Hindawi
International Journal of Reconfigurable Computing
Volume 2019, Article ID 4723838, 14 pages
https://doi.org/10.1155/2019/4723838

Research Article

A Real-Time Capable Dynamic Partial Reconfiguration System for an Application-Specific Soft-Core Processor

Michael Kirchhoff,[1] **Philipp Kerling**,[1] **Detlef Streitferdt**,[2] **and Wolfgang Fengler**[1]

[1]*Technische Universität Ilmenau, Group for Computer Architecture and Embedded Systems, Ilmenau, Germany*
[2]*Technische Universität Ilmenau, Group for Software Systems and Process Informatics, Ilmenau, Germany*

Correspondence should be addressed to Michael Kirchhoff; michael.kirchhoff@tu-ilmenau.de

Received 15 April 2019; Revised 30 July 2019; Accepted 31 August 2019; Published 22 September 2019

Academic Editor: John Kalomiros

Modern FPGAs (Field Programmable Gate Arrays) are becoming increasingly important when it comes to embedded system development. Within these FPGAs, soft-core processors are often used to solve a wide range of different tasks. Soft-core processors are a cost-effective and time-efficient way to realize embedded systems. When using the full potential of FPGAs, it is possible to dynamically reconfigure parts of them during run time without the need to stop the device. This feature is called dynamic partial reconfiguration (DPR). If the DPR approach is to be applied in a real-time application-specific soft-core processor, an architecture must be created that ensures strict compliance with the real-time constraint at all times. In this paper, a novel method that addresses this problem is introduced, and its realization is described. In the first step, an application-specializable soft-core processor is presented that is capable of solving problems while adhering to hard real-time deadlines. This is achieved by the full design time analyzability of the soft-core processor. Its special architecture and other necessary features are discussed. Furthermore, a method for the optimized generation of partial bitstreams for the DPR as well as its practical implementation in a tool is presented. This tool is able to minimize given bitstreams with the help of a differential frame bitmap. Experiments that realize the DPR within the soft-core framework are presented, with respect to the need for hard real-time capability. Those experiments show a significant resource reduction of about 40% compared to a functionally equivalent non-DPR design.

1. Introduction

Increasing performance with simultaneous miniaturization and the corresponding increase in the integration density of microelectronic circuits allow the realization of more complex and efficient embedded systems. These can solve problems that previously could only be solved even with large computing systems. At the same time, it is also necessary to develop new methods and approaches that can adapt to the more complex development and scale with increasing complexity. A major problem that occurs in this context is the negative relations of system properties. If, for example, the computing power has to be increased, the form factor usually also increases. For this reason and to achieve the best possible adaptation to the problem, most embedded systems are developed from scratch, which is very time and cost intensive. As an alternative approach, soft-core processors provide an excellent compromise between task-specific problem adaptation and reusability and thus cost reduction. In this paper, we present an application-specific soft-core architecture addressing this approach in Section 5. In order to maximize the reusability and therefore minimize the costs for each new project, a compatible tool-chain is presented in Section 5.2, which is designed to support the hard real-time compliance of the soft-core processor.

There are special requirements for the real-time application field addressed in this article that prevent the use of general soft-core processors. For the calculation of real-time critical tasks such as control algorithms or real-time critical image processing, it is necessary to guarantee full-time analysis capability at design time. Section 5.1 is dedicated to the special adjustments needed in order to achieve full design-time analyzability.

A prime example for this field of application is the image processing as part of the driver-assistance systems like AutoVision [1]. In this application, soft-core processors are used as coprocessors for real-time image analysis of, e.g., road tracks. Since the requirements for algorithms and processing units in driver-assistance systems can change significantly over time, both processing hardware and software must be adapted. An example is a transition from a sunny to a dark road when the car enters a tunnel. This task can be solved with conventional embedded systems only with difficulty and with a very complex hardware.

This is where a big advantage of FPGAs emerges: Their dynamic partial reconfiguration (DPR) capability enables the exchange of parts of the logic at run time and thus allows a problem-specific dynamic adaptation to changing conditions. This allows to fully utilize the potential of FPGAs compared to similar embedded solutions without DPR. This article combines the advantages of soft-core processors with the possibilities of DPR and presents a novel method for efficient implementation and problem-based realization. After an overview of related work in the respective areas has been given in Section 3, a method for creating partial bitstreams is presented in Section 4. A real-time capable soft-core processor is introduced in Section 5, and the necessary architectural decisions to ensure compliance with hard real-time limits will be discussed. Subsequently, the practical implementation of the adaptation of the processor to the DPR feature is presented in Section 5.3 in conjunction with a complete DPR system design (Section 6) that delivers partial bitstreams via a custom partial reconfiguration controller (PRC). The DPR system is presented in Section 7, and the PRC is explained in Section 7.2. Selected parts of the experiments are carried out and their results are analyzed in Section 8 before the article concludes with a summary and an outlook on future work.

2. Background

The research presented in this paper is performed on a Xilinx device of the Zynq-7000 family [2]. It is a System-on-a-Chip (SoC) that consists of an ARM Cortex-A9 based hard-core processor and a Kintex-7 series FPGA. The processor (also referred to as Application Processing Unit (APU)), peripherals, and bus interconnect including memory controllers form the Processing System (PS). The FPGA fabric and all supporting circuitry are in entirety called Programmable Logic (PL). PS and PL can communicate using several AXI buses that also allow direct access to the PS memory controller.

DPR requires partial bitstreams to be sent to the FPGA that only reconfigure a part of the device (the dynamic logic) while another part (the static logic) continues to operate. The flow of the Xilinx Vivado development environment used here requires to define at least one reconfigurable partition (RP) to host dynamic logic in the form of exchangeable reconfigurable modules (RMs). Furthermore, each RP requires the assignment of a resource Pblock, which is a defined area of logic resources such as logic slices and DSPs in the FPGA fabric. DPR functionality from within the device (self-reconfiguration) is available in the Zynq via dedicated configuration interfaces in PS and PL. In this paper, the Internal Configuration Access Port (ICAP) in the PL is used. The ICAP can be instantiated as hardwired primitive in hardware designs and receives partial bitstreams in 4-byte units at a maximum frequency of 100 MHz, thereby achieving a maximum throughput of 400 MB/s. This device-specific parameter is the main factor limiting the speed of DPR since it is a result of the capabilities of the hardware and cannot easily be influenced.

Sending bitstreams (raw streams of commands and data for the FPGA configuration controller) to the ICAP puts configuration data into 1-bit SRAM cells that define the behavior of the FPGA. These cells can only be addressed in configuration frames with the size of 3,232 bits.

3. Related Work

In this section, we give an overview of other relevant attempts at both using DPR capabilities in soft-core processors and implementing partial reconfiguration controller.

Most similar to the research at hand, exchanging integer execution units in the open-source LEON3 processor is discussed in [3]. The LEON3 is a customizable 32-bit RISC microprocessor suitable for general-purpose computing applications. It is not designed for hard real-time tasks. This specific paper mainly presents an area, energy, and power analysis showing that savings in these aspects can be achieved by DPR in practice. It does not discuss performance aspects and the design of the DPR system and PRC in detail.

Another good example of an application-specific instruction set processor (ASIP) is the Invasive Core (i-Core) [4]. This run-time reconfigurable processor works with two separate instruction sets (IS): a static (permanently available) IS and a task-specific (dynamically reconfigurable) IS. The i-core is embedded into a heterogeneous loosely-coupled multicore system which is partitioned into tiles that consist of processor cores and local shared memory.

Moving on to PRCs, those utilizing the ICAP are often designed specifically for usage in combination with a general-purpose CPU that provides commands and/or bitstream data. For this purpose, soft-cores such as the Xilinx MicroBlaze [5] included in the Vivado design suite can be used in combination with a peripheral on an AXI bus that enables access to the ICAP. The most basic design will use the AXI HWICAP IP core by Xilinx [6] (or similar) that is essentially a low-level bridge between the AXI bus and the ICAP. All control logic is implemented in software (cf. Figure 1). The MicroBlaze processor is in complete control of the reconfiguration process and responsible for fetching bitstream data from the memory and forwarding it to the ICAP.

More advanced approaches like the RT-ICAP [7] move some aspects of the process out of the CPU and provide additional features such as bitstream decompression but still offer an interface that is best suited for interoperating with a

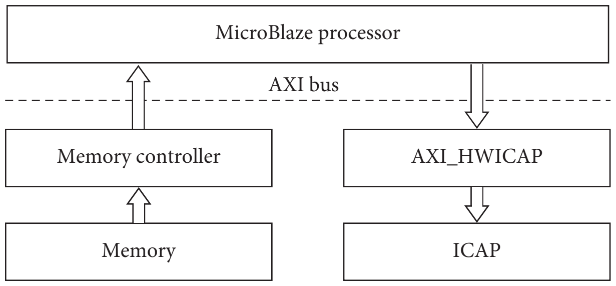


Figure 1: Exemplary structure of a microprocessor-based DPR system using the HWICAP solution by Xilinx.

processor. There are several similar designs along these lines [8–11]. A variant of this approach is to provide integrated extensibility of a general-purpose processor via DPR such as in [12]. Specifically for the Zynq SoC, it is possible to use the APU to control the reconfiguration process, which is done, e.g., by the ZyCAP [13].

