

TECHNISCHE UNIVERSITÄT DRESDEN
DEPARTMENT OF COMPUTER SCIENCE
INSTITUTE FOR COMPUTER ENGINEERING
CHAIR FOR VLSI DESIGN, DIAGNOSTIC AND ARCHITECTURE

Study's Thesis

Virtualisation of FPGA-Resources for Concurrent User Designs Employing
Partial Dynamic Reconfiguration

Paul Richard Genßler
born on May 26, 1990 in Berlin
(Mat.-Nr.: 3569856)

Supervising Professor:
Prof. Dr.-Ing. habil. Rainer G. Spallek
Advisor:
Dipl.-Inf. Oliver Knodel

Dresden, Friday 30th October, 2015

Statement of Authorship

I do solemnly declare that I have written the presented study's thesis

Virtualisation of FPGA-Resources for Concurrent User Designs Employing Partial Dynamic Reconfiguration

by myself without undue help from a second person others and without using such tools other than that specified. Where I have used thoughts from external sources, directly or indirectly, published or unpublished, this is always clearly attributed. Furthermore, I certify that this study's thesis or any part of it has not been previously submitted to the exam office.

Paul Richard Genßler, Dresden, Friday 30th October, 2015

Competition Remark

I have not directly nor indirectly received any monetary benefit from third parties in connection to this study's thesis. The mention of Names, Products, Manufacturer and company names only serves as information and does neither represent the use of trade marks nor a recommendation for a product or a company.

Contents

List of Figures	V
List of Tables	V
Acronyms	VII
1 Introduction	13
1.1 Field Programmable Gate Array	13
1.2 Cloud Computing	13
1.3 The Power of FPGAs in the Cloud	14
2 Partial Reconfiguration	17
2.1 Background	17
2.2 Related Work	18
3 Design	21
3.1 Extending the RC2F Framework	21
3.1.1 User Interface	21
3.1.2 Enhancing and Optimizing the Existing Flow	22
3.2 Designing a Reconfiguration Interface	22
3.2.1 Available Technologies	22
3.2.2 MicroBlaze Compared to a State Machine	23
3.2.3 The Interface Between Host and ICAP	23
3.2.4 ICAP Data Bus Width	24
3.2.5 Compensating a Non-Continuous Data Stream	24
4 Implementation	27
4.1 The ICAP Controller	27
4.1.1 Bitstream Structure	28
4.1.2 ICAP Status Word	28
4.1.3 State Machine	29
4.2 Decoupling of a User Design	29
4.3 RC2F User Frame	30

4.4	Extended RC2F Flow	32
5	Results	35
5.1	Compression of Bitfiles	35
5.2	Overhead of JTAG and PCIe	36
5.3	Influence of the FIFO on the Data Throughput	36
5.4	Performance Comparison of ICAP Controllers	39
5.5	Influence of DPR on the Tool Flow	40
5.6	Resource Usage of Standard and DPR Designs	41
6	Conclusions and Future Work	43
6.1	Conclusions	43
6.2	Future Work	43
	Appendices	47
A	Measurments	49
A.1	JTAG	49
	References	LI

List of Figures

2.1	RC2F Framework	18
4.1	Architecture of the ICAP Controller.	27
4.2	Important Aspects of a Bitstream.	28
4.3	ICAP Status Word.	29
4.4	Synchronous State Machine Controlling the ICAP.	30
4.5	Reset and Decoupling of the User Design.	31
4.6	RC2F User Frame.	31
4.7	Existing and Extended RC2F Flow	32
5.1	ICAP Throughput with FIFO Sizes and Without PCIe Bus Loads.	37
5.2	ICAP Throughput with FIFO Sizes and a Shared PCIe Bus.	38
5.3	Performance Comparison of ICAP Controllers	39

List of Tables

2.1	Feature Comparison of ICAP Controllers.	19
5.1	Bitfile Compression Rates	35
5.2	Throughput of JTAG and ICAP.	36
5.3	Performance Comparison of ICAP Controllers.	39
5.4	Time to Bitfile for a Single User Design.	40
5.5	Time to Bitfile for a Dual User Design.	40
5.6	Utilization of a Standard Flow Compared to the New Flow.	41
A.1	Throughput of JTAG With Different Bitfiles.	49

Acronyms

ARM	Acorn RISC Machine
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface Bus
BAaaS	Background Acceleration as a Service
BRAM	Block RAM
CPU	Central Processing Unit
DDR RAM	Dual Data Rate RAM
DPR	Dynamic Partial Reconfiguration
DSP	Digital Signal Processing Unit
FIFO	First In First Out Buffer
FPGA	Field Programmable Gate Array
FSM	Finite-State Machine
GPU	Graphics Processing Unit
HLS	High Level Synthesis
ICAP	Internal Configuration Access Port
IDE	Integrated Development Environment
IP core	Intellectual Property Core
JTAG	Joint Test Action Group's Boundary Scan Port Standard
LUT	Look Up Table
NIST	National Institute of Standards and Technology
PCIe	Peripheral Component Interconnect (PCI) Express
PIP	Programmable Interconnection Point
PLB	Processor Local Bus

RAaaS	Reconfigurable Accelerators as a Service
RC2F	Reconfigurable Cloud Computing Framework
RC3E	Reconfigurable Common Cloud Computing Environment
RSaaS	Reconfigurable Silicon as a Service
SQL	Structured Query Language
SRAM	Static RAM
TCL	Tool Command Language

1 Introduction

This work enhances an existing framework to support concurrent users on a virtualized resource. In the beginning of this chapter Field Programmable Gate Arrays (FPGAs) are motivated with a quick insight into some use cases. After that, the current state of cloud computing is discussed and the final section shows the integration of FPGAs into the cloud.

1.1 Field Programmable Gate Array

The first FPGA was introduced by Xilinx in 1984, but the term itself was popularized by Actel around 1988 [Tri15]. At that time Application-Specific Integrated Circuits (ASICs) were widely used to build custom logic circuits but had the disadvantage of a long turnaround time and with the increasing mask cost it got more and more expensive to develop the chip. Today they are only used in high numbers or if constraints like power consumption or speed are very important. FPGAs on the other hand have a much quicker turnaround time and were and are cheaper for low to medium numbers of units. This makes them perfect for prototyping or low cost yet fast and efficient solutions. This is possible thanks to the chip's flexible design which is mainly based on Static RAM (SRAM) to configure Look Up Tables (LUTs) and Programmable Interconnection Points (PIPs) connecting those LUTs. Their flexibility and special features like on-chip memory, Digital Signal Processing Units (DSPs) or PCI Express (PCIe) transceiver let them perform very well in high bandwidth streaming applications. [Müh+10] proposed a FPGA based honeypot to collect malicious code samples and simulate thousands of potential targets at once. Even in the field of image processing they outperform Graphics Processing Units (GPUs) in many scenarios ([Che+08]) as well as in the processing of large datasets ([Hus+11]) and are more power efficient at the same time ([Pap+09]). Their versatility is further proven by applications in finance ([MTL09]), biology ([KPS11]), space flight ([Jac+12]) and many more.

1.2 Cloud Computing

The term *Cloud Computing* was first mentioned in a business plan by Compaq in 1996 [Cor96] and took a big step forward in 2006 with the *Amazon Elastic Compute Cloud* a service offering costumers scalable on-demand compute resources. It has evolved since then into a major buzz word but it took the National Institute of Standards and Technology (NIST) until October 2011 to release their final definition of the term.

Definition 1. *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. [MG11]*

Using services and applications on a remote, more suitable architectures became even more important with the rise of the smart phone and its limited battery life. The driving force in cloud computing is the philosophy that "using 1000 servers for one hour costs no more than using one server for 1000 hours" [Arm+09] which lead to a simple increase of processors with a traditional architecture. But the servers high power consumption limits the growth of data centers and hence providers are exploring how to improve the performance through alternative architectures like GPUs or FPGAs.

1.3 The Power of FPGAs in the Cloud

Deploying such versatile devices like FPGAs into an on-demand and easy to use cloud environment provides three levels of advantages. First, the setup process changes from physically augmenting a computer with a FPGA board and the need to install appropriate drivers to simply connecting to the cloud and to start developing. This lowers the entry level significantly, a board does not even have to be purchased and new users can start testing immediately. Senior engineers profit from advanced build in features like an out of the box framework for fast data connections and host-device communication. Additionally it decreases the time spent on generating the programmable configuration due to the reuse of predefined components. Secondly it abstracts the FPGA into a resource like a processor or another accelerator. This virtualization allows an easy exchange of the physical device but more importantly it makes it scalable and one user might utilize multiple devices or a few user share one chip. Because of their power efficiency compared to GPUs or Central Processing Units (CPUs) they allow bigger compute capabilities to be deploy into one data center. In the last step the FPGA might become transparent and with the help of High Level Synthesis (HLS) systems accelerate the user's calculations on the fly. Custom or pre-build configurations would have to be swapped dynamically which requires a fast and effective technique.

Currently FPGAs find use in the network infrastructure ([Sys99], [WMC06]) or in the field of security ([EV12]) of cloud computing, however they are not accessible by the customers. Their use as a computation resource remains the area of research as the amount of papers indicates. This work has enhanced the Reconfigurable Cloud Computing Framework (RC2F) framework with a fast embedded controller to reconfigure predefined regions of the chip without interrupting the other parts and it also has explored ways to decrease the configuration time. To support multiple users on a single FPGA an advanced tool flow has been developed effectively virtualizing the resource.

The following chapter 2 explains partial reconfiguration and shows the state of the art. Important

design decisions are discussed in chapter 3 succeeded by the description of the implementation in chapter 4. Chapter 5 evaluates the design and the last chapter 6 concludes the work and presents future work.

