Appendix A
Simulation Infrastructure

This chapter presents the integrated tool flow, simulation environment, and fault injection setup that are developed in this work. The reliability analysis and evaluations of the individual techniques compared to state-of-the-art approaches are already discussed in the previous chapters, i.e., Chaps. 4, 5, and 6. The complete overview of the developed tool chain and infrastructure is shown in Fig. A.1.

Based on a processor description (e.g., of LEON3) in the form of VHDL files and a technology library containing different gates, the processor is synthesized using the *Synopsys Design Compiler* in order to obtain the netlist and a set of critical paths. This data is then used to estimate the area and fault probabilities of different processor components, and the aging of the critical paths. Additionally, logic simulations using *ModelSim* are performed executing different applications on the synthesized processor for extracting their respective activity and signal probabilities. The information about the area and fault probabilities of different processor components, which is obtained after the processor synthesis, is used to estimate the program reliability. In the *reliability-aware manycore simulator*, the run-time aging estimation results are finally used jointly with the design time process variations (that are input into the infrastructure as variation maps) to account for varying performance characteristics of different cores. The simulation environment is based on an Instruction Set Architecture (ISA)-simulator for LEON3 cores generated using the ArchC tool chain [119]. For reliability analysis and estimation, different applications are simulated and the required data for devising the models and for parameter estimation is obtained. Furthermore, the simulator is equipped with a configurable fault generator, fault injector, and error logging modules for characterizing the impacts of the reliability threats on different applications. Different benchmark applications from the MiBench benchmark suite [111] are used for evaluation, which form the input to the reliability-driven compilation setup that generates versions with different reliability and performance characteristics by employing different techniques. In the following, the individual parts and tools of the infrastructure are discussed.

S. Rehman et al., *Reliable Software for Unreliable Hardware*,
DOI 10.1007/978-3-319-25772-3

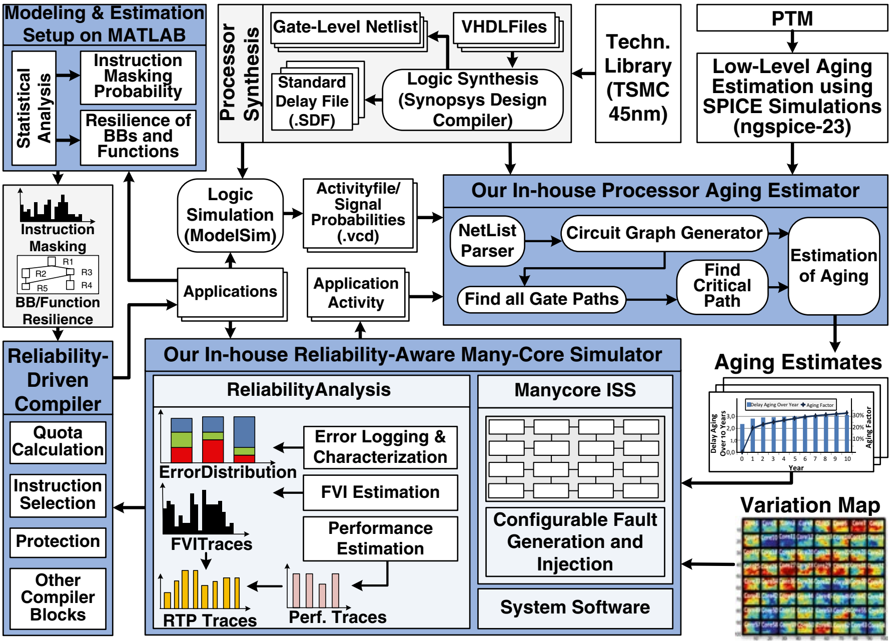

Modeling & Estimation Setup on MATLAB
Statistical Analysis
Instruction Masking Probability
Resilience of BBs and Functions
Processor Synthesis
Gate-Level Netlist
VHDLFiles
Standard Delay File (.SDF)
Logic Synthesis (Synopsys Design Compiler)
Techn. Library (TSMC 45nm)
PTM
Low-Level Aging Estimation using SPICE Simulations (ngspice-23)
Instruction Masking
R1
R2
R3
R5
R4
BB/Function Resilience
Logic Simulation (ModelSim)
Activityfile/ Signal Probabilities (.vcd)
Our In-house Processor Aging Estimator
NetList Parser
Circuit Graph Generator
Estimation of Aging
Find all Gate Paths
Find Critical Path
Applications
Application Activity
Reliability-Driven Compiler
Quota Calculation
Instruction Selection
Protection
Other Compiler Blocks
Our In-house Reliability-Aware Many-Core Simulator
ReliabilityAnalysis
ErrorDistribution
Error Logging & Characterization
FVI Estimation
FVITraces
Performance Estimation
RTP Traces
Perf. Traces
Manycore ISS
Configurable Fault Generation and Injection
System Software
Aging Estimates
Delay Aging Over Year
Aging Factor
Delay Aging Over 10 Years
Year
Variation Map

Fig. A.1 Tool flow for processor synthesis, processor aging estimation, and reliability simulation and evaluation for manycore processors

Several tools developed in the scope of this manuscript are made available online for download at http://ces.itec.kit.edu/846.php.

A.1: Reliability-Aware Manycore Instruction Set Simulator and Fault Injection

The reliability-aware Instruction Set Simulator (ISS) is based on the ArchC architecture description language and related tools [119] (described in Sect. A.2). It simulates a *SPARC v8* pipelined architecture with 16 kB of ECC-protected instruction and data caches; see area details in Table 7.1 and processor layout in Fig. 3.3. The simulation environment is extended with a configurable fault generation and injection module that injects faults in different processor components during the application execution; see different input parameters in Table A.1.

The fault rate (in #faults/MCycles) is obtained using the neutron flux calculator [126] and city coordinates which determine the geographical location and altitude where the device will be used. Considering various locations, we obtained three different fault rates in our experiments (1, 5, 10 faults/MCycles) to cover a wide range of cases (terrestrial to aerial), which conforms to the test conditions opted by

Table A.1 Different parameter for fault scenario generation

Parameter	Description	Properties/values
Distribution	Distribution models for fault generation	Random, uniform
Bit flips	Min/Max number of bits flipped	1/1, 1/2, 1/3,…
Fault probability	Probability that strike becomes a fault	Output of Sect. 4.1.3
Fault location	List of target processor components	Register file, PC, IW, IM, DM, etc.
Processor layout/area	Size of the complete target device	in mm^2 (Output of Sect. 7.1.1)
Component area	Area of different processor components given as percentage of processor area	0–100 %
Place and altitude	City and altitude at which the device is used to determine the flux rate	Karlsruhe, Germany; 1–20 km
Frequency	Operating frequency of the processor	100–500 MHz

prominent related work [77, 80] and as such eases comparison. The errors are observed at the application software layer and are classified in different error categories. Numerous fault injection campaigns were performed with different configurations like flux rate, operating frequency, fault models (single or multiple bit flips), and distribution models; see fault injection parameters in Table A.1. The complete methodology of the reliability-aware simulation and analysis is done in two major steps: (1) fault generation and injection during simulation and (2) error analysis and estimation.

A.2: ArchC Architecture Description Language

ArchC [119] is an Architecture Description Language (ADL) which is based on SystemC and is used to define processor architectures following the C++/SystemC syntax style. ArchC facilitates the designers to model new architectures and also to experiment with existing ones. Furthermore, the architecture can be described on various abstraction levels (e.g., functional or cycle-accurate description of an architecture) and afterwards the generation of software tools (e.g., simulators, assemblers, and linkers) can be performed automatically. For various architectures, the ArchC descriptions are available in [127], for instance, MIPS, Intel 8051, and *SPARC v8* (which is adopted for evaluation in this research and described in the following), that can be used to generate functional simulators. Furthermore, ArchC offers a co-verification mechanism that enables checking the consistency of a refined model against a reference model.

For the simulator generation two basic input descriptions are required:

- *Architecture Resources* (*AC_ARCH*): the information regarding the resources, e.g., pipeline structure and memory hierarchy needs to be defined (see *sparcv8.ac* for the *SPARC v8* architecture).

- *Instruction Set Architecture* (*AC_ISA*): details about every instruction such as the format, opcode, and behavior need to be described (see *sparcv8_isa.ac* for the instruction declarations).

These descriptions serve as an input to the *ArchC Simulator Generator* (acsim), which outputs the C++ classes and SystemC modules required to build the simulator. Additionally, the ArchC Simulator Generator uses a decoder generator and a preprocessor for lexical analysis and parsing of the language, which extracts information from the description files. The following files are created; only the important ones for a *SPARC v8* architecture are listed below.

- *main.cpp*: this file provides the facility to instantiate the model and several features can be set here. It can be extended for the usage of additional SystemC modules.
- *sparcv8.cpp*: the processor module is implemented in this file. Amongst others, it contains a loop in which the decoding and the appropriate instruction behavior are called.
- *sparcv8_isa.cpp*: the behavior description for all instructions of the *SPARC v8* architecture is presented here. This file is created as a template and the behavior method for every instruction is placed inside by the designer. The description of an instruction behavior comprises of a general instruction behavior, which is common for all instructions, a format behavior, which is common for all instructions that have the same instruction format, and a specific instruction behavior for each individual instruction.

Figure A.2 shows the complete flow for the ArchC simulator generation. A GCC compiler is used to compile (i.e., by running "*make -f Makefile.archc*") the created model or to extend the existing ones and produces an executable specification of the target architecture. The generated simulator executes instruction decoding (that can be speeded up using a cache for decoded instructions), scheduling and behavior dynamically. Moreover, it supports operating system (OS) call emulation so that it is possible to simulate applications which contain I/O operations.

This step outputs a file *sparcv8.x* which is used for an application simulation that has been compiled using the automatically generated tools. Further detailed descriptions of ArchC and the related tools can be found in [119] and [128].

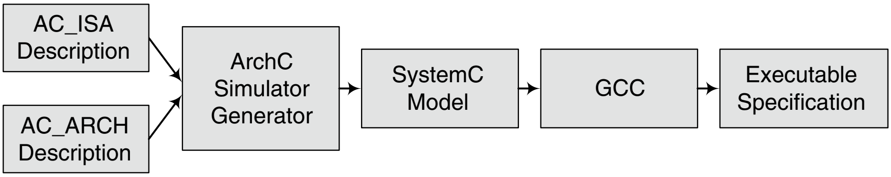


Fig. A.2 ArchC simulator generation [128]

A.3: Reliability-Aware Simulation and Analysis Methodology

Figure A.3 shows an overview of the developed simulation and analysis methodology for evaluating different software program reliability techniques. It works in two main phases that operate in an automated flow.

Fault Injection and Simulation Phase: The fault injection technique integrated in the instruction set simulator (ISS) is equipped with a configurable fault generation engine (described below in detail). The fault generator generates different fault scenarios considering different fault models (e.g., number of bit flips and distribution), fault rates, and faults in different architectural components (e.g., register file, Program Counter (PC), Instruction Word (IW)). Processor-specific details (chip footprint, component area, number of registers, etc.) and fault model configurations are passed as input (Table A.1).

The number of injected faults per component is determined by the component area (obtained after RTL synthesis) to incorporate spatial vulnerability. For example, fewer faults are injected in the PC compared to the ALU/Multiplier. The fault modeling procedure at the ISS level is illustrated in Fig. A.4. For example, a fault in the instruction decoder or in the IW is modeled as corrupting one/multiple fields of the IW in the ISS that results in a wrong opcode or wrong operand. The faults are injected during the application execution. If a fault is injected into the multiplier while an *add* instruction is being executed, it will have no effect on the application output. Note that the modeling procedure and fault injection are generic and independent of a particular architecture implementation. In case of a protected component, the correct state is resumed immediately after the fault injection. In the following, the fault generation and fault injection steps are explained in more detail.

