Online Test Strategies and Optimizations for Reliable Reconfigurable Architectures

Lars Bauer, Hongyan Zhang, Michael A. Kochte, Eric Schneider, Hans-Joachim Wunderlich, and Jörg Henkel

1 Introduction and Motivation

Runtime/reconfigurable architectures based on Field-Programmable Gate Arrays (FPGAs) are a promising augment to conventional processor architectures such as Central Processing Units (CPUs) and Graphic Processing Units (GPUs). Since the reconfigurable parts are typically manufactured in the latest technology, they may suffer from aging and environmentally induced dependability threats. In this chapter, strategic online test methods for dependable runtime-reconfigurable architectures as well as cross-layer optimizations for high reliability and lifetime are developed. Firstly, two orthogonal online tests are proposed that ensure reliable configuration of the reconfigurable fabric and aid fault detection. Secondly, a novel design method called module diversification is presented that enables self-repair of the system in case of faults caused by degradation effects as well as single-event upsets in the configuration. Thirdly, a novel stress-aware placement method is proposed that aims for slowing down system degradation by aging effects. The combined methods ensure reliable operation across architectural and gate level and allow to prolong the lifetime of dependable runtime-reconfigurable architectures.

The dependable operation of VLSI circuits is not only threatened by test escapes, intermittent or transient errors, but also by emerging hardware defects due to *aging* [11–13]. In nano-scale CMOS circuits, aging is related to *stress* which is defined as the condition under which a circuit structure experiences electrical and physical

L. Bauer (✉) · H. Zhang · J. Henkel
Chair for Embedded Systems, Karlsruhe Institute of Technology, Karlsruhe, Germany
e-mail: lars.bauer@kit.edu; henkel@kit.edu

M. A. Kochte · E. Schneider · H.-J. Wunderlich
Institute of Computer Architecture and Computer Engineering, University of Stuttgart, Stuttgart, Germany
e-mail: schneiec@iti.uni-stuttgart.de; wu@informatik.uni-stuttgart.de

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_12

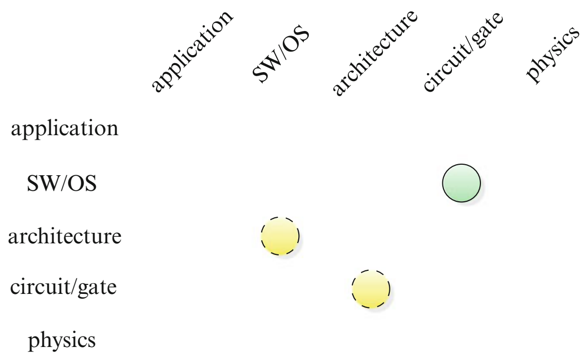


Fig. 1 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

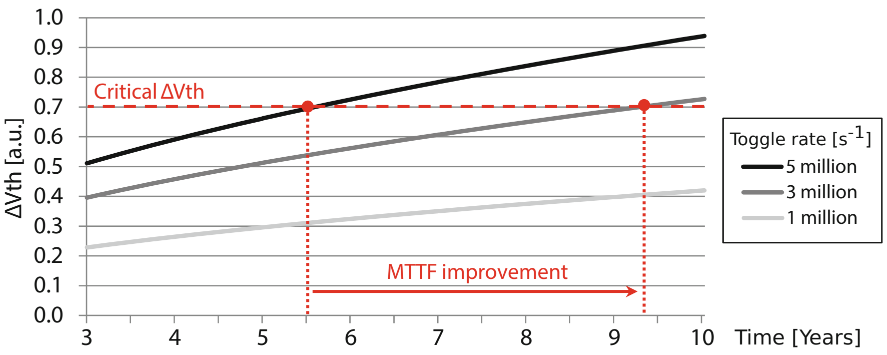


Fig. 2 Threshold voltage increase due to HCI-related stress (based on [22])

degradations. Two types of stress are distinguished: *static stress* and *dynamic stress*. Dynamic stress is typically characterized by the toggle rate of a transistor during which high currents flow between drain and source. A transistor is under static stress when an electric field is exerted across its gate oxide to induce a conducting channel. The stress is characterized by the duty cycle, i.e., the fraction of operation time the transistor is conducting. Dynamic stress leads to aging effects like Hot Carrier Injection (HCI), while static stress can lead to Bias Temperature Instability (BTI). Both are dominating aging mechanisms in nano-CMOS technologies [8, 16] and cause shifts in the threshold voltage ΔV_{th} of a transistor, which ultimately impacts the device performance over time. In this chapter, strategic online test methods for dependable runtime-reconfigurable architectures as well as cross-layer optimizations for high reliability and lifetime are developed (see Fig. 1).

The Mean Time to Failure (MTTF) of a transistor is defined as the time until its threshold voltage exceeds a certain critical value at which the transistor cannot deliver the required performance anymore. As shown in Fig. 2, the MTTF can be greatly increased if the transistor stress and consequently the threshold voltage shift are reduced.

Different aging models exist [2, 8], which indicate that both dynamic and static stress are generally *additive* through accumulation of the degradation effects. As a result, this additive stress accumulation causes a *monotonic* increase in the transistor degradation over long terms. Although BTI degradation may experience a recovery effect, the recovery requires complex conditions or long relaxation periods [10] and will thus hardly affect the additive property. The monotonic and additive properties allow to consider stress during runtime (e.g., for resource management) with limited computational resources.

1.1 Application Model

In this work, a general application model is considered, as shown in Fig. 3. An application (Fig. 3a) consists of a mixture of normal operations, e.g., memory allocation and data preparation, and one or multiple computationally intensive parts, the so-called *kernels*. A kernel (Fig. 3b) corresponds to an outer loop that iterates through the whole data set and that contains one or multiple inner loops that work on small data parts, specified by the current iteration of the outer loop. For example, in a stencil operation of an image, the outer loop iterates over each output pixel and the inner loop computes the output value based on multiple neighboring input pixel values. Such an inner loop is a good candidate to be implemented as a *Special Instruction* (SI) that is composed of one or multiple *accelerators* of potentially different types. An SI (Fig. 3c) is represented by a data-flow graph (DFG) where each node corresponds to an accelerator and the edges correspond to data-flow between the accelerators [4]. Before the execution of an SI, all required accelerators need to be configured into the reconfigurable fabric, or otherwise the SI has to be emulated in software on the GPP. A sophisticated H.264 video encoder is the main application used for evaluation. The encoder consists of three kernels that require different SIs, implemented by nine types of accelerators [6].

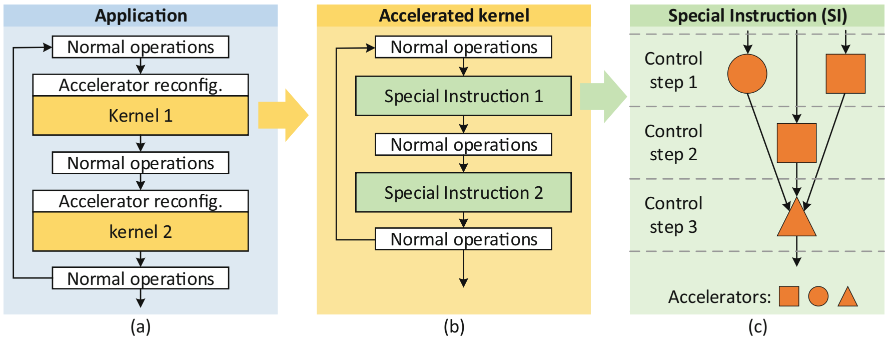


Fig. 3 This generic application model considers applications that consist of one or multiple kernels that may use Special Instructions (SIs) that are implemented by accelerators (based on [23])

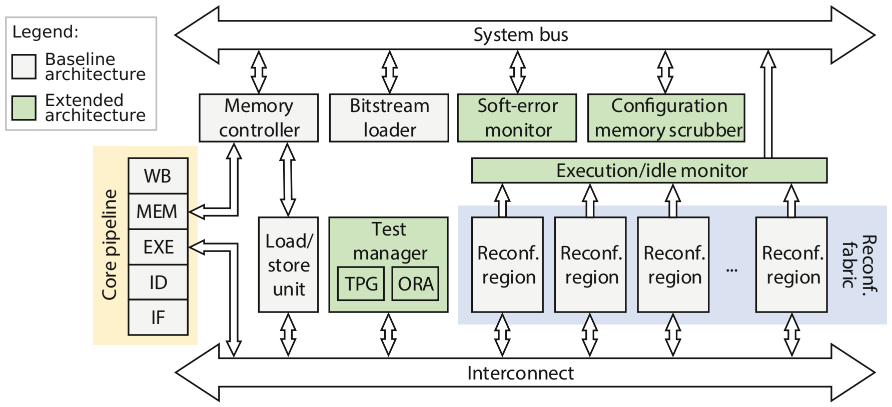


Fig. 4 Target reconfigurable architecture (based on [7])

1.2 Runtime-Reconfigurable Architectures

Runtime reconfiguration enables dynamic hardware customization to adapt to changing application requirements or environmental constraints, which maximizes performance at very low energy consumption. A reconfigurable architecture consists of a general-purpose processor and a *reconfigurable fabric*, partitioned into multiple *reconfigurable regions* (used to implement application-specific accelerators on-demand) that are interconnected via a communication infrastructure.