2 Partial Reconfiguration

The configuration of a FPGA can be generalized in three steps: halt the device, configure it and start it again. This flow is not a problem in a single user environment, however it is not feasible for a virtualized device, resetting it would also disrupt other users who not even know that they are sharing the resource. Thus, a different flow has to be used, allowing the on-line exchange of partitions within the whole design. Dynamic Partial Reconfiguration (DPR) is therefore the foundation of this work and future development directed to FPGAs as an embedded custom accelerator. The following section explains the basic methodology and the section afterwards surveys use cases in research.

2.1 Background

In order to work with DPR the design has to be split into a static and one or more dynamic partitions. They contain the users logic and can be swapped later on. The interface between them and the static can not be changed after the initial configuration, however, different partitions may have different ports. The static contains special resources like PCIe, clock generators and also the controller to drive the Internal Configuration Access Port (ICAP). This is one of various methods to reconfigure the device [Xil12b] but the only embedded solution, other interfaces need an external on-board controller or are too slow, for example Joint Test Action Group's Boundary Scan Port Standard (JTAG). Furthermore is the ICAP, and its external counterpart SelectMap, the only way to reach the maximum reconfiguration speed of 400 MB/s. From the users perspective the tool flow itself does not change significantly, only an extra file describing the states of the partitions has to be added. At the end bitfiles for each partition have been generated and can be used to alter the corresponding region, which is inevitable smaller than the whole device, hence the reconfiguration is faster. Nonetheless the configuration time is still a big concern in application with frequent swapping and various approaches were explored. In [DF07] caching techniques were introduced, [Hau98] researched prefetching algorithms and bitstream decompression to reduce the required bandwidth was investigated by [LH99] and [Hue+04].

The Reconfigurable Common Cloud Computing Environment (RC3E), described in details in [KS15], is a hypervisor controlling several nodes hosting one or multiple FPGAs. It offers the user three service models, Reconfigurable Silicon as a Service (RSaaS) to control the full device, Reconfigurable Accelerators as a Service (RAaaS) to allocate a partition, which can be configured with an individual design, and standard services are provided through Background Acceleration as a Service (BAaaS). The later two virtualize the FPGA into smaller partitions, thus, they

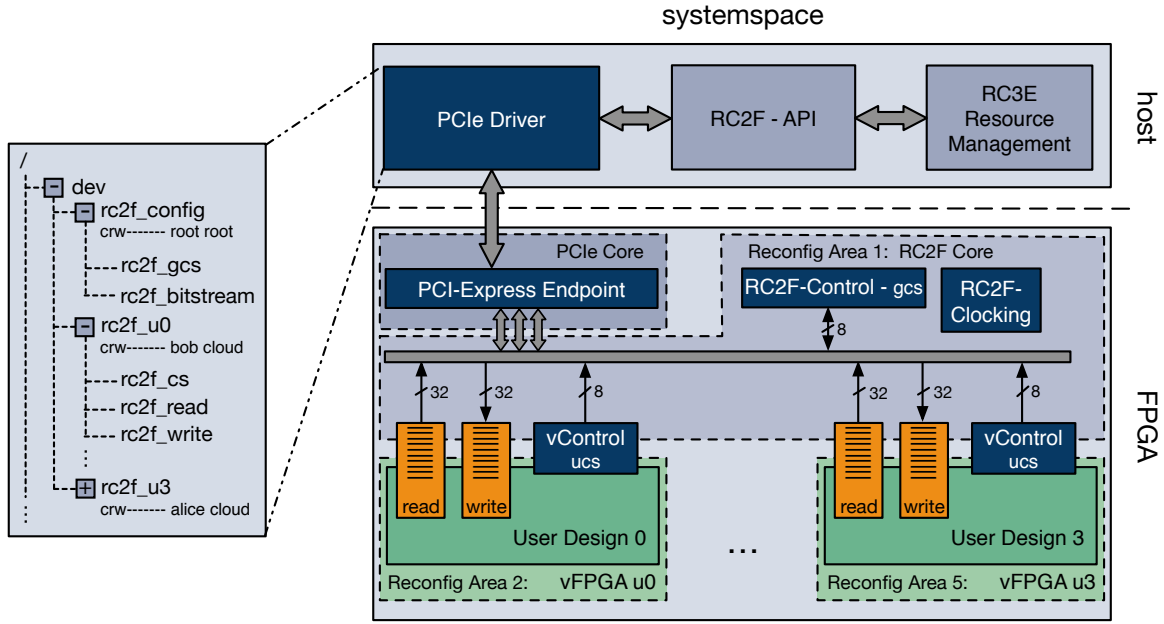


Figure 2.1: RC2F Framework with Partial Reconfigurable Areas Integrated into a Host System. (from [KS15])

require a static to manage them. Therefore the RC2F is designed to host up to four users or services on a single FPGA and to provide them with high throughput communication using PCIe.

2.2 Related Work

In the early days of FPGAs their resources were limited, thus DPR was suggested to improve their capabilities. Since then it is an area of research to exploit this technology to accelerate different computations, save energy and utilize less resources through time multiplexing. Changing the instruction set available to a processor was one of the first use cases presented in [WH95] with the goal to decrease the execution time.

Implementing a network-on-chip in a FPGA causes difficulty for the throughput and latency. The performance penalty of a general purpose communication layer on top of the FPGAs already existing one would not be acceptable in a hardware design. Thus, [She+10] exploited the ICAP as a high bandwidth on chip communication bus. In their proof of concept implementation they use Block RAMs to store data and then read the content with the help of the ICAP. Afterwards they reconfigure the receivers Block RAM with the read values.

A System-on-Chip is described by [Oet+10] utilizing partial reconfiguration to swap different filter and processing algorithms used in a smart camera system. In their example implementation they show skin color detection, a pixel marker module, a motion tracker and more. The

CPU, which remains in the static partition together with a Dual Data Rate RAM (DDR RAM) memory controller and the ICAP, uses a processor local bus to communicate with the modules. Processing large database queries is still a big challenge which is being explored by several research groups. [Bec+14] uses DPR for performance comparable to an in-memory database on a x86 machine, however the design consumes as little as 5 % power in contrast to the server, proving again the power efficiency of FPGAs. Another approach [DZT13] focuses on exchanging the execution modules to achieve a speed up and to provide a framework to cover the available Structured Query Language (SQL) instructions. [Koc+12] pipelines reconfigurable modules which communicate via a special I/O bars.

Various methods of controlling the ICAP and exchanging data width it have been investigated. Xilinx officially supports a component attached to their bus protocol Processor Local Bus (PLB) or on newer devices Advanced eXtensible Interface Bus (AXI). This allows it to communicate with a wide variate of data suppliers like DDR RAM, Flash or PCIe, however, due to the overhead it is a slow solution. [Cla+08] used an enhanced version of the PLB bus, but the DDR2 memory controller provided by Xilinx does not support bursts slowing their solution down. To compensate such delays [Liu+09] built a Block RAM cache in front of Xilinx’s controller *HWICAP*, however this limits the size of the bitfile significantly and uses a lot of valuable resources. [BML03] presented a self reconfiguring system using a MicroBlaze embedded processor to read a portion into a Block RAM, alter it and write it back. But to achieve maximum speed, own controllers and Finite-State Machines (FSMs) have to be used, like [VPP14] or [HKT11] did. There they even gone further and overclock the ICAP and reach higher throughputs therefore minimizing the reconfiguration time. A feature comparison of different approaches is presented in table 2.1, section 5.4 compares the performance aspect.

Table 2.1: Feature Comparison of ICAP Controllers.

Implementation	Bit Width	Frequency	Data Supplier	Controller	Readback
Xilinx (PLB) [Xil10]	32, 64, 128 ¹	100	PLB Bus	MicroBlaze	✓
Xilinx (AXI) [Xil11]	32 ¹	100	AXI Lite Bus	MicroBlaze	✓
[BML03]	8 ²	50 ²	OPB ³ Bus	MicroBlaze	✓
[Cla+08]	32	100	DDR2	FSM	✓
[Liu+09]	32	100	Block RAM	HWICAP	✗
[VPP14]	32	125	PCIe	Host CPU & DMA ⁴	✗
Proposed	32	100	PCIe	FSM	✗

¹ bus width of the controlling component

² maximum on Virtex II

³ on-chip peripheral bus

⁴ direct memory access

3 Design

Important design aspects are discussed in this chapter to draw conclusion for the implementation. First, possible ways of extending the RC2F framework and its flow are explored. In section 3.2 a decision between JTAG and ICAP is made followed by an investigation of parameters and surrounding components.

3.1 Extending the RC2F Framework

Every framework is being developed with a typical use case in mind. The RC2F design is no different and could in theory host a multitude of users, how to connect them to the static partition is discussed first. After that the extension of the current design flow is explored.

3.1.1 User Interface

The connections between a static partition and a reconfigurable area in a DPR design can not be altered after the initial configuration. Thus, the interface has to be as flexible as possible yet as simple as possible. The RC2F framework connects the host with the device over PCIe and the Xillybus Intellectual Property Core (IP core), which only provides a few control and data signals for each stream. In order to receive or send data the user has to wait until the core is ready for a transfer, which probably stalls the rest of the design and unnecessarily decrease the performance. However, if the core is ready it accepts data at 250 MHz (7 Series), a speed neither achievable nor required by every design. A simple solution is a First In First Out Buffer (FIFO), a very common element throughout the framework and many other systems. It provides a data buffer but it also supports two different clock domains so the user's design might run with a lower, the same or even a higher speed. This makes them almost inevitable, hence many users will need them so they could be placed inside the unchangeable static region. The FIFOs are designed to stream data and not to hold small values or a configuration. This is done using an addressable memory, but again the user would have to implement a controller, so it could be moved into the static as well. On the other hand some advanced users might not need such a controller and would like to invest the resources into other parts of their module. The same applies to the FIFOs, a random number generator does not need a steady input or some operator might reduce a stream into a single result kept in the memory.

Therefore both possibilities were combined into a two level approach to satisfy all user groups. The static region simply provides the low level PCIe ports to the users so they have more resources

available and a direct access. The second higher level is included within the user's region and can be removed if need be. The standard template contains FIFOs and a memory controller to lower the initial effort. Somebody prototyping or a less experienced user is able to start directly without the burden to implement basic components first. This approach also scales better if the number of users is increased later on.

