Virtualization of Hardware – Introduction and Survey

Christian Plessl and Marco Platzner
Computer Engineering & Networks Lab
Swiss Federal Institute of Technology (ETH) Zurich, Switzerland
<plessl@tik.ee.ethz.ch>

Abstract— **In this paper we introduce to virtualization of hardware on reconfigurable devices. We identify three main approaches denoted with temporal partitioning, virtualized execution, and virtual machine. For each virtualization approach, we discuss the application models, the required execution architectures, the design tools and the run-time systems. Then, we survey a selection of important projects in the field.**

Keywords: hardware virtualization, FPGA, partitioning

I. Introduction

Literally, hardware virtualization means that an application executes on virtualized hardware as opposed to physical hardware. "Virtual hardware" seems to be a contradictory term because hardware is supposed to have a physical existence. Furthermore, the terms "virtual hardware" and "hardware virtualization" are used for different concepts in literature.

Initially, the term virtual hardware was coined to show the analogy to virtual memory. There, pages of memory are swapped in and out a computing system, allowing applications to address a much larger memory than physically existent. In the same way, a reconfigurable computing system can swap in and out portions of the hardware by a reconfiguration process, allowing applications to use more hardware than physically existent. Later on, the term hardware virtualization was used to describe mapping techniques and architectures that allow for a certain degree of independence between the mapped application and the actual capacity of the target architecture. The most radical form of hardware virtualization is to strive for complete independence of the hardware execution from the actual underlying hardware.

The purpose of this paper is to introduce into the field of hardware virtualization, to discuss the main issues involved, and to provide a survey of important projects in this field. In Section II, we identify three basic approaches of hardware virtualization: temporal partitioning, virtualized execution and virtual machine. Sections III, IV and V discuss the programming models, execution architectures and tools for these approaches. Further, each section presents a survey of the most important projects for each virtualization style. Finally, Section VI concludes the paper.

II. Virtualization Approaches

We identify three approaches of hardware virtualization that, although related, differ in their motivation. The approaches are denoted as *temporal partitioning*, *virtualized execution*, and *virtual machine*.

A. Temporal Partitioning

The motivation for this virtualization approach is to be able to map an application of arbitrary size to a reconfigurable device with insufficient hardware capacity. Temporal partitioning splits the application into smaller parts, each of which fits onto the device, and runs these parts sequentially. Temporal partitioning was the first virtualization style investigated. It was a necessity when reconfigurable devices were too small for many interesting applications, and is still of importance with todays multi-million gate FPGAs—in particular in embedded systems—to save area and thus cost.

B. Virtualized Execution

The motivation for virtualized execution is to achieve a certain level of device-independence within a device family. An application is mapped to or specified directly in a programming model. The programming model defines some atomic unit of computation. Such a unit is commonly called operator, actor, or—more generally—task. An application is then described by a collection of such tasks and their interactions. The execution architecture for such an application is defined as a whole family of devices. All devices support the abstractions defined by the programming model, i.e., the computations (tasks) and the interactions (communication channels). The members of a device family can differ in the amount of resources they provide, e. g., the number of tasks that can execute concurrently, or number of tasks that can be stored on-chip. Since all implementations of the execution architecture support the same programming model, an application can run on any member of the device family without recompilation. Task scheduling denotes the process of determining the exact execution order of the tasks. This order is partly given by the dependencies of the tasks, and partly induced by the resource constraints of the chosen device. This model of virtualized execution requires some form of a runtime system that resolves resource conflicts at runtime and schedules the tasks appropriately. The resulting independence of the device size allows to trade-off performance for cost. Furthermore, the forward compatibility lets us exploit advances in technology that result in larger and faster devices. This virtualization approach can be compared with defining the instruction set of a microprocessor. Compilers map application programs to binaries that can run on any processor instance executing the instruction set.

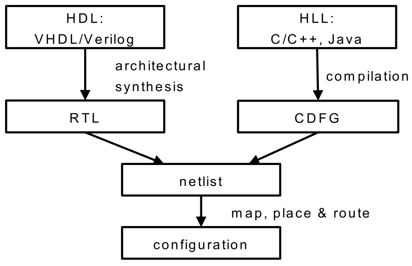

Fig. 1. Application models and tool flow

C. Virtual Machine

The motivation for this virtualization approach is to achieve an even higher level of device-independence. Instead of mapping an application directly to a specific architecture, the application is mapped to an abstract computing architecture. A hardware virtual machine is able to execute an application which has been mapped to such an abstract architecture. There are two basic ways to construct a hardware virtual machine. First, the application is remapped to the native application code of a concrete architecture. The virtual machine is then formally the tools for remapping. Second, the concrete architecture can run an "interpreter" that allows for direct execution of the abstract application code. This style of virtual hardware is analogous to the approach of platform-independent software that is used for instance in the Java virtual machine. Conceptually, a hardware virtual machine features platform-independent mobile hardware. This might be of increasing importance as most reconfigurable systems are connected to networks.

