Embedding FPGA Overlays into Configurable Systems-on-Chip: ReconOS meets ZUMA

Tobias Wiersema, Arne Bockhorn, and Marco Platzner
University of Paderborn, Germany
{tobias.wiersema, xoar, platzner}@upb.de

Abstract—**Virtual FPGAs are overlay architectures realized on top of physical FPGAs. They are proposed to enhance or abstract away from the physical FPGA for experimenting with novel architectures and design tool flows. In this paper, we present an embedding of a ZUMA-based virtual FPGA fabric into a complete configurable system-on-chip. Such an embedding is required to fully harness the potential of virtual FPGAs, in particular to give the virtual circuits access to main memory and operating system services, and to enable a concurrent operation of virtualized and non-virtualized circuitry. We discuss our extension to ZUMA and its embedding into the ReconOS operating system for hardware/software systems. Furthermore, we present an open source tool flow to synthesize configurations for the virtual FPGA.**

I. Introduction

Virtualization of resources has a long tradition in computing. Generally, virtualization is an abstraction technique that presents a different view on the resources of a computing system than the physically accurate one. Virtualization is mostly used to pretend users complete and exclusive access to a resource, to isolate users of a resource and guarantee their non-interference, to optimize resource usage, or to simplify application development by abstracting away from the details of physical devices.

Virtual FPGAs, also denoted as FPGA overlays, apply the concept of virtualization to the domain of reconfigurable computing. Interest in FPGA virtualization has been fueled by several motivations, which include overcoming the limited hardware resources, enriching the capabilities of existing FPGAs, achieving portability of reconfigurable logic implementations and, finally, providing an experimental testbed for FPGA architecture and CAD tool research.

Although research in FPGA virtualization has been increasing over the last decade, the field is still in its infancy with only a few prototypical systems described in literature. Recent work has been addressing two main issues with virtual FPGAs, to minimize the overheads of FPGA overlays with respect to area and speed and to devise novel reconfigurable architectures and corresponding tool flows. An equally important issue that has not been sufficiently addressed yet is the embedding of virtual reconfigurable fabrics into complete configurable systems-on-chip. Circuits configured onto virtual FPGAs can not exist in isolation, they require interfaces to other, non-virtualized reconfigurable hardware, memory, peripherals and software running on CPUs.

The novel contribution of this paper is the conceptual and practical integration of an FPGA overlay into a platform FPGA comprising CPU and reconfigurable hardware cores, managed by a Linux operating system. To this end we extend ZUMA [1], a freely available state-of-the-art virtual FPGA architecture and tool flow, and embed it into the ReconOS architecture and operating system [2], [3]. ReconOS is open source and semantically integrates reconfigurable logic cores into a standard Linux environment by introducing a multithreaded programming model not just for software but also for hardware. The resulting so-called hardware threads have complete access to all operating system services and virtualized main memory.

The paper is structured as follows. In Section II we review related work in the area of virtual FPGAs, in Section III we present our extensions of ZUMA and its tool flow and in Section IV we explain the integration of the extended ZUMA overlay into ReconOS. Section V shows experimental results and Section VI concludes the paper.

II. Related Work on Virtual FPGAs

Virtualization of reconfigurable hardware has been addressed in research for more than a decade. Early concepts of virtualization drew an analogy to virtual memory and proposed to load and remove reconfigurable hardware modules from an FPGA similar to pages of memory that can be swapped in or out of main memory, e.g., Brebner [4] or Fornaciari and Piuiri [5]. The main motivation of this work was to overcome the limited hardware resources of FPGAs.

Some years later, Lagadec et al. [6] introduced a definition of a virtual FPGA as a separate overlay on top of a physical FPGA. The authors discussed advantages of having an overlay that is unconstrained by the underlaying physical FPGA. The main advantages were described as the portability of circuits and, provided the virtual architecture is open and adaptable, as providing a means to investigate and experiment with new FPGA architectures. Lagadec et al. also mentioned potential disadvantages of using overlays, namely the area overhead, the reduced clock frequency, and a lack of tool chains for synthesizing to virtual FPGAs.