This chapter presents *Online Test Strategies for Reliable Reconfigurable Architectures* (OTERA), which targets FPGA-based fine-grained reconfigurable architectures as shown in Fig. 4. While transient faults due to single-event upsets are also addressed by OTERA (more details in [7]), this chapter focuses on aging-related challenges. To support dependable operation by online testing, stress balancing, and resource management for reliability and graceful degradation, a reconfigurable baseline architecture is extended by the following components:

- a test manager including a *test-pattern generator* (TPG) and an *output response analyzer* (ORA) to perform structural tests on the reconfigurable fabric and functional tests on the reconfigured accelerators;
- a workload monitor to track when a region is reconfigured and how often the currently configured accelerator is executed, which is used for stress estimation;
- a configuration memory scrubber to detect and correct errors in the configuration memory by periodical read-back and check of the configuration;
- a runtime system for dynamic dependability management by environmental monitoring, online test, reliability management, and aging mitigation.

The architecture is implemented using a LEON processor [9] and a parameterizable number of reconfigurable regions. A SystemC-based cycle-accurate simulator is used to evaluate the architecture and its runtime system. A hardware prototype is developed on a Xilinx Virtex-5 FPGA and operates at a clock frequency of 100 MHz with a reconfiguration bandwidth of 50 MB/s.

FPGA hardware is composed of a two-dimensional array of reconfigurable primitive logic elements and routing structures that logic functions are mapped to. The two essential components are Configurable Logic Blocks (CLBs) and Programmable Switching Matrices (PSMs). The CLBs are the basic reconfigurable resources for implementing combinatorial and sequential logic functions. The interconnection between the components is configured using the PSMs. The logic function in a reconfigurable region is determined by configuration bits, called its *bitstream*, stored in SRAM-based configuration memory. Modern FPGAs support partial reconfiguration and allow to change the logic function without interrupting the operation in other parts of the chip [19].

An FPGA-based reconfigurable fabric, manufactured in latest technology nodes (e.g., 16 nm for Xilinx' UltraScale+ family), may suffer from degradation due to aging [10, 18]. Due to the increasing susceptibility of ever-shrinking nano-CMOS devices, these effects cannot be ignored anymore [11–13]. The resilience of the reconfigurable fabric is essential to the dependability of reconfigurable architectures, since most of the application's computations are offloaded to the fabric. The dependable operation of a hardware accelerator in the reconfigurable fabric relies on both the structural integrity of the fabric and the accelerator's functional correctness. While structural integrity of the reconfigurable fabric is a prerequisite for functional correctness of accelerators, the latter requires the correct completion of the reconfiguration process and correctness of the configuration data. However, the functionality of accelerators can be impacted during operation, for instance by SEUs that corrupt configuration data [7] as well as degradation of the hardware. To increase the dependability of the reconfigurable architecture, the structural integrity and functional correctness need to be addressed at different layers.

2 Fault Detection Through Strategic Online Testing

As latent defects and aging threaten the structural integrity of nano-CMOS devices, conventional manufacturing and burn-in tests are no longer sufficient to guarantee dependable operation over the whole lifetime. Therefore, *online tests* are required to check the system functionality. This task is particularly challenging for runtime-reconfigurable architectures, since the hardware organization changes during runtime as part of the normal operation [4]. This chapter presents two complementing types of online tests that are scheduled concurrently by the runtime system: *pre-configuration online tests (PRET)* and *post-configuration online tests (PORT)*.

2.1 Generation and Runtime Scheduling of Online Tests

PRET is designed to exhaustively test the underlying hardware structure in the reconfigurable fabric (e.g., logic resources in CLBs) periodically or on-demand. For PRET, an array-based structural test approach is used to generate *test configurations* for the exhaustive test of all logic resources in a reconfigurable region [1, 5]. Additional PRET test configurations are generated to target the application-dependent interconnects [6].

Since errors may also occur during the loading of bitstreams (e.g., due to faults in the configuration logic or transient events like SEUs), the configured function of the targeted region may be wrong or the configuration in other parts of the reconfigurable fabric may be adversely altered. For this reason, PORT is designed to perform at-speed functional tests on accelerators after their instantiation to ensure that they were configured correctly. At runtime, PORT also periodically checks the accelerators for malfunctions due to emergent permanent faults or soft errors in the configuration memory. An Automatic Test Pattern Generation (ATPG) tool is used to generate accelerator-specific test patterns to target the LUTs, combinational functions, and sequential elements in CLBs, as well as interconnects. The stuck-at fault model is used for components for which sufficient structural information is available to derive the faults and for the interconnects. For the remaining components, structural and cell faults are targeted during test generation resulting in a hybrid fault model [6].

Figure 5 shows the proposed online test flow for a reconfigurable fabric with three regions. In the first step (Fig. 5a), the runtime system decides that an accelerator shall be reconfigured into a particular region, which triggers the demand to test the hardware structures in that region before the actual configuration of accelerators (the so-called *on-demand PRET*). To exhaustively test all reconfigurable resources in the region, multiple *test configurations* (TCs) are required. The runtime system can choose to execute PRET incrementally to reduce the delay, applying only a subset of TCs (possibly none) prior to an accelerator reconfiguration. In practice, on-demand PRET-TCs are only scheduled after a certain number of *accelerator configurations* (ACs) have been configured. To reduce the impact on the application performance due to unavailable regions, PRET is only executed at times when the system needs to be reconfigured anyway. The runtime system tracks which TCs were applied to a region in the past and how much time passed since the last exhaustive PRET. Depending on this history, it activates PRET prior to an AC, reconfigures the selected TCs into the region, and uses TPG and ORA of the Test Manager to exercise the region (Fig. 5b).

In addition to on-demand PRETs, the runtime system also schedules *periodic* PRETs to ensure that seldom-reconfigured regions are properly tested. Note that PRET also needs to be executed regularly for regions that the application only reconfigures once and then never again (e.g., if the application only consists of one kernel; see Sect. 1.1). The reason is that PORT—despite its generally high fault coverage (see [6])—cannot always identify all faults. For instance, when an

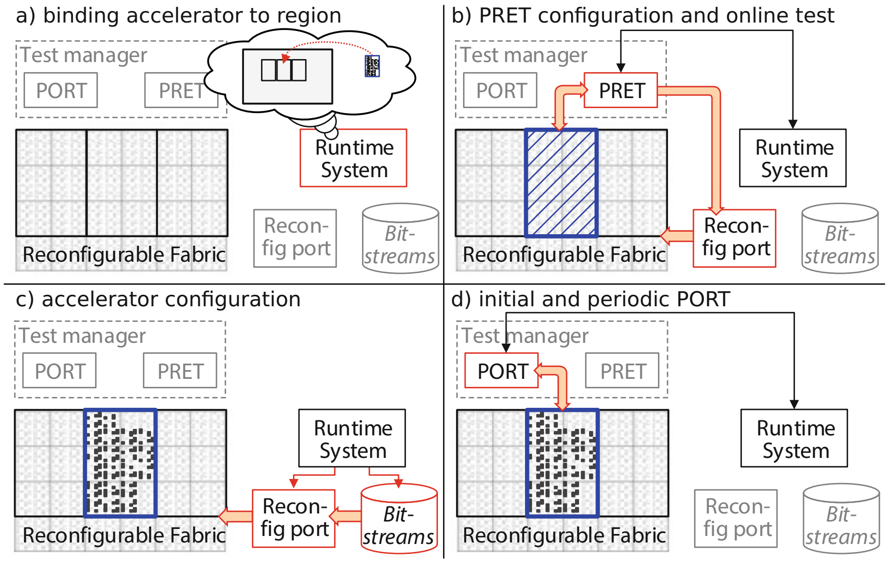


Fig. 5 Test flow with PRET and PORT (based on [6])

accelerator contains internal state, it is not always possible to apply an input value that propagates a possibly faulty value to an observable output. The periodic PRET is implemented using a timer interrupt and a handler that consists of two phases: (1) triggering the reconfiguration of a TC for a particular region and (2) executing PRET after the TC is reconfigured.

