GUARD: GUAranteed Reliability in Dynamically Reconfigurable Systems

Hongyan Zhang[1], Michael A. Kochte[2], Michael E. Imhof[2], Lars Bauer[1], H.-J. Wunderlich[2] and Jörg Henkel[1]
[1]Chair for Embedded Systems, Karlsruhe Institute of Technology, Karlsruhe, Germany
{hongyan.zhang, lars.bauer, henkel}@kit.edu
[2]Institute of Computer Architecture and Computer Engineering, University of Stuttgart, Germany
{kochte, imhof}@iti.uni-stuttgart.de, wu@informatik.uni-stutgart.de

ABSTRACT

Soft errors are a reliability threat for reconfigurable systems implemented with SRAM-based FPGAs. They can be handled through fault tolerance techniques like scrubbing and modular redundancy. However, selecting these techniques statically at design or compile time tends to be pessimistic and prohibits optimal adaptation to changing soft error rate at runtime.

We present the GUARD method which allows for autonomous runtime reliability management in reconfigurable architectures: Based on the error rate observed during runtime, the runtime system dynamically determines whether a computation should be executed by a hardened processor, or whether it should be accelerated by inherently less reliable reconfigurable hardware which can trade-off performance and reliability. GUARD is the first runtime system for reconfigurable architectures that guarantees a target reliability while optimizing the performance. This allows applications to dynamically chose the desired degree of reliability. Compared to related work with statically optimized fault tolerance techniques, GUARD provides up to 68.3% higher performance at the same target reliability.

1. INTRODUCTION AND RELATED WORK

Reconfigurable architectures such as the Xilinx Zynq platform allow to dynamically optimize application performance and energy dissipation. Fine-grained reconfigurable architectures [1], the focus of this work, implement computationally intensive parts as so-called *Accelerated Functions* (AFs) by implementing them on dedicated hardware *accelerators*. To optimally adapt to changing application performance requirements and data-dependent execution flows, the accelerators instantiated in hardware are determined at runtime.

In SRAM-based FPGA platforms the reconfiguration capabilities can not only be used to optimize performance and energy dissipation, but also to increase availability [2] by diagnosis and repair [3, 4], or to balance stress to mitigate aging [5]. In contrast, in this work the adaptability of the system is exploited to *guarantee a given target reliability at minimal cost*.

Harsh environmental conditions (radiation, temperature, power noise) may cause transient errors and failures which are not acceptable in safety-critical applications (e.g. automotive, industrial, medical or aviation), where stringent reliability requirements such as ASIL [6] have to be met under different environmental and operating conditions and changing error rates in the system. Shrinking CMOS technology further aggravates these threats [7].

DAC'14, June 01 - 05 2014, San Francisco, CA, USA

http://dx.doi.org/10.1145/2593069.2593146.

In SRAM-based FPGAs, the dominant reliability threat are soft errors in the configuration memory and functionally used memory (e.g. block RAMs and flipflops), which may alter the functionality of hardware accelerators and lead to wrong results. To ensure reliable computation in the reconfigurable fabric, accelerators must be protected by fault tolerance methods such as modular redundancy (e.g. duplication with comparison (DWC), triple modular redundancy (TMR)), or information redundancy (self-checking circuits, ECC of memory). Errors in the configuration memory in FPGAs can be effectively handled by periodic scrubbing [8], i.e. reading out and checking the memory contents, and periodic functional self tests [9], followed by correction.

These fault tolerance methods have different costs in terms of hardware resources, performance, and energy. Typically, the cost is dominated by error detection [10] which must run concurrently to regular system operation. Error *correction* by re-execution after an error has been detected typically incurs only a small performance cost and happens rarely.

Due to changing error rates, application requirements (data dependencies, multi-threading) and system states (available/used resources), it is not possible to *statically* determine appropriate error detection methods for a given target reliability at minimal cost. A static optimization is pessimistic since it must consider the worst case and when the error rate is low, the system is over-protected at additional hardware or performance cost.

A static selection of fault tolerance methods as in [11] can not adapt to changing soft error rates and thereby hinders trading off performance and reliability. The runtime selection of DWC and TMR for accelerators according to static soft error rate thresholds as in [12] increases performance and *availability*, but does not guarantee a target *reliability*.

Contributions.

In contrast to the static and therefore pessimistic selection of fault-tolerance methods in the state-of-the-art, we present the GUARD method for reconfigurable architectures. This fault tolerance method guarantees an application-specified minimum level of reliability of the accelerated computation at minimal cost. This is achieved by use of monitoring information to dynamically choose between different reliability methods so that the error-detection overhead is minimized:

- At runtime, the soft error rate is monitored and the reliability of future computations is estimated. Based on statically or dynamically given target reliability constraints, runtime reliability management is performed.
- Based on the reliability estimation, the selection of ac-

celerators and the application of optimal fault tolerance methods are performed at runtime. This allows for fast adaptation to changing reliability threats and guarantees the given reliability constraints while maximizing the performance.