3.1.2 Enhancing and Optimizing the Existing Flow

The current design flow for the RC2F framework is does not differ from a standard single user flow. Every user design has to be available when the synthesis starts, a small change after that and it has to start over again delaying the results for everyone. Furthermore some sort of communication between the designers has to happen when to start the flow making them dependent on others who might not be available all the time. Hence, a new flow has to make the user independent of others and of their designs. It can also decreases the flow time if only the user's part has to be implemented not the whole system again. Giving away the source code to another, maybe unknown, user might even be a security concern the big to ignore for a designer. It has to be the goal to isolate the users so they do not have to trust each other. But the most important feature of an enhanced flow is the ability to dynamically reconfigure parts of the design without disrupting other users. All in all, a new design flow has to abstract those issues and be more secure, faster and easier to use.

3.2 Designing a Reconfiguration Interface

The ability to dynamically reconfigure parts the design is essential in a virtualized environment. Several ways exists to implement this feature as an on-board, external or internal solution. This section discusses two of them, the serial interface JTAG and the embedded ICAP and complements it with an investigation of parameters and controllers.

3.2.1 Available Technologies

A FPGA can be configured in multiple ways and the two approaches considered in this work are quiet different. A very common one is JTAG, an interface originally developed for debugging and testing purpose. Many boards offer this port and via a special programmer it is connected to a computer host. An alternative is the ICAP introduced in section 2.1 which can be wired to different sources like onboard DDR3 memory or the PCIe bus. JTAG has the advantage of being very simple to use and it can configure the FPGA regardless of its current state. A component to control the ICAP is not implemented by default, therefore it can not be used for initial configuring or to apply changes to itself. However, in a productive environment this would not be a big problem, after the chip was configured and installed, its core configuration including the ICAP will not change. As an external interface JTAG relies on a cable connection and a

programmer supported by the host, at point of writing only available for the x86 architecture rendering Acorn RISC Machine (ARM) powered servers not usable. The ICAP on the other hand can be included into the existing host-to-device data transfer structure making it independent of the actual host implementation. Furthermore doubles the PCIe bus the maximum ICAP write rate of 400 MB/s. It offers a superior speed of up to 800 MB/s [Xil14] compared to JTAG's theoretical maximum of 4 MB/s at 24 MHz [Xil15b]. The extra resources needed for the ICAP controller are insignificant in comparison to the speedup of 200.

3.2.2 MicroBlaze Compared to a State Machine

Xilinx offers an extension to the MicroBlaze microprocessor to control the ICAP with only a few lines of software code. This makes it easy to deploy and use with any PLB/AXI bus compatible memory like DDR3. But neither the current RC2F framework nor the Xillybus core provides a PLB connection. An adapter would have to be developed to interface with the rest of the design. Furthermore the XPS HWICAP module [Xil10] supports Virtex-6 and older devices and connects via the PLB bus, the AXI HWICAP [Xil15c] on the other hand can only be used with 7 Series devices and the AXI bus. The additional adapter between the buses alone would lead to a 20 % resource increase [Xil12a] compared to a homogeneous implementation.

While the device utilization has to be considered, the achievable bandwidth is the most important feature. The ICAP is theoretically capable of processing 400 MB/s but the IP cores provided by Xilinx are significantly slower as shown by [VF12]. In that work an ICAP controller is being proposed, which can achieve the full bandwidth and at the same time has a smaller footprint. In another work the clock frequency driving the ICAP was increased up to 550 MHz yielding up to 2200 MB/s [HKT11]. Although such high speeds can not be guaranteed it shows the huge potential of a specifically tailored solution.

Because the Xilinx macros do not offer any significant advantages over other simpler solutions and only achieve 2.28 % of the theoretical bandwidth, a new controller has to be implemented. It should be easy to connect to the existing RC2F structure, which is used on a Virtex 6 and 7-Series devices at the same time. A small footprint and of course full speed reconfiguration are important, too.

3.2.3 The Interface Between Host and ICAP

The Xillybus core connects a simple file in the host system with a FIFO in the FPGA design for every single stream. To save resources the user's FIFO could be reused to receive the bitstream and feed it into the ICAP. It would only require a few multiplexers to switch between the different users or in a two user design every partition could use its own ICAP. Through the RC3E framework the access would be serialized to prevent an illegal parallel usage. Also a hijack might be prevented using non user writeable memory to switch into configuration mode. However, other problems arise if the user design space starts directly with the PCIe interface, a freedom an advanced user might esteem, or in later iterations a reconfigurable partition might

not even have a host connection.

Therefore a neat and future proof solution has to have its own streams. It not only simplifies the design around the user, but also decouples itself from the number of users or reconfigurable areas. A single private connection is easier to manage within the controller and also on the host, no concurrent request or dynamically switching the files to write to. An extra stream is with an additional cost of 14 to 28 slices on a Virtex-6 [Xil] more than affordable, the FIFO could be small to save resources.

The later solution, to separate the controller entirely from the users, is mostly independent of further development and offers a cleaner interface to both, the user and the service provider. The slightly higher utilization is negligible in comparison to the overall resource usage of the rest of the static design.

3.2.4 ICAP Data Bus Width

The ICAP offers 8, 16 or 32 bit wide input and output ports. The clock frequency is independent from the width so the throughput scales accordingly. Therefore the smallest version performs worst while the 32-bit option is required for the maximum speed. This alone is reason enough to choose the widest interface, it comes in handy that the PCIe bus has the same word size. On the other hand no data alignment is needed in case of the simpler 8-bit interface and different endianness are not a problem. But these minor difficulties can be easily solved in software and hardware, respectively.

3.2.5 Compensating a Non-Continuous Data Stream

The PCIe bus is controlled by the host system, which is a standard Linux without real time constraints or a dedicated PCIe resources. Thus, it can not be guaranteed to get new configuration data every clock cycle, the controller has to handle those periods and there are several solution possible. The easiest one is a big buffer using the first in first out principle, a FIFO. Such a component is needed anyway to decouple the clock domains of the PCIe interface and the ICAP controller. This cross clock FIFO can vary in buffer size from a single entry to a few thousands. But an uncompressed bitfile for a Virtex-7 is about 20 MB, too big for even the largest 7 Series device with 8.46 MB of Block RAM [Xil15a]. With a smaller buffer it can not be assured to successfully load the new configuration.

The ICAP also offers two solutions on its own, a controlled clock or a free running clock with an enable signal. The first requires clock gating, a technique only enabling the clock if new data is present. The main reason for that feature is the SelectMap [Xil15d] interface, which uses the same internal logic as the ICAP, but is controlled by an external driver who also controls the clock. However, clock gating at the logic level leads to a difficult timing analysis and is therefore generally avoided. The more feasible and simpler solution for an internal controller is a continuous clock with a separate enable signal, where the ICAP is simply disabled if no new data is available.

4 Implementation

In order to build an easy to use dynamically partial reconfigurable system, several hardware and software components have to work together. The core of the hardware implementation is the ICAP controller which is described first of all including its state machine. The second section presents a save way to swap a user design, finally its interface, the RC2F User Frame, is explained.

4.1 The ICAP Controller

The ICAP is a very complex component even though it provides only a small and basic interface. To simplify it even further and to connect it to the PCIe interface the ICAP controller shown in figure 4.1 was developed. It handles the communication between the host and the ICAP while providing a reset signal to the user design being replaced. For a better understanding of the state machine, the core of the controller, several concepts have to be explained first.

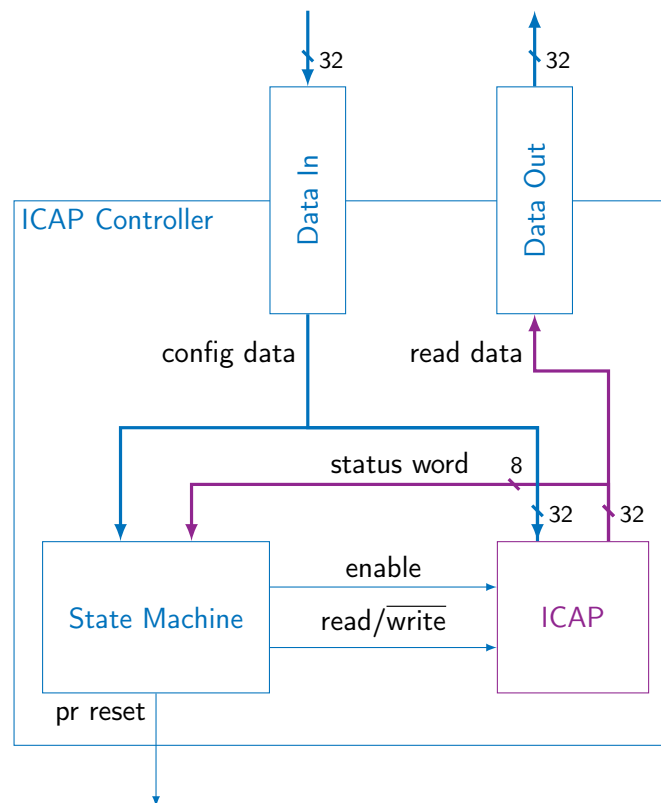


Figure 4.1: Architecture of the ICAP Controller.

4.1.1 Bitstream Structure

A bitstream holds all the information needed to configure a region or even a whole FPGA. Thus, it is possible to conclude which area of the chip will be reconfigured and cross check this with the allocated region by the user to prevent interferences with other users in case of a malicious manipulation. Those checks are neither trivial nor needed for a normally generated and trustworthy bitfile and therefore not within the scope of this work.