If no structural fault is found by PRET, the runtime system reconfigures the desired accelerator into the region (Fig. 5c). Before the accelerator is used by the application, the runtime system triggers an *on-demand PORT* (Fig. 5d) to test whether the reconfiguration process has completed without error. Additionally, accelerators instantiated in other regions are tested as well to check that they were not adversely affected by the reconfiguration. As PORT does not require any reconfiguration of TCs, it operates significantly faster than PRET and is also scheduled periodically during normal operation.

2.2 Online Test Integration

The test manager, TPG, and ORA are integrated into the reconfigurable architecture and coupled to the interconnect for the reconfigurable fabric such that communication channels between the regions and the test manager can be established. PRET and PORT are implemented as dedicated test-SI. In the base architecture, all SIs implicitly configure the interconnect infrastructure for the required data-flow among accelerators and the system. The test-SIs reuse this mechanism to establish the connections between the test manager and the regions under test.

When the runtime system initiates a test-SI, the test parameters such as the target region or selection of test patterns are sent as the SI input data from the register file of the processor to the test manager. The test manager then generates the patterns by the TPG or sends stored patterns to the regions. While the PRET responses are sent back to the test manager for comparison, the PORT responses are compacted locally in space and time using a 32-bit multiple input signature register (MISR). The MISR is integrated into the interconnect infrastructure such that the outputs and the bus interface of a region are tested as well. After the test, the locally stored signatures are transferred to the test manager and compared with the expected signatures that are specific for each accelerator. At the end of PRET, the pass/fail information is written back to the register file of the processor. On-demand PORT is executed directly after an accelerator configuration to assure that the reconfiguration process completed without error and that the configured accelerator delivers the expected functionality. As PORT tests *all* configured accelerators in one test session, errors in the other accelerators, e.g., due to address decoder faults, are detected as well.

2.3 Experimental Evaluation

The effectiveness of PRET and PORT as well as the impact on the system performance is evaluated for the targeted platform. A test session consists of multiple test configurations (TCs) as shown in Table 1. In total nine TCs are required to test all logic primitives in the CLBs [1], and another nine TCs are required to test the interconnects of the accelerators of the H.264 application [6]. Each TC tests a subset of the logic primitives in the CLBs of a region or a set of interconnects used by the accelerator to be configured (Column 2). Columns 3 and 4 give the area overhead of PRET and the size of the generated partial bitstreams. The total area overhead introduced by PRET for all TCs is 17 CLBs. That is a one-time overhead to implement the test-pattern generator (TPG) and output response analyzer (ORA) for PRET, independent of which reconfigurable region is to be tested, whereas the other numbers in the table are per reconfigurable region. Note that the configuration time with tens of thousands of cycles dominates the actual application of the test patterns (Column 6).

The PRET overhead for the interconnect TCs is not applicable as the deterministic patterns are not generated by a TPG but stored similar to PORT patterns. The responses are compacted in the MISR introduced for PORT. In total 3780 bytes are required to store the test patterns of all interconnect TCs together with their signatures. The interconnect test reaches a fault coverage of up to 100% with the lowest being 98.28% [6].

The application performance loss introduced by PRET depends on the test frequency and number of reconfigurable regions. In this experiment, architectures

Table 1 Test configurations for CLBs and interconnects for reconfigurable regions of 4 × 20 CLBs (based on [6])

TC	Tested primitives	PRET overhead [CLBs]	Bitstream size [KB]	Freq. [MHz]	Test length [Patterns]
1	LUT conf. as XOR, connected to FF	2	24.0	207	64
2	LUT conf. as XNOR, connected to FF	2	24.0	207	64
3	Carry MUX, interleaved with MUX and latch	1	28.6	168	6
4	Carry MUX, interleaved with MUX and latch	1	26.1	154	6
5	Carry XOR, interleaved with MUX and FF	1	28.0	168	6
6	Carry XOR, interleaved with MUX and FF	1	28.2	154	6
7	Carry-in/-out with multiplexed scan chain	1	27.1	183	6
8	LUT conf. as SR with slice MUX	1	22.9	157	6
9	LUT conf. as RAM with slice output	7	22.3	225	320
10–18	Interconnect and PIPs of 9 accelerators	n.a.	29.6	78.8–191.9	13–123

with 5 and up to 14 reconfigurable regions are considered. The PRET handler is triggered every 1 ms and performs PRET if a region has not been tested for 500 ms. The observed test latencies until a region is completely tested ranged from 3.8 to 8.1 s, i.e., emergent faults do not remain undetected in the system for longer than 1.9 to 4.05 s on average. Table 2 reports the PORT performance impact and test latency. The upper part of the table shows the performance impact for PORT frequencies from 143 to 1000 Hz, i.e., test intervals from 1 to 7 ms. For each PORT frequency, the table shows the minimum and maximum performance loss of ten reconfigurable systems with different number of regions (5–14). The performance overhead due to PORT is very low (between 0.51% and 3.73%) and scales well with higher PORT frequencies. The observed worst case test latency, which corresponds to the longest untested time period of a region, is shown in the lower part of Table 2.

With PRET and PORT both enabled, the system is able to defend the configured accelerators against structural faults induced by aging effects or latent faults and transient events such as radiation [6]. For a PORT frequency of less than 100 Hz, the performance loss was dominated by the configuration frequency. After that point, the PORT frequency dominates the performance loss. The highest observed performance loss of only 4.4% occurs for a PORT frequency of 1000 Hz and a configuration frequency of 41 Hz.

Table 2 Performance loss and worst case test latency under PORT (based on [6])

		PORT application frequency [Hz]						
		143	167	200	250	333	500	1000
Performance loss	min.[a] [%]	0.51	0.59	0.72	0.89	1.20	1.81	3.68
	max.[a] [%]	0.56	0.63	0.75	0.92	1.23	1.85	3.73
Worst case test latency[b]	min.[a] [ms]	7.0	6.0	5.0	4.1	3.3	2.3	1.7
	max.[a] [ms]	7.8	6.8	5.8	4.8	3.8	2.8	1.8

[a]Summarizing ten reconfigurable systems with 5–14 regions
[b]Corresponds to the longest time period in the whole runtime in which a configured accelerator remains untested

3 Self-Repair by Module Diversification

Using PRET and PORT we can *detect* faults in the reconfigurable fabric. We now present a design method called *module diversification* [21] that generates a set of *diversified* configurations for each module/accelerator to *tolerate* any single-CLB fault and part of multi-CLB faults. The diversified configurations of an accelerator provide all the same functionality, but they vary in their CLB usage. They are reconfigured into the region at runtime without performance degradation. If a faulty CLB is detected, it is isolated from the system (i.e., a configuration is chosen that does not use it) to avoid any errors.

3.1 *Diversified Configurations*

A module defines the logic functions to be implemented in a region which consists of CLBs that are arranged regularly in a 2-dimensional array in the FPGA fabric. The CLB usage of a configuration is described by a *configuration matrix* as shown in Eq. (1) whose dimensions $X \times Y$ match the width X and height Y of a region in CLBs. If a configuration uses a certain CLB, the corresponding element in the matrix is 1, otherwise 0. For each module, a set $C = \{\mathbf{A_1}, \cdots, \mathbf{A_w}\}$ of configurations matrices with different CLB usage is generated. To be able to tolerate any single-CLB fault, this set of configurations must satisfy the *completeness condition* (Eq. (2)), which ensures that for any CLB in a region at least one diversified configuration $\mathbf{A_i}$ exists where the CLB is not used. Given that all diversified configurations implemented in a $X \times Y$ region occupy the same amount $U(< X \cdot Y)$ of CLBs (with at least one free CLB) a minimum number of w_{min} configurations (Eq. (3)) is required for the completeness condition [21].