III. Temporal Partitioning

A. Application Models

The traditional way to specify an application for a reconfigurable device is using a hardware description language, e. g., VHDL or Verilog. Such a description is synthesized to an RTL description and, finally, to a netlist of combinational and sequential logic elements. Design implementation tools for reconfigurable devices take this netlist and perform technology mapping, placement, and routing to generate the configuration data for the reconfigurable device. This process is shown in Figure 1.

Alternatively, an application can be specified in a high-level programming language (HLL), e. g., C/C++ or Java. A compiler builds an internal representation of the program in the form of a control data-flow graph (CDFG). From this graph, RTL descriptions and netlists are generated. The results are further processed by the same tools as in the HDL-based tool flow. Both HDLs and HLLs are *implementation languages* in the sense that they define the computations performed by the application.

Temporal partitioning can be applied at different levels. The main approaches are shown in Figure 2: temporal partitioning at the level of netlists (see Section III-D.1), at the level of data-flow graphs (see Section III-D.2), and at the level of CDFGs (see Section III-D.3).

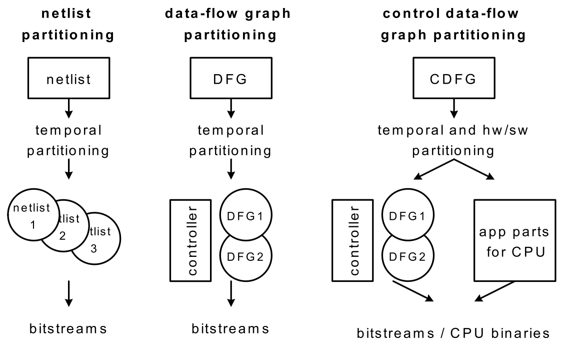

Fig. 2. Approaches for temporal partitioning: netlist partitioning, data-flow graph partitioning, and CDFG partitioning

Temporal partitioning on the netlist level operates on the netlists obtained by synthesis or compilation tools. Netlist partitioning is rather generic as it does not depend on the actual implementation language that was used for application specification. The main drawback of netlist partitioning is that one cannot make use of application-specific knowledge. In general, it is extremely difficult to regain information about the high-level structure or even dynamic behavior of the application from the netlist.

Data-flow graph partitioning works on operation or task graphs. Such graphs result from HDL and HLL design flows, but are also used as specification models in embedded systems design. An operation graph is a directed acyclic graph (DAG), where the nodes represent operations and the edges represent dependencies and communication.

Temporal partitioning at the CDFG level becomes possible if a high-level language is used for application specification. The main advantage is that CDFGs reveal information about the control-flow, e. g., loops, conditional execution, function calls, etc., which can be used to perform high-level optimizations and transformations. Common optimization techniques address runtime-intensive loops and increase the parallelism by unrolling, software pipelining, or pipelined vectorization.

B. Reconfigurable Architectures

Temporal partitioning decomposes an application into smaller parts and executes them sequentially. As this might require frequent reconfiguration, the efficiency of the approach is largely dependent on the ratio between a configuration's execution time and the device reconfiguration time. Conceptually, any *re*-configurable device can be used for temporal partitioning, e. g., todays dominating SRAM-based fine-grained FPGAs. However, the demand for low reconfiguration overheads favors advanced FPGA architectures that allow for fast reconfiguration, e. g., multi-context FPGAs.

All temporal partitioning approaches need a configuration controller. Usually, this control function is mapped to a processor. The resulting target architectures combine a processor with a reconfigurable device either in a single chip (as hybrid processor) or as a board-level system. Further, many approaches use an external memory to store the interconfiguration data.

C. Design Tools and Runtime System

The conventional tool flow needs only moderate modifications to support hardware virtualization with temporal partitioning. At some abstraction level in the design tool flow the application is split up into a number of smaller parts. Additionally, a configuration controller and circuitry for interconfiguration communication is generated. The reconfiguration controller implements the runtime system which executes a static configuration schedule. The controllers for interconfiguration communication store the output data of configurations and provide subsequent configurations with their input data. Each of the resulting application parts can be implemented with the same tools as in the non-virtualized case.