The concept of virtual FPGAs has also been used by researchers from the domain of evolvable hardware, e.g., Sekanina [7] and Glette et al. [8]. Evolutionary circuit design requires very frequent synthesis and evaluation of evolved circuit candidates. Synthesis and reconfiguration times for commercial fine-grained FPGAs have been found to be far too slow. Hence, most approaches in evolvable hardware leverage some form of coarse-grained reconfigurable architecture and reconfigure this overlay through setting multiplexers, a process denoted as virtual reconfiguration.

In 2004, Plessl and Platzner [9] published a survey of approaches for virtualization of hardware. One of the approaches uses an abstract overlay with a different architecture than the underlay and is denoted as virtual machine [10]. In this approach, the virtual machine is a runtime system that adapts and synthesizes the configuration for an abstract FPGA to an actual reconfigurable device. The configuration was termed hardware byte code.

Lysecky et al. [11] presented first measurements of an actual virtual FPGA, reporting a 100x area overhead and a 6x decrease in circuit performance through virtualization. They concluded that virtualization is only viable if circuit portability is of paramount importance. Brant and Lemieux later improved on these findings by presenting ZUMA [1], an FPGA overlay that lowers the area overhead to a reported 40x through careful architectural choices. One of these choices is to store the virtual configuration not in flip flops but in RAM built from lookup tables (LUTRAM), by far the most abundantly available resource on FPGAs. Brant and Lemieux also addressed the lack of tool chains for virtual FPGAs and used the well-known open source tool flow VTR (from Verilog To Routing) [12] to generate configurations for their virtual FPGA. The generator source code for ZUMA virtual FPGAs has been released as open source. Hübner et al. [13] present a system-on-chip with an ARM Cortex M1 soft core processor and a virtual FPGA on one physical FPGA. They describe their virtual FPGA and a supporting tool chain, which bases on VPR, the place & route tool at the heart of the VTR flow. Unfortunately, Hübner et al. do not present a quantitative analysis of the area overhead for their virtual FPGA and the architecture is not openly available to the research community.

Coole and Stitt present a slightly different approach to FPGA overlays called intermediate fabrics [14]. They do not address the advantages and disadvantages discussed in earlier work, but instead focus on FPGA synthesis times. Placing and routing sophisticated designs on high density devices using vendor tools can take hours or days, which Coole and Stitt consider a weakness. Consequently, they came up with intermediate fabrics as general concept for virtual overlays built from more coarse grained building blocks than lookup tables. These intermediate fabrics should greatly simplify the placement and routing steps, speeding them up by a factor of 800x. Fine grained virtual FPGAs can be seen as special case of this approach, albeit not a very interesting one, as place & route would not be significantly faster than for usual FPGAs.

In summary, we can identify a number of reasons why researchers have been looking into virtual FPGA architectures. First, portability of synthesized hardware designs across FPGA devices, families or even vendors is a long term goal and would help reduce dependence on single manufacturers and lower costs of migrating to new hardware. In addition, the overlay can provide architectural features the underlay lacks, for example, dynamic and partial reconfigurability. However, hardware portability still remains an active research topic rather than a practically used feature given the huge overheads in area and delay as well as the rather limited virtual architectures presented so far. Second, when speeding up place & route or the reconfiguration process is the main motivation then the overheads of current overlays might be bearable. Third, FPGA overlays are excellent experimental environments to study new reconfigurable architectures and design tool flows. This holds especially true if researchers have open access to virtual architectures and their bitstream formats, as well as to the corresponding tool flows.

In our work we follow the definition of a virtual FPGA provided by Lagadec et al. [6] and use an extended version of the original ZUMA virtual FPGA architecture and tool flow [1].