The next section introduces reconfigurable architectures and gives a problem definition. Section 3 presents the GUARD method. Section 4 presents experiment results, followed by the conclusion.

2. SYSTEM OVERVIEW AND PROBLEM DEFINITION

2.1 Reconfigurable Architectures

Fig. 1 shows the common structure of a reconfigurable architecture. It consists of a processor core and an attached reconfigurable fabric that is composed of so-called *containers* that are implemented on an embedded SRAM-based FPGA. A dedicated bus for the reconfigurable fabric manages the communication among containers and the communication with the system bus. We assume that the processor core is a reliable computing base such as [13], i.e. it is hardened by manufacturing technology or by redundancy [14]. Bus-structures and memory are protected by ECC. Thus, they are much less susceptible to soft errors than accelerators implemented in the reconfigurable fabric. To be able to apply modular redundancy methods to the containers, one non-reconfigurable hardened voter is provided per container. They allow to duplicate/triplicate any pair or triplet of neighbored containers.

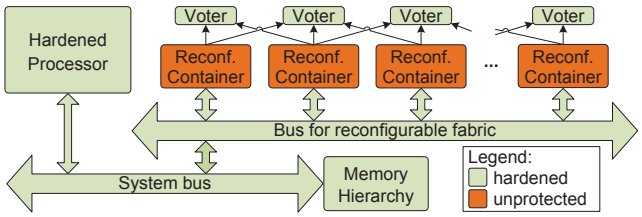

Figure 1: Basic reconfigurable architecture.

Applications running on reconfigurable architectures may use so-called *Accelerated Functions* (AFs) to implement computationally intensive parts. Many applications execute as a (repetitive) series of *phases* [15], e.g. tasks in a task graph, that differ in required processing time, suitability for acceleration, or susceptibility to soft errors. In reconfigurable architectures, offline profiling and runtime monitoring is used to track these phases and to determine which parts of the computation can be mapped to AFs. An AF can be implemented by one or multiple so-called *accelerators* that are reconfigured into containers (one accelerator per container at any given time). Alternatively, AFs can be executed in software on the processor, e.g. when the required accelerators are not available.

AFs can be of different size, from a complex function down to a short sequence of instructions. They are represented by a data-flow graph (DFG) where each node corresponds to an accelerator and the edges correspond to data-flow between the accelerators. Fig. 2a) shows an example AF that consists of three different accelerator types (A_1, A_2, A_3) and requires at least three different containers (one for each accelerator type) to be implemented. The example in Fig. 2a) uses exactly three containers and thus the two instances of A_3 in the DFG have to be executed in different *steps*.

An AF may have multiple hardware implementation variants that trade-off performance and resource usage (i.e. number of containers). The two variants shown in

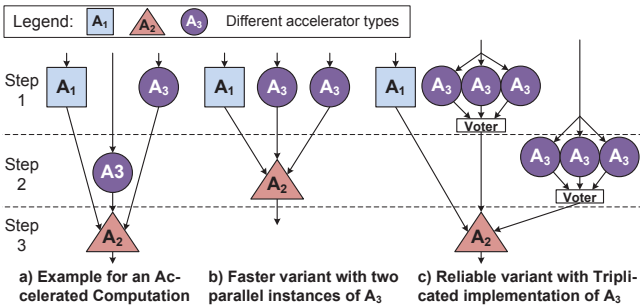

Figure 2: Different hardware implementation variants of an Accelerated Function (AF).

Fig. 2a) and 2b) differ in latency and resource usage per step. Variant a) uses only one instance A_3 per step and finishes in 3 steps while variant b) uses two instances of A_3 in parallel (demanding two separate containers) in step 1 and thus finishes in 2 steps. Variants that use more accelerators exploit more parallelism and can achieve higher performance. It is also possible to provide a partially or completely fault tolerant variant, e.g. by triplicating A_3, as shown in Fig. 2c). This variant has the same schedule as variant a) but uses A_3 in TMR mode to increase reliability at higher resource usage. Variant a) is called the *base variant* of the reliable variant c), which is derived from variant a) by duplicating or triplicating a subset of its accelerators.

2.2 Problem Definition

The term *reliability* denotes the probability of error-free operation for a specified period of time [2]. The reliability of a system depends on the reliability of its components. We assume the processor core to be a reliable computing base (see Section 2.1) and focus on the reliability of the AFs: We assume an AF to be error free if its execution is not affected by soft errors. An AF that is executed as a software routine on the reliable computing base is reliable. The reliability of an AF that is executed on the reconfigurable fabric depends on the current error rate, system state, and hardware usage:

Current error rate is determined by the environment.

System state corresponds to the reliability history of the containers, i.e. the time since a container was last known to be error free because it was tested, reconfigured, or scrubbed (i.e. reconfiguring it with the configuration data of the accelerator that was already configured or reading back the configuration data and correcting possible errors by an error correction code).