Although the Zynq includes the Processor Configuration Access Port (PCAP) to stream configuration data directly from the APU, its performance is inferior to the ICAP. Xilinx specifies a typical throughput of the PCAP of around 145 MB/s on the Zynq-7000 [14]. Experiments suggest that the realistically achievable PCAP data rates are a bit lower in practice at around 128 MB/s [13, 15]. Since the reconfiguration time is directly related the throughput of the access port and the size of the bitstream, it is necessary to choose the fastest possible access port. This way, the architecture will meet stricter timing constraints which are a consequence of the real-time requirements. The ZyCAP, e.g., for this reason does not make use of the PCAP, but its general approach still has the disadvantage of restricting the design to the Zynq SoC family as pure FPGAs do not contain a comparable hard-core CPU.

There are other existing PRCs proposed in the literature that are designed for standalone usage in hardware designs without any supporting processor [16–19]. Unfortunately their design files are not publicly available, with the exception of [20]. Additionally, they do not take real-time constraints into account and often do not come close to the device limits in terms of configuration speed (or require bitstreams to fit in FPGA-internal RAM). All of these PRCs furthermore use custom or older protocols for accessing bitstreams and do not support the standardized AXI bus.

Xilinx itself also offers a closed-source partial reconfiguration controller [21] as part of its IP core library. The timing and speed characteristics are not specified at all. This turns the controller essentially into a black-box, not suitable for hard real-time systems.

Previous research has leveraged the idea of streaming bitstream data from external RAM [13, 20, 22, 23]. This allows to reach very high reconfiguration speeds without having to store the complete bitstream in BRAM, which is usually not desirable in terms of FPGA resource usage and constraints.

The concept of DPR (in combination with PRC) is used in the AutoVision architecture [1] to dynamically adapt the integrated soft-core coprocessors to changing conditions. A driver-assistance system is implemented which allows the scenery to be analyzed in real time with the help of coprocessors enabling a hardware acceleration for the image processing algorithms. The requirements of the algorithms can change significantly over time (e.g., when entering a tunnel on a sunny day), which makes an adjustment necessary. This, in turn, requires adjustment of the hardware acceleration which is carried out by using DPR. AutoVision guarantees compliance with the real-time deadlines for image analysis.

4. The torCombitgen Tool

Partial bitstreams required for DPR can be generated in several ways despite the result having the same function and effect. The most important ones are briefly presented in the course of Section 4.1, followed by the torCombitgen approach combining the advantages of these methods in Section 4.2.

4.1. Reconfiguration Flows. The typical generation that Vivado will perform when using the partial reconfiguration flow uses a module-based approach that writes one bitstream per reconfigurable module containing the data for all configuration frames associated with the RP and ignores everything else in the design [24]. All frames inside those partitions are included unconditionally.

A completely different approach oblivious of individual modules is difference-based generation [25, 26]. It takes exactly two full top-level bitstreams including the static design and one or more RPs as input and compares all configuration frames to find which ones differ from each other. Only those are written to a partial bitstream file. Compared to the module-based method, this has the advantage that potentially fewer frames have to be written in case there are frames that are identical between the two analyzed designs. All frames containing only static logic will therefore be excluded, so the bitstream is never larger than an equivalent one generated by the module-based approach.

Identical frames in dynamic logic are most likely to be found when logic is unused since dormant resources always have the same representation in configuration data. Additionally, smaller parts of logic that are functionally the same in multiple RMs might be removed, but it is still an open research question if this has a measurable impact in practice: Minute changes in any part of an RM might cause the placer to position all logic at an entirely different location within the Pblock. As a consequence, configuration data would be vastly different also in other unchanged parts, and the chance of encountering identical frames would be severely diminished.

When three or more RMs have to be considered, a bitstream must be generated for every pair of top-level bitstreams so that it is possible to switch from any RM to every other one. The number of partial bitstreams necessary for n configurations is $n(n-1) = n^2 - n$; i.e., it is quadratic in n. The amount of bitstreams quickly becomes unmanageable with higher n.

4.2. Optimized Difference-Based Bitstream Generation. Claus et al. have suggested another approach in [26] that effectively combines the advantages of the module-based and the difference-based generation. Their tool Combitgen represents an approach in between difference-based generation that requires many bitstreams and module-based generation that does not consider potential size savings by identical frames. Since it was developed only for the now obsolete Xilinx Virtex-II family, it cannot be used flexibly and in modern devices. torCombitgen, developed by the authors, is the natural successor to Combitgen. It allows to use the advantages of Combitgen on contemporary Xilinx FPGAs. Since torCombitgen uses the open-source framework Tools for Open Reconfigurable Computing (Torc) [27] for reading, manipulating, and writing Xilinx bitstreams, all FPGA families implemented in Torc are supported. It is adaptable by the community also to future FPGAs.

As can be seen in Figure 2, torCombitgen takes multiple full bitstreams as input, reads the contained data into a memory array representing the state of the FPGA configuration cells, and marks frames that are identical in each and every one of them in a differential frame bitmap. Identical frames are removed from each individual bitstream to produce one partial bitstream per top-level one. Similarly to the difference-based approach, the static logic is guaranteed to disappear as it must be the same in every input.

On the one hand, if no identical frames can be identified, the generated bitstreams will be exactly equal to the output that the module-based approach would have produced. On the other hand, if there are two input bitstreams, this method is exactly equal to the difference-based approach. As such, the size of the bitstreams should lie between these two extremes.

Claus et al. report an experimentally identified time saving of 3 to 4% in an exemplary design that consisted of a clock divider with three different division factor options. This seems to be due to an additional optimization that Combitgen performs on the bitstream by converting regular frame write commands into multiple frame writes, allowing to remove one frame of padding per write on the target platform.

5. The ViSARD Soft-Core Processor

The ViSARD (VHDL Integrated Soft-Core Architecture for Reconfigurable Devices) is a hard real-time application-specific soft-core processor, which can be configured from a generic customizable soft-core library. An early state of the ViSARD has already been published in [28]. Data can be entered or returned in any numerical format; internal calculations are performed in IEEE floating-point format [29] to ensure maximum accuracy. The processor can currently operate in single-precision (32 bit) or double-precision (64 bit) mode. The range of possible operations includes more than 30 instructions. The soft-core was designed in a way that the arithmetic logic unit (ALU) is fully modular and can be adapted at any time to an extension or replacement of the hardware execution units (EUs). That means the scope of functions for a specific problem can be carried out. The ViSARD uses a 5-stage pipeline, and every EU is internally pipelined as well.

Figure 3 shows the architecture of the ViSARD soft-core processor in the multicore configuration. This processor consists of a data path (blue) and a control path (orange), which are described in detail in this section. It has to be mentioned that, for reasons of clarity, in Figure 3, only one core is illustrated in detail. Each core communicates exclusively via the shared memory with other cores, and the architecture of every core is identical (except for the ALU).

The control path checks the current state of the execution of the given program code in each clock cycle via the program counter in combination with a RAM module (*DPRAM (Program)*) that stores the actual instructions. This program is set during design time for each core by the assembler (explained in detail in Section 5.1). The program counter passes the current program address to the program memory in each clock cycle. There, the next machine instruction is read and passed on to the decoder. This represents the instruction fetch. The next step, the instruction decode, is executed subsequently. All necessary addresses and control flow signals are passed on to the respective modules in this step. The loading of the necessary data, i.e., the content of the memory cells, for the operations and the transfer of the data to the ALU is done in the operand fetch step. Finally, the calculated result is stored in the local and/or shared memory. A special feature in this processing chain is the set/reset module. The program counter can be manipulated with this module and allows to realize a hardware loop. Since this is a crucial part to ensure the hard real-time ability of the soft-core processor, it is explained in detail in Section 5.1.

There are several optimization modules realized in the soft-core. The memory activation module (*Data RAM enable*) is one of those optimization modules and part of the control path. The task of this module is to switch off the clock of the respective data memory with the help of the information from the decoder, as long as neither a reading nor a writing operation is executed on it. This mechanism aims to reduce the power usage of the architecture. Another module that aims to minimize the energy consumption is the *ALU OP enable* module. It stores information about which values are currently being calculated in each EU pipeline of each execution unit in the ALU. If the pipeline of an EU is empty, the module will turn off that EU, further reducing power consumption.