III. Extending ZUMA

ZUMA is the most advanced FPGA overlay freely available today. ZUMA was designed for a low virtualization overhead and the open reference implementation helps others to integrate it easily into any given design. We use and extend the original ZUMA tool flow depicted in Figure 1. We start with a parametrized description of the overlay that is translated to an architecture file in XML and a behavioral description of the virtual circuit in Verilog, and use the VTR flow tools ODIN, ABC and VPR to synthesize, technology map, as well as pack, place & route the virtual circuit. Additionally, VPR is used to compute an abstract routing graph for the overlay architecture. From this routing graph we generate the behavioral description of the overlay itself, to be synthesized to an actual FPGA, and from the netlist and placement & routing information we create the resulting bitstream that can be used to configure the virtual FPGA with the virtual circuit.

To operate ZUMA overlays in a complete hardware/software system-on-chip platform, we had to extend the reference implementation in several ways. In this section, we first describe our ZUMA extensions with regard to interfacing between the overlay and the underlay and to implementing sequential circuits, and then discuss ongoing and future work in making the tool flow for ZUMA more sophisticated.

A. Virtual-physical Interface

To embed a virtual FPGA architecture into a physical reconfigurable fabric we must define an interface, just as FPGAs have I/O pads to connect to off-chip components. This interface needs to have a fixed interpretation on both ends, the virtual and physical, because the embedding and wiring of the overlay into the underlay will be fixed once the overlay architecture is synthesized. All virtual configurations will have to adhere to the resulting virtual I/O pad locations. Since the ZUMA reference implementation had no concept of constraining the placement of virtual I/O pads, we have developed a work-around that effectively enables us to synthesize virtual circuits that can be connected to the underlay.

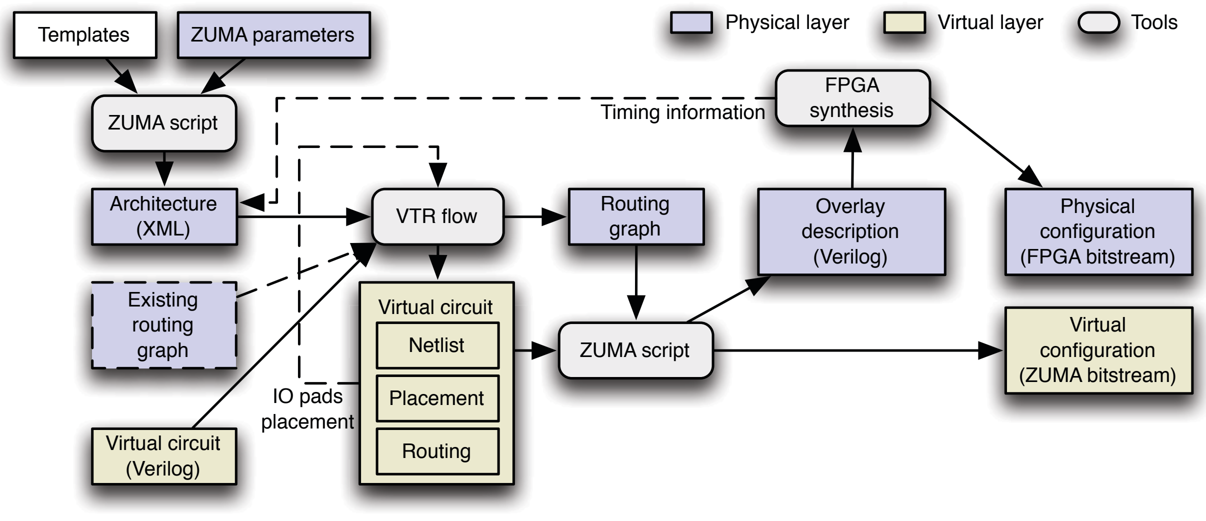

Fig. 1. Tool flow to create a ZUMA overlay and configurations for it. Dashed lines denote further improvements on the tool flow, described in Section III-C.