Hardware usage depends on the accelerators that implement the AF. The configuration information of an accelerator is stored in the SRAM configuration memory of the FPGA, which is susceptible to soft errors. The *critical bits* of an accelerator are those configuration bits that define its functionality. Different accelerators exhibit different susceptibility to soft errors in their configuration memory depending on the number of critical bits.

The variety in soft error vulnerability of accelerators also extends to the hardware implementations of AFs: different AFs and various implementation variants of an AF differ in their soft error vulnerability. This variety can be exploited by the runtime system of the reconfigurable architecture. In order to guarantee a given target reliability while optimizing the performance, the challenge is threefold:

1. Whenever an AF shall execute, ensure that it meets the target reliability for the current error rate and system state. If the reliability constraint cannot be satisfied at

the moment due to pending reconfigurations of redundant accelerators or limited hardware resources, then the AF needs to be executed on the reliable computing base.
2. For all AFs of the executed application phase, decide which implementation variant shall be reconfigured and find a good trade-off that ensures the target reliability while maximizing performance for the monitored error rate and system state.
3. Decide for each container when to perform scrubbing. After scrubbing, an accelerator is known to be error free. As no other container can be reconfigured until scrubbing completes, scrubbing also reduces performance.

The runtime system needs to address all three challenges at runtime. The optimization problem in Challenge 2 corresponds to a Knapsack problem where—in addition to satisfying the reliability constraint—the number of accelerators to implement the chosen AF variant must not exceed the number of containers (capacity of the Knapsack) and the performance of the AFs shall be maximized (optimization).

3. GUARD METHOD

3.1 Runtime Estimation of the Soft Error Rate

The current soft error rate in the system changes with its environment and depends for instance on the radiation level, temperature or voltage [7, 14]. We estimate the current soft error rate λ per bit by computing the maximum of two indicators available in the system:

1. The error rate per bit λ_{scrub} in the configuration bits obtained from periodic scrubbing,
2. The error rate per bit λ_{cache} in the cache SRAM array of the hardened processor. This error rate can be obtained since in our architecture the cache is protected by a single-error correcting code. λ_{cache} is derated by ρ according to the cache size and technology parameters (critical area/cross-section per bit).

The current soft error rate per bit in the system is then computed conservatively and concurrently to system operation as their maximum: $\lambda := \max(\lambda_{scrub}, \rho \cdot \lambda_{cache})$.

3.2 Reliability of Accelerated Functions

The reliability of an accelerated function depends on the soft error rate, the type, structure and size of the used hardware accelerators, and the *resident* time the accelerators have been instantiated without errors in the reconfigurable fabric, i.e. the time elapsed since the last reconfiguration or scrubbing event of the container.

If the soft error rate λ per bit is constant, the probability that a 1-bit memory element is not flipped due to a soft error during time period t is $e^{-\lambda t}$ [16]. In other words, if a bit is correct at t_0, the probability that the bit is still correct at $t_0 + t$ is $e^{-\lambda t}$. The probability that n independent correct bits remain correct throughout a time period t is then $e^{-n\lambda t}$ if all bits have the same error rate.

Since the soft error rate may change over time, we conservatively use the maximum observed error rate during the resident time of an accelerator for the reliability estimation.

For an accelerator A_i with n_i critical bits, the probability that none of its critical bits are affected by soft errors from t_0 to $t_0 + t$, i.e. A_i is able to compute the correct results, is $e^{-n_i \lambda t}$, given that all critical bits are correct at t_0. This is the case if the accelerator is reconfigured at t_0, or scrubbed at t_0 without errors. t is then the *resident time* of the accelerator.

Functionally used memory in the FPGA, i.e. block RAMs and flipflops, is not protected by scrubbing, but is implicitly protected if modular or temporal redundancy is employed. Furthermore, block RAMs are readily used with ECC in the recent FPGA generations [17]. Compared to the number of flipflops contained in an FPGA, the amount of configuration bits is higher by two to three orders of magnitude (e.g. a logic slice has 1184 configurations bits and 4 flipflops [17]). Also, flipflops are not susceptible to upsets throughout the entire clock cycle, and not every upset leads to an error observed by the system or user. The time during which data in memory are vulnerable is bound by the duration of the accelerator execution (in the order of cycles), which is much smaller than the resident time of configuration bits of accelerators (in the order of million cycles). Therefore, soft errors in block RAM and flipflops are not considered in the following.

For accelerators without any fault-tolerance methods, the reliability of an accelerated function AF (probability that it produces the correct result) is

$$R(AF, t, \tau) := \prod_{i}^{A_i \in AF} e^{-n_i \lambda (t_i + \tau_i)}, \tag{1}$$

where t_i is the resident time of accelerator A_i until the accelerated function starts to execute, and τ_i denotes the time period until accelerator A_i finishes all its executions. Since $\tau_i \ll t_i$, we ignore τ in the following calculation.

