Invited Paper in Special Session “Embedded Resiliency: Approaches for the Next Decade”

Jörg Henkel, Lars Bauer, Hongyan Zhang, Semeen Rehman, Muhammad Shafique
Chair for Embedded Systems, Karlsruhe Institute of Technology, Karlsruhe, Germany
{henkel, lars.bauer, hongyan.zhang, semeen.rehman, muhammad.shafique}@kit.edu

ABSTRACT
We show in this paper that multi-layer dependability is an indispensable way to cope with the increasing amount of technology-induced dependability problems that threaten to proceed further scaling. We introduce the definition of multi-layer dependability and present our design flow within this paradigm that seamlessly integrates techniques starting at circuit layer all the way up to application layer and thereby accounting for ASIC-based architectures as well as for reconfigurable-based architectures. At the end, we give evidence that the paradigm of multi-layer dependability bears a large potential for significantly increasing dependability at reasonable effort.

1. INTRODUCTION
Dependability has become a major design constraint about a decade ago when it became clear that scaling to nano-scale CMOS structures would cause a variety of reliability threats including aging, increased susceptibility to soft errors, device variability, thermal density etc. First approaches to cope with these problems came from physical, device and circuit layer. Later on, so-called ‘cross-layer’ approaches emerged where means to increase dependability where applied by looking at the interface between two adjacent abstraction layers. A comprehensive survey of different reliability threats and single-/cross-layer dependability approaches can be found in [1]. All these approaches were not good enough to pursue a true paradigm shift where dependability appears as the most important design paradigm: in fact, it must involve a major part of the design abstraction layer stack and dependability increasing means must be proactively adapted to each other. This is what we call ‘multi-layer’ dependability. It is defined as follows:

Definition: Multi-layer Dependability. We use the term multi-layer dependability if two or more hardware and/or software abstraction layers pass formation and/or are adapted to each other (at design time or at runtime) with the purpose to reduce error propagation through these layers. It is the goal to minimize the effort in terms of hardware/software costs through this information exchange and adaption compared to the case where the layers are optimized without being aware of applied error propagation means at the adjacent abstraction layers.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

DAC’14, June 01 - 05 2014, San Francisco, CA, USA
Copyright 2014 ACM 978-1-4503-2730-5/14/06...$15.00.
http://dx.doi.org/10.1145/2593069.2596683.

Following this approach we present in this paper a comprehensive design flow for multi-layer dependability. It is part of the DFG funded priority program SPP 1500 “Dependable Embedded Systems” [2, 3]. In order to bridge the gap between hardware and software for enabling efficient reliability-driven optimizations, we have developed reliability models that account for detailed hardware-layer reliability estimates to accurately quantify the reliability at the software layer.

This paper is part of the DAC special session on “Embedded Resiliency: Approaches for the Next Decade”. Other papers in this session are: “Monitoring Reliability in Embedded Processors – A Multi-layer View” [4], “Multi-Layer Memory Resiliency” [5] and “Workload- and Instruction-Aware Timing Analysis – The Missing Link between Technology and System-level Resilience” [6].

2. MULTI-LAYER DEPENDABILITY
2.1 Overview
Applications typically differ in their performance and reliability requirements. In heterogeneous manycore architectures, processing cores with different performance/reliability/efficiency trade-offs can be combined to address these issues. Fig. 1 shows a generic instance of such a heterogeneous architecture in the middle of the ‘Architecture Layer’. In addition to regular general purpose cores (fully hardened, partially hardened, or unhardened) [7], it also contains reconfigurable cores that couple fine-grained runtime reconfigurable fabrics (e.g. embedded FPGAs) to a (partially) hardened core pipeline to provide application-specific accelerators on demand [8]. Both, the challenges and potential solutions for regular cores (left side in Fig. 1) and reconfigurable cores (right-hand side in Fig. 1) differ significantly as we will see in the following.

In the vertical dimension, Fig. 1 illustrates all layers that are affected by different reliability threats (as outlined in Section 1). The actual reliability threats manifest in the ‘Device and Circuit Layer’ and propagate to higher layers. Modeling reliability threats is correspondingly done in a bottom-up fashion, whereas the optimization strategies are performed top-down. The lower layers offer different types of reliability actions and the higher layers decide which reliability action to use at which time. In the following, we describe dependability modeling and optimization techniques for all layers.

