Partial Dynamic Reconfiguration in a Multi-FPGA Clustered Architecture Based on Linux

Vincenzo Rana[1], Marco Santambrogio[1], Donatella Sciuto[1],
Boris Kettelhoit[2], Markus Koester[2], Mario Porrmann[2], and Ulrich Rückert[2]

[1]Politecnico di Milano
Dipartimento di Elettronica e Informazione
Via Ponzio 34/5, 20133 Milano, Italy
vincenzo.rana@microlab-mi.net
{santambr,sciuto}@elet.polimi.it

[2]University of Paderborn
Heinz Nixdorf Institute
Fuerstenallee 11, 33102 Paderborn, Germany
{kettelhoit,koester}@hni.upb.de
{porrmann,rueckert}@hni.upb.de

Abstract

Dynamically reconfigurable hardware allows for implementing systems that can be adapted at run-time according to the needs of the user. This paper presents an architecture that is composed of multiple FPGAs that are connected to an embedded processor. Thus, the architecture is referred to as a Multi-FPGA Clustered Architecture (MFCA). All FPGAs can be partially and dynamically reconfigured to integrate user-defined IP-Cores into the system at run-time. For the resource management and communication management we have implemented a Linux Operating System on the embedded processor that can be used to control the reconfiguration of the FPGAs by means of simple function calls. Furthermore, the Linux OS completely hides the physical infrastructure of the MFCA from user applications, offering a consistent interface to utilize partial reconfiguration. [1]

1. Introduction

Nowadays, most commonly used reconfigurable devices are *Field Programmable Gate Arrays* (FPGAs), employed both as components of more complex systems (playing the role of co-processors), and as System-on-Programmable-Chips (SoPCs), integrating all system components. FPGAs consist of arrays of configurable logic blocks (CLBs). Each CLB is connected to local and global routing resources, which are used to exchange data between the CLBs and with external devices. Classically, FPGAs are used to implement static circuits that are not changed at runtime. Their ability to be reconfigured is commonly used to upgrade the functionality of the system by changing the configuration data, e.g., due to changed specifications. Some FPGAs offer even more flexibility by allowing to reconfigure some CLBs independently while all other CLBs continue processing. This is called partial reconfiguration and enables the implementation of dynamically reconfigurable system architectures.

Due to these capabilities, FPGAs can be used to create flexible hardware/software platforms that can be adapted after fabrication, allowing the development of complex SoPCs. Modern FPGAs can also contain a general-purpose processor, which can be both a physical CPU embedded in the FPGA fabric, or a soft core, mapped to a part of the FPGA. The CPUs can be used to implement system functionalities in software (SW), e.g., for controlling the hardware (HW) components. The SW part of a reconfigurable system can be either a standalone code, dealing directly with HW at a low level, or a complete operating system, including multiprocessing and resource scheduling. A standalone code is usually an application which uses SW libraries exporting functions to interface with HW components. This approach can be reasonable for small systems, involving few components and configurations. As soon as the complexity of the system increases, it becomes more difficult to develop a complete application dealing with all those aspects. The use of an operating system

[1]This work was partly supported by the Collaborative Research Center 614 – Self-Optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

allows more flexibility, since it is possible to implement the SW part as one or multiple *userspace* processes, including inter-process communication and scheduling techniques. HW is managed by the OS, allowing the userspace processes to access system peripherals at a higher level of abstraction. However, the operating system needs to be enhanced to support dynamic exchange of reconfigurable hardware components.

The proposed work aims at introducing a complete methodology that allows a simple implementation of an FPGA system specification, exploiting an operating system that is extended to support partial and dynamic reconfiguration. In order to meet the software requirements of complex systems, the solution is based on a standard Linux OS which simplifies the handling of reconfiguration through the use of */dev* and */proc* device files. This allows software processes to exploit a rich set of features by simply using the developed Linux modules.