We have introduced an extra ordering logic layer between the underlay and the overlay, which we use to permute all of the I/O pads from their *random* locations to the locations in which the designer expects them. The ordering layer consists of large multiplexers for all the virtual I/O pads that we automatically insert in place of the virtual FPGA I/Os during the second ZUMA script stage of the tool flow. To that end we have extended ZUMA to offer support for larger multiplexers, which were capped in the original ZUMA design at the square of the number of LUT inputs of the physical FPGA k_{host}^2, e.g., 36 for modern FPGAs with 6-input LUTs. We automatically deduce the correct permutation, and thus the correct multiplexer configurations, from VTR's output files and include the corresponding configuration bits also into the bitstream. The ordering is thus transparent to the designer as it happens automatically in the background, if it is enabled.

B. Sequential Virtual Circuits

We have extended the ZUMA reference design by adding optional virtual flip flops after each virtual LUT to enable sequential virtual circuits. While the description of the ZUMA architecture [1] includes flip flops for this purpose, the actual reference implementation supported only combinational circuits. To create virtual flip flops in the most area conserving manner, we have developed a special version of the LUTRAM macro that is the basic building block of the ZUMA overlay. The special version is only used for the eLUTs, i.e., the embedded LUTs that are each implemented by a single LUTRAM macro, and not for any of the multiplexers of the virtual routing fabric, which are still implemented using the original LUTRAM macro. The new macro uses both possible outputs of the LUTRAM, the unregistered one, as before, and the registered one to be used as virtual flip flop. To make the flip flop optional, we have added a 2-input multiplexer node after each eLUT and derive its configuration from the netlist and routing of the virtual circuit. The LUTRAM macro accepts a second clock to be used with the registered output, so that the configuration of the overlay and its operation can be driven by different clocks. We have made use of this possibility, because in this way the clock network used for the overlay will be an actual clock network of the physical FPGA, allowing for fast clock signals that are synchronized with the underlay. Accordingly, we instruct the VTR flow to treat the clock of the ZUMA overlay as external network, which does not have to be routed using virtual resources.

C. Further Extensions to the ZUMA Tool Flow

For the experiments shown in this paper we have used the original ZUMA tool flow with the extensions described in III-A and III-B. The improvement and automation of the tool flow along the following three directions is ongoing and future work; Figure 1 shows them as dashed lines.

First, we are implementing a second work-around for constraining the placement of virtual I/O pads. This second work-around uses the optional *io.pads* placement file of the VTR tool flow, in which the location of each pad can be fixed using the pad names which are used internally in the flow, i.e., the names assigned by ODIN. Unfortunately VTR does not allow designers to specify a generic file for this purpose. Thus, one has to interrupt the ZUMA flow after VTR, generate the constraints for the I/O pads and then re-run the flow again from the VTR step with this new constraints file. Both work-arounds come with different trade-offs. The ordering layer requires additional logic but gives VTR full flexibility in placing I/O pads which can potentially lead to more efficient designs. Fixing the I/O pad locations saves the logic for the ordering layer but constrains VTR's placement.

Second, similar to related work on virtual FPGAs our current ZUMA tool flow lacks an accurate timing analysis. Running a timing analysis on the overlay architecture using the underlay's timing information results in extremely pessimistic delay estimates which are basically given by the longest possible combinational path in the overlay without taking into account the actual circuit configuration. To derive useful bounds on the clock frequency for the overlay, we are working on propagating the timing information from the underlay's synthesis tools back to the VPR tool flow for the overlay. Since VPR allows for timing information in the architecture description file, the ZUMA tool flow could easily be modified to use it as shown in Figure 1: The generator scripts of ZUMA create the hardware description of the overlay in Verilog. Then, we include this overlay into a ReconOS design and synthesize it. The resulting timing information for the physical paths that implement virtual edges is extracted from the synthesis and used to annotate the ZUMA architecture description. VPR will be run in the timing-driven clustering and packing mode in order to give a meaningful timing analysis of the virtual circuit. The resulting maximum clock frequency will be embedded in the ZUMA bitstream. In the system-on-chip architecture we will employ a solution such as the Xilinx Dynamic Reconfiguration Port to alter the output clock frequency of a Digital Clock Manager to the desired frequency for the ZUMA overlay at runtime.