2.2 Device and Circuit Layer
For regular cores and the core pipeline of reconfigurable cores we developed a framework to estimate the aging of an in-order RISC processor. It estimates the delay degradation of $x\%$ critical paths under varying activity, workload and temperature conditions. An important input to our framework is the low-level aging estimate of different logic elements (i.e. SRAM cells, Logic Gates) that are obtained through an accurate device-layer aging estimator.
based upon the ngspice-23 and actual device parameters (like transistor dimensions, gate structure from TSMC data sheets, activity, temperature, load capacitance, etc.). Fig. 2 shows an example aging result for a NOR gate for various signal probabilities. For processor aging estimation, first, a LEON3 processor (VHDL design) is synthesized using TSMC 45nm low-power library and the Synopsys Design Compiler with different system clock frequencies (250 MHz – 1 GHz), 0.81 Volt, and 125°C junction temperature. The netlist is parsed and stored in a graph-based data structure. Afterwards, critical path analysis is performed. The switching activities/signal probabilities are obtained

Figure 1: Multi-Layer Dependability Modeling and Optimization
through ModelSim simulation. The initial delay estimates are obtained from the Standard Delay Files (SDFs). Our tools generate aging estimates for 2% critical paths over 10 years (see Fig. 1; left-hand side of the ‘Device and Circuit Layer’) by applying the delay degradation to individual logic elements in the SDF and re-performing the critical path analysis.

Not all faults in a logic element of a combinational circuit affect its output; a fault may be masked due to logical masking effects of the subsequent gates. In order to obtain fault probabilities of different processor components, we estimate gate-level error masking probabilities using conditional probability analysis of error propagation through multiple gates for a given circuit netlist.

For reconfigurable processors we want to detect, tolerate and avoid permanent structural faults in the reconfigurable fabric’s logic and routing. This is especially important, as we do not know which parts of the reconfigurable fabric will be used at runtime and thus all parts need to be tested to ensure (i) a reliable reconfiguration process and (ii) reliable operation of accelerators that use the reconfigurable fabric.

We use the stuck-at fault model with single-fault assumption in the region under test (if a particular fault is already known in a region, that part can be excluded from the test to maintain the single-fault assumption in the remaining region). We designed different test configurations that can be reconfigured into the reconfigurable fabric to exercise its resources and provide exhaustive coverage of all logic resources [10, 11]. These tests are provided as reliability actions to higher layers (see Section 2.6) that decide ‘how often’ and ‘when’ they shall be executed.

To accurately estimate BTI and HCI aging for the reconfigurable fabric, we developed a transistor model for a configurable logic block (CLB), apply input stimuli to it and extract the toggle rate (for HCI) and signal probability (for BTI) for each transistor [12].

Figure 2: Aging of a NOR gate for different signal probabilities [9].

Figure 3: Different ISA-compatible reliability-heterogeneous cores [7].

2.3 Microarchitecture Layer

Different instructions use diverse processor components with distinct area (spatial vulnerability) and for different number of cycles (temporal vulnerability) in various pipeline stages. Therefore, we formulate the Instruction Vulnerability Index (IVI) model that quantifies the error probability at instruction granularity by jointing accounting for the spatial vulnerability, i.e. area-dependent error probability; and temporal vulnerability, i.e. time-dependent error probability [13]. To quantify these vulnerabilities for a given program, we perform (1) vulnerable bit analysis for different pipeline stages/processor components considering the fault probabilities and area of different processor components (like register file, ALU, and cache controller); and (2) vulnerable period analysis because different instructions spend different time in processing components. For instance, the ‘store’ instruction uses ALU and Data Cache Controller in ‘execute’ and ‘memory’ pipeline stages (see Fig. 1), respectively, that denotes its higher spatial and temporal vulnerability compared to that of an ‘add’ instruction. Similarly, when comparing ‘store’ and ‘multiply’ instructions, the later is more vulnerable in the ‘execute’ stage.

Reconfigurable processors are also susceptible to SEUs. Unlike regular processors where an SEU is a transient fault, an SEU in the reconfigurable fabric of a reconfigurable processor may affects a configuration bit and thus potentially modifies the accelerator that is configured by this bit and is permanent until it is corrected (i.e. reconfigured). Therefore, frequent testing and correcting of configuration bits (so-called scrubbing) is required and offered as a reliability action to higher layers. We have developed a light-weight model that determines the fault likelihood of accelerators depending on the measured self-error recovery rate [14]. The fault likelihood can then be used by the runtime system to enable pre-synthesized redundancy modes (e.g. triplication; see Sections 2.5 and 2.6).

