Leveraging the Partial Reconfiguration Capability of FPGAs for Processor-Based Fail-Operational Systems

Tobias Dörr, Timo Sandmann, Florian Schade,
Falco K. Bapp, and Jürgen Becker

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{tobias.doerr,sandmann,florian.schade,becker}@kit.edu

Abstract. Processor-based digital systems are increasingly being used in safety-critical environments. To meet the associated safety requirements, these systems are usually characterized by a certain degree of redundancy. This paper proposes a concept to introduce a redundant processor on demand by using the partial reconfiguration capability of modern FPGAs. We describe a possible implementation of this concept and evaluate it experimentally. The evaluation focuses on the fault handling latency and the resource utilization of the design. It shows that an implementation with 32 KiB of local processor memory handles faults within 0.82 ms and, when no fault is present, consumes less than 46 % of the resources that a comparable static design occupies.

Keywords: Fail-operational system · Graceful degradation · Partial reconfiguration · Dynamic redundancy · Simplex architecture · Fallback processor · Multiprocessor system-on-chip · Soft-core processor.

1 Introduction

Digital systems perform a large variety of tasks in a steadily increasing number of applications. Their advance into certain safety-critical realms, such as autonomous driving, imposes stringent dependability requirements on them. In order to meet these requirements, designers need to pay attention to the challenges that current state-of-the-art hardware brings along. At the same time, they often need to achieve their goals with as little redundancy as possible.

A *dependable system* has the property that reliance on its correct functioning is justified [1]. A system is *safe* if it does not endanger humans or the environment [18]. In practice, all electronic systems are at risk of experiencing *faults*. These anomalies or physical defects can lead to situations in which a system is unable to fulfill its desired function [12]. Such a condition is called a *failure* and might, in particular, impair the safety of the considered system.

Certain systems have a so-called *safe state*. It describes a state that can be entered in response to faults and ensures that the system continues to satisfy its safety requirements. At the same time, the actual function of the system becomes unavailable. Such systems are referred to as *fail-safe systems* [18].

A *fail-operational system* needs to maintain a certain minimum level of functionality, even when it is subject to a certain number of faults. Depending on the exact requirements, however, a degraded functionality might be sufficient [10].

Considerable research has been conducted in the field of *fault tolerance*. Fault tolerance techniques try to mitigate faults in a way that the emergence of failures is prevented [1]. They are usually based on some kind of redundancy [12] and play an important role in the design of fail-operational systems.

A known fault tolerance technique that aims at safety-critical systems with fail-operational requirements is the simplex architecture [16,2]. Its general idea is to deal with the complexity of today's control systems by providing an additional controller. This controller is considerably simpler than the main one, able to deliver a functionality that meets all safety requirements of the system, and is disabled during normal operation. As soon as the main controller fails, however, the simple controller is activated and ensures safe but degraded operation.

Motivated by the need for efficient fail-operational systems in the automotive context, [4] builds upon the described concept and adapts it for use on modern and heterogeneous multiprocessor system-on-chips (MPSoCs).

Both the original and the adapted concept assume that some kind of fallback unit, i.e., a plant controller or a processor, is physically available during normal operation of the system. No attempts have yet been made to develop a processor-based simplex architecture in which the fallback processor is introduced on demand, i.e., in response to faults of the main controller.

In this work, we review the concept from [4], derive a motivation for the dynamic provision of the fallback processor, and extend the existing concept accordingly. In addition, we present an implementation of the concept on a commercially-available device, the Zynq UltraScale+ MPSoC from Xilinx. To introduce the processor on demand, our implementation employs partial reconfiguration of the MPSoC's programmable logic. We optimize the design systematically and compare certain figures of merit to those of an equivalent design in which the fallback processor is present at all times.

2 Related Work

Extensive research has been conducted on the partial reconfiguration (PR) of field-programmable gate arrays (FPGAs). A survey that focuses on the performance of a PR process is given by Papadimitriou et al. in [13]. In a more recent work, Vipin and Fahmy [20] present the state of the art in this field and compare the PR performance values of several commercially-available architectures.

A survey of fault tolerance mechanisms for FPGAs is given in [6]. Some of the considered approaches, such as [9], make use of PR to tolerate faults at runtime. These mechanisms have in common that they deal with low-level details of the FPGA architecture to provide fine-grained fault tolerance. The fault tolerance approach described in [5] makes use of partial reconfiguration as well, but acts on coarse-grained logic blocks of an FPGA. All these techniques handle faults of

the programmable logic itself. The approach that we present makes use of the programmable logic to increase the dependability of the overall system.

The techniques described in [14] and [19] employ PR to achieve fault tolerance of soft-core processors in FPGAs. As part of [15], the authors present a similar approach that does not require an external controller to handle the partial reconfiguration. This process is instead performed by a hardened part of the soft-core processor itself. Di Carlo et al. [7] propose a partial reconfiguration controller to perform the partial reconfiguration process in a safe way.