The Multi-FPGA Clustered Architecture that is presented in this paper also allows to adapt the amount of available reconfigurable resources to the needs of the designer since it allows an easy integration of additional FPGAs into the system. Our scalable rapid prototyping platform RAPTOR2000 (www.raptor2000.de), which allows the integration of several FPGAs, is used to implement the MFCA. Furthermore, the MFCA can be considered as a basic component of a reconfigurable supercomputer consisting of a distributed network of MFCAs.

This paper is organized as follows: Section 2 gives an overview of existing operating system support for dynamically reconfigurable hardware while Section 3 presents the proposed Multi-FPGA Clustered Architecture. In this Section both software and hardware architectures of the proposed solution are described as well as a prototype implementation of the approach. In Section 4, an analysis of the hardware requirements and performance is given. Finally, Section 5 presents the conclusions of this paper.

2 Support for dynamic reconfiguration in operating systems

The need for a *run-time infrastructure* in operating systems to manage and exploit reconfigurable logic is discussed in [9]. The paper proposes an approach for the design of an operating system for reconfigurable systems, called OS4RS. This OS must provide a set of services for reconfigurable devices similar to what a traditional OS does for multiprocessor systems, e.g., multitasking, concurrency, and inter-task communication. The OS4RS, which runs on a standard general purpose processor, has not been implemented from scratch, but it is constructed based on the RTAI system, [5]. The main reason for this choice are source code availability, large number of supported devices and modular architecture.

The reconfiguration capabilities of FPGAs can be used not only to map computational tasks in HW, but also to dynamically swap peripherals in form of reconfigurable hardware components. [1] presents a Linux-based platform with support for dynamic reconfiguration based on a modular architecture for reconfigurable SoCs, called *Egret*. [11] presents different examples to manage the reconfiguration using the ICAP (Internal Configuration Access Port) interface via standard UNIX commands. The Linux device interface, which presents devices as files located under the */dev* directory, makes it possible to send bitstreams to the driver using commands such as *cat* and redirecting the output to the device. Furthermore, it is possible to modify the bitstream on-the-fly by pipes between commands. Another possible application is a remote configuration download from a bitstream server.

An operating system designed to run on a single FPGA architecture able to support dynamic reconfiguration is proposed in the BORPH project, [10]. This solution is an extended Linux OS kernel for reconfigurable computing that handles FPGA resources as if they were CPUs.

As indicated by the examples given above, a lot of work has been done to add reconfiguration support to standard operating systems, both in the field of driver support for dynamic reconfiguration HW and in the development of drivers for reconfigurable IP-Cores. A complete approach that allows exploitation of the dynamically configured devices from the operating system is presented in [4]. This work defines the basis of the MFCA approach proposed in this paper, as shown in Section 3.

3 Reconfigurable multi-FPGA cluster

Our approach is based on single-FPGA solutions, as proposed in [3] and [6], which implement a system on a single FPGA that is able to perform partial self-reconfiguration. In a multi-FPGA reconfigurable scenario these concepts can be extended to satisfy the needs of the proposed multi-FPGA system. Therefore, a hardware abstraction layer has to be implemented that hides all implementation details to the application. This includes, e.g., the distribution of the resources among several FPGAs and the communication infrastructure.

3.1 The proposed solution

Dynamic partial reconfiguration of the FPGA can be performed by a dedicated HW embedded in the FPGA or by external entities, using a configuration interface (such as JTAG). In both cases a device driver is needed to allow interfacing between the operating system and the reconfiguration controller, in order to allow *userspace* processes to request reconfiguration.

The standard way the Linux kernel provides such access is through device nodes, which appear in the filesystem as normal files that can be opened, read, and written. The reconfiguration manager driver follows the same method, providing the necessary system calls and registering a specific device node. The driver completely hides the characteristics of the underlying HW offering transparency at application level from the actually used reconfiguration mechanism. The reconfiguration controller driver, shown in Figure 1, follows the kernel modules philosophy to implement mechanisms and not policies. This means that the driver just provides the system calls to make the device usable from *userspace*, and does not deal with process privileges in resource access. This kind of policy is in fact typically managed at a higher level in the kernel hierarchy, for example using standard UNIX permissions for the device node.