2.4 Architecture Layer

Since different applications may exhibit varying vulnerability and error masking properties, not all applications would require the same level of protection/reliability from the underlying hardware, which may not even be possible under stringent power constraints. Therefore, we envision a “Reliability-Heterogeneous Manycore Processor” with a set of reliability-customized cores providing distinct reliability, performance, power, and area properties [7]. We perform a design-time reliability-aware core customization where different cores are enhanced with specialized reliability features under a given area constraint. Different cores have heterogeneous error recovery functionality, i.e. some cores are fully hardened (FHC), some are partially hardened cores (PHC) (e.g. offering information redundancy with ECC based memories), some are unprotected cores (UPC), and some cores are enhanced with reconfigurable features (ReC) as shown in Fig. 1. Note that in such an architecture, all the cores exhibits same Instruction Set Architecture (ISA). Our customization algorithm generates appropriate architectural templates aims at jointly maximizing the functional and timing reliability. Fig. 3 illustrates that different applications exhibit varying vulnerabilities on different core types.

Given such reliability-heterogeneous manycore processor and a power constraint, our runtime system (see Section 2.6) performs reliability-aware core allocation and reliable code version selection (generated through reliability-driven compilation; see Section Section 2.5) under power constraints, while accounting for multiple fault types.

2.5 Compiler and Synthesis Layer

To tolerate latent permanent fault in the reconfigurable fabric, we propose the concept of module diversification [12]. If a certain part of a runtime reconfigurable region (a so-called container) has a permanent fault due to aging, we can still load accelerators into it if they do not use that faulty part. Fig. 1 shows some examples for diversified configurations in the right-hand side of ‘Compiler and Synthesis Layer’. To generate them, the CLB usage of an initial configuration is extracted and stored in a data structure. The module diversification algorithm [12] uses this information to generate a minimal set of alternative configurations that are diversified in terms of CLB usage.
such that for any CLB in the container, there always exists a diversified configuration that does not use that CLB. The CLB usage information for all diversified configurations are translated into placement constraints for the vendor place-and-route tool to finally obtain the corresponding diversified modules (i.e. partial bitstreams). The place-and-route configurations are also taken as input for the stress estimation process, where the signal activities of CLB transistors are estimated, to obtain the stress pattern of each configuration.

In performance-oriented reconfigurable architectures, applications contain accelerated functions that use the accelerators in the containers. Each of them is typically composed of several accelerators and thus they are available in multiple hardware implementation variants to trade-off performance and resource usage. We implemented additional variants that use partial spatial redundancy (i.e. DWC (duplication) or TMR (triplication) for some selected accelerators based on their vulnerability) or complete spatial redundancy (i.e. all accelerators are protected), as shown in Fig. 1 in the right-hand side of ‘Compiler and Synthesis Layer’.

To enable reliability-driven compilation, static estimation of program reliability (i.e. IVI) is required before applying the reliability-aware transformations and instruction scheduling. For static estimation of vulnerable time periods, the execution probabilities of basic blocks are considered [15]. Several instructions’ output errors may be masked due to data flow properties (depending upon the type and operand variables of subsequent instructions) and changing control flow properties. To quantify these effects, we developed an Instruction Masking Index (IMI) [16] model that estimates the probability of an error at an instruction being masked until the visible program output. In case an error is not masked, it may be propagated to multiple program outputs. For a given instruction, our Instruction-Level Error Propagation Index (EPI) [16] quantifies these effects using the non-masking error probabilities (i.e. the probability that an error is visible at the program output) of its successor instructions. Both IMI and EPI require program’s data and control flow graph analysis.

Fig. 4 shows varying IMI and EPI values that need to be jointly considered along with the IVI for quantifying the reliability-importance of different instructions. Considering IVI, IMI, and EPI, we model the reliability of a software program at different granularity (i.e. instruction, basic block, function, and task) such that different reliability optimizations can be applied at the compiler and runtime system layers. Moreover, in order to account for the timing effects, we also define the Reliability-Timing (RT) penalty [17] as the linear combination of functional reliability (i.e., the reliability penalties in terms of the vulnerability indexes) and timing reliability (i.e., the deadline misses).

