization of multiple processing elements adds speed, efficiency, fault tolerance and other desired features. Dynamic reconfiguration on a general purpose platform resulted in an economical implementation of a virtual instrument. A multi sensor computer system can operate in real-time on a scaleable hardware platform without expensive software re-writes or re-tuning. The main contribution of this paper is twofold: Firs we propose a generic and independent framework which provides a structured approach to the implementation problem. The framework will help the designer to organize and generate all necessary files (hardware as well as software) for the dynamically reconfigurable target system of choice. Second a detailed description is given of how an existing software application, designed to run on a Pentium IV desktop environment, is ported to an implementation that executes on a dynamically reconfigurable platform in order to achieve a considerable speedup in execution time.**

1. Introduction

Interest in reconfigurable devices today is encouraged by the fact that manufacturers promote some of their products as dynamically reconfigurable. Because dynamic reconfiguration is in fact a new design paradigm, there exists a great design productivity gap. In an effort to fill this gap we experimented with partial and dynamic reconfiguration in order to develop a generic and consistent design flow. In particular, we show by means of a case study how an existing software application can be transformed to an implementation that executes on a dynamically reconfigurable platform.

This paper gives an overview of how an application designed to run on a general purpose desktop PC environment can be ported to a dynamically reconfigurable platform. This results in a considerable decrease in execution time and a decrease in power consumption, without losing much flexibility and without introducing much design complexity. To achieve this goal we developed a framework which we will use to come to a final, generic, implementation.

This paper is organized as follows. In Section 2 we discuss some related work and explain why we believe that our approach is different. In Section 3 we explain what is meant by a virtual instrumentation machine and how it can be used to create a custom measurement infrastructure, which we call a virtual instrument. We present the original software-environment that is used to create a virtual instrument in a fast, accurate and cost effective way. Furthermore, the different elements that compose the virtual instrument are discussed and also how they relate. Finally, the benefits and drawbacks of a virtual instrumentation machine are discussed if the virtual instrument is running on a Pentium IV desktop environment. In Section 4 we will outline the details of porting the virtual instrumentation machine towards a platform where the virtual instrument is implemented according to the dynamic reconfiguration paradigm. Then in Section 5 the setup of the target platform is discussed. In this context we also point to difficulties we encountered regarding board design and chip layout, and propose possible solutions, if any. Furthermore we provide some performance measurement results and a short comparison with the original application. We conclude the paper in Section 6.

2. Related Work

A system that integrates reconfigurable hardware extends the classical computer system composed of one or more processing units, memory and a communication structure, with an amount of programmable hardware [8]. This programmable, or reconfigurable, hardware was traditionally used as a co-processing unit to speedup the overall execution of the application. Dynamic reconfiguration adds an extra benefit to this scheme in the form of virtual hardware [9], which makes it possible to virtually enlarge the available space of reconfigurable hardware by reusing the same space with different functionality over time.

A recent trend in the design community is to integrate the reconfigurable hardware as close as possible to a general purpose (embedded) processor. This results in a number of different architectures and supporting tools that are introduced during the past decade.

For example, Atmel offers the FPSLIC architecture which integrates a rather small amount of reconfigurable hardware coupled with an 8-bit AVR RISC processor. The processor interfaces with the reconfigurable hardware by a fixed 8-bit bus structure [10]. The FPSLIC architecture is supported by the System Designer development tools.

Stretch recently introduced the STRETCH S5000 series of software-configurable processors. The S5000 closely couples an amount of reconfigurable logic to a 32-bit XTensa RISC processor of Tensilica. The main objective is to exploit more parallelism by extending the instruction set of the processor, by introducing pragmas in the application code, and dynamically loading the desired circuit into the reconfigurable hardware [11].

Xilinx offers the Virtex-II/II-Pro/IV devices that support dynamic reconfiguration. Those devices can contain an embedded PowerPC405 core, or a softcore processor can be programmed in the reconfigurable hardware. The development tools of Xilinx provide the possibility to create partial bitfiles that can be dynamically loaded into the reconfigurable hardware [1].

Many more architectures exist. All adapt their own slightly different approach, and all require vendor-specific design rules and constraints. The framework that is presented in this paper proposes a design flow that is generally applicable, regardless of the device that is used, and regardless of the design language in which the specifications are given, whether this is Handel-C, SystemC, C++, UML, VHDL or any other alternative. On the contrary, in the first phases it requires the designer to do a careful analysis of the application and the target architecture such that this information serves as an input for the framework.