The main part of the data path is the ALU (*fpALU*). It is designed in a way that all EUs are connected to the data links. A multiplexer within the ALU decides at which clock cycle the results of which execution units are stored in memory. Two local memories (*Data A* and *Data B*) and an optional *shared* memory are used. The memory is flattened, which means that there is only one memory level used. Other memory levels such as caches, working storage or hard disks are deliberately omitted. Memory and register are therefore identical, meaning that each memory cell corresponds to one register. Dual-port RAM (*DPRAM*) is used in each memory module, resulting in a total of two identical DPRAM modules (*Data A* and *Data B*) and an optional third memory

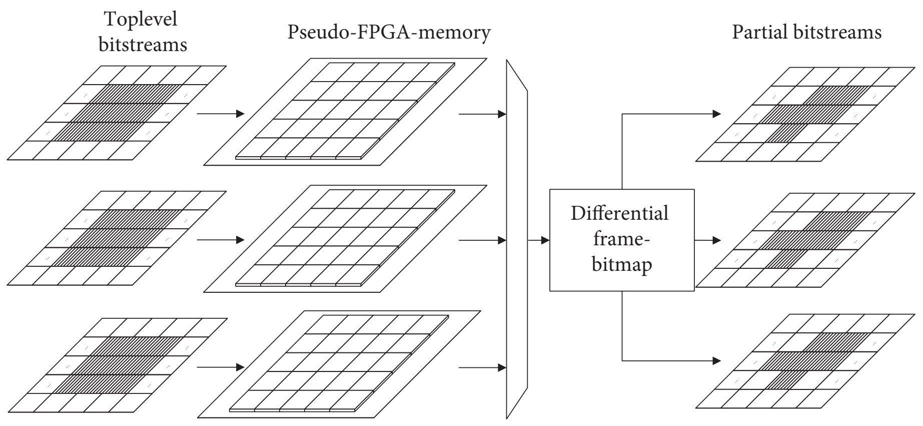


FIGURE 2: Functionality of the torCombitgen tool.

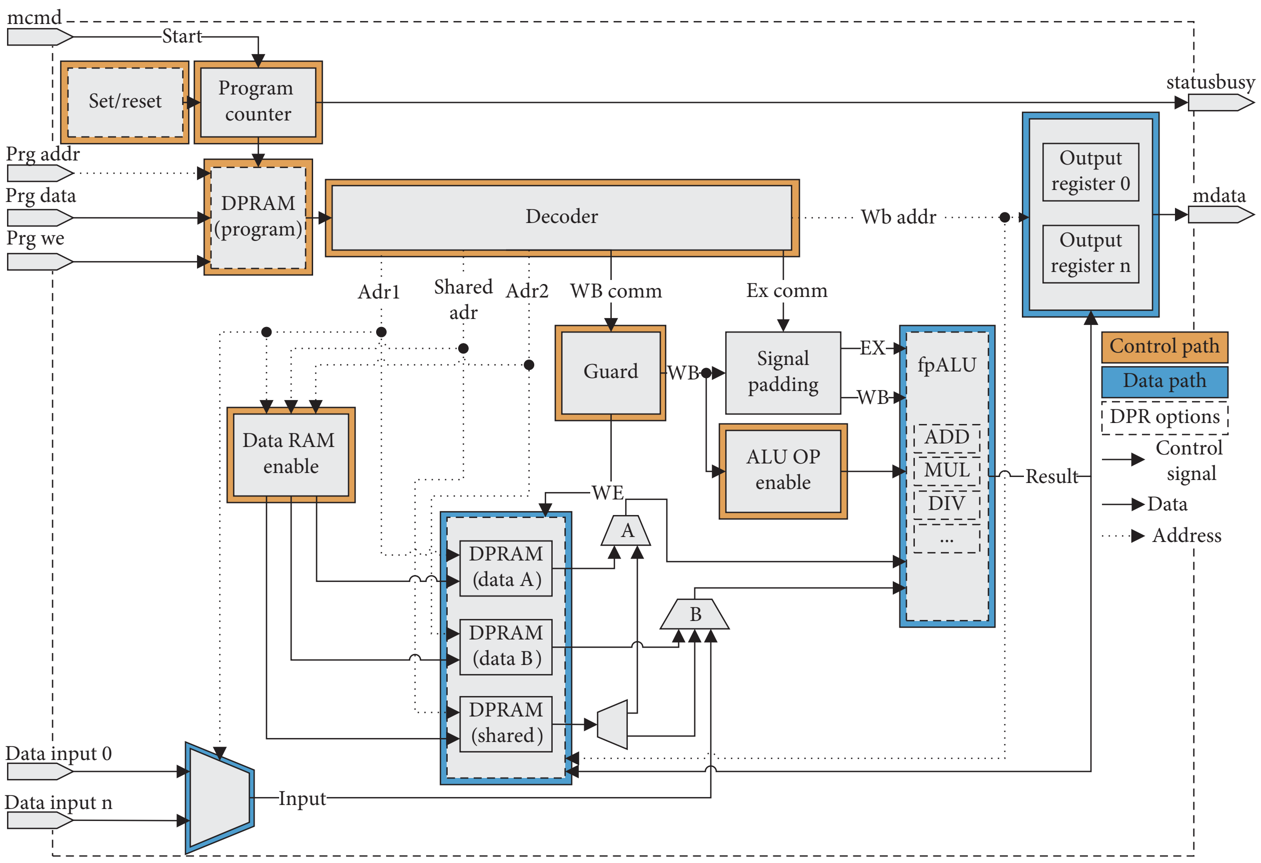


FIGURE 3: Schematic of one core of the ViSARD multi-soft-core processor (based on [30]).

module. With this architecture, it is possible to read and write each memory module in every clock cycle. Because the two memory modules are identical at any time, the soft-core is able to read up to two arguments for an EU and therefore is able to start a new operation in each clock cycle. Additionally, a result can be stored in each clock cycle. External inputs can be written to the memory via the data path using corresponding data input ports. Since these must be addressed by an assembly instruction, the data path goes through the ALU. The shared memory is only used in the case of a multi-soft-core configuration of the ViSARD and is not identical to the processor-internal memory. Due to the architecture, a maximum of one argument per clock cycle can be replaced. If none of the two required arguments are available in the processor's internal memory, the content of one of the needed memory cells is copied from the shared memory into the processor's internal memory with the help of a preceding instruction (inserted at design time), and the actual operation is started afterwards. The last module of the data path is the output module. It contains a (at design time) defined number of output registers and is used to pass the results of the soft-core to the surrounding logic. The output registers can be set independently using an assembly instruction.

As depicted in Figure 3, the architecture of the soft-core processor is designed for modular application-specific adaptation. The functionality of the ALU can be specially adapted according to the task, and multicore architectures can be realized with the *shared* memory. The fully modular design also makes it possible to exchange individual modules over time using the DPR approach. The exchange of modules using DPR is called time multiplexing, and necessary adaptations are described in Section 5.3.

5.1. Hardware Loop and Jump Instructions. One of the main differences between the ViSARD and general soft-core processors is the hard real-time capability. This characteristic is derived from the field of application in which the processor has to operate. The processor is used in domains in which real-time constraints must be met, such as control loops or real-time image processing. As previously mentioned in the introduction, a very good real-world example to clarify the hard real-time demands is driver assistance systems like AutoVision [1]. In this system, a camera records a video with 31 frames per second. This means that the hardware has to process 31 pictures in a second, resulting in a total time frame per picture of about 32.26 ms. If such a picture is comprised of, for example, 1024×1024 pixels, the soft-core has a deadline of 30.76 ns for each pixel.

In order to ensure this real-time characteristic, some adjustments were needed both in the assembly code and in the architecture:

The use of conditional jump instructions in the assembly code is prohibited and thus not supported by the processor. However, this limitation is an insignificant disadvantage in the addressed domain, since the tasks realized in this domain are fixed at design time, and mechanisms can be implemented to replace the corresponding jumps without showing a negative effect.

When programming at higher abstraction levels, for example, at model level, only loops with a fixed number of iterations may be used. These can then be unrolled by a compiler or assembler at design time. Since all information about these loops (e.g., how many iterations it has to perform) is available during design time, every loop can be unrolled, and thus no conditional jumps are needed. This information is available during design time because the algorithm to be realized does not change over time (e.g., control loop tasks) and consequently has a constant number of iterations for each loop. To avoid overly long assembly code and to avoid unrolling every loop in the assembly code, a hardware loop was implemented in the ViSARD. This loop is controlled by the set/reset module. The program counter is set to a certain value at fixed clock cycles and thus jumps to the corresponding position in the code, realizing hardware loops. Furthermore, the scheduling is performed at design time by an optimizing assembler [30, 31] and not at run time by the soft-core itself. Every time a loop is not unrolled and a hardware loop is realized, the information of where the loop starts (program counter), where it ends, and how often it should be run are provided by the optimizing assembler to the set/reset module during design time. The module is therefore able to realize these jumps during run time.

With these novel mechanisms, it is possible to perform a cycle-accurate timing analysis of the processor at design time and to determine the exact computing time, thereby ensuring compliance with the hard real-time barrier.

