A Communication Architecture for Complex Runtime Reconfigurable Systems and its Implementation on Spartan-3 FPGAs

Dirk Koch, Christian Beckhoff and Jürgen Teich
University of Erlangen-Nuremberg
Am Weichselgarten 3
D-91058 Erlangen, Germany
{dirk, christian}@recobus.de, teich@cs.fau.de

ABSTRACT

In this paper, we present and analyze a sophisticated communication architecture that allows to integrate many different modules into a system by FPGA reconfiguration at runtime. Furthermore, we examine how this architecture can be implemented on low-cost Spartan-3 devices. It will be demonstrated that modules can be exchanged in a system without disturbing the communication architecture. The paper points out, that the capabilities of Spartan-3 FPGAs are sufficient to build complex reconfigurable systems.

1. INTRODUCTION

Supporting Spartan-3 FPGAs for building dynamically reconfigurable systems is important as these FPGAs provide significant more logic per monetary cost as compared to devices providing glitch-less partial reconfiguration modes (e.g., all Xilinx Virtex FPGAs). As saving monetary cost is one major goal for using partial runtime reconfiguration, it is obvious to apply advanced partial reconfiguration techniques also to these devices.

With the ReCoBus architecture, we developed a sophisticated communication architecture that allows to build complex reconfigurable systems with many runtime reconfigurable modules. Based on an analyzes of this architecture in Section 3, we will focus on the implementation of this architecture on the low cost Spartan-3-FPGA architecture in Section 4. The goal is to Implement the ReCoBus architecture in such a way that modules can be exchanged in a system at runtime without influence to other modules connected to a ReCoBus. We classify disturbances due to a reconfiguration process into three categories:

1. *Glitch-less reconfiguration*: No influence to other modules or bus transactions if modules are exchanged.
2. *Glitch reconfiguration*: A bus transfer may be disturbed within one clock cycle
3. *Interrupted reconfiguration*: The bus is disturbed for multiple clock cycles

In almost all cases when the Xilinx documentation reports about glitches during runtime reconfiguration, this is not only a short term event but an interruption or disturbance for many hundreds of configuration clock cycles. The reason for this behavior is that Spartan-3 are *blockwise reconfigurable FPGAs* that requires a clear operation to the reconfigurable area prior to the particular configuration data write operation. After the clear operation, a new configuration can be written in a frame by frame manner. A configuration frame is the smallest atomic piece of configuration data that stores a fraction of the configuration data of a complete CLB column. A CLB is a configurable logic block that clusters 8 four bit look-up tables and the according routing fabric in one cell. In the case of Spartan-3 devices, it requires to write 19 complete frames to configure a complete CLB column. The delete operation is performed by sending a so called *snowplow* command to the Spartan-3 FPGA. With this command, the configuration data of one complete column of CLBs will be reset to a logical '`0`' value.

2. RELATED WORK

Systems using FPGA resources in a time variant manner by exploiting partial reconfiguration have been presented in various academic publications. The main issue of this work is the interfacing of runtime reconfigurable hardware accelerators to on-chip buses (OCBs) and to dedicated I/O channels on low cost FPGAs without glitches. Existing approaches for the on-chip communication of partially reconfigurable modules are based on i) *circuit switching* techniques, ii) *packet switching* mechanisms, and iii) on-chip *buses* and will be outlined in the following.

Circuit Switching
Circuit switching is a technique where physically wired links are established between two or more modules for a certain amount of time. Typically, these links are implemented by more or less complex crossbar switches. The switching state of the switches may be controlled centralized by the static part of the system [17] or distributed by some logic in the crossbar switches itself with respect to the currently used routing resources within the switches [5, 2]. All these approaches prevent any partially reconfigurable module from directly accessing any memory of the system. In addition, the large multiplexers required for circuit switching allow

only a very coarse-grained placement of modules. The large multiplexers have also significant propagation delays leading to decreased throughputs when using circuit switching.

For all proposed systems using circuit switching, we found that the switching state for the multiplexers is only changed when a module is deleted or loaded to the FPGA. As a consequence, there is no switching activity in the crossbar switches during the operation of the modules. Therefore the crossbar switches should be implemented directly within the routing fabric of the FPGA without using additional LUTs for implementing the multiplexers.

Packet Switching
The motivation for packet switching comes from the ASIC domain where more and more functional units were integrated. Here, networks on a chip (NoC) [3] have been proposed in order to deal with multiple clock domains, modularity for IP-reuse, and in order to support parallelism in communication and computation This was the motivation in [1] to integrate a dynamically reconfigurable network on an FPGA called a *DyNoC*. Here, a grid of routers is used for the communication among the reconfigurable modules. Each router decides locally based on the destination address where to send a packet further and the routers are capable to deal with obstacles. However, the approach in [1] demands a relatively fine grid of routers while the presented implementation results revealed that the logic for a single router requires several hundreds or even thousands of look-up tables (depending on the supported packet sizes).

We can conclude that a NoC for FPGAs makes only sense if the routers are implemented as dedicated primitives within the FPGA architecture. This would allow a reasonable area efficiency and a high transfer bandwidth at the same time.

Buses
The most common way for linking together communicating modules within an SoC is to use buses. Consequently, all major FPGA vendors offer tools that allow easily integrating a set of user-defined modules or IP cores into complete systems by the use of on-chip buses. Therefore, partially reconfigurable modules should also be able to directly connect themselves via a bus into a system at runtime. The first publications done in this field that are based on older Xilinx FPGAs utilize tristate wires spanning over the complete horizontal device width [18, 14, 10, 4, 16]. However, tristate buses come along with some place and route restrictions and require timing parameters that must be met to turn buffers on-and-off. This leads typically to lower clock speeds as compared to multiplexer-based buses. Consequently, most established bus standards including AVALON, CoreConnect, and Wishbone provide multiplexer-based bus implementations with unidirectional wires and all newer FPGA architectures possess no more internal tristate drivers.

In order to permit to exchange modules by partial reconfiguration at runtime, the communication resources for linking the partial must be bound to fixed positions for all partial modules. In the case of busses, it is advantageous to implement the communication infrastructure for partially reconfigurable modules in a completely homogenous manner for all possible module positions. This allows modules to be exchanged and relocated to different positions on the chip. In the case of a bus, this means that the communication architecture should be constructed in a regular fashion with regular tiles, each having exactly the same internal logic and routing layout for providing the connectivity among the tiles and to the modules.