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{1}$$

$$\forall x, y, 1 \leq x \leq X, 1 \leq y \leq Y : \exists \mathbf{A_i} \in C \text{ with } [\mathbf{A_i}]_{x,y} = 0 \tag{2}$$

$$w_{min} := \left\lceil \frac{X \cdot Y}{X \cdot Y - U} \right\rceil \tag{3}$$

Two configurations $\mathbf{A}_i, \mathbf{A}_j \in C$ are said to be *maximally diversified* if their difference in the CLB usage is maximized. The *max diversification condition* [21] states that for every configuration $\mathbf{A_i} \in C$ there exists a maximally diversified configuration $\mathbf{A_j} \in C$ with a common number of CLBs:

$$\forall i, 1 \leq i \leq w_{min} : \exists \mathbf{A_j} \in C, j \neq i \text{ such that}$$
$$\sum_{x,y} \left([\mathbf{A_i}]_{xy} \cdot \left[\mathbf{A_j}\right]_{xy} \right) = \begin{cases} 2U - X \cdot Y & \text{if } \left(U > \frac{1}{2} X \cdot Y\right) \\ 0 & \text{else.} \end{cases} \tag{4}$$

3.2 Generation Algorithm

Algorithm 1 allows to generate maximally diversified configurations that satisfy the completeness condition [21]. Starting from an initial configuration $\mathbf{A_1}$ (Line 1) of a module, it incrementally generates diversified versions. A score matrix $\mathbf{G}$ stores

Algorithm 1 Generation of diversified configurations C

```
1.  $C := \{\mathbf{A_1}\}$ // $\mathbf{A_1}$ is the initial configuration $(X \times Y)$
2.  $\mathbf{G} := \mathbf{A_1}$ // Score matrix $\mathbf{G}$ stores swapping priority of CLBs $(X \times Y)$
3.  $\mathbf{A_{new}} := \mathbf{A_1}$
4.  while $|C| \neq$ desired number of config. $\wedge\ |C| \neq \binom{XY}{U}$ do
5.      zero_elem_list := $\{(x, y) \mid [\mathbf{A_{new}}]_{xy} = 0\}$ // unused CLBs
6.      cand_list := $\{(x, y) \mid [\mathbf{A_{new}}]_{xy} = 1\}$ // candidate list
7.      sort cand_list in descending order according to the score in $\mathbf{G}_{xy}$
8.      for all $(x, y)$ in zero_elem_list do
9.          swap_candidates := $\{(p, q) \mid (p, q) \in$ cand_list and $\mathbf{G}_{pq} = \mathbf{G}_{\texttt{cand\_list[0]}}\}$ //
            all CLBs with the highest score
10.         farthest_swap_candidate := $(p, q) \in$ swap_candidates with max.
            Manhattan distance between $(x, y)$ and $(p, q)$
11.         swap($[\mathbf{A_{new}}]_{xy}$, $[\mathbf{A_{new}}]_{\texttt{farthest\_swap\_candidate}}$)
12.         cand_list.pop(farthest_swap_candidate)
13.         if cand_list = $\emptyset$ then
14.             break
15.         end if
16.     end for
17.     while $\mathbf{A_{new}} \in C$ do
18.         swap a random zero- with random one-element in $\mathbf{A_{new}}$
19.     end while
20.     $\mathbf{G} := \mathbf{G} + \mathbf{A_{new}}$ // update CLB score
21.     $C := C \cup \{\mathbf{A_{new}}\}$
22. end while
```

for each CLB the number of available diversified configurations in C that use the respective CLB resources. The new configuration matrix $\mathbf{A_{new}}$ is initialized by $\mathbf{A_1}$ and modified in the inner loop (Lines 8–16) by swapping zero- and one-elements. The loop iterates over each element in $\mathbf{A_{new}}$ and swaps all zero-elements with one-elements in an order given by the score matrix (Line 7). If a CLB has a higher score, it is used more often in the diversified configurations. Thus the corresponding one-element in $\mathbf{A_{new}}$ will be swapped first. If CLBs have the same score, the distance-wise farthest one from the current zero-element is swapped first (Lines 9–11) so that the used CLBs are located near each other in the resulting configuration. The first w_{min} generated configurations correspond to the *minimal* set of configurations [21]. More configurations can be generated to achieve higher reliability or more alternatives during stress balancing (see Sect. 4). Random swapping in Line 18 allows to shuffle CLBs with different stress profiles. The algorithm terminates when either the desired number of configurations or all possible configurations have been generated.

3.3 Experimental Evaluation

To evaluate the reliability improvement and timing costs, the presented method is applied to a set of functional modules from the MCNC benchmark suite [20] and OpenCores.[1] The dimensions of the reconfigurable regions were chosen as 20 CLBs in height (80 CLBs for large modules) and 3–13 CLBs in width, which provides different degrees of CLB redundancy. For each module and region size the minimal set of configurations is generated using the proposed module diversification method. Since the design method applies additional constraints to prohibit certain CLB placements (`PROHIBIT` commands in Xilinx tools), additional routing effort is introduced that can affect the maximum clock frequency. To assess the impact on the system performance, the maximum frequency of diversified modules was compared to the original configuration. Initially, the clock frequencies of the modules ranged from 122.4 MHz (`apex2`) to 150.8 MHz (`pdc`). Experiments show that the timing penalty of the diversified configurations ranges from 0.04% (`aes_core`) to 9.7% (`misex3`). While the maximal frequency is given by the slowest configuration of a module, the original implementation also belongs to the configuration set and can be used when full performance is required. Also, if the system frequency is lower than the maximal frequency of the diversified modules, there are no timing penalties at all. Thus, module diversification is a promising approach to obtain fault tolerance without additional area overhead and little to no cost in system performance.

The *reliability* of an entity is the probability that the entity can operate without failure over a time period t. Without any fault-tolerance techniques applied, the overall reliability of a module with U CLBs depends on the reliability $R_{\mathrm{CLB}}(t)$ of each individual CLB (Eq. (5)). With module diversification, the reliability of the

[1] https://www.opencores.org.

module changes, as shown in Eq. (6). The first term states the probability that all CLBs are fault free. The second term aggregates all possible scenarios of multiple fault occurrences until all CLBs become faulty. The fault coverage $C_f \in [0, 1]$ is the fraction of f-CLB faults which are detected by an online test or concurrent error detection scheme such that reconfiguration with a diversified configuration allows to continue the operation. The fraction of f-CLB faults which can be tolerated with the set of available configurations is denoted by $\alpha_f \in [0, 1]$.

$$R_{\text{No\_FT}}(t) = (R_{\text{CLB}}(t))^U \tag{5}$$

$$R_{Div}(t) = R_{\text{CLB}}(t)^{XY} + \sum_{f=1}^{XY} \underbrace{C_f \alpha_f \binom{XY}{f} (1 - R_{\text{CLB}}(t))^f \, R_{\text{CLB}}(t)^{(XY-f)}}_{\text{Probability that } f\text{-fold CLB failures can be tolerated}} \tag{6}$$

We use the module apex4 for the reliability analysis. Without fault-tolerance measures, the module has a very low reliability (≈0.91). Figure 6 shows the module reliability for a varying number of configurations and region sizes with CLB reliability $R_{\text{CLB}}(t) = 0.999$ and $C_f = 1.0$. The region size varies from 20×6 to 20×9 CLBs and corresponds to CLB redundancies from 22.4% to 111.8%. Larger region sizes reduce the overall module reliability since they have increased probability of a faulty CLB. By using diversified configurations, the module reliability increases dramatically. As shown, the tolerance of f-CLB faults rises with increasing number of configurations and very high module reliability is achieved (>0.999).

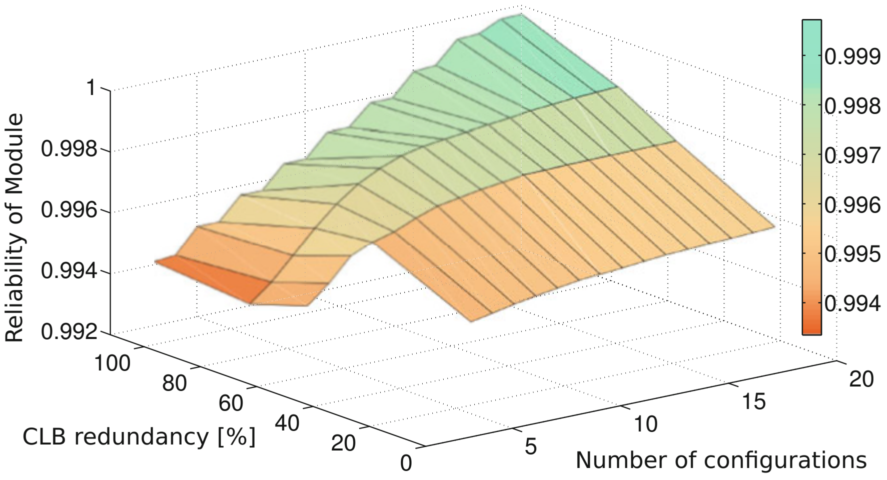


Fig. 6 Module reliability of apex4 for different ratios of CLB redundancy and number of configurations with CLB reliability 0.999 (based on [21])

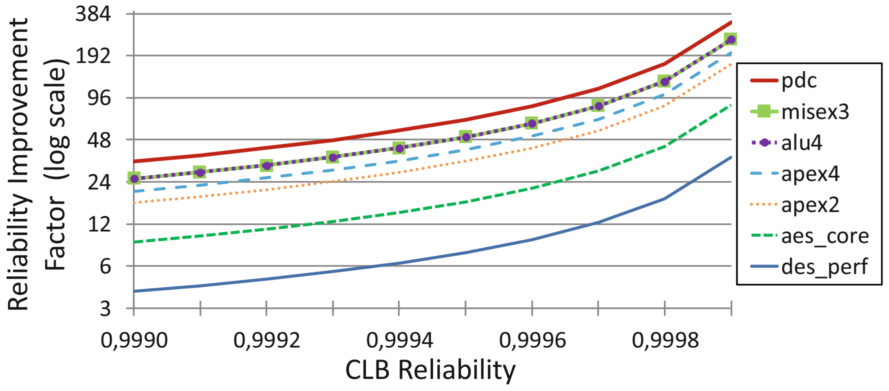


Fig. 7 Reliability improvement factor after module diversification (based on [21])