We assume conservatively that an accelerator computes the correct results only if all its critical bits are correct, i.e. logic and data-dependent masking of errors are ignored here. Such error masking can be added to this computation by derating factors derived for instance from fault injection experiments.

The *reliability constraint* is the requirement that the failure probability of each execution of the accelerated function AF_k, i.e. $1 - R(AF_k, t_k)$, is less than or equal to a statically or dynamically given threshold, usually written in powers of ten as 10^{-r_k}:

$$\forall k \ : \ 1 - R(AF_k, t_k) \leq 10^{-r_k}. \tag{2}$$

For instance, when $r_k = 5$, the failure probability of each execution of AF_k must be less than 10^{-5}. In Eq. (1) and (2), the values of n_i and τ_i are derived from the AF implementations at design time. λ, t_i, and the target reliability r_k are variables whose values may dynamically change during runtime.

Implementation variants of accelerators may include partially or completely protected accelerators based on duplication or triplication (cf. Section 2.1). For accelerators in TMR mode with hardened voter, the probability that it delivers the correct output is the probability that at most one of the three replicated accelerators is affected by soft errors in their critical bits, which is

$$\begin{aligned} R(A_i^{TMR}) := {} & (1 - R(A_a))\, R(A_b) R(A_c) + \\ & (1 - R(A_b))\, R(A_a) R(A_c) + \\ & (1 - R(A_c))\, R(A_a) R(A_b) + \\ & R(A_a) R(A_b) R(A_c) \\ := {} & e^{-n\lambda(t_a + t_b)} + e^{-n\lambda(t_a + t_c)} + e^{-n\lambda(t_b + t_c)} \\ & - 2e^{-n\lambda(t_a + t_b + t_c)}, \end{aligned} \tag{3}$$

where $R(A_a)$, $R(A_b)$ and $R(A_c)$ denote the reliability of the three replicated accelerators. t_a, t_b and t_c denote the resident times of the three replicated accelerators.

For accelerators in DWC mode, the accelerated function is re-executed on the hardened processor if an error is detected. Thus the probability of correct results equals to the probability that at most one of the replicated accelerators is erroneous:

$$R(A_i^{DWC}) := e^{-n\lambda t_a} + e^{-n\lambda t_b} - e^{-n\lambda(t_a + t_b)}. \tag{4}$$

3.3 Runtime Reliability Management

3.3.1 Maximum Resident Time

To satisfy the reliability constraint in Eq. (2), the runtime system must ensure that unprotected accelerators used in the next execution of AF_k are still sufficiently reliable. This requires that the resident times of non-redundant accelerators in AF_k satisfy the inequality: $\prod_i^{A_i \in AF_k} e^{-n_i \lambda t_i} \geq 1 - 10^{-r_k}$. After applying the logarithm on both sides, we obtain

$$\sum_i^{A_i \in AF_k} n_i t_i \leq -\frac{1}{\lambda} \log\left(1 - 10^{-r_k}\right). \quad (5)$$

By making t_i small enough, e.g. by scrubbing accelerators more frequently, the reliability constraint can be fulfilled. However, there are many combinations of resident times t_i which satisfy Eq. (5). To find the optimal combination which maximizes every t_i so that the scrubbing overhead is minimized, the runtime system has to solve a max-min problem involving $\|AF_k\|+2^{\|AF_k\|}$ constraints, where $\|AF_k\|$ is the number of accelerators required by AF_k. This is too complex for the runtime system and would decrease its responsiveness to other important tasks.

To simplify the problem, let t_{max} denote the maximum resident time of all accelerators required by an accelerated function AF_k, i.e. $t_{max} = \max_i\{t_i\}$. Then,

$$\sum_i^{A_i \in AF_k} n_i t_i \leq \sum_i^{A_i \in AF_k} n_i t_{max}, \quad (6)$$

and Eq. (5) is automatically satisfied when

$$t_{max} \leq \underbrace{\frac{1}{\sum_i^{A_i \in AF_k} n_i} \left(-\frac{1}{\lambda} \log\left(1 - 10^{-r_k}\right)\right)}_{T_k^{up}}. \quad (7)$$

We denote the right-hand side of Eq. (7) as T_k^{up}, the upper bound of t_{max} for AF_k. With the above *tightening*, the runtime system only needs to schedule scrubbing for non-redundant accelerators such that t_{max} satisfies Eq. (7), which is stricter than required.