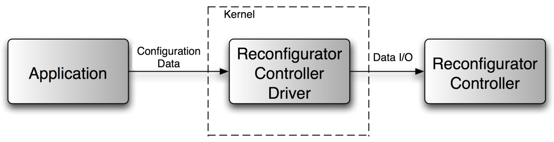

Figure 1. The Reconfiguration Controller working model

Partial reconfiguration data is usually provided by the HW platform development tools in the form of a bitstream file. It is called *partial bitstream*, since it only contains data for the FPGA areas that are affected by the partial reconfiguration. The bitstream format can be processed directly by the reconfiguration interface of the FPGA. Partial bitstreams must currently be created offline, since any modification to the HW requires a new synthesis, which cannot be executed at run-time. Exceptions are basic parameter variations in the partial bitstreams. In [11] the authors present a methodology that uses bitstream manipulations at run-time to modify parameters of the component that is loaded. Furthermore, bitstream manipulation can be utilized to relocate modules, as described in the next Subsection.

The main advantage of using the proposed device driver is that the application can simply access the controller to perform partial reconfiguration, i.e., a *userspace* process only needs to open the device for writing the partial bitstream. This method makes it possible to implement new applications managing the reconfiguration process and adjust existing ones with very little modifications.

3.2 Multi-FPGA Clustered Architecture

The proposed solution can also be applied to a multi-FPGA scenario where the reconfigurable resources are distributed on several interconnected FPGAs. The main challenge in such a scenario is to hide system characteristics and additional efforts (e.g., the setup of the communication) from the user application without any performance losses.

Dynamically reconfigurable systems usually consist of a static and a dynamic part. The static part includes all system components that do not change at run-time while the dynamic part comprises the reconfigurable resources for dynamic components. The description of the functionality of the dynamic components is called an Intellectual Property Core (IP-Core). This can be a behavioral description (e.g., in a hardware description language, HDL) as well as a structural description (HDL, netlist). The implementation of an IP-Core for a given system architecture is called a dynamic module. Dynamic Modules can be loaded into the system and can then be used as dynamic components. In general, it is possible to have multiple instances of one dynamic module loaded into the system at a time, each of which being a separate dynamic system component. It is also possible to have several dynamic modules implemented from one IP-core specification, differing, e.g., in shape or performance.

For an efficient use of the dynamic part the system architecture as well as the dynamic modules have to be designed with respect to the reconfigurable system approach. In this context the main issues are the placement approach, which is used to assign a dynamic module to the reconfigurable resources at run-time, and the communication infrastructure, which is used to interconnect the dynamic and static components. Figure 2 shows two different placement approaches and according communication infrastructures. Both are one-dimensional placement approaches, i.e. the dynamic modules can only be placed along a vertical line. Figure 2 b) shows a *Fixed Tiles* approach, where the reconfigurable resources are divided into disjunct areas (tiles) of predefined size. The size of the dynamic modules is limited to the size of the tiles whereas small modules lead to a remarkable waste of resources. The communication infrastructure is connecting the tiles and thus the dynamic components at predefined positions. Figure 2 a) shows a *Free 1D* approach. Here, all modules have a fixed height while their width is adapted to their actual size. This leads to a much more efficient use of the reconfigurable resources since small modules only occupy few resources. The size of big modules is restricted only by the size of the whole dynamic part of the system, rather than by the size of a tile. This approach requires a more sophisticated communication infrastructure with the ability to be accessed at many positions in order not to restrict the possible module locations. Besides the above mentioned placement approaches there are other approaches like the 2D-placement approach as discussed [6]. However, an implementation of the 2D-placement approach cannot be efficiently realized with Virtex-II FPGAs, since the device only supports a column-wise partial reconfiguration.