The framework can be used for system level design where different parts are to be executed in software and others in hardware. It has a preference for, but is not limited to, dataflow and DSP applications. In addition we show by means of a case study that the framework stays valid if the design makes use of the dynamic reconfiguration paradigm. This is because the framework obliges a generic, modular approach for building the system.

Furthermore, we developed supporting tools for the automatic creation of top-level VHDL code, and for the automatic code-generation for the reconfiguration engine. Automatic creation of top-level VHDL is possible because the framework restricts the designer to adapt a known modular approach to design the system. In addition the framework can be automated to a great extent by calling the individual implementation tools by means of batch files.

We want to emphasize that this paper is not about the automation of assigning software tasks to hardware. It is more related to hardware software co-design like the Ptolemy Project with great contributions from Lee et al [12], or the Hardware Software Codesign group of Georgia Institute of Technology [13]. However, our approach is different, because the framework we propose is not restricted to one formal language, but can be applied to an existing software implementation, no matter in how it is specified. This makes the framework more generally applicable at the cost of losing some flexibility because in one of the first phases of the framework it requires a careful analysis of the specifications. Rather than proposing a new scheduling strategy in this paper, we simply reuse the scheduling of the original application and try to execute more of the same computations in parallel.

In addition we emphasize that we want to prove that an application can sustain a considerable speed-up by executing tasks in hardware and that an additional benefit comes from the use of dynamic reconfiguration in the form of virtual hardware. According to these benefits it becomes possible to move the application to a real-time environment which is required and useful for a broad range of applications.

3. A Virtual Instrumentation Machine

A virtual instrumentation machine is a design environment that is used to create a custom measurement system. In other words, it offers the possibility to compose a particular virtual instrument, from which a software program is generated that emulates the dedicated measurement apparatus. In this context, an example virtual instrument is depicted in Figure 1. It shows a tool that captures audio data and displays its energy spectrum in the frequency domain.

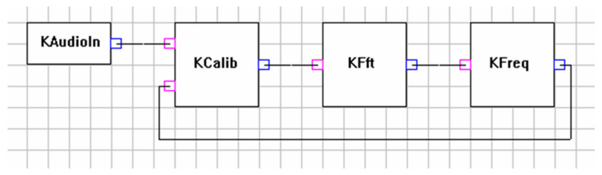
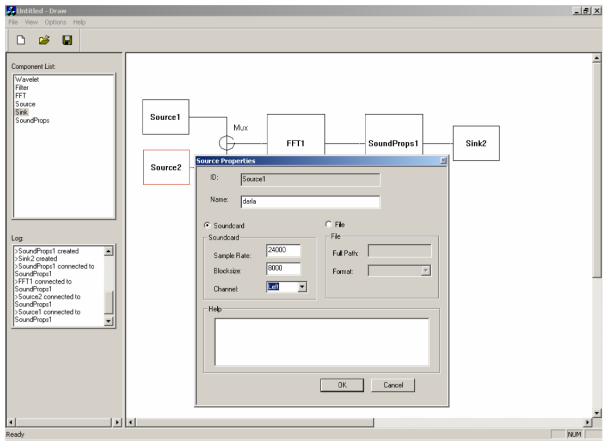


Fig. 1. A virtual instrument to display the frequency spectrum of audio data

In 2001 we developed an automation tool for the rapid implementation of virtual instruments, using Matlab. It is this software environment that we call the virtual instrumentation machine. It allows one to define a virtual instrument by means of interconnected multi-port operations (which we will call *objects* from here on), whereby the sources are data-acquisition elements and the sinks are storage elements. Furthermore it is possible to assign parameters to each object. Examples of parameter assignment are: depth of an FFT, the length of a cross-correlation window, pass and/or cut-off frequencies for digital filters and sampling rate. Figure 2 shows a screen shot of the environment.