However to successfully use the ICAP and build a robust system, it is important to understand the structure of a bitstream specified in [Xil15d] and shown in figure 4.2. Especially when using the full 32-bit data width, the alignment of the key words becomes crucial. The first pair, which must be correctly aligned, is the automatic bus width detection. It starts with the trigger $000000BB_{16}$ followed by the detection pattern 11220044_{16} . The ICAP observes which code follows the BB_{16} directly on the lowest 8-bit of the input port. In other words, 11_{16} succeeds if the bus is 8 bit wide, 16_{16} in case of 22_{16} and 44_{16} signalises 32 bit. After some dummy data the ICAP is being synchronized by $AA995566_{16}$ and a few clock cycles later the status word changes accordingly. Now the configuration data follows and a cycle redundancy check is done in the background and the checksum will be compared to the one provided at the end of the stream. The last keyword $0000000D_{16}$ desynchronises the ICAP.

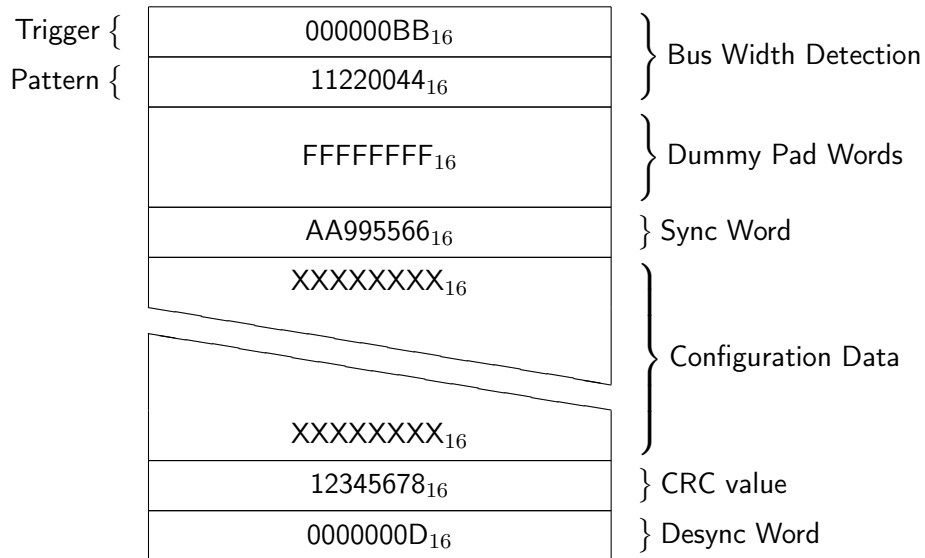


Figure 4.2: Important Aspects of a Bitstream.

4.1.2 ICAP Status Word

The ICAP constantly drives an 8-bit word on the data output reporting its current status. There are four different bits to describe four states as shown in figure 4.3. The initial and common status word is $9F_{16}$ meaning no error, not aligned, no readback and no abort. With receiving the sync word it changes to DF_{16} and in case of an error free reconfiguration it falls back to

7	6	5	4	3	0
CFGERR	DALIGN	RIP	ABORT	RESERVED	
CFGERR	0 - Configuration Error	1 - No Configuration Error	RIP	0 - No Readback	1 - Readback in Progress
DALIGN	0 - No Sync Word	1 - Sync Word Received	ABORT	0 - Abort in Progress	1 - No Abort in Progress

Figure 4.3: ICAP Status Word.

9F₁₆ after the de-sync word has been processed. A configuration error can occur if the checksum provided by the bitstream does not match the one calculated by the ICAP.

4.1.3 State Machine

The state machine presented in figure 4.4 manages the ICAP and can request data from the controller. During the initial state **start** the ICAP is disabled and waiting for data. If there is a sufficient amount provided by the host the buffer asserts the *almost full* signal and the state machine transits to **ready**. From this point on the *pr reset* signal is asserted to decouple the user from the PCIe interface. Now data is being pulled from the FIFO and every clock cycle a valid word is being presented, the state machine switches into the **write** state enabling the ICAP to process it. After the data has been written it remains in the **ready** state.

The status word is constantly being monitored and an *error* as well as an *abort in progress* signal are derived. In case of an error the state machine transits into the **abort0** state to enable the ICAP and pulling the read/write signal *rw* high in the following clock cycle using state **abort1** to trigger an abort. After four clock cycles the process is completed, *abort* is low and the state machines returns to the **start** state releasing a potentially set *pr reset* signal. An abort is also triggered if the host closes the file to begin every new reconfiguration with a consistent ICAP state.

4.2 Decoupling of a User Design

During the reconfiguration of a region the state of the boundary crossing signals is undetermined. To prevent false or even malicious signals to the PCIe interface it is important to assign well defined values. Furthermore there is no feasible way to assure the user's design complies with an asserted reset, ceases its operations and does not stress the PCIe bandwidth. Therefore the instantiation of the user frame is surrounded with additional logic shown in figure 4.5.

The user design can be decoupled in two different ways. First it is always detached in case of a global reset, which can only be driven from the common global memory inaccessible by the user, from within the design and from the controlling host environment. In the second case the region will be reconfigured. To target a specific frame its ID is written into the global memory. Regardless of that the ICAP controller drives the *pr reset* signal to all frames. The logic within a frame compares its local ID with the one from the global memory and in case of a match it forwards the *pr reset* as the *user reset* signal which also decouples the PCIe interface. After that

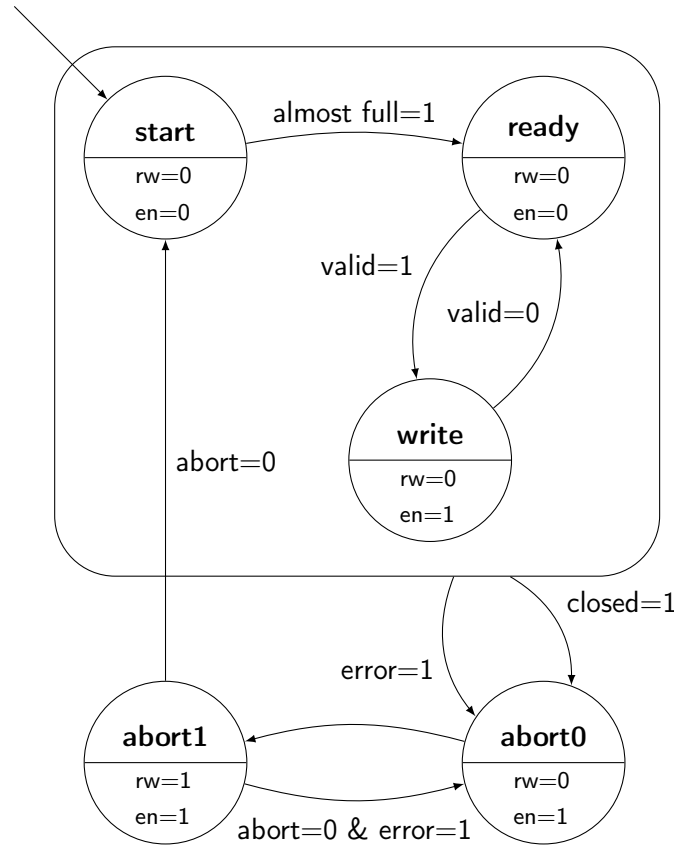


Figure 4.4: Synchronous State Machine Controlling the ICAP.

it is save of reconfigure the user frame.

If the user needs a self controllable way to reset the design he can utilize the memory associated with his region. A closer look to the internals of a frame is given in section 4.3.

4.3 RC2F User Frame

One goal of virtualization is the abstraction of everything impeding the user from focusing on the main task. In this framework the RC2F User Frame and RC2F User Container provide two different levels of abstraction. The later one is the most simplest and does only provide a basic interface with a clock, reset and write and read signals. Also the name of the design can be specified there. The lower level RC2F User Frame offers far more flexibility and exposes the whole PCIe interface, a simple memory and a stream in each direction. In support for an easy start, the standard design, completely replaceable by the user, includes a small memory controller and the PCIe loop, which can cut of the user's output and instead loop the hosts input back. It provides cross clock FIFOs used by the rest of the module, including the RC2F User Container, to safely pass data from and to the PCIe bus. The memory is writeable and readable from host side and also from the module.

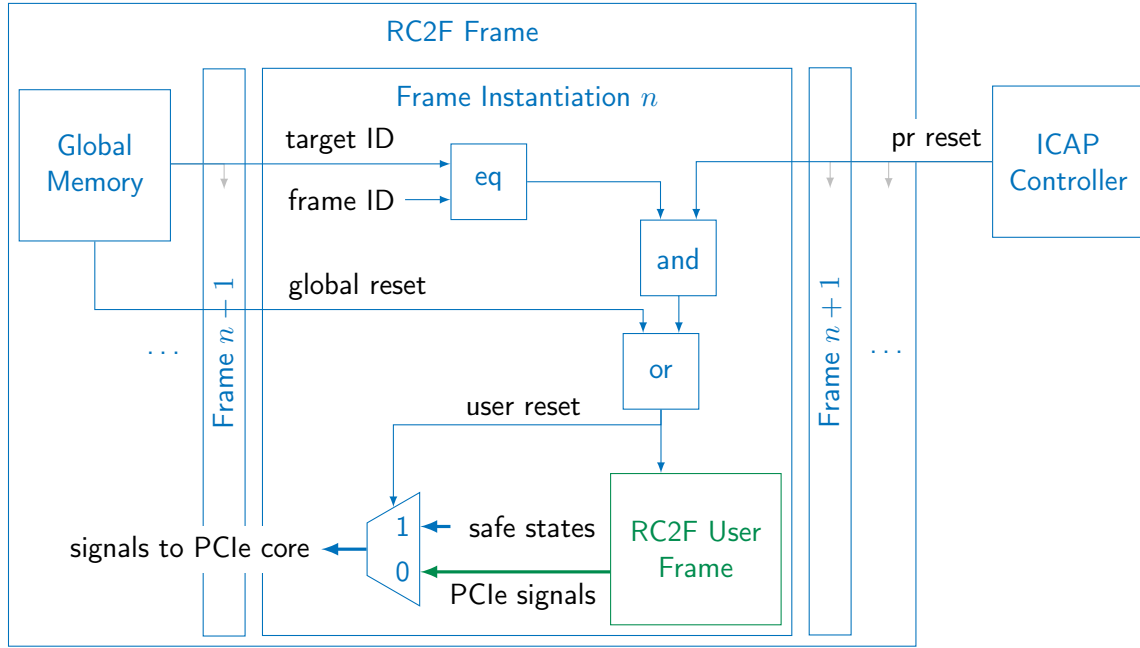


Figure 4.5: Reset and Decoupling of the User Design.