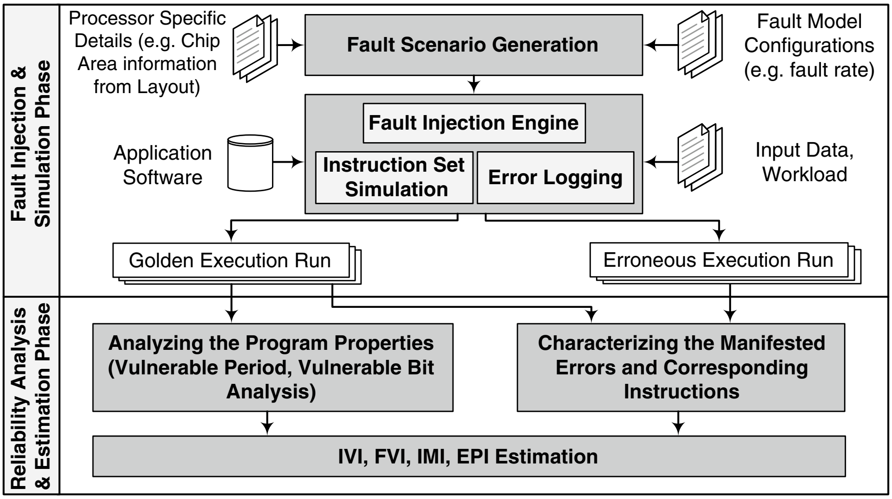
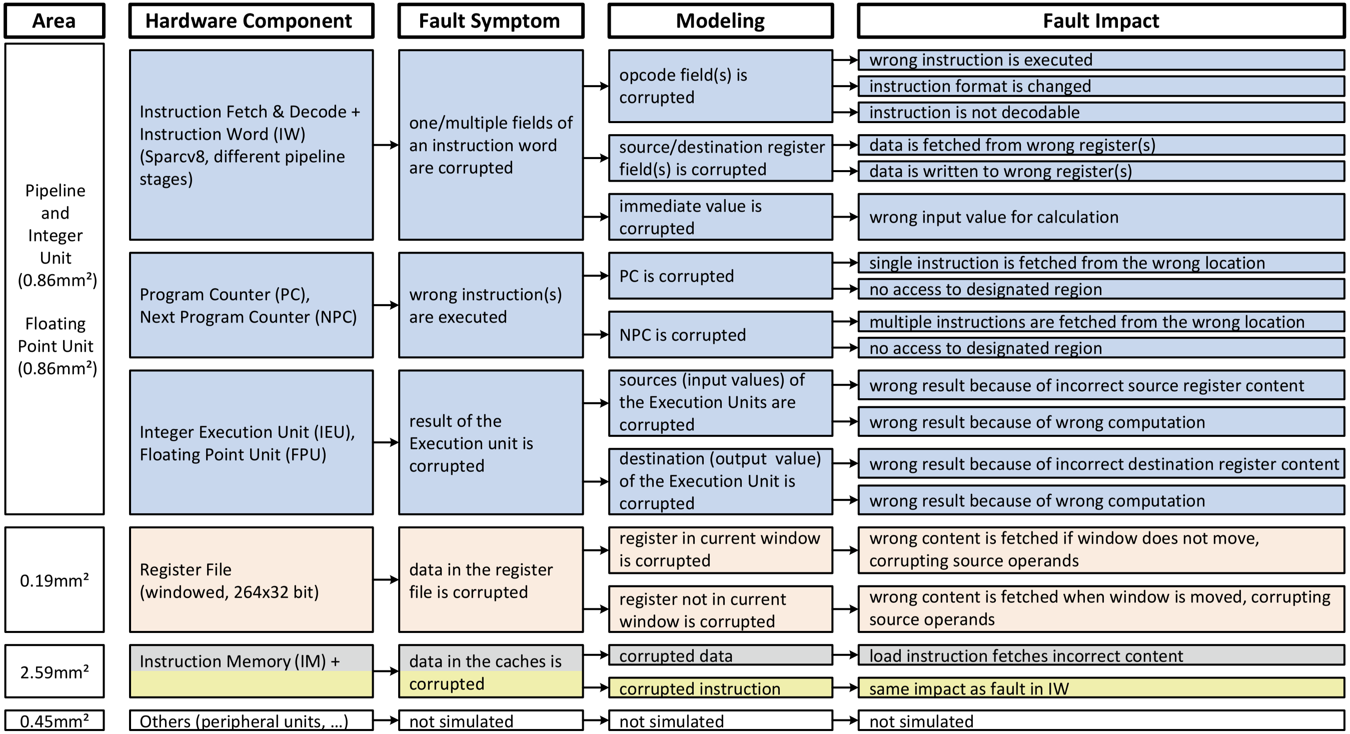
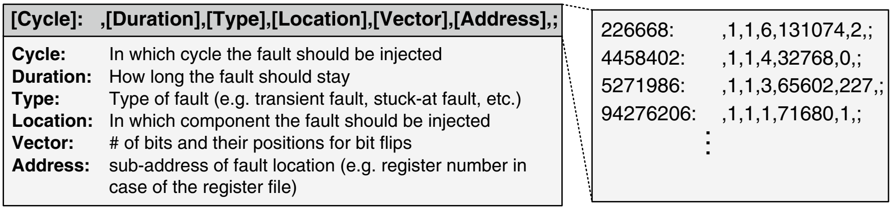

Fault Injection & Simulation Phase
Processor Specific Details (e.g. Chip Area information from Layout)
Fault Scenario Generation
Fault Model Configurations (e.g. fault rate)
Fault Injection Engine
Application Software
Instruction Set Simulation
Error Logging
Input Data, Workload
Golden Execution Run
Erroneous Execution Run
Reliability Analysis & Estimation Phase
Analyzing the Program Properties (Vulnerable Period, Vulnerable Bit Analysis)
Characterizing the Manifested Errors and Corresponding Instructions
IVI, FVI, IMI, EPI Estimation

Fig. A.3 Flow of the reliability-aware simulation and analysis



Area	Hardware Component	Fault Symptom	Modeling	Fault Impact
Pipeline and Integer Unit (0.86mm²) Floating Point Unit (0.86mm²)	Instruction Fetch & Decode + Instruction Word (IW) (Sparcv8, different pipeline stages)	one/multiple fields of an instruction word are corrupted	opcode field(s) is corrupted	wrong instruction is executed
				instruction format is changed
				instruction is not decodable
			source/destination register field(s) is corrupted	data is fetched from wrong register(s)
				data is written to wrong register(s)
			immediate value is corrupted	wrong input value for calculation
	Program Counter (PC), Next Program Counter (NPC)	wrong instruction(s) are executed	PC is corrupted	single instruction is fetched from the wrong location
				no access to designated region
			NPC is corrupted	multiple instructions are fetched from the wrong location
				no access to designated region
	Integer Execution Unit (IEU), Floating Point Unit (FPU)	result of the Execution unit is corrupted	sources (input values) of the Execution Units are corrupted	wrong result because of incorrect source register content
				wrong result because of wrong computation
			destination (output value) of the Execution Unit is corrupted	wrong result because of incorrect destination register content
				wrong result because of wrong computation
0.19mm²	Register File (windowed, 264x32 bit)	data in the register file is corrupted	register in current window is corrupted	wrong content is fetched if window does not move, corrupting source operands
			register not in current window is corrupted	wrong content is fetched when window is moved, corrupting source operands
2.59mm²	Instruction Memory (IM) +	data in the caches is corrupted	corrupted data	load instruction fetches incorrect content
			corrupted instruction	same impact as fault in IW
0.45mm²	Others (peripheral units, ...)	not simulated	not simulated	not simulated

Fig. A.4 Modeling hardware-level faults in different processor components at the ISS-level (an example for the case of SPARC v8 architecture)



[Cycle]: ,[Duration],[Type],[Location],[Vector],[Address],;

Cycle:	In which cycle the fault should be injected
Duration:	How long the fault should stay
Type:	Type of fault (e.g. transient fault, stuck-at fault, etc.)
Location:	In which component the fault should be injected
Vector:	# of bits and their positions for bit flips
Address:	sub-address of fault location (e.g. register number in case of the register file)

```
226668:     ,1,1,6,131074,2,;
4458402:    ,1,1,4,32768,0,;
5271986:    ,1,1,3,65602,227,;
94276206:   ,1,1,1,71680,1,;
              ⋮
```

Fig. A.5 Format and an excerpt of a fault file

Configurable Fault Generation Engine: This component generates a set of *fault files*, that contain the information about the faults to be injected later (see Fig. A.5 for an excerpt) providing details on *when* (i.e., in which cycle) and *where* (i.e., in which processor component) a fault is to be injected. The fault generation module works independent from the fault injection module. The reasons for this are: (1) to reuse the same fault scenarios for different applications for comparison and for reproducibility of the results and (2) to extend (if required) the fault generation module with additional parameters. Depending upon the input test conditions (e.g., number of bit flips and fault rates), the input parameters are configured.

Once the Fault Generation Engine finishes its execution, it outputs a *fault file* which comprises of two parts: (1) a header, which summarizes the information regarding the configuration settings for the fault file generation module and (2) the content, which shows the detailed information regarding the faults, where each line has the cycle information, i.e., the start cycle of a fault. The fault injection related entries are divided into blocks of five comma-separated fields. Each block represents one fault with its duration, type of fault, the fault location (i.e., the component in which the fault should be injected), a fault vector which specifies the number and the exact position of the bits where the faults are injected, and a specific address within a fault location (e.g., memory/register address) where the faults are injected. Figure A.5 shows the content section of the *fault file*. Numerous fault files are generated representing different scenarios and configuration settings using a script. For every script execution a separate directory is created which contains a set of fault files. These fault files are finally given as an input to the fault injection engine that injects faults during the program execution.

Fault Injection Engine: The step-by-step operational flow of the fault injection engine is shown in Fig. A.6. The compiled application versions are executed on an Instruction-Set Simulator which is enhanced with the capability to trace the application execution. During the simulation of the application, the fault scenarios are applied using a fault injector and the errors triggered are logged. For fairness of comparison and reproducibility, the same fault scenarios are used for the evaluation of all applications and their versions. The results of the fault injection experiments are obtained later by analyzing the effects of hardware level faults on the application software program level for each individual simulation. Afterwards, the program output errors are categorized and the reliability for the different application versions is computed using different reliability metrics. The following steps are taken:

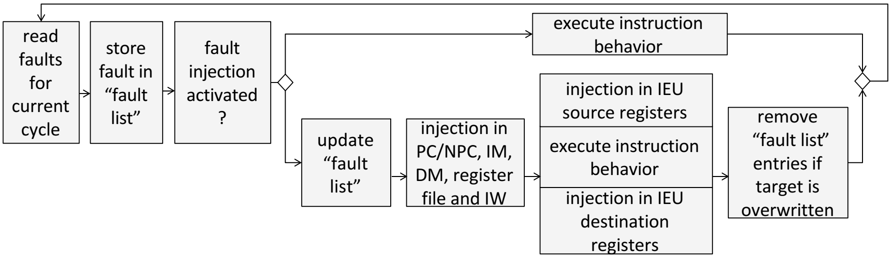

read faults for current cycle
store fault in "fault list"
fault injection activated ?
execute instruction behavior
update "fault list"
injection in PC/NPC, IM, DM, register file and IW
injection in IEU source registers
execute instruction behavior
injection in IEU destination registers
remove "fault list" entries if target is overwritten

Fig. A.6 Flow of the fault simulation process

1. The faults for the current cycle are read from the fault file and are stored in a *fault list* which contains all faults that are currently injected.
2. If the fault injection is activated (a functionality implemented to be able to inject faults only in specific functions/parts of a program), the *fault list* is updated and bit flips are injected in the respective components.
3. Afterwards the current instruction is executed.
4. Then, the *fault list* is inspected for entries whose target is overwritten. Those entries are removed from the *fault list*.
5. If the fault injection is deactivated, only the instruction is executed.

Note that the fault injection does not introduce any unwanted side effects like changing performance counters. The application program is additionally simulated without fault injection to obtain a "golden run" (i.e., correct execution). It is later used for comparison with the "erroneous run" to identify the potential errors in the program output.

Error Analysis and Reliability Estimation: An error analysis is performed for application reliability analysis while considering the application properties (e.g., histograms of the executed instructions). The error characterization and the properties of an application are used to obtain reliability metrics at different levels of granularity (i.e., the instruction and function/task level), which are used to quantify the susceptibility of an application program towards Application Failures.

Different scripts are used to automatically analyze the results of the application simulations that output a set of files, i.e., application output, a log file containing a trace of all executed instructions and summary file containing the execution time of the application and potential warning/error messages. Afterwards, the set of files obtained from the fault injection simulation of an application are compared to the set of files obtained after the fault-free simulation, i.e., the "golden run" output and the results are grouped into different categories. The category Correct Output is assigned in case the application terminates successfully and the output matches with that of the "golden run." In case the faulty and fault-free application simulations produce different or no output, a more detailed error analysis is presented in Sect. 3.2. In case of an Application Failure, an abnormal termination has to be detected. Furthermore, an error or a warning message can be seen inside the summary file. However, the log file is required to be analyzed for some special subcategories of

Application Failures, e.g., in order to identify the reason for a Segmentation Fault, the last executed instruction is required to be identified that is responsible for this type of Application Failure. In case the application is not terminated in an abnormal way, the output files have to be inspected in detail. In case the comparison shows incorrect data in the output, the category Incorrect Output is labeled.

The same procedure is repeated to generate all the simulation results and their outcome, i.e., the error distribution categories, which are produced on the basis of the created fault files with the same configuration settings and are stored in a comma-separated list. This format makes it easier to use the data for plotting the results in a graphical representation. The error characterization distribution shows the impacts that are caused by the injected faults. However, the reason for a certain application behavior (i.e., erroneous/error-free category) cannot be explained with this; therefore, a thorough analysis of the application source code and the log file is required in order to explain the reasons for a certain error category. The log file produced after the golden run is inspected to obtain the application characteristics, i.e., instruction profile, which is computed by counting the number of times each instruction type is executed. For completeness, the instructions are categorized: *call/branch/jump*, *sethi/nop*, *load*, *store*, and *logic and arithmetic* instructions. Furthermore, an average instruction profile over all executions can also be generated for a certain function. This information can be used to decide if an application is more data dominant or more control-flow dominant, and also provides an insight about the usages of the hardware components (e.g., multiplier, ALU). Additionally the value of the susceptible time for each register is calculated, which is the sum of the times between a register write and the respective last read access.