Shreejith et al. [17] react to faults of an electronic control unit's primary function, which is implemented on an FPGA, by performing a switch to a backup function. While the backup function is active and ensures that the safety requirements are met, the primary function is restored using partial reconfiguration.

Ellis [8] considers a network of processors and deals with the dynamic migration of software in response to failing nodes. [3] and [11] focus on processor-based systems and discuss certain aspects of fault-tolerant and fail-operational architectures in the automotive domain. However, neither of the three references deals with the utilization of FPGAs to achieve dependability or fault tolerance.

3 Background and Motivation

The problem that [4] considers can be described as follows: Assume that a given fail-operational system in a safety-critical environment has to perform a certain functionality. It is connected to its surroundings via dedicated interfacing components, such as sensors, actuators, or I/O controllers. Not all aspects of the normally delivered functionality are necessary from a safety perspective. The system comprises the interfacing components, an interconnection network, and a so-called *complex system*. The complex system consists of components that fulfill the actual system functionality. While both the interfacing components and the interconnect are assumed to be dependable, the complex system might be subject to faults that it cannot tolerate. As a result of the aforementioned fail-operational requirements, it must be ensured that such a fault does not lead to a failure of the overall system. Since at least a degraded functionality has to be maintained, suitable fault tolerance techniques must be applied.

To accomplish this in an efficient way, the authors propose a concept we will refer to as the *static simplex architecture*. Figure 1 shows a simplified block schematic of this concept from a logical perspective. A so-called *transaction* represents a communication channel from a transmission initiator (*master*) to a receiver or responder (*slave*). It is assumed that the complex system is able to detect all internal faults that the architecture needs to protect against. It could, for instance, comprise a lockstep processor (to protect against single faults of the CPU) or make use of a watchdog timer. The static simplex architecture defines the mechanism that is triggered after such a fault is detected. In this case, the control entity disables the complex system and enables the *fallback system*. The latter is considerably simpler than the complex system. However, it focuses on and is able to meet the overall system's safety requirements. A set of

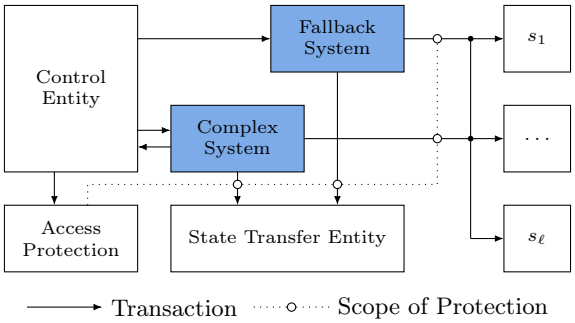

Fig. 1. Logical view of the static simplex architecture

application-specific slave modules, $S = \{s_1, \ldots, s_\ell\}$, is used to model slaves that both the complex and the fallback system need to interact with. One example of such an $s \in S$ is a dependable CAN controller that both the complex and the fallback system share. The overall system is always in one of two possible modes, which are given by $C = \{c_{\text{complex}}, c_{\text{fallback}}\}$ and referred to as *contexts*.

Depending on the active context, an access protection mechanism ensures that the disabled system is logically isolated from the slaves. During context switches, the state transfer entity can be used to transfer consistent snapshots of state variables (such as CPU register values) between the two systems.

It is important to understand that this concept makes use of dynamic redundancy to mitigate faults: If necessary, the essential functions of the complex system are dynamically moved to the fallback system. The reason we refer to this approach as the static simplex architecture is as follows: The fallback system needs to be present at all times, even when the complex system fulfills the functionality of the overall system. This implies a static resource overhead, which could be reduced by providing the fallback system on demand. It is the aim of this work to research and evaluate such an approach.

4 Extension of the Concept

We propose the concept of the *dynamic simplex architecture*. It addresses the same problem as the static simplex architecture and adopts the same general idea to achieve fail-operational behavior. The proposed concept, however, constitutes two distinguishing characteristics: First, it is assumed that the functionality of the fallback system can be implemented on a processor. Second, this processor must be partially reconfigurable on an FPGA that is part of the overall system. By partially reconfigurable we mean that a part of the FPGA can be reconfigured during runtime while the remaining logic continues to operate.

At any point in time, the overall system is in one of two contexts: either in c_{complex} or in c_{fallback}. In the first case, the complex system is enabled and the fallback system is disabled. In the second case, it is vice versa. At any time, the

currently enabled system has access to the application-specific slave modules and the state transfer entity. The disabled system is isolated from these components.