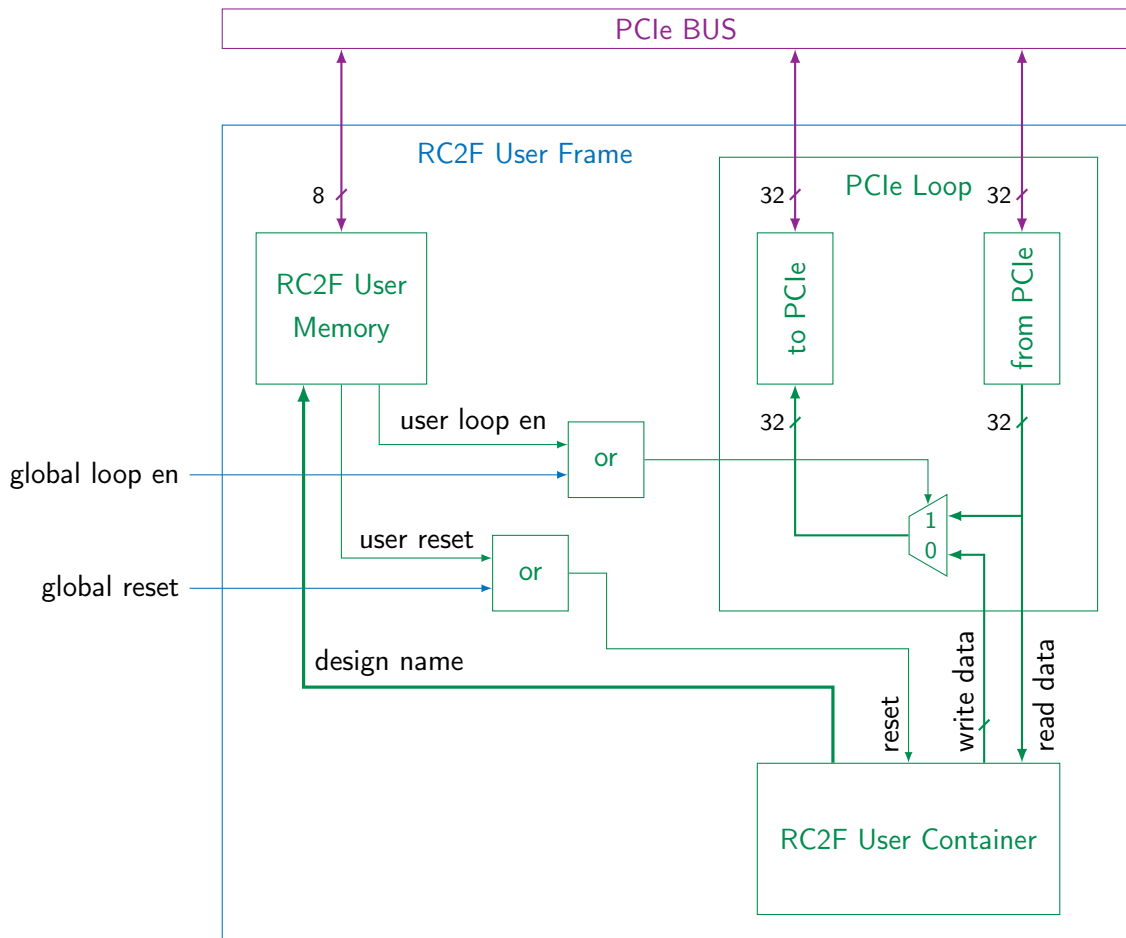


Figure 4.6: RC2F User Frame.

4.4 Extended RC2F Flow

The existing RC2F flow was only focusing on a single user and the common tool flow was sufficient. With the abstraction of a FPGA and the transparent multi user approach any significant interruption, like a hard reset of a user’s design, to serve another one has to be prevented. To achieve this pseudo single user behaviour partial dynamic reconfiguration is essential and requires a more sophisticated approach. Figure 4.7 shows those additional steps to a partial bitfile with almost all of them automated and therefore decreasing the effort for the user.

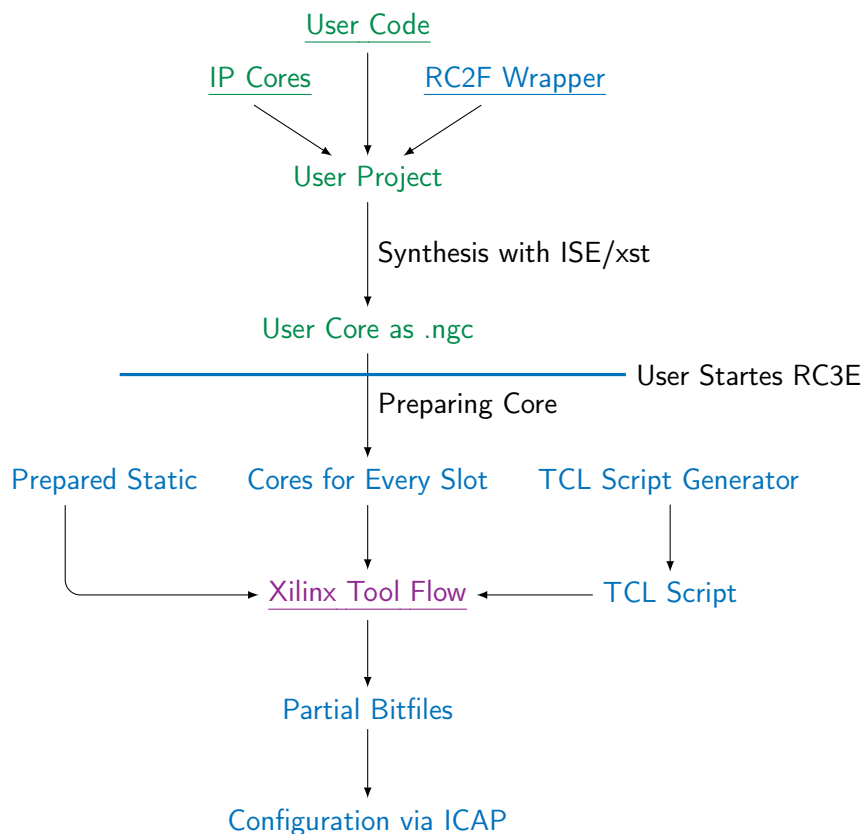


Figure 4.7: Existing and Extended RC2F Flow.

A RC3E command generates a directory structure including an ISE project and the RC2F wrappers. The user then includes own source code and IP cores if needed and starts the synthesis his preferred way. This way errors are directly reported and if using an Integrated Development Environment (IDE) it does not require further command line interactions. The path to the resulting .ngc is the only parameter another RC3E command takes to start the rest of the flow in the background.

In the first step the framework determines where to allocate a slot and renames the core accordingly. This is necessary because two modules in a DPR design can not have the same name. With n users the static design obviously has n times the same wrapper module. Furthermore the user's IP cores are bundled with the renamed one into a single .ngc file.

The second step utilizes a Tool Command Language (TCL) script controlling the Xilinx tool flow to implement the static partition together with a placeholder design or other user cores. To reduce the overhead of implementing other users' designs again it is possible to import them from a previous run and save time especially in the mapping phase as shown in table 5.5.

In the last step, after the Xilinx tools have finished and the partial bitfiles have been generated, the user's one is streamed over the PCIe interface to the ICAP. But before the pipe is opened, the RC3E tool buffers the bitfile and counts the bytes until it reads the sync word. This word has to be aligned within a single 32-bit read cycle by the ICAP and the bitstream needs to be padded accordingly. Section 4.1.1 describes the internal structure of a bitstream more detailed.

5 Results

This chapter evaluates the performance of the new flow as well as the speed up due to the new controller. The benefits of bitfile compression are presented first, than the overhead of JTAG is compared to PCIe. Section 5.3 shows the throughput with and without stress on the PCIe bus followed by a comparison of this work against examples from the literature. The final two sections investigate the overhead of the new flow. All the tests were done with a ML605 board featuring the Virtex-6 XC6VLX240T-1FFG1156 hosted on an x64 Ubuntu with Kernel 3.13.0 and connected through four PCIe 2.0 lanes.

5.1 Compression of Bitfiles

In comparison to the long map and par phases bitgen is a rather quick process and the resulting files are always the same size even if the utilization is low. As shown in table 5.2 the PCIe connection has a small overhead, thus, the time it takes to (re)configure a device correlates almost linear with size of the configuration file. Hence, decreasing this size also decreases the time spent waiting for the process to be done. That becomes even more important if a quick switch between designs is crucial, for example with the FPGA as an accelerator for special software instructions like in the BAaaS service model. The tool flow offers the option to compress the stream at the cost of a longer generation step.

Table 5.1: Compression Rates of Three Single and Two Partial Bitstreams in Comparison to the Increased Runtime of the Complete Flow.

Utilization	Low	Medium	High	Very Low	High
Bitfile Size	—————	8.8 MiB	—————	4.15 MiB	3.43 MiB
Compressed	5.07 MiB	6.16 MiB	6.24 MiB	0.48 MiB	2.59 MiB
Savings in Size	42.47 %	30.09 %	29.16 %	88.52 %	24.57 %
Increase in Time	9.67 %	1.89 %	1.13 %	8.21 %	−0.09 %

Table 5.1 confirms the expectations that a small configuration can be better compressed than one utilizing a huge part of the chip. This enables more flexible partitioning without the risk of an overhead when scheduling a small design in a big area. The compression option is a good tradeoff between often longer flow time and a faster configuration, the time that matters in a productive system or for the engineer eager to test the new iteration. In a few designs the runtime of bitgen decreases, which might be correlated to the import feature described in section 5.5 extending

also to this last phase. However, it could not be verified and therefore might be subject to future work.