Performance Evaluation of the Developed Reliability Analysis Methodology: The reliability analysis methodology and fault injection experiments were performed using a 24-core (2.4 GHz) Opteron processor 8431 with 64 GB memory. The average performance of our fault injection and simulation is 72×10^3 SIPS (simulated instructions per second) with extensive error logging (40 MB/MCycles). The performance of the SymPLFIED [98] program-level fault injection approach is 15.2 SIPS. It shows that the proposed reliability analysis methodology offers significant performance improvement (>4K times) compared to the state-of-the-art approach, i.e., SymPLFIED, which is primarily due to the extensive model computation in SymPLFIED.

Figure A.7 illustrates the comparison of the reliability estimation accuracy of the proposed methodology and tool flow with SymPLFIED [98]. This comparison illustrates the benefits of bridging the gap between the hardware and the software to obtain accurate reliability analysis. It can be observed that in case of SymPLFIED, the number of application software *crashes due to wrong access to Instruction Memory* increases significantly. This is because of an increased number of faults that were injected in the PC. The main reason is the ignorance of the processor layout with several architecture-specific features in SymPLFIED's machine model. Therefore, the percentage of fault in the PC increases from 0.1 to 7.1 %, which leads to an average 27 % overestimation of Application Failures. The comparison analysis in Fig. A.7 demonstrates that when using the SymPLFIED technique, the

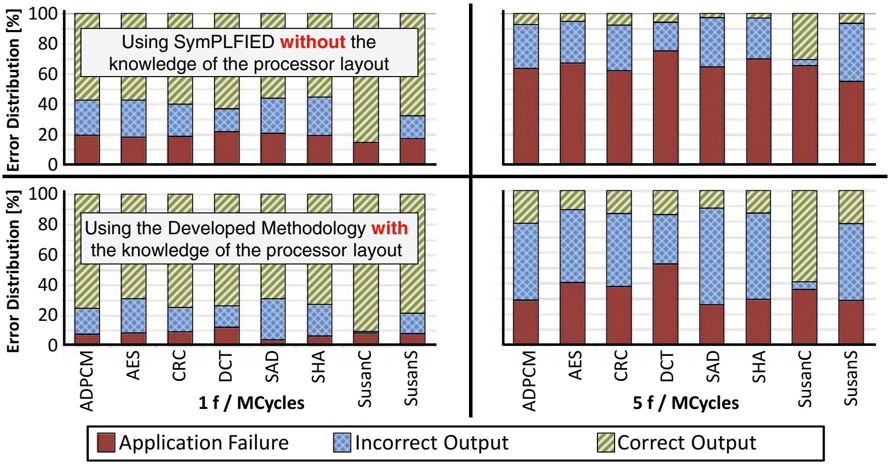


Fig. A.7 Detailed error characterization in different applications using our methodology and SymPLFIED [98]

probabilities for Application Failures and Incorrect Outputs are overestimated, which lead to an inaccurate FVI estimation. *It thereby demonstrates the improved accuracy of the proposed reliability analysis methodology and tool flow.*

Reliability-Driven Compilation: In order to compile the applications and their different versions, the GNU Compiler Collection (GCC) [129] is used. In this work, GCC is used due to its compatibility with the ArchC tool chain. In GCC several optimization options exist ranging from "O0," which enables a fast compilation and expected debugging results, to "O3," where many optimizations are activated that target performance improvement but lead to a longer compilation time. Besides that, the optimization option "Os" targets a reduction of the code size. Consequently, it is possible to create different application versions using the basic optimization options of GCC, whose impact on the software reliability can be analyzed using the presented analysis models and tools. The compiler-level reliability optimizing techniques are implemented at the source-code or assembly level or using the CDFG. The reliability analysis and estimation are done on CDFG or assembly code. In the following, potential ways for automatic application inside the integrated compiler flow are discussed.

Besides the standard optimization options for the *Reliability-Driven Software Transformations*, several additional, already existing compiler optimization passes can be used to analyze their impacts on the reliability of an application. Different application and function versions can be generated by activating/deactivating certain (optimization) options of the compiler. In GCC this can be done by either using "–f{optionName}" for activation or "–fno-{optionName}" for deactivation of an option. For example, the loop unrolling can be turned off using "–fno-unroll-loops." Additionally, several compiler constants/parameters can be changed using "– –param name=value," e.g., by setting "max-unroll-times" to a certain value the maximum amount of unrolling of a single loop can be defined [129]. Consequently, taking the example of the Reliability-Driven Loop Unrolling, the different versions that are used for the fault injection analysis can either be (1) generated automatically using GCC (using the above mentioned options and parameters) or can be (2) implemented in a high-level language at the source-code level.

To enable the *Selective Instruction Protection* and *Reliability-Driven Instruction Scheduling*, two alternatives can be selected: (1) After the compilation stage is finished or in case the high-level language source code is not available, the assembly code can be used as an input for, e.g., duplicating certain instructions, changing register allocations, and adding check instructions based on the instruction vulnerabilities. Afterwards, the modified assembly code can be assembled and linked. (2) The compiler can be enhanced by adding an additional optimization pass with additional input data, e.g., the instruction vulnerabilities. As an alternative, the vulnerability model at the required granularity can also be integrated in the compiler, e.g., using information on instruction dependencies and register allocation, as a static vulnerability estimation at compile time. Taking the example of the Reliability-Driven Instruction Scheduling, the basic block separations and the corresponding branch probabilities available in GCC can be taken advantage of.

Appendix B
Function-Level Resilience Modeling

In this appendix, a function-level resilience modeling technique is presented that was developed in the scope of this manuscript. The proposed resilience model quantifies the resilience of a given application function against the hardware-induced errors. This model can be used for characterizing the reliability importance of different functions and employing function-level reliability optimization techniques.

B.1: Definition

The *resilience* of an application function is defined as the probabilistic measure of functional correctness (output quality) in the presence of faults.

B.2: Modeling Function Resilience

Modeling resilience requires error probabilities for basic blocks outputs. There are two possible error types: Incorrect Output and Application Failure. Therefore, output of each instruction in a given basic block can be modeled as a Markov Chain with three states: S_{C}, S_{IC}, and S_{F} denoting Correct Output, Incorrect Output, and Application Failure states, respectively (see Fig. B.1). Considering that the execution of a program is a stochastic process, we adopt the Markov Chain technique for output modeling as it provides a fair tradeoff between the model complexity and accuracy when compared to exhaustive Monte-Carlo Simulations, Fault-Tree Analysis, and Principal Component Analysis based reliability models.

Assuming that each state is dependent upon the previous instructions' output and the error state can only be observed at the end or at the time of Application Failure, the execution path can be modeled as a Hidden Markov Chain, with the above-discussed

S. Rehman et al., *Reliable Software for Unreliable Hardware*,
DOI 10.1007/978-3-319-25772-3

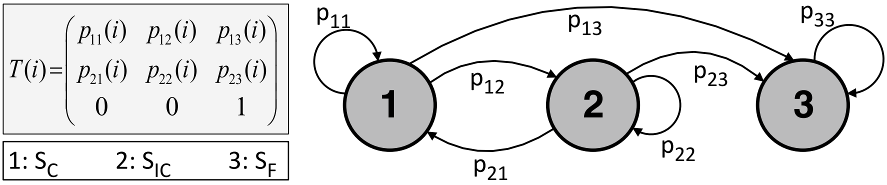


Fig. B.1 Markov Chain for instruction output with state transition probabilities

three states as hidden states and the observation state as "application failed" or "not-failed." The parameters of this model are the state transition probabilities as given in the matrix T and shown in Fig. B.1. These probabilities depend upon the executed instructions. Note, the Markov Chain is non-homogeneous as the transition probabilities change depending upon an instruction I_{ijk}.

After these probabilities are estimated (see parameter estimation later in this section), we can then compute the final state probability for a given basic block B_{ij} using Eq. A.1, where ξ is the final state probability vector containing the probability of three states: p_{C}, p_{IC}, and p_{F}.

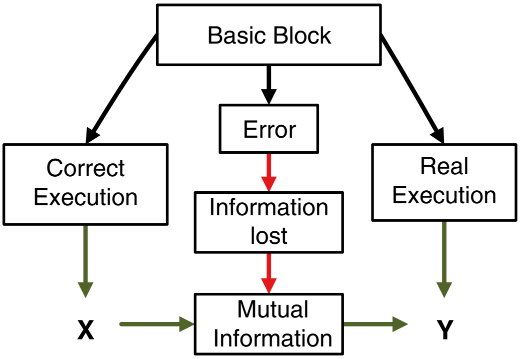
$$\xi\left(B_{ij}\right)=\left[p_{\mathrm{C}} \quad p_{\mathrm{IC}} \quad p_{\mathrm{F}}\right]_{B_{ij}}=\xi\left(B_{ij-1}\right)\times \Pi_{x\in I_{ij}}T\left(x\right) \tag{A.1}$$

Following the information theory concepts, *the resilience of a function is modeled as the normalized mutual information between the required correct result* (*from a golden execution run X*) *and the result at the end of a function execution* (from a potentially faulty execution under a given fault rate), i.e., amount of useful function output. Mutual information is a measure of the amount of correct/useful information that can be inferred from the true result of the function/basic block (Fig. B.2 explains this concept). A large value of mutual information illustrates that more information about the correct output can be inferred, i.e., high resilience.

The mutual information between an always correct execution X and a real execution Y that may have some errors is represented as $I(X;Y)=H(X)-H(X|Y)$. The function $H(X)$ is the information obtained from the correct execution, which is 1 since the correct execution contains all the information possible. The conditional entropy is the information lost $H(X|Y)$ out of $H(X)$ which is the correct information. These concepts are used to quantify the resilience of a basic block B_{ij} which can be computed as $R(B_{ij})=1-H(X|Y)/H(X)$, where $H(X)$ is the information about the correct execution, i.e., $H(X)=b_{\mathrm{Live}}$, where b_{Live} denotes the bits of live output registers of B_{ij}.

The conditional entropy $H(X|Y)$ is now the information lost in B_{ij} and given as Eq. A.2, where $p_{\mathrm{C}}(x)$ represents the probability of correct value being x; and $p_{[\mathrm{IC,F}]}(x, y)$ is the conditional probability of faulty output being Incorrect Output or Application Failure.

Fig. B.2 Flow for estimating the mutual information for function resilience



$$R\left(B_{ij}\right)=1-H\left(X\mid Y\right)/H\left(X\right);\quad H\left(X\right)=b_{\text{Live}}$$
$$H\left(X\mid Y\right)=\sum_{x\in X,y\in Y} p_{[\text{IC,F}]}\left(x,y\right)\times\log_2\left(p_{\text{C}}\left(x\right)/p_{[\text{IC,F}]}\left(x,y\right)\right) \tag{A.2}$$

Assuming, resilience of a basic block $R(B_{ij})$ can be characterized as resilience to Incorrect Output and resilience to Application Failures, we can compute the conditional entropy separately for both cases. $H(X|Y)_{\text{F}}$ is given as $p_{\text{F}}(B_{ij})$ using Eq. A.1, while $H(X|Y)_{\text{IC}}$ is given by Eq. A.3.

$$H\left(X\mid Y\right)_{\text{IC}}=-\left[p_{\text{IC}}\times\log_2\left(p_{\text{IC}}/\left(2^n-1\right)\right)+\left(1-p_{\text{IC}}\right)\times\log_2\left(1-p_{\text{IC}}\right)\right]_{B_{ij}} \tag{A.3}$$

By replacing the terms of Eq. A.2 with Eq. A.3, we can compute the resilience of a basic block against Application Failures and Incorrect Outputs where the second term in Eq. A.4 denotes the combined information loss.

$$R\left(B_{ij}\right)=1-\left[H\left(X\mid Y\right)_{\text{IC}}+H\left(X\mid Y\right)_{\text{F}}-\left(H\left(X\mid Y\right)_{\text{IC}}\times H\left(X\mid Y\right)_{\text{F}}\right)\right]/H\left(X\right) \tag{A.4}$$

Given the resilience values of all basic blocks B_i of a function f_i, resilience $R(f_i)$ can be computed using Eq. A.5.

$$R\left(f_i\right)=\sum_{\forall b\in Bi}\left(R\left(b\right)/eF_i\left(b\right)\right)\times\sum_{\forall b\in Bi,\forall fi\in F}\left(eF_i\left(b\right)\right) \tag{A.5}$$

Parameter Estimation: For estimating the model parameters, i.e., transition probabilities given in Eq. A.1, a few assumptions are made:

1. Observation of faulty output is made at the end of function
2. No recovery mechanism and no error protection is available, i.e., starting from a base case of unreliable hardware $\Longrightarrow p_{33}=1;\ p_{21}=0$.
3. Initial state and input is error-free; $[p_{\text{C}}\ p_{\text{IC}}\ p_{\text{F}}]_{(t=0)f1}=[1\ 0\ 0]$.