To estimate the effectiveness of the module diversification, the *reliability improvement factor* (RIF) is used [15]. The RIF is the ratio of the failure probability of the original system and the failure probability of the fault tolerant system using diversified module configurations (Eq. (7)). Figure 7 plots the RIF for the five investigated modules and CLB reliabilities ranging from 0.9990 to 0.9999. As shown, the proposed design method achieves reliability improvement factors of up to 330×.

$$\mathrm{RIF} := \frac{1 - R_{\mathrm{No\ FT}}}{1 - R_{Div}} \tag{7}$$

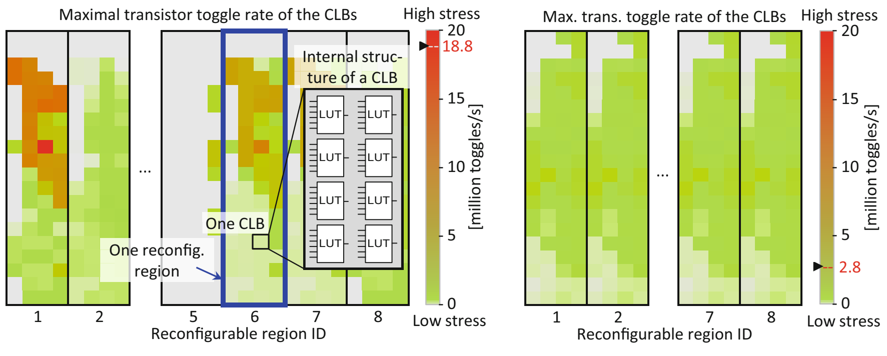
4 Prolonging Lifetime via Stress Balancing

In addition to *reacting* on detected faulty CLBs (e.g., by using diversified modules as in Sect. 3), it is of crucial importance to *proactively* delay the occurrence of permanent faults (or increasing transistor switching delay) by aging mitigation via stress balancing. Different aging mechanisms have been reported for the current generation of CMOS designs, as discussed in Sect. 1. The main causes of these effects are environmental and electrical *stress*. Stress can be induced in different ways, e.g., through the presence of strong electrical fields or high current density [17, 18]. We propose the novel STRess-Aware Placement method STRAP that reduces the peak stress by aging mitigation. It combines complex offline optimizations at synthesis time with situation-dependent adaptation at runtime to optimize the intra- and inter-region stress distribution simultaneously. At runtime, STRAP places accelerators to different reconfigurable regions (i.e., it decides to which region they shall be reconfigured) while considering the induced intra- and inter-region stress distribution simultaneously. At synthesis time, STRAP diversifies stress during place-and-route by preventing overlapping of high stress CLBs from different accelerators, which further improves the intra-region stress distribution at runtime.

4.1 Overview of the Stress-Aware Placement Method STRAP

The MTTF of a system is constrained by the component with the highest stress [17]. In order to prolong the MTTF of a reconfigurable fabric, stress accumulation on individual resources need to be avoid to reduce the peak stress. Figure 8a shows a typical reconfigurable fabric with 8 reconfigurable regions and 4 × 20 CLBs per region. The figure visualizes the distribution of HCI stress after running an H.264 video encoder. Higher HCI stress corresponds to more toggles per second of a transistor (see Sect. 1). For each CLB, the highest toggle rate of any transistor is identified and plotted in a color-scale from 0 (*low stress*, bright gray) to 20 million toggles per second (*high stress*, dark red). It is noticeable that several CLBs are not used (e.g., most parts of region 5), whereas some CLBs in region 1 contain transistors that are highly stressed. The latter represent *stress hotspots* where high stress accumulates in some of the components in the fabric which have a higher chance to fail much earlier than others, hence reducing the MTTF of the system.

The basic idea of STRAP is to place accelerators such that the *maximal* stress is minimized. Our *method* abstracts stress to the granularity of CLBs, whereas the *evaluation* of our method in Sect. 4.6 considers stress at transistor granularity. If the stress from a stress hotspot can be distributed to less stressed CLBs (like in region 5



(a) without stress balancing (b) with STRAP

Fig. 8 Transistor stress distribution in a reconf. fabric with eight regions; each region consists of 4 × 20 CLBs with 8 LUTs each (same setup as for evaluation); the color of a CLB corresponds to the highest toggle rate of any of its transistors; the symbol "filled triangle right" on the scale denotes the maximum stress over all regions (based on [22]). (**a**) Conventional execution without stress balancing. (**b**) Stress-aware placement in STRAP

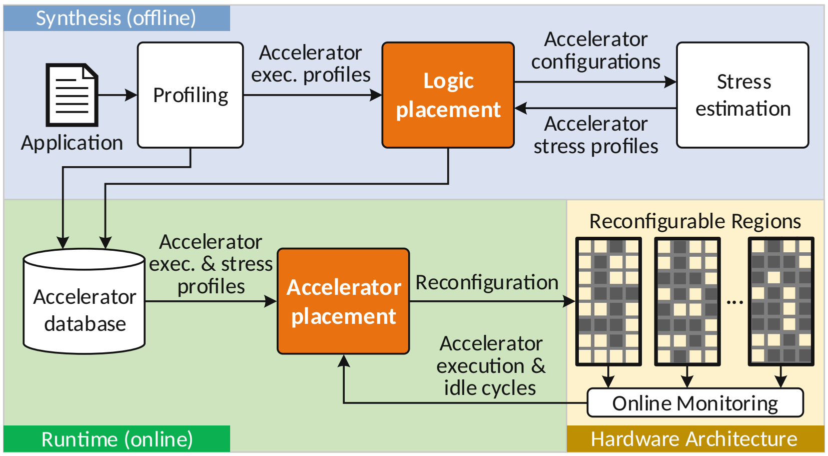


Fig. 9 Overview of the stress-aware placement method (based on [23])

in Fig. 8a), then the maximum stress in the reconfigurable regions is reduced (like in Fig. 8b), leading to increased MTTF.

Figure 9 provides an overview of the stress-aware placement method STRAP, showing the synthesis time techniques, the runtime techniques, and how they interact with the hardware architecture of a reconfigurable system. For logic placement at synthesis time, the challenge is to place-and-route accelerators in a way that supports stress balancing at runtime, but without having runtime information. STRAP first performs an offline application profiling of each application kernel to obtain estimates on (1) how often accelerators will be executed relative to each other and (2) how long each accelerator executes to finish its task. This information is used to steer runtime accelerator placement (Sect. 4.3) and synthesis time logic placement (Sect. 4.4).

Based on the accelerator configuration after place-and-route, the stress estimation process in Fig. 9 analyzes the signal activities in all CLBs used by the accelerator to obtain the information how much stress it induces to a reconfigurable region. Accelerator execution and stress profiles are stored together with the accelerator bitstreams in main memory for runtime decision making.

At runtime, STRAP decides into which reconfigurable region an accelerator shall be reconfigured, whenever the application demands different accelerators. It performs online monitoring of each region to track when the region was reconfigured last and how often the currently reconfigured accelerator was executed. Whenever a region is reconfigured, the execution counter and reconfiguration timestamp are read and reset. Together with the accelerator stress profile created at synthesis time, STRAP then calculates the exact *stress state* for all CLBs of the region. This information is used to decide the runtime accelerator placement.

4.2 Representation of Stress

Stress Granularity In order to handle the transistor stress in an algorithmic way, it needs to be represented compactly to allow an efficient runtime computation for the stress states of regions and the placement decision making. The transistors of a reconfigurable region are stressed by the reconfigured accelerator in a way that is determined by its logic functionality and input signal patterns. As the number of transistors in a region may be huge, the stress experienced by individual transistors is lumped to CLB granularity for the stress-aware placement method. *CLB stress* is defined as the sum of the stress experienced by all transistors in a CLB. With this definition, CLB stress preserves the additive property of transistor stress, i.e., the total stress a CLB experienced from different accelerators is the sum of the induced stress from individual accelerators.