For an AF_k consisting of only triplicated accelerators and applying tightening by $t_{max} = \max\{t_a, t_b, t_c\}$, the reliability constraint $1 - R(A_i^{TMR}) \leq 10^{-r_k}$ becomes $3e^{-2n\lambda t_{max}} - 2e^{-3n\lambda t_{max}} \geq 1 - 10^{-r_k}$. This can be easily solved by substitution to obtain the bound for t_{max}. But it becomes difficult when we compute t_{max} for partially fault tolerant variants as shown in Fig. 2c). However, we can always find a suitable q (usually < 1) such that

$$3e^{-2n\lambda t_{max}} - 2e^{-3n\lambda t_{max}} \geq e^{-qn\lambda t_{max}} \quad (8)$$

holds for all t_{max} where $e^{-n\lambda t_{max}}$, the reliability of a non-redundant accelerator, is assumed to be larger than a very conservative value such as 0.99. Therefore the reliability constraint for an arbitrary accelerated function combining non-redundant and triplicated accelerators is tightened to

$$\prod_i^{A_i \in AF_k, \text{non-red.}} e^{-n_i \lambda t_{max}} \prod_j^{A_j \in AF_k, TMR} e^{-q n_j \lambda t_{max}} \geq 1 - 10^{-r_k}, \quad (9)$$

where t_{max} is the maximum resident time of all accelerators. After taking the logarithm on both sides, we obtain

$$t_{max} \leq \underbrace{\frac{1}{\sum_i n_i + \sum_j q n_j} \left(-\frac{1}{\lambda} \log\left(1 - 10^{-r_k}\right)\right)}_{T_k^{up}}, \quad (10)$$

where the right-hand side of Eq. (7) is denoted as T_k^{up}, the upper bound of t_{max} for AF_k. In a similar way, tightening is also applied to accelerated functions with accelerators in duplicated mode.

3.3.2 Acceleration Variants Selection

When the application requests to execute accelerated functions in hardware, the runtime system has to select from a large set of acceleration variants to configure, which have distinct performance, reliability and resource usage characteristics. The variants of an AF consists of a common set of accelerators and the bitstream of these accelerators are stored in the memory for online reconfiguration. As an motivational example, Fig. 3 shows the selection space for a complex H.264 encoder application, in which nine AFs are implemented. Each data point in the figure denotes

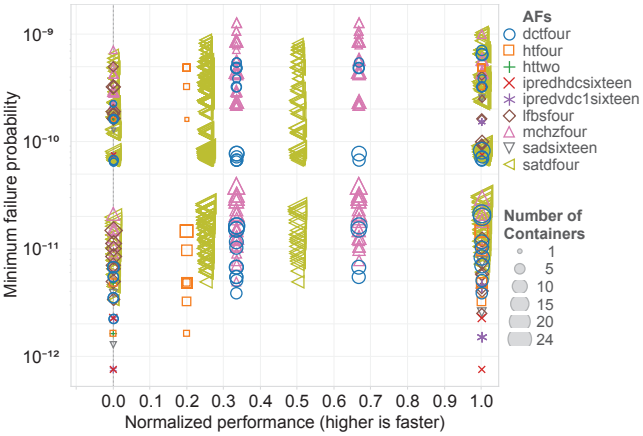

Figure 3: Variants selection space for an error rate of 10 errors $\text{Mb}^{-1}\text{month}^{-1}$.

an acceleration variant of a specific AF (coded in color and shape) including partially and completely fault tolerant variants. Each variant is described by three metrics: minimum failure probability (Y-axis), performance (X-axis) and number of containers (size of the data point). The *minimum* failure probability of a variant is its failure probability when t_{max} equals the minimum scrubbing period of the system. The minimum scrubbing period (MinScrubPeriod) is the time required to scrub all containers once (i.e. scrubbing the whole system at highest frequency). For the variants, the failure probability differs by more than three orders of magnitude.

The performance shows the speedup of each variant compared to software execution, normalized for each AF. The absolute speedup ranges from 6.3 to 70.2×.

The runtime system selects the accelerator variants upon an application request. Thus, the selection must complete in a short time period despite of the large selection space. This makes it computationally inviable to obtain an exact solution to the underlying NP-complete Knapsack problem (see Section 2.2). Alg. 1 shows our greedy algorithm that selects the appropriate variants for requested accelerated functions such that the target reliability and resource constraints are satisfied and the performance of the whole application is maximized. The worst-case complexity of Alg. 1 is $O(n^2)$, where n is the number of variants to be selected.

The variant selection is guided by a *performance score* which ensures that the selection is resource efficient and the performance of the whole application increases: Line 1 collects those acceleration variants v for the requested accelerated functions ($v.\text{fct} \in \mathcal{F}$) into set $\mathcal{C}$ which are able to meet the reliability constraint, i.e. the upper bound of t_{max} for the

Algorithm 1 Acceleration variants selection

Input: The set of accelerated functions to be executed $\mathcal{F}$.
Output: Selected variants for each accelerated function in $\mathcal{F}$.