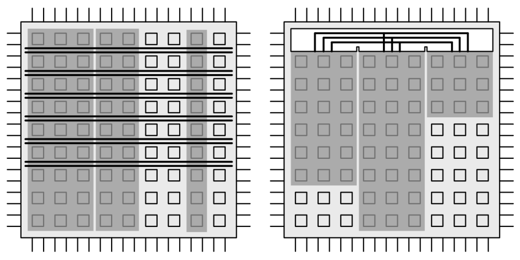

Figure 2. Two placement approaches

All placement approaches have in common that they require a run-time environment that is able to place and remove dynamic components on demand. On the one hand all reconfiguration issues have to be considered, e.g., searching for a free position to place the instance of a dynamic module. On the other hand the communication between the dynamic components and the application has to be established at run-time. In the following we will introduce an example implementation of such a run-time environment on our dynamically reconfigurable prototyping platform RAPTOR2000.

3.3 The RAPTOR2000 system

Establishing a hardware/software implementation for dynamic reconfiguration of multiple FPGAs requires a platform that offers high bandwidth and low latency communication between the FPGAs a well as low reconfiguration times. The RAPTOR2000 system (www.raptor2000.de) has been developed at the University of Paderborn exactly for this application scenario. The system consists of a motherboard and up to six application specific extension modules (ASMs). Basically, the motherboard provides the communication infrastructure between the modules and links the RAPTOR2000 system via the PCI bus to a host computer.

For the implementation of partially reconfigurable FPGA designs, various FPGA ASMs have been developed, integrating Xilinx Spartan FPGAs, Virtex(-E) FPGAs, Virtex-II FPGAs (up to XC2V8000), and Virtex-II Pro FPGAs (XC2VP20), respectively. Virtex-4 and Virtex-5 ASMs are currently under development. Additionally, up to 512 MByte SDRAM, SRAM and debugging interfaces are available on these ASMs. Various additional ASMs have been developed, e.g., providing network interfaces (Ethernet, USB, FireWire, etc.) and analog and digital I/Os. A graphical user interface can be used to configure RAPTOR2000 and all modules attached to it from a host PC. By means of this GUI, the board can be configured (e.g., clock frequency of the modules and address ranges), designs can be downloaded to the FPGAs on the ASMs, and data can be read from or written to the ASMs (i.e., to the FPGAs, and to the memory of the modules). In addition to the GUI a software library has been developed, which enables access to RAPTOR2000 from C programs on the host computer (or remotely) under Windows and Linux.

Various facilities for an efficient communication between the ASMs have been integrated into RAPTOR2000. Every ASM is connected to a Local Bus for internal communication with other ASMs and for external communication with the host processor or with other PCI bus devices. An additional Broadcast Bus can be used for simultaneous communication between the ASMs. Additionally, a dual port SRAM can be accessed by all ASMs via the Broadcast Bus (e.g., utilized as a buffer for fast direct memory accesses to the main memory of the host system). Direct communication between adjacent ASMs is realized by 128 signals that can be variably used, depending on the actual implementation. Due to the used high-speed connectors a data transfer rate of up to 10 GBit/s can be achieved between neighboring ASMs.

Especially when dealing with dynamic reconfiguration, a crucial aspect concerning FPGA designs is the configuration of the devices. For an efficient utilization of dynamic reconfiguration it is essential to minimize the reconfiguration time. Therefore, the reconfiguration interfaces have been implemented in hardware on two CPLDs on the RAPTOR2000 motherboard. Reconfiguration of an ASM can be started by the host computer, by another PCI bus device or by another ASM. Thus, it is possible that an FPGA