5.2. The ViSARD Tool-Chain. To be able to use the soft-core efficiently within the scope of a new embedded system, a processing chain, from now on referred to as the tool-chain, is necessary. This tool-chain will be briefly introduced in the following section. In every new project, *requirements* and constraints have to be defined, see Figure 4. With this input, it is possible to describe the target application that needs to execute an embedded algorithm with regard to the hard real-time characteristics and massive parallelism requirements. The *application* sets a scenario, and the user can model the problem with the first part of the tool-chain, the *model-based assembly code generator*, as can be seen in Figure 4. This MATLAB/Simulink-based tool, which was already published in [32], gives the user the ability to realize any given algorithm without special knowledge of any programming language. The user simply drags and drops blocks that realize the needed functionality and connects them as desired. With the help of this tool, it is possible to realize any algorithm as a data-flow model. As soon as the user-defined model-based algorithm is finished, it is possible to automatically generate the special *assembly code* needed to run the ViSARD soft-core. In addition, different optimizations can be used to optimize the graph generated by the model (or even the model itself) by replacing blocks with faster equivalent logic or removing dead parts in the model, resulting in a shorter assembly code. It is also possible to minimize the usage of variables of the generated assembly code. This minimization is necessary in order to create a suitable assembly code for further processing. For detailed information on all optimizations that are performed by the *model-based assembly code generator*, refer to [32].

After the *assembly code* is generated, it is then compiled using an *optimizing assembler*. This assembler translates any given assembly code to *machine code* for the soft-core. In the course of this translation process, an optimized scheduling is processed that makes optimal use of the ViSARD-internal pipelines and thus minimizes the computing time. The assembler also determines the time points required for setting up the hardware loop. In case of a multicore setup such as a dual core, the assembler must be provided all information on which EUs are available on which cores. Additionally, the assembler must know when any core may be reconfigured and what EUs are available before or after the reconfiguration. With this information, it can determine an optimized scheduling for each core, maximizing the pipeline usage and thus minimizing the execution time, while maintaining full design time analyzability.

Based on the *requirements* of the *application* and the operations used in the assembly code, the ViSARD *EU library* is adapted to the given task. The resulting ViSARD

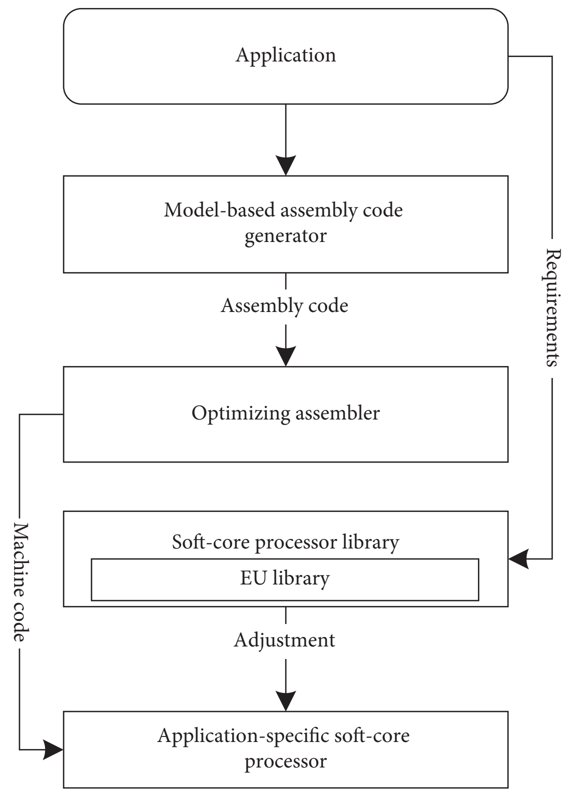

Application
Model-based assembly code generator
Assembly code
Optimizing assembler
Soft-core processor library
EU library
Adjustment
Application-specific soft-core processor
Requirements
Machine code

Figure 4: ViSARD tool-chain (simplified, based on [31]).

incorporates only the required EU modules, resulting in a minimized *application-specific soft-core*.

5.3. Adaptation of the ViSARD to DPR. To use the full potential of FPGAs, a method for the dynamic partial reconfiguration (DPR) of the ViSARD soft-core has to be developed. For this purpose, adjustments had to be made in the architecture of the processor, which will be discussed briefly in this section. The reason why DPR as an optimization possibility is very well suited to the addressed domain is the hard real-time deadline itself. A massive time shortfall of this deadline does not bring any advantages, because real-time systems generally have to be only as fast as absolutely necessary and not as fast as possible. By time-division multiplexing EUs or even entire soft-core processors in the time domain, the computing time is potentially increased, but a large amount of resources can be saved on the FPGA. A reduction of the FPGA resources required by the processor always results a profit in contrast to the faster processing of the algorithm. In some cases, even a smaller and therefore cheaper FPGA can be used with this method.

As the ViSARD was not initially designed with DPR in mind, system and component reset and enable signals had to be included to deal with startup in an undefined state after a reconfiguration. This also ensured that no spurious writes to the memory or state changes occur while the core or EUs are being exchanged. Furthermore, the ability to accommodate multiple different programs in the internal memories (program and data) had to be added. Finally, as all EUs that are to be exchanged have to be contained in one reconfigurable partition, the corresponding calculation result inputs of the ALU result multiplexer were connected to this RP instead of to the default static versions of the EUs.

Compared to solutions like the i-Core [4], which has a static instruction set (IS) as well as a dynamically reconfigurable IS, the ViSARD is not limited to those reconfiguration choices. This is because it is possible to change all EUs or even more (e.g., one complete core including all EUs and memory). Therefore, every EU and the respective assembly instructions can be seen as dynamic. Another prime difference is the fact that the ViSARD has to fulfill hard real-time constraints, and in order to do so, the processor behavior as well as the assembly code must be fully analyzable at design time. Because of this fact, it is necessary to define all reconfiguration points during design time, and it is not mandatory to predefine static and dynamic IS parts like the i-core does. It is also possible to even reconfigure complete cores in a multicore setup, resulting in a complete change of IS in that specific core. Since the task domain the soft-core is designed for does not have requirements that change over time, it is possible to make all the mentioned decisions at design time. It has to be mentioned that it is not possible to realize different reconfiguration granularities at the same time (and therefore realize DPR regions inside coarser DPR regions). During design time and with the help of the estimations of the 1, which is presented in the following section in detail, the user has to decide which reconfiguration granularity is the best fit for the concrete problem. This reconfiguration granularity determines the DPR regions that will be realized inside the soft-core architecture.

6. Reconfiguration Method

During the development of an embedded system, it must be assessed whether the use of DPR will be appropriate in a specific project. In a further step, it has to be decided which parts could be reconfigured. With the help of the method presented below, the timing-related costs T (in μs) of partial reconfiguration can be estimated.

$$T = \frac{H + n \cdot W + m \cdot F + E}{S}, \qquad (1)$$

where H is the size of the header, $n \cdot W$ is the number of write commands times the size of one command, $m \cdot F$ is the number of configuration frames times the size of one command, E is the size of the end sequence, and S is the actual speed of the reconfiguration port in bit μs^{-1}. It should be noted that with every new clock region, another write command is needed. As an example, the necessary values from equation (1) for the Zynq-7000 FPGA family are given in Table 1. They were determined by inspecting partial bitstreams generated by the Vivado design suite.

T is the minimum time required for the planned reconfiguration. With this lower limit and the estimated computation time of the given algorithm, it is possible to assess whether the planned partial reconfiguration would violate the limits of the maximum computing and reconfiguration times of the hard real-time deadline. If the

Table 1: Values of the components.

Symbol	Meaning	Value
H	Header	960 bit
W	Write command	256 bit
E	End sequence	736 bit
F	Configuration frame	3,232 bit
S	Speed	400 bit·μs^{-1}

evaluation is positive that a partial reconfiguration can be used and is beneficial, there are different reconfiguration granularities, i.e., different levels on which reconfigurations can be performed. This estimation has to be done manually at this time, but with this formula it should be possible to implement an automatized reconfiguration checker.

In the literature, there is only a very general distinction between fine-grained and coarse-grained reconfiguration. Fine-grained means a reconfiguration of processing elements working at bit level, and coarse-grained describes the replacement of complex functional blocks such as an ALU [33].

In the context of the reconfiguration of soft-core processors, a more refined classification is necessary. Therefore, a distinction should be made between the different modules and module levels as presented in Figure 5.

A distinction is made based on the granularity of the modules. Individual EU modules within the ALU can be reconfigured, which corresponds to the finest granularity. Swapping an EU on the hardware level is equivalent to changing the availability of assembly instructions on the software level. Because of the time a reconfiguration of different EUs would take compared to the time a typical assembly code would need to be computed, it is not useful to realize this reconfiguration granularity during an active computation period of the soft-core processor. Rather, it is meant to reconfigure the soft-core between two computation runs to adapt the processor to different algorithms that require different EU allocations.

The coarser steps would replace the entire ALU as a module or even entire soft-core processors. On the next reconfiguration granularity, single cores of a multicore architecture can be reconfigured as a complete module. The reconfiguration of one of the cores of a multicore architecture or even the complete soft-core processor can be associated with a function and program swap. So, with the use of this granularity, the DPR region would implement a separate program to be executed. The coarsest level of granularity is the reconfiguration of entire multicore soft-core processors in a MIMD processor.