1. $\mathcal{C} := \{\text{all variants } v \text{ for } v.\text{fct} \in \mathcal{F} \mid T^{up}(v) \geq \text{MinScrubPeriod}\}$
2. $\mathcal{C} := \{v \mid v \in \mathcal{C} \text{ and } \forall u \in C, u.\text{base} = v.\text{base} : \|u\| > \|v\|\}$
3. N := NumberOfContainers // Total number of containers
4. $\mathcal{R} := \emptyset$ // Result set
5. **while** $\mathcal{C} \neq \emptyset$ **do**
6. $\quad \mathcal{C} := \mathcal{C} \setminus \{v \mid v \in \mathcal{C}, \|v\| > N\}$
7. $\quad$ **if** $\mathcal{C} = \emptyset$ **then**
8. $\quad\quad$ break
9. $\quad$ **end if**
10. $\quad v_{best}$:= NULL ; BestScore := $-\infty$
11. $\quad$ **for all** $v \in \mathcal{C}$ **do**
12. $\quad\quad v_{sel}$:= fastest variant $w \in \mathcal{R}$ with $w.\text{fct} = v.\text{fct}$
13. $\quad\quad$ **if** v_{sel}=NULL **then**
14. $\quad\quad\quad$ Score := $f_{\text{EX}}(v.\text{fct}) \cdot (v.\text{sw\_cycles-}v.\text{hw\_cycles})/\|v\|$
15. $\quad\quad$ **else**
16. $\quad\quad\quad$ Score := $f_{\text{EX}}(v.\text{fct}) \cdot (v_{sel}.\text{hw\_cycles-}v.\text{hw\_cycles})/\|v\|$
17. $\quad\quad$ **end if**
18. $\quad\quad$ **if** Score > BestScore **then**
19. $\quad\quad\quad v_{best} := v$
20. $\quad\quad\quad$ BestScore := Score
21. $\quad\quad$ **end if**
22. $\quad$ **end for**
23. $\quad v_{replace} := \ v \in \mathcal{R} \wedge v.\text{fct} = v_{best}.\text{fct}$; $\mathcal{C} := \mathcal{C} \setminus \{v_{best}\}$
24. $\quad$ **if** $v_{replace}$ = NULL **then**
25. $\quad\quad \mathcal{R} := \mathcal{R} \cup \{v_{best}\}$; N := N $- \|v_{best}\|$
26. $\quad$ **else if** $v_{replace}$.hw_cycles > v_{best}.hw_cycles **then**
27. $\quad\quad \mathcal{R} := \left(\mathcal{R} \setminus \{v_{replace}\}\right) \cup \{v_{best}\}$
28. $\quad\quad$ N := N $+ \|v_{\text{Replace}}\| - \|v_{best}\|$
29. $\quad$ **end if**
30. **end while**
31. **return** $\mathcal{R}$ // Selected variants to be configured

variant is greater or equal to the minimum scrubbing period of the system. As discussed in Section 3.3.1, the upper bound of t_{max} depends on the used resources and applied fault tolerance method of the variant. Line 2 keeps the smallest derived variant per base variant (see Section 2.1) in $\mathcal{C}$, i.e. the variant using the fewest containers ($\|v\|$ denotes the number of containers required by v). The loop from Line 5 to Line 30 iteratively selects the variant with the highest *performance score* among others in $\mathcal{C}$, and which still fits into the available containers. Line 16 calculates the performance score of a variant as the weighted speedup gain compared to a previously selected variant for the same accelerated function: The weight is the history execution frequency f_{EX} of the accelerated function divided by the number of containers required by the variant. If there is no previously selected variant, the speedup gain is calculated relative to the software execution (Line 14). The variant v_{best} with highest score is added to the result set $\mathcal{R}$ if there is no faster variant (fewer execution cycles) of the same function already in $\mathcal{R}$. The main loop continues until $\mathcal{C}$ is empty, or no variant with the targeted reliability fits into the remaining containers.

Before the actual execution of an accelerated function, the runtime system checks if the hardware variant selected by Alg. 1 is already configured, and if it still satisifies the reliability constraint for the current error rate (both might have changed since the last execution of Alg. 1). If that is not the case, the AF is executed in software by the hardened processor.

3.3.3 Non-uniform Accelerator Scrubbing

The scrubbing rate for each container is determined by the accelerator implemented in it. If the accelerator belongs to an accelerator variant which requires a short resident time to satisfy the reliability constraint, the container must be scrubbed more frequently. More precisely, if t_{max} of a variant has to satisfy Eq. (10), then all the containers it uses are scrubbed as soon as the resident time exceeds (T_k^{up} − MinScrubPeriod). In this way, t_{max} of every implemented variant is guaranteed to satisfy the tightened reliability constraint and the scrubbing overhead is minimized.

4. EXPERIMENTAL EVALUATION

We evaluated the presented approach in a reconfigurable architecture as described in Section 2.1, implemented on a Xilinx Virtex-5 LX110T FPGA. The target application is an H.264 video encoder which was selected because it contains multiple accelerated functions with distinct performance and reliability characteristics (cf. Fig. 3). The H.264 encoder contains nine accelerated functions whose hardware implementations employ nine accelerator types in total. The number of critical bits of the accelerators range from 19036 to 86796 bits and are obtained using the Xilinx `bitgen` tool.