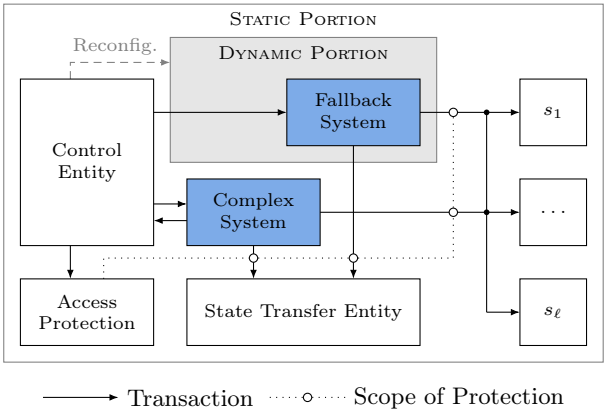

Fig. 2. Logical view of the dynamic simplex architecture

Figure 2 shows a block schematic of this concept from a logical perspective. The depicted *dynamic portion* represents a partially reconfigurable region of the FPGA. The complex system encapsulates a set of arbitrary components that deliver the full functionality of the overall system. It must be able to detect all relevant internal faults and notify the control entity about their occurrence. The fallback system consists of a soft-core processor and occupies the dynamic portion of the FPGA if and only if c_{fallback} is active. If this is not the case, the dynamic portion can be utilized for other purposes. It could, for instance, be used to implement hardware accelerators that perform non-safety-relevant tasks.

c_{complex} is the initially active system context. If faults of the complex system endanger safety, c_{fallback} becomes active. A switch back to c_{complex} is possible if the faults are no longer present. Context switches are orchestrated by the control entity. If a switch is pending, the entity initiates the partial reconfiguration of the FPGA and sets the access permissions in such a way that the disabled system is isolated from the slaves. Adherence to the access permissions is enforced by the visualized access protection mechanism. The state transfer entity provides a certain amount of buffered memory. Application developers can utilize this memory to transfer consistent snapshots of internal state variables.

It is important to note that the dynamic simplex architecture is a generic concept that focuses on the dynamic context switching mechanism. A valid implementation of the dynamic simplex architecture must behave according to the concept, but is nothing more than a framework that protects the overall system from faults of the complex system. An application developer who makes use of it needs to build upon the provided platform and supplement both the complex and the fallback system with their functions.

5 Implementation

As part of the previous work described in [4], the static simplex architecture was implemented on a Zynq UltraScale+ MPSoC from Xilinx. This device combines a block of hard-wired components, such as a dual-core Cortex-R5 from Arm, and an FPGA on a single chip. These portions are commonly referred to as the processing system (PS) and the programmable logic (PL), respectively. For brevity, we will abbreviate the Zynq UltraScale+ MPSoC as ZynqMP.

To allow for a quantitative comparison with the above-mentioned implementation, we will retain its structure wherever possible, but extend it by the fault-triggered partial reconfiguration of the fallback system. Our implementation aims at processor-based fail-operational systems on the ZynqMP and can be described as follows: The complex system is realized by the real-time processing unit (RPU), its generic interrupt controller (GIC), and its tightly-coupled memory (TCM). In fact, the TCM contains software to fulfill the overall system's complex functionality. This software is executed by the RPU's pair of Cortex-R5 cores operating in lockstep mode. As a proof of concept, we trigger a context switch to c_{fallback} whenever the RPU detects a lockstep error and assume that no other faults can occur. Doing so allows us to focus on the context switching mechanism, which is the focus of this work. If required by a particular use case, more sophisticated fault detection techniques may be applied.

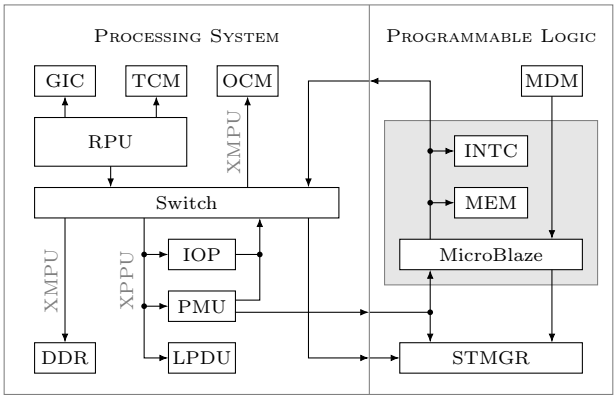

Fig. 3. ZynqMP-based implementation of the dynamic simplex architecture

Figure 3 shows the physical implementation of the system with the dynamic portion highlighted in gray. The fallback system consists of a MicroBlaze processor, its local memory (MEM), and an interrupt controller (INTC). To simplify the debug access to the MicroBlaze, we also include a MicroBlaze debug module (MDM). For technical reasons, the MDM cannot be partially reconfigured and therefore needs to be moved outside of the dynamic portion. Strictly speaking, this means that it is not part of the fallback system.