Coarser granularities will naturally lead to a higher amount of identical logic in the dynamic region of the design as for example the control units of the processors are the same or very similar, depending on the configuration of the soft-core. However, torCombitgen can reduce the influence of this duplication, and the bigger reconfigurable regions can reduce the resource waste incurred as a result of DPR. In Section 8, we present experimental results for the finest granularity (exchanging EUs) and a medium granularity of one soft-core processor. Additional research is needed to

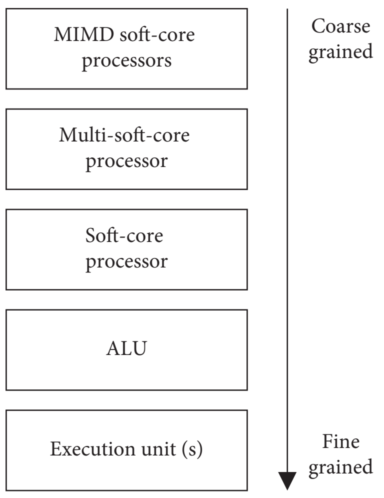


Figure 5: Soft-core-based reconfiguration granularities.

evaluate which granularity level delivers the best compromise between logic utilization and reconfiguration time.

7. DPR System

Partial reconfiguration without an external controller requires additional logic in the form of a partial reconfiguration controller (PRC), and control logic is required on the FPGA. Furthermore, a DPR system needs memory for storing partial bitstreams. This section will explain the system architecture and components.

The resource requirements of all additional components required for DPR must be kept to a minimum in order to ensure the advantage of DPR of minimized space requirement. As the analysis of the available PRCs from the literature in Section 3 has shown, there is no PRC that satisfies the requirements of the architecture (low overhead, predictable timing satisfying hard real-time constraints, throughput close to device limits, designed for standalone operation without general-purpose CPU). For this reason, an optimized PRC was realized and is presented in detail hereafter.

7.1. Memory Solution. Choosing a bitstream storage location that is sufficiently large and fast is key to achieving high reconfiguration throughput.

Block RAM (BRAM) is internal memory distributed in the fabric of Xilinx FPGAs that can be initialized with data specified at design time and is guaranteed to have no access latency. However, BRAM is a comparatively scarce resource in most FPGAs. A full bitstream for the device used in our projects requires about 4.05 MB of storage in contrast to the available 0.6125 MB of BRAM on the FPGA [2], so a partial bitstream for 10% of the resources already exceeds the available space. In any case, bitstream storage in BRAM incurs considerable overhead, occupying valuable resources on the FPGA that might be required for the actual application.

The only alternative is to use external memory. Out of the storage chips available on the Trenz Electronic GigaZee platform used in this research (QSPI flash, eMMC, and

DDR3 SDRAM), only the DDR RAM offers sufficient bandwidth exceeding the 400 MB/s required for sustained partial reconfiguration. It is connected to the processing system (PS) memory controller, so in order to access it from the programmable logic (PL), one of the PS AXI buses has to be used. When choosing the high-performance AXI_HP bus as done here, the maximum read bandwidth is 1,200 MB/s at a bus clock of 150 MHz [14]. Timing-wise, a maximum latency for memory accesses can be guaranteed by making sure that no other components in the SoC (such as the APU or the DMA controller) use the DDR3 SDRAM or the AXI_HP access is given highest priority [14].

Due to the volatile nature of the external RAM, the bitstreams cannot be saved permanently. Instead, they have to be copied from some other persistent storage device on system startup. The eMMC is considered most appropriate for this purpose since it does not contain any other data required for the system to function as opposed to the QSPI flash which contains the Zynq boot loader. Still, the on-chip SD/MMC controller is not directly accessible to the logic on the FPGA. In order to copy the bitstream data to the DDR3 SDRAM, a software routine was implemented that runs once on the APU when the system is initialized and after that turns inert. This component is referred to as *bin provider*. The complete structure of the developed system is illustrated in Figure 6. In addition to the already introduced components, the application-specific *execution control* unit is responsible for initiating and monitoring both DPR and the execution of programs on the ViSARD. Typically, the execution control will observe activity of the ViSARD, and once a given program has finished execution (this is indicated by the ViSARD), stop the processor and request reconfiguration from the PRC. Once the PRC confirms that the bitstream has been loaded, the execution control may restart the processor with a different program matching the new set of EUs. Which programs require which EUs (and therefore partial bitstreams) is specific to the application being implemented. More complicated setups that, e.g., evaluate external input to decide which program to run, are also possible.

To be adaptable to a wide range of DPR-enabled FPGAs, the design of this system is conceived to be as independent of the individual hardware components as possible under the constraint that it can function on the target hardware of this research. Instead of the eMMC, any other memory that is reachable to the APU can be easily substituted in *bin provider*. Even if the target device is not an SoC, the PRC does not have to be modified due to its capability of loading bitstreams from any source via the AXI bus. It can be connected to a memory controller instantiated in the FPGA design instead of using the one in the PS.

7.2. Partial Reconfiguration Controller. The second key component that determines the characteristics of the DPR system such as the configuration data throughput is the PRC itself. It has to be designed so that it can keep up with the data flow and stream it to the ICAP at maximum speed.

The ICAP takes 32 bits of data in each clock cycle at a maximum frequency of 100 MHz. For simplicity, the AXI bus uses the same 100 MHz clock. The AXI_HP bus of the Zynq-7000 uses a data width of 64 bits, so for each cycle twice the amount of data needed for the reconfiguration is read. This allows for a considerable amount of leeway for delivering the bitstream to the ICAP. Nevertheless, some sort of width adaption/buffering between AXI and ICAP is necessary. Depending on the concrete bitstream provider on the AXI bus, there might be a few clock cycles of inactivity between successive read transactions (this is the case for the DDR memory controller). A first-in-first-out (FIFO) buffer with a write port width of 64 bits and a read port width of 32 bits allows using this period of inactivity for bitstream delivery. For low resource overhead, the minimum possible buffer depth of 16 64-bit units was chosen.

The basic structure of the custom PRC implemented in VHDL is shown in Figure 7, which also presents its interfaces (partial reconfiguration control to the right of the PRC, bitstream storage above the PRC, and ICAP below the PRC). Additionally, there are clock and reset inputs (not depicted). The individual components of the PRC are explained further below.

One of the advantages of the custom PRC over similar approaches presented in Section 3 is the usage of the modern and standard AXI bus to achieve separation of bitstream access and delivery. It enables adaption to different system requirements with less effort by allowing to replace the storage module with another ready-made one. VHDL generics allow the specification of parameters of the AXI bus (address width, database address) as well as the number of different bitstreams to be loaded.

Partial bitstreams can differ in size depending on the options used to create them and the resources the logic occupies. One of the novel contributions of this paper is a method and tool for delivering partial bitstreams minimized in size and number with the torCombitgen tool as decribed in Section 4. For the PRC to be able to access and load bitstreams quickly, all of them must be available in a contiguous region of memory and additionally have to be indexed to allow locating the start address of an arbitrary bitstream without performing multiple look-ups or skipping through other bitstreams in order to find the correct one. The index format used is a plain table of all available bitstreams that starts at offset 0 to the AXI database address. Each bitstream gets one 8-byte entry consisting of two 4-byte unsigned integer values: the offset of the first byte of the bitstream and its size in bytes. All entries are stored successively, enabling instant lookup of the values of one bitstream by loading 8 bytes of data from the base address plus eight times the index of the desired bitstream. The actual bitstreams are stored without any postprocessing and likewise in succession. Compared to the Xilinx PRC, this technique enables testing bitstreams of different sizes (e.g., generated with different methods and/or settings) without having to perform any modifications to the PRC configuration or the implemented netlist.

The PRC contains three state machines, each responsible for one channel of information: the AXI address channel, the AXI data channel, and the ICAP channel (see Figure 7). By treating the two AXI channels of information separately, it is possible to make the best use of the bandwidth and burst read capability of the DDR memory controller. Although the

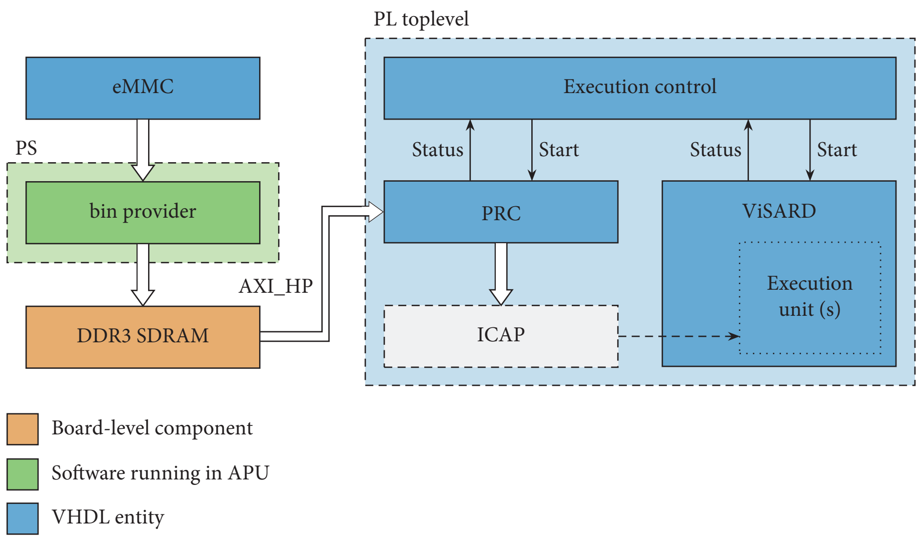


Figure 6: Structure of the custom DPR system.