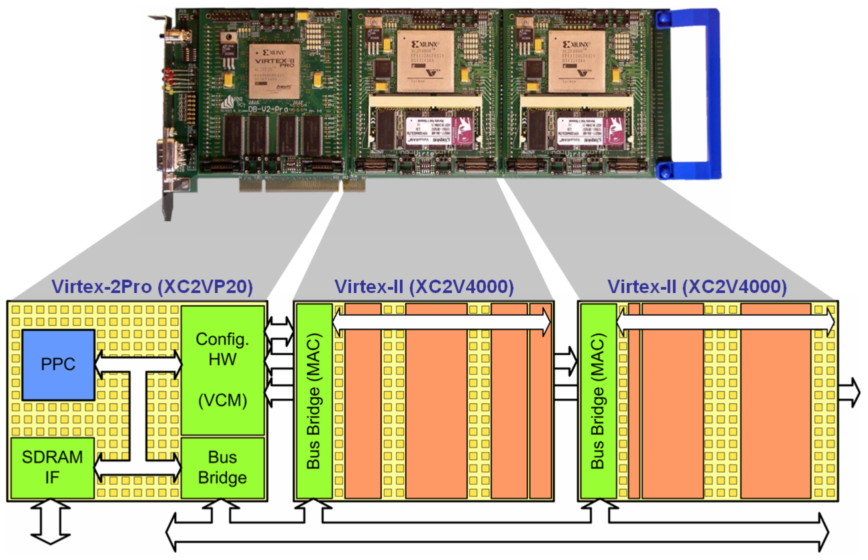

Figure 3. Multi-FPGA environment on RAPTOR2000

autonomously reconfigures itself by configuration data that is located anywhere in the system, thus enabling self-reconfiguration. Due to the hardware implementation of the reconfiguration interfaces on RAPTOR2000, Xilinx FPGAs can be reconfigured with the maximum configuration speed the devices permit, supporting complete reconfiguration as well as partial reconfiguration.

3.4 Reconfigurable hardware architecture

For a prototypical implementation of the proposed MFCA reconfigurable system, the architecture shown in Figure 3 has been developed. It consists of a Xilinx Virtex-2Pro FPGA and two Xilinx Virtex-II FPGAs. The Virtex-2Pro FPGA, which is used to run our SW solution (the OS), is constituted by an embedded PowerPC 405 and all the static hardware components, such as a memory controller, general purpose inputs/outputs, and the reconfiguration manager (*VCM, Virtex Configuration Manager*), which uses the SelectMap interface of the FPGA. The Virtex-II FPGAs represent the reconfigurable resources used to dynamically load hardware modules into the system. These resources are used according to a 1D-placement with a granularity of four CLB (configurable logic block) columns. This means that the dynamic modules always use the full height of the FPGA, while their width is a multiple of four CLB columns [6].

Each Virtex-II FPGA includes a Wishbone Bus the hardware modules are dynamically connected to. The bus-bridges that are used to connect the modules to the processor system include the Medium Access Control (MAC) for the communication with the modules. These MACs differ considerably from those used in standard on-chip bus systems, since they have to deal with a changing number of communication participants. Thus, they provide the ability to allocate address space for each loaded module at run-time. This allows for a very flexible use of the available bandwidth as well as for multiple instantiations of modules (e.g., two identical ALU-modules loaded for different tasks).

The reconfiguration manager is the hardware interface to the configuration ports of the FPGAs. A special feature of this component is its direct memory access (DMA) to the local SDRAM memory. This enables dynamic reconfiguration at the maximum configuration speed of the FPGA devices. Currently, we are able to download bitstreams from a given position within the memory to any selected FPGA within the RAPTOR2000 system at 50 MB/s (see Chapter 4).

Our implementation of the reconfiguration manager offers additional features that enable a very efficient and versatile realization of dynamically reconfigurable systems. It comprises methods for the relocation of dynamic modules by bitstream manipulation and enables context saving and restoring. The placement of dynamic modules is an online problem, i.e., the final position of a module is not known before run-time and especially not during the implementation process (synthesis, place and route). Typically, this would require IP-Core implementations for any possible position in the FPGA. For the large number of possible positions in our *Free 1D* approach, this leads to a high implementation time and effort as well as high memory requirements. To overcome these drawbacks, we have implemented the REPLICA2Pro (Relocation per online Configuration Alteration in Virtex-2/-Pro) filter, which is capable of performing task relocations by manipulating the bitstream of a dynamic module during the regular allocation process without any extra time overhead [7]. REPLICA2Pro is implemented in hardware as an integral part of our reconfiguration manager.