The platform management unit (PMU) contains a triple-redundant processor for various platform management tasks. We run a custom PMU firmware that implements the control entity. In response to lockstep error notifications from the RPU, it performs a context switch from c_{complex} to c_{fallback}. Following this, the control entity resets the RPU and, in case of a transient fault, performs a controlled context switch back to c_{complex}, i.e., the initial context.

The access protection described in the concept is performed by the Xilinx peripheral protection unit (XPPU). This module is part of the PS and provides detailed control over accesses to the I/O peripherals (IOP), the low-power domain units (LPDU), and the PMU. In applications, IOPs and LPDUs are frequently used as application-specific slave modules. Developers who employ them in their designs have to define access permissions for each such module and context. During a context switch, the control entity uses the permission definitions to reconfigure the XPPU. Here, only the context-dependent part of the XPPU configuration (*context-sensitive apertures*) is written to save time. The state manager (STMGR) implements the state transfer entity from the concept.

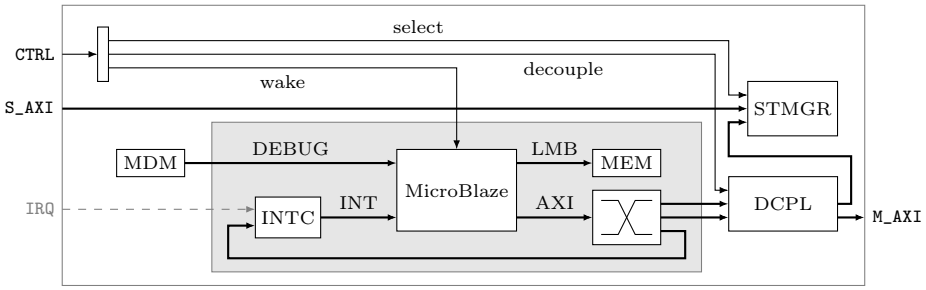

Fig. 4. Block schematic of the PL implementation

At design time, the developer creates two partial bitstreams for the dynamic portion: one containing the fallback system, including the software in MEM, and another one, for c_{complex}, describing its replacement logic. At runtime, both are stored in DDR memory and need to be accessible from the PMU. The partial reconfiguration of the dynamic portion is managed by the control entity. During a switch to c_{fallback}, it reads the partial bitstream from memory and configures the fallback system into the dynamic portion of the PL via the processor configuration access port (PCAP). During a switch to c_{complex}, it configures the custom replacement logic, such as a hardware accelerators, to this portion.

Figure 4 shows a more detailed block schematic of the PL implementation. The dynamic portion is again shown in gray, while the external ports connect to the PS. `CTRL` represents a control signal vector from the PMU. `S_AXI` refers to an AXI slave port from the low-power domain of the PS. `IRQ` represents an application-specific vector of PS-PL interrupt signals. `M_AXI` refers

to an AXI master port to the low-power domain of the PS. The decoupling block (DCPL) is necessary to protect the outgoing AXI signals during the partial reconfiguration process. After this process, both the fallback system and its replacement may access the AXI connections. In particular, the MicroBlaze can use the `M_AXI` interface to communicate with application-specific slave modules. Note that the replacement block needs to have the same interface as the fallback system. The select, decouple, and wake lines are control signals that originate from the PMU. They are operated by the custom PMU firmware.

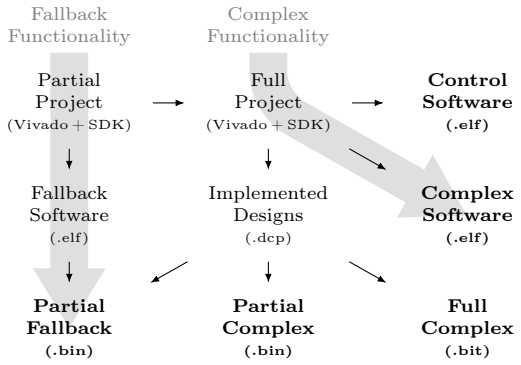

Fig. 5. Proposed development process

Figure 5 visualizes the development process through which application developers can make use of this implementation. Black arrows indicate precedence relations, while final items are shown in bold. The process is based on the Vivado Design Suite 2018.2 and employs the Xilinx SDK for the software portions. We developed a Tcl script that generates a partial Vivado project for the fallback system. The corresponding SDK workspace can be used to develop and build the fallback software. The partial project is then used by another script to generate a full Vivado project for the overall system. This project allows users to develop the logic that replaces the fallback system in c_{complex}. Its synthesis and implementation generates a full bitstream for c_{complex} and partial bitstreams for both contexts. In the partial bitstream for c_{fallback}, the fallback system's local memory (MEM) is automatically initialized with the fallback software. Using the SDK workspace of the full project, the complex software for the RPU can be implemented. The control software for the PMU is automatically generated by our scripts. The shown final items are suitable for use on the ZynqMP.

6 Evaluation and Results

The proposed implementation of the dynamic simplex architecture is a generic framework that aims to protect a ZynqMP-based system from faults of its RPU.