In [18] and [13], systems are proposed where fixed resource areas with also fixed connection points to a bus interface are used for the integration of partially reconfigurable modules into a runtime system. Here, all signals if the communication architecture are separated from the modules and dedicated communication primitives bound to fixed positions of the FPGA provide a bridge connection to the bus logic that is located in the static part of the system. A drawback of this solution is that all modules have to fit into a resource area of the same size and the resource areas could not be shared by multiple modules, even if the logic of multiple modules would fit into the same resource area. A suitable bus infrastructure for integrating reconfigurable modules should be able to connect modules of different sizes efficiently to the system.

Hagemeyer et al. [6] presents a more flexible approach that uses on-chip tristate drivers or alternatively distributed multiplexers in order to build buses for reconfigurable systems with a much higher count of module tiles. In this work, the complete bus based communication architecture is arranged directly within the tiles allocated for reconfigurable modules and modules may use multiple tiles to implement their logic. However, this high flexibility comes along with an enormous resource overhead for the communication architecture preventing this approach to take a significant benefit of runtime reconfiguration in many cases.

Runtime Reconfiguration on Spartan-3 Devices
A lot of recent work on runtime reconfiguration on Xilinx Spartan-3 devices deals with the fact that some members of the Spartan-3 Family do not possess an internal access to the FPGA reconfiguration port. Thus, for allowing self-reconfiguration, these devices need an external feedback path for sending the configuration data to the FPGA fabric via external pins.

The implementation of complex systems with many reconfigurable modules was not examined for Spartan-3 FPGAs. And so far, no sophisticated communication architecture has been presented for these devices that masks the effects of the snowplow command to the rest of the system.

3. THE RECOBUS ARCHITECTURE

The ReCoBus architecture [12] was developed to provide a sophisticated communication infrastructure for enabling a new dimension in modular FPGA design. The goal of the ReCoBus approach is to link together configuration bitstreams or completely routed netlists based on a predefined communication architecture. This is similar to the system integration that is based on backplane buses where cards are plugged together into bus sockets. In the case of FPGA-based systems, this allows to exchange a module without any influence to the rest of the system. Modules are placed in one or more *resource slots* of the FPGA. A resource slot is the smallest atomic piece of FPGA area that can be allocated by a module while a module may occupy multiple resource slots for implementing its logic. In this sense, a resource slot has some similarities with a sector on a harddrive. A sector is the smallest junk of memory that is directly accessible and a file may occupy multiple sectors for storing its data.

The size of a resource slot should be small for reducing the effect of internal fragmentation that arises when modules have to be fit into resource slots boundaries. On the other side, a high amount of resource slots may lead to a logic overhead or a timing penalty for linking a module within a resource slot to other modules, some I/O pins, or the static part of the system. The ReCoBus architecture optimizes the granularity (size of the resource slots), the logic overhead, and the latency of the communication architecture at the same time. The most important properties of the ReCoBus architecture are:

i) *Direct interfacing*: Reconfigurable modules have a direct interface to the on-chip bus (OCB) and to other modules or I/O pins.

ii) *Module relocation*: Instead of binding modules to a fixed reconfigurable area, they can be placed freely within the span of the reconfigurable OCB.

iii) *Flexible widths*: Modules can be integrated into the system with a *fine resource slot grid*.

iv) *Multiple instances*: Modules can be instantiated a couple of times and connected to the system.

v) *Low logic overhead* to implement the communication architecture.

vi) *High performance*: The communication bandwidth and the latency of the reconfigurable bus can compete with traditional static only systems.

The ReCoBus communication architecture is designed to have a completely uniformed layout of the logic and routing resources among all resource slots of the same type. In the case of Spartan-3 FPGAs, we only have to distinguish if a resource slot contains dedicated block ram resources or not.

One basic concept of the ReCoBus architecture is that the module interface can grow with the module complexity. This follows the observation that in typical systems a complex (or larger) module has a wider interface than a simple one. For instance, while for a simple UART an 8 bit connection to the system bus may be sufficient, an Ethernet core will typically demand a 32 bit interface.

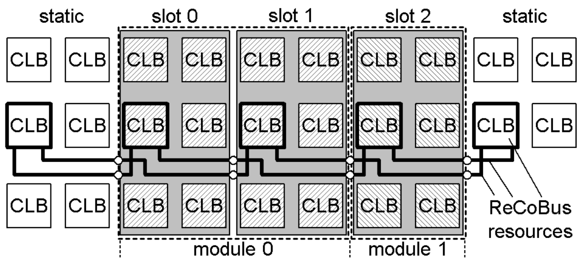

Figure 1: Example of a system providing three resource slots $R = 3$, each being two CLBs wide $W = 2$ that are currently running two modules $M = 2$. The fat boxes indicate CLBs used for implementing the ReCoBus architecture. The resource slots are twofold interleaved $N = 2$.

The flexible interface width is achieved by interleaving N multiple independent connecting chains that are connected once per N resource slots, as illustrated in Figure 1. As shown in the figure, this still allows to build the complete communication architecture in a completely homogeneous manner which permits to relocate modules to different resource slots. This can be used for building static only systems as well as for providing a flexible placement for runtime reconfigurable systems.

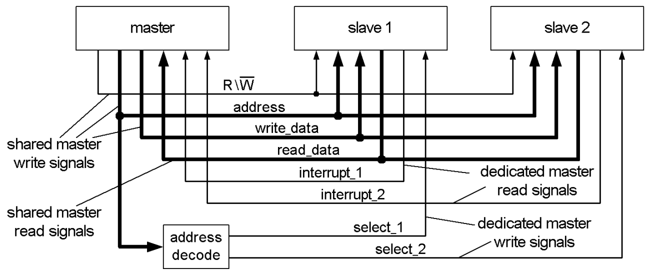
master
slave 1
slave 2
R\W̄
address
write_data
read_data
interrupt_1
interrupt_2
select_1
select_2
address decode
shared master write signals
shared master read signals
dedicated master read signals
dedicated master write signals

Figure 2: Examples for the four signal classes allowing to implement any bus protocol: shared read, shared write, dedicated read, and dedicated write.

Furthermore, the interleaving reduces the depth of the combinatorial path of the ReCoBus architecture while still allowing a fine resource slot grid. For Xilinx Spartan-3 FPGAs this means that a resource slot is only one or a few CLB columns wide. We declare the granularity as the width w of a resource slot in terms of CLB columns.

The ReCoBus architecture provides a physical implementation of the system communication infrastructure. Consequently, it is not limiting the protocols that are used for the communication. We found that all FPGA on-chip buses can be implemented by the use of the following four signal classes: 1) shared read, 2) shared write, 3) dedicated read, and 4) dedicated write. Examples for all signal classes are given in Figure 2 that shows a simple master slave system. If we want to implement a multi master system, the bus request from the master to the arbiter would represent the same problem as to link an interrupt request to the master, thus, it is also implemented as a dedicated read signal. Respectively, the grant signal from the arbiter has to be implemented as a dedicated write signal.