A SystemC-based cycle-accurate simulator with parameters extracted from the hardware implementation is used to evaluate the GUARD method with respect to related work. It simulates the execution of an application cycle-accurately by modeling the reconfigurable architecture of Fig. 1 including the reconfigurable FPGA-resources, implementation constraints for the accelerated functions (e.g. bus accesses), the duration of reconfigurations, access arbitration to the ICAP-configuration-port, and the runtime system which decides when and which reconfiguration to perform. This is extended by the reliability model of section 3 and Alg. 1.

To evaluate the behavior of the system in response to different environmental conditions, we need to change the simulated soft error rates between 0 (no errors) and 10 errors $\text{Mb}^{-1}\text{month}^{-1}$ to comprise the realistic cases [18]. The variation speed is in the order of seconds to stress the dynamic system adaptation. Therefore, we use a sinusoidal soft error rate as input stimuli for our runtime system. The period corresponds to 10 s in real time for a 100 MHz clock frequency.

For the performance evaluation, we apply the GUARD method with reliability constraints from $r = 8$ to $r = 11$ (cf. Section 3.2), i.e. the failure probability of each AF execution must be less than 10^{-r}. We compare it to a threshold based approach similar to [12] which duplicates (DWC) or triplicates (TMR) the accelerators when the error rate exceeds 1.8 $\text{Mb}^{-1}\text{month}^{-1}$. This ensures that the AF failure probability is always less than 10^{-10}. In the threshold based approach, scrubbing is performed at maximum rate.

The results are shown in Fig. 4. Depending on the error rate, the system reacts and implements fault tolerance methods. These require hardware resources which are not any longer available for acceleration and thus the performance decreases. With more relaxed reliability constraints (i.e. smaller values of r), it is less probable that fault tolerance methods are required and therefore less performance impact is observed. When the threshold-based methods switch to duplicated or triplicated implementations, much more resources are consumed. This causes a stark performance drop. For a low error rate, the performance is still below the GUARD approach since the high scrubbing frequency blocks the configuration port.

Fig. 5 shows the average AF failure probability of the approaches. In the unprotected system, the failure probability reaches $4.4 \cdot 10^{-6}$. For the GUARD method, the failure probability is effectively bound by the given reliability constraint, even for higher error rates. The step-shaped change in the curve for $r = 11$ is due to the large gap in failure probability in the selection space (cf. Fig. 3). A system which applies only scrubbing at maximal frequency

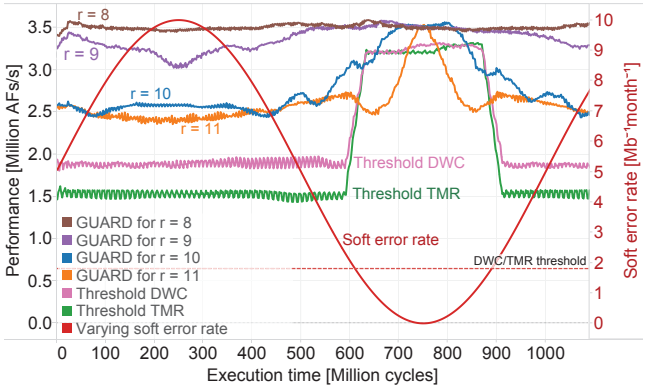

Figure 4: Performance under varying soft error rate.

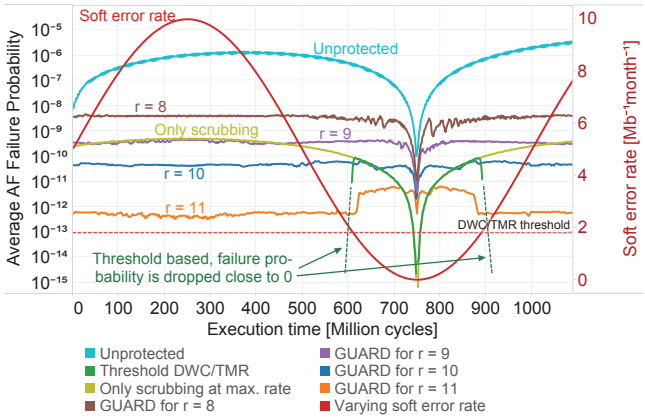

Figure 5: Average AF failure probability for different fault tolerance methods under varying soft error rate.

can obtain a minimum failure probability of approx. 10^{-9}. The threshold based modes over-protect the system when scrubbing alone cannot ensure sufficient reliability and accelerators are replicated. Then, the resource usage is excessive with adverse performance impact (cf. Fig. 4).