As described in Section 4, we see it as a platform that can be used by application developers when being faced with certain safety requirements. To compare it against an implementation of the static simplex architecture, we will now evaluate two particularly important characteristics of our implementation:

Fault handling latency. In comparison to the static simplex architecture, every context switch now comprises a partial reconfiguration of the PL. To evaluate how this affects the duration that the framework needs to react to faults, we will experimentally determine typical durations of context switches.

Utilization of PL resources. The primary motivation for the dynamic provision of the fallback system is to save PL resources during fault-free operation. To quantify the achieved resource savings, we will consider the PL resource utilization reports that Vivado creates after implementing a design.

6.1 Evaluation Procedure and the Reference Design

The implementation is characterized by many degrees of freedom. Designs can differ, for instance, by the configuration of the MicroBlaze, the size of its local memory, the capacity of the state transfer entity, the number of context-sensitive apertures, the PL clock frequency, and the region of the PL that the dynamic portion is constrained to. We therefore performed a semi-automated design space exploration by considering a fixed MicroBlaze configuration and a fixed state transfer capacity, while varying the other parameters. The automation was realized by a Tcl script that received the parameters, generated the final items from Figure 5, and transferred them to the ZynqMP via JTAG. A dedicated PMU firmware module made it possible to inject a lockstep error into the RPU, measure the latencies of the initiated process, and output the results via UART.

To perform these experiments, we employed a ZCU102 evaluation board from Xilinx. This board is based on the XCZU9EG variant of the ZynqMP. In all our designs, Vivado's routing expansion option was enabled.

As the *reference design*, we consider a design with a PL clock frequency of 100 MHz, a dynamic portion in the X2Y1 region, 32 KiB of local memory, 2 KiB of state manager capacity and $N_{\mathrm{CS}} = 1$ context-sensitive apertures. Using this as our starting point, we varied certain parameters while keeping the others fixed.

6.2 Fault Handling Latency

We will now consider measurements of subsequent context switching intervals. D_i corresponds to the i-th interval of the fault handling process. The intervals that belong to a context switch to c_{fallback} can be described as follows:

- D_1 starts with the lockstep error injection and ends with the point in time at which the control entity is notified about the occurrence of the fault.
- At the end of D_2, the XPPU and STMGR reconfiguration is complete.
- During the third interval, D_3, the partial bitstream that describes the fallback system is read form the DDR and written to the PL.

– At the end of D_4, the fallback software that is part of MEM starts to execute on the MicroBlaze. This constitutes the end of a context switch to c_{fallback}.

D_5 and D_6 correspond to the initiation of an RPU reset, the actual reset of the RPU and the execution of its startup procedure. These intervals are not considered here. The remaining intervals belong to a context switch to c_{complex}:

– The following interval, D_7, corresponds to the time that the MicroBlaze needs to terminate the execution of the fallback software.
– D_8 comprises the partial reconfiguration of the PL as well as the XPPU and STMGR reconfigurations. It can be seen as the counterpart to D_2 and D_3.
– At the end of D_9, the RPU begins to execute the complex software again. This completes the context switch back to the initial context.

Table 1. Measured latencies for the reference design and a variation of N_{CS}

	Average interval duration and uncertainty (in µs)						
N_{CS}	D_1	D_2	D_3	D_4	D_7	D_8	D_9
1	2.340(4)	1.741(3)	1477.2(2)	19.76(5)	1.4(1)	1477.5(3)	5.8(2)
10	2.340(4)	4.82(1)	1477.2(2)	19.73(4)	1.4(1)	1480.5(4)	5.8(2)
100	2.340(5)	34.97(5)	1477.2(2)	19.74(4)	1.4(1)	1510.8(4)	5.8(1)

Table 1 shows the calculated means ($\hat{\mu}_i$) and standard deviations for the reference design in its first row ($N_{\text{CS}} = 1$). For this design, the average measured duration of a fault-induced context switch from c_{complex} to c_{fallback} sums up to $\tau_{\text{fallback}} \approx \sum_{i=1}^{4} \hat{\mu}_i = 1.5\,\text{ms}$. This is the average time between the injection of a fault and the point in time at which the fallback system begins to execute its program. The subsequent context switch back takes $\tau_{\text{complex}} \approx \sum_{i=7}^{9} \hat{\mu}_i = 1.5\,\text{ms}$. Note that the overall latency of a context switch is heavily dominated by the duration of the dynamic portion's partial reconfiguration process (D_3 and the largest part of D_8). The table also illustrates that a variation of N_{CS}, the number of context-sensitive apertures, has an effect on the duration of intervals D_2 and D_8. Each row of the table is based on 100 independent measurements.