Stress Accumulation With the established stress properties (see Sect. 1), the stress in the reconfigurable fabric can be described in a formal way. The stress state of a reconfigurable region (as it is visualized in Fig. 8) is denoted as matrix **S**, where each entry represents the stress experienced by the corresponding CLB in the region. The stress that a particular accelerator induces per clock cycle is obtained from offline stress estimation and called *unit stress*, denoted by a matrix of the same size as **S**. In general, the stress increase due to the work done by an accelerator is shown in Eq. (8). Matrices $\mathbf{s}_{\mathbf{exec}}^{\mathbf{unit}}$ and $\mathbf{s}_{\mathbf{idle}}^{\mathbf{unit}}$ denote the unit stress induced by the accelerator during execution or idle time and Sect. 4.6 explains how we use aging models to obtain these values by power/temperature analysis of placed-and-routed accelerators. Scalars τ_{exec} and τ_{idle} denote the number of clock cycles when the accelerator is executing or idle.

$$\mathbf{s} := \tau_{exec}\mathbf{s}_{\mathbf{exec}}^{\mathbf{unit}} + \tau_{idle}\mathbf{s}_{\mathbf{idle}}^{\mathbf{unit}} \tag{8}$$

The values for τ_{exec} and τ_{idle} are obtained from offline application profiling to construct the stress matrices (Eq. (8)) for every accelerator. The runtime system uses them to determine how much stress an accelerator would induce to a region *before* actually placing it. It also uses online monitoring (see Sect. 4.1) that provides the actual number of accelerator executions and idle times for each region *after* a computational kernel finished execution. This allows to keep track of the actual stress that a region experienced, which is the starting point for the next placement decision.

4.3 Runtime Accelerator Placement

The reconfigurable fabric consists of N equally sized rectangular regions. During runtime, the application requests to configure $M \leq N$ accelerators to speed up its computational kernels. The runtime system has to decide to which regions the M accelerators shall be configured, by first deciding which $N - M$ regions shall *not* be reconfigured, e.g., by using a least recently used replacement policy. The decision to which of the remaining regions an accelerator is placed does not affect the application performance, but it affects the stress applied to the regions.

Each region contains $X \times Y$ CLBs with an (x, y) coordinate. The stress experienced so far by the CLBs in region k is denoted as $[\mathbf{S_k}]_{xy}$ and the stress that will be induced by an accelerator j is denoted as $[\mathbf{s_j}]_{xy}$ (see Eq. (8)). It depends on how often the accelerator will be executed, as determined by offline profiling (see Sect. 4.1). If an accelerator j is placed into region k, then the accelerator executions increase the stress state of the region to $\mathbf{S'_k} = \mathbf{S_k} + \mathbf{s_j}$. The challenge is to place each accelerator to a region, such that upon completion of the application kernel the maximum CLB stress over the N regions is minimized, i.e., $\max_{k,x,y} [\mathbf{S'_k}]_{xy}$ is minimized. It can be easily seen that the strict lower bound of the maximum CLB stress is given by Eq. (9), which is reached if and only if the stress is uniformly distributed over all CLBs. To achieve this at runtime, we propose a heuristic that follows these two rules: (1) maximal utilization of under-stressed CLBs within one region, i.e., the stress shall be evenly distributed among different CLBs within the region (*intra-region* distribution) and (2) avoid placing high stress accelerators into highly stressed regions, i.e. the stress shall be evenly distributed among different regions (*inter-region* distribution). The heuristic uses a profit function (Eq. (10)) for placing accelerator j into region k that considers the stress distribution within one region and across all regions, respectively.

$$\frac{1}{NXY}\left(\sum_{k}^{N}\sum_{x,y}[\mathbf{S_k}]_{xy} + \sum_{j}^{M}\sum_{x,y}\left[\mathbf{s_j}\right]_{xy}\right) \tag{9}$$

$$\text{Profit}_{jk} = \text{Profit}_{jk}^{intra} + \text{Profit}_{jk}^{inter} \tag{10}$$

To calculate $\text{Profit}_{jk}^{intra}$, the average CLB stress in region k is determined as AvgStress_k and then used to calculate the absolute deviation of the stress of CLB_{xy} in region k from AvgStress_k. The sum over all CLBs in region k denotes the intra-region stress imbalance. It is calculated (1) before placing accelerator j to region k and (2) after hypothetically placing it. The difference of these two values corresponds to the degree of increased stress imbalance if placing accelerator j to region k and is used as $\text{Profit}_{jk}^{intra}$. The idea for $\text{Profit}_{jk}^{inter}$ is very similar. There, the stress of region k is compared with the average stress of all regions before and after hypothetically placing accelerator j to region k [22].

The stress-aware runtime accelerator placement iterates over all required accelerators. In each iteration, it calculates the profits of placing the accelerator into all available regions and then places the accelerator into the region that provides the highest profit. The complexity of this algorithm is $\mathcal{O}\left(M^2XY\right)$. If the application does not reconfigure a region for a longer time, then this region would be constantly stressed by one accelerator without stress redistribution. As a solution, the runtime accelerator placement forces that region to be reconfigured after a user-defined time period that should not be too short to prevent increased reconfiguration overhead and also not too long to avoid stress accumulation. For instance, a time period of 100 million cycles (1 s at 100 MHz) is short enough to avoid aging accumulation and the induced application performance degradation is only 0.21%.

4.4 Synthesis Time Logic Placement

Our runtime accelerator placement uniformly distributes the stress over all reconfigurable regions, compared to the stress-unaware placement. The maximal transistor toggle rate is reduced by more than 73% from 18.8 million toggles/s (see Fig. 8a) down to 5.0. However, when high stress CLBs of different accelerators *overlap* at the same relative (x, y) location, the runtime accelerator placement cannot achieve intra-region stress distribution. STRAP addresses this problem by applying placement constraints at *synthesis time* to diversify (similar to Sect. 3.1) the CLB usage among different accelerators, which reduces the overlapping of high stress CLBs. To minimize the timing impact on accelerators, STRAP only constrains which CLBs shall be used and leaves everything else to the vendor place-and-route algorithm.

The logic placement algorithm (Algorithm 2) diversifies the high stress CLBs of different accelerators to different CLB locations in the regions. First, unconstrained configurations of all accelerators are generated (Lines 1–5). For each accelerator

Algorithm 2 Stress-diversifying logic placement

```
Input: List of accelerators Acc.
1.  for j := 1 to len(Acc) do
2.    Place-and-route Acc[j] without any placement constraints
3.    $\mathbf{s_j}$ := get_stress(Acc[j])
4.    Acc[j].max_freq := get_max_freq(Acc[j])
5.  end for
6.  Acc := sort_ascending(Acc, key=max_freq)
7.  $\mathbf{R}$ := $\mathbf{s_1}$
8.  for j := 2 to len(Acc) do
9.    prohibit_xy := ∅
10.   for x := 1 to Acc[j].n_cols do
11.     for y := 1 to Acc[j].n_rows do
12.       if Condition Eq. (11) is satisfied for (x,y) then
13.         prohibit_xy.add((x,y))
14.       end if
15.     end for
16.   end for
17.   Place-and-route Acc[j] with prohibited CLB locations listed in prohibit_xy
18.   if Place-and-route failed then
19.     prohibit_xy.remove$\left( argmin_{xy \in \texttt{prohibit\_xy}} \left\{ \left[ \hat{\mathbf{R}} + \hat{\mathbf{s}}_\mathbf{j} \right]_{xy} \right\} \right)$
20.     goto Line 17
21.   end if
22.   $\mathbf{R}$ := $\mathbf{R}$ + get_stress(Acc[j])
23. end for
```

configuration the CLB stress is estimated (see Sect. 4.2), and the maximal achievable frequency is extracted from the place-and-route log files (Lines 3–4). The generated initial configurations are then sorted in ascending order of their maximal achievable frequencies (Line 6). The fabric typically runs at the frequency of the slowest accelerator f_{min}. In order to minimize the impact on system performance, it is placed and routed without stress-diversifying placement constraints. Its CLB stress distribution is taken as the initial reference distribution (Line 7). As long as the proposed logic placement does not reduce the frequency of an accelerator below f_{min}, there is no performance impact/penalty for the whole system. During the generation of other accelerator configurations, $\mathbf{R}$ keeps track of the sum of the stress distribution of all $j-1$ previously generated accelerators, i.e., $\mathbf{R} = \sum_{i=1}^{j-1} \mathbf{s_i}$.