Log:
```
>SoundProps1 created
>Sink2 created
>SoundProps1 connected to
SoundProps1
>FFT1 connected to
SoundProps1
>Source2 connected to
SoundProps1
>Source1 connected to
SoundProps1
```

ID: Source1
Name: darla
[✓ Soundcard] File
Soundcard
Sample Rate: 24000
Blocksize: 8000
Channel: Left
File
Full Path:
Format:

Fig. 2. GUI of the software-environment to define the virtual instrument

Once the user has defined the virtual instrument, an evaluation is executed on it. First a numbered graph representation is created using an ASAP (as soon as possible) numbering strategy. Next a consistency checking is executed on the instrument. This includes a type checking (time vs. frequency domain), port checking (e.g. an input can only connect to an output of a predecessor or a source) and a check to ensure that no infinite loops are created. When no errors are detected, a serialized file is generated that implements the virtual instrument. This representation of the virtual instrument is then passed to the simulator. These simulation results are used later on to verify the correct behavior of the virtual instrument.
Once the virtual instrument is created, one can choose to apply it on data that is available in a file, or in real-time on data captured by the soundcard. The results are saved to a file and can be visualized on various ways: a graphic, raw data, sound file, log file, etc…
For more documentation about the details of this software environment we refer to the publications and documentation that can be found on our website [2, 3].

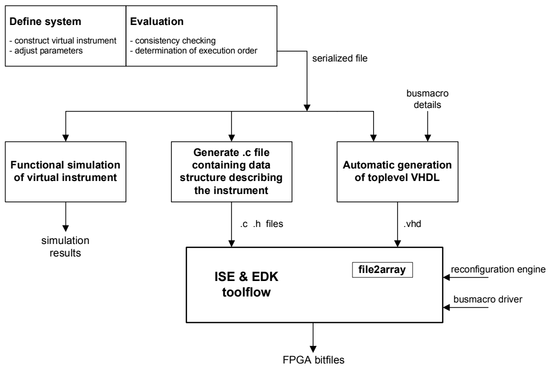
The purpose of this software implementation was to provide a flexible alternative for measuring and analyzing sound and vibrations, without the need of buying each time a specific and expensive measurement apparatus, and without loosing much accuracy compared with those state-of-the-art analyzers. We can state that this goal is achieved [4]. However, there are some serious shortcomings when we need to do real-time measurements if multiple channels need to be analyzed in parallel. For example a Pentium IV desktop environment (1.4 GHz is capable of analyzing one sound channel in real-time. However, in a multi-sensor environment the Pentium IV machine cannot meet the real-time deadlines. It is clear that the equipment misses the necessary computing power to provide correct results in time, caused by the Von Neumann bottleneck.

4. The Case Study

In an effort to overcome the Von Neumann bottleneck, we want the virtual instrument to run on a dynamically reconfigurable platform. The objective is to introduce more specialization, and thus performance, while keeping flexibility in the form of reconfigurability. Furthermore, we successfully limited the increase in complexity by composing the final system of standard IP cores and system buses in the fixed part of the system, and working with a generic interface for the objects implemented in reconfigurable hardware.
Migrating towards dynamic reconfiguration for this application is a natural step to take because the objects that compose the virtual instrument are well defined computations and therefore lend themselves well to execution in reconfigurable hardware. Partitioning thus is easy because each object corresponds one to one with a well defined computation. In addition, the scheduling problem can be recycled from the numbered graph representation. We reuse this schedule as the base for the reconfiguration engine.

4.1 Reusing the Front End

The entire front-end of the virtual instrumentation machine can be reused. The design environment to create and evaluate the virtual instrument is unaltered and therefore is identical to what was explained in Section 3. There can be rigorously taken advantage of what has been done in the past.
However, before writing the serialized file, some additional information is included compared with the original software based implementation. This extra information is related to the number of reconfigurable modules that will be used in the final system and information about which off-chip peripherals interface with the platform we use (e.g., if the system will use external DDR SDRAM memory). This extended serialized file will be used, similar to a compiler's intermediate representation, as the base for several upcoming design phases. These phases are discussed in more detail in the following subsections.
Figure 3 shows a schematic view of this case study.


Define system
- construct virtual instrument
- adjust parameters
Evaluation
- consistency checking
- determination of execution order
serialized file
busmacro details
Functional simulation of virtual instrument
Generate .c file containing data structure describing the instrument
Automatic generation of toplevel VHDL
simulation results
.c .h files
.vhd
ISE & EDK toolflow
file2array
reconfiguration engine
busmacro driver
FPGA bitfiles
Fig. 3. General flow for the virtual instrumentation machine

4.2 The Simulator

The simulation process is essential to ensure that the composed instrument behaves as expected. The simulator takes the serialized file which describes the virtual instrument and uses this information to build a data structure preserving the circuit graph. For simulation purposes, each object has been mapped to a software simulation with the expected behavior. Simulation results are written to files on a module basis, so after simulation there is one file for each instantiated object.
In a later phase, each object will eventually be mapped on a piece of reconfigurable area.