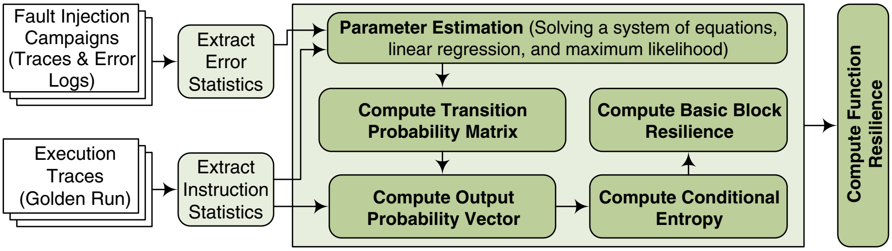

Fault Injection Campaigns (Traces & Error Logs)
Execution Traces (Golden Run)
Extract Error Statistics
Extract Instruction Statistics
Parameter Estimation (Solving a system of equations, linear regression, and maximum likelihood)
Compute Transition Probability Matrix
Compute Basic Block Resilience
Compute Output Probability Vector
Compute Conditional Entropy
Compute Function Resilience

Fig. B.3 Flow of steps to compute basic block and function resilience

Moreover, $p_{11}+p_{12}+p_{13}=1$ and $p_{21}+p_{23}=1$. To expedite the parameter estimation process, the instructions are grouped into N_T primitive instruction categories (like arithmetic, multiply, divide, logical, load/store, calls/jumps, and floating point) such that all instructions in a given category share the same transition probabilities. The parameters can be estimated through extensive fault injection campaigns. Consider there are N_S different fault-injection experiments at a given fault rate, N_C and N_{IC} are the number of cases with Correct Output and Incorrect Output, respectively. For a particular fault injection experiment s, for a certain instruction category t_k, the transition probability p_{11} can be estimated using the maximum likelihood, thus deriving Eq. A.6. NI(t,s) denotes the number of instructions of type t in simulation s.

$$\log\left(p_{11}\left(t_k\right)\right)=-\mathrm{NI}\left(t_k,s\right)\times\left(\frac{\sum_{\forall s\in S}\log\left(N_S/N_C\left(s\right)\right)+\sum_{t=0,t\neq ts}^{N_T}\mathrm{NI}\left(t,s\right)\times\log\left(p_{11}\left(t\right)\right)}{\left(\sum_{\forall s\in S}\mathrm{NI}\left(t_k,s\right)\right)^2}\right) \tag{A.6}$$

Assuming $p_{23}(t)=p_{13}(t)$, Eq. A.6 is utilized to obtain the probability $p_{22}(t_k)$. In this way all the remaining transition probabilities are computed, such that, $p_{23}(t_k)=p_{13}(t_k)$ $=1-p_{22}(t_k)$; and $p_{12}(t_k)=p_{22}(t_k)-p_{11}(t_k)$.

Figure B.3 shows a simplified flow of different steps of our scheme towards modeling and estimation of function resilience along with parameter estimation and computation of conditional entropy.

Complexity: The complexity of resilience estimation is O($|B_i|\times N_T\times\log(|I_{ij}|)$), which is much smaller than the complexity of fault tree based methods (i.e., O($|B_i|\times|I_{ij}|^3$)) and Monte-Carlo simulations (i.e., O($|B_i|\times|I_{ij}|^2$)) for each basic block.

B.3: Results

The resilience model quantifies the reliability properties at a coarse-grained level, i.e., function and basic block that can be used to facilitate in prioritizing different functions and basic blocks for selective protection/constrained reliability optimization. For example, allocating the performance quotas to different functions/basic blocks depending upon their higher/lower resilience values.

In this work the resilience is used as a metric to quantify the coarse grained reliability, i.e., at function/basic block level and then using the resilience values for distributing the tolerable performance overhead quota among different functions and different basic blocks inside the application program. A more resilient function would get a less quota for protection compared to a less-resilient function that may not tolerate more errors. Figure B.4 shows the resilience (in log scale) and the performance overhead quota for different application functions. The resilience and quota are provided separately for the Incorrect Output and Application Failure cases along with the combined case. Note, here Incorrect Output and Application Failure are both treated as information loss. Due to the high resilience, *DCT* gets lesser quota in comparison to the *ADPCM*, *SHA*, and *SAD*. The resilience of *DCT* is high because it is an unrolled version, with a relatively lesser number of branches, i.e., critical instructions, compared to other applications that lead to fewer control flow errors in *DCT*.

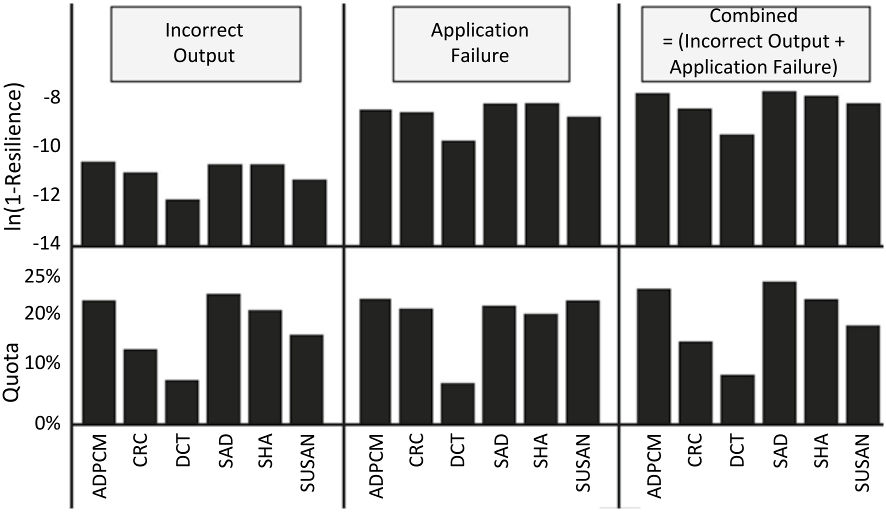


Fig. B.4 Resilience of various application functions (inverse values in log scale): resilience is shown separately for Incorrect Output and Application Failure, and *Combined*

Appendix C
Algorithms

C.1: Algorithm for Computing the Error Masking Probability PDP(*I*, *p*)

```
Error Masking Probability Computation

Input: G (V, E), $L_G$, (P, S)
Output: Masking probabilities due to data flow for each instruction I for path p, $P_{DP}$ (I, p)
    FOR all I $\in$ G DO
        $P_D$ (I) $\leftarrow$ compute$P_D$ (I); // Eq. 4.7
    END FOR
    FOR all I $\in$ $L_G$ DO
        $P_{DP}$ (I, p) $\leftarrow$ $P_D$ (I) //  for all leaf nodes
    END FOR
    List L();
    FOR all x $\in$ $L_G$.P DO
        L.add(x); //  list of ready nodes
    END FOR
    WHILE (!L.isEmpty()) DO
        FOR all I $\in$ L DO
            I.Paths $\leftarrow$ generatePaths(I);    //  generate all instruction paths
            FOR all p $\in$ I.Paths DO
                $N_B$ $\leftarrow$ 0;    p' $\leftarrow$ p;
                FOR all x $\in$ p' DO    //  compute
```

S. Rehman et al., *Reliable Software for Unreliable Hardware*,
DOI 10.1007/978-3-319-25772-3

```
number of consecutive instructions of Type B
                        IF (x == typeB) THEN
                        $N_B$ ← $N_B$ + 1;    p'.remove(x);
                        ELSE
                        NB ← 0;    p'.remove(x);
                    END IF
                END FOR
                IF ($N_B$ < 1) THEN    // compute masking probabilities
                    $P_{DP}$ (I, p) ← $P_D$ (I) + (1 - $P_D$ (I)) × $P_D$ (I.s);
                ELSE
                    $P_D$' (I) ← $\sum_{x=l}^{1+N_B} P_D(x)$;
                    $P_{DP}$ (I, p) ← $P_D$' (I) + (1 - $P_D$' (I)) × $P_D$ (I.s);
                END IF
            END FOR
            L.remove(I);
            FOR all ip ∈ I.P DO
                L.add(ip);
            END FOR
        END FOR
    END WHILE
```

C.2: Algorithm for Computing the Instruction Error Propagation Index

```
________________________________________________________________
Instruction Error Propagation Index Computation
________________________________________________________________

Input: Instruction flow graph G (V, E), set of leaf nodes $L_G$, masking probabilities due to dataflow $P_{DP}$(I, p),control flow probabilities $P_{CF}$ (p | I).
Output: set of error propagation indices for each instruction I, EPI (I).
    FOR all I ∈ $L_G$(G) DO
        EPI (I) ← 1; // initialization to consider error propagation of leaf nodes
    END FOR
    List C($L_G$);    Queue Q();// list of traversed
```

```
instructions and queue of evaluated instructions
FOR all I ∈ $L_G$(G) DO
    FOR all i ∈ I.P DO
        Q.Enqueue(i);
    END FOR
END FOR
WHILE (!Q.isEmpty()) DO
    I ← Q.Dequeue(I);
    IF (∀ s ∈ I.S, s ∈ C) THEN    // compute EPI for all instructions with successors in C
        EPI ← 0;
        FOR all s ∈ I.S DO
            IF (IMI(s) == 0) THEN    // if successor is a non-masking instruction
                EPI ← EPI + (EPI(s) × $P_{Execution}$(s | I));
            ELSE
                EPI ← EPI + $\sum_{\forall p \in s.Paths} ((1-P_{DP}(s,p))) \times P_{CF}(p \mid I) \times EPI(L_G(p)))$;
            END IF
        END FOR
        EPI(I) ← EPI × ($P_{IO}$(I) / ($P_{IO}$(I) + $P_{Cr}$(I)));    C.add(I);
        FOR all I ∈ I.P DO
            Q.Enqueue(I);
        END FOR
    ELSE
        Q.Enqueue(I);
    END IF
END WHILE
```

C.3: Algorithm for FVI-Driven Data Type Optimization

Algorithm C.3 presents the pseudo-code targeting load merging (for store instructions, the procedure is similar).

Input: Graph $G(V, E)$ of the function F, P_τ as the tolerable performance overhead, Data Type, *FVI* and performance of the original code (FVI_{Orig}, P_{Orig}).

Output: Transformed function *fd* with merged loads and extraction code as a result of the data type optimization.

```
FVI-Driven Data Type Optimization

Input: G (V, E), P$_{\tau}$, FVI$_{orig}$, P$_{orig}$, DataType
Output: Transformed function fd
    A ← getAllArrays(G);
    FOR all a ∈ A DO
        List <V> L ← getLoads(a, G);
        IF (DataType == INT) THEN
            continue;
        END IF
        FVI$_{Best}$ ←FVI$_{Orig}$;
            WHILE L ! = ∅ DO
                G’ ← G;
                (l$_1$, l$_2$) ← getCurrent&NextLoads(L);
                l ← Merge(l$_1$, l$_2$);
                G’.remove(l$_1$, l$_2$);   G’.insert(l);   G’.insertExtractionCode();
                (FVI, P, Spill) ← Evaluate(G’); // compile and execute, estimate FVI, performance, and check for spilling
                IF ((P/P$_{Orig}$ – 1) > P$_{\tau}$) THEN
                    break;
                END IF
                    IF ((FVI < FVI$_{Best}$) && (!Spill)) THEN
                        FVI$_{Best}$ ← FVI;   L.remove(l$_1$,l$_2$);
                        G.remove(l$_1$,l$_2$);   G.insert(l);
                        G.insertExtractionCode();
                END IF
            END WHILE
    END FOR
    fd ← G;
    return fd;
```

C.4: Algorithm for FVI-Driven Loop Unrolling

Input: a set of maximum unrolling factors for all loops, the FVI, performance, and code size of the original function *F*.

Output: the transformed function *fd* with loop unrolling applied by an *FVI-minimizing unrolling factor*.

```
FVI-Driven Loop Unrolling

Input: Function F, Set of maxUnrollFactors, $FVI_{Orig}$, $P_{Orig}$, $C_{Orig}$, µ.
Output: Transformed function fd
list< Loop > L ← getLoops(F);
FOR all l ∈ L DO //  determine unrolling factor for each loop
    maxUF = getFactor(l, maxUnrollFactors);
    $unrollProfit_{Best}$ ← minINT;    $uF_{Best}$ ← 1;
    FOR $uF_i$ = 1 to maxUF DO
        $l_{temp}$ ← l;    //  create a temporary copy of the loop
        $F_{uFi}$ ← Unroll (F, $l_{temp}$, $uF_i$);    //  Unroll by a factor $uF_i$
        (FVI, P, C, Spill) ← Evaluate($F_{uFi}$); // compile. execute, estimate FVI and
                                                     performance, and check for spilling.
        $FVI_{Benefi}$t ← ($FVI_{Orig}$ – FVI)/$FVI_{Orig}$;
        $P_{Loss}$ ← (P – $P_{Orig}$)/$P_{Orig}$; //  Performance loss
        $C_{Loss}$ ← (C – $C_{Orig}$)/$C_{Orig}$; // Code size increase
        unrollProfit = computeProfit($FVI_{Benefit}$, $P_{Loss}$, $C_{Loss}$, µ);
        IF ((unrollProfit > $unrollProfit_{Best}$) && (!Spill)) THEN
            $unrollProfit_{Bes}$t ← unrollProfit;    $uF_{Best}$ ← $uF_i$;
        END IF
    END FOR
    setBestUnrollFactor(l, $uF_{Best}$);
END FOR
FOR all l ∈ L DO //  generate the transformed function using the best unroll factors
    $UF_{Best}$ = getBestUnrollFactor(l);
    fd ← Unroll(fd, l, $UF_{Best}$);
END FOR
return fd;
```

C.5: Algorithm for Applying Common Expression Elimination

Algorithm C.5 shows the pseudo-code of the algorithm to evaluate the reliability benefit of replacing common expressions.