The remaining accelerators will be placed-and-routed again in ascending order of their maximal frequencies (Lines 8–23). To avoid that high stress CLBs of the currently placed accelerator `Acc[j]` overlap with those in previously placed accelerators `Acc[1],...,Acc[j-1]`, we prohibit the placement to specific CLB locations for `Acc[j]` (Lines 9–17) if Eq. (11) is satisfied, where L_j is the number of used CLBs by the currently place-and-routed accelerator `Acc[j]`. $\hat{\mathbf{R}}$ and $\hat{\mathbf{s}}_\mathbf{j}$ are normalized stress matrices of $\mathbf{R}$ and $\mathbf{s_j}$. In earlier iterations, the reference distribution is less even, which implies that few CLB locations in the reference distribution have much higher values than the others, and therefore it is less likely that the condition in Eq. (11) is satisfied. In turn, fewer locations are prohibited for placement in earlier iterations, which implies less timing impact on slower accelerators. If place-and-route fails due to too many prohibited CLB locations, the locations xy where the stress overlapping $[\hat{\mathbf{R}} + \hat{\mathbf{s}}_\mathbf{j}]_{xy}$ is lowest are removed from `prohibit_xy` (Line 19), and place-and-route is re-executed with the relaxed constraints.

$$\left[\hat{\mathbf{R}}\right]_{xy} > \frac{1}{L_j} \sum_{uv} [\hat{\mathbf{s}}_\mathbf{j}]_{uv}$$
$$\text{with } \hat{\mathbf{R}} = \frac{\mathbf{R}}{\max_{uv} [\mathbf{R}]_{uv}} \text{ and } \hat{\mathbf{s}}_\mathbf{j} = \frac{\mathbf{s_j}}{\max_{uv} \left[\mathbf{s_j}\right]_{uv}} \tag{11}$$

With synthesis time stress diversification, high stress CLBs from different accelerators are placed to different CLB locations, and thus better intra-region stress distribution can be achieved during runtime placement. After applying both stress-aware runtime placement and synthesis time stress diversification for dynamic stress, the maximal transistor toggle rate is further reduced by additional 44% from 5.0 million toggles/s down to 2.8 (see Fig. 8b).

4.5 *Extended Accelerator Placement with Module Diversification*

The module diversification method (see Sect. 3) generates a set of configurations for each accelerator that are diversified in terms of CLB usage. This not only allows to tolerate any single-CLB fault in a region but can also improve the stress distribution with the extra CLB diversity. When faults are detected in the reconfigurable fabric, the placement freedom of accelerators is reduced. The *placement freedom* of an accelerator corresponds to the number of regions for which the accelerator has at least one diversified configuration that can be placed into that region (i.e., that tolerates the permanent faults in that region). Such a region is called a *compatible region*. If the available regions (i.e., those into which no accelerators are placed by the placement algorithm so far) have rather many permanent faults, it can happen that no configuration of the accelerator can be placed into any of them. If an accelerator cannot be placed, then its hardware functionality has to be emulated in software on the processor pipeline, which comes at a significant performance loss.

To avoid such situations, the runtime accelerator placement (see Sect. 4.3) is modified to place the accelerators one after the other in ascending order of their number of compatible regions. If it comes to the situation that some accelerator cannot be placed into the available regions, then the algorithm re-evaluates some of its previous placement decisions (note that the actual reconfigurations are just started after all placements are finally decided). It tries whether it can *swap* one of the already placed accelerators into one of the still available regions such that accelerator can be placed into the region that became free due to swapping. When calculating the placement profit (see Eq. (10)), the algorithm also iterates through all diversified configurations to find out which configuration of the accelerator produces the highest placement profits.

4.6 *Experimental Evaluation*

For prototyping purposes, we have integrated STRAP into the Xilinx tool-chain and the runtime system of the target reconfigurable architecture. In our evaluation platform, each region consists of 4×20 CLBs with eight 6-input LUTs per CLB. STRAP performs optimizations on CLB granularity. To evaluate the actual stress for each transistor, a transistor-level model of LUTs using NMOS pass transistors for multiplexers is used [22]. To evaluate the threshold voltage shift due to stress, state-of-the-art aging models are employed (detailed equations and used parameters are given in [22]). The resource usage of each accelerator within one region for the H.264 application ranges from 8.8% to 66.3%. Our architectural simulator is used to evaluate the STRAP method for systems that differ in the number of reconfigurable regions and runtime strategies, and to compare it with related work.

Evaluation Flow The placed-and-routed accelerators are fed to Xilinx XPower analyzer to obtain the signal activities and power consumption of logic elements and nets. The power consumption is then aggregated to CLB granularity by summing up the power consumed by LUTs and their fan-in nets in one CLB. The leakage power of a region is proportional to its size. Architectural simulation produces the accelerator execution trace, i.e., the complete execution and idle history of each accelerator in each region. Together with the power profile of each accelerator, we obtain the power trace of each CLB. The power trace and the fabric floorplan of the FPGA[2] are then fed into Hotspot[3] [14] to obtain the temperature trace of each CLB, which will be used to evaluate the threshold voltage shift. The accelerator execution trace and the LUT signal activities of each accelerator are combined to calculate the LUT signal activities for the regions. This is then used to evaluate the stress of individual transistors by using the before-mentioned LUT transistor model.

The number of regions is varied from 5 to 12 and separate evaluation is performed for dynamic and static stress mitigation, since STRAP optimizes either for dynamic or for static stress. The baseline system does not use any stress distribution method. For comparison, two state-of-the-art stress distribution methods [3, 21] were implemented. Zhang et al. [21] use three different configurations for each accelerator and switch between them to migrate stress, whereas Angermeier et al. [3] consider the peak stress of regions to place an accelerator. As proposed for STRAP, Angermeier et al. [3] and Zhang et al. [21] were extended to replace an accelerator if its reconfigurable region has not been reconfigured for 100 million cycles (see Sect. 4.3). This improvement reduces the peak stress of [3, 21] and thus makes the comparison with state-of-the-art more competitive. Regarding temperature variation, a conservative comparison is performed. To calculate the threshold voltage shift for [3, 21], the lowest temperature that was observed for any CLB at any time in the obtained temperature trace is used as the constant temperature for all CLBs, while the highest observed temperature is applied for STRAP. Thus, the threshold voltage shift reported for [3, 21] is a lower limit, whereas the one for STRAP is a conservative upper limit.

Timing Overhead STRAP's stress-diversifying logic placement at synthesis time may affect the accelerator frequency. The place-and-route tool is given a target frequency of 250 MHz as timing constraint to obtain the maximum operating frequency of each accelerator. On average, the maximum accelerator frequency decreases by 7%. Since accelerators with longer critical path (lower maximum frequency) are imposed with fewer constraints (see Sect. 4.4), their maximum frequencies are less affected. The maximum *system* frequency is however limited by the accelerator with the longest critical path (in our case the PointFilter accelerator, which runs at f_{min} = 89 MHz). Therefore, STRAP has no negative timing impact on the system.

[2]Based on a high-resolution die image acquired from https://chipworks.com (now https://techinsights.com).

[3]Smallest possible heat spreader and heat sink with 10 µm thickness, ambient temperature 50 °C.

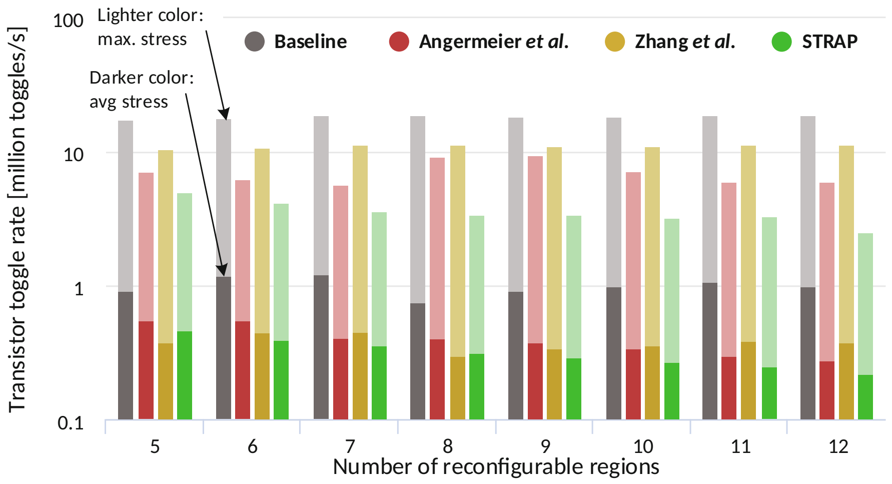


Fig. 10 The dynamic stress in systems with different number of reconfigurable regions when using our STRAP approach compared to the baseline, Angermeier et al. [3] and Zhang et al. [21] (based on [22])

Stress Reduction and MTTF Improvement Figure 10 shows the maximal (lighter color) and average (darker color; arithmetic mean) dynamic transistor stress, measured in million toggles/s, in the whole reconfigurable fabric for systems with different number of regions. It shows that all methods reduce the average stress compared to the baseline because they all distribute the stress to more transistors. While the reduction of the average stress is similar for all three methods, the reduction of the maximal stress (i.e., the critical part for system mean time to failure (MTTF)) differs significantly and requires both runtime and synthesis time optimization. The reason is that Angermeier et al. [3] perform only runtime inter-region stress distribution, while Zhang et al. [21] perform only synthesis time intra-region stress distribution for individual accelerators. In contrast, STRAP performs cross-layer stress-aware placement at runtime and synthesis time, which leads to the highest reduction of maximal stress in all evaluated cases. The reduction of the maximum stress by STRAP is up to 64% and 35% higher than the closest competitors w.r.t. dynamic and static stress, respectively. Table 3 summarizes the stress reduction.