In the following, we will analyze the most common implementations of these signal classes more detailed with respect to a homogeneous layout on an FPGA.

Shared Read Signals

Shared read signals have multiple sources among the resource slots and one output in the static part of the system. An example for this signal class is the read data signal from several slaves located in some resource slots that is linked to a master in the static part. In common static only systems, this signal class is implemented with wide multiplexers. As revealed in Figure 3, such multiplexers can be implemented in a completely regular fashion among a set of resource slots.

In this example, the amount of look-up tables required to build a regular structured read multiplexer chain for R resource slots and S_{SR} shared read signals is $R \cdot S_{SR}$ LUTs. If a module will never be exchanged by reconfiguration at runtime, we can remove unused LUTs from the read multiplexer chains and the amount of required LUTs is simply the sum of all used shared read outputs from all modules placed in the resource slots that will not be exchanged at runtime. Removing LUTs from a read multiplexer chain can also be used for runtime reconfigurable modules but would lead to

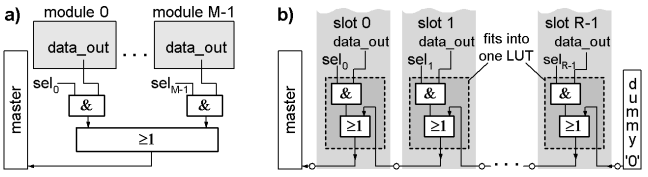

Figure 3: a) Common implementation of a shared read signal in a multiplexer-based bus. b) Homogeneous distributed read multiplexer chain.

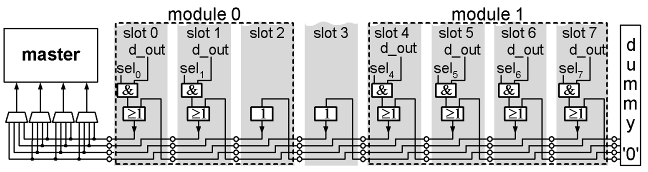

Figure 4: System with N=4 interleaved read multiplexer chains and additional alignment multiplexers.

an interruption of the chain during the reconfiguration of such a module.

We can compare the cost of a regular read multiplexer chain with a static and heterogeneous implementation of a multiplexer. In the best case, the number of lookup tables is the minimum resource requirements for all multiplexers over each individual shared read bit signal. The minimum look-up table requirements for implementing specific multiplexers on Xilinx Spartan-3 FPGAs have been determined by using the Xilinx ISE 8.2 suite. The results are listed in Table 1. If we assume an example with four 8-bit wide modules and three 32 bit wide modules, it requires 8 7-input multiplexers and 24 3-input multiplexers that need $8 \cdot 4 + 24 \cdot 2 = 80$ LUTs for the implementation.

Table 1: Relationship between the look-up table count and the input width of a multiplexer implemented on a Spartan 3 FPGA.

Mux inputs	2	3	4	5	6	7	8	9	10	11	12
4-bit LUTs	1	2	2	3	4	4	4	5	5	6	6

However, as the modules are spread somehow over the FPGA, these multiplexers are typically not implemented in the most resource efficient way in order to enhance the speed of the multiplexers. In typical systems, the amount of look-up tables is in the range of what a homogeneous read multiplexer chain would require when unused connections are removed from the chain.

If a read multiplexer chain is used in runtime reconfigurable environments that support a completely glitch-less exchanging of modules, all LUTs of a read multiplexer chain that are not connecting any output signals are producing an overhead. This overhead can be extensive if the system provides high placement flexibility for many reconfigurable modules of different size. As mentioned earlier in this section, the ReCoBus architecture may interleave bus signals for reducing this overhead in systems containing many small resource slots. This is illustrated in Figure 4 for a simple system with two modules being connected to a master in the static part of the system. Let us assume that each of the $N = 4$ interleaved read multiplexer chains represents 8 individual bit signals. Then, the first module (module 0) has a 16-bit interface and the second module has access to all $N \cdot 8 = 32$ bits of the bus. In order to allow a free module placement to the resource slots of the system, we have to put an alignment multiplexer between the read multiplexer chains and the component located in the static part of the system. This alignment multiplexer adjusts the order of the different chains depending on the placement position. This multiplexer is controlled by a small configurable look-up table that is connected to the address selecting a particular module within the slot.

The amount of look-up tables required to implement S_{SR} shared read signals for R resource slots that are N times interleaved is:

$$L_{SR} = S_{SR} \cdot MUX(N) + S_{SR} + \left\lceil \frac{S_{SR}}{N} \right\rceil \cdot R. \qquad (1)$$

With $MUX(N)$ being the amount of LUTs required to implement an N-input multiplexer (see Table 1). The additional amount of S_{SR} LUTs is required to built the routing towards the static part of the system in the same homogenous fashion as it is between the resource slots. The example in Figure 4 with $N = 4$ interleaved chains with an assumed width of 8 bit per chain requires $32 \cdot 2 + 32 + 8 \cdot 8 = 160$ LUTs. The depth of the combinatory path is in the example five look-up table levels deep: two for the read multiplexer chains, two for the alignment multiplexer, and the additional LUT providing the homogeneous routing layout to the static system.

Shared Write Signals

As a counterpart to shared read signals, a bus has to provide some signals in backward direction for driving data from the static part of the system to all resource slots. An example would be the write data or the address from a master that is located in the static region of the system. We are able to share resources for implementing shared write signals with the resources used for implementing shared read signals. This is possible as Xilinx FPGAs permit to use the flip-flop and the look-up table within a logic element independent from each other. Note that the flip-flops can alternatively be configured as latches that can be permanently enabled for allowing a pure combinatorial signal path through the flip-flop terminals.

If shared write signals are interleaved, we have to put an alignment demultiplexer between the static system and the resource slots. This demultiplexer requires the same amount of look-up table resources as the alignment multiplexer adjusting the shared read signals from the different read multiplexer chains. If the system demands more shared write signals as shared read signals ($S_{SW} > S_{SR}$), we are not able to share all write signals in the resources used for implementing the read multiplexer chains of the shared read signals. Then $S'_{SW} + \lceil \frac{S'_{SW}}{N} \rceil \cdot R$ additional LUTs are required within the resource slots for providing the connections to the additional S'_{SW} shared write signals (with $S'_{SW} = S_{SW} - S_{SR}$). Note that in a ReCoBus signals of different interleaving factors N can be combined within the same bus. For example, we may interleave multiple address chains, but we may distribute a control signal to each resource slot.