4.3 The Reconfiguration Engine

The Reconfiguration Engine is a software program running on a microprocessor on the target platform. It is designed to be generic and scalable, such that it can drive a number of modules varying from only one to as many as the FPGA can contain. In Section 5 we discuss this limit, and provide a general way to find it for an arbitrary system.
The circuit scheduling algorithm of the reconfiguration engine is based on the same As Soon As Possible (ASAP)

algorithm that was used in the evaluation phase of the virtual instrument.

When a reconfigurable area comes available, the engine reads and stores the data from the finished module and saves the state of the internal registers. It then loads a new configuration and restores the state of the internal registers according to the new configuration. The computation data is written to the module's memory, and finally the module is activated. The engine continuously polls the ModuleReady signal of each module to detect whether they have finished their task.

4.4 Hardware Generation

To create the hardware for the reconfigurable system, a similar generic approach is taken to the one introduced in the previous section. The system is composed of communicating modules with a unified interface. Two types of modules exist: fixed and reconfigurable. A fixed module does not change its implementation throughout the runtime of the application, but a reconfigurable module does. The fact that each reconfigurable module implements an identical interface with the fixed module also makes them generic. This enables the system to be scalable to a certain limit. For a quantification of this limit, we refer to Section 5, where we discuss the implementation results.

Each reconfigurable module computes its intended function before it alerts the fixed module that the computation is done. Remember that every reconfigurable module corresponds to an object that was instantiated in the composing phase of the virtual instrument (in what we call the front end of the virtual instrumentation machine). The fixed module will implement the reconfiguration engine, the global communication control between the implemented reconfigurable modules, the sources and the sinks, and the communication with “off chip” devices.

In this context it is possible that a part of the fixed module is available as hard-coded logic in the target device (e.g., an embedded processor system), but it is also possible that a part of the fixed module is implemented in the reconfigurable hardware but does not change throughout the runtime of the application.

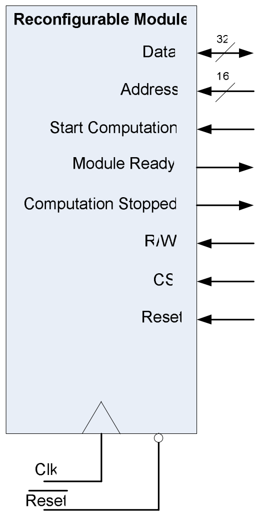


Fig. 4. Black box representation of a generic reconfigurable module

Figure 4 shows a template of a reconfigurable module. The reconfigurable modules are available as VHDL descriptions in a library. They are described according to the principles of genericity and modularity as explained before. As such, each reconfigurable module implements a module control block which serves as the link between the module’s generic interface and the module’s custom DSP functionality. Furthermore, each reconfigurable module implements a FIFO-in and a FIFO-out memory which is also controlled by the module control block. We refer to Figure 6 for a schematic view of the internals of a reconfigurable module.

4.5 Generating Top-level VHDL

Thanks to the information contained in the serialized file and the design decisions concerning modularity made early in the implementation phase, it becomes possible to automatically generate a VHDL description of the top-level file for the dynamically reconfigurable system. As already stated in section 4.4 it depends on the target device that is used how this VHDL file needs to be interpreted. However, because of the modular approach, this top-level description will always have the same structure. Only the implementation of the instantiated modules needs to be adapted between different applications, but this is a separate phase in the framework. For simplicity reasons it was assumed that the hardware descriptions for the reconfigurable modules are available in a library.

All reconfigurable modules have an identical entity (cfr. Fig. 4), and because the number of modules is included in the serialized file, these modules can be automatically instantiated as components. Furthermore, all information concerning the peripheral off- or on-chip devices is known up front and is categorized in the first phase of the framework (cfr. Fig. 5). This information enables the automatic generation of the entity for the fixed module. The top-level VHDL file of the system contains the fixed and all reconfigurable modules, the clock generation

components, if necessary the input – output connections of the system, and it describes how these components logically interconnect. The required information now is reduced to the knowledge about the interconnection of the different components.
Because the interconnection information is done according to a fixed scheme based on a predefined macro, this macro is predefined and can be instantiated automatically.