5.2 Overhead of JTAG and PCIe

Every connection has some overhead to manage the transfer, so do JTAG and also the PCIe connection. This section investigates the impact of the metadata on the throughput. To prevent any uncontrollable events influencing the results all measurements were done three times. The ML605 was configured by Impact in batch mode using the Platform Cable USB running at 12 MHz, the highest frequency possible. The time was measured by the Linux command line tool *time* displaying the results in milliseconds, which is an order of magnitude below the average values. The direct control over the ICAP allows the counters embedded into the design to be as precise as a single clock cycle. However, to compare both methods such a precision is not necessary and therefore the results are saved as microseconds. The system columns includes the hosts and PCIe or JTAG communication overhead while the user bandwidth reports the actual ICAP throughput.

Table 5.2: Throughput of JTAG and ICAP.

Bitfile Size	JTAG	ICAP		Speedup	
	System	System	User	System	User
0.45 MB	0.16 MB/s	386.80 MB/s	399.99 MB/s	2461.97	2545.87
1.79 MB	0.32 MB/s	396.53 MB/s	400.00 MB/s	1256.34	1267.34
5.85 MB	0.52 MB/s	398.46 MB/s	400.00 MB/s	769.45	772.42

As table 5.2 shows, correlates the net bandwidth with the size of the bitstream but levels after the payload reaches a certain size. Data provided in table A.1 indicates this size to be at about 4 MB for the JTAG interface and 400 kB for the PCIe connection. Hence, 0.53 MB/s and 398.46 MB/s represent the maximum speed achievable using JTAG and the ICAP, respectively. But the serial connection falls short of a theoretical maximum of 1.5 MB/s due to the big overhead during the transfer, the time it takes to establish the connection and the use of Impact. PCIe on the other hand does not have such an extensive protocol therefore reaching over 99% of the theoretical possible.

5.3 Influence of the FIFO on the Data Throughput

Running with stock hardware and a standard Linux the PCIe communication might be interrupted at any point. Thus, a buffer between the ICAP controller and the host is in place to minimize the effect of such non-continuous transfers but also to reduce the reconfiguration time from the user's perspective. This section measures the ICAP throughput while the PCIe bus is free and while it is utilized to capacity. The test was conducted with a two user design where one

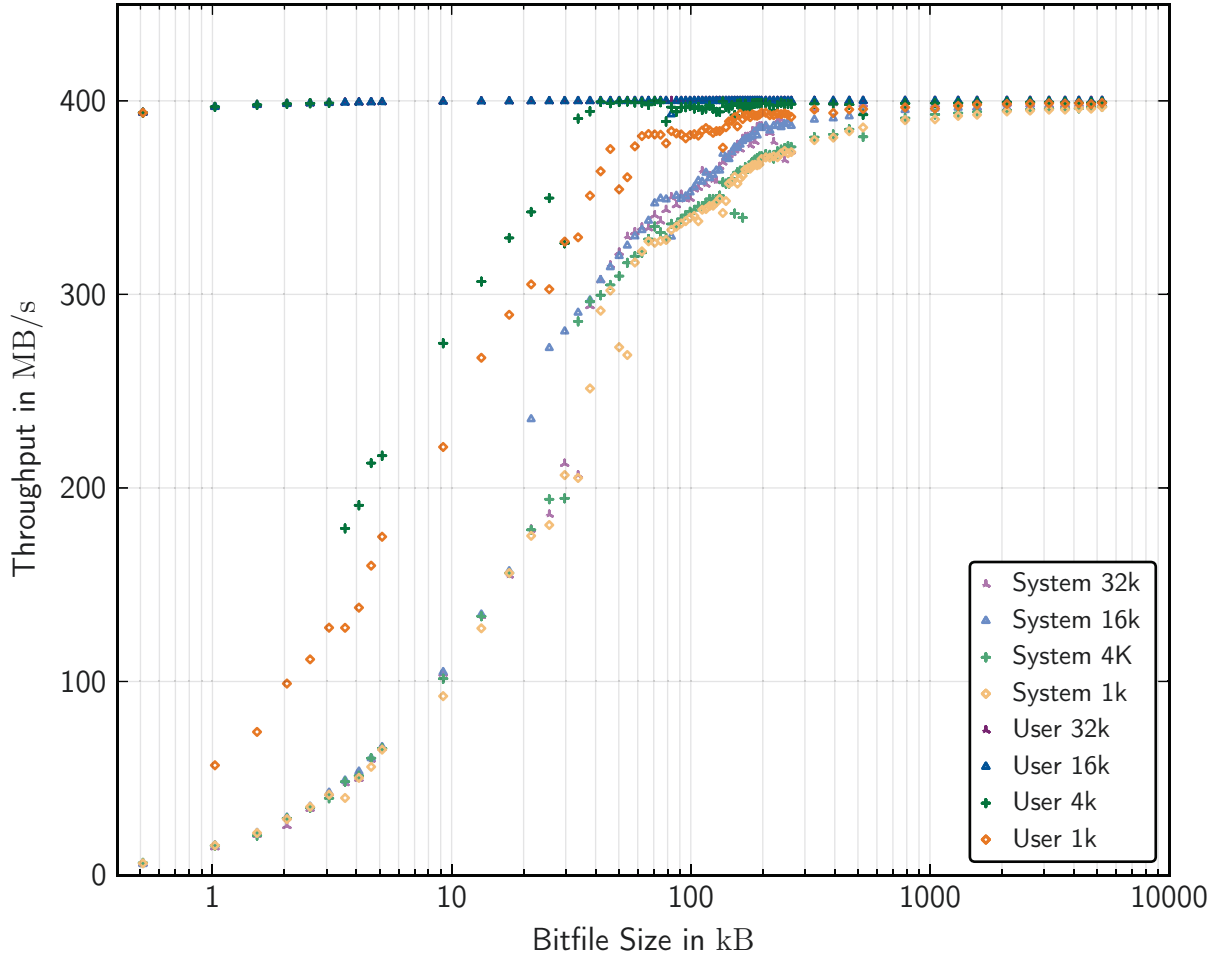


Figure 5.1: ICAP Throughput with Different FIFO Sizes and Without Additional PCIe Bus Loads.

partition has to be reconfigured. In the second test the user stressing the PCIe bus is simulated with the other partition triggered to loop back all incoming data. After a continuous 400 MB/s stream of random data was established, dummy bitfiles of various sizes were passed to the ICAP. All the tests were repeated seven times and the results were averaged. As a variation the size of the ICAP's FIFO was changed from 1,024 over 4,096 and 16,384 to 32,768 byte.

Different conclusions have to be drawn from figure 5.1 showing the throughput without additional utilization of the data connection. First, the size of the FIFO does not noticeably affect the performance from the system's perspective, although the bigger FIFOs yield slightly better results for bitfiles larger than 40 kB. In those cases the PCIe bus can fill up the bigger buffer and finish the operation earlier. Secondly it is quite different from the user perspective, but it depends on the size of the bitfile, too. The reconfiguration lasts from the FIFO being filled to 75 % until it is empty and the connection was closed from the host, or, if the buffer did not reach the filling level because of a small bitfile, from the close of the connection until the FIFO is empty. In the later case the about 40 μ s overhead of the communication does not affect the user, therefore a high performance is achieved. However, if the reconfiguration has started before the transmission has

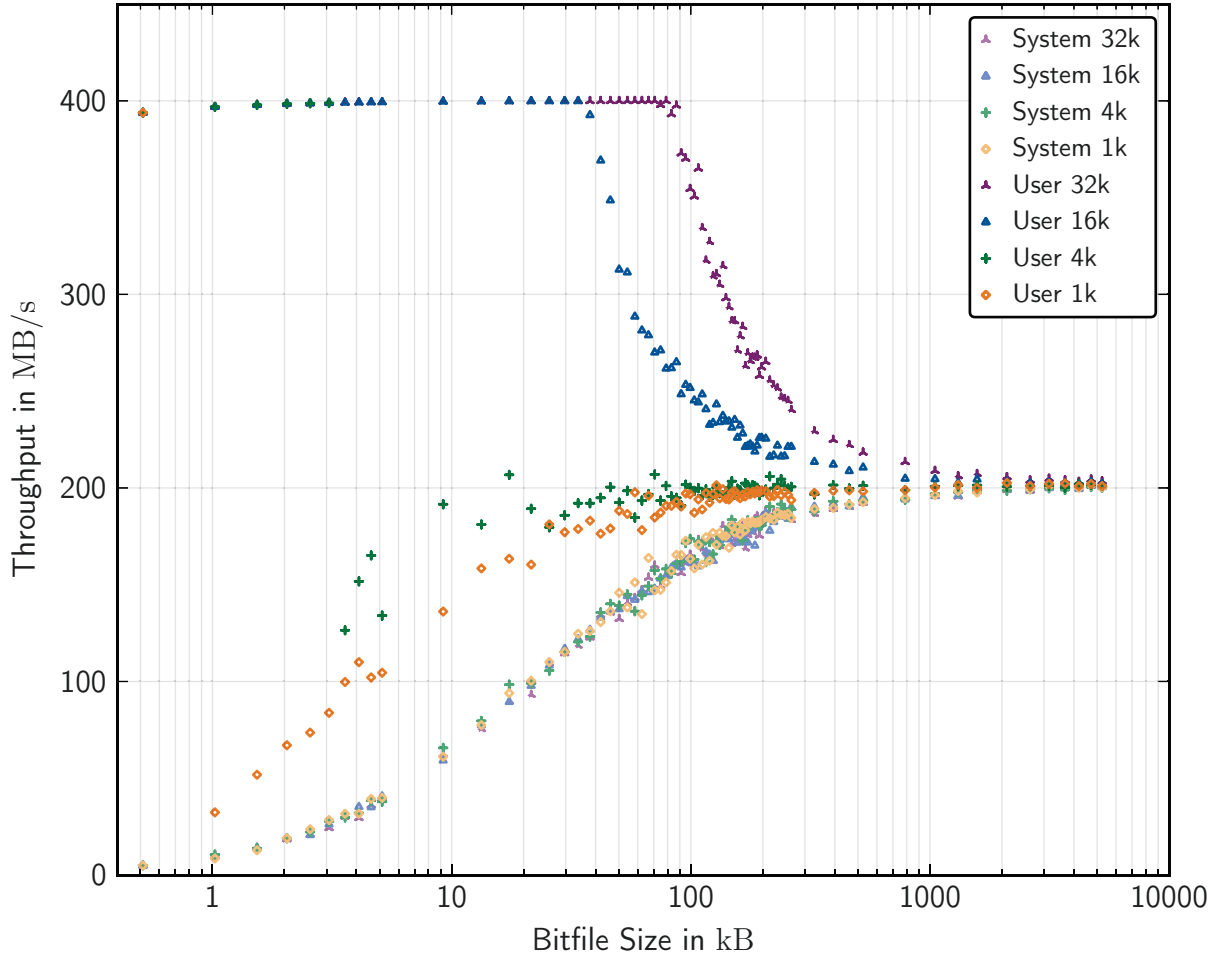


Figure 5.2: ICAP Throughput with Different FIFO Sizes and a Shared PCIe Bus.

finished, the user has to wait until the host is done which takes a relative long time compared to the actual data transfer. This effect is negligible for all buffer sizes with bitfiles larger than 40 kB.

Figure 5.2 presents a similar behaviour as above from the system perspective, but with the decreased bandwidth due to a second user the bigger FIFOs can better show off their superior performance for the user. Each time the bitfile gets larger than the buffer, the user's throughput drops off significantly, although the 16,384 byte and 32,768 byte FIFO are big enough to compensate the initial communication penalty and can sustain the full 400 MB/s with payloads up to 37 kB and 81 kB, respectively. Streaming bitfiles larger than 450 kB is almost independent of the buffer size, 95 % or more of the possible 200 MB/s can be achieved.

In addition it can be concluded that an implementation should not assume predictable reconfiguration times to control time critical systems, even though the measurements were done seven times a few outliers exist.

5.4 Performance Comparison of ICAP Controllers

The slow speed of JTAG made the use of the ICAP inevitable, hence a controller had to be developed. Literature analysis showed a number of very different approaches some of which use a PCIe interface to supply data to the ICAP (e.g. [VPP14]). The new implementation had to deliver maximum performance combined with a small footprint to be considered state of the art. The proposed controller listed in table 5.3 uses a 1 kB FIFO, it reaches 99.2 % of the theoretical maximum and does not show unusually high resource utilization.

Table 5.3: Performance Comparison of ICAP Controllers.

Implementation	Throughput MB/s	Rel. Effectiveness %	Data Bus	Registers	LUTs	BRAMs
Xilinx (PLB)	8.48	2.12	NA	746	799	1
Xilinx (AXI)	9.10	2.28	NA	477	502	1
[Cla+08]	295.40	73.85	DDR2	NA	NA	NA
[Liu+09] ¹	371.40	92.85	BRAM	963	469	32
[VF12]	399.80	99.95	DDR3	74	38	8
[VPP14] ²	488.00	97.60	PCIe	NA	NA	NA
Proposed	398.46	99.61	PCIe	251	177	2

¹ Virtex-4 FX20

² overclocked to 125 MHz

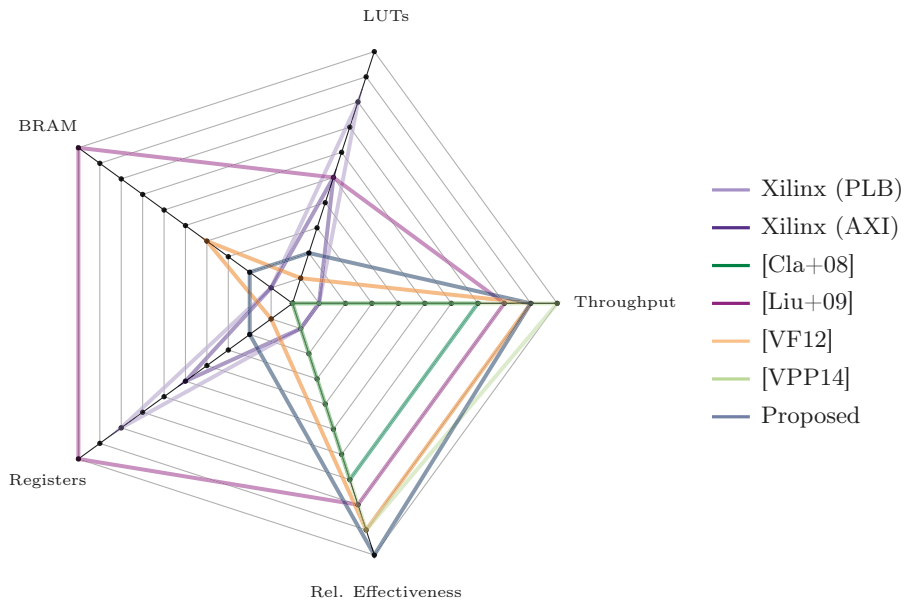


Figure 5.3: Performance Comparison of ICAP Controllers

5.5 Influence of DPR on the Tool Flow

To determine how the usage of partial reconfiguration affects the time it takes to generate a bitfile from a netlist compared to a standard flow, several test were done. In every test the same area constraints are in place and available optimization options were enabled. This includes the multithreading support of map and par which only decreases the runtime slightly. It is not known why the tools do not support this in the partial flow even though it looks embarrassingly parallel. However, the reuse of a previous implementation of a partition is require, especially the static must be imported.

Table 5.4: Time to Bitfile for a Single User Design.

Design Flow	v6_u1_BS2			v6_u1_BS10		
	Standard	DPR	Increase	Standard	DPR	Increase
overall	796 s	1513 s	0.9	1960 s	5691 s	1.9
ngdbuild	39 s	46 s	0.18	102 s	119 s	0.17
map	581 s	908 s	0.56	1528 s	3200 s	1.09
par	102 s	253 s	1.48	203 s	1910 s	8.41
bitgen	74 s	306 s	3.14	127 s	462 s	2.64

But as table 5.4 shows does this not compensate for the overall longer runtime. The two design only host a single user and the overall utilizations is low and medium with 19% and 47%, respectively. The runtime of bitgen increases with the complexity of the design but not as much as map or par do. In most of the standard flows it takes less than 10 % of the overall time. This changes especially in a partial flow featuring more than one user. Next to the complete bitfile a partial one is created for every partition present in the design, which starts a new run of bitgen every time.

Table 5.5: Time to Bitfile for a Dual User Design.

Design Flow	v6_u2_L8BT			v6_u2_BTBS10			v6_u2_BS9BS10		
	Usual	DPR	Increase	Usual	DPR	Increase	Usual	DPR	Increase
overall	215 s	1084 s	4.04	2044 s	2888 s	0.41	3749 s	8602 s	2.29
ngdbuild	12 s	20 s	0.67	107 s	122 s	0.14	286 s	280 s	0.98
map	94 s	352 s	2.74	1563 s	980 s	-0.37	2811 s	5027 s	1.79
par	58 s	267 s	3.60	228 s	962 s	3.22	407 s	2132 s	5.24
bitgen	51 s	445 s	7.73	150 s	824 s	4.49	245 s	1163 s	4.75

In table 5.5 the three listed examples utilize 7 %, 42 % and 70 % of the Virtex-6 slices, the later two also 38 % and 73 % of the DSP. The positive effect of the import of a previously implemented module is demonstrated by the design v6_u2_BTBS10 where a big portion can be reused hence the map phase is faster than in the standard flow.

5.6 Resource Usage of Standard and DPR Designs

A more detailed analysis of the results presented in table 5.6 revealed that in some designs like v6_u1_BS2 the standard flow uses more LUTs as a route through resource but in others like v6_u2_BS9BS10 the DPR flow does. The reconfigurable version of the design v6_u2_L8BT allocates more resources in the static partition increasing the utilization from 64% to 84%. It is possible that the tools just made use of the remaining space to ease the timing afterwards. Further tests with very high utilizations of a partition should answer the question about any overhead due to DPR. However, based on these numbers it can be said that using DPR does not require noticeable more resources than the standard flow without that capability.

Table 5.6: Utilization of a Standard Flow Compared to the New Flow.

Design	Registers		LUTs		Block RAMs (BRAMs)	
	Standard	DPR	Standard	DPR	Standard	DPR
v6_u1_BS2	23 708	23 711	18 744	19 357	16	16
v6_u1_BS10	54 375	54 378	43 180	44 066	16	16
v6_u2_L8BT	5092	5225	5175	5710	28	36
v6_u2_BTBS10	54 857	55 069	44 112	44 380	16	24
v6_u2_BS9BS10	102 765	102 771	81 402	81 044	28	28

6 Conclusions and Future Work

Today's data centers are limited by their power consumption making adding more general purpose processors to achieve higher performance not a feasible solution any longer. The transition to add more power efficient accelerators is easier with GPUs due to their simple tool set and a programming model similar to CPUs. FPGAs on the other hand offer great performance combined with a low power consumption and are therefore a powerful computation resource. However, their integration into a flexible and scalable cloud environment remains an area of research.