Input: Graph G (V, E) of the function F, P_τ as the tolerable performance overhead, FVI and performance of the original code ($\mathrm{FVI}_{\mathrm{Orig}}$, P_{Orig}).

Output: Transformed function *fd* where the common expressions are (partially) replaced.

```
Common Expression Elimination

Input: G (V, E), $P_\tau$, $FVI_{Orig}$, $P_{Orig}$
Output: Transformed function fd
CG ← getCEs(G); // get all common sub-graphs/expressions in G
FOR all c ∈ CG DO // Evaluation Phase
    O ← getOccurrences(c, G);
    FOR all o ∈ O DO
        $f_1$ ← computeFVI(replace CE at o in G);
        $f_2$ ← computeFVI(keep CE at o in G);
        $\Delta \leftarrow f_2 - f_1$;
        IF ($\Delta \geq 0$) THEN
            o.mode ← set(replace_CE);
        ELSE
            o.mode ← set(keep_CE);
        END IF
    END FOR
    G′ ← updateGraph(G); // replace occurrences based on o.mode
    $FVI_c$ ← computeFVI(G′); // FVI
    $P_c$ ← computePerformance(G′); // performance
    $\varepsilon_c \leftarrow (FVI_{Orig} - FVI_c)/(P_c - P_{Best})$; // efficiency
END FOR
sort($L_{CE}$); // sort common sub-graphs/expression by their efficiency
FOR all l ∈ $L_{CE}$ DO // Elimination Phase
    IF (l.$P_c$ − $P_{Bes}$t ≤ $P_\tau$) THEN
        O ← getOccurences(l.c, G);
        FOR all o ∈ O DO
            IF ((o.mode == replace_CE) && (!Spill))
```

```
THEN
                G.remove(o);   G.insert(CE_variable);
                $P_\tau \leftarrow P_\tau - o.latency()$;
            END IF
        END FOR
    END IF
END FOR
fd ← G
return fd; // return the code with expression elimination
```

C.6: Algorithm for Soft-Error-Driven Instruction Scheduler

```
Soft Error Driven Instruction Scheduler

Lookahead (): Input: Instruction Graph G = (V, E), Tolerable performance overhead $P_\tau$.
        Output: Instruction Schedule $G_S$.
$G_S \leftarrow \emptyset$; $G_{SC} \leftarrow \emptyset$; // set of scheduled and candidate instructions
FOR all n ∈ V DO
    S[n] ← n.getSucc();   P[n] ← n.getPred();
    IF (ready(n)) THEN
        $G_{SC} \leftarrow G_{SC} \cup n$; // add to the ready list
    END IF
END FOR
FOR all $n \in G_{SC}$ DO
    eT[n] ← 0; // initialization of the earliest time
END FOR
$T_{curr} \leftarrow 0$;   $i_{PSel} \leftarrow \emptyset$ ;   $i_{RSel} \leftarrow \emptyset$ ; // Performance and reliability maximizing instruction
$\Psi_{max} \leftarrow -\infty$;   $\delta_{max} \leftarrow -\infty$
WHILE ($G_{SC}$ != ∅ ) DO
    FOR all $i \in G_{SC}$ DO
            Ψ[i] ← estimateReliabilityWeight(i); // Eqs. 5.2-5.5;
            IF ((δ[i] > $\delta_{max}$) && eT[i] ≤ $T_{curr}$) THEN // obtain instruction considering performance
                $\delta_{max} \leftarrow$ δ[i];      $i_{PSel} \leftarrow$ i ;
            END IF
            IF ($P_\tau$ > 0) THEN
                FOR all j ∈ ($G_{SC}$ − {i}+ j.getSchedulableSN()) DO// evaluate candidates with lookahead
                    $\Psi_{ij} \leftarrow \Psi[i] + \Psi[j]$;    $\delta_{Loss} \leftarrow \delta_{max} - \delta[i]$;
                    IF (($\Psi_{ij} > \Psi_{max}$) && ($\delta_{Loss} < P_\tau$)) THEN
                        $\Psi_{max} \leftarrow \Psi_{ij}$;    $i_{RSel} \leftarrow i$;
                    ELSE IF (($\Psi_{ij} == \Psi_{max}$) && ($\delta_{Loss} < P_\tau$) && ($\delta[i] > \delta[i_{RSel}]$) && ($eT[i] \leq Tcurr$)) THEN
                        $\Psi_{max} \leftarrow \Psi_{ij}$;    $i_{RSel} \leftarrow i$;
                    END IF
                END FOR
                IF ($i_{RSel}$ != ∅ ) THEN
                    $i_{Sel} \leftarrow i_{RSel}$;    $P_\tau \leftarrow P_\tau - (\delta_{max} - \delta[i_{Sel}])$; // select reliability-wise best solution
                ELSE
                    $i_{Sel} \leftarrow i_{PSel}$; // select performance-wise best solution
                END IF
            ELSE
                $i_{Sel} \leftarrow i_{PSel}$;
            END IF
        END FOR
        $T_{curr} \leftarrow T_{curr} + T[i_{Sel}]$;    $G_{SC} \leftarrow G_{SC} - i_{Sel}$;    $G_S \leftarrow G_S \cup i_{Sel}$;
        FOR all s ∈ S[$i_{Sel}$] DO// update the set of scheduling candidates
            IF (∀m ∈ P[s] ∃ t | $V_{GS}[t]$ = m) THEN
                $G_{SC} \leftarrow G_{SC} \cup s$;    $eT[s] \leftarrow T_{curr} + eT[i_{Sel}]$;
            END IF
        END FOR
    END WHILE
    return $G_S$;
```

C.7: Algorithm for Selective Instruction Protection Technique

```
________________________________________________________________
Selective Instruction Protection
________________________________________________________________

Input: Unprotected function F from the software program as G = (V, E), user provided tolerable performance overhead in cycles $P_\tau$, set of instruction vulnerabilities IVI, set of error
```

```
propagation indices for all instructions EPI, user provided program reliability method R (for instance, SWIFT-R 0)
Output: Function with selective instruction protection F'.
    List L;
    FOR all i ∈ G DO// compute the reliability profit for all instructions
        RPF(i) ← (EPI(i) ∈ IVI(i))/ω(i);
    END FOR
    Sort(L, RPF, Descending order);
    WHILE (!L.empty() && ($P_\tau$ > 0)) DO
        I ← L.pull();
        IF (ω(I) ≤ $P_\tau$ ) THEN
            Protect(I);        $P_\tau$ ← $P_\tau$ - ω(I);
            FOR all i ∈ (I.S ∈ I.P) DO
                GI ← generateGroups(I, i); // groups of consecutive instructions
                FOR all g ∈ GI DO // compute overhead of instruction groups
                    g' ← g;
                    FOR all i ∈ g DO
                        IF ((i.S > I) && ($\exists_s \in_{i.s}$ inGroup(s,g') == False)) THEN
                            setCheckInstructionPt(i, g');
                    END IF
                        setCheckInstructionPt(Leaf(g), g');
                    END FOR
                    ω(i, g') ← getOverload(g', R);
                END FOR
                RPF(i) ← (EPI(i) × IVI(i))/ω(i, g');
            END FOR
            Sort(L, RPF, DescendingOrder);
        END IF
    END WHILE // end while loop if budget is over or all instructions are protected
```

C.8: Algorithm for Offline Table Construction

Algorithm C.8 describes the procedure of offline table construction by adopting the approximations explained in Sect. 6.1, and by using δ as the timing unit and α as the reliability penalty unit.

```
Offline Table Construction

Input: n functions, CDF and PDF of the functions, units δ and σ, weighted parameter α, and the default versions θ () after observing the deadline misses;
1.  FOR r ← $\frac{R_{\min}(n)}{\sigma}\sigma,\ldots,\frac{R_{\max}(n)}{\sigma}\sigma$ , stepped by σ DO
2.      FOR t ← $0,\ldots,\frac{D}{\delta}\delta$ , stepped by δ DO
3.          Calculate j*(n, r, t) and G(n, r, t) by using Eq. 6.2
4.      END FOR
5.  END FOR
6.  FOR i ← n-1, n-2,…., 2 DO
7.      FOR r ← $\frac{R_{\min}(i)}{\sigma}\sigma,\ldots,\frac{R_{\max}(i)}{\sigma}\sigma$ , stepped by σ DO
8.          FOR t ← $0,\ldots,\frac{D}{\delta}\delta$ , stepped by δ DO
9.              IF (t == 0) THEN
10.                 j*(i, r, t) ← θ$_i$ ; G(i, r, t) ← α (r + ρ(i)) + (1 − α)
11.             ELSE
12.                 FOR each j = 1, 2,…., K$_i$,
13.                     Calculate H$_j$ ← $\int_{x=0}^{t} P_{i,j}(x).G\left(i+1,\frac{r+R_{i,j}}{\sigma}\sigma,\frac{t-x}{\delta}\delta\right)dx$ + $\left(1-C_{i,j}(t)\right).\left(\alpha\left(r+R_{i,j}+\rho(i+1)\right)+(1-\alpha)\right)$;
14.                 j*(i, r, t) ←argmin j=1,2,….,Ki H$_j$ ;
15.                 G(i, r, t) ← $H_{j^*(i,,r,,t)}$
16.             END IF
17.         END FOR
18.     END FOR
19. END FOR
20. Calculate j*(1, 0, D) and G (1, 0, D) using the same procedure as in Steps 14 and 15.
21. Return the table j*.
```

C.9: Algorithm for Hybrid RMT Tuning

Hybrid RMT Tuning

Input: set T of tasks; a list of available/free cores $List_c$; list of different compiled versions $\{t_1 = \{t(_{1},_{1}),…, t(_{1},_{K1})\},…, t_M = \{t(_{M},_{1}), …, t(_{M},_{KM})\}$; history H of last S_H RMT comparison results; list $TR\{_{RMT},_{NR}\}$ of running tasks (sorted by RMT level and RTP).

Output: performance-wise sorted version lists $V_{t1},…, V_{tM}$ containing different reliability performance tradeoffs per task, list L of tasks with a protection type pt and number of allocated cores rc used for RMT.

```
FOR all t ∈ T DO // estimate performance and RTP for all task versions
    FOR all cv ∈ t DO   // loop over all task versions
        cv.L = estimatePerformance(cv);
        cv.RTP = calculateRTpenalty(cv);
        $V_t$.insert(cv);
    END FOR
    $V_t$.sort(cv.L);
END FOR
List TL;
FOR all t ∈ T DO
    t.v = $V_t$.head(1); TL.insert(t);
END FOR
TL.sort(t.RTP);    // sort task list by RTP
$N_{FC}$ = $List_c$.size(); // number of free cores
IF ($TR_{RMT}$.size() == 0 && $N_{FC}$ > 1) THEN // at least one task with RMT
    t = TL.pop_front(); t.pt = RMT; $TR_{RMT}$.push_back(t);
    rc  = min (coreDemandRMT, $N_{FC}$); $N_{FC}$ = $N_{FC}$ - rc; t.rc = rc;
END IF
WHILE ($N_{FC}$ > 0 && TL.size() > 0) DO // allocate a core to each task
        t = TL.pop_front(); t.pt = NR; $T_{NR}$.push_back(t);
        $N_{FC}$ -- ; t.rc = 1;
    END WHILE
    WHILE (TL.size() > 0) DO // preemption for tasks running with RMT
              IF ($TR_{RMT}$.size() > 1) THEN
                  $t_r$ = $TR_{RMT}$.back();
              ELSE
                  break;
              END IF
              IF ($t_r$.rc == 2) THEN   // 2 is the required number of cores for RMT
                        $t_r$.pt = NR; $T_{NR}$.push_front($t_r$); $TR_{RMT}$.pop_back();
              END IF
              $t_r$.rc = $t_r$.rc - 1;
          t = TL.pop_front(); t.pt = NR; $TR_{NR}$.push_back(t); t.rc =1;
    END WHILE
    h = $\sum_{i=1}^{5} H[i]*i$; // evaluate RMT comparison history
   IF (h> SH) THEN // activate RMT mode based on comparison history
        WHILE ($N_{FC}$ ≥ coreDemandRMT - 1) DO
         t = $T_{NR}$.pop_front(); t.pt = RMT; t = $TR_{RMT}$.insert();
               $N_{FC}$ = $N_{FC}$ - coreDemandRMT - 1; t.rc = coreDemandRMT;
        END WHILE
    END IF
```

C.10: Algorithm for Reliable Code Version Tuning

```
Version Tuning

Input: lists $T_{pt}$ of tasks with protection levels pt ∈ {NR, RMT}; performance-wise sorted version lists $V_{t1}$,….,$V_{tM}$; list of available/free cores $List_c$.
Output: tasks T with set of selected versions {t($_{i}$,$_{j}$), ∀ i ∈ T, j ∈ $K_i$ }
   FOR all t ∈ $T_{RMT}$ ∪ $T_{NR}$ DO
       t.c = $List_c$.tail(1);
   END FOR
```

```
FOR all t ∈ $T_{RMT}$ ∪ $T_{NR}$ DO  // loop over all tasks
    v = $V_t$.head(1);
    FOR all j ∈ {1,…,t.rc} DO      // initialize with performance-wise best version
        t.$v_j$ = v;
    END FOR
    IF (t.rc ≤ 2) THEN  // tasks running with no redundancy
        FOR all v ∈ $V_t$  DO
            IF (v.RT > t.$v_1$.RT && v.L < t.D) THEN  // select the version with best reliability within the deadline constraint
                FOR all j ∈ {1,…,t.rc} DO
                t.$v_j$ = v;
                END FOR
            END IF
        END FOR
    END IF
END;
FOR all t ∈ $T_{RMT}$ ∪ $T_{NR}$ DO
    t.c =  ∅;
END FOR
```