Shared write signals came typically along with a high fan out. The resources providing access to a shared write signal

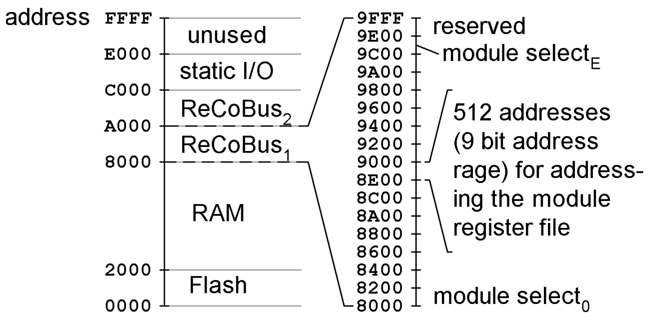

Figure 5: Address map of an example system with two independent ReCoBus subsystems.

act also as a driver. This decouples the timing of the ReCoBus from the entire modules connected to this bus.

Dedicated Write Signals
A ReCoBus implements dedicated write signals with a LUT comparing an address or identifier within the resource slots. If a dedicated write signal is not interleaved, as typical for a module select signal, it requires one LUT per resource slot plus a LUT per internal bus signal (for linking the internal address or identifier) for its implementation. As a consequence, the amount of LUTs scales only with the number of resource slots R. In the case of 4-bit LUTs this holds true for up to 15 different select signals within a complete ReCoBus subsystem. Note that by taking one or more shared write signals into account (e.g. an address), multiple modules may share the same dedicated write signal encoding.

Related approaches [6, 8] use a slot identifier to provide a dedicated write signal to a particular resource region. This also allows individually accessing multiple modules in a flexible manner but comes along with a much larger resource overhead for the comparators. The look-up table overhead for all comparators scales with $R \cdot log(R)$. A system with 50 resource slots, for example, requires 200 LUTs when comparing slot addresses and only 50 LUTs when directly comparing an address value within the resource slots. In addition, directly comparing addresses is much faster (shorter combinatorial path).

Figure 5 reveals an address space mapping of an example system with a 16-bit overall address space (64 K). A complete ReCoBus subsystem will be mapped into one consecutive address space of the system. In the example, this is a 13-bit (8 K) wide address block that is selected by the 16-13=3 most significant address bits. The next 4 address bits are directly connected to the address comparator look-up tables within the resource slots. All 16-3-4=9 lower address lines are connected as shared write signals to the resource slots and allow to address the internal registers within the individual modules located in the resource slots of the system. Note that a system may contain multiple independent ReCoBus subsystems.

The module addresses are directly encoded within the comparator LUTs as a one-hot encoded value. This value can either be set by manipulating the LUT value in the configuration bitstream or by shifting in this value after the module reconfiguration in a second step. In the latter case, we implement the comparator with the help of a so called SLR16 shift register primitive. This primitive configures the LUT flip-flops to a shift register with a write port and random read access to the LUT values. More details on this addressing technique can be found in [12].

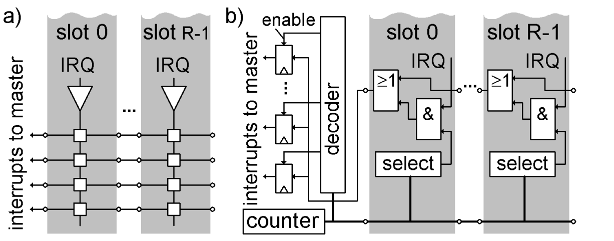

Figure 6: Dedicated read signal distribution exemplary shown for interrupts.

Dedicated Read Signals
The last ReCoBus signal class is required for providing direct connections from a module located in some resource slots to the static part of the system (for example, an interrupt line). The difficulty that arises at the implementation of this signal class is to allow multiple instances of the same module while constructing the logic and routing layout of the resource slots completely homogeneous. The approaches discussed in [7] solve this issue but require an enormous amount of logic and are consequently not suitable for systems with many individual resource slots.

The ReCoBus architecture allows to select between two implementation alternatives. The first one, illustrated in Figure 6 a) allocates a set of homogeneous routed wires that run over the resource slots. Within the resource slots, there exists another wire that drives the dedicated output signal. Connections between both signals can be set by directly modifying the configuration bitstream. The driver within the resource slot can be shared with resources required to implement the dedicated write signals. Thus, this approach is almost for free, but requires the capability to perform simple bitstream manipulations at runtime.

The second alternative for implementing dedicated read signals uses a time multiplexed shared read multiplexer chain, as shown in Figure 6 b). This chain requires one LUT plus another one for the select generation within each resource slot. This reasonable overhead is bought with an additional latency for transferring the dedicated write state into the static part of the system.

3.1 Point-To-Point Links

Beside the pure ReCoBus, we developed a further technique called I/O bars for providing point-to-point communication for modules located in the resource slot of the system. The idea of the I/O bars is to allocate homogeneously routed wires similar to the ones allocated for the dedicated read lines in Figure 6 a). In each resource slot, we can decide to either route through this signal or to access this signal by a module that may modify the incoming data while sending the results further on the same set of wires. Thus, I/O bars can be segmented to allow data streaming among multiple modules within the resource slots. This technique requires only one LUT per signal wire of a bar in each resource slots that requires access to the particular I/O bar. Depending on the system requirements, multiple I/O bars may be independently used for different data streams (e.g. a video bar and an independent audio bar). More details on I/O bars can be found in [11].

3.2 The Cost of Modularization

This work aims to completely encapsulate modules in predefined resource slots that provide the communication infrastructure to the rest of the system. The impact on routability, power consumption, and speed when constraining modules into fixed resource slots was examined in [9]. The authors found that smaller modules pay a considerable penalty with respect to the achievable clock frequency for extreme aspect ratios. For example, a divider that is just one CLB column wide and spanning over the full height of the device has had a 94% increase in the propagation delay. For larger modules with more than 1000 slices (equivalent to 2000 4-bit LUTs), the timing overhead was in average below 10%.