4.6 The Framework

At this point all elements are available to assemble the complete system and finalize the design phase. This is done according to the framework depicted in Fig. 5. This framework is developed and validated. It supports the creation of partially and dynamically reconfigurable systems on a consistent and efficient manner. We refer to [5] and to our website [2] for more detailed information about how to read and use the framework.

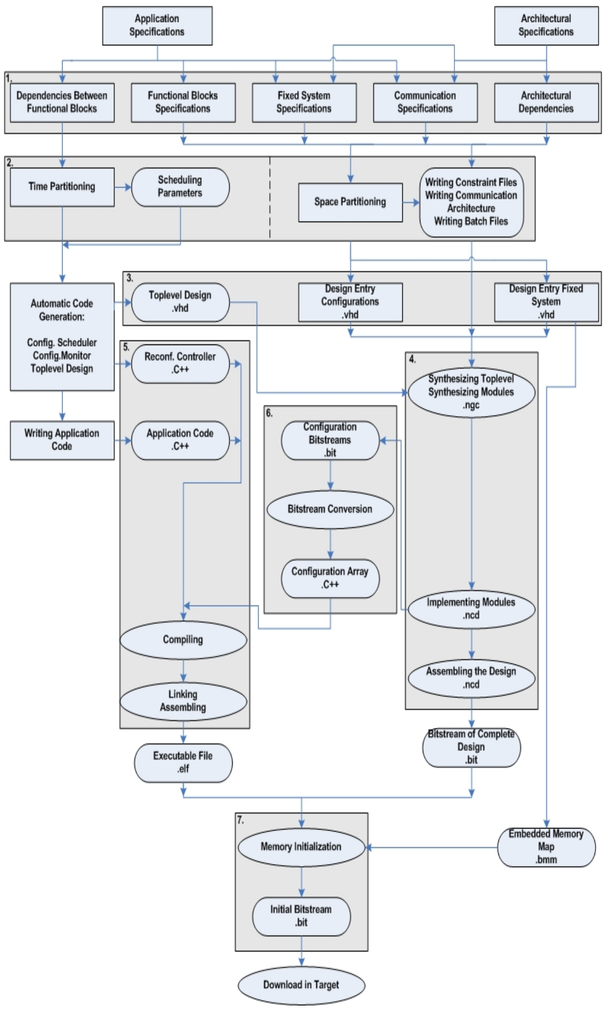

Application Specifications
Architectural Specifications
1.
Dependencies Between Functional Blocks
Functional Blocks Specifications
Fixed System Specifications
Communication Specifications
Architectural Dependencies
2.
Time Partitioning
Scheduling Parameters
Space Partitioning
Writing Constraint Files Writing Communication Architecture Writing Batch Files
3.
Automatic Code Generation: Config. Scheduler Config.Monitor Toplevel Design
Toplevel Design .vhd
Design Entry Configurations .vhd
Design Entry Fixed System .vhd
5.
Reconf. Controller .C++
Writing Application Code
Application Code .C++
4.
Synthesizing Toplevel Synthesizing Modules .ngc
6.
Configuration Bitstreams .bit
Bitstream Conversion
Configuration Array .C++
Implementing Modules .ncd
Assembling the Design .ncd
Compiling
Linking Assembling
Executable File .elf
Bitstream of Complete Design .bit
7.
Memory Initialization
Embedded Memory Map .bmm
Initial Bitstream .bit
Download in Target

Fig. 5. The implementation phase of the final system is done according to this framework