In the proposed *Free 1D* approach dynamically loading and erasing modules will inevitably result in a fragmentation of the available resources, i.e., the contiguous regions of unused FPGA resources are split into small fragments over time. It has been shown that the implementation of appropriate defragmentation algorithms is a promising approach to increase the utilization of the FPGA [2, 8]. In order to enable defragmentation an enhanced mechanism for task relocation has been integrated into the reconfiguration manager, including methods for saving and restoring the state information of the relocated task [8].

3.5 Software architecture

The software solution proposed in [4] has been adapted to exploit the features of the RAPTOR2000 system, introducing the *VCM* kernel module and the *MAC* kernel module, as shown in Figure 4.

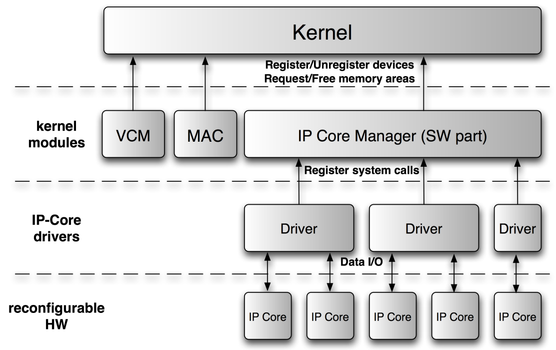

Figure 4. Device driver hierarchy

The *VCM* hardware component is used as configuration controller instead of the internal configuration access port (ICAP). Consequently, the *ICAP* device driver has been replaced by the *VCM* device driver in our new software architecture. The DMA properties of the *VCM* drastically reduce the involvement of the processor (and of the OS) in a configuration process compared to the programmed I/O that is used for the ICAP interface. In fact, a reconfiguration can be initiated by making three IOCTL calls only, which set the *VCM* device registers:

- the first IOCTL call sets the base address register of the *VCM* with the base address of the bitstream in the memory;
- the second call sets the size register of the *VCM* with the numbers of bytes of the bitstream;
- the last IOCTL call sets the control register of the *VCM*. It defines whether a partial or complete reconfiguration shall be performed and selects the target FPGA.

In addition, the *MAC* device driver has been added to the software architecture to allow a dynamic address allocation of the module on the Wishbone Buses. The address space allocation for each module is performed directly after a module is loaded. Therefore, the *MAC* device driver accesses the hardware MACs that are located within the Wishbone bridges (see Figure 3). An address space allocation consists of three simple IOCTL calls:

- the first IOCTL writes the base address of the newly loaded module to the base address register of the selected *MAC*;
- the second call writes the address range of the new module to the range register of the *MAC*;
- finally, the last IOCTL call assigns the newly loaded module to a physical communication port by transmitting the position of the module to the hardware *MAC*.

With these kernel modules it is possible to integrate IP-Cores into the system at run-time if the according bitstreams are located in local memory. The configuration can be performed with IOCTL calls to the *VCM* (*/dev/vcm*) while the correct address space for a new module is set by making a few number of IOCTL calls

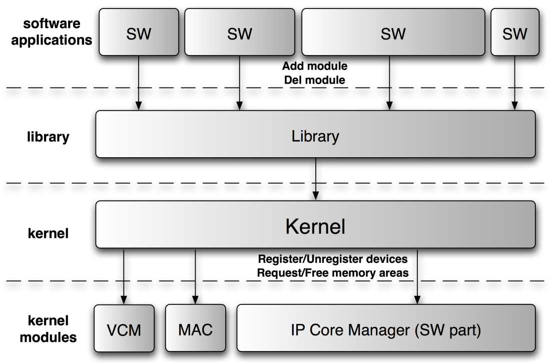

Figure 5. Software implementation hierarchy

to the *MAC* (*/dev/mac*) device. When the reconfiguration is finished the driver for the configured module can be loaded by following the standard flow described in [4]. The main advantage of the proposed solution is that all the details of the reconfiguration process are hidden to the software application due to a layered approach that is used to implement the software architecture. This is achieved by a systematic abstraction from hardware implementation details to very high level system calls and easy-to-use device drivers, as shown in Figure 5.

