Virtualized On-Chip Distributed Computing for Heterogeneous Reconfigurable Multi-Core Systems

Stephan Werner[1], Oliver Oey[1], Diana Göhringer[2], Michael Hübner[1], Jürgen Becker[1]
ITIV, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany[1]
Fraunhofer IOSB, Ettlingen, Germany[2]
{stephan.werner, oliver.oey, michael.huebner, becker}@kit.edu[1]
diana.goehringer@iosb.fraunhofer.de[2]

Abstract—**Efficiently managing the parallel execution of various application tasks onto a heterogeneous multi-core system consisting of a combination of processors and accelerators is a difficult task due to the complex system architecture. The management of reconfigurable multi-core systems which exploit dynamic and partial reconfiguration in order to, e.g. increase the number of processing elements to fulfill the performance demands of the application, is even more complicated. This paper presents a special virtualization layer consisting of one central server and several distributed computing clients to virtualize the complex and adaptive heterogeneous multi-core architecture and to autonomously manage the distribution of the parallel computation tasks onto the different processing elements.**

Keywords- Multiprocessor, Virtualization, Parallel Computing, FPGA, Reconfigurable Computing

I. INTRODUCTION AND MOTIVATION

Nowadays, multi-core systems are available in many systems to fulfill the performance requirements of the applications. Embedded systems mainly use heterogeneous systems, consisting of processors, accelerators and a communication infrastructure, as these achieve a higher energy efficiency compared to homogeneous systems. However, achieving an efficient scheduling and mapping for the parallel application tasks onto these systems is a very complex task. For heterogeneous reconfigurable multi-core systems an additional challenge is the exploitation of the mutual exclusive tasks in the applications and the runtime selection of an appropriate underlying hardware for these application tasks.

Due to the increasing chip-size of modern FPGAs, multi-core systems on a single FPGA, distributed over multiple FPGAs, or combinations between powerful external processor cores and soft-core processors on the FPGA can be found. An example for such a system is the Xilinx Zynq platform which combines an ARM A9 dual-core processor with a Xilinx FPGA. Furthermore, Xilinx supports dynamic and partial reconfiguration. This allows adapting the hardware architecture at runtime to the demands of the application. This is for example done in the RAMPSoC [1] multi-core system. For such a system it is not only sufficient to schedule the execution of the application tasks and map them onto a specific processing element. These kind of systems have a further degree of freedom in terms, that the operating system (OS) has to select if a task will be executed in software or in hardware based on the performance and energy constraints of the application. To virtualize the complexity of such a system a common programming model is important.

In this paper a system architecture is presented, which allows a virtualized on-chip distributed computing. It uses the message passing interface standard programming model (MPI [2]) for the communication between the software tasks. Furthermore, the MPI programming model was extended to include also the communication with hardware accelerators. The task scheduling and mapping onto the processing elements is done using a distributed virtualization layer, which also hides the low-level communication routines required for the inter-processor communication. Due to this, the user only sees a single command shell and is not aware on which physical cores the application is executed.

The paper is organized in the following manner: Related work is presented in Section II. The design and functionality of the system architecture is given in Section III. Section IV and its subsection describe the distributed virtualization layer consisting of a Master-OS, several Slave-OSes and special OSes for handling hardware accelerators. A case study and results are given in Section V. Finally, the paper is closed by presenting the conclusions and an outlook to future work in Section VI.

II. RELATED WORK

Traditionally, operating systems, such as Linux, are responsible for the scheduling and allocation of software executables. However, the adaptation of the underlying hardware architecture at runtime using dynamic and partial reconfiguration, like it is done in reconfigurable systems, involves additional challenges for operating systems. These operating systems need to be able to schedule and allocate both, software as well as hardware tasks. The scheduling and the allocation of hardware tasks additionally involve the knowledge of the availability of hardware resources and also the time which is required for the reconfiguration.

Examples for such operating systems are ReConfigME[3], ReconOS[4] and BORPH[5]. All these systems are based on a single processor with several hardware accelerators. Therefore, all software tasks are mapped on the processors and only the scheduling and mapping of the hardware accelerators is considered. This restricts these systems, as they cannot execute multiple independent applications in parallel.

With our new approach the handling of hardware accelerators is integrated into the design flow as well as into the MPI system it is running on. This way multiple applications can be handled at the same time by either supplying enough processors or speeding up the execution with specially designed accelerators. In contrast to Hthreads[6], our approach is based on a distributed memory multi-core architecture, which leverages the memory bottleneck, as each task has its own local memory.