We performed similar experiments and explored in addition the impact of blocking specific wires for implementing the internal routing of a module. This points out the cost of the ReCoBus communication architecture in terms of routability and latency. Xilinx FPGAs provide different routing resources and we put focus on blocking some horizontal double and hex lines (wires that span two and respectively 6 CLBs far) from the module routing. We examined four different modules that all occupy more than 90% of the available logic resources in a bounding box that is 32 CLBs height and 5 CLBs wide, thus, providing 1280 look-up tables. Beside a FIR filter, a DES56 core, and a FFT module, we used a synthetic module that implements a huge barrel shifter in order to have one benchmark module that will lead to strong congestion during the routing steps. The experiments have been carried out on a Xilinx Virtex-II device that has 10 double as well as 10 hex lines starting in each cell towards all horizontal and vertical directions. However the results are still related to Spartan-3 FPGAs that provide just 8 double and hex lines towards all directions but have a similar logic architecture. We blocked in each CLB from one to five double lines, hex lines, and double and hex lines together. The blocking was only done for horizontal wires, but in both directions, as the ReCoBus architecture occupies almost only horizontal wires.

Table 2 lists the total number of wires that can be used to distribute data over the entire test module. This implies that wires are interleaved and that a wire is extended if it ends in a CLB switch matrix within the module bounding box. For example in the case of blocking 3 hex & double lines in both horizontal directions, the hex lines will be interleaved in 6 independent chains each containing three wires that are starting and respectively stopping in six horizontal consecutive aligned CLBs outside the module bounding box. Analogous, the double lines are interleaved in 2 independent chains. Therefore, one CLB row reserves $(6+2)\cdot 3 = 24$ wires per horizontal direction that may be used to implement the ReCoBus architecture. These are 48 wires for both directions and 1536 wires in total over the full module height of 32 CLBs.

In addition to the amount of blocked wires, the table lists the relative achieved latency after routing the module with respect to a routed module without area constraints. In the case of the synthetic module, the table lists also the number of signals that could not be routed at all. With two marked exceptions, all place and route steps have been successfully completed for all other modules. Note that we forced the router not to cross the module border.

The first row of the table shows the influence of just fitting a module into a fixed bounding box. The determined values confirm the results presented in [9]. All other values reveal the impact of the wire blocking. As expected, we haven't found a strong influence when blocking hex lines. The reason for this is that the module is only 5 CLBs wide. Therefore, only 40% of all hex lines can be accessed within the module bounding box via the mid-ports that allow to tab each hex line in the middle after a routing distance of three CLBs. In other words, 60% of all horizontal hex lines could not be used for implementing the module routing because these wires would leave the module bounding box.

Table 2: Impact of wire blocking on the latency and the number of unrouted nets (only for the synthetic module in the last column). The results are for a Virtex-II FPGA.

		#wires	FIR	DES	FFT	synthetic	
no		0	113,1%	103,1%	102,5%	123,6%	317
double	1	128	114,1%	105,6%	94,9%	142,7%	392
	2	256	113,2%	106,2%	98,0%	156,5%	400
	3	384	120,9%	108,3%	97,5%	156,1%	450
	4	512	121,1%	111,1%	106,4%	128,1%	523
	5	640	136,7%	114,3%*	106,9%	129,5%	580
hex	1	384	111,7%	100,2%	102,5%	132,4%	333
	2	768	113,9%	102,1%	95,5%	131,1%	372
	3	1152	110,6%	102,1%	95,5%	122,5%	372
	4	1536	113,2%	102,1%	95,9%	138,2%	385
	5	1920	115,4%	100,5%	97,5%	148,5%	365
hex & double	1	512	114,3%	107,0%	94,8%	130,4%	393
	2	1024	121,7%	112,3%	103,1%	133,2%	407
	3	1536	126,6%	105,4%	100,4%	128,3%	505
	4	2048	116,6%	117,6%	103,4%	131,3%	560
	5	2560	131,5%	145,7%*	112,7%	116,4%	645

*less than 5 unrouted nets

4. RECOBUSES ON SPARTAN-3 FPGAS

In the last section, we have presented and analyzed the ReCoBus architecture that can be directly used for runtime reconfiguration on Xilinx Virtex FPGAs without taking the risk of glitches. Here, we will examine how this architecture can be implemented on Spartan-3 FPGAs with the same capability and how much overhead this will produce

We will firstly examine the implementation of shared read signals before looking on write signals

4.1 Read Multiplexer Chains for Spartan-3 FPGAs

In order to perform completely glitch free reconfigurations on block erasable FPGAs, such as Xilinx Spartan-3-FPGAs, we must be able to set up the exchanged logic and routing without influence to the surrounding system. In addition, we must be able to bypass the communication after any block erase.

In order to achieve the first point, we examined the behavior of wires that are driven from a CLB column that is deleted and reconfigured again on a Xilinx Spartan-3-FPGA. We found that all horizontal double and hex lines carry a logical '1' value if the column is deleted. This is independent to the state of all possible sources to such a line. We assume that the multiplexers of the Spartan-3 routing fabric are implemented with pass transistors. Then a level restorer

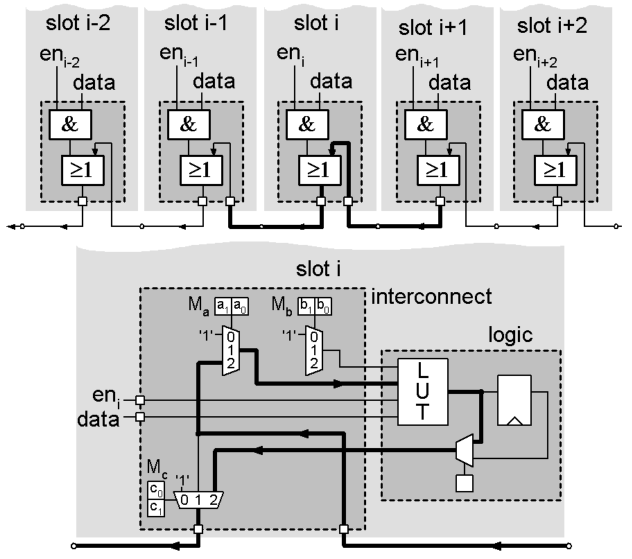

Figure 7: Signal path of a read multiplexer chain.

is required for restoring the threshold voltage drops of the pass transistors. This level restorer works as a '1' keeper latch together with the according line driver. More information on switch matrix design can be found in [15].

When the CLB-column is reconfigured again by writing all 19 frames to the FPGA, we found that the value may change from frame to frame. To overcome this problem, the configuration write was split into two phases. In the first phase, the complete CLB-column is configured with the exception that the routing resources that may influence the rest of the system will be configured in a second step. The signals in the second reconfiguration step are typically the outgoing signals to the reconfigurable bus. For simplification, let us assume that a module and the according resource slot is only one CLB-column wide when being implemented on a Spartan-3-FPGA.