With this approach, software applications only need to know the name of the desired module to make a *module request* function call to the *Library*. According to this, the Library handles the corresponding IOCTL calls to perform the whole reconfiguration process by using the previously introduced kernel modules. In particular, in order to handle a module request, the *Library*

- downloads the right bitstream to free resources on an FPGA;
- assigns free address space to the new module;
- loads the correct device driver to manage the requested module;
- creates a device in */dev* with the major number of the device driver.

The only value returned to the application is the device name that allows access and use of the new module. The same concept is used when an application does not need a module any more. In this case, the application makes a *module release* function call to the *Library* with the name of the according module, after which the *Library* marks the selected module as *inactive*. Inactive modules are deleted from the FPGAs when the system runs out of available reconfigurable resources. Otherwise, they stay in the system until a software application makes a corresponding module request. This caching technique can significantly decrease the average configuration time.

4 Experimental Results

The performance of the proposed system architecture is affected mainly by the configuration time introduced by the partial reconfiguration of the FPGA (hardware latency) and by the overhead caused by the operating system (software latency).

The hardware latency is in turn composed of three parts. First, a constant time that is required to initiate the DMA transfer of the partial configuration bitstreams from the SDRAM to the configuration interface (VCM). Second, the time required to initialize the configuration interface of the FPGA and to flush the configuration buffer at the end of the configuration. The third part is the time needed to download the bitstream to the FPGA. While the data transfer time and the initialization time introduce a constant overhead, the configuration time depends on the size of the requested module.

Table 1. Hardware Reconfiguration Latency

Columns	Latency
4	1469.88 μs
8	2920.12 μs
12	4370.36 μs

The static time is composed of 158 clock cycles before reconfiguration and 824 clock cycles for buffer flushing after reconfiguration. In the worst case the number of clock cycles needed to reconfigure one CLB column of the used Xilinx Virtex-II FPGA (XC2V4000) is 18,128. If no data compression is used for the partial bitstreams, the worst case time to reconfigure a hardware module in the proposed MFCA system architecture (T_{CFG}) can be approximated by:

$$T_{CFG}(N) = (158 + N \cdot 18128 + 824) \cdot 20\ ns \qquad (1)$$

where N is the number of reconfigured CLB columns. The reconfiguration clock period used in our prototypical implementation is 20 ns. Table 1 shows the reconfiguration time introduced by the hardware for typical module sizes if only CLB columns are used. The download time changes insignificantly if embedded multipliers or uninitialized BlockRAMs are used. If the BlockRAM content shall also be written during reconfiguration, an additional 1054.72 μs apply per BlockRAM column. The modules used for elaboration vary in complexity and range from very simple designs to complex arithmetic units (e.g., a floating point unit). The software latency is shown in table 2. The presented results were measured by means of a hardware timer, physically implemented on the FPGA. The times were verified by a standard software timer running on the embedded PowerPC. The first task that has to be executed is the setup of the device driver, which loads the correct driver and initializes all necessary devices for a specific module. This takes quite a long time – approximately 650 ms on average – but it is executed only once for each IP-Core. Module loading takes about 3450 μs, while reading and writing from and to a configured module takes around 3.6 μs to read 4 bytes and 2.7 μs to write 4 bytes.