III. THE SYSTEM ARCHITECTURE

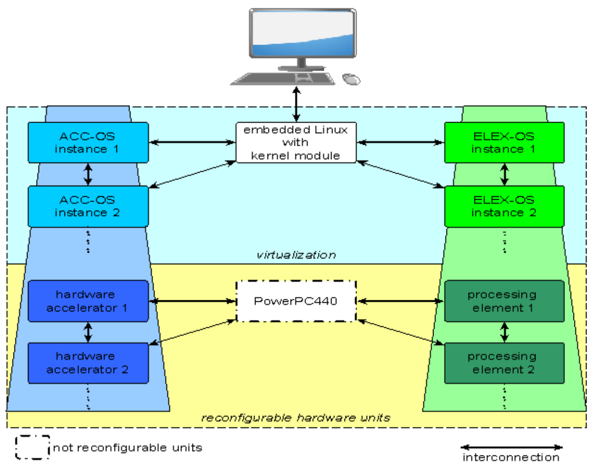

Figure 1. Virtualization layers

A. Hardware Architecture

The system uses a Network-on-Chip (NoC) called Star-Wheels NoC [1] which is a hybrid between star and spidergon [7]/wheel topology. For data transfers packet- and circuit-switched routing is supported. This allows high performance data channels and best effort packet communication. The NoC is divided into 4 Subnets each containing one Superswitch and up to 7 Subswitches which are connected with the processing elements via a Star-Wheels interface.

To make the reconfiguration of the MicroBlaze (MB) processors easier to use, an encapsulated version is introduced. The new MB module consists of a MB together with its block RAM (BRAM) (Figure 2.). With the integrated interface and subswitch they can directly be connected to the NoC. The memory is divided into 2 parts, differentiated by the use of separate BRAM blocks. The OS is located in the upper part whereas the lower part is reserved for user applications. The lower part can be configured to use up to 128 KB.

Hardware accelerators have the same interface to the network to allow the exchange between processors und accelerators at runtime via reconfiguration. A minimal MB is used as a microcontroller. It handles the channel creation in the network and sends configuration data to the accelerator. Depending on the configuration of the MB it is also possible to use it for other tasks. A minimum of 4 KB of BRAM is sufficient for the MB to work as the microcontroller; extra memory is only needed when other applications should also run on the MB.

For reconfiguration an additional interface is necessary. It is placed between the NoC and the processing elements (PE). It is used to disable the clock and the communication between the PEs to avoid that disturbing signals can leave the modules.

Like illustrated in Figure 1. the two kinds of reconfigurable modules are virtualized by an interface that is provided by a kernel module we developed for an embedded Linux running on a PowerPC440. To the user, who wants to run a program on this architecture, the reconfiguration of the needed modules and distribution of particular parts of the complex application is transparent and will be done by the kernel module.

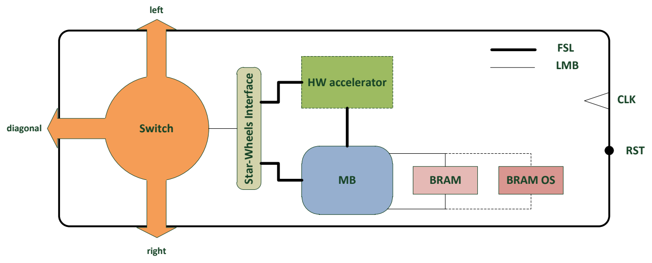

Figure 2. Microblaze module with accelerator or additional BRAM for operating system

B. Assisted Design Flow

The MB modules and hardware accelerators are designed to be used as a pcore in Xilinx Platform Studio (XPS). They consist of VHDL files and pre-synthesized netlists. As the layout of the modules has always to be the same, the design is assisted by scripts. The most important options of the MB processors can be changed (number of pipeline stages, floating point unit (FPU), multiplier, divider, barrel shifter, BRAM size). This information can be accessed via the Processor Version Register (PVR) at runtime. Addresses of the network and other parameters of the switches can be changed later in the XPS.

Designing HW accelerators is done in a similar way. With the help of a script a template can be created, where the new accelerator can be added as VHDL code or as a netlist. A generated VHDL file where the interface and file/entity names are predefined can be used as a starting point for own designs.

IV. THE VIRTUALIZATION LAYER

A. Embedded Linux as Master

In our approach an embedded Linux system serves as master. It uses the kernel version 2.6.37+ and runs on a PowerPC440. This system handles the reconfiguration, placement and scheduling of the tasks running on several PEs. These PEs communicate over the Star-Wheels-NoC.

To support and abstract the communication with the Star-Wheels-NoC with MPI, two separate components are used: a loadable kernel module which handles all internal processes such as the placement and an MPI-library which abstracts the file operations calling methods of the kernel. So the user can code a normal MPI-application and the interaction between the MPI-library and the module assure a correct execution.

To solve a greater problem the user writes a complex MPI-application and then runs some tools of a specialized tool flow [1]. These tools accomplish a communication analysis and cluster functions to tasks. This helps separating the complex application so that it can be run on several PEs in parallel. The only thing the user must do to start the execution of this parallelized and distributed application is to run an MPI-program on the master. It sends an XML-file to the kernel module with the MPI_Init-call. This file contains information about the tasks and their relations. Additionally, it specify which tasks should be placed directly side by side to utilize one special feature of the Star-Wheels-NoC: neighbored units can directly communicate over their Subswitches. This reduces latencies and disburdens the Superswitch. This approach could also be of interest in other NoC-topologies szch as a 2D-Mesh where each PE can have up to 4 best neighbors.

```
<task>  <ID>1</ID>
    <child>
        <ID>2</ID> <cost>50000</cost>
    </child>
    <child>
        <ID>3</ID> <cost>80000</cost>
    </child>
    <BestN>
        <ID>3</ID> <cost>80000</cost>
    </BestN>
    <SecBest>
        <ID>2</ID> <cost>50000</cost>
    </SecBest>
</task>
…
<task>  <ID>5</ID>
    <BestN>
        <ID>3</ID> <cost>20000</cost>
    </BestN>
    <SecBest>
        <ID>4</ID> <cost>500</cost>
    </SecBest>
</task>
```

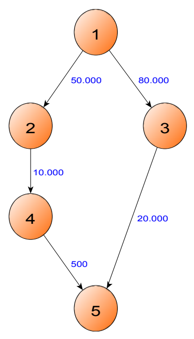

Figure 3. Task graph described by the XML-file

To do so, the XML-file only makes notes about a relative placement of the tasks as shown in Figure 3. The module takes this information and generates with the algorithm shown in Figure 4. a linked list with all tasks corresponding to the neighborhood (Figure 5.). After building this list the kernel module tries to place the tasks in the network. For this purpose it takes the task graph (TG) information the XML-file contains. Since the kernel module knows the available resources it tries to find an area where the list can be mapped. If it finds a subnet where all tasks can be placed as stated in the list then the kernel module reconfigures them starting with the root-node of the TG. If no such area exists the module searches relations in the list with relatively small traffic between the tasks and interprets them as predetermined break points. So the module can split the list and must than only place the resulting fragments of the list. This can be repeated until the fragments contain only one task to place.

```
cnt_tasks = get_tasks();
make_matrix(cnt_tasks);

for_all_tasks_in_xml
            id = get_id();
            if ( !exists(id) )
                        append_new_node();
                        if ( !start ) start = id;

            analyze_BestN();
            analyse_SecBest();
            analyze_all_GoodN();
            adapt_comm_cost_with_matrix();
```

Figure 4. Body of the algorithm for relative placement

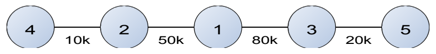

Figure 5. List reasulting by analyzing the neighborhood-information

The facility to split the list and place fragments is important when the user runs several applications at the same time. This will lead to a fragmentation of the free slots in the NoC.

Due to the fact that there are two different kinds of PEs in the network, two different operating systems (OS) are needed. On the normal processors ELEX-OS is running. On the microcontrollers of the accelerator nodes which can be used by any other node in the network, ACC-OS is used.

B. *ELEX-OS for Slaves*

The ELEX-OS (ELF executing OS) runs on the normal processors placed in the NoC. In our approach we use MicroBlazes. The major task of ELEX-OS is to load and run executable code transferred from the master und to manage the communication with the successors of the current node.

After the processor is started, ELEX-OS waits for a request from the master. The system gets the address of itself and the the server from the received request and sends back an answer. When the master has received the answer-packet, it knows that the processor that it just reconfigured is ready for use.

While the code is executed the interrupt-handler is active. So any incoming packet that is received by packet-switching is processed by this handler. Data packets are copied in the receive buffer of the running application. All other packet types such as flexible packets are handled by this handler.

Also, ELEX-OS has its own optimized MPI-library to support point-to-point-communication with MPI_Send and MPI_Recv. Furthermore some broadcasting operations are supported, such as MPI_Bcast, MPI_Scatter. When the executed application calls MPI_Scatter the interaction of the MPI-library and ELEX-OS manages the communication of the affected PEs.

C. *ACC-OS for accelerators*

At the moment this OS runs for test and verification issues on a minimized MicroBlaze which has only a barrel-shifter as special hardware feature. In this configuration the MicroBlaze needs 1240 LUTs and 965 FFs. The ACC-OS (Accelerator OS) basically presents a finite state machine (FSM) (Figure 6.). It is responsible for the registration on the master and the handling of incoming packets over the command-channel of the Star-Wheels-Interface. The registration process is the same as in ELEX-OS. After that the handling of the accelerator hardware begins by sending an ACC_READY-command to the server. The server answers with information about the successors the node has. Now, ACC-OS knows the receivers of the output data of the accelerator hardware. To do so, ACC-OS builds up a channel to the receiver in the same way the sender builds up a channel to the accelerator node. If both channels are created the accelerator hardware is directly coupled to the NoC.

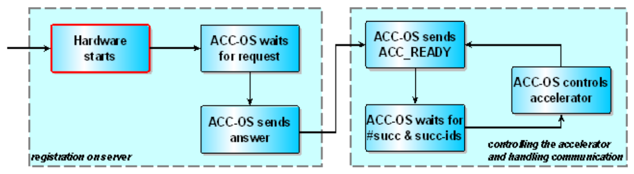
Hardware starts
ACC-OS waits for request
ACC-OS sends answer
registration on server
ACC-OS sends ACC_READY
ACC-OS controls accelerator
ACC-OS waits for #succ & succ-ids
controlling the accelerator and handling communication

Figure 6. FSM of the ACC-OS

The developer must regard only a few constraints to use ACC-OS. The accelerator must understand the control commands of ACC-OS, e.g. GET_STATE. It also must send a DONE to ACC-OS after all data are processed.

To use the accelerator in an MPI-application the developer can write a function that initializes the accelerator hardware, e.g. with a quantization matrix, by using MPI-commands. This function can be introduced using MPI_Op_create at runtime.

D. Calling accelerators with MPI

The MPI_Op_create command normally is used to define own operands for MPI_Reduce. Here it is expanded for defining own operations to access an accelerator node in the NoC. So that ELEX-OS knows that an accelerator is to be used if this function is called, it must be introduced using MPI_Op_create. This defines a bypass over an internal function which handles of the accelerator and calls the operation the developer passes to MPI_Op_create as argument (Figure 7.).

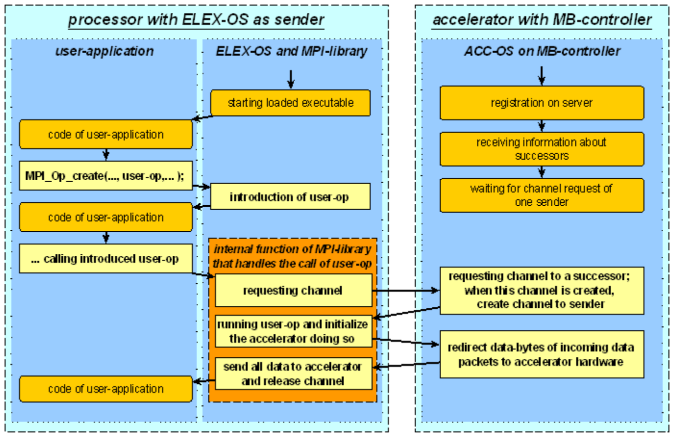
processor with ELEX-OS as sender
user-application
ELEX-OS and MPI-library
starting loaded executable
code of user-application
MPI_Op_create(..., user-op,...);
introduction of user-op
code of user-application
internal function of MPI-library that handles the call of user-op
... calling introduced user-op
requesting channel
running user-op and initialize the accelerator doing so
send all data to accelerator and release channel
code of user-application
accelerator with MB-controller
ACC-OS on MB-controller
registration on server
receiving information about successors
waiting for channel request of one sender
requesting channel to a successor; when this channel is created, create channel to sender
redirect data-bytes of incoming data packets to accelerator hardware

Figure 7. Interaction of ELEX-OS and ACC-OS using MPI_Op_create

Every time the MPI-application calls the user-operation, the MPI-library and ACC-OS manage the channel-creation-processes. Then ELEX-OS jumps into the user-operation to send the data for initializing the accelerator. After returning from it the MPI-library and ELEX-OS handle communication with the accelerator unit in the NoC. Because of the existing channels for receiving and sending words the accelerator can process one word per clock period.

V. CASE STUDY AND RESULTS

For testing purposes an example design was built on a Xilinx ML507 evaluation board with a Virtex 5 XC5VFX70T FPGA, as shown in Figure 8. It consists of one PowerPC440 processor which runs Linux and 3 partitions which can be filled with either a MB with 5 pipeline stages, barrel shifter, multiplier and 32KB block RAM or a 3x3 Sobel filter controlled by a small MB with 3 pipeline stages and a barrel shifter. The PowerPC clock runs at 400 MHz, the rest of the system at 100 MHz.. For reconfiguration the system uses the XPS-ICAP (Internal Configuration Access Port). The reconfiguration is done by the Embedded-Linux kernel module running on the PowerPC. The bitstreams are located on a PC and accessed over NFS. Before reconfiguration bitstreams and executables are prefetched by the module by copying them in the main memory of the board. The partial bitstreams have a size of 308,699 bytes. The average duration of a reconfiguration is about 9.1 milliseconds.
As a simple application scenario image filtering with a Sobel operator is realized. The generated XML-file contains the relations of the three tasks: 1 sender, 1 accelerator and 1 receiver. This information is used by the master to place the tasks on partitions of the FPGA that correspond with ports of a Superswitch of the Star-Wheels-NoC. The software running on the sender and the receiver communicates with the accelerator using MPI. Therefore a user-operation is introduced to the system using MPI_Op_create. This user-operation initializes the matrix of the Sobel filter. Then the data can be sent to the Sobel filter and 1 clock cycle after that received by the second MB.

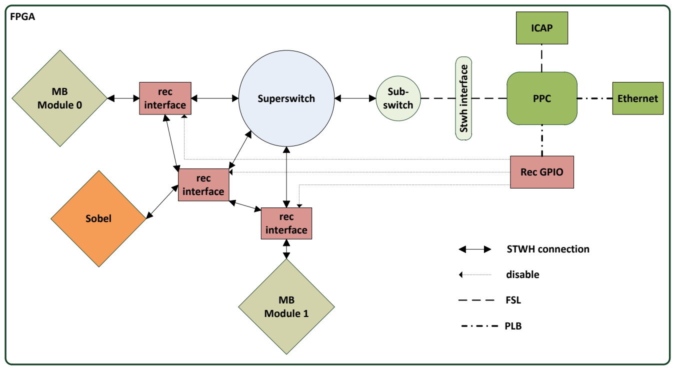

Figure 8. Implemented System on the ML507 board

VI. CONCLUSIONS AND OUTLOOK

With our new approach it is possible to use special hardware accelerators designed for special tasks in an existing MPI application. As the system supports the execution of multiple applications at the same time, the accelerators can even be re-used if the applications share some common tasks. The approach is very flexible due to the placement of processing elements which are reconfigured in the system at runtime. The information the user provides to the master in form of an XML-file only contains some recommendations of the neighborhood of each task. So the placement algorithm can decide autonomously at runtime where in the network the tasks should be reconfigured regarding to the occupancy of the available partitions on the FPGA.
Further steps are the implementation of ACC-OS in hardware using a FSM to achieve better results in area consumption and timing behavior of the accelerating nodes in the NoC. Additionally, a defragmentation of the subnets could be implemented to ease the constraints for the placement algorithm.

REFERENCES

[1] D. Göhringer: “Flexible Design and Dynamic Utilization of Adaptive Scalable Multi-Core Systems”, PhD thesis, 2011, Verlag Dr. Hut München
[2] MPI: A Message-Passing Interface Standard, Version 2.2, Message Passing Interface Forum, Sept. 2009. Available at: www.mpi-forum.org
[3] G. Wigley, D. Kearney, M. Jasiunas: “ReConfigME: A Detailed Implementation of an Operating System for Reconfigurable Computing”; IPDPS 2006, April 2006.
[4] Enno Lübbers. Multithreaded Programming and Execution Models for Reconfigurable Hardware. PhD thesis, Computer Science Department, University of Paderborn, 2010. Logos Verlag Berlin.
[5] H. K.-H. So, R. Broderson: “A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers using BORPH”; ACM Trans. on Emb. Comp. Sys., vol. 7, no. 2, pp. 14:1-14:28 , Feb. 2008.
[6] Jason M. Agron, Hardware Microkernels – A Novel Method for Constructing Operating Systems for Heterogeneous Multi-Core Platforms, PhD thesis, University of Arkansas, August 2010
[7] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, A. Scandurra: “Spidergon: a novel on-chip communication network”; In Proc. of Intern. Symposium on SoC, Nov. 2004