With respect to the example shown in Figure 7, this means that after the snowplow command, we will configure firstly all look-up table values and internal routing (e.g., the multiplexers M_a and M_b). After this, the multiplexer M_c that connects the look-up table output to the bus is configured in a second step. This establishes the connection to slot i-1.

When configuring a framewise reconfigurable FPGA, it is possible to exchange smaller portions of configuration data than a complete frame by overwriting frame data with the same content that is already stored inside the FPGA. In the case of Spartan-3-FPGAs, we found that the configuration data within a frame can be selectively written to '1' while it is only possible to delete a bit by clearing all frames within a complete CLB-column. For instance, let us assume that the configuration bits for the multiplexers M_a, and M_c are stored within one frame in the order $a_1\, a_0\, c_1\, c_0$ (see Figure 7). Then during the first configuration step, we set the internal routing in this frame by writing a `10 00` to the FPGA. This connects the lookup-table input with the preceding slot i+1. Then, after writing the complete internal logic and routing configuration, we can connect the outgoing bus signal that is controlled by multiplexer M_c by writing a `10 10` to the FPGA. In the case of an Spartan-3-FPGA it is alternatively possible to write a `00 10` to the FPGA, because configuration bits can only be set by write operations.

So far, we have assumed that configuration bits of the multiplexer M_c are stored inside the same frame. In general, this multiplexer has much more inputs and consequently more control bits that may be spread over multiple configuration frames. We classify this situation with the *frame distance* FD that we define as the number of frames that differ between two configuration settings for a particular resource (e.g., a single wire). FD_{max} specifies the number of all frames that store configuration data for a particular resource. To be more general, FD denotes the minimum number of atomic write operations to an FPGA that are required to change a particular configuration setting for an individual resource. The frame distance is defined analog to the Hamming distance HD that specifies the number of bits that differ between two codewords.

During the second reconfiguration step where the connection to the bus is activated, it is possible that the bus is disturbed for the time required to write $FD - 1$ frames to the FPGA. A frame distance larger one indicates that a configuration codeword cannot be written in one atomic step. As the write time for one frame can be up to several hundreds of clock cycles, resources should be selected for implementing the bus that can be set with a frame distance of one ($FD = 1$). In the case of the Spartan-3-FPGA example, all `'1'` configuration bits defining the setting of a single output wire to the bus must be stored inside the same frame. With this limitation, we found that the state of an output wire to an adjacent resource slot will stay on a `'1'` value during the configuration of a single frame. However, when the Hamming distance between two configuration codes to activate a particular bus wire is larger than 1, the according bus wire may glitch, because different propagation delays to exchange the value of multiple configuration bits may set the output multiplexer to an unintended input for a short time. We have been able to detect such glitches on Xilinx Spartan-3-FPGAs on write operations as well as on clear operations with the snowplow command.

In order to deal with glitches on Spartan-3-FPGAs we can chose an adjusted timing that includes effects from the reconfiguration or we chose routing resources that can connect a wire to the reconfigurable bus by exchanging only a single bit. In the case of a Spartan-3-FPGA, this is equivalent to a one-hot configuration code to activate a bus wire as the state of all configuration bits within a CLB-column is reset to a `'0'` value due to a snowplow command. Note that it is not important that we configure the output multiplexer (M_c in the example in Figure 7) with a direct connection to the look-up table output. Other multiplexer inputs are possible by the use of additional routing resources.

A test system on a Spartan-3-FPGA board demonstrated that bus wires can be connected completely glitch-less despite the CLB columnwise clear operation required on these devices. A reconfiguration of a module that is one CLB-column wide ($w = 1$) can be performed glitch-less when the following steps are performed:

1. Disable any read or write operations to the modules that are involved into the reconfiguration process. The logic in the look-up tables of the read multiplexer chains must result in a logical '1' value when the resource slot is not selected.

2. Clear the complete CLB column by sending the column address information followed by the snowplow command to the Spartan-3-FPGA.

3. Write the module configuration with the bus logic to the FPGA. All configuration bits that connect the output wires are cleared in this process.

4. Configure the output wire connections to the bus.

5. Enable all read and write operations to the bus.

Within the steps 2) to 4), the inputs of the adjacent resource slots will detect a '1' value without any glitch. The assumption that a resource slot and a module is just one CLB column wide is in general to restrictive. If multiple CLB columns have to be reconfigured, the steps 2) and 3) have to be repeated for each involved CLB column until a complete resource slot is reconfigured. With this method, the logic of one resource slot is connected glitch-less to the bus in step 4). This process repeats from step 2) again until all resource slots of a module are reconfigured.

In our test system we proved the absence of glitches by sending static and random values to all unused multiplexer inputs that select the output wire source. In the example in Figure 7 this would be the input with the index 1 that links the output wire with the input from the previous slot.

So far we have presented techniques that ensure that the state of an outgoing signal from a CLB-column will stay constant and glitch-free on an '1' value during the complete columnwise clear operation and the following framewise reconfiguration process. In order to perform communication during the reconfiguration process of a Spartan-3-FPGA, we have to find a possibility to bypass a CLB-column that is deleted or reconfigured.

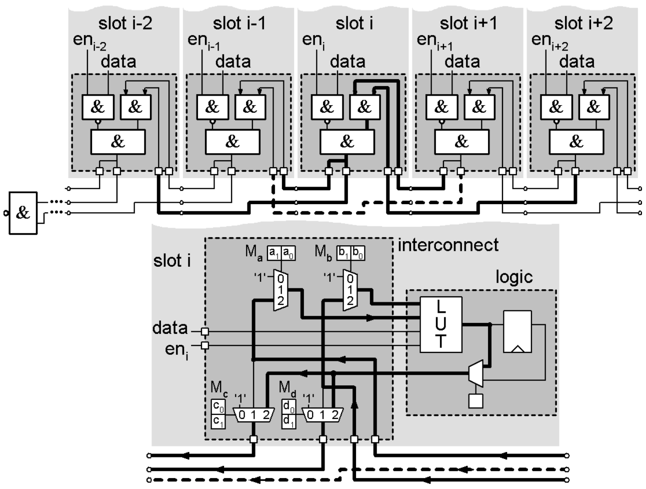

Figure 8: Read multiplexer chain for glitch-less reconfiguration on Xilinx Spartan-3-FPGAs.