C.11: Algorithm for Core Tuning and Version Update

Algorithm C.11: Pseudo-Code for the Core Tuning and Version Update

```
Core Tuning and Final Version Tuning
Input: task graph G, lists $T_{pt}$ of tasks with protection levels pt ∈ {NR, RMT}; performance-wise sorted version lists $V_{t1}$,…,$V_{tM}$; performance-wise sorted list of available/free cores $List_c$.
Output: tasks T with set of selected versions {t($_{i}$,$_{j}$), ∀ i ∈ T, j ∈ $K_i$ } and set of assigned cores {t.$c_1$,…,t.$c_{t.rc}$}
List L = $T_{RMT}$ ∪ $T_{NR}$;    L.sort(t.RTP);
FOR all t ∈ L DO //  loop over all tasks
    List $L_{dC}$ = t.getDependentTasks(G); //  list of dependent cores
```

```
        List $L_{exec}$;
        FOR all c ∈ $List_c$.head(3) DO //  analyze 3 performance-wise best cores
            comm = 0;
            FOR all $t_{dep}$ ∈ $L_{dC}$ DO //  analyze communication overhead
                dist = computeDistance(c, $t_{dep}$.c);
                comm += estimateCommOverhead(dist, $t_{dep}$.data);
            perf = estimatePerformance(c,t.v);
            END FOR
            t.exec = comm + perf;
            $L_{exec}$.insert(pair< c,t.exec >);
        END FOR
        t.$c_1$ = getCore(min(∀ t.exec ∈ $L_{exec}$)); // select core with lowest combination of execution time  and communication
        $List_c$.remove(t.$c_1$);
        i = 2;
        WHILE (i ≤ t.rc) DO // find close cores for redundant executions
            t.$c_i$ = min (∈ c ∈ $List_c$ {calcDistance(t.$c_1$,c)});
            $List_c$.remove(t.$c_i$); i + +;
        END WHILE
    END FOR
    FOR all t ∈ L DO
        tuneVersion(t);
    END FOR
```

Appendix D
Notations and Symbols

This appendix presents the table of notations/symbols/terms and their descriptions used in this manuscript. The parameters/symbols are listed in the alphabetic order in the following table.

Parameter/symbol	Description
A_c	Area of the processor component c
BB	Basic Block
CAB(I)	Number of critical address bits of the instruction I that lead to "memory segmentation" errors
ci	Checking instructions
CI	Critical Instruction
COB(I)	Number of critical operand bits of the instruction I that lead to "non-decodable instruction" errors
$C_{i,j}(D)$	Probability for which the execution time of jth version of function i is less than or equal to the deadline D
C_{Orig}	Code size of the of the Original Code without reliability-driven transformations
csi	Consecutive instructions
D	Application Deadline
DB	Dependent Basic Blocks
E	Set of edges in an application graph G
ep	Execution Path
EPI	Instruction Error Propagation Index
$\mathrm{ER}_{\mathrm{raw}}$	Raw error rate
f	Fault rate
FVI	Function Vulnerability Index
$\mathrm{FVI}_{\mathrm{Failures}}$	Function's vulnerability to Application Failures
$\mathrm{FVI}_{\mathrm{IncorrectOP}}$	Function's vulnerability to Incorrect Outputs

(continued)

S. Rehman et al., *Reliable Software for Unreliable Hardware*,
DOI 10.1007/978-3-319-25772-3

Parameter/symbol	Description
FVI_{Orig}	FVI of the of the Original Code without reliability-driven transformations
G	Application graph with a set of vertices V (or T as a set of Tasks) and edges E
$G^*(i, r, t)$	A single function version entry in the table of function schedules corresponding to the instruction i, with reliability level r and time t
I.Paths	A set of paths p for an instruction I
I_{D}	Dependent instruction
IMI	Instruction Error Masking Index
IMI(I)	Instruction Masking Index of instruction I
i_{PSel}	Performance-maximizing selected instruction during instruction scheduling
i_{RSel}	Reliability-maximizing (i.e., vulnerability minimizing) selected instruction during instruction scheduling
IVI	Instruction Vulnerability Index
IVI_i	Instruction Vulnerability Index of instruction i
IVI_{ic}	Instruction Vulnerability Index of instruction i in processor component c
$j^*(i, r, t)$	Index of the function version entry in the table of function schedules corresponding to the instruction i, with reliability level r and time t
maxUnrollFactor	Maximum value of the unrolling factor for a given function
miss rate	Percentage of deadline misses for a given application
nCI	Non-critical Instruction
P	A set of predecessor instructions for a given instruction
p	An instruction path, such that each instruction in the path has exactly one successor and one predecessor instruction
PC	Set of different pipeline stages PC = {F, D, E, M, W}, where F = Instruction Fetch, D = Instruction Decoder, E = Execute, M = Memory, W = Writeback stage in a 5-stage pipeline processor like LEON2
Proc	Set of processor components, i.e., $c \in \text{Proc}$
$P_{\text{AF}}(I)$	Probability of Application Failures in case a fault occurs during the execution of the instruction I
$P_{\text{CF}}(\text{ep}\|I)$	Execution Path Probability for an instruction I given an execution path *ep*
$P_{\text{D}}(I)$	Error Masking Probability during the execution of an instruction I with O as a set of operands
$P_{\text{D}}(x, I)$	Error Masking Probability for the operand bit x during the execution of instruction I
$P_{\text{DP}}(I, p)$	Error Masking Probability, due to data flow properties, for an instruction I along the path p until the final visible program output at the end of the path
$P_{\text{e}}(x)$	Error Probability in an operand bit x
$P_{\text{eAd}}(b, I)$	Error Probability in the address bits b of the instruction I
$P_{\text{EM}}(c)$	Error Masking Probability of a processor component c

(continued)

Parameter/symbol	Description
$P_{\mathrm{EM}}(i, PC)$	Error Masking Probability for an instruction *i* in the pipeline stage *PC*
$P_{\mathrm{eOP}}(b, I)$	Error Probability in the opcode bits *b* of the instruction *I*
$P_{\mathrm{Execution}}(s\|I)$	Execution probability of a successor instruction *s* corresponding to an instruction *I*
P_{Failures}	Probability of Application Failures
$P_{\mathrm{fault}}(c)$	Probability of Fault in a processor component *c*
$P_{\mathrm{IncorrectOP\text{-}CI}}$	Probability of Incorrect Outputs due to critical instructions
$P_{\mathrm{IncorrectOP\text{-}nCI}}$	Probability of Incorrect Outputs due to non-critical instructions
$P_{\mathrm{IO}}(I)$	Probability of Incorrect Outputs in case a fault occurs during the execution of the instruction *I*
P_{Orig}	Performance (in terms of execution time given as *cycles*) of the Original Code without reliability-driven transformations
P_{sig}	Signal probability
$P\tau$	Tolerable performance overhead
Q	Queue
R	Reliability function for quantifying the functional correctness. It can either be FVI or function resilience or any other function reliability metric
R_{max}	Maximum range of the RTP value for a given function
R_{min}	Minimum range of the RTP value for a given function
RTP	Reliability-Timing Penalty
S	A set of successor instructions for a given instruction
$TotalBits_c$	Total Bits representing the architecturally defined size of the processor component *c*
ω	Protection overhead
$vulBits_{ic}$	Vulnerable Bits of the processor component *c* executing instruction *i*
$vulP_{ic}$	Vulnerable Period of instruction *i* in processor component *c*
V	Set of vertices in an application graph *G*
ψ	Instruction Reliability Weight as a joint function of IVI, criticality and dependent instructions
ψF	Function Reliability Weight of a function *F*
$\Pi_{i,j}$	Probability of deadline misses for a given function version *j* of a function *i*

Bibliogrphy

1. G. Moore, “Cramming more components onto integrated circuits”, *Electronics*, vol. 38, no. 8, 1965.
2. Intel, http://www.intel.com/pressroom/kits/quickref.htm [Online; accessed: Apr 2015].
3. Intel, http://ark.intel.com/products/80555/Intel-Xeon-Phi-Coprocessor-7120A-16GB-1_238-GHz-61-core [Online; accessed Apr 2015].
4. Nvidia, http://www.nvidia.com/object/white-papers.html [Online; accessed Apr 2015].
5. International Technology Roadmap for Semiconductors, in *ITRS 2013 Edition—Process Integration, Devices, and Structures*, 2014.
6. K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer, “High-performance CMOS variability in the 65-nm regime and beyond”, *IBM Journal of Research and Development—Advanced silicon technology*, vol. 50, pp. 433–449, Jul. 2006.
7. S. Mitra, K. Brelsford, Y. Kim, H. Lee, and Y. Li, “Robust System Design to Overcome CMOS Reliability Challenges”, *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 1, pp. 30–41, 2011.
8. A. W. Strong, E. Y. Wu, R. P. Vollertsen, J. Sune, G. La Rosa, T. D. Sullivan, and S. E. Rauch III, “Reliability Wearout Mechanisms in Advanced CMOS Technologies”, *Wiley-IEEE Press*, vol. 12, ISBN: 978-0471731726, 2009.
9. S. Borkar and A. A. Chien, “The Future of Microprocessors”, *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.
10. S. Borkar, “Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation”, *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.
11. A. Leon, B. Langley, and J. Shin, “The UltraSPARC T1 processor: CMT reliability”, in *Proceedings of the Custom Integrated Circuits Conference*, pp. 555–562, 2006.
12. T. Austin, V. Bertacco, S. Mahlke, and Y. Cao, “Reliable Systems on Unreliable Fabrics”, *IEEE Transactions on Design & Test of Computers*, vol. 25, no. 4, pp. 322–332, 2008.
13. M. A. Alam, S. Mahapatra, “A comprehensive model for PMOS NBTI degradation”, *Microelectronics Reliability*, pp. 71–81, 2005.
14. S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De., “Parameter variations and impact on circuits and microarchitecture”, in *Proceedings of the 40th Annual Design Automation Conference (DAC)*, pp. 338–342, ACM, 2003.
15. K. Bowman, A. Alameldeen, S. Srinivasan, and C. Wilkerson. “Impact of die-to-die and within-die parameter variations on the clock frequency and throughput of multi-core processors”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 17, no. 12, pp. 1679–1690, 2009.

S. Rehman et al., *Reliable Software for Unreliable Hardware*,
DOI 10.1007/978-3-319-25772-3

16. R. Rajeev, A. Devgan, D. Blaauw, and D. Sylvester, “Parametric yield estimation considering leakage variability”, in *Proceedings of the 41st Annual Design Automation Conference (DAC)*, pp. 442–447, ACM, 2004.
17. D. Cheng and S. K. Gupta, “Maximizing yield per area of highly parallel cmps using hardware redundancy”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 33, no. 10, pp. 1545–1558, Oct 2014.
18. R. Baumann, “Radiation-induced soft errors in advanced semiconductor technologies”, *IEEE Transactions on Device and Materials Reliability,* vol. 5, no. 3, pp. 305–316, 2005.
19. J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M.Tahoori, and N.Wehn, “Reliable on-chip systems in the nano-era: Lessons learnt and future trends”, in *Proceedings of the 50th Annual Design Automation Conference (DAC)*, pp. 99, ACM, 2013.
20. “Firmware-based Platform Reliability”, Intel Corporation, 2004.
21. P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, “Modeling the effect of technology trends on the soft error rate of combinational logic”, in *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, pp. 389–398, 2002.
22. S. Mukherjee., J. Emer, and S. Reinhardt, “The soft error problem: An architectural perspective”, in *The 11th International Symposium on High-Performance Computer Architecture, 2005. HPCA-11*, pp. 243–247, 2005.
23. R. Lefurgy, A. Drake, M. Floyd, M. Allen-Ware, B. Brock, J. Tierno, and J. Carter, “Active management of timing guardband to save energy in POWER7”, in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–11, ACM, 2011.
24. M. Agarwal, B. C. Paul, M. Zhang, and S. Mitra, “Circuit Failure Prediction and Its Application to Transistor Aging”, in *Proceedings of the VLSI Test Symposium*, pp. 277–286, 2007.
25. B. Raghunathan, Y. Turakhia, S. Garg, and D. Marculescu, “Cherry-picking: exploiting process variations in dark-silicon homogeneous chip multi-processors”, in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pp. 39–44. EDA Consortium, 2013.
26. P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R. Gupta, R. Kumar, S. Mitra, A. Nicolau, T.Rosing, M. Srivastava, S. Swanson, and D. Sylvester, “Underdesigned and opportunistic computing in presence of hardware variability”, in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD),* vol. 32, no. 1, pp. 8–23, 2013.
27. N. Oh, P. Shirvani, and E. McCluskey, “Error detection by duplicated instructions in superscalar processors”, in *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.
28. G. Reis, J. Chang, N. Vachharajani, R. Rangan, D. August, and S. Mukherjee, “Software-controlled fault tolerance”, in *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 4, pp. 366–396, 2005.
29. S. Borkar, “Microarchitecture and design challenges for gigascale integration”, in *MICRO*, vol. 37, p. 3, 2004.
30. E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba, “Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule”, in *IEEE Transactions on Electron Devices*, vol. 57, no. 7, pp. 1527–1538, 2010.
31. S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor”, in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. pp. 29, 2003.
32. N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, V. Ambrose, Q. Shi, R. Allmon, and A. Bramnik, “Soft error susceptibilities of 22 nm tri-gate devices”, in *IEEE Transactions on Nuclear Science,* vol. 59, no. 6, pp. 2666–2673, 2012.
33. R. Vadlamani, J. Zhao, W. Burleson, and R. Tessier, “Multicore soft error rate stabilization using adaptive dual modular redundancy”, in *IEEE Design, Automation and Test in Europe Conference & Exhibition (DATE),*pp. 27–32, 2010.
34. N. Oh, P. Shirvani, and E. McCluskey, “Control-flow checking by software signatures”, in *IEEE Transactions on Reliability,* vol. 51, no. 1, pp. 111–122, 2002.