Although during optimization only one type of stress is considered, actually both types of stress are reduced simultaneously. With STRAP targeting the static stress distribution, a reduction of 52% in dynamic and 38% in static stress is observed. When targeting dynamic stress, STRAP delivers 82% reduction in dynamic stress and 21% reduction in static stress. The reason behind the reduction of both stress types is that STRAP implicitly distributes the transistor usage as well, which reduces the individual static and dynamic transistor stress.

Table 3 Reduction of avg./max. stress and MTTF increase of STRAP and state-of-the-art [3, 21] compared to the baseline; averaged over all numbers of reconfigurable regions (based on [22])

	Reduction of avg. stress [%]		Reduction of max. stress [%]		MTTF improvement [%]	
Strategy	Dyn.	Stat.	Dyn.	Stat.	HCI	BTI
Angermeier et al. [3]	60.6	47.4	61.2	0.02	157.7	0.0
Zhang et al. [21]	62.6	49.6	39.9	4.5	66.4	2.3
STRAP	**67.9**	**59.6**	**80.5**	**33.1**	**413.0**	**13.4**

The MTTF improvement due to the stress reduction is calculated by assuming that a device fails when ΔV_{th} of any transistor exceeds 50% of its original value (V_{th0}). The MTTF improvement due to dynamic and static stress reduction is shown in the last two columns in Table 3. With the STRAP method, the MTTF improvement relative to the baseline is 413% and 13% in average for HCI and BTI aging, respectively. Relative to the closest competitors, STRAP achieves up to 177% and 14% MTTF improvement w.r.t. HCI and BTI aging, respectively.

5 Conclusion

The dependable operation of runtime-reconfigurable architectures is threatened by aging. This chapter presented novel methods to ensure reliable reconfiguration, mitigate aging, and tolerate emerging faults in the reconfigurable fabric. The *pre-configuration online tests* (PRET) and *post-configuration online tests* (PORT) check with minor application performance loss, if the reconfigurable fabric is faulty and if the reconfiguration process completed without errors during runtime. The *module diversification* design method generates the minimal number of diversified configurations required to tolerate at least any single CLB-fault in a reconfigurable region. The cross-layer stress-aware placement method STRAP mitigates aging by balancing stress both within a reconfigurable region as well as across all reconfigurable regions in the system. Relative to the closest competitors, STRAP achieves up to 177% and 14% MTTF improvement w.r.t. HCI and BTI aging. This shows that intelligently considering and managing aging threats during runtime can significantly improve the system dependability at limited overheads.

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. Abdelfattah, M.S., Bauer, L., Braun, C., Imhof, M.E., Kochte, M.A., Zhang, H., Henkel, J., Wunderlich, H.-J.: Transparent structural online test for reconfigurable systems. In: IEEE International On-Line Testing Symposium (IOLTS), pp. 37–42 (2012)
2. Amrouch, H., van Santen, V.M., Ebi, T., Wenzel, V., Henkel, J.: Towards interdependencies of aging mechanisms. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 478–485 (2014)
3. Angermeier, J., Ziener, D., Glaß, M., Teich, J.: Stress-aware module placement on reconfigurable devices. In: International Conference on Field Programmable Logic and Applications (FPL), pp. 277–281 (2011)
4. Bauer, L., Shafique, M., Henkel, J.: Concepts, architectures, and run-time systems for efficient and adaptive reconfigurable processors. In: NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pp. 80–87 (2011)
5. Bauer, L., Braun, C., Imhof, M.E., Kochte, M.A., Zhang, H., Wunderlich, H.-J., Henkel, J.: OTERA: Online test strategies for reliable reconfigurable architectures. In: NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pp. 38–45 (2012)
6. Bauer, L., Braun, C., Imhof, M.E., Kochte, M.A., Schneider, E., Zhang, H., Henkel, J., Wunderlich, H.-J.: Test strategies for reliable runtime reconfigurable architectures. IEEE Trans. Comput. (TC) **62**(8), 1494–1507 (2013)
7. Bauer, L., Zhang, H., Kochte, M.A., Schneider, E., Wunderlich, H.-J., Henkel, J.: Advances in hardware reliability of reconfigurable many-core embedded systems. In: Many-Core Computing: Hardware and Software, pp. 395–416. Institution of Engineering and Technology (IET) (2019)
8. Cao, Y., Velamala, J., Sutaria, K., Chen, M.S.-W., Ahlbin, J., Esqueda, I.S., Bajura, M., Fritze, M.: Cross-layer modeling and simulation of circuit reliability. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. (TCAD) **33**(1), 8–23 (2014)
9. Gaisler, A.: Homepage of the Leon Processor. Online available: https://www.gaisler.com/index.php/products/processors/leon3. Accessed 13 Mar 2019
10. Guo, X., Burleson, W., Stan, M.: Modeling and experimental demonstration of accelerated self-healing techniques. In: IEEE/ACM Design Automation Conference (DAC), pp. 1–6 (2014)
11. Gupta, P., Agarwal, Y., Dolecek, L., Dutt, N., Gupta, R.K., Kumar, R., Mitra, S., Nicolau, A., Rosing, T.S., Srivastava, M.B., Swanson, S., Sylvester, D.: Underdesigned and opportunistic computing in presence of hardware variability. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. (TCAD) **32**(1), 8–23 (2013)
12. Henkel, J., Bauer, L., Becker, J., Bringmann, O., Brinkschulte, U., Chakraborty, S., Engel, M., Ernst, R., Härtig, H., Hedrich, L., Herkersdorf, A., Kapitza, R., Lohmann, D., Marwedel, P., Platzner, M., Rosenstiel, W., Schlichtmann, U., Spinczyk, O., Tahoori, M., Teich, J., Wehn, N., Wunderlich, H.-J.: Design and architectures for dependable embedded systems. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 69–78 (2011)
13. Henkel, J., Bauer, L., Dutt, N., Gupta, P., Nassif, S., Shafique, M., Tahoori, M., Wehn, N.: Reliable on-chip systems in the nano-era: lessons learnt and future trends. In: IEEE/ACM Design Automation Conference (DAC), pp. 1–10 (2013)
14. Huang, W., Ghosh, S., Velusamy, S., Sankaranarayanan, K., Skadron, K., Stan, M.R.: HotSpot: a compact thermal modeling methodology for early-stage VLSI design. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **14**(5), 501–513 (2006)
15. Lala, P.K.: Self-checking and Fault-Tolerant Digital Design. Morgan Kaufmann, San Francisco (2001)
16. Mahapatra, S.: Fundamentals of Bias Temperature Instability in MOS Transistors: Characterization Methods, Process and Materials Impact, DC and AC Modeling. Springer Series in Advanced Microelectronics, vol. 52. Springer, New Delhi (2015)

17. Srinivasan, S., Krishnan, R., Mangalagiri, P., Xie, Y., Narayanan, V., Irwin, M.J., Sarpatwari, K.: Toward increasing FPGA lifetime. IEEE Trans. Depend. Sec. Comput. (TDSC) **5**(2), 115–127 (2008)
18. Stott, E.A., Wong, J.S., Sedcole, P., Cheung, P.Y.: Degradation in FPGAs: measurement and modelling. In: ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), pp. 229–238 (2010)
19. Xilinx: Partial Reconfiguration User Guide, UG702 (v14.1) (2012)
20. Yang, S.: Logic synthesis and optimization benchmarks user guide: version 3.0. MCNC Technical Report, Microelectronics Center of North Carolina (MCNC). https://ddd.fit.cvut.cz/prj/Benchmarks/
21. Zhang, H., Bauer, L., Kochte, M.A., Schneider, E., Braun, C., Imhof, M.E., Wunderlich, H.-J., Henkel, J.: Module diversification: fault tolerance and aging mitigation for runtime reconfigurable architectures. In: IEEE International Test Conference (ITC), pp. 1–10 (2013)
22. Zhang, H., Kochte, M.A., Schneider, E., Bauer, L., Wunderlich, H.-J., Henkel, J.: STRAP: stress-aware placement for aging mitigation in runtime reconfigurable architectures. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 38–45 (2015)
23. Zhang, H., Bauer, L., Kochte, M.A., Schneider, E., Wunderlich, H.J., Henkel, J.: Aging resilience and fault tolerance in runtime reconfigurable architectures. IEEE Trans. Comput. (TC) **66**(6), 957–970 (2017)