Table 1 summarizes the results of the scenarios investigated in Fig. 4 and Fig. 5. For $r = 9$, GUARD achieves a *reliability improvement factor* of 2384 at only 5.3% average performance reduction compared to the unprotected system. Compared to the threshold based methods, GUARD guarantees the same target reliability while providing 20.0% (DWC) or 42.6% (TMR) higher performance in average. In the best case, GUARD is up to 34.8% (DWC) or 68.3% (TMR) faster.

Table 1: Performance and failure probability results

Mode	Perf. [Mil.AFs/s]			Failure Probability	
	min	avg	max	avg	max
Unprotected	3.45	3.56	3.60	8.70×10^{-7}	4.40×10^{-6}
Thresh.DWC	1.81	2.34	3.30	8.28×10^{-12}	9.12×10^{-11}
Thresh.TMR	1.45	1.97	3.32	8.28×10^{-12}	9.12×10^{-11}
GUARD r=8	3.40	3.50	3.59	3.49×10^{-9}	5.49×10^{-9}
GUARD r=9	3.03	3.37	3.58	3.65×10^{-10}	5.62×10^{-10}
GUARD r=10	2.44	2.81	3.56	4.77×10^{-11}	7.91×10^{-11}
GUARD r=11	2.36	2.61	3.53	1.48×10^{-12}	6.92×10^{-12}

5. CONCLUSIONS

The presented GUARD runtime method allows autonomous runtime reliability management in reconfigurable architectures. Considering the monitored error rate and derived reliability estimates of future computations, it dynamically selects appropriate acceleration variants and applies optimal fault tolerance methods such as scrubbing and modular redundancy. Thereby it guarantees an application-specific minimum level of reliability of the accelerated computations. Since it is not over-protective, the performance of the application is maximized for the given target reliability.

The experimental results show that GUARD dynamically trades-off reliability and performance depending on the application and environment and significantly increases reliability at small cost. Compared to related work, GUARD performs up to 68.3% faster.

Acknowledgments This work is supported in parts by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 – `http://spp1500.itec.kit.edu`).

References

[1] S. Vassiliadis and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing*. Springer, 2007.

[2] A. Avizienis *et al.*, "Basic Concepts and Taxonomy of Dependable and Secure Computing", *IEEE Trans. on Dep. and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[3] C. Stroud, E. Lee, and M. Abramovici, "BIST-based diagnostics of FPGA logic blocks", in *IEEE International Test Conference*, 1997, pp. 539–547.

[4] S. Mitra *et al.*, "Reconfigurable architecture for autonomous self-repair", *IEEE Design & Test of Comput. (D&ToC)*, vol. 21, no. 3, pp. 228–240, 2004.

[5] H. Zhang *et al.*, "Module Diversification: Fault Tolerance and Aging Mitigation for Runtime Reconfigurable Architectures", in *IEEE Int'l Test Conference*, 2013, paper 14.1.

[6] "ISO 26262: Road vehicles – Functional safety", ISO, 2011.

[7] R. Baumann, "Soft errors in advanced computer systems", *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, May–June 2005.

[8] C. Carmichael, M. Caffrey, and A. Salazar, "Correcting Single-Event Upsets Through Virtex Partial Configuration", *Xilinx Application Note, XAPP216 (v1.0)*, 2000.

[9] L. Bauer *et al.*, "Test strategies for reliable runtime reconfigurable architectures", *IEEE Transactions on Computers*, vol. 62, no. 8, pp. 1494–1507, 2013.

[10] S. Das *et al.*, "RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance", *IEEE Journal of Solid-State Circuits*, vol. 44, no. 1, pp. 32–48, 2009.

[11] L. Sterpone, M. Porrmann, and J. Hagemeyer, "A novel fault tolerant and runtime reconfigurable platform for satellite payload processing", *IEEE Transactions on Computers*, vol. 62, no. 8, pp. 1508–1525, 2013.

[12] A. Jacobs *et al.*, "Reconfigurable fault tolerance: A comprehensive framework for reliable and adaptive FPGA-based space computing", *ACM Trans. Reconfigurable Technol. Syst.*, vol. 5, no. 4, pp. 21:1–21:30, Dec. 2012.

[13] Xilinx, "Zynq-7000: A Generation Ahead", *Technology Backgrounder*, 2013.

[14] S. Mukherjee, *Architecture design for soft errors*. Morgan Kaufmann Publishers, 2008.

[15] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction", *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, pp. 336–349, 2003.

[16] S. Su, I. Koren, and Y. Malaiya, "A continuous-parameter markov model and detection procedures for intermittent faults", *IEEE Trans. Comp.*, vol. C-27, no. 6, pp. 567–570, 1978.

[17] K. Chapman, "SEU Strategies for Virtex-5 Devices", *Xilinx Application Note, XAPP864 (v2.0)*, 2010.

[18] E. Petersen, *Single Event Effects in Aerospace*. John Wiley & Sons, 2011.