We solved this problem by the use of a bypass wire (see the fat dashed wire in Figure 8). This wire allows to maintain the communication over the ReCoBus even in the case of a reconfiguration of resource slots on a Xilinx-Spartan-3-FPGA. As presented before in this section, our techniques allow reconfiguring a single CLB-column in such a way that output signals of this column will hold a stable '1' value during the complete delete and reconfiguration process on a Spartan-3-FPGA. Thus, we have to build the read-multiplexer chain in such a way that this '1' value will code the inactive state on the bus. The inactive state is the state of an unselected module or an unused resource slot. This can be achieved by applying De Morgan's theorem on the original read multiplexer chain (see Figure 7):

$$\bigvee_{k=0}^{R-1}(en_k \wedge data_k) = \overline{\overline{\bigvee_{k=0}^{R-1}(en_k \wedge data_k)}} = \overline{\bigwedge_{k=0}^{R-1}\overline{(en_k \wedge data_k)}}$$

Instead using an distributed OR-chain over all R resource slots as shown in Figure 7, the structure can be implemented by a distributed AND-gate. The result is shown in the top of Figure 8. For the distributed AND-gate a '1' value will not propagate through the chain. Figure 8 demonstrates that the complete read multiplexer chain together with an additional bypass wire can be evaluated by a single 4-bit look-up table. Furthermore, on all Xilinx Virtex or Spartan devices the already mentioned hex and double lines have a center tab. For example, on these devices the double lines reach from slot i (see Figure 8) slot i-1 and with the same double line also slot i-2. By using such routing resources, the extra bypass wire together with the according output multiplexer (multiplexer M_d in Figure 8) can be omitted.

4.2 Write Signal Distribution for Spartan-3 FPGAs

The bypass technique presented in the last section can also be applied to write signals as shown in Figure 9. Here, we used again a wire that is connected to the next and to the second next resource slot. The output wire configuration

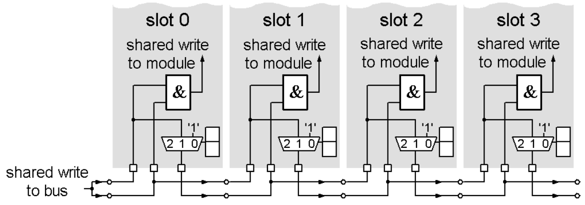

Figure 9: Write signal distribution for blockwise reconfigurable FPGAs using a bypass wire.

can only be ensured to be glitch-less if the according multiplexer can be set with a one-hot configuration codeword as mentioned in the last paragraph. In case of a Xilinx Spartan-3-FPGA, we found that we have to route the signal over a stopover point in order to set the output multiplexer glitch-free with an one-hot configuration codeword. The configuration data for this stopover point must be written prior to the configuration word that sets the output multiplexer.

As an alternative implementation, we tested a triple wire redundancy technique that is also suitable to distribute write signals completely glitch-less on Xilinx Spartan-3-FPGAs. As shown in Figure 10 this technique uses three different

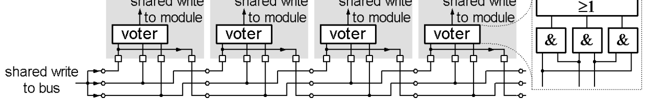

Figure 10: Triple wire redundancy write signal distribution.

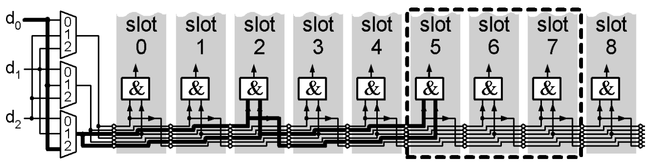

Figure 11: Write signal distribution with three interleaved chains for glitch-less operation on Spartan-3 FPGAs.

signal paths in parallel for distributing the write signal. The signal paths are displaced in such a way that a reconfiguration process of a resource slot (here a single CLB column) may disturb only one path, while the other paths are not influenced. A majority voter in each destination resource slot will mask these disturbances. We tested both techniques successfully on a Spartan XCS3-200 FPGA. In both cases, no glitches have been detected on a snowplow clear operation or the following reconfiguration step on the module side of a write signal that is evaluated behind in the currently reconfigured CLB column in the datapath.

As we require additional logic for masking the configuration effects from the bus, we cannot share write signals and read signals as mentioned for shared write signals in the last section. But it is possible to apply the interleaving technique. This reduces the overhead for all S_{MW} write signals in a system with N interleaved signal paths to $\lceil \frac{S_{MW}}{N} \rceil$ LUTs per resource slot. Thus, a module requiring all S_{MW} write signals mast be at least N resource slots wide. Let us assume a simple example with $S_{MW} = 3$ shared master write signals that are interleaved in $N = 3$ signal paths, as shown in Figure 11. In this example, a module being placed at the resource slots 5 to 7 is just wide enough to access all possible write signals. Figure 11 gives an example for the bypass wire approach tailored to Xilinx Spartan-3 FPGAs that provide hex lines that route over a span of six CLBs with a tab in the middle. The example shows the case where each resource slot is $W = 1$ CLB column wide. In general, each of the N signal paths consists of two tracks that span each over a routing distance of $2 \cdot N \cdot W$ CLBs. The two tracks are displaced by $N \cdot W$ CLBs and the paths are tapped in the middle such that the AND-gate for each interleaved write signal in a resource slot is connected to both according tracks (see also Figure 9).

4.3 Further Issues

The write signal distribution techniques for blockwise reconfigurable FPGAs are not limited to signals that are routed just straight across the area used for the runtime reconfigurable modules. As demonstrated in Figure 12, these techniques can be adapted to any zigzag path over a resource area reserved for partial runtime reconfiguration. This technique matches ideal to the current partial design flow provided by the Xilinx cooperation. In this flow, a resource area is allocated for exact one partial reconfigurable module at a time. Multiple resource areas can be defined and signals can cross such areas in the case of an implementation for a Xilinx Virtex-FPGA. In contrast, for Spartan-3-FPGAs this option is not available, thus limiting the usage this cheap FPGA family in such dynamic reconfigurable systems. However, using the techniques presented in Figure 12 would allow to cross signals related to the static part of the system through the reconfigurable part of the system.