6.1 Conclusions

The RC2F framework and the RC3E hypervisor combine multiple FPGAs on different nodes to provide a single interface for the user. This work extends the framework by abstracting a single physical FPGA into multiple virtual FPGAs. In order for them to be independent of each other but still be on the same chip, dynamic partial reconfiguration has to be used. An advanced flow was developed to support this method and at the same time hide it from the user to deliver a straightforward experience. It was investigated how that affects the time-to-bitfile and the resource usage on the FPGA. The flow time increase was in a single case as high as 4 times but averages at 1.66, however, the utilization does not change compared to a non reconfigurable design. Another major step is the inclusion of the ICAP through a new controller, which performs well against other implementations and accelerates a reconfiguration by a factor of more than 2500 compared to JTAG. With that speed and the ability to reconfigure at runtime, this work is the base of further virtualization of FPGAs making them viable in a cloud environment.

6.2 Future Work

As shown by [HKT11] the ICAP can be overclocked to achieve an even shorter configuration time. However, on Virtex-6 devices the PCIe bus is the bottleneck with 400 MB/s, thus, alternatives data connections like on-board DDR3 have to be explored. This high speed interface makes background acceleration viable, only a small command would be required to start the fast streaming of the new configuration into the ICAP.

The chip could be further virtualized by transferring the design to another partition or even a different device. The user design will be frozen and the state will be read through the ICAP,

which only requires a small enhancement of the controller. At the same time the state machine could be improved to process the synchronization word in the bitstream to lower the configuration time from the user perspective further.

[Bac+14] proposed an automated approach to identify suitable regions for a given design. This becomes increasingly important if the number of users per device should be increased or the designers are able to allocation specific resources, hence manually predefined areas might not suite the user's demands.

Appendices

A Measurements

A.1 JTAG

Table A.1: Throughput of JTAG With Different Bitfiles.

Bitfile Size (kB)	Time (s)	Throughput (MB/s)
454.06	2.89	0.16
499.92	2.96	0.17
592.39	3.02	0.20
1786.42	5.66	0.32
3260.03	7.21	0.45
3597.07	7.55	0.48
4353.63	8.90	0.49
4619.38	9.64	0.48
4408.34	9.65	0.46
5761.18	11.18	0.52
5311.77	11.25	0.47
5846.55	11.29	0.52
6454.45	13.01	0.50
6540.72	13.16	0.50
9232.56	17.50	0.53

Bibliography

- [Arm+09] Michael Armbrust et al. *Above the clouds: A Berkeley view of cloud computing*. Tech. rep. 2009.
- [Bac+14] Rico Backasch et al. “Identifying homogenous reconfigurable regions in heterogeneous FPGAs for module relocation”. In: *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*. IEEE. 2014, pp. 1–6.
- [Bec+14] A. Becher et al. “Energy-aware SQL query acceleration through FPGA-based dynamic partial reconfiguration”. In: *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. Sept. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927502.
- [BML03] Brandon Blodget, Scott McMillan, and Patrick Lysaght. “A lightweight approach for embedded reconfiguration of FPGAs”. In: *Design, Automation and Test in Europe Conference and Exhibition, 2003*. IEEE. 2003, pp. 399–400.
- [Che+08] Shuai Che et al. “Accelerating compute-intensive applications with GPUs and FPGAs”. In: *Application Specific Processors, 2008. SASP 2008. Symposium on*. IEEE. 2008, pp. 101–107.
- [Cla+08] Christopher Claus et al. “A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput”. In: *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE. 2008, pp. 535–538.
- [Cor96] Compaq Computer Corporation. *Internet Solutions Division Strategy for Cloud Computing*. Nov. 1996. URL: http://www.technologyreview.com/files/74481/compaq_CST_1996.pdf.
- [DF07] F. Dittmann and S. Frank. “Caching in Real-Time Reconfiguration Port Scheduling”. In: *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*. Aug. 2007, pp. 740–744. DOI: 10.1109/FPL.2007.4380758.
- [DZT13] C. Dennl, D. Ziener, and J. Teich. “Acceleration of SQL Restrictions and Aggregations through FPGA-Based Dynamic Partial Reconfiguration”. In: *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. Apr. 2013, pp. 25–28. DOI: 10.1109/FCCM.2013.38.
- [EV12] K. Eguro and R. Venkatesan. “FPGAs for trusted cloud computing”. In: *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. Aug. 2012, pp. 63–70. DOI: 10.1109/FPL.2012.6339242.

- [Hau98] Scott Hauck. “Configuration Prefetch for Single Context Reconfigurable Coprocessors”. In: *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*. FPGA ’98. Monterey, California, USA: ACM, 1998, pp. 65–74. ISBN: 0-89791-978-5. DOI: 10.1145/275107.275121. URL: <http://doi.acm.org/10.1145/275107.275121>.
- [HKT11] Simen Gimle Hansen, Dirk Koch, and Jim Torresen. “High speed partial run-time reconfiguration using enhanced ICAP hard macro”. In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 174–180.
- [Hue+04] M. Huebner et al. “Real-time configuration code decompression for dynamic FPGA self-reconfiguration”. In: *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. Apr. 2004, pp. 138–. DOI: 10.1109/IPDPS.2004.1303113.
- [Hus+11] Hanaa M Hussain et al. “Highly parameterized k-means clustering on FPGAs: Comparative results with GPPs and GPUs”. In: *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*. IEEE. 2011, pp. 475–480.
- [Jac+12] Adam Jacobs et al. “Reconfigurable fault tolerance: A comprehensive framework for reliable and adaptive FPGA-based space computing”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETs)* 5.4 (2012), p. 21.
- [Koc+12] D. Koch et al. “Partial reconfiguration on FPGAs in practice - Tools and applications”. In: *ARCS Workshops (ARCS), 2012*. Feb. 2012, pp. 1–12.
- [KPS11] Oliver Knodel, Thomas B Preußer, and Rainer G Spallek. “Next-generation massively parallel short-read mapping on FPGAs”. In: *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*. IEEE. 2011, pp. 195–201.
- [KS15] Oliver Knodel and Rainer G Spallek. “Computing Framework for Dynamic Integration of Reconfigurable Resources in a Cloud”. In: *Euromicro Conference on Digital System Design (DSD)*. IEEE. 2015, pp. 337–344.
- [LH99] Zhiyuan Li and Scott Hauck. “Don’T Care Discovery for FPGA Configuration Compression”. In: *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*. FPGA ’99. Monterey, California, USA: ACM, 1999, pp. 91–98. ISBN: 1-58113-088-0. DOI: 10.1145/296399.296435. URL: <http://doi.acm.org/10.1145/296399.296435>.
- [Liu+09] Ming Liu et al. “Run-time Partial Reconfiguration speed investigation and architectural design space exploration”. In: *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. Aug. 2009, pp. 498–502. DOI: 10.1109/FPL.2009.5272463.
- [MG11] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. 2011.
- [MTL09] Gareth W Morris, David B Thomas, and Wayne Luk. “FPGA accelerated low-latency market data feed processing”. In: *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*. IEEE. 2009, pp. 83–89.

-
- [Müh+10] Sascha Mühlbach et al. “Malcobox: Designing a 10 gb/s malware collection honeypot using reconfigurable technology”. In: *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE. 2010, pp. 592–595.
 - [Oet+10] A. Oetken et al. “A Bus-Based SoC Architecture for Flexible Module Placement on Reconfigurable FPGAs”. In: *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. Aug. 2010, pp. 234–239. DOI: 10.1109/FPL.2010.54.
 - [Pap+09] Alexandros Papakonstantinou et al. “FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs”. In: *Application Specific Processors, 2009. SASP’09. IEEE 7th Symposium on*. IEEE. 2009, pp. 35–42.
 - [She+10] M. Shelburne et al. “MetaWire: Using FPGA configuration circuitry to emulate a network-on-chip”. In: *Computers Digital Techniques, IET 4.3* (May 2010), pp. 159–169. ISSN: 1751-8601. DOI: 10.1049/iet-cdt.2009.0009.
 - [Sys99] Cisco Systems. *Cisco Router Architecture Session 601*. 1999. URL: www.cisco.com/networkers/nw99_pres/601.pdf.
 - [Tri15] Stephen M Trimberger. “Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology”. In: *Proceedings of the IEEE* 103.3 (2015), pp. 318–331.
 - [VF12] K. Vipin and S.A. Fahmy. “A high speed open source controller for FPGA Partial Reconfiguration”. In: *Field-Programmable Technology (FPT), 2012 International Conference on*. Dec. 2012, pp. 61–66. DOI: 10.1109/FPT.2012.6412113.
 - [VPP14] Charalampos Vatsolakis, Kyprianos Papadimitriou, and Dionisios Pnevmatikatos. “Enabling Dynamically Reconfigurable Technologies in Mid Range Computers Through PCI Express”. In: *HiPEAC Workshop on Reconfigurable Computing (WRC), Vienna* (2014).
 - [WH95] M.J. Wirthlin and B.L. Hutchings. “A dynamic instruction set computer”. In: *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*. Apr. 1995, pp. 99–107. DOI: 10.1109/FPGA.1995.477415.
 - [WMC06] Greg Watson, Nick McKeown, and Martin Casado. “NetFPGA: A tool for network research and education”. In: 2006.
 - [Xil] Xillybus. *The guide to defining a custom Xillybus IP core*.
 - [Xil10] Xilinx. *DS586 LogiCORE IP XPS HWICAP*. July 2010.
 - [Xil11] Xilinx. *UG761 AXI Reference Guide*. 13.1. Mar. 2011.
 - [Xil12a] Xilinx. *PG134 LogiCORE IP AXI PLBv46 Bridge*. July 2012.
 - [Xil12b] Xilinx. *WP374 Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite*. Tech. rep. 2012.
 - [Xil14] Xillybus. *Xillybus Product Brief*. Mar. 2014.
 - [Xil15a] Xilinx. *DS180 7 Series FPGAs Overview*. May 2015.
 - [Xil15b] Xilinx. *DS593 Platform Cable USB II*. 1.5. June 2015. URL: <http://www.xilinx.com/products/boards-and-kits/hw-usb-ii-g.html>.
 - [Xil15c] Xilinx. *PG134 AXI HWICAP v3.0*. Sept. 2015.

[Xil15d] Xilinx. *UG470 7 Series FPGAs Configuration - User Guide*. June 2015.