5. Results

The entire flow described in this paper can be applied to any partial and dynamically reconfigurable device, even if the reconfiguration process is steered from an off chip processor or if the embedded processor is coupled to the reconfigurable hardware by a particular interface. To test the framework a case study is performed where a virtual instrumentation machine is implemented on a development board that contains a Xilinx Virtex-II Pro FPGA [1] with an embedded PowerPC processor, 256 MB of external SDRAM attached to it and a RS232 connector to connect the device to a host PC for debugging purposes. The SDRAM is used to store the different partial bitfiles and the software code that is executed on the on-chip processor. The software code contains as most important functions the reconfiguration engine, drivers for the customized communication structure, and the drivers for the interface port to the configuration memory. The interface to the configuration memory of the device is required to read the state information, and write new configuration files into the configuration memory.
The reconfigurable modules are designed using Simulink/System Generator, in combination with hand coded VHDL. The fixed module is created using state-of-the-art CAD tools for embedded system design. The embedded system includes one embedded processor, 128kB of embedded RAM memory, a uart core, two 32-bit general purpose IO modules to connect to the communication structure, and a core to connect to the configuration memory, as the most important peripherals. All module files were synthesized, placed and routed with state-of-the-art CAD development tools that provide the ability to generate partial as well as the initial bitfiles. The partial bitfiles were transformed to a standard C array with a tool (file2array.exe) that we developed, and are included in the rest of the software code (drivers and reconfiguration engine).
Experiments are done with dynamically reconfigurable systems that implement one fixed and one reconfigurable module, and with systems with one fixed and two reconfigurable modules. The modules were chosen to be 12 slice columns wide and the full height of the device. The latter is a restriction on Virtex-II Pro devices and therefore this restriction was included in the architectural specifications that serve as an input to the framework. This resulted in uncompressed partial bitfiles of about 100 kB. The interface with the configuration memory is clocked at a rate of 50 MHz, so reconfiguring a module takes about 2ms (1 byte of configuration data every clock cycle). If we process data that comes at a rate of 48 kHz, a buffer with a depth of about 150 32-bit words is required to ensure safe operation for real-time data processing. However, for simplicity reasons buffers of 512 words are implemented, which correspond to one complete embedded RAM memory block.
Following reconfigurable modules where created: a lowpass FIR filter, a Hanning windowing function (512 samples wide) and a spectral sum generator are created. The modules communicate only with the fixed module to exchange data and results, following a simple four-phase handshake protocol. This protocol was part of the application specifications that serve as input to the framework. The design of the communication structure can

be automated as outlined in [6], and as such it can be included into the virtual instrumentation machine environment (however, we did not implement this in our case study, but hand coded it). This resulted in a pre-routed hard macro, consisting of the lines similar as the ones shown in Figure 4. To quantify the maximum number of modules, there can be remarked that if the maximum of communication lines is limited to 64 (two 32-bit gpio modules), only six reconfigurable modules can be instantiated in the final system. The reason is that in this implementation a separate CS and comp_stop signal line for each individual module needs to be available, while the other lines are shared among all modules.

Our experience is to minimize the number of reconfigurable modules, and to maximize the area of the fixed module, if applying the framework on a Virtex-II Pro component. The latter is useful because there will be more BRAM memory available in the fixed module, which serves as a sort of cache memory to pre-load the partial configuration streams. The former is useful to master overall complexity and design effort. Figure 6 gives a view of an implementation that is build according to the guidelines described in this section.

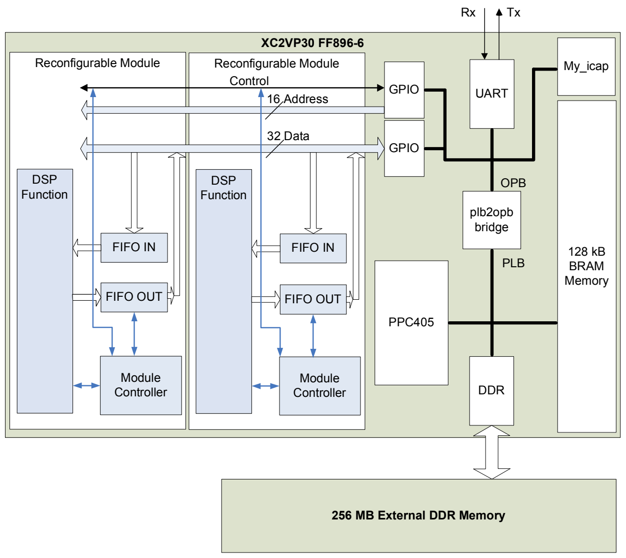


Fig. 6. A setup of a dynamic reconifgurable system

A very important remark in porting the application to a dynamically reconfigurable platform is that the designer is subject to the technological constraints related to the equipment one uses. In our case, such a constraint was the design of the ML310 board and the immaturity of the design tools for partial and dynamic reconfiguration. The design of the board was unsatisfying because the I/O pins to the external DDR memory were located on the far left side of the FPGA. This constrained the location of the fixed module to be on the left side. But, the ICAP primitive is located on the far right side of the FPGA, which causes the fixed module to reside on this side. This contradiction causes the 256 MB of external memory to be unusable in the system presented in Fig. 6. In a search for another board, our experience was that this was the case for all investigated boards. As a consequence the only solution is to invest in extra engineering effort in order to solve this issue. Off course, the concept stays valid, and when starting from scratch the designer simply needs to be aware of it.