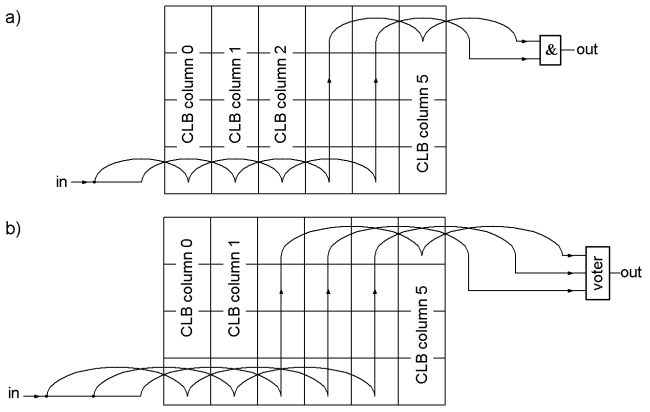

Figure 12: Example of a two dimensional glitch-free signal distribution scheme for Xilinx Spartan-3FPGAs. Signals can be laid over the CLB columns 0 - 5 that may be reconfigured at runtime. a) If routing resources are chosen such that wires involved in a reconfiguration process will stay stable on a (in this case) logical '1' value, two wires ensure together with the AND-gate a glitch-free communication during reconfiguration. b) Alternatively, the triple wire redundancy approach may be used to mask reconfiguration disturbances.

5. CONCLUDING REMARKS

In this paper we demonstrated that complex ReCoBus structures can be implemented with a reasonable resource overhead and with low influence on the module routing. Furthermore we presented that low-cost Spartan-3 FPGAs are suitable for implementing such ReCoBus structures for reconfigurable systems that allow runtime reconfiguration without interferring other parts of a system.

A remaining issue is that the I/O pins are also deleted during the reconfiguration of a Spartan-3 column. This is currently solvable with an according pin layout that allows only uncritical I/O signals to be routed via pins that are located above or below the reconfigurable area (such as LED drivers). For this reason, future low-cost FPGAs should provide a subcolumn-wise reconfiguration scheme as known from Virtex-IV and Virtex-V devices.

So far, we demonstrated the masking of glitches from a ReCoBus for smaller hand crafted buses. The current version of the ReCoBus-Builder tool (available online under `www.recobus.de`) can build ReCoBuses and I/O bars also for Spartan-3 FPGAs, but not in the glitch-less mode. Future versions will support this mode for building large scale systems with real world examples.

It requires only slightly more logic to implement the glitch-less reconfiguration mode on Spartan-3 FPGAs as compared to the much more expensive Virtex-II FPGAs. As the ReCuBus architecture occupies only a small fraction of the available FPGA resources, this logic overhead should not increase system cost, because blockwise reconfigurable FPGAs are cheaper to manufacture and the monetary benefit holds true for the complete chip. Furthermore, runtime reconfiguration is used very seldom in commercial applications. As a consequence, most customers have to pay for a feature they probably never use. This work demonstrates that low-cost blockwise reconfiguration does not limit the implementation of sophisticated runtime reconfigurabel systems.

6. REFERENCES

[1] A. Ahmadinia, C. Bobda, M. Majer, J. Teich, S. Fekete, and J. van der Veen. DyNoC: A Dynamic Infrastructure for Communication in Dynamically Reconfigurable Devices. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, pages 153–158, Tampere, Finland, Aug. 2005.

[2] Ali Ahmadinia and Christophe Bobda and Ji Ding and Mateusz Majer and Jürgen Teich and Sándor Fekete and Jan van der Veen. A Practical Approach for Circuit Routing on Dynamic Reconfigurable Devices. In *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP)*, pages 84–90, Montreal, Canada, jun 2005.

[3] L. Benini and G. D. Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, Jan. 2002.

[4] C. Bieser and K.-D. Mueller-Glaser. Rapid Prototyping Design Acceleration Using a Novel Merging Methodology for Partial Configuration Streams of Xilinx Virtex-II FPGAs. In *17th IEEE Int. Workshop on Rapid System Prototyping*, pages 193–199, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[5] H. A. ElGindy, A. K. Somani, H. Schroeder, H. Schmeck, and A. Spray. RMB - A Reconfigurable Multiple Bus Network. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA-2)*, pages 108–117, Feb. 1996.

[6] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann. Design of Homogeneous Communication Infrastructures for Partially Reconfigurable FPGAs. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'07)*, Las Vegas, USA, June 2007.

[7] J. Hagemeyer, B. Kettelhoit, and M. Porrmann. Dedicated Module Access in Dynamically Reconfigurable Systems. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece 2006, 2006.

[8] M. Huebner, T. Becker, and J. Becker. Real-time LUT-based Network Topologies for Dynamic and Partial FPGA Self-Reconfiguration. In *SBCCI '04: Proc. of the 17th Symp. on Integrated Circuits and System Design*, pages 28–32, New York, NY, USA, 2004.

[9] H. Kalte, M. Porrmann, and U. Rückert. Study on column wise design compaction for reconfigurable systems. In *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT'04)*, Brisbane, Australia, 6 - 8 Dec. 2004.

[10] H. Kalte, M. Porrmann, and U. Rückert. System-on-Programmable-Chip Approach Enabling Online Fine-Grained 1D-Placement. In *11th Reconfigurable Architectures Workshop (RAW 2004)*, page 141, New Mexico, USA, 2004.

[11] D. Koch, C. Beckhoff, and J. Teich. ReCoBus-Builder a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs. In *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL 08)*, pages 119–224, Heidelberg, Germany, Sept. 2008.

[12] D. Koch, C. Haubelt, and J. Teich. Efficient Reconfigurable On-Chip Buses for FPGAs. In *16th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2008)*, Palo Alto, CA, USA, Apr. 2008.

[13] S. Koh and O. Diessel. COMMA: A Communications Methodology for Dynamic Module Reconfiguration in FPGAs. In *14th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006)*, pages 273–274, Los Alamitos, CA, USA, 2006.

[14] Y. E. Krasteva, A. B. Jimeno, E. de la Torre, and T. Riesgo. Straight Method for Reallocation of Complex Cores by Dynamic Reconfiguration in Virtex II FPGAs. In *Proc. of the 16th IEEE Int. Workshop on Rapid System Prototyping (RSP'05)*, pages 77–83, Washington, DC, USA, 2005.

[15] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff, and J. Rose. The Stratix II logic and routing architecture. In *FPGA 2005: Proceedings of the ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 14–20, 2005.

[16] J. C. Palma, A. V. de Mello, L. Möller, F. Moraes, and N. Calazans. Core Communication Interface for FPGAs. In *SBCCI '02: Proc. of the 15th Symp. on Integrated Circuits and Systems Design*, page 183, Washington, DC, USA, 2002.

[17] M. Ullmann, M. Hübner, and J. Becker. An FPGA Run-Time System for Dynamical On-Demand Reconfiguration. In *Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, New Mexico, Apr. 2004.

[18] H. Walder and M. Platzner. A Runtime Environment for Reconfigurable Operating Systems. In *Proc. of the 14th Int. Conf. on Field Programmable Logic and Application (FPL'04)*, pages 831–835, 2004.