Table 2. Measured latencies for a variation of different parameters

	Average interval duration and uncertainty (in µs)						
Variation of...	D_1	D_2	D_3	D_4	D_7	D_8	D_9
(a) Memory size	2.340(4)	1.741(3)	1477.2(2)	19.75(4)	1.4(1)	1477.5(3)	5.8(2)
(b) Clock region	2.340(5)	1.741(3)	(▫)	19.8(1)	1.4(1)	(▫)	5.8(2)
(c) PL frequency	2.340(5)	1.741(3)	1477.6(8)	(⋄)	(⋄)	1478.0(8)	5.8(1)

Using the reference design as a starting point, we then performed a variation of (a) the memory size, (b) the clock region of the dynamic portion, and (c) the PL clock frequency to identify further parameters that have a significant influence on the latencies. Table 2 shows the average interval durations over these variations. Scenarios that are marked with (▫) or (◇) exhibited a strong dependence on the performed variation. A detailed analysis of these cases will be given in the following. From the table, it can be seen that the interval durations are largely independent of the size of the local memory (a). Note that the reference design with 32 KiB of it leaves many BRAMs of the X2Y1 region unutilized. In our experiments, X2Y1 provided room for up to 128 KiB of local memory.

Table 3. Measured latencies for varying locations of the reconfigurable partition

	Average duration and uncertainty of D_3 (in µs)						
Col.	Y0	Y1	Y2	Y3	Y4	Y5	Y6
X0	–	–	–	1861.46(7)	1861.4(2)	–	1861.4(2)
X1	3520.6(6)	3520.6(5)	3524.4(2)	–	1717.3(2)	1717.3(3)	1717.3(2)
X2	1477.2(1)	1477.2(2)	1477.2(2)	1477.23(8)	1477.2(2)	1477.2(1)	1477.2(1)
X3	1378.9(1)	–	1378.9(1)	1378.9(2)	1370.23(5)	1370.25(8)	1370.2(1)
	Average duration and uncertainty of D_8 (in µs)						
Col.	Y0	Y1	Y2	Y3	Y4	Y5	Y6
X0	–	–	–	1861.6(5)	1861.6(5)	–	1861.7(4)
X1	3520.6(10)	3520.4(12)	3524.9(2)	–	1717.4(5)	1717.4(6)	1717.5(4)
X2	1477.5(3)	1477.5(3)	1477.5(4)	1477.4(4)	1477.4(4)	1477.5(3)	1477.5(4)
X3	1379.2(3)	–	1379.2(3)	1379.2(4)	1370.4(4)	1370.4(4)	1370.5(3)

Table 4. Size of the partial bitstream as a function of the clock region

	Size in KiB						
Column	Y0	Y1	Y2	Y3	Y4	Y5	Y6
X0	–	–	–	1292.73	1292.73	–	1292.73
X1	2441.41	2441.41	2441.41	–	1192.91	1192.91	1192.91
X2	1026.78	1026.78	1026.78	1026.78	1026.78	1026.78	1026.78
X3	958.63	–	958.63	958.63	952.63	952.63	952.63

As indicated by the (▫) symbols in Table 2, varying clock region constraints for the dynamic portion (b) lead to significant changes in D_3 and D_8. More detailed measurement results for this variation are shown in Table 3. No implementation was possible for the cases with omitted values. It is important to note that the clock regions differ not only in their location, but also in their size and

resource composition. Table 4 gives the size of a partial bitstream for a design in which the dynamic portion is constrained to the specified clock region. Comparing the values from the two tables shows a strong correlation between the size of a partial bitstream and the average durations of D_3 and D_8. However, note that this observation alone does not prove a causal relation between minimizing the bitstream size and achieving a minimum fault handling latency.

The ($\diamond$) symbols in Table 2 indicate that a variation of f_{PL} (c) has a significant influence on D_4 and D_7. More detailed results for this variation are shown in Table 5. The achievable savings, however, are small compared to the overall fault handling latency. Since the latter is dominated by D_3 and D_8, we focused on the location constraint of the dynamic portion for further improvement.

Table 5. Measured latencies for varying frequencies of the PL clock (f_{PL})

f_{PL} in MHz	D_4 in µs	D_7 in µs
100	19.76(5)	1.4(1)
150	13.6(1)	1.24(1)
215	9.77(2)	1.24(1)
300	7.41(1)	1.24(1)

A more detailed analysis showed that within a certain clock region, lower reconfiguration times can be achieved by reducing the width of the reserved reconfigurable region. We did not consider reconfigurable regions spanning multiple clock regions or the influence of an enabled bitstream compression. Starting off with the reference design again, we reduced the width of the reconfigurable region as much as possible, ending up with what we refer to as the *optimized design*. 100 measurements of it resulted in $\hat{\mu}_3 = 800.85(9)\,\mu\mathrm{s}$ and $\hat{\mu}_8 = 801.2(2)\,\mu\mathrm{s}$. Taking the region-independent durations from the reference design into account leads to overall latencies of $\tau_{\mathrm{fallback}} \approx 0.82\,\mathrm{ms}$ and $\tau_{\mathrm{complex}} \approx 0.81\,\mathrm{ms}$.