The immaturity of the tools is illustrated by fact that, even if it was not the intention to use dynamic reconfiguration, but the system was created according to the Xilinx Modular Flow [7], the external DDR memory even then was not accessible. Until the moment of this writing, we do not have any idea what problem caused this behavior. However, in a next release this error could be solved. This resulted in the fact that the calculating phase of the system was tested separately from the dynamic reconfiguration phase. Because of the unavailability of the external storage capacity, we were limited to creating dynamically reconfigurable systems with one reconfigurable module, while the partial bitstreams and a limited part of the source code (a stripped version of the reconfiguration engine, and the drivers for the communication structure and the ICAP module) are both contained in the internal BRAM memory of the fixed system. Tests with multiple reconfigurable modules, such as depicted in Figure 6, are executed, but without the use of dynamic reconfiguration. Also the source code for the entire virtual instrument (reconfiguration engine, drivers, application specific code, and partial bitfiles) was too big to reside completely in the BRAM memory of the fixed system. As a consequence, the entire reconfiguration engine is only simulated. However, we emphasize that we succeeded in dynamically reconfiguring the platform, initializing the module, starting its computation, collecting the results, reading out and saving its state.

6. Conclusion

In this paper we've discussed the porting of an existing complex application on a dynamically reconfigurable platform. We have presented a clean overview of the process on how one can port an existing application to a dynamically reconfigurable platform in order to achieve a performance boost. There has been showed that a lot of concepts and knowledge of the existing application can be reused when taking this step, especially when the original application was already designed in a generic way. Furthermore, a validated framework to come to the dynamically reconfigurable system has been proposed. Accordingly, some useful design guidelines which were derived from our experience in this matter were shared in this paper.

We conclude this paper by stressing the importance of the completeness and accuracy of the architectural specifications of the dynamically reconfigurable component, as well as the target board. The relevance is illustrated clearly in the fact that those specifications serve as an input for the framework proposed in Figure 5. We hope that this importance is also made clear by our experiences outlined in Section 4. Great up-front care is required in order to avoid obstructions at the final assembling phase of the design, and to master the engineering cost of the application. This paper contributes in this matter by providing a validated framework to fixate the flow, by highlighting problems that can occur, and by relating design guidelines to them. However such problems are highly technologically determined, and therefore are not considered as conceptually relevant, it is always true that the architectural specifications of the target equipment need to be analyzed accurately at the very beginning of the design.

References

[1] Xilinx User Guide 012, Virtex-II PRO Platform FPGA Handbook, San José (2002)

[2] http://elektronica.ehb.be/reco/

[3] http://akoestiek.ehb.be/

[4] Touhafi, A., Braeckman, G., Raadschelders, M.: *Implementation of an Integrated Virtual Instruments Environment for Acoustic Measurements*, International Conference on Noise and Vibration Measurement (ISMA), ISBN 90-73802-79-2, Leuven (2002)

[5] Van den Branden, G., Touhafi, A., Dirkx, E.: *A Design Methodology to Generate Dynamically Self-Reconfigurable SoCs for Virtex-II Pro FPGAs*. FPT05, NUS Singapore (2005)

[6] Thorvinger, J., Lenart, T.: *Dynamic Partial Reconfiguration of an FPGA for Computational Hardware Support*, Masters Thesis, Lund Institute of Technology (2004)

[7] Xilinx Application Note 290: *Two Flows for Partial Reconfiguration: Module Based or Difference Based*, http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf, San José (2004)

[8] Touhafi, A.: *Scalable Runtime Reconfigurable Computing Systems –Design, implementation and use for simulation purposes*, PhD Thesis, VUB (2001)

[9] Hideharu, A., et al: *Techniques for virtual hardware on a dynamic reconfigurable processor -An approach to tough cases-*, FPL2004, Antwerp (2004)

[10] http://www.atmel.com/products/FPSLIC/

[11] http://www.stretchinc.com/technology/

[12] Buck, J. T., et al.: *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*, Int. Journal of Computer Simulation vol. 4, pp 155-182, (1994)

[13] Mooney, V.: *Hardware/Software Codesign of Run-Time Systems*, Technical Report CSL-TR-98-762, (1998)