35. J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC v8 architecture", in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks(DSN)*, pp. 409–415, 2002.
36. S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives", in *Proceedings of the 29th Annual IEEE International Symposium on Computer Architecture (ISCA)*, pp. 99–110, 2002.
37. A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors, "PLR: A software approach to transient fault tolerance for multicore architectures", in *IEEE Transactions on Dependable and Secure Computing,* vol. 6, no. 2, pp. 135–148, 2009.
38. J. Smolens, B. Gold, B. Falsafi, and J. Hoe, "Reunion: Complexity-effective multicore redundancy", in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO),* IEEE Computer Society, pp. 223–234, 2006.
39. C. Constantinescu, "Trends and challenges in VLSI circuit reliability", in *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.
40. H. Kufluoglu and M. Alam, "A Generalized Reaction–Diffusion Model With Explicit H–Dynamics for Negative-Bias Temperature-Instability (NBTI) Degradation", in *IEEE Transactions on Electron Devices,* vol. 54, no. 5, pp. 1101–1107, 2007.
41. H. Hanson, K. Rajamani, J. Rubio, S. Ghiasi, and F. Rawson, "Benchmarking for Power and Performance", in *SPEC Benchmarking Workshop*, 2007.
42. S. Dighe, S. Vangal, P. Aseron, S. Kumar, T. Jacob, K. Bowman, J. Howard, J. Tschanz, V. Erraguntla, N. Borkar, V. De, and S. Borkar, "Within-die variation-aware dynamic-voltage-frequency scaling core mapping and thread hopping for an 80-core processor", in *IEEE International Solid-State Circuits Conference*, 2010.
43. L. Wanner, C. Apte, R. Balani, P. Gupta, and M. Srivastava, "Hardware variability-aware duty cycling for embedded sensors", in *IEEE Transactions on VLSI,* 2012.
44. J. Xiong, V. Zolotov, and L. He, "Robust extraction of spatial correlation", in *IEEE Transactions on Computer Aided Design (TCAD)*, vol. 26, no. 4, pp. 619–631, 2007.
45. S. Herbert and D. Marculescu, "Characterizing chip-multiprocessor variability-tolerance", in *IEEE Design and Automation Conference*, pp. 313–318, 2008.
46. P. Murley and G. Srinivasan, "Soft-error Monte Carlo modeling program, SEMM", in *IBM Journal of Research and Development*, vol. 40, no. 1, 1996.
47. M. Omana, G. Papasso, D. Rossi, and C. Metra, "A Model for Transient Fault Propagation in Combinatorial Logic", in *Proceedings of the 9th IEEE International On-Line Testing Symposium (IOLTS)*, pp. 11–115, 2003.
48. S. Krishnaswamy, G. F. Viamonte, I. L. Markov, and J. P. Hayes, "Accurate Reliability Evaluation and Enhancement via Probabilistic Transfer Matrices", in *Proceedings of Design, Automation and Test in Europe (DATE)*, pp. 282–287, 2005.
49. Y. Dhillon, A. Diril, and A. Chatterjee, "Soft-Error Tolerance Analysis and Optimization of Nanometer Circuits", in *Proceedings of Design, Automation and Test in Europe (DATE)*, pp. 288–293, 2005.
50. S. Kiamehr, M. Ebrahimi, F. Firouzi, and M. Tahoori, "Chip-level modeling and analysis of electrical masking of soft errors", in *The 31st IEEE VLSI Test Symposium (VTS),* pp. 1–6, 2013.
51. H. Asadi, and M. Tahoori, "An Accurate SER Estimation Method Based on Propagation Probability", in *Proceedings of Design, Automation and Test Conference in Europe (DATE)*, 2005.
52. M. Ebrahimi., L. Chen, H. Asadi, and M. Tahoori, "CLASS: Combined logic and architectural soft error sensitivity analysis", in *18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 601–607, 2013.
53. K. Itoh, R. Hori, H. Masuda, Y. Kamigaki, H. Kawamoto, and H. Katto, "A single 5V 64k dynamic ram", in *IEEE International Solid-State Circuits Conference (ISSCC)*, Digest of Technical Papers, vol. 23, pp 228–229, 1980.
54. M. Kohara, Y. Mashiko, K. Nakasaki, and M. Nunoshita, "Mechanism of electromigration in ceramic packages induced by chip-coating polyimide", in *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, vol. 13, no. 4, pp. 873–878, 1990.

55. M. Bruel, "Silicon on insulator material technology", in *Electronics Letters*, vol. 31, no. 14, pp. 1201–1202, 1995.
56. E. Cannon, D. Reinhardt, M. Gordon, and P. Makowenskyj, "Sram ser in 90, 130 and 180 nm bulk and soi technologies", in *Proceedings of 42nd Annual IEEE International Reliability Physics Symposium*, pp. 300–304, 2004.
57. D. Burnett, C. Lage, and A. Bormann, "Soft-error-rate improvement in advanced bicmos srams", in *Proceedings of 31st Annual Reliability Physics Symposium,* pp. 156–160, 1993.
58. S Mitra, T. Karnik, N. Seifert, and M. Zhang, "Logic soft errors in sub-65 nm technologies design and cad challenges", in *Proceedings of 42nd Design Automation Conference (DAC),* pp. 2–4, 2005.
59. D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. Kim, and K. Flautner, K, "Razor: circuit-level correction of timing errors for low-power operation", in *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.
60. S. Das, C. Tokunaga, S. Pant, M. Wei-Hsiang, S. Kalaiselvan, K. Lai, D. Bull, and D. Blaauw, "RazorII: In situ error detection and correction for PVT and SER tolerance", in *IEEE Journal of Solid-State Circuits,* vol. 44, no. 1, pp. 32–48, 2009.
61. H. Wunderlich and M. Tahoori, "Tutorial Workshop in the frame of the DFG SPP 1500: Defects, Faults, and Errors - Approaches to Cross-Layer Fault-Tolerance", in *GMM/GI/ITG-Fachtagung Zuverlässigkeit und Entwurf (ZuE)*, 2011.
62. IBM® XIV® Storage System cache: http://publib.boulder.ibm.com/infocenter/ibmxiv/r2/index.jsp [Online; accessed Apr. 2015].
63. AMD Phenom™ II Processor Product Data Sheet 2010.
64. R. Hamming, "Error detecting and error correcting codes", in *Bell System Technical Journal*, vol. 26, no. 2, pp. 147–160, 1950.
65. K. Kang, S. Gangwal, S. Park, and A. Roy, "NBTI Induced Performance Degradation in Logic and 66. Memory Circuits", in *Proceedings of the Asia and South Pacific Design Automation Conference* (*ASPDAC)*, 2008.
66. Aeroflex, http://aeroflex.com/ams/ [Online; accessed Apr 2015].
67. S. Reinhardt and S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading", in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 25–34, 2000.
68. D. Tullsen, S. Eggers, and H. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism", in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2, pp. 392–403, ACM, 1995.
69. A. Avizienis, "The N-version approach to fault-tolerant software", in *IEEE Transactions on. Software Engineering*, vol. 11, no. 12, pp. 1491–1501, 1985.
70. R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems", in *IEEE Transactions on Software Engineering*, vol. 1, pp. 23–31, 1987.
71. G. Reis, "*Software modulated fault tolerance*", Ph.D. Thesis, Princeton University, 2008.
72. J. Lee and A.Shrivastava, "A compiler optimization to reduce soft errors in register files", in *ACM Sigplan Notices*, vol. 44, no. 7, pp. 41–49, ACM, 2009.
73. J. Yan and W. Zhang, "Compiler-guided register reliability improvement against soft errors", in *Proceedings of the 5th ACM International Conference on Embedded Software*, pp. 203–209, 2005.
74. V. Sridharan, "Introducing Abstraction to Vulnerability Analysis", *Ph.D. Thesis*, March 2010.
75. V. Sridharan and D. Kaeli, "Eliminating Micro-architectural Dependency from Architectural Vulnerability", in *IEEE International Symposium on High Performance Computer Architecture*, pp. 117–128, 2009.
76. D. Borodin and B. Juurlink, "Protective redundancy overhead reduction using instruction vulnerability factor", in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, pp. 319–326, 2010.
77. J. Hu, S. Wang, and G. Ziavras, "In-register duplication: Exploiting narrow-width value for improving register file reliability", in *IEEE International Conference on Dependable Systems and Networks (DSN 2006)*, pp. 281–290, 2006.

78. P. Lokuciejewski and P. Marwedel, “Combining worst-case timing models, loop unrolling, and static loop analysis for WCET minimization”, in *21st IEEE Euromicro Conference on Real-Time Systems (ECRTS),* pp. 35–44, 2009.
79. V. Sarkar, “Optimized Unrolling of Nested Loops”, in *International Journal on Parallel Programing,* vol. 29, no. 5, pp. 545–581, 2001.
80. J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. Irwin, “Compiler-directed instruction duplication for soft error detection”, in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE),* pp. 1056–1057, 2005.
81. J. Xu, Q. Tan, and R. Shen, “The Instruction Scheduling for Soft Errors based on Data Flow Analysis”, in *IEEE Pacific Rim International Symposium on Dependable Computing*, pp. 372–378, 2009.
82. L. Spainhower and T. Gregg, “IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective”, in *IBM journal of Research and Development*, vol. 43, no. 5/6, 1999.
83. T. Li, M. Shafique, S. Rehman, J. A. Ambrose, J. Henkel, and S. Parameswaran, “DHASER: Dynamic Heterogeneous Adaptation for Soft-Error Resiliency in ASIP-based Multi-core Systems”, in *IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 646–653, 2013.
84. J. Maiz, S. Hareland, K. Zhang, and P. Armstrong, “Characterization of multi-bit soft error events in advanced SRAMs”, in *Electron Devices Meeting* (IEDM), pp. 21.4.1–21.4.4, 2003.
85. K. Osada, K. Yamaguchi, Y. Saitoh, and T. Kawahara, “SRAM immunity to cosmic-ray-induced multierrors based on analysis of an induced parasitic bipolar effect”, in *IEEE Journal of Solid-State Circuits*, vol. 39, no. 5, pp. 827–833,2004.
86. J.-M. Palau, G. Hubert, K. Coulie, B. Sagnes, M.-C. Calvet, and S. Fourtine, “Device simulation study of the seu sensitivity of srams to internal ion tracks generated by nuclear reactions”, in *IEEE Transactions on Nuclear Science*, vol. 48, no. 2, pp. 225–231, 2001.
87. N. Miskov-Zivanov and D. Marculescu, “Circuit reliability analysis using symbolic techniques”, in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 12, pp. 2638–2649, 2006.
88. M. Zhang and N. Shanbhag, “A Soft Error rate Analysis (SERA) Methodology”, in *Proceedings of ACM/IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 111–118, 2004.
89. N. George, C. Elks, B. Johnson, and J. Lach, “Transient fault models and AVF estimation revisited”, in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN),*pp. 477–486, 2010.
90. A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan, “Computing architectural vulnerability factors for address-based structures”, in *Proceedings of the 32*nd *Annual International Symposium on Computer Architecture (ISCA)*, pp. 532–543, 2005.
91. N. Wang, J. Quek, T. Rafacz, and S. Patel, “Characterizing the effects of transient faults on a high-performance processor pipeline”, in *IEEE International Conference on Dependable Systems and Networks (DSN),* pp. 61–70, 2004.
92. R. Venkatasubramanian, J. Hayes, and B. Murray, “Low cost online fault detection using control flow assertions”, in *Proceedings of 9th IEEE On-Line Test. Symposium (IOLTS)*, pp. 137–143, 2003.
93. P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson, “On latching probability of particle induced transients in combinational networks”, in *Proceedings of Fault-Tolerant Computing Symposium*, pp. 340–349, 1994.
94. J. Ziegler, H. Curtis, H. Muhlfeld, J. Montrose, and B. Chin, “IBM experiments in soft fails in computer electronics (1978–1994)”, in *IBM journal of research and development*, vol. 40, no. 1, pp. 3–18, 1996.
95. L. Chen, M. Ebrahimi, and M. Tahoori, “CEP: Correlated Error Propagation for Hierarchical Soft Error Analysis”, in *Journal of Electronic Testing: Theory and Applications (JETTA)*, Springer, 2013.

96. H. Ziade, R. Ayoubi, and R. Velazco, "A survey on fault injection techniques", in *International Arab Journal of Information Technology*, vol. 1, no. 2, pp. 171–186, 2004.
97. V. Chippa, D. Mohapatra, A. Raghunathan, K.Roy, and S. Chakradhar, "Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency", in *Proceedings of the ACM 47th Design Automation Conference (DAC)*, pp. 555–560, 2010.
98. K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "SymPLFIED: Symbolic program-level fault injection and error detection framework", in *IEEE International Conference on Dependable Systems and Networks (DSN)*, pp. 472–481, 2008.
99. R. Velazco, A. Corominas, and P. Ferreyra, "Injecting bit flip faults by means of a purely software approach: a case studied", in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 108–116, 2002.
100. J. Coppens, D. Al-Khalili, and C. Rozon, "VHDL Modelling and Analysis of Fault Secure Systems", in *Proceedings of the IEEE Conference on Design Automation and Test in Europe (DATE),* pp. 148–152, 1998.
101. R. Shafik, P. Rosinger, and B. Al-Hashimi, "SystemC-Based Minimum Intrusive Fault Injection Technique with Improved Fault Representation", in *IEEE International On-Line Testing Symposium (IOLTS)*, pp. 99–104, 2008.
102. P. Simonen, A. Heinonen, M. Kuulusa, and J. Nurmi, "Comparison of bulk and SOI CMOS Technologies in a DSP Processor Circuit Implementation", in *Proceedings of the 13th International Conference on Microelectronics (ICM)*, pp. 107–110, 2001.
103. J. Yao, S. Okada, M. Masuda, K. Kobayashi, and Y. Nakashima, "DARA: A low-cost reliable architecture based on unhardened devices and its case study of radiation stress test", in *IEEE Transactions on Nuclear Science*, vol. 59, no. 6, pp. 2852–2858, 2012.
104. C. Weaver and T. Austin, "A fault tolerant approach to microprocessor design", in *IEEE International Conference on Dependable Systems and Networks (DSN)*, pp. 411–420, 2001.
105. G. Messenger, "Collection of Charge on Junction Nodes from Ion Tracks", in *IEEE Transactions on Nuclear Science*, vol. 29, no. 6, pp. 2024–2031, 1982.
106. P. Dodd and F. Sexton, "Critical charge concepts for CMOS SRAMs", in *IEEE Transactions on Nuclear Science,* vol. 42, no. 6, pp. 1764–1771, 1995.
107. J. Henkel, L. Bauer, H. Zhang, S. Rehman, and M. Shafique, "Multi-Layer Dependability: From Microarchitecture to Application Level", in *ACM/IEEE/EDA 51st Design Automation Conference (DAC), 2014.*
108. C. Nguyen and G. R. Redinbo, "Fault tolerance design in JPEG 2000 image compression system", *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 1, pp. 57–75, 2005.
109. M. A. Makhzan, A. Khajeh, A. Eltawil, and F. J. Kurdahi, "A low power JPEG2000 encoder with iterative and fault tolerant error concealment", *IEEE Transaction on Very Large Scale Integration (TVLSI)*, vol. 17, no. 6, pp. 827–837, 2009.
110. A. K. Djahromi, A. Eltawil, and F. J. Kurdahi, "Exploiting fault tolerance towards power efficient wireless multimedia applications", in *IEEE Consumer communications and networking conference*, pp. 400–404, 2007.
111. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", in *IEEE 4th Annual Workshop on Workload Characterization*, 2001.
112. S. Rehman, F. Kriebel, M. Shafique, and J. Henkel, "Reliability-Driven Software Transformations for Unreliable Hardware", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Volume 33, Issue 11, pp. 1597–1610, 2014
113. V. Kleeberger, C. G. Dumont, C. Weis, A. Herkersdorf, D. M. Gritschneder, S. R. Nassif, U. Schlichtmann, and N. Wehn, "A Cross-Layer Technology-Based Study of How Memory Errors Impact System Resilience", *IEEE Micro*, vol. 33, no. 4, pp. 46–55, 2013.
114. S. Sinha, G. Yeric, V. Chandra, B. Cline, and Y. Cao, "Exploring sub-20 nm FinFET design with Predictive Technology Models", in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 283–288, 2012.

115. F. Kriebel, S. Rehman, D. Sun, P. V. Aceituno, M. Shafique, and J. Henkel, “ACSEM: Accuracy-Configurable Fast Soft Error Masking Analysis in Combinatorial Circuits”, in *IEEE/ACM 18th Design, Automation and Test in Europe Conference (DATE)*, March 2015.
116. F. Oboril, “Cross-Layer Approaches for an Aging-Aware Design of Nanoscale Microprocessors”, *Ph.D. Thesis*, 2015.
117. H. Amrouch, V. M. van Santen, T. Ebi, V. Wenzel, and J. Henkel, “Towards interdependencies of aging mechanisms”, in *IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 478–485, 2014.
118. DFG SPP1500 Program on Dependable Embedded Systems: http://spp1500.itec.kit.edu/.
119. R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. C. de Araujo, and E. Barros, “The ArchC Architecture Description Language and Tools”, *International Journal of Parallel Programming*, vol. 33, no. 5, pp. 453–484, 2005.
120. C.-C. Han, K. G. Shin, and J. Wu, “A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults”, *IEEE Transactions on Computers (TC)*, vol. 52, no. 3, pp. 362–372, 2003.
121. T. Ball and J. R. Larus, “Branch Prediction for Free”, *ACM SIGPLAN*, vol. 28, pp. 300–313, 1993.
122. Synopsys, “Synopsys Design Compiler User Guide” [Online]. *Available:* https://solvnet.synopsys.com/dow_retrieve/latest/dcug/dcug.html.
123. Synopsys, “Accelerate Design Innovation with Design Compiler®”, Synopsys [Online]. *Available:* http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/default.aspx.
124. ModelSim, “ModelSim—Leading Simulation and Debugging”, Mentor [Online]. *Available:* http://www.mentor.com/products/fpga/model.
125. Synopsys, “TSMC 45 nm High Speed Tapless Standard Cell Logic Library”, TSMC [Online]. *Available:* http://www.synopsys.com/dw/ipdir.php?c=dwc_logic_ts45nkkslogcassst000f.
126. Flux calculator: www.seutest.com/cgi-bin/FluxCalculator.cgi.
127. http://www.archc.org; The ArchC Website.
128. http://downloads.sourceforge.net/archc/ac_lrm-v2.0.pdf; The ArchC Architecture Description Language v2.0 Reference Manual.
129. GCC: https://gcc.gnu.org/.
130. M. Shafique, L. Bauer, and J. Henkel, “Optimizing the H.264/AVC Video Encoder Application Structure for Reconfigurable and Application-Specific Platforms”, *Journal of Signal Processing Systems (JSPS)*, vol. 60, no. 2, pp. 183–210, 2010.
131. *Haifa Scheduler:* http://gcc.gnu.org/, http://opensource.apple.com/ source/gcc_os/gcc_os-1660/gcc/haifa-sched.c.
132. A. Parikh, S. Kim, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, “Instruction scheduling for low-power”, *Journal of VLSI Signal Processing systems*, vol 37, no. 1, pp. 129–149, 2004.
133. J. Yan and W. Zhang, “Compiler guided register reliability improvement against soft errors”, in *IEEE International Conference on Embedded Software (EMSOFT)*, pp. 203–209, 2005.
134. J. Cong and K. Gururaj, “Assuring Application-Level Correctness Against Soft Errors”, in *IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 150–157, 2011.
135. R. Baumann, “Soft errors in advanced computer systems”, in *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.
136. R. Heald, “How cosmic rays cause computer downtime”, in *IEEE Reliability Society Meeting (SCV),* pp. 15–21, 2005.
137. K. Kang, S. Gangwal, S. Park, and K. Roy, “NBTI induced performance degradation in logic and memory circuits: how effectively can we approach a reliability solution?”, in *Proceedings of Asia and South Pacific Design Automation Conference*, pp. 726–731, 2008.
138. M. Shafique, M. U. K. Khan, O. Tuefek, and J. Henkel, “EnAAM: Energy-Efficient Anti-Aging for On-Chip Video Memories”, in *ACM/EDAC/IEEE 52nd Design Automation Conference*, San Francisco, CA/USA, June 8–12, 2015.

139. S. Herbert, S. Garg, and D. Marculescu, “Exploiting process variability in voltage/frequency Control”, *IEEE Transactions Very Large Scale Integration (VLSI) Systems*, on 20, no. 8, pp. 1392–1404, 2012.
140. T. Li, R. Ragel, and S. Parameswaran, “Reli: Hardware/software Checkpoint and Recovery scheme for embedded processors”, in *IEEE Design, Automation & Test in Europe Conference & Exhibition,* pp. 875–880, 2012.
141. M. Demertzi, M. Annavaram, and M. Hall, “Analyzing the effects of compiler optimizations on application reliability”, *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 184–193, 2011.
142. S. Rehman, A. Toma, F. Kriebel, M. Shafique, J.-J. Chen, and J. Henkel, “Reliable Code Generation and Execution on Unreliable Hardware under Joint Functional and Timing Reliability Considerations”, in: *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 273–282, 2013.
143. J. B. Velamala, K. Sutaria, T. Sato, and Y. Cao, “Physics matters: statistical aging prediction under trapping/detrapping”, in *49th IEEE/ACM Annual Design Automation Conference (DAC)*, pp. 139–144, 2012.
144. K. Kuhn, C. Kenyon, A. Kornfeld, M. Liu, A. Maheshwari, W. Shih, S. Sivakumar, G. Taylor, P. VanDerVoorn, and K. Zawadzki, “Managing Process Variation in Intel’s 45 nm CMOS Technology”, in *Intel Technology Journal*, vol. 12, no. 2, 2008.
145. C. Li and W. Fuchs, “Catch-compiler-assisted techniques for checkpointing”, in *20th International Symposium of Fault-Tolerant Computing (FTCS-20)*, Digest of Papers, pp. 74–81, 1990.
146. J. Plank, M. Beck, G. Kingsley, and K. Li, “Libckpt: Transparent Checkpointing under Unix”, in *Proceedings of Usenix Technical Conference*, pp. 213–223, 1995.
147. Y. Huang and C. Kintala, “Software implemented fault tolerance: Technologies and experience”, in *Proceedings of the IEEE Fault-Tolerant Computing Symposium (FTCS)*, vol. 23, pp. 2–9, 1993.
148. L. Wang, Z. Kalbarczyk, W. Gu, and R. Iyer, “An OS-level framework for providing application-aware reliability”, in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 55–62, 2006.
149. T. Ebi, M. A. Al Faruque, and J. Henkel, “TAPE: Thermal-aware agent-based power econom multi/many-core architectures”, in *IEEE International Conference on Computer Aided Design (ICCAD),* pp. 302–309, 2009.
150. H. Khdr, T. Ebi, M. Shafique, H. Amrouch, and J. Henkel, “mDTM: Multi-Objective Dynamic Thermal Management for On-Chip Systems”, in *IEEE/ACM 17th Design Automation and Test in Europe Conference (DATE)*, 2014.
151. J. Henkel, T. Ebi, H. Amrouch, and H. Khdr, “Thermal management for dependable on-chip systems”, in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 113–118, 2013.
152. H. Amrouch, T. Ebi, and J. Henkel, “RESI: Register-Embedded Self-Immunity for Reliability Enhancement”, *IEEE Transactions on CAD of Integrated Circuits and Systems (TCAD)*, vol. 33, no. 5, pp. 677–690, 2014.
153. L. Bauer, C. Braun, M. E. Imhof, M. A. Kochte, E. Schneider, H. Zhang, J. Henkel, and H.-J. Wunderlich, “Test Strategies for Reliable Runtime Reconfigurable Architectures”, in *IEEE Transactions on Computers (TC)*, vol. 62, no. 8, pp. 1494–1507, 2013.
154. H. Zhang, M. A. Kochte, M. E. Imhof, L. Bauer, H.-J. Wunderlich, and J. Henkel, “GUARD: GUAranteed Reliability in Dynamically Reconfigurable Systems”, in *IEEE/ACM Design Automation Conference (DAC)*, pp. 32:1–32:6, 2014.
155. D. Gnad, M. Shafique, F. Kriebel, S. Rehman, D. Sun, and J. Henkel, “Hayat: Harnessing Dark Silicon and Variability for Aging Deceleration and Balancing”, in *ACM/EDAC/IEEE 52nd Design Automation Conference (DAC)*, 2015.
156. J. Henkel, L. Bauer, J. Becker, O. Bringmann, U. Brinkschulte, S. Chakraborty, M. Engel, R. Ernst, H. Härtig, L. Hedrich, A. Herkersdorf, R. Kapitza, D. Lohmann, P. Marwedel, M. Platzner, W. Rosenstiel, U. Schlichtmann, O. Spinczyk, M. B. Tahoori, J. Teich, N. Wehn, and H. J. Wunderlich, “Design and architectures for dependable embedded systems”, in *IEEE*

International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS), pp. 69–78, 2011.

157. J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R.K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe, "Invasive Manycore Architectures", in *17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 193–200, 2012.
158. J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting, "Invasive Computing: An Overview", in *Multiprocessor System-on-Chip—Hardware Design and Tool Integration*, M. Hübner and J. Becker (Eds.), pp. 241–268, Springer, 2011.