To perform a quantitative comparison with a design in which the fallback system is always present, we created an implementation of the static simplex architecture that is—apart from the missing PR aspect—equivalent to the optimized design. Measurements of this version yielded overall latencies of $\tilde{\tau}_{\mathrm{fallback}} \approx 5.31\,\mu\mathrm{s}$ and $\tilde{\tau}_{\mathrm{complex}} \approx 7.5\,\mu\mathrm{s}$. This means that with respect to the static case, a dynamic provision of the fallback system leads to a significant time overhead.

6.3 Resource Utilization

We now compare the resource utilization of the optimized design to that of its static equivalent. In particular, we focus on the following two aspects:

- The number of resources that can be saved during fault-free operation when employing the dynamic instead of the static simplex architecture.
- The resource overhead that goes along with the dynamic simplex architecture while the fallback system is active, i.e., while c_{fallback} is active.

Table 6. PL resource utilization of the optimized design and its static equivalent

Type	Static design	Dynamic design	
		c_{fallback}	c_{complex}
LUT (in CLB)	2932	2888	1209
Register (in CLB)	3171	3182	1459
Multiplexer (F7)	117	149	38
BRAM (36 Kb)	10	10	2

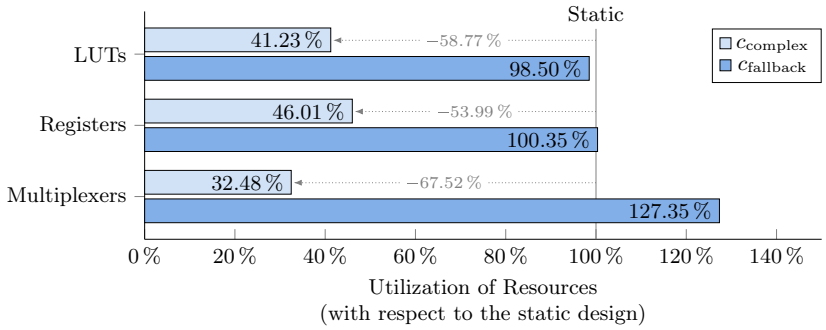

Fig. 6. Relative resource utilization of the optimized design in its two contexts

Table 6 shows that the relative utilization of PL resources in c_{complex} decreases considerably when employing our implementation of the dynamic simplex architecture instead of an equivalent static simplex architecture. This is also visualized in Figure 6. From the figure, it can be seen that the optimized design in c_{complex} saves 59 % of the LUTs, 54 % of the registers, and 68 % of the multiplexers that its static equivalent consumes. In c_{fallback}, its resource overhead is negligible for LUTs and registers, and amounts to 27 % for multiplexers.

7 Discussion

From a qualitative perspective, the evaluation results show that the choice between a static and a dynamic simplex architecture involves a specific trade-off. The dynamic version exhibits prolonged context switching latencies and a slightly increased utilization of FPGA resources in c_{fallback}. At the same time, it consumes considerably fewer FPGA resources during fault-free operation of the system. The saved resources can, for instance, be used to implement hardware accelerators that are required for non-safety-relevant tasks in c_{complex} only.

It should be noted that context switching latencies are critical in the sense that during these intervals, no processor fulfills the desired functionality of the system. In general, we consider the dynamic simplex architecture a feasible solution for cases in which the context switching latencies are tolerable and the PL resources in c_{complex} are too scarce to have the fallback system available

at all times. We believe that the semi-automated design space exploration that we performed is a helpful procedure to map an implementation of the dynamic simplex architecture to arbitrary ZynqMP devices in an efficient manner.

The fault handling latencies that we achieved for our exemplary implementation with 32 KiB of MEM are lower than 1 ms. The results indicate that designs with up to 128 KiB of MEM have fault handling latencies of about 1.5 ms. In cases where these latencies are tolerable, we consider the implementation to be a suitable choice for systems that are subject to certain safety requirements.

8 Conclusion

Our goal was to develop a more resource efficient version of the static simplex architecture, a concept that aims at particular fail-operational systems.

The dynamic simplex architecture utilizes the partial reconfiguration capability of an FPGA to protect the overall system from hazardous failures. It does so by partially reconfiguring a fallback system to the FPGA in response to certain faults. We proposed an implementation of this concept and systematically optimized its fault handling latency. An exemplary design with 32 KiB of local MicroBlaze memory handles faults within 0.82 ms and, considering the non-faulty case, consumes less than 46 % of the resources that an equivalent design in which the fallback system is present at all times occupies.

Our future work will focus on an even more comprehensive design space exploration and an application to practical use cases. The latter will especially include an extensive analysis of the overall safety in such use cases.

