Virtualizing Hardware with Multi-Context Reconfigurable Arrays

Rolf Enzler, Christian Plessl, and Marco Platzner

Swiss Federal Institute of Technology (ETH) Zurich*, Switzerland
E-mail: enzler@ife.ee.ethz.ch

Abstract. In contrast to processors, current reconfigurable devices totally lack programming models that would allow for device independent compilation and forward compatibility. The key to overcome this limitations is hardware virtualization. In this paper, we resort to a macro-pipelined execution model to achieve hardware virtualization for data streaming applications. As a hardware implementation we present a hybrid multi-context architecture that attaches a coarse-grained reconfigurable array to a host CPU. A co-simulation framework enables cycle-accurate simulation of the complete architecture. As a case study we map an FIR filter to our virtualized hardware model and evaluate different designs. We discuss the impact of the number of contexts and the feature of context state on the speedup and the CPU load.

1 Introduction

Reconfigurable computing fabrics have shown great potential in many high-performance applications that benefit from hardware customization while still relying on some amount of programmability. A major drawback of current reconfigurable devices, in particular field-programmable gate arrays (FPGAs), is the lack of programming models. Applications are compiled (synthesized) to given fixed-size hardware. The resulting configuration bitstream cannot be reused to program a device of different type or size. Thus, to leverage advances in VLSI technology, i. e. increased transistor count and higher clock rates, a complete recompilation is required.

The key to overcome this limitation is *hardware virtualization* [1–3]. In order to achieve hardware virtualization, we have to define a set of basic operators a hardware can execute. Together with a description of the data flow (communication paths between operators) and the control flow (sequencing of operators) a hardware programming model is defined that compilers can target. Processors use a well-established form of hardware virtualization and define an instruction set architecture that decouples the compiler from the actual hardware organization. Achieving virtualization of reconfigurable hardware is more complex. Reconfigurable hardware excels when computations are organized spatially. The

* This work is supported by ETH Zurich under the ZIPPY project and the Wearable Computing Polyproject.

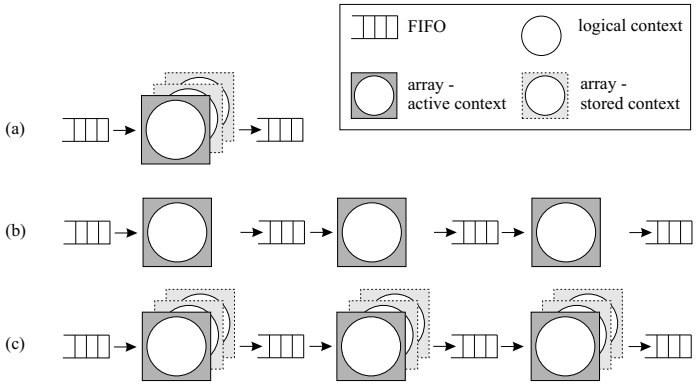


Fig. 1. Models for virtualized macro-pipelining

basic operators will thus have greater complexities than processor instructions and the number of possible operators is very large. Further, the reconfigurability allows to implement many basic operators with just one type of hardware block.

In this paper, we consider data streaming applications that map well to (macro-)pipelines, where one pipeline stage is implemented by one basic hardware block. Our basic hardware block is a 4×4 coarse-grained reconfigurable array. The inputs and outputs of the array connect to FIFO buffers to facilitate data streaming. One set of configuration data for the array is denoted as a *context*. Applications are organized by pipelining several *logical context* executions. Although this model is rather restrictive, it is amenable to true hardware virtualization and targets an important application domain.

Figure 1(a) shows our model with one physical array that is reconfigured to implement logical contexts as needed. To minimize or even hide the reconfiguration time the array stores multiple *physical contexts*. Figure 1(b) displays an alternative implementation with several single-context physical arrays arranged in a pipelined fashion. The arrays are still reconfigured to execute different logical contexts. However, as several contexts run in parallel the throughput increases. Both multiple contexts and physical pipelining can be combined which is shown in Fig. 1(c). All these architectures achieve virtualization as they provide the logical pipeline of array executions as programming model, but differ in their performance and hardware cost.

While there exists already a substantial body of work on coarse-grained arrays, macro-pipelining of stream computations and multi-context devices, a system-level evaluation of the performance and the various features of multi-context devices is missing. To this end, we form a reconfigurable hybrid system by coupling our multi-context array to a CPU. The CPU takes care of data I/O, context loading, and control of the multi-context array. We develop a system-wide, cycle-accurate architecture model and investigate the following issues by means of a co-simulation environment: First, we determine the performance gains for the hybrid over the CPU only, depending on the number of physical contexts.

Second, we try to identify whether and when the capability to resume the state of a previous context is advantageous. Third, we measure the CPU load for the different designs.

Section 2 summarizes related work. The hybrid architecture model and our co-simulation environment are discussed in Section 3. Section 4 presents an FIR filter case study, while Section 5 discusses the results. Finally, Section 6 summarizes our findings and points to further work.

2 Related Work

PipeRench [1, 4] is a reconfigurable architecture that supports hardware virtualization. The device is organized into a physical pipeline of stripes, which represent the minimal reconfigurable hardware blocks. A stripe's output is strictly pipelined and connects to the next stripe via an interconnection network. Thus, PipeRench is similar to the model in Fig. 1(b). Fast reconfiguration of stripes is supported by 256 contexts held on-chip. Each stripe comprises 16 processing elements, which implement addition/subtraction or a programmable logic function. Application kernels are mapped to virtual pipeline stages. During runtime, the virtual stages are configured to the physical stripes that are available on the device. The implementation described in [4] features 16 physical stripes.

Multi-context techniques for both fine-grained and coarse-grained reconfigurable devices have been investigated by several researchers. DeHon [5] demonstrated that adding multi-context support to FPGAs can increase computational density. Due to the moderate contribution of the configuration memory to the total chip area, a small number of contexts can be added with reasonable impact on cost. Trimberger et al. [2] introduce a multi-context extension of the Xilinx XC4000 architecture. The proposed device holds eight contexts on-chip. The flip-flops of the device are eight times replicated and each logic cell can write to any of these flip-flops. The authors propose to use the multi-context feature for emulation of arrays of arbitrary size. The fine-grained, memory-poor architecture prefers logic emulation rather than macro-pipelining.

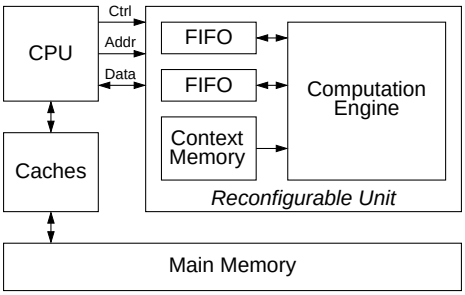
PACT's XPP device [6] uses cells with a functionality similar to our model, but targets a different execution model. Data is transfered between the cells using a handshake protocol. This ensures that dataflow dependencies are met and makes the computation self-timed. Configurations are loaded on demand using a hierarchical configuration management. A configuration context is not necessarily activated for the whole device at the same point in time. For each cell, the new configuration is activated as soon as the current configuration is not used anymore. MorphoSys [7] integrates a CPU with a coarse-grained, multi-context ALU array. The device holds 32 contexts on-chip.

Our work targets a coarse-grained, multi-context reconfigurable hybrid. The main differences to related approaches are that we focus on macro-pipelining of contexts that execute for a longer time period, use FIFOs to transfer data between contexts, and couple the reconfigurable array with a CPU to form a hybrid device.

3 Architecture Model and Co-simulation

3.1 System Model

We investigate a hybrid reconfigurable device, which couples a coarse-grained reconfigurable unit closely to a CPU core. Figure 2 outlines the basic system model, which comprises the CPU core, instruction and data caches, and the reconfigurable unit (RU). The reconfigurable unit is attached to the CPU via a dedicated coprocessor interface and provides a number of coprocessor registers.