Third, a disadvantage of the current ZUMA tool flow as shown in Figure 1 is the simultaneous processing of both layers in the VTR flow step, which means that the overlay description will be re-generated every time a new virtual configuration is computed. Apart from being unnecessary overhead, this can also lead to incompatibilities if a new VTR version introduces changes to the routing graph computation, as this would render ZUMA unable to generate any new configurations for previously generated overlays. Here we are working on an import routine for VTR that can parse routing graphs instead of generating them anew from the architecture description. Then new ZUMA configurations could be computed by starting with an existing routing graph from a previous run.

IV. Embedding Virtual Fabrics into ReconOS

ReconOS is an architecture and execution environment for hybrid hardware/software systems featuring a multithreaded programming model which allows for regular software threads as well as hardware threads [2], [3]. Figure 2 shows the architecture for a ReconOS v3.1 system on Xilinx Zynq with $n + 1$ software threads (SWT) and $m + 1$ hardware threads (HWT). Hardware threads are basically circuits that can interact with other threads and operating system services, such as semaphores, through a FIFO-based operating system interface (OSIF). To enable this interaction in a fully transparent way, ReconOS instantiates one delegate thread (DT) in software per running hardware thread. The delegate thread accesses the operating systems services and communicates with other threads on behalf of the hardware thread. Additionally, each hardware thread can access the ReconOS memory subsystem through a FIFO-based memory interface (MEMIF). Using ReconOS for embedding a virtual FPGA provides us with a mature, Linux-based infrastructure for implementing hardware/software systems, including a CPU core, memory controller, peripherals

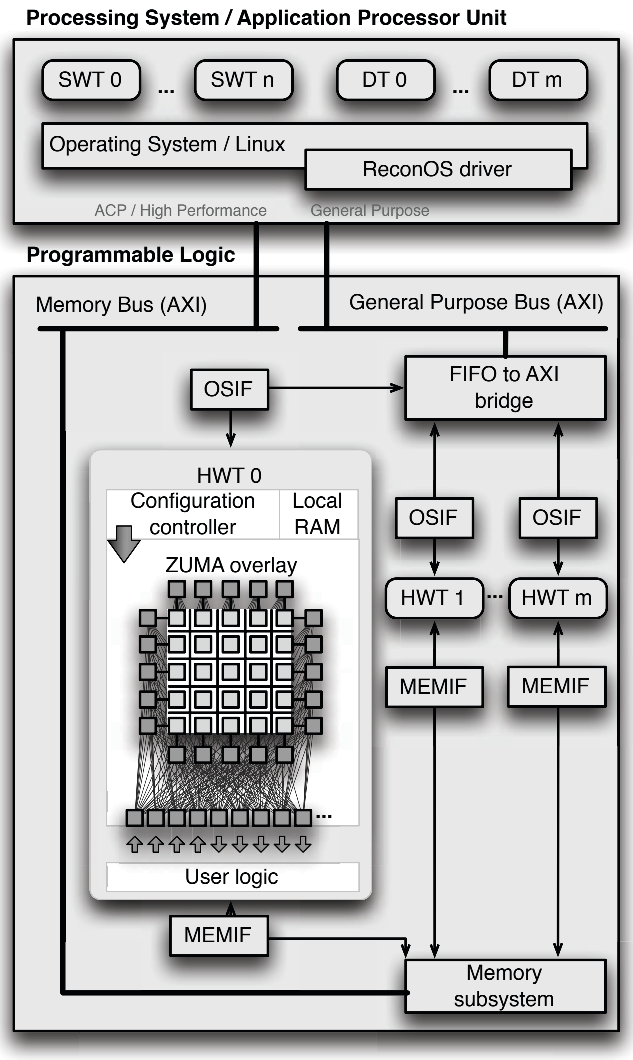

Fig. 2. Xilinx Zynq version of ReconOS [2], with $n + 1$ software threads (SWT), $m + 1$ hardware threads (HWT), their $m + 1$ delegate threads (DT), and a 5x5 ZUMA overlay embedded into HWT 0.

and a standard software operating system.

A ZUMA overlay can be embedded into several locations of a ReconOS architecture, depending on the intended use of the overlay. For example, we can embed the overlay into the MEMIF or OSIF to realize functions for controlling memory access or operating system interaction in virtualized hardware. Alternatively, we can integrate the overlay as separate core connected to an AXI bus. However, the most flexible way of embedding the overlay is to integrate it into a ReconOS hardware thread. Inside the hardware thread, the overlay can either replace or augment the non-virtualized user logic. Figure 2 depicts this integration option in HWT 0. The hardware thread contains a ZUMA overlay with an ordering layer for the virtual I/Os as described in Section III-A, a configuration controller including a local memory, and a block of non-virtualized user logic.

We have developed a prototype system as shown in Figure 2

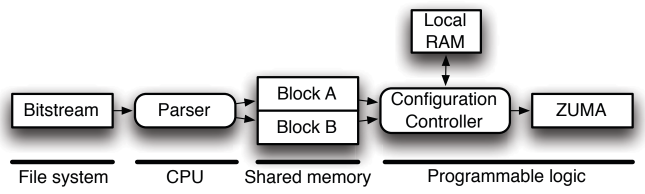

Fig. 3. Configuration process for the ZUMA overlay embedded in a ReconOS hardware thread.

including software functions for configuring and communicating with the virtual FPGA. The software connects to the hardware thread containing the overlay via message boxes and allocates a shared memory region in the system memory for data exchange. The subsequent configuration process is depicted in Figure 3. The software reads a bitstream for the virtual FPGA from the file system and parses it to verify the integrity of the bitstream using ZUMA's line checksums [1]. The file system in ReconOS versions using Linux as host operating system is either local or on a remote server connected via NFS. To better utilize the bandwidth to the ReconOS memory subsystem and thus to reduce configuration time, the software as well as the configuration controller for the virtual FPGA operate on 8 KiB blocks of configuration data. The shared memory region is operated in a double-buffering scheme. Each time one block of the shared memory is filled with new configuration data, the software sends a message to the hardware thread and continues to parse the bitstream into the second block. In the meantime, the configuration controller copies the first block into a local on-chip RAM and from there it feeds the configuration data into the ZUMA overlay, which shifts it into the LUTRAMs. Once the configuration data in the local memory blocks has been completely processed, the hardware thread send a message to the software to request the next block.

The software functions can be included in any user-defined software thread. In our test setup we have compiled them to an executable that can be called in ReconOS/Linux with the required bitstream as a command line parameter. After configuring the overlay, the software process remains connected with the hardware thread containing the overlay. New input data is continuously generated and sent to the hardware thread via message box calls. The hardware thread provides these data to the overlay using the I/O pad ordering layer. The outputs of the overlay are, again properly ordered, written to shared memory from where the executable can read and display them. Naturally, this is just a test setup we are using to showcase the integration of the overlay into ReconOS. The virtual circuit running in the overlay has full access to all ReconOS features, including all operating system services and the memory subsystem.

V. Experimental Results

In this section we report on experimental results for our extended ZUMA overlays embedded into ReconOS. We have used ReconOS v3.1 on an Avnet ZedBoard, containing a Xilinx Zynq integrated processing system with two ARM Cortex-A9 MPCore application processors and programmable logic fabric on a single die. The ZUMA layout we have synthesized is based on the island-style layout found in [1] and uses a 2d-array of nxm clustered logic blocks. Each cluster comprises 8 basic logic elements (BLE), and each BLE consists of one 6-input lookup table and one by-passable flip flop. Each cluster receives 28 inputs from outside the cluster and 8 feedback connections from the internal BLE outputs. Each cluster input and output input can connect to 6 different virtual wires of the surrounding channels, which are 112 virtual wires wide. We have generated all bitstreams, virtual and physical ones, on a machine with an Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz processor with 4 GiB RAM.

Table I lists hardware area and speed for different system configurations. The hardware area is measured in LUTs and LUTRAMs, where the LUTRAM count is also included in the LUT measure. LUTRAMs are listed separately, since only 50% of the LUTs available on our Zynq programmable logic fabric can act as LUTRAM, creating an area restriction for larger overlays. The second column of Table I presents as a reference data for a base ReconOS system with one empty hardware thread, i.e., no actual user logic. The third and fourth columns show data for a hardware thread with an embedded extended ZUMA overlay of 3×3 clusters with ordered and unordered I/Os, respectively. The data shows that our I/O ordering layer leads to a rather small area overhead with a 3.7% increase in area compared to an overlay with unordered I/O, mainly attributed for by added LUTRAMs. To quantify this area overhead also for larger overlays, we have measured the area increase through inserting the I/O ordering multiplexers for overlays of sizes from 2×2 up to 100×100 clusters. We have found that the addition of the ordering multiplexers resulted in area increases between 3% to 6%.

We have also estimated the area overhead for virtualization. To that end, we have compared the area required for implementing the overlay with the area required in the physical FPGA to realize a circuit with an equal number of LUTs. The 72-LUT overlay used for Table I needs $4,787$ LUTs to implement, so we achieve a 66x increase in area with our ZUMA enhancements, which is still rather close to the 40x reported area overhead for the original ZUMA [1]. The third and fourth columns of Table I also report the maximum clock frequency for the overlays; as discussed in Section III this is an overly pessimistic timing estimation. The last column of Table I displays the area requirements for the user logic and the configuration controller. The user logic added to the overlay implements only some basic functionality such as receiving new inputs from the software side and sending back the outputs, and together with the configuration controller needs about $1/7$-th of the size of the overlay.

Table II shows for differently sized overlays the area requirements, synthesis and reconfiguration times and the bitstream sizes. The area requirements in the second column

TABLE I
AREA AND SPEED MEASUREMENTS FOR RECONOS SYSTEM WITH AND WITHOUT A 3×3 OVERLAY.

Measure	ReconOs with bare HWT	ReconOs with ZUMA and ordered I/Os	ReconOs with ZUMA and unordered I/Os	HWT area usage (unordered I/Os) Overlay	HWT area usage (unordered I/Os) Config controller & UL
area [LUTs]	3,270	9,919	9,568	4,787	669
area [LUTRAMs]	181	5,877	5,557	4,784	0
max frequency [MHz]	102.05	0.71	0.83		

TABLE II
MEASUREMENTS OF RECONOS WITH OVERLAYS OF DIFFERENT SIZES.

Size	LUTRAMs	synth. [s]	reconf. [s]	bitstream [KiB]
1×1	757 (4%)	0.38	0.01	13 (2.3)
2×2	2,437 (14%)	0.53	0.03	53 (13)
3×3	5,077 (29%)	1.03	0.07	118 (30)
4×4	8,677 (49%)	1.62	0.13	206 (53)
5×5	13,237 (76%)	2.47	0.19	317 (85)
6×6	37,514 (>100%)	3.26	—	452 (121)

state the number of LUTRAMs and the LUTRAM utilization on our Zynq programmable logic fabric. Using the ZUMA architecture parameters detailed above, we can only fit overlays with a size of 5×5 clusters on our Zedboard. The synthesis of a new overlay configuration is quite fast; the runtimes in the third column of Table II comprise the complete tool flow from Figure 1 up to the virtual configuration, as well as the mentioned overhead of re-creating the whole overlay every time. The overlay reconfiguration times in the fourth column include every step from Figure 3, measured as wall time on the software side. In our ZUMA tool flow setup, bitstream sizes depend only on the virtual architecture and not on the actually implemented virtual circuit. The last column of Table II lists the virtual bitstream sizes and, in parentheses, the bitstream sizes when compressed using standard ZIP. On average the textual representation of ZUMA bitstreams allows for an 75% size reduction using compression. As expected the virtual bitstream sizes are quite small compared to the bitstream for the physical Zynq fabric, which amounts to 3.9 MiB.

VI. CONCLUSION

In this paper we have presented the embedding of an extended ZUMA virtual FPGA fabric into a ReconOS/Linux system running on a Xilinx Zynq. We have discussed our extensions to ZUMA and the detailed embedding into ReconOS, and presented experiments using our prototype architecture and tool flow. While our experiments have confirmed that FPGA overlays still come with high virtualization overheads, the main result of our work is the greatly simplified experimentation with virtual FPGAs. Circuits mapped to virtual FPGAs can now easily call Linux operating system services and thus communicate with other threads or machines and utilize a standard virtual memory subsystem. Future work includes further improvements and automation of the tool flow as well as an investigation of and experiments with portable reconfigurable circuits.

ACKNOWLEDGMENT

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre On-The-Fly Computing (SFB 901).

REFERENCES

[1] A. Brant and G. G. Lemieux, "ZUMA: An open FPGA overlay architecture," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, pp. 93–96.

[2] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An operating system approach for reconfigurable computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71. [Online]. Available: http://dx.doi.org/10.1109/MM.2013.110

[3] E. Lbbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, pp. 8:1–8:33, October 2009.

[4] G. Brebner, "The swappable logic unit: A paradigm for virtual hardware," in *Proceedings IEEE Symposium FPGAs for Custom Computing Machines (FCCM)*, 1997, pp. 77–86.

[5] W. Fornaciari and V. Piuri, "Virtual fpgas: Some steps behind the physical barriers," *Parallel and Distributed Processing*, pp. 7–12, 1998.

[6] L. Lagadec, D. Lavenier, E. Fabiani, and B. Pottier, "Placing, routing, and editing virtual fpgas," in *Field-Programmable Logic and Applications*. Springer, 2001, pp. 357–366.

[7] L. Sekanina, "Virtual reconfigurable circuits for real-world applications of evolvable hardware," *Lecture Notes in Computer Science*, pp. 186–197, 2003.

[8] K. Glette, J. Torresen, and M. Yasunaga, "Online evolution for a high-speed image recognition system implemented on a virtex-ii pro fpga," in *Proceedings 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE CS Press, 2007, pp. 463–470.

[9] C. Plessl and M. Platzner, "Virtualization of hardware introduction and survey," in *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. CSREA Press, jun 2004, p. 6369.

[10] Y. Ha, P. Schaumont, M. Engels, S. Vernalde, F. Potargent, L. Rijnders, and H. Man, "A hardware virtual machine for the networked reconfiguration," in *Proceedings IEEE International Workshop on Rapid System Prototyping*, 2000, pp. 194–199.

[11] R. Lysecky, K. Miller, F. Vahid, and K. Vissers, "Firm-core virtual FPGA for just-in-time FPGA compilation (abstract only)," *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays - FPGA'05*, 2005.

[12] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The VTR project: Architecture and CAD for FPGAs from Verilog To Routing," in *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2012, pp. 77–86.

[13] M. Hubner, P. Figuli, R. Girardey, D. Soudris, K. Siozios, and J. Becker, "A heterogeneous multicore system on chip with run-time reconfigurable virtual fpga architecture," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 143–149.

[14] J. Coole and G. Stitt, "Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2010 IEEE/ACM/IFIP International Conference on*. IEEE, 2010, pp. 13–22.