Acknowledgements. This work was funded by the German Federal Ministry of Education and Research (BMBF) under grant number 01IS16025 (ARAMiS II). The responsibility for the content of this publication rests with the authors.

References

1. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing **1**(1), 11–33 (Jan 2004). https://doi.org/10.1109/TDSC.2004.2
2. Bak, S., Chivukula, D.K., Adekunle, O., Sun, M., Caccamo, M., Sha, L.: The system-level simplex architecture for improved real-time embedded system safety. In: 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium. pp. 99–107 (April 2009). https://doi.org/10.1109/RTAS.2009.20
3. Baleani, M., Ferrari, A., Mangeruca, L., Sangiovanni-Vincentelli, A., Peri, M., Pezzini, S.: Fault-tolerant platforms for automotive safety-critical applications. In: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems. pp. 170–177. CASES '03, ACM, New York, NY, USA (2003). https://doi.org/10.1145/951710.951734
4. Bapp, F.K., Dörr, T., Sandmann, T., Schade, F., Becker, J.: Towards fail-operational systems on controller level using heterogeneous multicore SoC architectures and hardware support. In: WCX World Congress Experience. SAE International (Apr 2018). https://doi.org/10.4271/2018-01-1072

5. Bolchini, C., Miele, A., Santambrogio, M.D.: TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs. In: 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007). pp. 87–95 (Sep 2007)
6. Cheatham, J.A., Emmert, J.M., Baumgart, S.: A survey of fault tolerant methodologies for FPGAs. ACM Trans. Des. Autom. Electron. Syst. **11**(2), 501–533 (Apr 2006). https://doi.org/10.1145/1142155.1142167
7. Di Carlo, S., Prinetto, S., Trotta, P., Andersson, P.: A portable open-source controller for safe dynamic partial reconfiguration on Xilinx FPGAs. In: 2015 25th International Conference on Field Programmable Logic and Applications (FPL). pp. 1–4 (Sept 2015). https://doi.org/10.1109/FPL.2015.7294002
8. Ellis, S.M.: Dynamic software reconfiguration for fault-tolerant real-time avionic systems. Microprocessors and Microsystems **21**(1), 29 – 39 (1997)
9. Emmert, J., Stroud, C., Skaggs, B., Abramovici, M.: Dynamic fault tolerance in FPGAs via partial reconfiguration. In: Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00871). pp. 165–174 (Apr 2000). https://doi.org/10.1109/FPGA.2000.903403
10. Isermann, R., Schwarz, R., Stölzl, S.: Fault-tolerant drive-by-wire systems. IEEE Control Systems **22**(5), 64–81 (Oct 2002)
11. Kohn, A., Käßmeyer, M., Schneider, R., Roger, A., Stellwag, C., Herkersdorf, A.: Fail-operational in safety-related automotive multi-core systems. In: 10th IEEE International Symposium on Industrial Embedded Systems (SIES). pp. 1–4 (Jun 2015). https://doi.org/10.1109/SIES.2015.7185051
12. Nelson, V.P.: Fault-tolerant computing: Fundamental concepts. Computer **23**(7), 19–25 (Jul 1990). https://doi.org/10.1109/2.56849
13. Papadimitriou, K., Dollas, A., Hauck, S.: Performance of partial reconfiguration in FPGA systems: A survey and a cost model. ACM Trans. Reconfigurable Technol. Syst. **4**(4), 36:1–36:24 (Dec 2011). https://doi.org/10.1145/2068716.2068722
14. Pham, H.M., Pillement, S., Piestrak, S.J.: Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor. IEEE Transactions on Computers **62**(6), 1179–1192 (Jun 2013). https://doi.org/10.1109/TC.2012.55
15. Psarakis, M., Vavousis, A., Bolchini, C., Miele, A.: Design and implementation of a self-healing processor on SRAM-based FPGAs. In: 2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). pp. 165–170 (Oct 2014). https://doi.org/10.1109/DFT.2014.6962076
16. Sha, L.: Using simplicity to control complexity. IEEE Softw. **18**(4), 20–28 (Jul 2001). https://doi.org/10.1109/MS.2001.936213
17. Shreejith, S., Vipin, K., Fahmy, S.A., Lukasiewycz, M.: An approach for redundancy in FlexRay networks using FPGA partial reconfiguration. In: 2013 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 721–724 (Mar 2013). https://doi.org/10.7873/DATE.2013.155
18. Storey, N.R.: Safety-Critical Computer Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1996)
19. Vavousis, A., Apostolakis, A., Psarakis, M.: A fault tolerant approach for FPGA embedded processors based on runtime partial reconfiguration. Journal of Electronic Testing **29**(6), 805–823 (Dec 2013)
20. Vipin, K., Fahmy, S.A.: FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. ACM Comput. Surv. **51**(4), 72:1–72:39 (Jul 2018). https://doi.org/10.1145/3193827