RU coproc. registers	CPU
RU reset	W
FIFO 1	R/W
FIFO 1 fill level	R
FIFO 2	R/W
FIFO 2 fill level	R
Cycle count register	R/W
Context memory [1...n]	W
Context select register	W

R: read access, W: write access

Fig. 2. System model outline

We have developed a co-simulation framework that combines a cycle-accurate CPU model with an RU model specified in VHDL and allows for cycle-accurate simulation of the whole system. Details on the design and implementation of the co-simulation framework have been published in [8].

Currently the RU does not have its own memory access port, but all data communicated to and from the RU is passed via the CPU's register file. On the RU side, data transfers are performed via the FIFO buffers. Both FIFOs are readable and writable by the CPU as well as the RU.

The synchronization mechanism between CPU and RU is similar to the one proposed in the Garp processor [9]. The execution of the RU is started by writing the number of clock cycles the RU shall perform to the *cycle count register*. In every clock cycle, the cycle count register is decremented by one and stops the execution of the RU when reaching zero. By reading the cycle count register the remaining execution cycles can be determined.

3.2 CPU Model

For CPU simulation, we leverage on the SimpleScalar processor simulator [10]. SimpleScalar's CPU model is based on a 32-bit RISC processor architecture and has a MIPS-like instruction set. The CPU core's data and control path as well as the memory hierarchy are widely parameterizable. Thus, the CPU model can be configured to resemble a broad range of CPU architectures, from small embedded CPUs to powerful high-end CPUs.

In order to couple the RU to the CPU, we have extended SimpleScalar with a coprocessor interface. To this end, coprocessor read and write instructions have been added to the instruction set, which allow the CPU to access the coprocessor registers of the RU.

3.3 Model of the Reconfigurable Unit

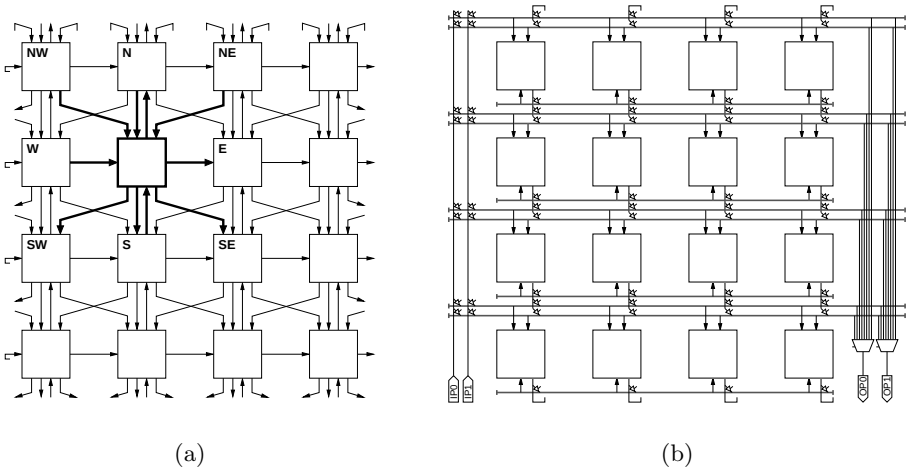
The RU model comprises two FIFO buffers, the context memory, and the computation engine. Some RU characteristics are parameterizable: the data path width, the depth of the FIFO buffers, and the number of configurations the context memory holds. Another RU parameter determines whether the contexts contain state or not. An RU with context states replicates the registers in the data path in a way that each context is assigned a separate set of registers. An RU without context states provides only one register set that all contexts must share.

The context memory holds a set of configurations for the computation engine. The configuration data is written from the CPU to the RU via the configuration interface. The RU supports the download of full and partial configurations for any of the contexts. The CPU selects a context on the RU for execution by writing the number of the context to the *context selection register*. The context is immediately switched and the CPU can trigger the RU to run by writing the desired number of cycles to the *cycle count register*.

The computation engine is a 4×4 array of homogeneous, coarse-grained cells, which are connected by a 2-level network: direct interconnects between certain adjacent cells, Fig. 3(a), and horizontal buses between cell rows, Fig. 3(b). The computation engine has two input and two output ports, which are connected to the two FIFOs of the RU. Inside the computation engine, they are routed via the horizontal buses.

Figure 4 outlines the data path of a cell consisting of a fixed-point arithmetic logic unit (ALU), several multiplexers and registers. Figure 4(a) shows a cell without context state; all contexts have to share the same registers which are reset on context switches. Figure 4(b) displays a cell supporting context state. All the registers are replicated according to the number of physical contexts. This allows to preserve register values over several context switches. Alternatively, the register can also be reset on a context switch. The ALU implements the common arithmetic and logic operations (addition, subtraction, shift, OR, NOR, NOT, etc.) as well as multiplication. The control signals for the ALU and the multiplexers are part of the RU's configuration. The configuration contains also a constant operator, which can be routed to both ALU inputs.

The configuration of the computation engine is responsible for the functionality of the cells and the routing of the data path between the cells, from the input ports to the cells, and from the cells to the output ports. Since the configuration incorporates constant cell operators, the amount of required configuration bits depends on the datapath width. Given a datapath width of 16 bit, the configuration data results in 918 bits.



(a) (b)

Fig. 3. 2-level interconnect scheme of the computation engine: (a) direct interconnects (*highlighted connections of one cell*), and (b) horizontal buses and I/O ports (*IPx*, *OPx*)

4 Case Study and Experimental Setup

4.1 FIR Filter Partitioning and Mapping

As a case study, we have implemented a 56th-order FIR filter on our virtualized hardware. The filter is implemented as a cascade of eight subfilters of 7th-order. The input samples are processed in data blocks. An FIR filter implementation requires delay registers. These registers form the state of the context, which must be saved between two executions of the same context. Depending on the capabilities of the reconfigurable array, there are two ways to achieve this:

- If all contexts of the RU share the same register set, the state must be explicitly saved and later on restored. For the filter implementation this is achieved by overlapping subsequent data blocks, which forms an execution overhead.
- If the RU provides dedicated register sets for each context the state is kept automatically. For the filter implementation, no extra cycles are needed for state handling if we can hold all logical contexts on the array.

4.2 System Model Setups

We have set up our system model to study the following cases: CPU only (no RU present), CPU with attached single-context RU, CPU with attached multi-context RU having 2, 4 and 8 contexts, and finally a CPU with attached 8-context RU incorporating a dedicated register set for each context.

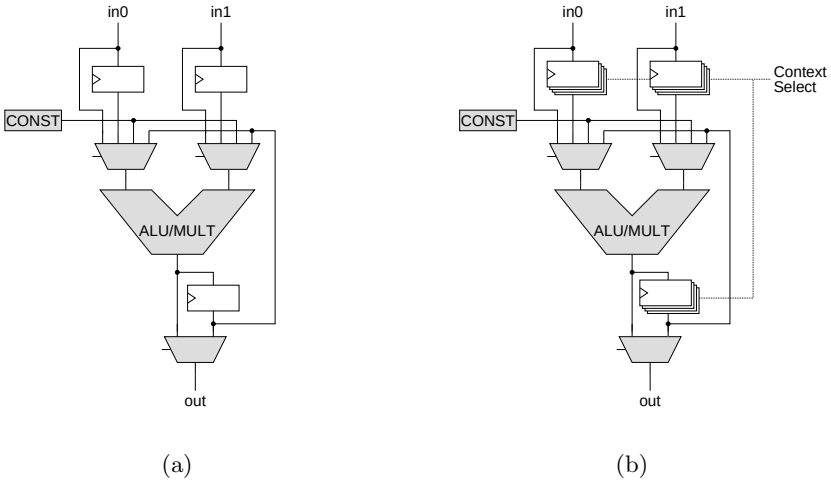


Fig. 4. Data path of a cell: (a) with *a single register set* for all contexts, and (b) with *a dedicated register set* for each context. The *shaded parts* are controlled by the configuration

The SimpleScalar CPU model is configured such that it resembles an embedded CPU. Table 1 lists the most important CPU parameters. In each experiment, 64K samples organized in data blocks are processed. The size of the data blocks depends on the FIFO depth available on the RU (cf. Fig. 2). We vary the depth of the FIFO buffers between 128 and 1k words.

For the coprocessor cases, a data block is written to the RU, processed sequentially by the eight FIR filter stages (the eight logical contexts), and finally read back. At the beginning, a controller task running on the CPU downloads as many contexts as fit onto the RU. If not all logical contexts fit, the contexts are loaded on demand. Each time a filter context is required that is not present,

Table 1. CPU model resembling an embedded CPU

Parameter Class	Setup
Computation units	1 int. ALU, 1 int. multiplier 1 FP ALU, 1 FP multiplier
Caches	32-way 16K L1 I-cache, 32-way 16K L1 D-cache, no L2 cache
Memory interface	32-bit memory bus, 1 memory port
Queue sizes[1]	instruction fetch: 1, register update unit: 4, load/store: 4
Bandwidths[1]	decode width: 1, issue width: 2, commit width: 2
Execution order	in-order
Branch prediction	always not-taken

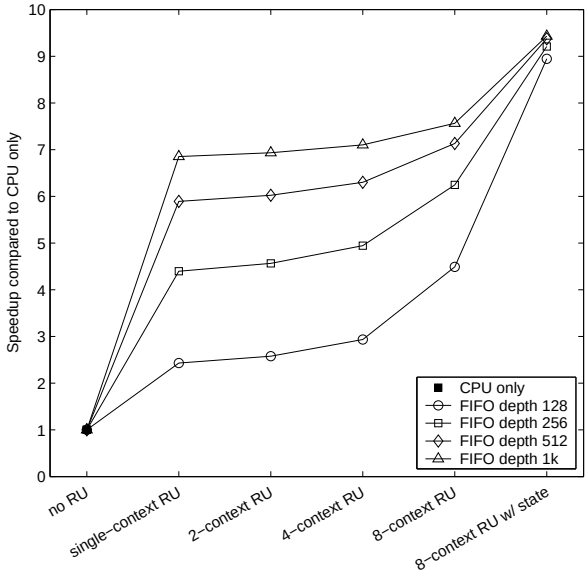
[1] in number of instructions

the controller performs the download by overriding always the same physical context. This is done to hold as many contexts as possible unchanged on the array with the goal to reduce the amount of reconfiguration data that has to be downloaded onto the RU.

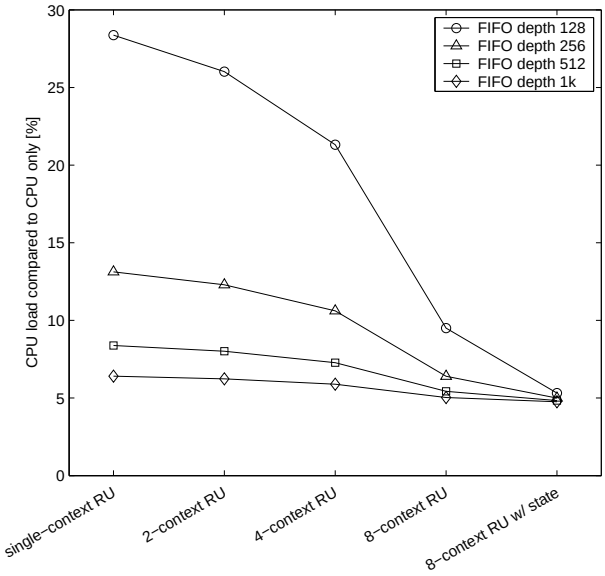
5 Results and Discussion

Figure 5 illustrates the results of the experiments as functions of the device architecture (shown on the horizontal axis) and the FIFO buffer size. The execution time of the filter for the CPU only is 110.65 million cycles. Figure 5(a) shows the speedups relative to this computation time and Fig. 5(b) presents the CPU load. We assume a real-time system that filters blocks of data samples at a given rate. When the filter computation is moved from the CPU to the reconfigurable array, the CPU is relieved from these operations and can use this capacity for running other tasks. However, the CPU still has to transfer data to and from the FIFOs, write contexts to the RU on demand, and control the context switches. The load given in Fig. 5(b) determines the spent CPU cycles normalized to the CPU only system. We point out the following observations:

- Hardware virtualization is an extremely useful concept. We were able to run the same filter implementation on reconfigurable array models with different features without resynthesizing the application.
- Using an RU we achieve significant speedups, ranging from a factor of 2.4 for a 128 word FIFO single-context device up to a factor of 8.9 for an 8-context RU that restores context state.
- The performance of the system in terms of speedup and CPU load depends on the length of the FIFO buffers. Enlarging the FIFOs increases the performance and at the same time the filter delay. Practical applications could limit these potential gains by imposing delay constraints. For instance, a 2-context RU using a FIFO with 1k words instead of 128 words improves the speedup by a factor of 2.7, while increasing the latency by a factor of 8.
- Figure 5 shows that a multi-context array storing the context states greatly benefits our application if we can store *all* logical contexts. In this case, we can avoid the overlapping of data blocks. For an 8-context array with a 128 word FIFO the speedup increases by factor of 2.0. In addition, as no reconfiguration is required, the speedup becomes almost independent of the FIFO size.
- Employing a reconfigurable coprocessor not only speeds up the computation but also lowers the CPU load significantly. As Figure 5(b) displays, for a single-context RU the CPU load drops from 100% to 28.4% for a 128 word FIFO and to 6.4% for a 1k word FIFO. Increasing the number of physical contexts the load approaches the asymptotic value of 4.8%, because the CPU task reduces to data transfer and context switches.



(a) Speedup



(b) CPU load

Fig. 5. Performance figures in comparison to the CPU only case

6 Summary and Future Work

In this paper, we have discussed the concept of hardware virtualization and the use of multi-context architectures to achieve it. We have presented a co-simulation framework based on a hybrid system model consisting of a reconfigurable unit attached to a CPU. As a case study, we have mapped an FIR filter to our virtualized hardware and run it on various architectures by cycle-accurate simulation. The results show that hardware virtualization is a valuable concept and that multi-context features can be successfully employed. Further work includes:

- Implementation of application types that require more complex context sequences (control flow).
- Integration of a dedicated RU memory port.
- Investigation of context prediction and prefetching techniques.
- Development of an area model for the reconfigurable unit in order to quantify the hardware overhead introduced by the multi-context features.

References

1. Goldstein, S.C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., Taylor, R.R.: PipeRench: A reconfigurable architecture and compiler. IEEE Computer **33** (2000) 70–77
2. Trimberger, S., Carberry, D., Johnson, A., Wong, J.: A time-multiplexed FPGA. In: Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM). (1997) 22–28
3. Caspi, E., Chu, M., Huang, R., Yeh, J., Wawrzynek, J., DeHon, A.: Stream computations organized for reconfigurable execution (SCORE). In: Field-Programmable Logic and Applications (Proc. FPL), LNCS 1896, Springer-Verlag (2000) 605–614
4. Schmit, H., Whelihan, D., Moe, M., Levine, B., Taylor, R.R.: PipeRench: A virtualized programmable datapath in 0.18 micron technology. In: Proc. 24th IEEE Custom Integrated Circuits Conf. (CICC). (2002) 63–66
5. DeHon, A.: DPGA utilization and application. In: Proc. 4th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA). (1996) 115–121
6. Baumgartne, V., May, F., Nückel, A., Vorbach, M., Weinhardt, M.: PACT XPP – a self-reconfigurable data processing architecture. In: Proc. 1st Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA). (2001) 64–70
7. Singh, H., Lee, M.H., Lu, G., Kurdahi, F.J., Bagherzadeh, N., Chaves Filho, E.M.: MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. IEEE Trans. on Computers **49** (2000) 465–481
8. Enzler, R., Plessl, C., Platzner, M.: Co-simulation of a hybrid multi-context architecture. In: Proc. 3rd Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA). (2003)
9. Hauser, J.R., Wawrzynek, J.: Garp: A MIPS processor with a reconfigurable coprocessor. In: Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM). (1997) 12–21
10. Austin, T., Larson, E., Ernst, D.: SimpleScalar: An infrastructure for computer system modeling. IEEE Computer **35** (2002) 59–67