Efficient Execution of Process Networks on a Reconfigurable Hardware Virtual Machine

Matthias Dyer, Marco Platzner and Lothar Thiele
Computer Engineering & Networks Lab
Swiss Federal Institute of Technology (ETH) Zurich, Switzerland
<dyer@tik.ee.ethz.ch>

Abstract— **In this paper we present a novel use of an FPGA as a computing element for streaming based application. We investigate the virtualized execution of dynamic reconfigurable tasks. We use the process networks model as a coordination language which is interpreted on a virtual machine run-time system. We present and discuss the results of a design space exploration, which evaluates the performance of the system architecture for different configurations.**

I. Introduction

Flexibility is achieved by virtualizing hard- and software components. A streaming application executed on a reconfigurable embedded node can be virtualized on different levels. In model-based design, such applications are specified explicitly by a directed graph where the nodes are tasks, which represent computations and the arcs represent communication. The tasks are arbitrary subprograms and are specified in a conventional programming language such as C or VHDL, but the interaction between tasks is defined by a precise semantics. We call the language defining this interaction the *coordination language*. Examples of coordination languages are *process networks*, *synchronous dataflow (SDF)* graphs or *petri-nets*.

A language can be either interpreted or compiled. Our approach is to virtualize and interpret the coordination language and to use precompiled and optimized tasks. Thus, we achieve a promising trade-off between flexibility and performance.

We have chosen the process network model as an example for the coordination language, since it has a simple semantic and makes the task-level parallelism and communication explicit. A process network consists of a set of processes (tasks) which operate in parallel and communicate over unidirectional FIFO channels. Each of the processes performs computation on its private state space. The computation is interleaved with communication actions that read data from input channels and write data to output channels.

With precompiled tasks and an interpreted coordination language, system designers and application programmers can benefit from a number of advantages. (1) Since the application is split up into tasks, the run-time system can potentially execute arbitrary large programs which would normally not fit on a single device. (2) Using a coordination language allows the application programmer to easily build new algorithms by a new composition of existing tasks. (3) For the precompiled tasks, the efficient synthesis-, place- and route tools from the manufactures can be used, which leads to efficient and optimized execution. (4) Since the coordination language is not compiled, the information about the task interaction and communication is retained and can be used to analyze the performance of the system at any time.

The work presented in this paper was supported by ETH Zurich under the Polyproject 'Miniaturized Wearable Computing' and the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

II. Related Work

One related group uses the process network model for streaming applications, but only in a static scenario without reconfiguration [1]. Another group is working towards a general reconfigurable operating system, which does not investigate a specific model for the optimized execution of streaming based applications [2] [3] [4]. There are very few existing concepts, which investigate both the dynamic reconfigurability and a specific coordination language for streaming based applications. The SCORE [5] compute and execution model is closely related with the work presented in this paper. However, in contrast to SCORE, we present a run-time system which is compliant with today available FPGA devices. In addition we present a prototype implementation and performance results that can be compared to other similar system architectures.

III. Run-Time System

A run-time system is needed to schedule and reconfigure the tasks dynamically. The run-time system is a virtual machine, since it abstracts the underlying hardware and it provides the execution of tasks as a service to the application programmer. It hides the exact implementation of the service but guarantees an execution with the semantics of the coordination language. An overview of the run-time system is depicted in Figure 1. The two main components are a partial reconfigurable FPGA and a CPU.

a) Task Slots: The computation is done in the task slots, which are continuously reconfigured by the loader. A task correspond to the processes in the process network model. They have a number of input- and output ports which are virtually connected to the according FIFO channel. In order to load a task as a partial reconfigurable core, a precise and static task interface and protocol was defined. A task usually has an internal state and variables. When the scheduler preempts a task, its context therefore needs to be saved and restored after its next instantiation. We apply an internal state save

Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)
0-7695-2230-0/04 $ 20.00 IEEE

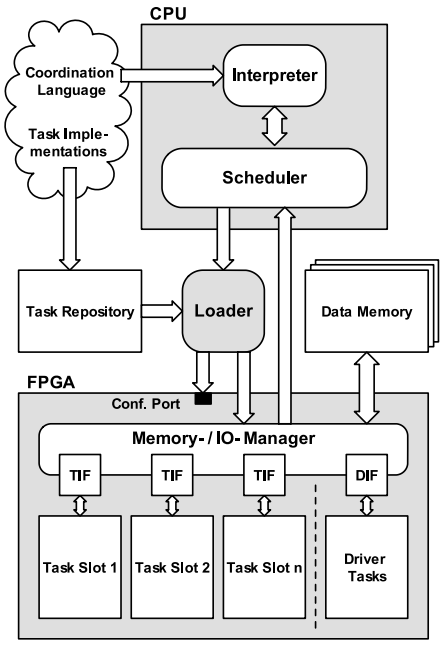

Fig. 1. Overview of the virtual machine run-time system on a embedded reconfigurable node.

mechanism, which automatically stores the content of the tasks state registers into memory and loads it back upon next instantiation.

b) Loader: The loader receives commands from the scheduler, which defines the task to load and the target slot. The tasks are stored in the task repository as partial bitstreams. The loader modifies the header information inside the bitstreams to set the target. The loader is connected the configuration port of the FPGA. The time for partial reconfiguration linearly depends on the size of the task slots.

c) Memory Manager: The tasks communicate with the memory and I/O-manager (MM). The MM acts as a crossbar for the data memory and sends events to the scheduler. The FIFO channels are implemented as circular buffers and the data is stored in the data memory of the run-time system. The MM computes the addresses for the data memory and handles the communication. For a flexible management of the FIFO channels, the MM uses internally a virtual address space. To increase the memory bandwidth, the data memory can consist of multiple physical devices. We also allow to use the on-chip dual-ported block-memory.

d) Scheduler: Based on the events, provided by the memory manager, the scheduler decides which task to load next. Conceptually, all tasks in the process network run in parallel. In our run-time system, this parallelism is reduced to the number of task slots in the FPGA. In order to reduce the number of context switches, a task should run as long as possible before being preempted. We apply a data driven scheduling [6] strategy, where the tasks are activated on the availability of data. A task is running until it reads from an empty channel or writes to a full channel (blocking reads/writes).

e) Driver Tasks: Driver tasks are typically connected to I/O pins of the FPGA to handle external devices, such as streaming input- or output devices. This handling usually requires the driver tasks to be present continuously in order to meet the timing requirements e.g. of a communication protocol. As a consequence a driver task can not be reconfigured and scheduled by the run-time system. We have addressed this problem by creating a static slot, which includes all the driver tasks.

IV. Performance Results

We have implemented the run-time system to obtain quantitative results on the performance. One the one hand we have implemented a parameterizable emulator for the exploration, on the other hand we have built single-slot prototype system from which we can obtain realistic input data for the emulator. The prototype is based on a custom FPGA module (Xilinx SpartanII200 FPGA, 256 kByte SRAM, 8 MBit Flash and a Xilinx Coolrunner CPLD) which is connected to an IPAQ PDA. The emulator is implemented in VHDL and can load and execute the same tasks as the prototype. A number of parameters can be set to define the run-time system such as the number of slots, the memory configuration or the properties of the scheduling algorithm. We use the ModelSim VHDL simulator to execute and profile the applications.

We have executed a sample application with our run-time system emulator, to do a number of experiments and to obtain quantitative data. The application is a transcoder with a stereo-to-mono mixer in the ADPCM compressed format. It consists of seven tasks, including two streaming driver tasks, two decoder-, one encoder- and two filter tasks. The process network of the sample application is depicted in figure 2.

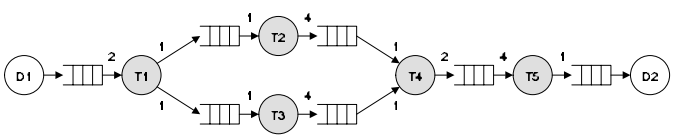

Fig. 2. Process network of the sample application.

The first experiment shows the effect of the total memory capacity and the number of slots on the performance and latency of the system (Figure 3).

We have measured the performance for systems with one slot, where the tasks are sequentially configured, up to five slots, where every task run in its own slot. The performance of the five-slot system is not dependent on the total amount of memory as the other systems, since after the first instantiation, no further reconfiguration is required. This system actually correspond to a static solution. The overhead in terms of execution time of the other systems is due to the reconfiguration and not to the virtualization. With little memory, the tasks fill or empty the FIFO channels with little capacity so fast, that the task slots stay most of the time in the reconfiguration state or the state which waits for the access to the configuration port. The performance increases if more data memory is used.

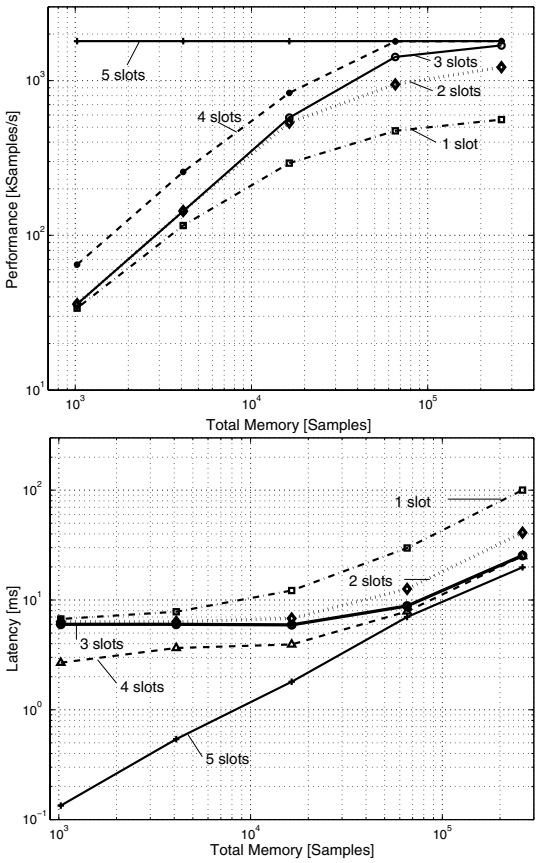

Fig. 3. Performance and latency of the sample application in the run-time system.

In the first experiment, we did not restrict the access to the data memory. The maximal number of possible simultaneous memory accesses is equal to the number of task slots plus the number of driver tasks. However, the access is restricted by the memory configuration. In figure 4 we show the effect of restricting the access to the data memory to one simultaneous access such as in our prototype. The experiment is done with a total memory capacity of $64K$ samples.

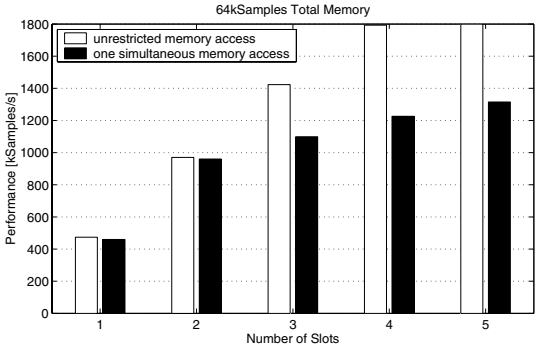

Fig. 4. Performance with restricted access to the data memory.

In the figure, we can see that the static five-slot solution is faster by a factor of two, compared to the two-slot system, with unrestricted memory access. If there is only one single-ported memory device such as in our prototype, this factor decreases to 1.37.

V. Conclusions and Future Work

In this paper we presented a virtual machine run-time system for reconfigurable embedded nodes, which efficiently execute streaming applications. We discussed the implementation of the process network model and the advantages of an interpreted coordination language. We further explained the functionality of the run-time system and presented quantitative results, which are obtained from a prototype implementation and a parameterizable system emulator.

We have shown that arbitrary large process networks can be executed with a virtual machine and dynamic reconfiguration. The run-time system executes an application with low overhead. The performance of the case study application, executed on a two-slot run-time system is close to the one of a static implementation. For such a system architecture, it is the first time that quantitative results are presented, which are based on a realistic streaming application. We have shown the effect of the number of task slots and different memory configurations on the performance.

As a next step, we want to make use of the analyzability of the applications that are specified in the coordination language. We plan to investigate formal verification of real-time properties, which can be checked at design-, load- or run-time. We further work on the implementation of an enhanced prototype with multiple task slots and multiple memory devices.

An extended version of this paper is available in [7].

VI. Acknowledgements

We would like to thank Herbert Walder, Roman Plessl and Peter Fercher on their work on the prototype and Christian Plessl for his expertise on virtualization.

References

[1] C. Zissulescu et al., "Laura: Leiden Architecture Research and Exploration Tool," in *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL)*. LNCS 2778, Springer-Verlag, 2003, pp. 911–920.

[2] H. Simmler, L. Levinson, and R. Männer, "Multitasking on FPGA Coprocessors," in *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications (FPL)*. LNCS 1896, Springer-Verlag, 2000, pp. 121–130.

[3] G. Brebner and O. Diessel, "Chip-Based Reconfigurable Task Management," in *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications (FPL)*. LNCS 2147, Springer-Verlag, 2001, pp. 182–191.

[4] H. Walder and M. Platzner, "Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations," in *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*. CSREA Press, June 2003, pp. 284–287.

[5] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream computations organized for reconfigurable execution (SCORE)," in *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL)*. LNCS 1896, Springer-Verlag, 2000, pp. 605–614.

[6] T. Parks, "Bounded Scheduling of Process Networks," Ph.D. dissertation, Berkeley, University of California, 1995.

[7] M. Dyer, M. Platzner, and L. Thiele, "Efficient Execution of Process Networks on a Reconfigurable Hardware Virtual Machine," Swiss Federal Institute of Technology (ETH) Zurich, TIK Report Nr. 192, April 2004.