Furthermore, additional FPGA resources are required to implement the Virtex Configuration Manager (VCM) introduced in Section 3.4. Our implementation of the VCM uses 1726 slices (18.6%) and 6 BlockRAMs (6.8%) of the Xilinx Virtex-2Pro FPGA (XC2VP20). The high area requirement is caused mainly by the integrated readback functionality, which will be used in future implementations to extract internal states of loaded modules as required, e.g., for defragmentation. The other components of our implementation are also required in a static system architecture and the additional resources that are required for partial reconfiguration (e.g., the extended bus bridge described in Section 3.4) can be neglected since the resource overhead in these components is smaller than 1%.

Table 2. SW Solution Performances

Task	Time	Notes
Device driver setup	650 ms	once each driver
Module loading	3450 μs	each loading
Read	3.6 μs	4 bytes read
Write	2.7 μs	4 bytes write

5 Concluding Remarks

With the Multi-FPGA Clustered Architecture presented in this paper, an arbitrary number of FPGAs can be attached to an on-chip processor system as dynamically and partially reconfigurable resources. All available resources can be used to hold system components that are loaded to and erased from the FPGA at run-time. For the management of the reconfigurable resources, an embedded Linux is implemented on an embedded processor that hides all physical details of the system from user applications. Loading and erasing dynamic components, the Linux offers a driver that massively eases the use of dynamic reconfiguration. Similarly, drivers are provided for each loaded dynamic system component, managing all communication issues between applications and loaded components. The MFCA is a scalable architecture whose amount of available resources can be adapted easily. The MFCA has been implemented on the RAPTOR2000 prototyping platform and has proven to be a promising architecture for future work. The reconfiguration speed of currently available FPGAs can be fully exploited. In following studies we will analyze the potential of the presented MFCA to be used as node in a reconfigurable supercomputer.

References

[1] N. W. Bergmann and J. Williams. The Egret Platform For Reconfigurable System-On-Chip. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*, pages 340–343. IEEE, 2003.

[2] O. Diessel and H. A. ElGindy. Run-time compaction of FPGA designs. In *Field-Programmable Logic and Applications, 7th International Workshop (FPL)*, volume 1304 of *Lecture Notes in Computer Science*, pages 131–140. Springer, 1997.

[3] A. Donato, F. Ferrandi, M. D. Santambrogio, and D. Sciuto. Exploiting partial dynamic reconfiguration for soc design of complex application on fpga platforms. In *IFIP VLSI-SOC 2005*, 2005.

[4] A. Donato, F. Ferrandi, M. D. Santambrogio, and D. Sciuto. Operating system support for dynamically reconfigurable soc architectures. In *IEEE-SOCC*, 2005.

[5] L. Dozio and P. Mantegazza. Linux Real Time Application Interface (RTAI) in low cost high performance motion control. In *Motion Control 2003, a conference of ANIPLA, Associazione Nazionale Italiana per l'Automazione*, 2003.

[6] H. Kalte, B. Kettelhoit, M. Koester, M. Porrmann, and U. Rückert. A system approach for partially reconfigurable architectures. *International Journal of Embedded Systems*, 1:274–290, 2005.

[7] H. Kalte and M. Porrmann. REPLICA2Pro: task relocation by bitstream manipulation in Virtex-II/Pro FPGAs. In *Proceedings of the Third Conference on Computing Frontiers*, pages 403–412. ACM Press, May 2006.

[8] M. Koester, H. Kalte, and M. Porrmann. Relocation and defragmentation for heterogeneous reconfigurable systems. In *Proceedings of ERSA '06*, pages 70–76, Las Vegas, USA, June 27-30 2006. CSREA Press.

[9] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In *DAC '04: Proceedings of the Design, Automation and Test in Europe Conference and Exibition*. ACM Press, 2004.

[10] H. K.-H. So and R. W. Brodersen. Improving usability of FPGA-based reconfigurable computers through operating system support. In *Proc. of the Int. Conf. on Field-Programmable Logic and Applications (FPL)*, Madrid, Spain, Aug. 2006. IEEE Computer Society.

[11] J. Williams and N. Bergmann. Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip. In T. P. Plaks, editor, *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 163–169. CSREA Press, June 2004.