First, the tolerable total performance overhead (provided by a user) is distributed among different functions/tasks in an application based on their resilience/vulnerability properties [18]. Afterwards, several reliability-aware front-/middle-/backend compiler transformations are employed to generate multiple reliable versions for each function (with different binaries) under constraints of (i) tolerable performance overhead; and (ii) number of versions. These function versions are identical in terms of their functionality and the output but differ w.r.t. performance, power, and reliability indexes. These transformations aim at reducing applications’ susceptibility towards failures (like crashes, hangs, etc.) through reducing the execution probability and vulnerabilities of so-called critical instructions (like branches, calls, load/stores, and address generation instructions) [13]. Afterwards, we employ a soft-error driven instruction scheduler that determines instruction execution sequence to enhance a software program’s reliability under a user-provided tolerable performance overhead [19]. It incorporates a joint cost function of statically-estimated instruction vulnerabilities and number of dependent instructions. The scheduler employs a lookahead-based heuristic for evaluating the reliability cost of various scheduling candidate instructions while taking into account the reliability weights of the successor instructions.

After generating multiple reliable versions (see examples in Fig. 5), the design space is pruned by taking the Pareto optimal front-tier options that are forwarded to the runtime system that exploits these multiple versions for appropriate reliability management depending upon the execution contexts [17].

### Figure 4: IMI and EPI for different Instructions for the Susan Application [16].

![Image](image4.jpg)

### Figure 5: Average execution time and vulnerabilities of different compiled versions of different functions [17].

#### 2.6 Runtime System Layer

For given multiple reliable function versions, our runtime system composes and executes the application in a reliable way. To facilitate the runtime system, first, an offline algorithm is employed to construct a schedule table while minimizing the Reliability-Timing (RT) penalty [17]. Probability distribution of the task execution time is taken into account while constructing the schedule tables to exploit the dynamic execution behavior. Given offline constructed schedule tables for multiple task versions and the task scheduling sequence, our runtime system dynamically selects an appropriate task version depending upon the current execution behavior and reliability layer (i.e. RT-penalty) along with the remaining time to the final deadline [17].

For dynamic reliability management, it is important to account for multiple reliability threats. Our dynamic reliability management system jointly account for soft errors and cores’ performance variations due to design-time process variation and/or runtime aging-induced performance degradation [9, 20]. It performs reliability-driven core allocation and task mapping at runtime while leveraging varying vulnerability and error masking properties of different applications. Afterwards, an appropriate soft-error tolerant compiled version is selected based on the properties of the allocated core. Compared to four different state-of-the-art techniques, we achieve on average 44% and up to 63% improved task reliability for different chip configurations, different variability maps (generated using the model of [21]), and different aging years (see Fig. 6) [9].

The runtime system for the reconfigurable processors periodically schedules test configurations into reconfigurable containers to ensure the absence of structural faults in the
underlying fabric. Fig. 7 shows the simulation results for the performance loss (solid lines, left Y-axis) and average test latency (dashed lines, right Y-axis) under different test frequencies, depending on the number of reconfigurable containers.

The test frequencies varies from one test configuration before every accelerator configuration (1 TC/AC) to one test configuration before every 4th accelerator configuration (1 TC/4 ACs). Lower test frequencies induce less performance overhead, since less testing implies that the reconfiguration port or the reconfigurable containers is less likely to be blocked. For all benchmark settings, the performance impact (solid line) due to testing the structure of containers is below 1%. The dashed lines indicate how long does it take in average for any container to be tested completely, i.e. until all test configurations (9 in our case [10, 11]) are applied. The observed test latencies show that any emerging permanent faults in the system can be detected latest after 3.8s to 8.1s [22].

Module diversification [12] enables to tolerate permanent faults in reconfigurable containers by using different configurations that are diversified in their CLB usage (see Section 2.5). To show its effectiveness for fault tolerance, we use the ‘reliability improvement factor’ (RIF) metric. It is the ratio of the failure probability of a module without module diversification and the failure probability of a module with diversified configurations. Fig. 8 shows the RIF of different benchmark modules depending on the individual CLB reliability ranging from 0.999 to 0.9999999. The higher the CLB reliability, the less probable the CLB becomes faulty. It is clear that RIF increases with increasing CLB reliability. The reason behind it is that when the individual CLB reliability is high, it is less probable that multi-CLB faults occur and it is more probable that a single-CLB fault occurs, which can be always tolerated by module diversification. In extreme case, when only a single-CLB fault can occur, the failure probability of the module is zero, which would lead to an infinite reliability improvement.