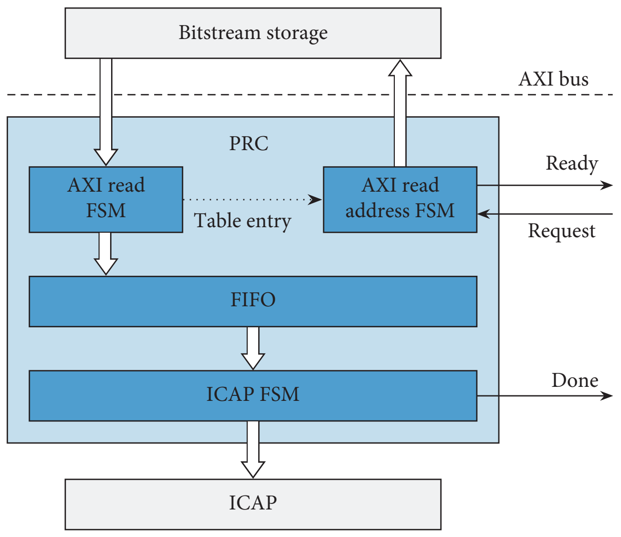


Figure 7: Block diagram of the custom PRC and its interfaces (excluding clock/reset).

memory controller requires a number of cycles of latency to answer read requests, it continuously streams out data once it begins to send the response. In particular, further reads can be issued while the response to a previous request is still pending. The PRC makes sure to always keep as many read requests as possible in the queue of pending AXI transactions. The latency of memory read requests does not impact the bitstream throughput beyond the start of the partial reconfiguration. When a reconfiguration with a particular partial bitstream is requested, first the entry in the bitstream table as described above needs to be read. Only when the table entry information is available, it is possible to begin issuing AXI read commands for the actual bitstream data, as before that the address to read the bitstream from is unknown.

The user-facing control interface is very simple to use: When the PRC is ready to accept a request for reconfiguration, it indicates this via the "ready" output port. The application may start such a request using the "request" input and communicate which partial bitstream to load using the corresponding bitstream table index. When reconfiguration has been completed, the "done" output is asserted for the duration of one clock cycle. This is guaranteed to happen only after all data was written to the ICAP, which in turn guarantees that the reconfigured region is ready to be used when the application receives this information. The "ready" output is asserted again whenever the PRC is able to begin processing another request. Specifically, this might be the case when a reconfiguration that is currently underway is nearing its end, so that another partial bitstream can already begin preloading for reduced latency between successive reconfiguration cycles.

The implementation of the PRC has predictable timing characteristics and satisfies the hard real-time requirement. Given the clock frequency f (typically 100 MHz), a partial bitstream of n bytes, and an AXI slave that responds to requests in at most d clock cycles and can sustain the required configuration throughput (i.e., answers with new data before the FIFO runs empty), the upper bound of the time required for a reconfiguration cycle is

$$T_{\mathrm{r}} = \frac{3 + 2 \cdot d + (n/4)}{f}. \tag{2}$$

This equation is derived from a small number of cycles of initial latency needed to communicate between the state machines and issue AXI requests, the time these requests take to be answered, and finally the duration of the transfer of the bitstream data to the ICAP. As described above, two unpipelined AXI requests need to be answered before data is streamed continuously (which is why d is multiplied with 2): Firstly, the read of the bitstream table entry, and secondly, the read of the first bitstream data block.

8. Experimental Results

A number of experiments were carried out on a Xilinx Z-7020 SoC to evaluate the benefits of DPR for the ViSARD and verify the feasibility of the DPR system presented in Section 7. This section explains two of them in detail and presents what results were achieved in terms of resource savings, reconfiguration speed, and bitstream sizes. Other experiments are given as summary.

As first step, the operation of the PRC and its interaction with the AXI bus were verified in behavioral simulation. The whole DPR system (including the PRC, software on the APU, and all utilities) was tested by continuously exchanging one execution unit in the ViSARD.

The following experiment consisted of one ViSARD core embedded in the DPR system (Figure 6) using a basic execution control module that continually cycles through three different EU configurations and ViSARD test programs in the following order (wrapping back to the first configuration after the last one):

(1) Divide

(2) Square root (Sqrt) and sine/cosine (SinCos)

(3) Sqrt, natural exponential function (NatExp) and multiply

This experiment represents the finest-grained reconfiguration granularity, as execution units are being reconfigured (Figure 5). It is adopted from the results presented in [34]. The configurations were chosen such that they differ in the number of EUs (to verify that changing not only the EUs itself but also the amount of EUs in an RP is possible) and have a compatible resource footprint. Even so, the NatExp EU is the only EU that uses BRAM resources, which means that the resource Pblock must include BRAM which is unused in configurations 1 and 2.

The soft-core processor is stopped during the reconfiguration. It is possible that all parts of the processor that are not reconfigured remain operational during the reconfiguration, but for the sake of simplicity of the experiments this was not applied. If those parts stay operational, the assembler will ensure that the processor will not access those parts during the reconfiguration. Due to the time that the reconfiguration occupies, this scenario makes sense only with higher reconfiguration granularities. An example for this would be a multi-soft-core scenario, where one core is reconfigured while all other cores stay operational.

Running the experiment on hardware revealed that the ViSARD produced correct results in all calculations, thereby verifying that partial bitstreams are forwarded to the FPGA and the DPR system works.

The speed of reconfiguration was measured by capturing a histogram of the number of clock cycles it took the DPR system to fully deliver one partial bitstream, i.e., switch from any EU configuration to the next. As shown in Table 2, the average cycle count in 65,536 continuous measurements was 61,833.7 with a standard deviation (SD) of 13.7 cycles, allowing for $(618.337\,\mu s)^{-1} = 1617.24$ reconfigurations per second. The average bitstream throughput was 247, 116 bytes/ 618.337 Âμs = 399.646 MB/s or 99.91% of the 400 MB/s maximum.

"Transferred to the real-world example AutoVision [1], reconfiguration would be triggered, e.g., when the car drives into a tunnel. A picture has to be computed in a total time frame per picture of 32.25 ms. As one reconfiguration run would take less than 0.62 ms, the soft-core processor would be left with 31.63 ms for the actual image processing. This reconfiguration period would take less than 2% of the available time frame. The time frame the soft-core processor has for each pixel is slightly reduced from 30.75 ns to 30.16 ns after the DPR is completed, but it has the advantage of reduced resource usage, thanks to DPR."

As the latency of the PRC is completely deterministic and static (see Section 7.2), the variance is the result of the Zynq DDR memory controller having different access latencies depending on when the request is made. This is a known property of DDR memory accesses caused mainly by refresh cycles that must be interleaved with data transfer. As can be seen from both past research projects [35] and the results here (30 cycles of variance between fastest and slowest execution of an operation that takes 61,834 cycles on average), its effect is minimal. It is therefore not considered further. In summary, these results show that the PRC is able to deliver hard real-time reconfiguration in practice with the limitation of assuming a bitstream provider that can satisfy this constraint as well.

The resources used by the design and its components are listed in Table 3.

For comparison purposes, an equivalent non-DPR design was included that implements the same functionality but does not have a reconfigurable partition. It uses identical IP cores and settings, ViSARD test program, and execution control unit. When correlating the DPR and the non-DPR design, it is not immediately obvious what numbers should be examined. Manufacturer tools provide usage broken down into components, but additional considerations have to be taken into account. The DPR design has resources in the Pblock(s) designated for reconfiguration that are blocked and not usable for implementing additional logic but do not appear in those numbers. Therefore, we take the static part of the DPR design and add the resources allocated for reconfiguration to it.

Applying this methodology, DPR allowed to save around 3,500 LUTs and 7,000 FFs. This is a very good result that amounts to a relative saving of 40.7% when directly comparing full DPR and non-DPR designs. Both BRAM and DSP resources showed an increase (8.5 and 5 additional tiles, respectively) mainly due to BRAM/DSP functional units that had to be allocated to the DPR Pblock but could not be used by the dynamic logic. Realistically, these dormant resources cannot be completely avoided when dealing with diverse execution units and are considered to be the cost of partial reconfiguration. However, since these resources are not needed much within processors in the application domain and the LUTs are the critical resource, this small increase is negligible.

Of the PRCs presented for comparison in Section 3, usage figures on Kintex-7-based FPGAs are available for the RT-ICAP, the Xilinx PRC, and the ZyCAP. They are compared in Table 4. Our PRC requires about 270 LUTs plus one BRAM tile for the FIFO, which is less than 1% of the resources available on the device in all categories. It is small

Table 2: Statistics of clock cycle count required for DPR in the EU-swapping experiment.