Irrespective of the chosen temporal partitioning approach, the resulting partitions are mapped, placed, and routed by the device-specific design implementation tools. Whenever the resulting configuration exceeds the device capacity, the temporal partitioning step must be iterated.

When the application is specified in a HLL, the compiler can perform hardware/software partitioning in addition to temporal partitioning. Only the runtime-intensive inner loops of an application are mapped to the reconfigurable device, while the rest of the application runs on the CPU.

D. Survey of Approaches

1) Temporal Partitioning of Netlists: Netlists are commonly represented by graphs. Temporal partitioning of netlists leads to graph partitioning problems which are, although having a different motivation, quite similar to structural partitioning problems. Structural partitioning of large netlists for execution on FPGAs has been studied in the context of FPGA-based emulation systems [1]. These methods take a large netlist that does not fit onto a single FPGA and determine a static partitioning to map the overall circuit onto a multi-FPGA emulation system. Each partition is required to fit onto a single FPGA while respecting the limited interconnect resources between the FPGAs. Hence, the used algorithms try to minimize the number of edges that cross the partitions. Temporal partitioning and time-multiplexed execution of partitions has not been considered for logic emulation, because the long reconfiguration times for conventional FPGAs would lead to excessive overheads.

With the advent of FPGAs that support fast reconfiguration, in particular multi-context FPGAs, temporal partitioning of netlists became realistic. Inspired by early multi-context FPGAs like DPGA [2] and TMFPGA [3], first studies looked at special cases of digital logic. Trimberger [4] and DeHon [5] discuss the mapping of FSMs in levelized logic form onto the multi-context architectures. The methods they propose are based on list scheduling.

Figure 3 presents the mapping of an FSM to a multi-context FPGA. The combinational state transition logic is partitioned into a number of configurations that form device contexts (e.g., configurations 1 and 2 in Figure 3). The state of the FSM is mapped in a separate configuration

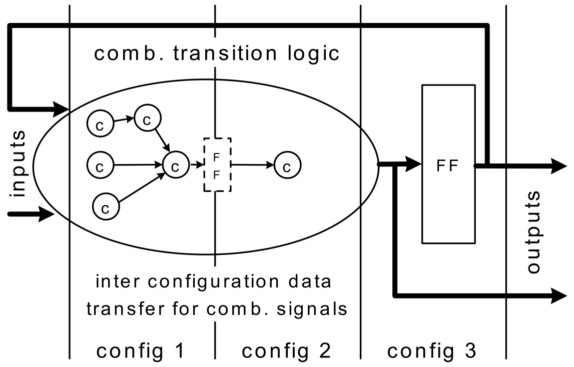

Fig. 3. Temporal partitioning of netlists (logic engine mode of TMFPGA, after [4])

(e.g., configuration 3 in Figure 3). The combinational signals crossing the configurations are held in a set of registers (or inter-context communication buffers) that are shared among all contexts. At runtime, the contexts are cyclically executed.

The difficulty of implementing FSMs with levelized logic is the high number of intermediate signals that might be required, which can lead to an excessive amount of registers. Therefore, several authors proposed improved heuristics that try to reduce the size of the inter-context communication buffers. A force-directed scheduling based method was studied by Chang and Marek-Sadowska [6][7]. Mak and Young [8] and Liu and Wong [9] used a network-flow based method. Wu et al. presented an exact ILP based algorithm [10].

2) Temporal Partitioning of Operation and Task-Graphs: The approaches in this group define temporal partitioning as an optimization problem over acyclic data-flow graphs. Similar to the netlist approach, the graph is partitioned into a number of configurations such that each single configuration does not exceed the device capacity. Further, there must exist at least one acyclic precedence relation between the configurations, i.e., the construction of a feasible configuration schedule must be possible. The primary optimization objective is the minimization of the overall execution time.

Purna and Bhatia proposed two greedy heuristics for this problem, the *level-based* and *cluster-based* partitioning algorithms [11] [12]. These methods do not try to minimize the overall execution time directly, but tend to minimize the number of configurations, the configuration's execution times, and the amount of interconfiguration data, respectively. Both methods construct configurations by adding node after node until the capacity of the device is reached. The level-based partitioning algorithm prefers to group nodes with the same ASAP (as soon as possible) level together, resulting in high degree of parallelism inside a configuration. This leads to short configuration runtimes. The cluster-based heuristic tries to group nodes with successor nodes to reduce the amount of interconfiguration communication.

An exact method to solve the temporal partitioning problem was presented by Vemuri et al. [13]. The authors used a 0–1 linear program that minimizes the overall execution time. Later, some authors extended the linear program to respect an additional constraint on the size of the memory for storing the interconfiguration data. The ILP formulation was also extended to cover other system level design steps, such as high-level synthesis [14] and design space exploration [15] [16].

An approach for temporal partitioning of task graphs to partially reconfigurable devices was discussed in [17] [18] by Fekete et al. The resulting problem was cast as 3D packing problem and an optimal branch and bound procedure was given to solve it. Temporal partitioning was also combined with the hardware/software partitioning problem by Chatha and Vemuri [19]. This approach combines a greedy heuristic partitioner with a heuristic list scheduler.

3) Temporal Partitioning of Control Data-Flow Graphs:

a) Garp: The Garp project [20] aims at creating a compiler that accelerates arbitrary applications written in C. The target architecture is the Garp chip that features a MIPS CPU core and a custom reconfigurable array co-processor [21]. The co-processor interface is used for configuration download and data-transfers. Additionally, the array can access the main memory itself via several memory controllers. A configuration cache allows to store 4 configurations on-chip.

The Garp C compiler (GarpCC) does not define a specific application model, but uses fully automatic compilation from C. GarpCC performs automatic hardware/software partitioning and maps application kernels to the reconfigurable array. The compiler uses similar techniques that VLIW compilers use for instruction scheduling, e. g., trace-scheduling. The GarpCC compiler pipelines loops and tries to find iterative schedules. Code is generated for both, the CPU core and the reconfigurable co-processor. The CDFGs that represent the code portions for the co-processor are processed by mapping, placing and routing tools and are transformed to a configuration bitstream.

Garp does not use a dedicated runtime system. Rather, the runtime system is implicitly integrated within the application that runs on the CPU core and on the array. The code for the CPU core includes the instructions for array control and configuration sequencing. The controllers that control the array execution are synthesized as part of the configuration.

b) XPP-VC: XPP-VC [22] is a vectorizing C compiler for the PACT XPP architecture. In contrast to compilers that try to find inner loops that fit directly onto the reconfigurable device, XPP-VC also uses temporal partitioning within a loop. After loop unrolling and data-dependence analysis, the compiler applies pipeline vectorization [23] to inner loops. Pipeline vectorization tries to overlap loop iterations and to execute the loop in a pipelined fashion. The vectorized loop is transformed to XPP's native netlist format. If the vectorized loop does not fit onto the device, an iterative process consisting of temporal partitioning and design implementation (map, place, route) is started.

The XPP device implements a dedicated programmable unit (configuration manager) that executes the runtime system for a temporally partitioned execution. The configuration manager autonomously sequences and loads the configurations. A configuration cache is used to keep the most recently used configurations on-chip.

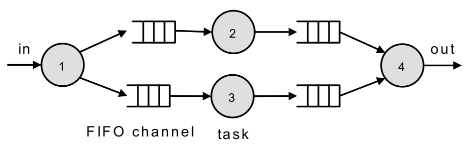

Fig. 4. Application specification with a process network

IV. VIRTUALIZED EXECUTION

A. Application Models

The use of *coordination languages* (or formalisms) for application modeling is widespread in embedded systems design. Coordination languages are semantically well-defined and usually more restricted than HLLs. Hence, they often allow for formal analysis of system properties such as the absence of deadlocks. Coordination languages are often tailored to specific application domains, e. g., SDF for signal-processing applications, or Petri-Nets for communication protocols.

In the majority of cases, coordination languages are used in combination with implementation languages. Large applications are decomposed into smaller execution objects (tasks). The interaction of these execution objects is specified using the coordination language, while the functionalities of the tasks are defined with an *implementation language*. This style of application modeling leads naturally to virtualized execution. Given a suitable device and a task scheduler, the application can be executed directly on the architecture, as the sequence and interaction of the tasks is defined by the coordination language.

Figure 4 shows an example for an application specified in a coordination formalism named *process network*. The tasks in the process network are connected via FIFO channels of unlimited capacity. A task can run when input data is available on all of its input FIFOs. Process networks model concurrency; thus tasks 2 and 3 in Figure 4 may execute at the same time.

B. Reconfigurable Architectures

There are two basic architectural approaches to support virtualized execution. In the first approach, the reconfigurable architecture directly supports the programming model. As an application is decomposed into a number of interacting tasks, the architecture needs to implement the logic resources to execute one or several tasks concurrently and the communication channels to implement the interaction between the tasks. Such an architecture is composed out of atomic units (often called hardware pages) that can accommodate one task.

The second approach relies on classical reconfigurable devices and assigns the reconfigurable resources to the tasks at runtime. A task is assigned an amount of resources that matches its demand, rather than a fixed-size hardware page. While such a scheme complicates the runtime system, it leads to improved device utilization.

C. Design Tools and Runtime System

The decomposition of an application into communicating operators, according to the chosen coordination language,

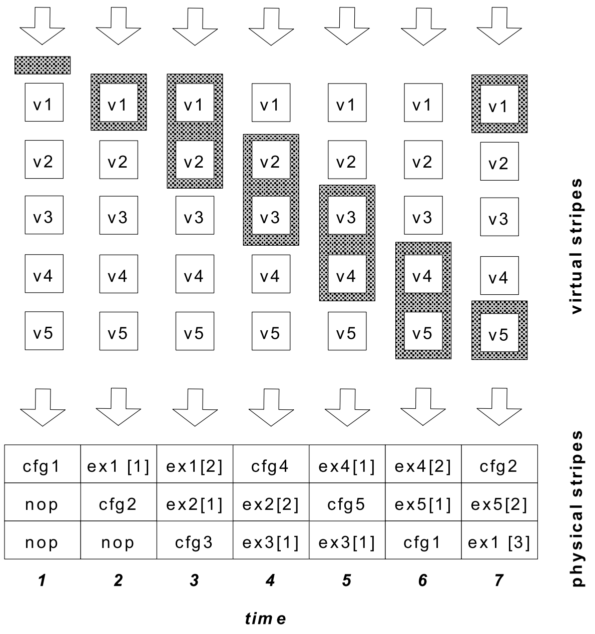
v1
v2
v3
v4
v5
virtual stripes
cfg1 ex1 [1] ex1[2] cfg4 ex4[1] ex4[2] cfg2
nop cfg2 ex2[1] ex2[2] cfg5 ex5[1] ex5[2]
nop nop cfg3 ex3[1] ex3[1] cfg1 ex1 [3]
1 2 3 4 5 6 7
time
physical stripes

Fig. 5. PipeRench: pipelined processing and pipelined reconfiguration

can either be done by the programmer or by design tools. The resulting operators are implemented by conventional synthesis and design implementation tools.

Virtualized execution requires a runtime system. For architectures with fixed-size hardware pages, the runtime system schedules the tasks according to the semantics of the coordination language and the available hardware resources. Additionally to scheduling, the runtime system has to perform allocation of resources such as communication channels, buffers and I/O ports. Although computing a static schedule is possible in general, it would diminish the benefits of device-independence. Architectures with variable-sized resource assignment require a more complex runtime system. Before a task can be scheduled, a feasible placement on the reconfigurable device must be found.

D. Survey of Approaches

c) PipeRench: The PipeRench architecture [24] focuses on the pipelined processing of data-streams. The programming model for PipeRench is to decompose the application into a sequence of pipelined operators, called stripes. PipeRench achieves hardware virtualization by allowing an application to use an unlimited number of virtual stripes. If the number of physically available stripes is smaller than the number of virtual stripes required by an application, the configurations for the stripes are loaded at runtime. PipeRench is restricted to forward pipelining of a fixed sequence of operators. Feedback is supported only within a stripe. Nonetheless, many important streaming applications can be mapped to this architecture.

A key feature of PipeRench is—additionally to pipelined data-processing—*pipelined reconfiguration*. Figure 5 shows the execution of a streaming application that requires 5 virtual stripes on a device with 3 physical stripes. The upper part of the figure shows the application specified in virtual stripes v1-v5. The shaded box moving from virtual stripe v1 to v5 shows which virtual stripes are mapped to physical stripes in a given time step. Two physical stripes are executing concurrently in each time step, while the third stripe is being reconfigured. In time steps 2 and 3 an input value is read into the array. The first output is produced by stripe v5 at time 6. The second output is produced at time 7, when also the next input value is read. The lower part of the figure shows the function of the physical stripes for each time step: cfgX means loading configuration X, exX[i] means executing configuration X on sample i.

A custom reconfigurable device that implements the PipeRench execution model was presented in [25]. The device features 16 physical stripes and on-chip configuration memory for 256 virtual stripes. The runtime system for PipeRench is implemented directly on the PipeRench chip.

PipeRench uses a compiler that takes C programs as input language. After inlining all functions, unrolling all loops and converting to single-assignment form, the compiler transforms the program to an intermediate data-flow language. Bit-widths are infered and complex operators are decomposed into simpler operators.

d) SCORE: Caspi et al. [26] developed the SCORE model that provides both, a specification model and a virtualized execution model for the important class of streaming applications. An application is defined as a graph of computation nodes (operators) that are connected by FIFOs of unbounded size. The coordination of the operators is defined by the data dependencies in the execution graph. The function of the operators is specified in the TDF language, an RTL language with C-like notation. The memory used by the operators is allocated in fixed-size memory pages.

So far there is no physical device that implements the SCORE application model. Thus SCORE was evaluated by simulation only. However, a hybrid execution architecture was defined that comprises a CPU and a reconfigurable array. The reconfigurable array consists of a number of equivalent and independent compute pages (that implement the operators), configurable memory blocks, and buffered interconnect.

The runtime system of a SCORE system is supposed to run on the CPU and consists of the instantiation engine and the scheduling engine. The instantiation engine interprets the compute graph and instructs the scheduler which tasks are to be scheduled. The scheduling engine takes care of resource allocation, placement of operators to compute pages and routing.

e) WASMII / DRL: The WASMII [27] is a multi-context reconfigurable device that is specifically tailored to virtual hardware execution. The WASMII architecture is data-driven and allows for execution of data-flow graphs. The application is first mapped to a data-flow graph, which is then decomposed into sub-graphs that fit the size of a page. A page is the basic computation unit of the device. The WASMII architecture also supports the connection of several WASMII devices to build a multichip WASMII architecture. The sequencing of pages is controlled by a static schedule that is generated at compile-time with the LS-M algorithm [28].

f) Reconfigurable HW Operating Systems: Reconfigurable hardware operating systems treat reconfigurable devices as dynamic resources that are managed at runtime. Similar to software operating systems, these approaches introduce tasks or threads as basic units of computation and provide various communication and synchronization mechanisms. The runtime system places and schedules tasks on the reconfigurable device in a multitasking manner. Such an operating system provides a minimal programming model, although being less restrictive than coordination languages.

The first description of hardware multitasking is due to Brebner [29]. More recently, Wigley et al. discussed operating system functions including device partitioning, placement and routing [30]. Multitasking and task preemption was investigated in [31], and [32], respectively. Scheduling and placement techniques were devised in [33] [34]. Functional prototypes that demonstrate multitasking on todays FPGA technology were also described, e. g., in [35] [36].

V. Virtual Machine

Hardware virtualization with the *virtual machine* approach requires neither a specific application model nor a specific reconfigurable architecture.

Application specification and synthesis can be performed with conventional tools, aside from the fact that the targeted technology is an abstract architecture. Mapping to an abstract architecture can easily be achieved by using generic synthesis and technology mapping libraries. The definition of the abstract architecture is key to the virtual machine approach. One one hand, the abstract architecture has to be generic enough to allow for an efficient remapping to different targets. On the other hand, the abstract architecture must be close to typical FPGA architectures to leverage their performance potential. At loading time or runtime, the abstract description is remapped to the actual architecture by a virtual machine. As the abstract architecture is a generalization of the actual architecture, the remapping involves running parts of the conventional tool flow, such as technology mapping, and place and route [37].

A. Survey of Approaches

Issues of circuit portability in a networked environment were first addressed by Brebner [38]. He coined the term *circlets (circuits + applets)* to denote mobile circuits. There is, however, no virtual hardware machine running on the target. The concept described circlets as location-independent circuits that are pre-synthesized, pre-placed and pre-routed to a specific technology. Circlets are mapped to the target FPGA by a runtime system, which is more in the line of reconfigurable hardware operating systems (see Section IV-D.0.f).

Ha et al. [37] proposed a virtual hardware machine that executes hardware bytecode. The hardware bytecode for a circuit is essentially a technology-mapped, placed, and routed netlist for an abstract FPGA architecture. The abstract FPGA architecture proposed in [37] is a generic fine-grained FPGA with symmetrical routing. The logic blocks are based on 4-LUT elements followed by a register that can be bypassed. The routing is structured into channels running in vertical and horizontal directions. The virtual machine has to perform a number of steps on the hardware bytecode to translate it to the target FPGA. These steps include the remapping of logic blocks and I/O pins as well as global and detailed routing. Temporal partitioning in case the hardware bytecode exceeds the FPGA capacity was not considered. Rather, a runtime system was proposed that places and executes the final circuit on the FPGA.

In [39], a framework was described that adopts the virtual hardware machine approach to a hybrid, networked target system consisting of a CPU and an FPGA. At design time, a hardware/software codesign environment partitions an application into software and hardware functions and generates the hardware and software bytecodes. A service bytecode binder combines all bytecodes into one file that is transferred to the target.

VI. Conclusion

Virtualization of hardware is a rather new research area. Up to date, three main approaches have emerged. Each approach has its own challenges concerning design tools and runtime systems, and sometimes also device architectures.

Temporal partitioning was the first virtualization approach, applied to netlists and operation graphs. In the meantime, the main focus there has shifted to compilation from HLLs. We see the main role for temporal partitioning in embedded systems, where saving chip area and thus cost is an important optimization goal. Virtualized execution comes in two flavors: approaches that compile to architectures with a fixed-size basic hardware element and approaches that map the atomic operator to variable-sized hardware elements at runtime. While the variable-sized approaches lead to a potentially higher device utilization, the algorithmic problems involved are hard and their solutions time-consuming. The virtual machine approach is the newest one. Although conceptually the most powerful virtualization approach, the practical realization has yet to be shown. Remapping circuit descriptions at the target involves complex design tools and requires significant runtime. Many target systems, especially networked embedded systems, might just be not powerful enough for that.

Reconfigurable devices deliver their peak performance when a maximum on low-level parallelism and specific device features are used. A virtualization technique that sacrifices too much of this performance for the sake of device-independence will most likely not be accepted.

References

[1] M. Butts, “Tutorial: FPGAs in logic emulation,” in *Proc. IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD)*. ACM, Nov. 1993.

[2] E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon, “A first generation DPGA implementation,” in *Canadian Workshop on Field-Programmable Devices (FPD)*, 1995, pp. 138–143.

[3] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, “A time-multiplexed FPGA,” in *Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 1997, pp. 22–28.

[4] S. Trimberger, “Scheduling designs into a time-multiplexed FPGA,” in *Proc. 6th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*. ACM, 1998, pp. 153–159.

[5] A. DeHon, “DPGA utilization and application,” in *Proc. 4th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 1996, pp. 115–121.

[6] D. Chang and M. Marek-Sadowska, “Buffer minimization and time-multiplexed I/O on dynamically reconfigurable FPGAs,” in *Proc. 5th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*. ACM, 1997, pp. 142–148.

[7] D. Chang and M. Marek-Sadowska, “Partitioning sequential circuits on dynamically reconfigurable FPGAs,” *IEEE Trans. on Computers*, vol. 48, no. 6, pp. 565–578, June 1999.

[8] W.-K. Mak and E. F. Young, “Temporal logic replication for dynamically reconfigurable FPGA partitioning,” in *Proc. ACM Int. Symp. on Physical Design (ISPD)*. ACM, Apr. 2002, pp. 190–195.

[9] H. Liu and D. Wong, “Network flow based circuit partitioning for time-multiplexed FPGAs,” in *Proc. IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD)*. ACM, Nov. 1998, pp. 497–504.

[10] G.-M. Wu, J.-M. Lin, and Y.-W. Chang, “Generic ILP-based approaches for time-multiplexed FPGA partitioning,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 10, pp. 1266–1274, Oct. 2001.

[11] K. M. G. Purna and D. Bhatia, “Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers,” *IEEE Trans. on Computers*, vol. 48, no. 6, pp. 579–590, June 1999.

[12] K. Purna and D. Bhatia, “Temporal Partitioning and Scheduling for Reconfigurable Computing,” in *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, 1998, pp. 329–330.

[13] S. Govindarajan, I. Ouaiss, M. Kaul, V. Srinivasan, and R. Vemuri, “An Effective Design System for Dynamically Reconfigurable Architectures,” in *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, 1998, pp. 312–313.

[14] M. Kaul and R. Vemuri, “Optimal Temporal Partitioning and Synthesis for Reconfigurable Architectures,” in *Proc. Design, Automation and Test in Europe Conf. (DATE)*. IEEE Computer Society, 1998, pp. 389–396.

[15] V. Srinivasan and R. Vemuri, “Task-level Partitioning and RTL Design Space Exploration for Multi-FPGA Architectures,” in *Proc. 7th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, 1999, pp. 272–273.

[16] V. Srinivasan, S. Govindarajan, and R. Vemuri, “Fine-Grained and Coarse-Grained Behavioral Partitioning With Effective Utilization of Memory and Design Space Exploration for Multi-FPGA Architectures,” *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 1, pp. 140–158, Feb. 2001.

[17] S. Fekete, E. Köhler, and J. Teich, “Optimal FPGA Module Placement with Temporal Precedence Constraints,” in *Proc. Design, Automation and Test in Europe Conf. (DATE)*. IEEE Computer Society, 2001, pp. 658–665.

[18] J. Teich, S. Fekete, and J. Schepers, “Optimization of Dynamic Hardware Reconfigurations,” *Journal of Supercomputing*, vol. 19, no. 1, pp. 57–75, May 2001.

[19] K. Chatka and R. Vemuri, “Hardware-Software Codesign for Dynamically Reconfigurable Architectures,” in *Field-Programmable Logic and Applications (Proc. FPL)*. Springer, 1999, pp. 175–184.

[20] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, “The Garp architecture and C compiler,” *IEEE Computer*, vol. 33, no. 4, pp. 62–69, Apr. 2000.

[21] J. R. Hauser and J. Wawrzynek, “Garp: A MIPS processor with a reconfigurable coprocessor,” in *Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 1997, pp. 12–21.

[22] J. M. Cardoso and M. Weinhardt, “XPP-VC: a c compiler with temporal partitioning for the PACT-XPP architecture,” in *Field-Programmable Logic and Applications (Proc. FPL)*, ser. LNCS, vol. 2438. LNCS 2438, Springer Verlag, 2002, pp. 864–874.

[23] M. Weinhardt and W. Luk, “Pipeline vectorization,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 234–248, Feb. 2001.

[24] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, “PipeRench: A reconfigurable architecture and compiler,” *IEEE Computer*, vol. 33, no. 4, pp. 70–77, Apr. 2000.

[25] H. Schmit, D. Whelihan, M. Moe, B. Levine, and R. R. Taylor, “PipeRech: A virtualized programmable datapath in 0.18 micron technology,” in *Proc. 24th IEEE Custom Integrated Circuits Conf. (CICC)*, 2002, pp. 63–66.

[26] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon, “Stream computations organized for reconfigurable execution (SCORE),” in *Field-Programmable Logic and Applications (Proc. FPL)*. LNCS 1896, Springer-Verlag, 2000, pp. 605–614.

[27] Xiaoping Ling and H. Amano, “WASMII: An MPLD with data-driven control on a virtual hardware,” *The Journal of Supercomputing*, vol. 9, no. 3, pp. 253–276, 1995.

[28] X.-P. Ling and H. Amano, “A static scheduling system for a parallel machine $(SM)^2$-II,” in *Proc. 2nd Parallel Architectures and Languages, Europe*, ser. LNCS 365. Springer, June 1989, pp. 118–135.

[29] G. Brebner, “A Virtual Hardware Operating System for the Xilinx XC6200,” in *Proc. 6th Int’l Workshop on Field-Programmable Logic and Applications (FPL)*, 1996, pp. 327–336.

[30] G. Wigley and D. Kearney, “The Development of an Operating System for Reconfigurable Computing,” in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. IEEE CS Press, April 2001.

[31] H. Simmler, L. Levinson, and R. Männer, “Multitasking on FPGA Coprocessors,” in *Proceedings of the 10th International Workshop on Field Programmable Gate Arrays (FPL)*. Springer, 2000, pp. 121–130.

[32] G. Brebner and O. Diessel, “Chip-Based Reconfigurable Task Management,” in *Proceedings of the 11th International Workshop on Field Programmable Gate Arrays (FPL)*. Springer, 2001, pp. 182–191.

[33] H. Walder and M. Platzner, “Online Scheduling for Block-partitioned Reconfigurable Devices,” in *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*. IEEE Computer Society, March 2003, pp. 290–295.

[34] C. Steiger, H. Walder, M. Platzner, and L. Thiele, “Online Scheduling and Placement of Real-time Tasks to Partially Reconfigurable Devices,” in *Proceedings of the 24th International Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, December 2003, pp. 224–235.

[35] “IMEC Interuniversity Micro Electronic Center, T-ReCS Gecko: Hardware / Softwar e multitasking on a reconfigurable platform, `http://www.imec.be`.”

[36] 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.

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

[38] G. Brebner, “Circlets: Circuits as Applets,” in *Proc. 6th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, 1998, pp. 300–301.

[39] Y. Ha, S. Vernalde, P. Schaumont, M. Engels, R. Lauwereins, and H. De Man, “Building a Virtual Framework for Networked Reconfigurable Hardware and Software Objects,” *Journal of Supercomputing*, vol. 21, no. 2, pp. 131–144, February 2002.