In addition, the diversified configurations exhibit diversified stress distribution among CLBs. This stress diversity is exploited to balance the stress among CLBs by optimally scheduling the operation time of each configuration. Thus, stress is not concentrated on individual CLBs, and CLB aging is mitigated to increase lifetime. Table 1 summarizes the HCI and NBTI stress reduction for optimally scheduled diversified configurations [12]. For each module, the reduction of the maximum stress and the average over all used CLBs are listed.

Table 1: Stress reduction [%] by proposed balanced schedule with min. number of configurations [12]

<table>
<thead>
<tr>
<th>Module</th>
<th>HCI stress reduction [%]</th>
<th>NBTI stress reduction [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W_{min}</td>
<td>W_{max}</td>
</tr>
<tr>
<td>pdc</td>
<td>33.8</td>
<td>43.6</td>
</tr>
<tr>
<td>misex3</td>
<td>37.7</td>
<td>41.2</td>
</tr>
<tr>
<td>alu4</td>
<td>32.6</td>
<td>51.7</td>
</tr>
<tr>
<td>apex4</td>
<td>35.2</td>
<td>54.6</td>
</tr>
<tr>
<td>apex2</td>
<td>24.8</td>
<td>50.3</td>
</tr>
<tr>
<td>des_perf</td>
<td>68.9</td>
<td>50.2</td>
</tr>
<tr>
<td>aes_core</td>
<td>26.7</td>
<td>49.7</td>
</tr>
</tbody>
</table>

Since the stress reduction depends on the number of unused CLB in the container, we investigate the cases with the minimal container width W_{min} and maximal container width W_{max}. The results show clearly that the reduction of both maximum and average stress are achieved for all modules. In larger containers, stress can be distributed to more CLBs and thus stress on individual CLBs can be further reduced.

GUARD [14] is the first runtime system for reconfigurable architectures that guarantees a target reliability of an execution while optimizing the performance. At runtime, depending on the observed soft-error rate, the runtime system adaptively determines the scrubbing rate and selects execution variants (see Section 2.5) for the accelerated functions that maximize the performance for a given reliability constraint.

To evaluate the behavior of the system in response to different environmental conditions, we use a sinusoidal soft error rate varying between 0 (no errors) and 10 errors Mb^{-1}month^{-1} in the configuration bits as input stimuli for the system. We impose a reliability constraints that the failure probability of each execution of accelerated functions must be less than 10^{-10}. We compare it to a threshold-based approach that duplicates (DWC) or triplicates (TMR) the accelerators when the error rate exceeds 1.8 Mb^{-1}month^{-1} to ensure that AF failure probability does not exceed 10^{-10}. In the threshold based approach, scrubbing is performed at maximum rate.

As shown in Fig. 9, GUARD reacts autonomously to the changing error rate. For a high error rate, hardware resources that are otherwise available for acceleration are devoted to implement fault tolerance methods and thus performance decreases. When the threshold-based methods
switch to DWC or TMR modes, much more resources are consumed, which causes a significant performance drop. For a low error rate, their performance is still below the GUARD approach since the high scrubbing frequency blocks the configuration port.

2.7 Application Layer

To employ application-layer techniques, it is important to quantify the reliability at a coarse granularity such that the underlying details are hidden from the application/system developer. To facilitate this, we modeled application resilience at different granularity (i.e. basic block, function, and task) that provide a probabilistic measure of functional correctness (output quality) in presence of hardware-layer faults.

We exploit the concepts from information theory to model resilience as the normalized mutual information between the required correct result (from a golden execution run) and the result at the end of an application execution (from a potentially faulty execution under a given fault rate), i.e. amount of useful application output [18]. To estimate the information loss, we modeled the application program as a Markov Chain and estimated the state transition probabilities using the maximum likelihood. To hide the underlying processor details, the error statistics are obtained through Monte-Carlo fault injection campaigns. The Task Resilience [23] is estimated as the weighted sum of the basic block resilience multiplied with the control flow execution probabilities of its predecessor basic blocks. Fig. 10 shows the resilience levels of different tasks.

We also leverage the unused space in available data structures to provide partial redundancy of critical data elements. As an initial case study, we exploit the statistical distribution of different data elements in an H.264 video encoder to predict the available free space and to identify critical data elements for redundancy [24, 25].

3. CONCLUSIONS

We have shown in this paper that adapting dependability enhancing means to each other significantly increases the dependability of the entire system at manageable effort. Our comprehensive design flow for multi-layer dependability spans abstraction layers from circuit layer to micro-architecture, compiler all the way up to application layer. It treats ‘dependability’ as the major design constraint – a prerequisite for complex and dependable future on-chip systems.

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