Minimum	Mode	Average (SD)	Maximum
61,824	61,824	61,833.7 (13.7)	61,854

Table 3: Resource utilization in the EU-swapping experiment.

	Component	LUT	FF	BRAM	DSP
(a)	Divide RM	3,273	6,546	0.0	0
(b)	Sqrt-Sin Cos RM	3,063	6,126	0.0	0
(c)	Sqrt-NatExp-Multiply RM	3,055	6,110	2.5	25
(d)	DPR design (complete with Divide RM)	4,575	9,150	4.0	3
(e)	DPR design (static part): (d) − (a)	1,302	2,604	4.0	3
(f)	Pblock for DPR	3,800	7,600	10.0	30
(g)	DPR design (effective): (e) + (f)	5,102	10,204	14.0	33
(h)	Equivalent non-DPR design	8,607	17,214	5.5	28
(i)	DPR vs. non-DPR: (g) − (h)	−3,505	−7,010	+8.5	+5

Table 4: Resource utilization and configuration throughput of the custom and other PRCs.

Controller	LUT	FF	BRAM	Throughput (MB/s)
Custom PRC	273	292	1.0	399.981
RT-ICAP [7]	245	101	0.0	382.2
Xilinx PRC [21]	897	954	0.5	n/a
ZyCAP [13]	620	806	0.0	382

Table 5: Bitstream sizes in the ViSARD-swapping experiment.

Method	Bitstream size (bytes)					
	NatExp		Sqrt		SinCos	
Module-based	191,784	**(100.00%)**	191,784	**(100.00%)**	191,784	**(100.00%)**
Difference-based	144,076	(75.12%)	140,440	(73.23%)	144,076	(75.12%)
torCombitgen	144,060	(75.12%)	144,060	(75.12%)	144,060	(75.12%)

and does not constitute considerable overhead. The sustained throughput of 399.981 MB/s or 99.995% of the device maximum was measured in another experiment with two ViSARD cores that allowed for overlapping bitstream delivery to the ICAP with buffering of the next bitstream to load. Data for the Xilinx PRC was determined in a test design (settings were chosen to be most similar to the custom PRC: no optional features (including AXI control interface), minimum FIFO depth of 32 implemented as BRAM, 4 clock domain crossing stages, and 1 virtual socket with 2 modules with 1 hardware trigger each). The RT-ICAP is competitive in terms of resource usage, but it was conceived for a different purpose, requires a supporting general-purpose CPU, and does not have an AXI memory interface. It additionally requires memory for communication with the CPU that is not represented in the reported BRAM usage. Furthermore, its throughput is around 4% below the result of this research. Vastly more logic resources are needed by the ZyCAP (more than twice) and the Xilinx PRC (more than three times). Conceptually, the proposed PRC is quite similar to a design proposed by Vipin [20] that was implemented on prior-generation FPGA hardware. However, it does not consider hard real-time requirements and uses a directly integrated DDR memory interface as opposed to the standard AXI bus, which means that the bitstream storage cannot easily be replaced.

In the following experiment that applies the same idea at the coarser granularity level of a whole soft-core processor, the principle was kept as above and the reconfigurable partition extended to cover the whole ViSARD core. To keep it simple, only one execution unit was inserted per core (NatExp, Sqrt, and SinCos). Table 5 lists the sizes of bitstreams generated by different methods. A module-based generation always produces the largest bitstream, so it is taken as reference. Difference bitstreams were generated only for switching from one configuration to the next as required by this experiment, reducing the size by about 24% on average. All or dynamic change patterns would require $n^2 - n = 3^2 - 3 = 6$ partial bitstreams as previously explained in Section 4.1, but their sizes will not differ significantly. torCombitgen similarly allowed for a 25% reduction in size, but the three produced bitstreams are already sufficient for switching from any configuration to any other one. The results confirm the validity of this approach and that the sizes of bitstreams generated by this tool lie between the module-based and the difference-based method.

9. Conclusion and Future Work

The ViSARD is a hard real-time capable soft-core processor that can be adapted for any application in the addressed domain. With the help of the accompanying tool-chain, it is possible to adapt the processor to the respective project in a model-based fashion. The novel mechanisms derived from the method and realized in the soft-core enable a cycle-accurate timing analysis of the processor at design time and determine the exact computing time, thereby ensuring compliance with the hard real-time barrier. Those mechanisms also allow the application-specific specialization and eliminate any disadvantage resulting from the hard real-time constraints. The soft-core was extended to DPR with respect to the hard real-time capability. The torCombitgen tool was developed based on the Combitgen tool, as part of the extension of the soft-core to DPR. This tool can be used to create minimized partial bitstreams for any DPR scenario and combines the advantages of the module-based and difference-based bitstream generation approaches.

The combination of torCombitgen and ViSARD enabled the implementation of a soft-core processor that adapts to a changing environment over time. This maximizes the suitability of the processor for a wide range of tasks and at the same time minimizes the necessary FPGA resources. The conducted experiments have shown that even with a reconfiguration on the finest granularity level, more than 40% of the critical resources can be saved while preserving identical resulting functionality. Considering that the size of the Pblock can be adapted more precisely to the respective problem with a larger area to be reconfigured, i.e., the overhead is reduced, even better results can be achieved with coarser granularities.

Within the scope of the performed experiments, the hard real-time capability of the used DPR approach (torCombitgen-generated partial bitstream and custom PRC) could also be demonstrated. The implemented PRC uses minimal FPGA resources and achieves the best possible reconfiguration speed compared to other PRCs in the literature (see Table 4).

In the future, the approach presented and all developed tools will be applied in real-world scenarios and compared to existing non-DPR solutions. Additional research is needed to evaluate which granularity level delivers the best compromise between logic utilization and reconfiguration time. With these results, it would be conceivable to integrate DPR into the ViSARD tool-chain as an automated step during the model-based assembly code generation. An essential part of this integration will be Equation (1). With this estimation, an automatized algorithm can be implemented that finds the possible reconfiguration options. This reconfiguration checker will also handle timing considerations arising from reconfiguring active soft-core processors and provide this information to the assembler. Currently, this has to be done manually by the user during design time.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

We acknowledge support for the article processing charge by the German Research Foundation (DFG) and the Open Access Publication Fund of the Technische Universität Ilmenau.

References

[1] C. Claus, S. Walter, and A. Herkersdorf, "Autovision—a run-time reconfigurable MPSoC architecture for future driver assistance systems (Autovision—eine zur Laufzeit rekonfigurierbare MPSoC-Architektur für zukünftige Fahrerassistenzsysteme)," *IT-Information Technology*, vol. 49, no. 3, pp. 181–187, 2007.

[2] Xilinx, Inc., *Zynq-7000 SoC Data Sheet: Overview. DS190 v1.11.1*, Xilinx, Inc., San Jose, CA, USA, 2018, https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[3] I. Zaidi, A. Nabina, C. N. Canagarajah, and J. Nunez-Yanez, "Evaluating dynamic partial reconfiguration in the integer pipeline of a FPGA-based opensource processor," in *Proceedings of the 2008 International Conference on Field Programmable Logic and Applications*, pp. 547–550, IEEE, Heidelberg, Germany, September 2008.

[4] J. Henkel, L. Bauer, M. Hübner, and A. Grudnitsky, "i-Core: a run-time adaptive processor for embedded multi-core systems," in *Proceedings of the 2011 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'11)*, pp. 163–170, Las Vegas, NV, USA, July 2011.

[5] Xilinx, Inc., *MicroBlaze Micro Controller System v3.0 LogiCORE IP Product Guide. PG116*, Xilinx, Inc., San Jose, CA, USA, 2017, https://www.xilinx.com/support/documentation/ip_documentation/microblaze_mcs/v3_0/pg116-microblaze-mcs.pdf.

[6] Xilinx, Inc., *AXI HWICAP v3.0 LogiCORE IP Product Guide. PG134*, Xilinx, Inc., San Jose, CA, USA, 2016, https://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v3_0/pg134-xihwicap.pdf.

[7] L. Pezzarossa, M. Schoeberl, and J. Sparso, "A controller for dynamic partial reconfiguration in FPGA-based real-time systems," in *Proceedings of the 2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 92–100, IEEE, Toronto, Canada, May 2017.

[8] M. Hübner, D. Gohringer, J. Noguera, and J. Becker, "Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs," in *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1–8, IEEE, Atlanta, GA, USA, April 2010.

[9] A. Nabina and J. L. Nunez-Yanez, "Dynamic reconfiguration optimisation with streaming data decompression," in *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, pp. 602–607, Milano, Italy, August 2010.

[10] G. Wang, L. Dongming, W. Fengzhou, A. Adetomi, and T. Arslan, "A tiny and multifunctional ICAP controller for dynamic partial reconfiguration system," in *Proceedings of the 2017 NASA/ESA Conference on Adaptive Hardware and*

Systems (AHS), pp. 71–76, IEEE, Pasadena, CA, USA, July 2017.

[11] L. A. Cardona and C. Ferrer, "AC_ICAP: a flexible high speed ICAP controller," *International Journal of Reconfigurable Computing*, vol. 2015, Article ID 314358, 15 pages, 2015.

[12] S. Liu, R. N. Pittman, A. Forin, and J.-L. Gaudiot, "Minimizing the runtime partial reconfiguration overheads in reconfigurable systems," *The Journal of Supercomputing*, vol. 61, no. 3, pp. 894–911, 2012.

[13] K. Vipin and S. A. Fahmy, "ZyCAP: efficient partial reconfiguration management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 41–44, 2014.

[14] Xilinx, Inc., *Zynq-7000 SoC Technical Reference Manual. UG585 v1.12.2*, Xilinx, Inc., San Jose, CA, USA, 2018, https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[15] J. C. Rodriguez and K. F. Ackermann, "Leveraging partial dynamic reconfiguration on Zynq SoC FPGAs," in *Proceedings of the 2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-On-Chip (ReCoSoC)*, pp. 1–6, IEEE, Montpellier, France, May 2014.

[16] V. Lai and O. Diessel, "ICAP-I: a reusable interface for the internal reconfiguration of Xilinx FPGAs," in *Proceedings of the 2009 International Conference on Field-Programmable Technology*, pp. 357–360, IEEE, Sydney, Australia, December 2009.

[17] S. Bhandari, S. Subbaraman, S. Pujari et al., "High speed dynamic partial reconfiguration for real time multimedia signal processing," in *Proceedings of the 2012 15th Euromicro Conference on Digital System Design*, pp. 319–326, IEEE, Izmir, Turkey, September 2012.

[18] J. Tarrillo, F. A. Escobar, F. L. Kastensmidt, and C. Valderrama, "Dynamic partial reconfiguration manager," in *Proceedings of the 2014 IEEE 5th Latin American Symposium on Circuits and Systems*, pp. 1–4, IEEE, Santiago, Chile, February 2014.

[19] C. Beckhoff, D. Koch, and J. Torresen, "Portable module relocation and bitstream compression for Xilinx FPGAs," in *Proceedings of the 2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, IEEE, Munich, Germany, September 2014.

[20] K. Vipin and S. A. Fahmy, "A high speed open source controller for FPGA partial reconfiguration," in *Proceedings of the 2012 International Conference on Field-Programmable Technology*, pp. 61–66, Seoul, South Korea, December 2012.

[21] Xilinx, Inc., *Partial Reconfiguration Controller v1.3. PG193*, Xilinx, Inc., San Jose, CA, USA, 2018, https://www.xilinx.com/support/documentation/ip_documentation/prc/v1_3/pg193-partial-reconfiguration-controller.pdf.

[22] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time partial reconfiguration speed investigation and architectural design space exploration," in *Proceedings of the 2009 International Conference on Field Programmable Logic and Applications*, pp. 498–502, IEEE, Prague, Czech Republic, August 2009.

[23] C. Claus, R. Ahmed, F. Altenried, and W. Stechele, "Towards rapid dynamic partial reconfiguration in video-based driver assistance systems," in *Reconfigurable Computing: Architectures, Tools and Applications*, vol. 5992, pp. 55–67, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[24] Xilinx, Inc., *Vivado Design Suite User Guide: Partial Reconfiguration. UG909 v2018.2*, Xilinx, Inc., San Jose, CA, USA, 2018, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug909-vivado-partial-reconfiguration.pdf.

[25] E. Eto, *Difference-Based Partial Reconfiguration. XAPP290 v2.0*, Xilinx, Inc., San Jose, CA, USA, 2007, https://www.xilinx.com/support/documentation/application_notes/xapp290.pdf.

[26] C. Claus, F. H. Müller, and W. Stechele, "Combitgen: a new approach for creating partial bitstreams in virtex-II pro devices," in *Proceedings of the Workshop on Reconfigurable Computing Proceedings (ARCS 06)*, pp. 122–131, Delft, Netherlands, March 2006.

[27] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: towards an open-source tool flow," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays—FPGA'11*, pp. 41–44, ACM, Monterey, CA, USA, 2011.

[28] M. Kirchhoff and W. Fengler, "Realization of an embedded hard realtime softcore processor," in *Proceedings of the 7th GI Workshop—Autonomous Systems 2014*, pp. 33–42, Cala Millor, Spain, 2014.

[29] IEEE Standard for Floating-Point Arithmetic, *IEEE Std 754-2008*, IEEE, Piscataway, NJ, USA, 2008.

[30] M. Kirchhoff, L. Wagnler, and W. Fengler, "Compiler optimization on instruction scheduling for a specialized real-time floating point soft-core processor," *Advances in Electrical and Computer Engineering*, vol. 19, no. 3, pp. 57–68, 2019.

[31] M. Kirchhoff, N. Kaptsova, D. Streitpferdt, and W. Fengler, "Optimizing compiler for a specialized real-time floating point softcore processor," in *Proceedings of the 2017 8th Annual Industrial Automation and Electromechanical Engineering Conference (IEMECON)*, pp. 181–188, Bangkok, Thailand, August 2017.

[32] M. Kirchhoff, J. Weisensee, D. Streitferdt, W. Fengler, and E. Rozova, "Increasing efficiency in data flow oriented model driven software development for softcore processors," in *Proceedings of the 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pp. 806–811, IEEE, Tokyo, Japan, July 2018.

[33] D. Koch, *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*, Springer, Berlin, Germany, 2014.

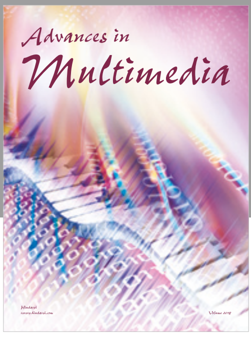
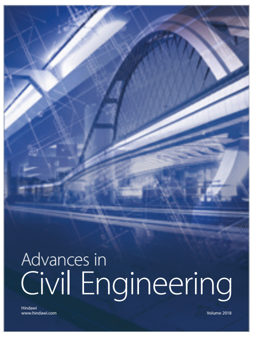
[34] P. Kerling, "Conceptual design and realization of a dynamic partial reconfiguration extension of an existing soft-core processor," Master's thesis, Technische Universität Ilmenau, Ilmenau, Germany, 2019.

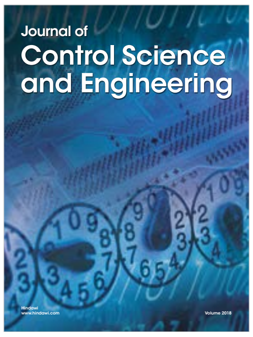
[35] P. Atanassov and P. Puschner, "Impact of DRAM refresh on the execution time of real-time tasks," in *Proceedings of the International Workshop on Application of Reliable Computing and Communication (In Conjunction with PRDC 2001)*, pp. 29–34, Seoul, Korea, December 2001.

Journal of
Engineering

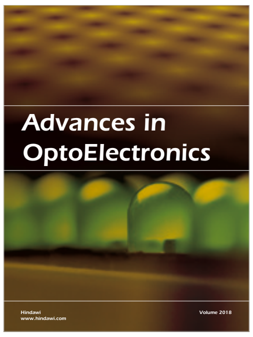
The Scientific World Journal

International Journal of
Rotating Machinery

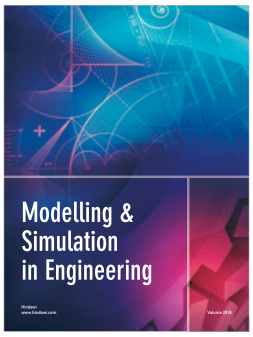
Journal of
Sensors

Advances in
Multimedia

Advances in
Civil Engineering

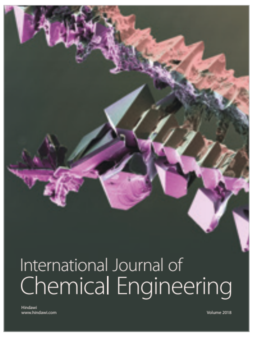
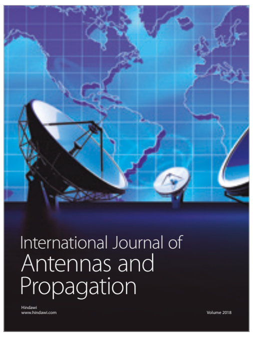

Journal of
Control Science and Engineering

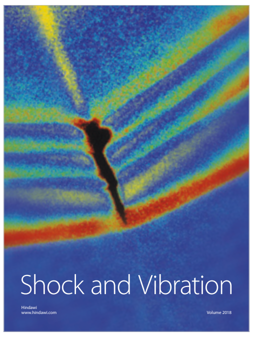
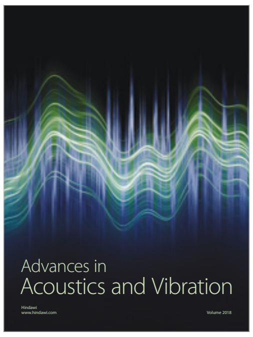
Journal of
Robotics

Journal of
Electrical and Computer Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and Observation

Modelling & Simulation in Engineering

International Journal of
Aerospace Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and Propagation

Active and Passive Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration