CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Dynamic fault generator for simulating gracefully degradable components

Master's thesis in Embedded Electronic System Design

TONY AKIKI

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY AND GOTHENBURG UNIVERSITY
Gothenburg, Sweden 2016

CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Dynamic fault generator for simulating gracefully degradable components

A tool that generates events to predict and inject permanent faults that could appear on a gracefully degradable adaptive system during its life time

TONY AKIKI

Department of Computer Science and Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden 2016

Dynamic fault generator for simulating gracefully degradable components
A tool that generates events to predict and inject permanent faults that will appear on a gracefully degradable adaptive system during its life time
TONY AKIKI

Supervisors: Prof.Dr. Ioannis Sourdis & PhD candidate Stavros Tzilis, Computer Science and Engineering Department, Chalmers University of Technology
Examiner: Prof.Dr. Per Larsson-Edefors, Computer Science and Engineering Department, Chalmers University of Technology

Department of Computer Science and Engineering
Chalmers University of Technology and Gothenburg University
SE-412 96 Gothenburg
Telephone +46 (0)31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2016

Dynamic fault generator for simulating gracefully degradable components
A tool that generates events to predict and inject the permanent faults that will appear on a gracefully degradable adaptive system during its life time
TONY AKIKI
Department of Computer Science and Engineering
Chalmers University of Technology and Gothenburg University

Abstract

Transistors dimensions are scaling down according to Moore's Law making integrated circuits much prone to failures. To help study the reliability of such complex systems, the aim of this thesis is to develop a high-level tool that inject faults dynamically on a gracefully degradable adaptive system during its lifetime. Our developed tool operates in closed-loop with the rest of the experimental setup, receiving feedback parameters that have an influence on the prediction routine, such as the components' utilization rates. Additionally, the tool is able to consider transistor wearout effects in the fault prediction routine, such as Negative Bias Temperature Instability (NBTI).

Furthermore, we were able to examine different degradable aspects of complex systems using different fault scenarios. For instance we studied the capacity degradation over time of the 3-D stacked DRAM Hybrid Memory Cube (HMC) and found that the remaining memory capacity after 10 years is around 86% considering normal failure rates and around 77% considering pessimistic failure rates. We also studied the degradation of a reconfigurable processor array in terms of number of operating processors over time for array sizes 8-by-7 and 4-by-8, and determined the best array size given the failure rate and expected mission time.

Keywords: Reliability, runtime management, permanent faults, transistor aging, fault modelling, graceful degradation, reconfigurablity, lifetime extension.

Contents

List of Figures

List of Tables

1 Introduction

The complexity of embedded systems has grown very fast in the last decades. As the transistor dimensions are shrinking, the defect densities of system components increase and conventional testing becomes harder and more expensive. Scaling these complex systems makes them sensitive, and one of the greater challenges is the reliability of a system built with different hardware components, which are furthermore prone to aging effects. To help study the reliability of such systems, this master thesis aims to develop a tool that injects permanent faults that will appear on a gracefully degradable adaptive system during its lifetime.

1.1 Motivation

The motivation of this master thesis comes from existing works on lifetime estimation of multicore systems, lifetime management, lifetime extension and graceful degradation of multiprocessor system-on-chip (MPSoC) components. In general, researching the effect of permanent faults on modern adaptive MPSoCs requires flexible and consistent methodologies for predicting (or injecting) these faults. Previous studies proposed runtime optimization algorithms for the management of a gracefully degradable adaptive MPSoC [1][2][4][5][6][8][10]. In this context, an event generator (EG) was developed in [2] to inject the permanent faults that will appear on the MPSoC during its lifetime, based on a set of parameters (mainly the fault rate of each component).

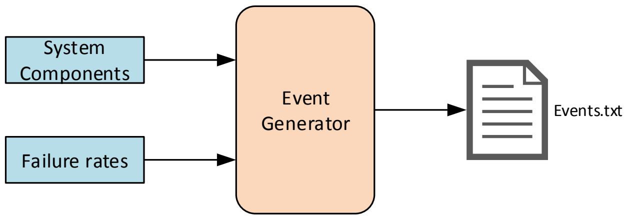


Figure 1.1: Basic event generator

This EG shown in Figure 1.1 is a basic tool that predicts events which are injected afterwards in the experimental setup. This is a good baseline to build on since the prediction algorithm is designed and implemented. However, it works in open-loop which is not realistic enough because it injects in advance a set of events without taking into account system parameters about the system state or configuration which can vary over time, either because of workload fluctuations or the impact of runtime decisions or the non-uniform, over time, impact of aging effects.

1.2 Problem Statement

Our goal is to improve the EG by building a dynamic event generator (DEG) that works in closed-loop with the reconfiguration module as shown in Figure 1.2 taking into consideration several indications that will be discussed later in this thesis. This solution is aimed to be more realistic, introducing to the system parameters such as Negative-Bias Temperature Instability (NBTI), and being aware of each component use (utilization rate or duty cycle) through the system reconfiguration module. These parameters serve as a feedback. Whenever system parameters change, e.g. after applying a new system configuration, the DEG inputs will be updated accordingly, affecting the prediction of the next permanent fault.

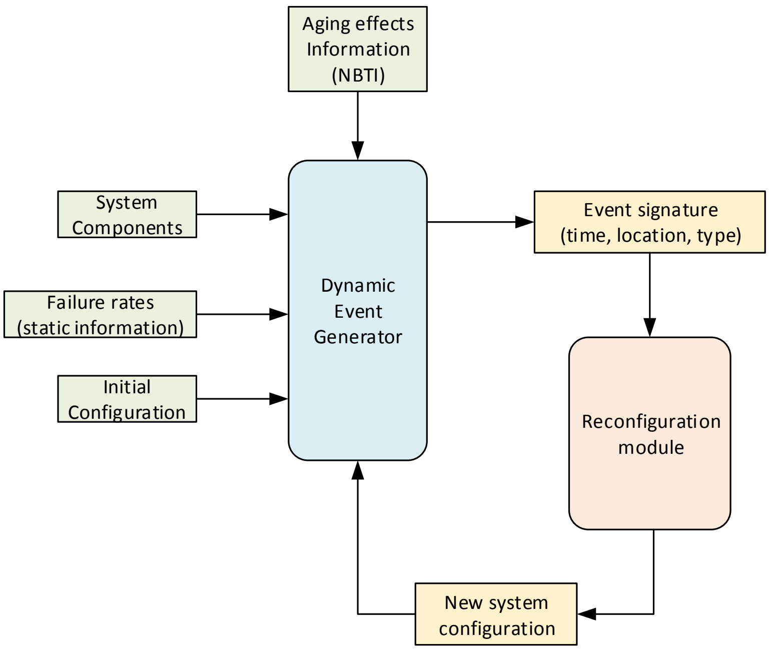


Figure 1.2: Dynamic event generator block diagram

The following subsection will list the main objectives of the thesis based on the above problem statement.

1.2.1 Thesis Objectives

The purpose of the project is to design an advanced tool taking into consideration these main objectives:

- Upgrade the basic event generator (EG) to work in closed-loop with the rest of the experimental setup, that is, a dynamic event generator (DEG), and make use of information provided by the system configuration. The predicted permanent faults are based on parameters that change during the system's lifetime such as the duty cycles of the components.
- Come up with more sophisticated fault models for various components of an adaptive, degradable system, like reconfigurable processors, processors that support frequency scaling and 3D-stacked DRAM Hybrid Memory Cube (HMC). For each of the examined components, a set of possible permanent faults are defined and a list of degraded modes of operation are determined based on these possible faults.
- Explore the NBTI aging effect and incorporate its properties as input parameters to the DEG. Further improve the previously implemented "dynamic" failure rates to account for the increased accuracy of input data when predicting the occurrence of permanent faults.
- Test extensively the developed DEG by verifying it against conventional reliability models such as a component with an exponential distribution as described in [9], against experiments concerning system lifetime for gracefully degradable components performed with the old EG [1], and against a low-level NBTI simulation.
- Use the developed DEG tool to evaluate and analyze the degradation of complex systems over time, such as the HMC and reconfigurable processor array, at different failure rates and design decisions.

These objectives will be brought up to discussion in Chapter 5 and Chapter 6. At that point we will discuss the results and elaborate on the achievements.

1.3 Limitations

The DEG is flexible to be used with any kind of fault model. We will be limited to a well defined set of fault models for various components of an adaptive, degradable system. These components are presented in the list below.

- Component with homogeneous modules
- Component supporting frequency scaling
- Memory with a controller able to isolate regions of the memory whenever a permanent fault occurs
- Heterogeneous component
- Reconfigurable m-by-n processor array
- 3D-stacked DRAM Hybrid Memory Cubes

The tool's infrastructure will allow modeling of other components not in the list with minimal effort, as long as their reliability model (faults which can occur and degraded modes of the component) is known.

One aging effect which will be integrated in the DEG is the aforementioned NBTI. Other aging effects could be considered like the Hot Carrier Injection (HCI) or Time-Dependent Dielectric Breakdown (TDDB) for future upgrade of the tool faults prediction routine.

The DEG operates in closed-loop with the experimental setup discussed in [1]. As the name indicates, this event generator works dynamically and the system configuration is updated at a specific frequency. We will limit the prediction precision of this tool by calling a new system configuration whenever an event occurs. A higher accuracy in the events prediction could be achieved if a new system configuration is called whenever a relevant parameter is modified, e.g. the workload of the system components. But this will not be considered in this thesis.

Lastly, our DEG will be able to inject faults in discrete time intervals of the system lifetime. However, depending on the accuracy requirements of each experiment, this interval can be defined to be arbitrarily short.

1.4 Thesis Outline

In this Section we briefly present the organization for the rest of the report.

Chapter 2 includes the theoretical background required to understand the contents of the thesis. We will highlight the traditional reliability analysis from literature. Thereafter, we will discuss relevant works related to the lifetime extension strategies of reliable systems, and how to design reliable systems from unreliable components. Furthermore, we provide a definition of graceful degradation and we finish up by presenting information about the aforementioned NBTI aging effect.

Chapter 3 presents the proposed graceful degradation models for various degradable components. For each of these components a brief description will be given and a list of possible events (permanent faults) that can appear on that component will be discussed. Beyond that, a Markov chain will show the degraded modes and how faults lead to them, and what are the degraded aspects of the system in each degraded mode.

Chapter 4 explains and justifies the main decisions throughout the design and implementation phases. With a good theoretical background from Chapter 2 and well defined models from Chapter 3, a good platform is sufficient to build on our design. We will present the theoretical construction of the event prediction based on the fault rate λ. In addition, we will explain how components and their various possible faults are modelled in the tool.

Chapter 5 discusses the experimental results and evaluates them. The DEG main functionality will be verified against conventional reliability models and experiments done with the basic EG [1]. The results of the new experiments produced by the DEG will be studied comparatively with the old ones using the basic EG and useful conclusions will be drawn. Likewise, the NBTI module will be calibrated using low-level NBTI experiments. Furthermore, the developed DEG will be used to evaluate different system degradations.

Finally, Chapter 6 summarizes and concludes the thesis. We will begin by recapitulating what is included in each chapter. Subsequently, we will summarize the thesis contributions and revisit the thesis objectives defined in Section 1.2.1 and state our achievements. Finally, we will draw conclusions from the work and list possible future prospects and suggest how the tool could be upgraded.

2
Background

This chapter provides the reader with a broad theoretical background, enough to understand the work done in this thesis. Section 2.1 summarizes traditional reliability analysis, on which the event generator (EG) is based. Section 2.2 discusses recently developed lifetime extension strategies to substantiate the claim that self-awareness is a useful property for modern adaptive systems. Section 2.3 gives a definition of "graceful degradation" and explains how standalone components can tolerate the occurrence of permanent faults and keep functioning. Lastly, Section 2.4 explores the Negative-Bias Temperature Instability (NBTI) aging effect and defines a way to incorporate its properties to the dynamic event generator (DEG).

2.1 Reliability Analysis

When systems are critical during mission time, it is required to perform some analysis to attain deeper knowledge of their survival properties. Fortunately, reliability analysis has been well established [9], and in this section we will review the basics needed to understand the rest of this report. The reliability $R(t)$ of a component is defined as the probability that its lifetime X is greater than t. In other terms, the reliability is the probability of a component to be functional during a specified time interval of duration equal to t. In fact, the reliability $R(t)$ of a hardware (HW) component has an exponential distribution function with a failure rate λ as parameter, and is represented in Equation 2.1.

$$R(t) = P(X > t) = e^{-\lambda t} \tag{2.1}$$

Accordingly, the probability of failure $F(t)$ of a HW component until time t can be expressed by Equation 2.2, and it is known as the distribution function.

$$F(t) = P(X \leq t) = 1 - R(t) = 1 - e^{-\lambda t} \tag{2.2}$$

In reality, a hardware component has a higher failure rate in the infant and wear-out phases as shown in the Bathtub curve in Figure 2.1. In this work we will consider the useful life period, and assume that the failure rate is constant as it is demonstrated in [9] for a HW component with exponential distribution function.

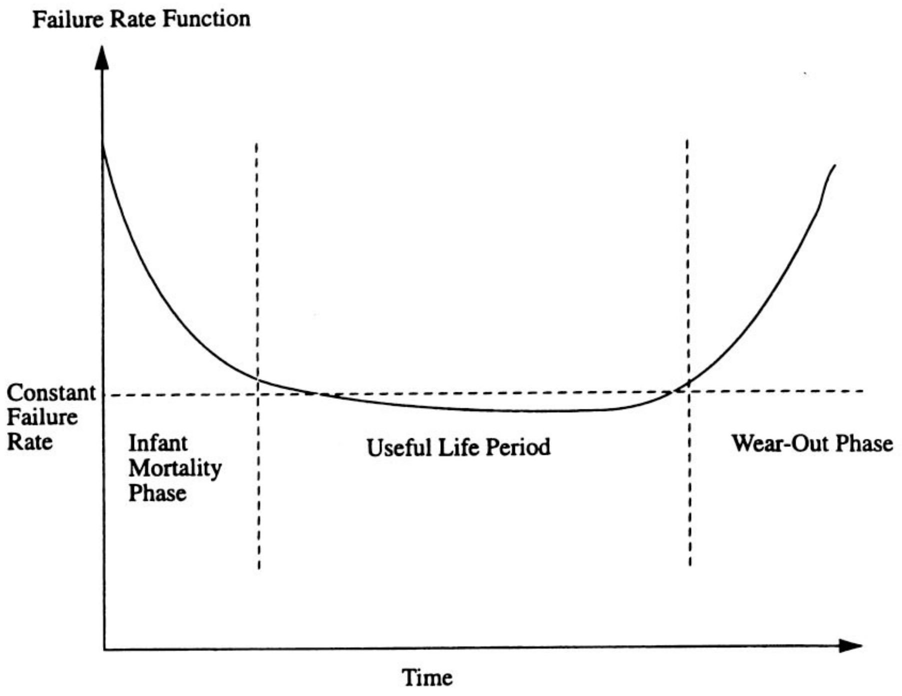


Figure 2.1: Illustration of the bathtub curve [29]

The DEG's main purpose is to generate and inject permanent faults that will appear on a gracefully degradable adaptive system during its life time. We based our approach on the basic event generator [1], according to which the probability that a fault occurs in an interval Δt is:

$$P(fault\ in\ \Delta t) = \int_{t}^{t+\Delta t} \lambda(S) dS \tag{2.3}$$

However, if $\lambda(t)$ is constant :

$$P(fault\ in\ \Delta t) = \lambda . \Delta t \tag{2.4}$$

Alongside with any functional unit, it is impossible to design a component without a single point of failure. Generally, these single points of failure are :

- the control signals which make up very low percentage of the total circuit, but their failure produces critical faults,
- the infrastructural circuitry, such as the clock tree,
- and some part of any protected circuit, such as the voter of a triple modular redundancy (TMR) system.

Therefore, in the implementation part we must take that into consideration and generate such permanent faults with fatal criticality that cause complete failure. On the other hand, we will introduce permanent faults with lower criticality related to the functional units. In case these faults happen, they degrade different aspects of the component and place it in degraded mode of operation. These levels of criticality differ by the fault rate λ of their corresponding generators.

The DEG uses quantization of time to predict if a permanent fault has occurred at time t when the non-decreasing function $\int_0^t \lambda(S)dS$ reaches a certain precalculated threshold E [1]. The threshold E in the fault prediction is an exponentially distributed number that is obtained by generating a uniformly distributed random number $y \in [0,1]$ and plugging it in $-ln(1-y)$. Observe that the threshold E is a number independent of λ. The failure rate λ will decide how fast we will reach this threshold E. The higher the failure rate λ, the faster the fault prediction system will attain the threshold E and generate a permanent fault in the corresponding time-stamp t. Furthermore, the DEG will be more realistic by varying the fault rate according to dynamically changing conditions, such as the component temperature and utilization rate. Furthermore, we will integrate an important aging effect, NBTI, that will be discussed in detail in Section 2.4. These factors will be added as parameters of $\lambda(t)$ and will certainly affect the fault prediction routine.

2.2 Lifetime Extension Strategies

As the transistor dimensions are being scaled down and the number of transistors on a die is increasing exponentially, the defect densities of system components also increase. Indeed, the oxide wearout effects tend to reduce the lifetime of the component due to high operating temperature [14] and the big number of transistors on a simple chip increases the power density with risk of creating thermal hot spots that lead to less reliable components [28]. Furthermore, with the presence of process variation, neighboring transistors behave differently as they might have different threshold voltages due to the random number of dopant atoms implanted [19]. Undoubtedly, the new challenge is to build reliable systems from components with delays that are non-uniformly distributed, both in space and in time due to the increase of leakage current.

Many strategies to extend the lifetime of a chip multiprocessor (CMP) has been discussed in literature. Most researchers examine the dynamic thermal (DTM) and reliability management (DRM) techniques to improve the performance while expecting lifetime extension. Some of these techniques have been discussed widely [14]. In contrast, the same article [14] considers a proactive approach to reliability named "Maestro", relying on low level sensors to manage and control the aging of the CMP. Maestro schedules dynamically the jobs to be done on the CMP in such a way preventing the weakest module to execute stressful workload. On one hand, this method avoids permanent failures induced by a single weak core. On the other hand, the same strategy ensures lifetime extension by improving the ability of the CMP to execute heavy workloads in the presence of aging effects. Other potential strategies and solutions have been discussed in [15] to improve the reliability of systems from unreliable components. One of these solutions is to include the test functionality as a part of the hardware to detect dynamically errors and isolate faulty components. These solutions must work together to substantiate the claim that modern systems should take care of themselves and extend their lifetime.

The above strategies try to ensure that CMPs degrade as gracefully as possible, a concept which will be defined in the following Subsection 2.3.

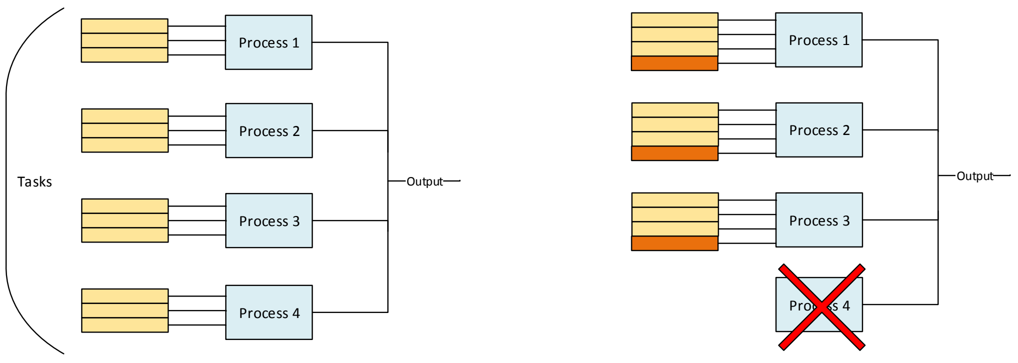
2.3 Graceful Degradation

Graceful degradation (henceforth GD) is the transition to a lower state of some system aspect as a response to the occurrence of an event that prohibits the manifestation of the fully fledged system behavior [16].

GD has been explored with the help of a runtime manager of an adaptive MPSoCs [1][3] in order to extend the lifetime of the system and prevent a total failure. Indeed, several solutions have been discussed, and the proposed algorithms implementing a runtime manager for GD were presented and evaluated in terms of complexity and quality of the delivered service [1][3][4][5].

Besides, another study shows the possibility of GD in a multiprocessor array [10]. The authors proposed an algorithm for reconfiguration considering that the multiprocessor array is partitioned into substitutable units at the granularity of pipeline stages. This algorithm aims to isolate defective units and connect fault-free units to form working processors.

In the context of this thesis, an event is a permanent fault generated by the DEG which affects the functionality of the component. Due to GD the component no more run with the same properties and some of its aspects will be degraded. These aspects could be performance (e.g. system speed, throughput), functionality (e.g. amount of total work done, precision of results), and energy consumption. However this transition to lower-performance system mode will prevent the system from catastrophic failures. Figure 2.2 shows an example of a component consisting of 4 working processors in its full-fledged mode. To complete 12 identical tasks, the system needs 3 time slots if we consider that each task demands one time slot to be done.



(a) full-fledged (b) degraded

Figure 2.2: Component operating in the full-fledged mode vs one degraded mode

Due to a permanent fault on the 4th processor, a graceful degradation takes place. The system will keep functioning with the remaining processors, and the tasks assigned to the faulty processor will be rescheduled to be achieved on the remaining 3 processors. As a consequence, the same 12 tasks require now at least 4 time slots to be executed, if the tasks of the failed processor are rescheduled equally between the remaining processors. In this degraded mode the same number of tasks will be achieved with fewer processors, the system will perform less and has now a lower throughput.

Apart from the performance degradation represented in Figure 2.2, we can have functional degradation, by dropping the tasks processor 4 was executing instead of rescheduling them.

2.4 Negative-Bias Temperature Instability

Negative-Bias Temperature Instability (NBTI) is the most serious oxide wearout mechanism that occurs at the transistor level [19]. With technology scaling, the oxide gate becomes thinner and the number of transistors per chip increases, leading to higher power densities and operating temperature. Accordingly, NBTI became worse especially with higher operating temperature [23] [24]. Consequently, NBTI should be taken into consideration when studying the reliability of a system, as it affects the CMOS lifetime [17] [19] [23] [24].

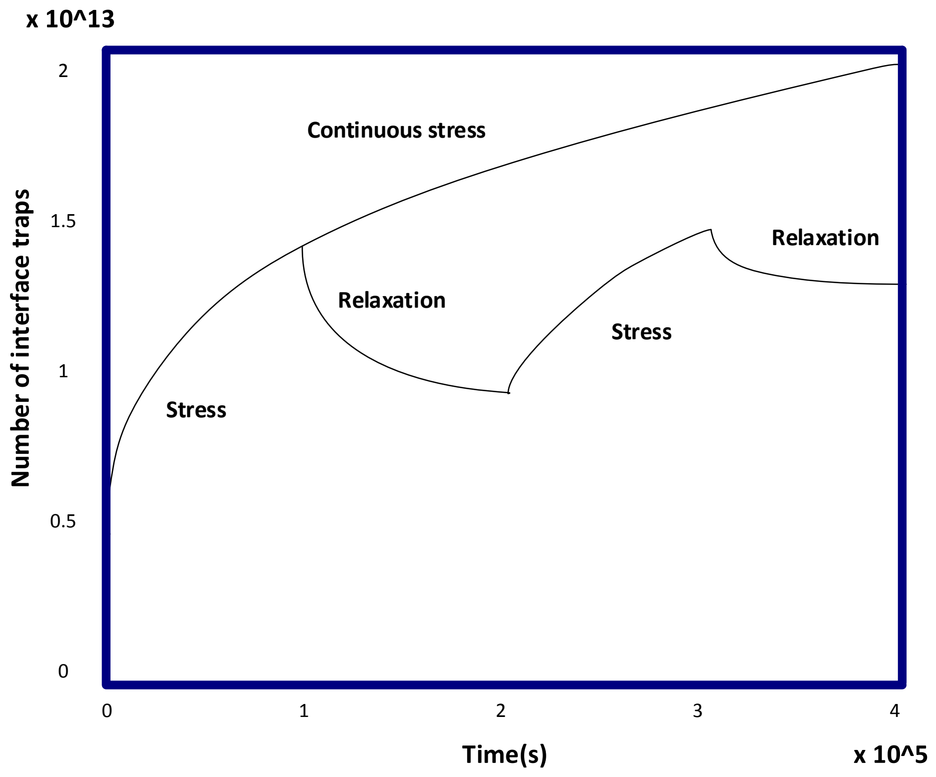


Figure 2.3: Comparison between static and dynamic NBTI, reproduced from [26]

NBTI arises in p-channel MOS transistors when a negative voltage ($V_{gs} = -V_{dd}$) is applied to the PMOS oxide gate under high temperatures. This is known as the *stress* phase. Interface traps are generated at the $Si - SiO_2$ interface. These traps are electrically active physical defects [26]. As a consequence, the threshold voltage increases as more traps develop, the gate leakage raises, and the drive current in the PMOS is reduced causing the transistor to switch slower. As the transistor becomes slower, this leads to a degradation in the system performance. Beyond that, at some point this delay causes the clock period to be violated which causes reliability issues potentially leading to total failure of the system. The NBTI phenomenon could be produced by either negative bias conditions or elevated temperatures [24], but became worse with the combination of these two effects. However, if the stress voltage is removed ($V_{gs} = V_{dd}$), parts of the interface traps are resolved helping the NBTI degradation to partially recover. This is known as the *recovery* phase. The NBTI can have an impact on electronic circuits in two different ways:

- Static NBTI: when the PMOS is constantly under stress
- Dynamic NBTI: when the PMOS go alternatively from stress to recovery phase

Figure 2.3 shows the number of interface traps for two cycles of periodic stress and recovery with $t_0 = 10^5$ seconds (≈ 1.16 days) and the measurement data are from [26]. On the other hand, the same figure emphasizes that a static stress produces an increasing number of interface traps. However, during dynamic circuit operation, it is more probable to suffer from dynamic NBTI rather than static NBTI. Therefore we will implement the NBTI effect in its dynamic flavor, and this will be explained later in Section 4.3.

3
Component Models

In the previous chapter, we provided a theoretical background in order for the reader to understand how we can deal with permanent faults and extend the components lifetime by degrading some of the system aspects. For instance, after each permanent fault, the component can lose performance, precision or functionality, and might not behave the same way the original one did.

Indeed, in order to implement fault models, we have first to identify what are the permanent faults that can appear in the component and second, develop a good understanding of the degradation aspects and consequences of faults on these aspects. In this chapter, we take the following steps:

- describe all the HW components which the dynamic event generator (DEG) will be able to deal with, and their organisation/architectural design,
- list all the events that can occur on each component,
- list the degraded modes for the component possibly by presenting their respective Markov chain,
- and finally discuss the degradation aspects and how each fault affects the original behavior of the component.

3.1 Component with Homogeneous Modules

To begin with, the simplest component is the one with homogeneous modules. This component consists of a number of identical modules which perform the same functionality and have the same reliability properties. Failure of any of the modules means that the component can keep functioning with the rest. We assume that a fault in one module cannot cause faults in any other module, or the control signals. Alongside the modules, there are control signals and interface circuitry with the rest of the system. A fault in these parts renders the whole component useless as this block is a single point of failure. This component is represented in Figure 3.1 showing n identical modules operating in parallel.

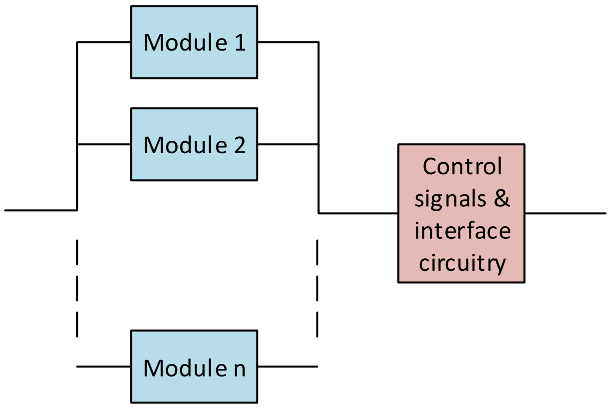


Figure 3.1: Component with n homogeneous modules

Each fault tolerant system consists of several fault containment regions. A fault containment region aims at preventing faults and errors from propagating from one module to another module ensuring that modules fail independently of each other. In this component there are $n + 1$ fault containment regions:

- n representing the blue modules in Figure 3.1,
- and one representing the whole component including the red part in Figure 3.1, since a fault in the red part disables the whole component

Therefore the reliability of the system is the product of the reliability of the modules and the reliability of the interface and control signals.

$$R_{sys}(t) = R_{modules}(t).R_{interface}(t) \tag{3.1}$$

Beyond that, the possible events that can occur on this component are:

- Permanent fault on the control signal and interface circuitry with high criticality, which renders the whole component useless whenever it occurs.
- Permanent faults with lower criticality that could appear on each of the modules. After each permanent fault, the corresponding module will be shut down and the whole component will work in a degraded mode with one module fewer than before.

The degraded modes of this component are shown using the Markov chain in Figure 3.2. Anytime a permanent fault with low criticality appears on a module, the component will still work if at least one module is working along with the interface and control signals. Note that by low criticality, we mean that this kind of faults cannot lead to failure unless the component is at the lowest level of graceful degradation.

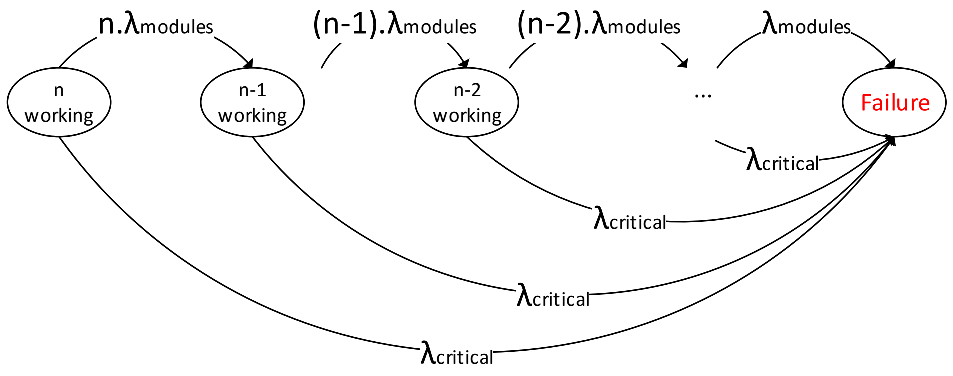


Figure 3.2: Markov chain for n modules working in parallel

However, after each permanent fault, the two different degradation aspects are:

- result precision,
- and/or performance,

depending on the application in question. If for example, this component is used for telecommunications, and each module is responsible to transmit certain frequencies, the degraded component will drop some packets of the voice data rather than closing the connection. Therefore, the voice quality delivered has less precision and is gracefully degraded to lower levels. On the other hand, if this component is responsible to do calculations with the aid of all the modules, after each permanent fault the remaining tasks will be divided to be done on the remaining modules, thus the job will need more time to be done. In this way the performance degrades as more time is needed to perform the same number of tasks.

3.2 Component Supporting Frequency Scaling

The next component we will discuss is the component that supports frequency scaling. The various parts of this component are irrelevant and its functionality cannot be degraded – it either works or does not. However, when (due to aging or due to a timing fault detected with testing), it cannot run at the intended frequency, it can work at a lower clock frequency to keep functioning at a slower speed. This frequency reduction can be performed a predefined number of times n before the component is considered entirely failed.

This component consists of $n + 1$ fault containment regions each representing the same component operating at different frequencies. The possible events that can occur on this component are:

- Permanent fault for the uncovered events with high criticality, which renders the whole component useless whenever it occurs. These faults affect the functionality of the component. Once they have happened the component cannot be repaired.
- Permanent faults with lower criticality that affect the speed of various parts with a coverage factor c. These faults causes the whole circuit to function

slower. As a consequence, to repair the component functionality after each permanent fault, the operating frequency will be lowered. When the number of frequency reduction reach the predefined number of time n, the component will end up in the "Failure" state.

Usually, the coverage factor c is the probability that a fault is covered by the fault tolerant mechanism, thus detected and repaired. In this case the fault tolerant mechanism is the frequency scaling. However, we don't have the ability to detect this type of faults, and by "coverage factor" we mean the percentage of faults that can be repaired by this mechanism. So in this context c is not about ability to detect faults, but is only with respect to ability to repair them. Accordingly, $1-c$ is the probability that a fault is non-covered, therefore whenever such fault happens, the component fails. The degraded modes of this component are shown using the Markov chain in Figure 3.3.

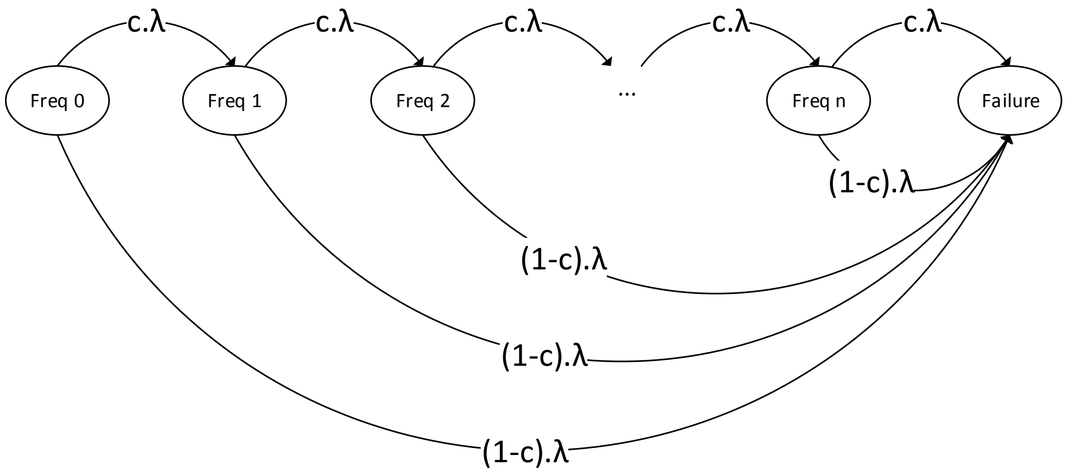


Figure 3.3: Markov chain for component supporting frequency scaling
n = predefined number of frequency reduction before total failure
c = coverage factor

Each time a permanent fault appears on this component, the main aspect that is affected is the performance. Due to reduction of the clock frequency, the component will run slower and cannot perform as well as before. The same tasks still can be done but they will need more time to be achieved.

3.3 Memory

Another interesting component to discuss is the memory that operates as following. A test routine is periodically applied to the memory of certain size (say, 4MB). When a permanent fault is detected in one or more memory bits, the controller is able to isolate regions of the memory, by fixing specific bits of the address bus to a constant value. In this manner, memory capacity is halved. The minimum size required for the memory to be usable is 512KB. Note that there are less wasteful memory fault tolerance techniques, but we chose this one for simplicity in demonstrating the abilities of the tool. This component is represented in Figure 3.4 showing eight identical regions each of 512KB.

As we managed to divide the memory into eight identical regions of 512KB each, this component consists of eight fault containment regions physically. However, the simple fault tolerance mechanism that we chose in order to demonstrate the DEG function, is wasteful and sacrifices more regions of the memory than it has to. Accordingly, a fault appearing in one of the eight regions sacrifices other regions and the fault propagates by disabling half of the active regions. The underlying model we use in the DEG can be used with a better fault tolerance mechanism, by changing the reconfiguration routine every time a fault happens, in order to implement a more efficient strategy. The possible events that can occur on the memory component are:

- Permanent fault for the uncovered events with high criticality, which renders the whole component useless whenever it occurs. One such example of these faults is when a memory controller which is responsible for the flow of data in and out of the memory, became non responding turning this component unusable.
- Permanent faults with lower criticality with coverage c that could appear on certain region of the memory. These non-critical faults are specific memory bits being corrupted. In order to tolerate such faults, the memory size will be halved when a fault occurs to reach the last stage of having only one working region. Beyond that, any kind of faults will lead to total failure.

512 KB
512 KB
512 KB
512 KB
512 KB
512 KB
512 KB
512 KB

Figure 3.4: Memory with 8 identical regions of 512KB

Note that the coverage factor c has the same meaning as discussed in section 3.2. The degraded modes of this component are shown using the Markov chain in Figure 3.5.

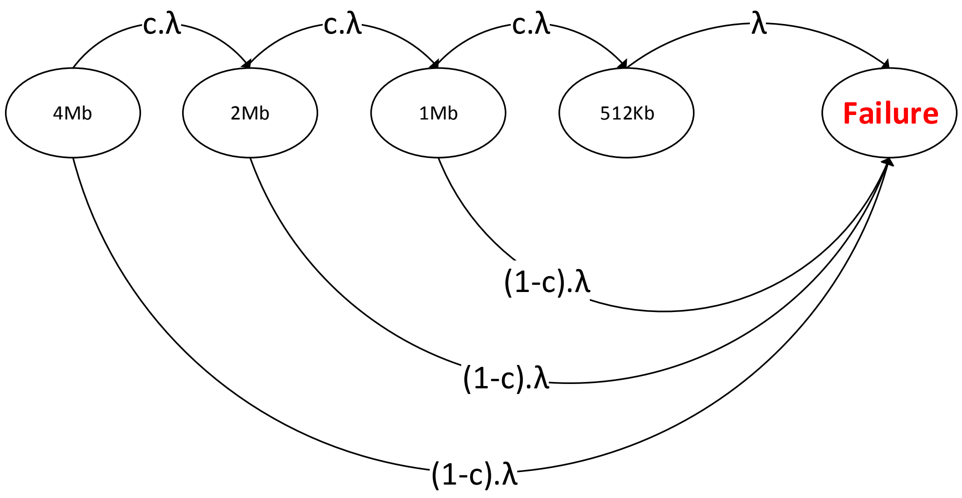


Figure 3.5: Markov chain for memory component of 4MB able to operate with a least 512KB

Due to reduction of the memory size after each permanent fault, the degradable aspect is the memory capacity. Accordingly, the system that uses the memory loses performance because of the lower memory capacity. The same tasks still can be done but they will need more time to be achieved.

3.4 Heterogeneous Component

The next component to discuss is a component which is composed of heterogeneous modules, with different reliability properties and the failures of which result in different degraded or failed modes. As an example, we will consider the following potential core of a multicore System on a Chip represented by Figure 3.6. The operations that are performed by the floating point and integer units, in the normal (full-fledged) component functionality are well-defined. The degraded modes resulting from failures of various units will be defined in this section.

This component consists of nine fault containment regions in total as follows:

- **8** for the cache as described in the previous section,
- **1** for the floating point unit,
- **1** for the integer unit,

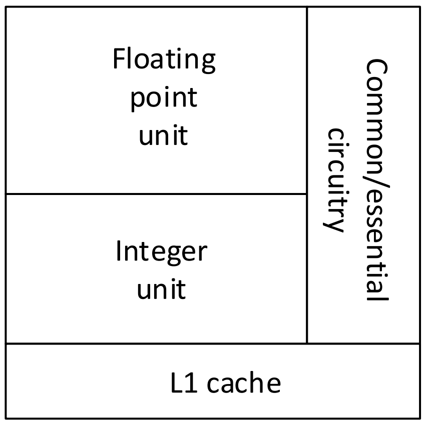


Figure 3.6: Block diagram of the heterogeneous componenent

The degraded modes of this component are shown using the Markov chain in Figure 3.7. The possible events that can occur on this heterogeneous component are defined below with their respective degradation modes:

- Permanent fault on the control signal and interface circuitry with high criticality, which renders the whole component useless whenever it occurs.
- Permanent faults with lower criticality that could appear on the integer unit. In the case where we have just one integer unit, a permanent fault on this module will render it useless and it will be shut down. At this point any integer operation, could still be performed using the floating point unit. Undoubtedly, a floating point unit performing both integer and floating operations will be more busy. Therefore, the utilization rate of this unit will be higher. Consequently, the performance of the component will degrade and the fault rate of the floating point will increase affected by higher duty cycle.
- Similarly, permanent faults could appear on the floating point unit. Likewise, operations requiring floating point unit could be done on the integer unit. However, precision will be sacrificed beside the performance issues discussed in the previous point. This degradation is possible only for tasks that can tolerate this precision loss, e.g. tasks of a program that uses the approximate computing paradigm. As well the integer unit performing both floating point and integer operations will have higher fault rate and be more likely to fail due to its busier state.
- Permanent faults with lower criticality with coverage *c* that could appear on the cache. The cache of the component degrades in the same way as the memory component described in the previous subsection.
- Permanent fault for the uncovered events with high criticality, which renders the cache unit useless whenever it occurs.

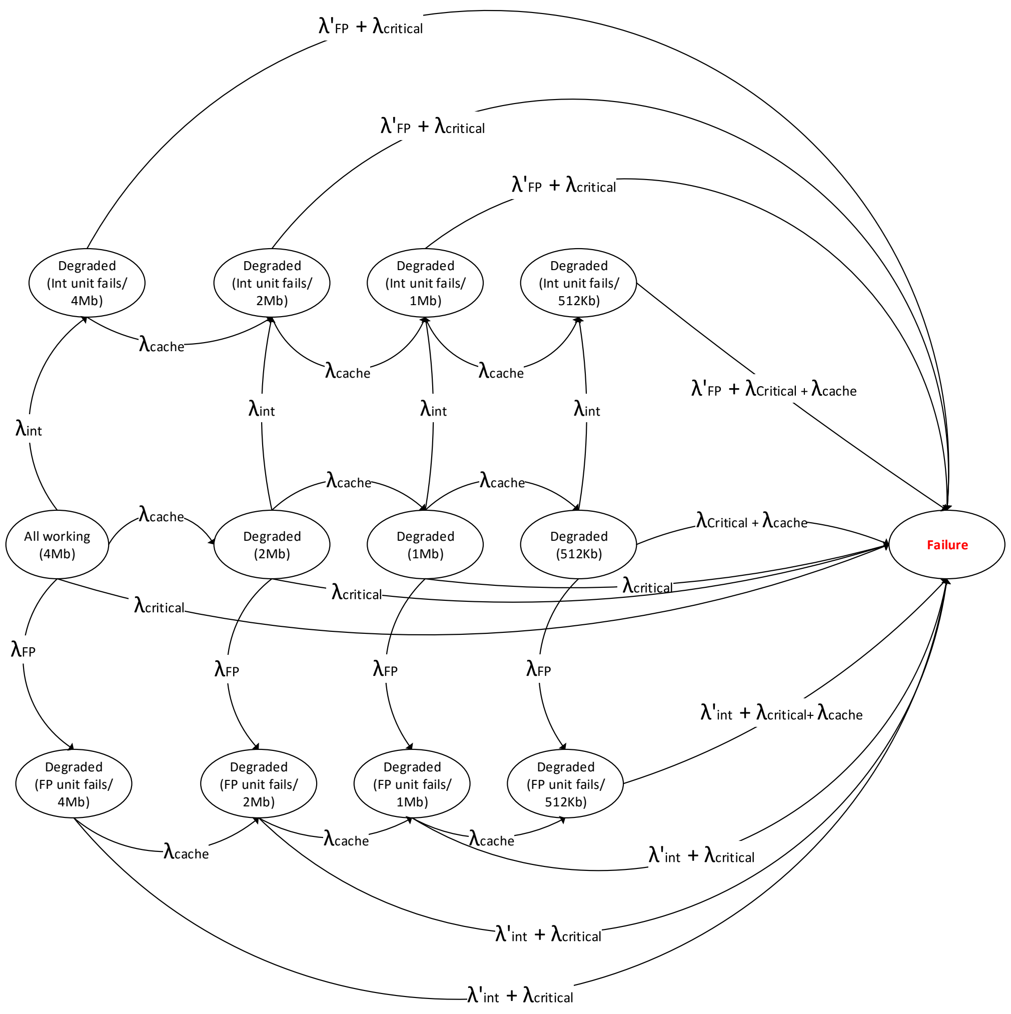


Figure 3.7: Markov chain of the heterogeneous componenent

This component can be extended with more functional units of any type, which have already been described in the previous sections and that degraded modes will exist accordingly.

3.5 Reconfigurable m-by-n Processor Array

The next component to be discussed is the reconfigurable m-by-n processor array, with m being the number of different substitutable units that one processor contains and n being the total number of processors. An array of processors is modified to support reconfigurability as follows: Each processor of the array is divided into a number (say 4) of substitutable units as shown in Figure 3.8. Reconfigurable interconnections are added between the parts, such that allow any part of one processor to be connected with any (fitting) part of another processor. The various pieces are laid out in such a way as to form an array, the rows of which are the different processors, and the columns are groups of identical processor parts. When pieces of different processors fail, the remaining pieces can be thus connected to form a working processor as shown in the degraded mode in Figure 3.9.

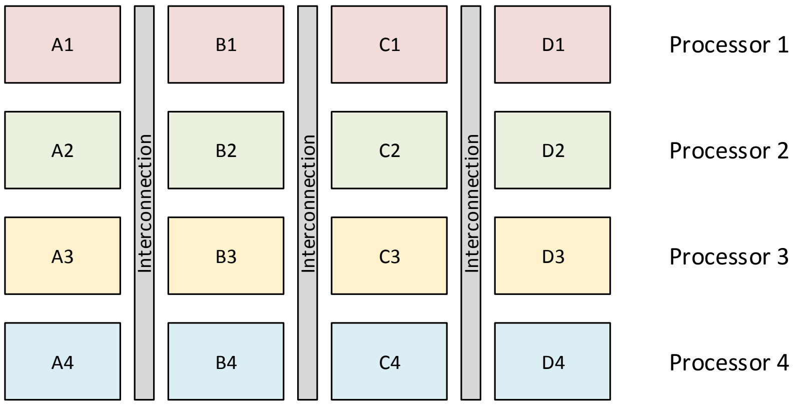


Figure 3.8: 4-by-4 processor array

In this example, we consider that B1, C2, D3, C4, and D4 are faulty. After reconfiguration, processors 1 and 2 could still operate because no interconnection has failed, thus rearrangement to achieve graceful degradation is possible.

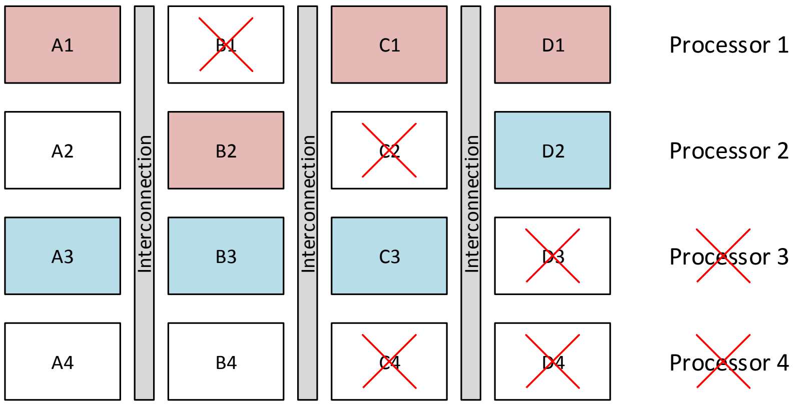


Figure 3.9: 4-by-4 processor array after failure of some parts

For this component we want to take into account the following kinds of faults:

- Faults that cause a substitutable unit to fail
- Faults that cause the interconnect infrastructure between two columns to fail. In this case, we assume that there is a dedicated connection between substitutable units of the same row, and if the interconnect fails, only this default connection is possible.

Accordingly, this component consists of $m*n$ fault containment regions representing the different substitutable units, and $m-1$ fault containment regions representing the interconnect infrastructures.

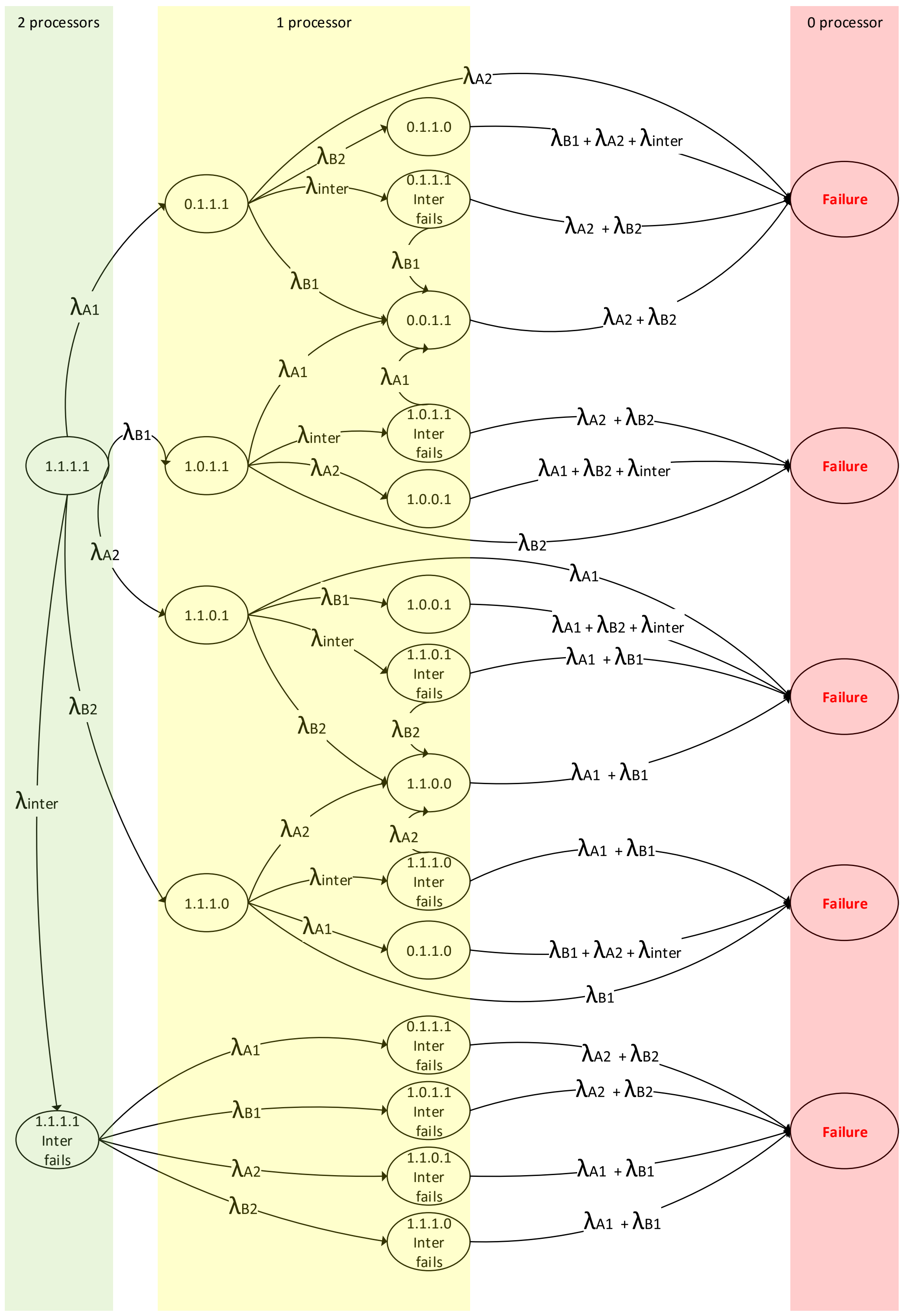


Figure 3.10: Markov chain of 2-by-2 processor array

The degraded modes for arrays of practical size are too many to be possible to enumerate. However, what can generally happen is that we have 0 to n working processors. Undoubtedly, for each processor we lose we will lose performance as this processor is shutdown. However, based on how many interconnects these reconfigured processors (as the one shown in Figure 3.9) use, they will be slower and less energy efficient than the original processors operating without the use of interconnections. A Markov chain of the degraded modes for a 2*2 processor array could be present to give a better idea of possible degradation modes as shown in Figure 3.10. The name of each state in the Markov chain has the pattern $x.x.x.x$ which represents if the parts $A1.B1.A2.B2$ have failed or not. 1 represents a functional part whereas 0 represents a defective one. Note that in the case of 2-by-2 processors array only one interconnect is used for reconfigurability.

3.6 3D-stacked DRAM Hybrid Memory Cube

The last component to be discussed is the 3D-stacked DRAM Hybrid Memory Cube (henceforth HMC). Multi-core processor performance are limited by "the memory wall" [11]. The three dimensional stacked DRAM architecture is a revolutionary design that reduces the distance that signals travel. As a consequence, the HMC is a memory die with improved latency, bandwidth, power and density [11].The HMC system diagram is shown in Figure 3.11. In this section we will take a brief look at this design in order to implement it in the DEG and study the graceful degradation aspects.

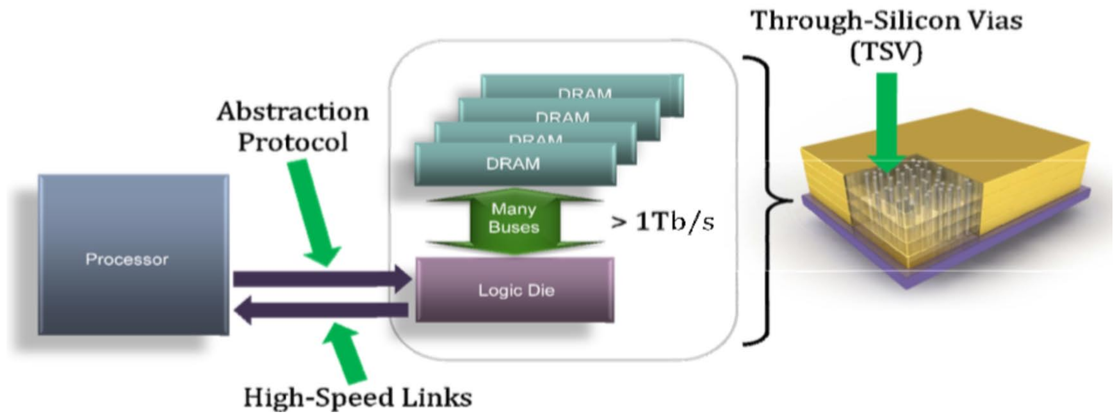


Figure 3.11: HMC System Diagram from [11]

As shown in Figure 3.12, the HMC in question consists of 8 stacked DRAM layers. Each layer contains 32 partitions, and each partition consists of 2 banks. Each 8 partitions are combined together to form a memory vault that is controlled by a vault controller implemented on the logic layer. So in total there are 32 vaults, and each vault contains 16 banks. The layers are connected using through-silicon via (TSV) technology and fine-pitch copper pillar interconnect. If we consider one vault, their partitions are connected using 32*8 = 256 TSV's plus 32 TSV's for the error-correcting code (ECC).

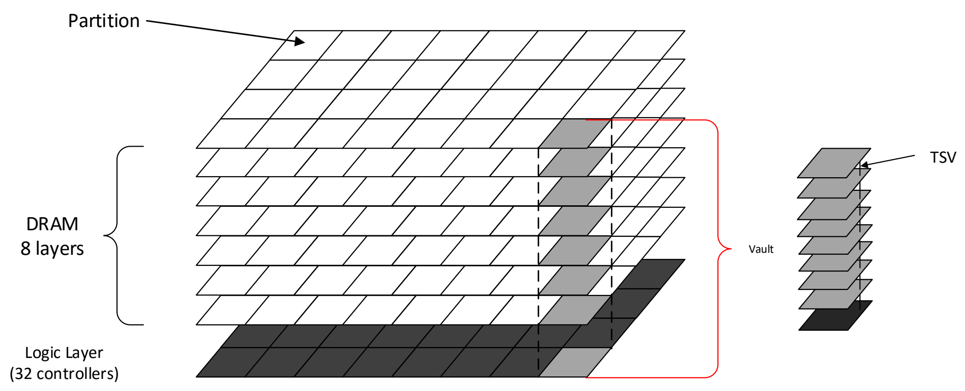


Figure 3.12: HMC architectural design

This vault could tolerate 2 faulty TSV's, otherwise will be considered faulty. At the bottom of the DRAM stacked layers exists one logic layer consisting of 32 controllers, and each of them is connected to its respective vault. Besides, there are 4 routers as shown in Figure 3.13. Each router is linked to one core from one side, and linked to 8 controllers from the other side. These routers are joined through links. In total there are 6 links.

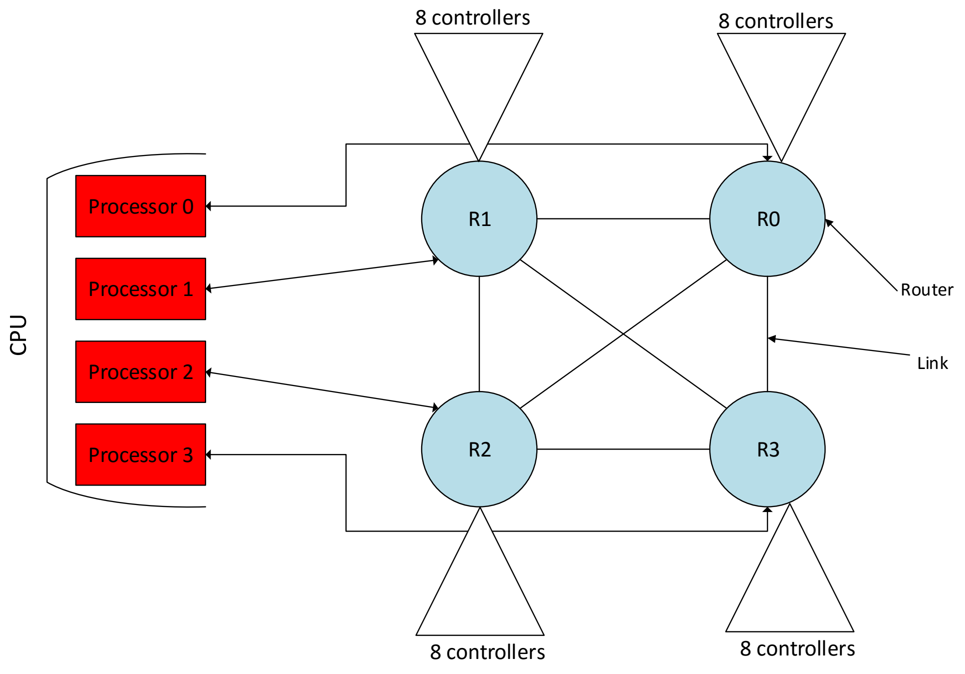


Figure 3.13: Routers Architecture

The fault containment regions for the HMC component are:

- **4** for the routers,
- **6** for the links,

- **32** for the controllers,
- **512** for the banks,
- and **9216** for the TSV's.

In total there are **9770** different fault containment regions.

A controller on which an event happens, is disabled and the cube continues to function in a degraded mode losing 1/32 of its original capacity. While a permanent fault that appears on one router will render the 8 controllers useless, respectively their 8 vaults. In this case we lose 1/4 of the total original capacity. Furthermore, a permanent fault that appears on one bank will render the neighbour bank as well useless and will be both deactivated losing 2/512 of the total capacity. However, a permanent fault that appears on a link between two routers doesn't necessarily isolate it. There is possibility for a core to address one router by performing multiple hops, in other words getting to the destination through other routers. Referring to Figure 3.13, if core 0 wants to send a request to router R2 and the link between R0 and R2 is broken, it is possible to send the request with two hops, by going from R0 to R1 and from R1 to R2 if the used links are reliable. The main degradable aspect for the HMC component is the memory capacity. However, the performance of the HMC can degrade, e.g. in the case when we have faulty links or routers, more hops have to be made for a specific request (read/write) which takes more time for a memory access.

3.7 Components Summary

In this chapter we presented the following fault tolerant components that degrade gracefully:

- Component with homogeneous modules
- Component supporting frequency scaling
- Memory component
- Heterogeneous component
- Reconfigurable m-by-n processor array
- 3D-stacked DRAM Hybrid Memory Cube

The degraded modes were shown with the aid of a Markov chain when it is possible or the main considerations were enumerated like in the case for the HMC.

Furthermore, we discovered that depending on the component, a non-critical fault can lead to a degradation in some aspects of the system. Such aspects could be loss of performance, result precision, energy efficiency, and memory capacity.

Finally, we defined fault containment regions for each component and based on these we defined a number of events that can happen. Each of these events has its own event generator, and the implementation of these event generators will be discussed in details in the next chapter.

4 Tool Implementation

Chapter 4 contains the main decisions throughout the design and implementation phases. With a theoretical background from Chapter 2 and well defined models from Chapter 3, a good platform is sufficient to build on our design. We will present the construction of the event prediction based on the fault rate λ and the exponentially distributed threshold E. Also, we will explain how components and their various possible faults are modelled in the tool. Section 4.1 defines the useful modules to port from the basic event generator, and how we upgraded the core structure to operate in closed-loop and to predict events dynamically. Section 4.2 discusses the integration of the dynamic event generator (DEG) using a reconfiguration module as an interface between the tool and the rest of the experimental setup. Section 4.3 explores the negative bias temperature instability (NBTI) effect implementation with the help of an abstract probabilistic model which does not need low-level information. Section 4.4 discusses the implementation of the reliability models defined in Chapter 3. Lastly, Section 4.5 summarizes the main points of this chapter. A user guide in appendix A.1 presents instructions about how to use the tool, for anyone that will do so in the future.

4.1 Porting of Useful Modules

Before the design and implementation phases for the DEG, a deep analysis of the basic event generator (EG) helps to understand the main functionality. This is a good baseline to build on, and some of the modules are useful to port rather than re-design and implement. In addition, the basic EG was implemented in JAVA language using object-oriented technique which makes developing of high quality software much easier. Accordingly, we decide to keep the main structure of the event generator by porting the useful modules and classes into C++ language which supports object oriented programming too. Most of these classes are needed to become the foundation of the DEG. Note that **generator** is an instance of the class **lamdaSeries** when it is not used as the name of the tool. The main classes, to generate an event are:

- **Chip:** This class populates the components with their respective generators.
- **HWComponent:** This class contains all the information needed to populate the generators with the requested λ type.
- **lamdaSeries:** This class generates different fault scenarios and contains a function that decides if an event occurs at a specific time.
- **methods:** This class contains the function that generates a random limit for

all the generators. Whenever that limit is reached, an event arises.

In fact, almost all the tool modules were ported to C++ language and were upgraded to meet our specifications. Even though we will probably not use all of the fault scenarios in our simulations, these functions were ported to the C++ tool version for future studies. Some of the different fault scenarios that vary in time and space are shortly described as follow;

- **Baseline:** This scenario creates constant failure rates equal for each time slot in the simulation. The climbing function toward the threshold E is a SUM function of the λ rates for each time slot. Thus, the constant failure rates scenario will produce a linearly increasing function toward the limit threshold E as shown in Figure 4.1.

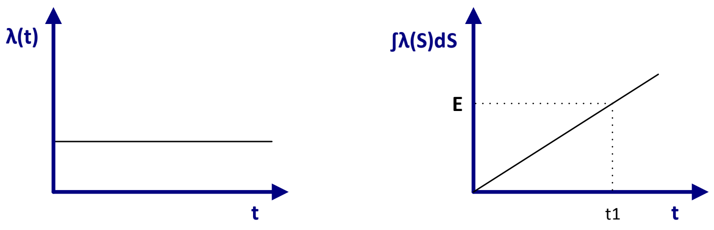


Figure 4.1: Fault generation procedure for the constant λ

- **Peaks:** The fault rate function λ has peaks in specific periods of time due to stressful workload or high temperature. During these peaks, λ is higher meaning more probable failure.
- **Fatal:** This scenario is pessimistic with the presence of faults that could disable the components completely, therefore they have high fault rates. There are still faults that do not disable the component completely.
- **Bathtub:** This scenario represents fault rates following a bathtub curve. There is high probability of failure during infant and wear-out phases as illustrated in Figure 2.1.

Considering the basic EG, the different fault scenarios were the way of simulating systems with variable temperature and workload features. However, the upgrades we made to the DEG make these scenarios obsolete, since now the tool is dynamic and can adapt the fault rates according to the actual values. The baseline scenario is the most convenient one that will be used in our simulations, because our target is to implement a DEG that works in closed-loop with the experimental setup. Furthermore, the fault rates λ will not be constant during the simulation lifetime, they can be dynamically modified according to configuration information (temperature, workload). This point will be discussed in detail in Section 4.2.

4.1.1 Upgrade to Operate in Closed-Loop

We have discussed previously in Section 1.2 that our goal is to implement a tool in C++ language that operates in closed-loop with the rest of the experimental setup.

The purpose of closing the loop is to get feedback from the components since we want to study their reliability and inject the predicted events by the DEG.

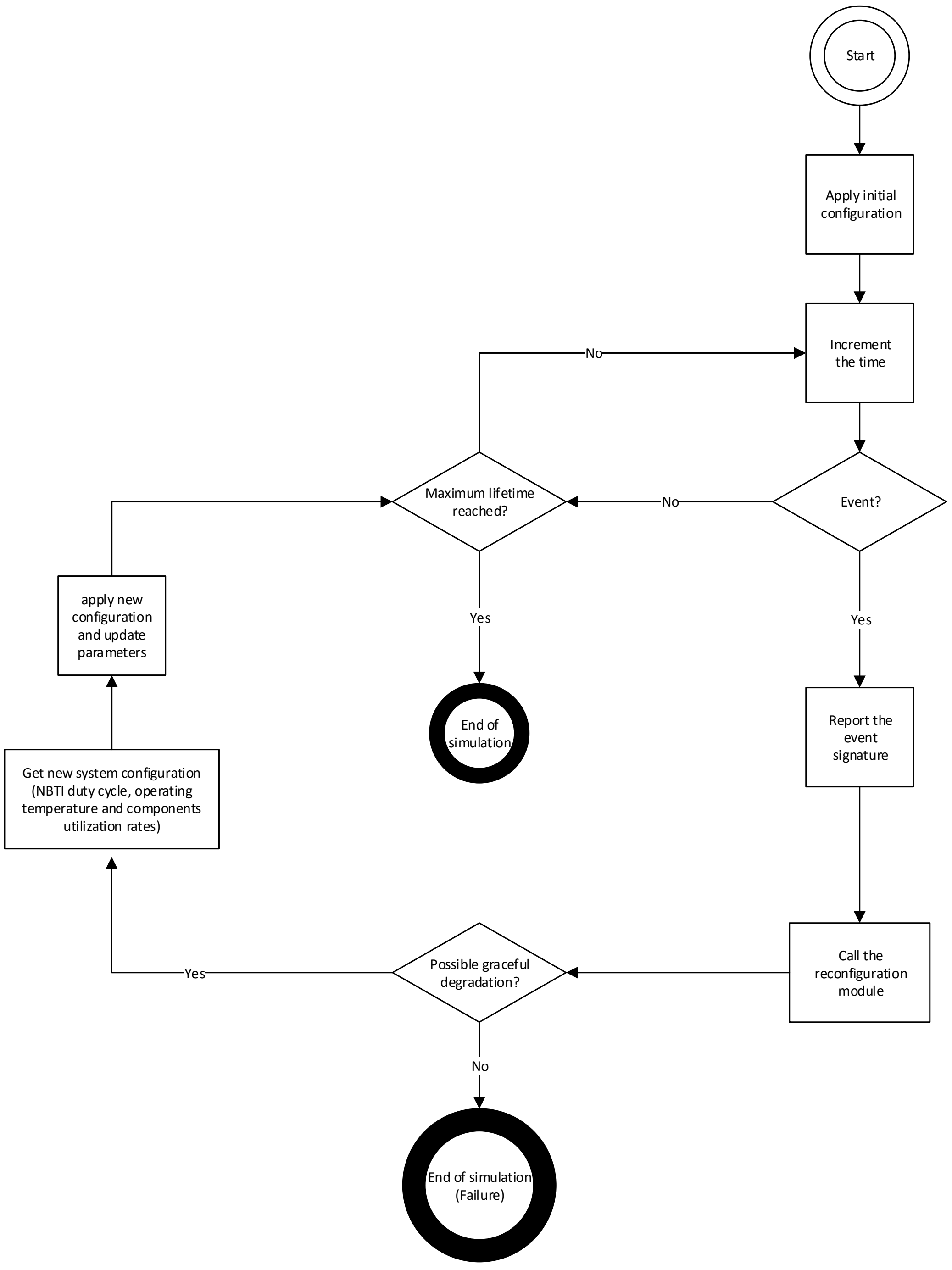

Start
Apply initial configuration
Increment the time
No
Event?
No
Maximum lifetime reached?
Yes
End of simulation
Yes
Report the event signature
Call the reconfiguration module
Possible graceful degradation?
Yes
No
End of simulation (Failure)
Get new system configuration (NBTI duty cycle, operating temperature and components utilization rates)
apply new configuration and update parameters

Figure 4.2: The closed-loop flow

These events have effects on the location they appear. They could degrade the component aspect(s) or declare it as failed. Once an event is produced a new system configuration will update the DEG parameters changing the predicting process. The closed-loop flow is illustrated in Figure 4.2. A simulation starts with static initial configuration defining the components with their failure rates. Temperatures and duty cycles for all components are passed to the DEG, since these parameters are needed in predicting the next fault. After this point, if we have not reached the end of simulation time, the DEG keeps testing if an event is produced by any of the generators.Whenever a generator produces event, the event signature is reported and the reconfiguration module is called. At this point if a graceful degradation is possible, a new system configuration is requested. This new system configuration contains the parameters that are relevant for fault prediction. These parameters are the operating temperature and the duty cycles for each component. A component with a high duty cycle means that it is stressed by a high workload, and the failure rate of its corresponding generator should increase. However, if we are on the last graceful degradation and no more fault could be tolerated, the simulation ends and reports that the system has failed.

4.1.2 Upgrade to Predict Events Dynamically

Undoubtedly, a tool that operates in closed-loop with the experimental setup is supposed to take new parameters dynamically as well report events immediately whenever they occur. The basic event generator produces events beforehand as shown in Figure 1.1. However, in order to predict events dynamically, we considered the following:

- We take into account changing parameters in real time. Thus, we get new fault-relevant parameters from the reconfiguration module and modify our fault rates accordingly. Right now our fault rates are updated whenever an event occurs. However the frequency of reading these parameters and updating the fault rates could be defined to be at maximum (every time step) which increases both the prediction accuracy and complexity of the tool.
- The DEG should check if the threshold limit E for every generator is reached every time step. For this purpose we create a variable named $current[i][j]$ that added the failure rates λ's for each generator in time. Note that 'i' and 'j' are indexes to identify the component and the generator respectively. Each time step and for every generator we update the $current[i][j]$ by adding the corresponding failure rate λ, and check if $current[i][j] \geq limit[i][j]$. If this condition is satisfied, an event occurs and the corresponding signature (time, location, type) is reported dynamically to the reconfiguration module.

4.2 Integration with the Experimental Setup

The reconfiguration module is the interface between the DEG and the rest of the experimental setup. This module is responsible to report any event that occurs on any component to the system manager. Figure 4.3 shows the exchanged parameters between the DEG and the reconfiguration module. The parameters that the DEG provides to the reconfiguration module are:

- event signature (time, location, type of the fault) whenever it occurs, and
- the generators status.

In contrast, the reconfiguration module provides the DEG with parameters that affect the prediction process. These parameters are :

- the traditional utilization rate,
- the steady-state-temperature, and
- the NBTI duty cycle λ_{NBTI} for each operating component.

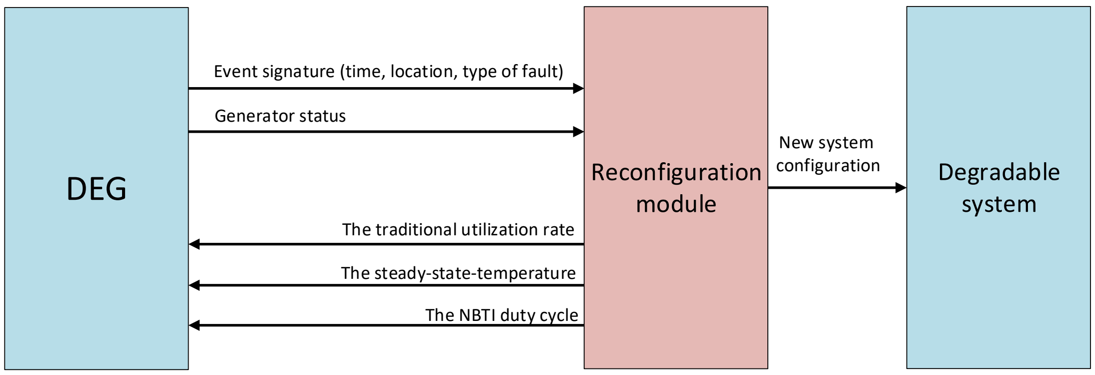

DEG
Event signature (time, location, type of fault)
Generator status
The traditional utilization rate
The steady-state-temperature
The NBTI duty cycle
Reconfiguration module
New system configuration
Degradable system

Figure 4.3: Exchanged parameters between the DEG and the reconfiguration module

A mock reconfiguration module has been implemented, providing customizable random values for the relevant parameters listed above, for debugging/testing purposes and for cases for which a reconfiguration module has not been provided. The steady-state-temperature T_i and the NBTI duty cycle λ_{NBTI} parameters serve as inputs to calculate the threshold voltage degradation ΔV_{th} and this will be discussed further in Section 4.3. Concerning the traditional utilization rate, this parameter gives a reasonable awareness of the component workload and the expected lifetime.

Furthermore, the reconfiguration module runs various algorithms for deciding the next system configuration and the specifics about the algorithms are irrelevant to the DEG, the only relevant part being the resulting configuration. For instance, it is essential for the m-by-n processor array to get feedback from the reconfiguration module to be aware of possible reconfigurations. This part is implemented in the reconfiguration module and it is integrated within the DEG to be able to run simulations. The evaluation of the reliability results will be discussed in Chapter 5.

4.3 NBTI Implementation

The NBTI effect is implemented using the dynamic version as stated in Section 2.4, because it is more likely to have alternating stress relaxation phases. We based our implementation on the long-term ΔV_{th} predictive degradation model as discussed in [25]. This model has been verified with the dynamic short-term variation (stress/relaxation cycles) in [18] and it is illustrated in Figure 4.4. The result shows that this predictive model matches well with the upper bound of the short-term model with a difference within 5% for different values of NBTI duty cycle α.

The absolute value of the threshold voltage degradation is a function of time and it is exponentially dependent on the operating temperature T, supply voltage

V_{dd} and NBTI duty-cycle as reported in Equation 4.1 [25],

$$|\Delta V_{th,t}| \approx \left(\frac{\sqrt{K_v^2 \alpha T_{clk}}}{1 - \beta_t^{1/2n}} \right)^{2n} \tag{4.1}$$

where β_t is the fraction parameter of the recovery [18] expressed in Equation 4.2,and t the time.

$$\beta_t = 1 - \frac{2\epsilon_1 t_e + \sqrt{\epsilon_2 C(1-\alpha)T_{clk}}}{2t_{ox} + \sqrt{Ct}} \tag{4.2}$$

C is dependent on the temperature [27], k is the Boltzmann's constant, T_0 another constant parameter, and T the operational temperature for the component.

$$C = T_0^{-1} exp(-E_a/kT) \tag{4.3}$$

K_v has dependence on electrical field and supply voltage [25]

$$k_v = \left(\frac{qt_{ox}}{\epsilon_{ox}} \right)^3 K^2 C_{ox}(V_{gs} - V_{th}) \sqrt{C} exp\left(\frac{2E_{ox}}{E_0} \right) \tag{4.4}$$

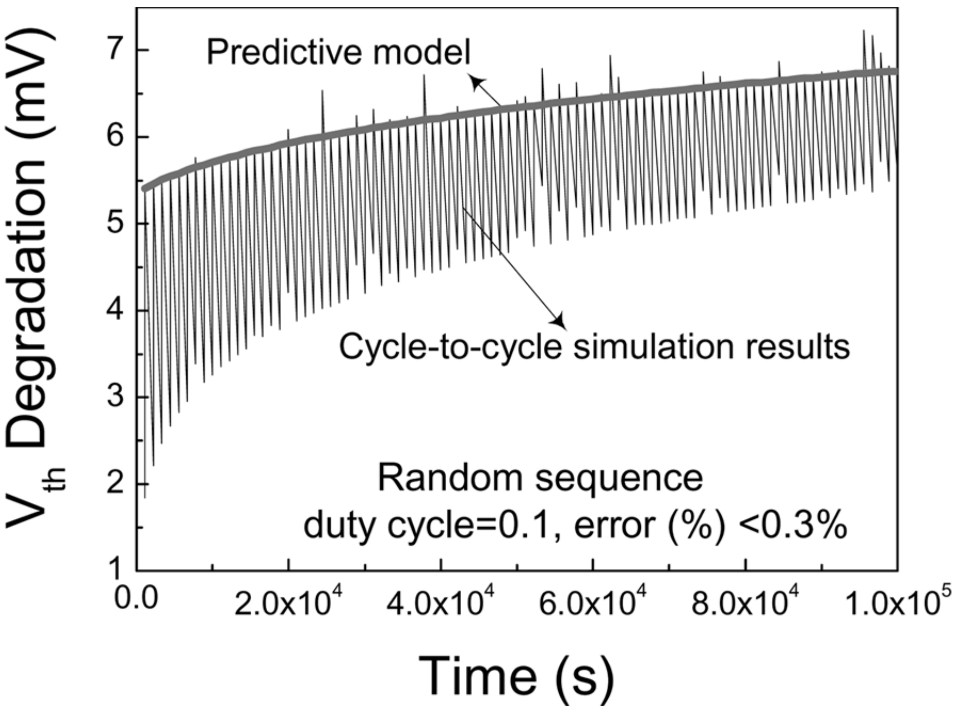


Figure 4.4: Long-term prediction model verification [18]

The NBTI duty cycle is different than the utilization rate and is defined in Equation 4.5 considering that the stress cycles are the NBTI stress period and that the recovery cycles are the relaxation period [28].

$$\alpha = \frac{StressCycles}{StressCycles + RecoveryCycles} * 100 \tag{4.5}$$

T_{clk} is the time period of one stress-recovery cycle and since the ΔV_{th} is not sensitive to the switching frequency for a value above 100 Hz [25], in our simulation we set T_{clk} = 100 Hz.

The values of the technology parameters are represented in Table 4.1 with their corresponding description and units [27] and to match with the 32 nm node, we considered the corresponding values for t_{ox}, V_{gs} and V_{th}.

All the functions and parameters described above serve to calculate the threshold voltage degradation ΔV_{th} in time for one specific transistor. In order to go from transistor level to logic circuit level, many considerations have been made.

Symbol	**Value**	**Unit**	**Description**
ϵ_1	0.9		backdiffusion constant
ϵ_2	0.5		backdiffusion constant
$t_e = t_{ox}$	1.1	nm	oxide thickness
T_0	1.1	s/nm^2	constant parameter
E_a	0.49	eV	the activation energy of hydrogen species
k	8.6173324E-5	$eV/Kelvin$	the Boltzmann's constant
q	1.60217662E-19	$Coulombs$	electron charge
ϵ_0	8.854187817E-12	$F.nm^{-1}$	vacuum permittivity
ϵ_{ox}	3.9 x ϵ_0	$F.nm^{-1}$	the oxide permittivity
K	7.5	$(C^{-0.5}nm^{-2.5})$	a parameter used in literature to minimize the overall error against measurement
C_{ox}	$\frac{\epsilon_{ox}}{t_{ox}}$	$F.nm^{-2}$	the oxide capacitance
V_{gs}	0.9	V	supply voltage
V_{th}	0.16	V	threshold voltage
E_{ox}	$(V_{gs} - V_{th})/t_{ox}$	$V.nm^{-1}$	the vertical electrical field
E_0	0.08	$V.nm^{-1}$	technology dependent parameter
n	$\frac{1}{6}$		time exponent parameter for the H2 diffusion species

Table 4.1: Model parameters

To implement the NBTI formula, temperature T is one of the parameters needed. Many temperature modeling tools (like HotSpot [12]) could produce heat and temperature maps for the whole chip, but require synthesized designs. However, our models do not have such low-level information. Therefore, the DEG will work with estimated steady-state-temperature for each component in the system. The set of estimated temperature T_i is produced by the reconfiguration module which is not part of this thesis. A new steady-state-temperature value is generated for each component in the system whenever a fault occurs in any of the generators.

The NBTI duty cycle α in Equation 4.5 is the other parameter needed in our implementation. Our assumption is that, when a component is idle, its transistors get in the recovery phase. Regarding the stress phase, this factor depends on workload of the component. As the temperature, the NBTI duty cycle in our tool will be approached in component granularity. Once more, the reconfiguration module is responsible to provide the DEG a typical duty cycle D_i for each component. Likewise the temperature, a new NBTI duty cycle value is generated whenever a fault occurs in any of the generators according to the new system configuration.

Above all, each component consists of millions of transistor that do not behave the same due to process variation caused by fabrication [21] [22]. Accordingly, a certain threshold voltage degradation of $X\%$ can cause a fault on a particular transistor but not on the others. Furthermore, transistors that are on the critical path or in regions on the die with high activity and workload are more probable to create faults. Similarly, this fine-grain information is not available to us, therefore we decided to work probabilistically. Accordingly, our goal is to implement a much more abstract predictive model, which does not need all the low-level information mentioned above. Unfortunately, this tool will trade ease of use for precision and we cannot expect it to be as precise as low-level tools. Indeed, we convert the threshold voltage degradation ΔV_{th} into failure rates λ and implement it in our DEG to generate faults caused by the NBTI phenomenon.

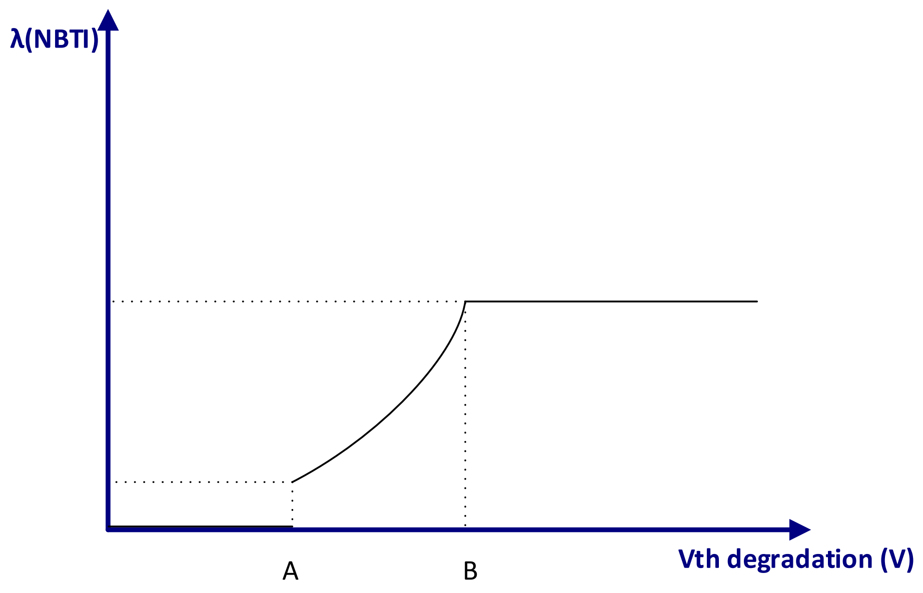


Figure 4.5: λ_{NBTI} in funtion of ΔV_{th}

Our probabilistic model is shown in Figure 4.5 and described using two threshold voltage degradation points, A% and B% defined as follows:

- **Threshold A:** All threshold voltage degradations below this value have no chance of generating faults due to NBTI. Even the worst transistor in the system can tolerate a threshold voltage degradation up to A%. Below this lower bound we consider that the failure rate $\lambda_{NBTI} = 0$.
- **Threshold B:** This level of degradation is so severe that there will almost certainly be NBTI-caused faults. Whenever this upper bound is reached, the failure rate of the corresponding NBTI generator has to be high enough to assure causing a fault. For example setting $\lambda_{NBTI} = 1$ means that an event will occur within the next few simulation steps.

- **Degradation between A and B:** For all other values of Vth degradation, the appearance of faults is almost deterministic, but in this region the rate should be increasing in a manner as realistic as possible as ΔV_{th} grows. In fact, the transistors initial threshold voltage values are usually assumed to follow a Gaussian distribution, meaning that any further ΔV_{th} degradation will constitute vulnerable a larger fraction of the total component transistors than the previous one, thus the fault rate should have a near-exponential behavior. The challenge is to find the corresponding exponential function describing $\lambda_{NBTI} = f(\Delta V_{th})$ and we are looking for the correct parameters of this distribution.

The calibration process will be presented in Section 5.4 and the results will be verified and discussed.

4.4 Implementation of Reliability Models

In this section we elaborate on the most important design decisions and explain how the models presented in Chapter 3 were transformed into actual fault-prediction routines. The implementation of the different reliability models is based on the DEG core structures and they are broadly implemented having the same simulation flow as presented in Section 4.1, but also realize the differences between the various components. The design decisions will help to understand the implementation of each reliability model and if necessary, further details will reveal how we managed to satisfy the required features. Since we have different reliability models, the modules had to be adapted to each others' functionality in order to operate properly.

4.4.1 Component with Homogeneous Modules

This component consists of n identical modules operating in parallel plus the necessary control signals and interface circuitry. In order to implement this fault tolerant component we need $2n + 1$ event generators.

- One critical event generator to predict the permanent faults that could occur on the control signals and the interface circuitry with a fault rate $\lambda_{critical}$. Whenever this generator produces an event, all of the $2n + 1$ generators will be deactivated to stop generating events as the component fails.
- n event generators, one for each module with a fault rate $\lambda_{modules}$.
- n NBTI event generators, one for each module with a fault rate λ_{NBTI}. Each n^{th} NBTI generator is tied to the normal n^{th} generator. If a fault occurs on the n^{th} module due to the NBTI effect or due to the normal degradation, both event generators will be disabled to stop generating events on that module.

Whenever all the modules fail or when a critical fault takes place, the component will be in $Failure$ mode, and no further degradation could be performed. The failure rate for the control signals and interface circuitry $\lambda_{critical}$, and the failure rate of each modules $\lambda_{modules}$ are subjected to adjustments by incorporating the utilization rate of each component. Note that components with higher workload have a higher probability to fail. Regarding the failure rate λ_{NBTI}, it changes dynamically in time as a function of the threshold voltage degradation ΔV_{th}. This part is discussed in details in Section 4.3 and it is similar for all the reliability models.

4.4.2 Component Supporting Frequency Scaling

In accordance to what is discussed in Section 3.2, to implement the component that supports frequency scaling with n frequency levels before total failure, we need $2(n+1)+1$ event generators in total:

- One critical event generator to predict the permanent faults that could occur on the component and cannot be repaired by scaling the frequency with a fault rate $(1-c).\lambda$. Whenever this generator produces an event, all the generators will be deactivated to stop generating events as the component fails.
- $n+1$ event generators, each one representing the module working at a specific scaled frequency plus the original frequency with a fault rate $c.\lambda$.
- n+1 NBTI event generators, one for each module with a fault rate λ_{NBTI}. Each n^{th} NBTI generator is tied to the normal n^{th} generator. If a fault occurs on the n^{th} module due to the NBTI effect or due to the normal degradation, both event generators will be disabled to stop generating events on that module.

In the start of component operation, three generators are activated and the rest are disabled:

- 1 critical event generator,
- 1 normal event generator, and
- 1 NBTI event generator

The normal and NBTI event generators correspond to the component operating at the normal frequency. Whenever a fault occurs and a degradation is possible, these two event generators are deactivated, and the next two (normal + NBTI) generators corresponding to the next scaled frequency are activated to be able to produce events. Whenever we reach the predefined number n of tolerable faults, or when a fatal event is produced, the component will be in *Failure* mode, and no further degradation could be done. Regarding the NBTI generators, the first one operating under normal frequency has the threshold voltage points A and B as discussed in Section 4.3. When the next NBTI generators are activated, these threshold voltage points increase when we move to a lower frequency. Note that we adopted the shifting method of the points A and B, because when a component works under lower frequency it can tolerate more gate delays which means higher threshold voltage degradation.

4.4.3 Memory

The memory component is implemented in a generic way. However we considered a specific memory of 4MB in the description above to clarify the implementation. In this particular example, we will need:

- One critical event generator to predict the permanent faults that could occur on the memory that render the whole memory unusable so they cannot be tolerated by degrading capacity with a fault rate $(1-c)\cdot\lambda$,

- 8 event generators, each one responsible to predict a permanent fault in a region of 512KB with a failure rate $c \cdot \lambda$.
- and 8 NBTI event generators, one for each module with a fault rate λ_{NBTI}. Each n^{th} NBTI generator is tied to the normal n^{th} generator. If a fault occurs on the n^{th} module due to the NBTI effect or due to the normal operation, both event generators will be disabled to stop generating events on that module. In addition, the two n^{th} generators (normal and NBTI) are either activated or deactivated together.

The NBTI effect has an impact on the read stability of static random-access memory (SRAM) cells [26]. We considered that this component is an SRAM and that's why we considered faults that could occur due to NBTI effect. Otherwise these NBTI generators have to be turned off. In the beginning, all of the 16 generators are activated. Whenever a permanent fault occurs in one region, the number of generators activated will be reduced to half (half normal, half NBTI generators). Accordingly, at the last stage of graceful degradation, two such event generators related to 512KB will be predicting events. At that point whenever a permanent fault occurs the memory component will be shut down. Note that at any point in time, if the fatal event generator produces an event, the memory component will be denoted as faulty.

4.4.4 Heterogeneous Component

Accordingly, to what is discussed in Section 3.4, to implement the imbalanced component several event generators will be used:

- The L1 cache will be implemented in the same way we implemented the memory component in the Section 4.4.3. However the event generator predicting the uncovered faults in the cache unit will be merged with the generator predicting faults on the essential circuitry. Hence, the fault rate of this generator will be higher than the one in Section 3.4 in order to cover the fault predictions of the essential circuitry.
- One ordinary event generator with low criticality to predict faults that could appear on the integer unit with fault rate λ_{int}. As discussed before, if the floating point unit failed, the fault rate of the integer unit will be affected and be $\lambda'_{int} > \lambda_{int}$.
- Likewise one event generator with low criticality to predict events on the floating point unit with fault rate λ_{FP}. Also, like the integer unit behavior, if the integer unit fails, the fault rate of the floating point will change to $\lambda'_{FP} > \lambda_{FP}$.
- Two NBTI event generators tied one to the integer unit and one to the floating point with fault rate $\lambda_{NBTI_{int}}$ and $\lambda_{NBTI_{float}}$ respectively.

Regarding the integer and floating point units, any generator tied to them that produces a fault, will shutdown the unit and deactivate all the generators attached to it.

4.4.5 Reconfigurable m-by-n Processor Array

To implement the m-by-n processor array in the DEG we will consider these steps:

- populate $m \cdot n$ event generators to predict faults that could appear on each substitutable unit (SU) with its corresponding fault rate $\lambda_{SU(i)}$.
- $m \cdot n$ NBTI event generators, one for each substitutable unit with a fault rate λ_{NBTI}.
- $m - 1$ event generators to predict permanent faults on the interconnect infrastructure with a fault rate of $\lambda_{interconnect}$.
- $m - 1$ NBTI event generators to predict permanent faults affected by the NBTI phenomenon on the interconnect infrastructure with a fault rate of λ_{NBTI}.

Note that λ_{NBTI} is a function of ΔV_{th} that differs from a module to the other as discussed previously. Whenever a fault occurs on a specific part, all the generators related to the parts (normal + NBTI) that belong to the same processor will be disabled and stop predicting events. Furthermore, the reconfiguration routine will be called in order to search for possible graceful degradation. If the reconfiguration module reactivates any disabled functioning part, its generator will be reactivated. Naturally, the system reconfiguration will keep track of all the working parts.

4.4.6 3D-stacked DRAM Hybrid Memory Cube

The 3D-stacked DRAM hybrid memory cube (HMC) is described in Section 3.6. Accordingly, to implement the HMC component, we will need several event generators presented in the list below:

- **4** event generators to predict permanent faults that could appear on the 4 different routers.
- **6** event generators to predict permanent faults that could appear on the 6 links connecting the routers.
- **32** event generators corresponding to the 32 controllers.
- As mentioned before, each vault contains 16 banks. In total we have 16 banks $\cdot$ 32 vaults = 512 banks. Accordingly, we need **512** event generators to predict permanent faults that could appear on these banks.
- Each vault consists of 288 TSV's. In total we have 288 TSV's $\cdot$ 32 vaults = 9216 TSV's in one 3D memory die. Accordingly, we have to populate **9216** different event generators to predict permanent faults on each of these TSV's. As revealed before, each vault could tolerate 2 faulty TSV's before being considered failed.

The processors that use the HMC can have NBTI failures, and these generators can be modeled in a similar manner as in any other NBTI-prone component. Regarding the HMC, we will not explore the NBTI effect on this component. However, we are interested in studying the capacity degradation over time due to faults that appear on different parts. This result will be discussed in Chapter 5.

4.5 Implementation Summary

In this chapter we defined the ported modules from the basic event generator (EG), and upgraded the tool to operate in closed-loop as well to predict events dynamically.

Furthermore, the NBTI effect was modeled in high level of abstraction using a well defined probabilistic predictive model.

In addition, each component was implemented according to its particular characteristics. In the next chapter we will present the experimental results on these components to evaluate the tool.

5

Evaluation and Experiments

This chapter presents a set of experiments which aim to evaluate the implemented dynamic event generator (DEG). We will first study the basic event generator (EG) and do some analysis by producing probability density and distribution function graphs similar to literature to verify that the core of the tool's implementation functions as expected. The second section is to evaluate the ported modules and test their integrity. For this purpose we compare the basic Java modules with the ported C++ ones by running the same scenarios on both environments. The third section is to compare the basic EG and the DEG by showing how dynamic parameters affect the event prediction process. The fourth section explores the Negative-Bias Temperature Instability (NBTI) calibration using a low-level tool as reference and verifies the NBTI impact on a heterogeneous component. In the fifth section we evaluate the HMC and draw a graph for the capacity in time using two different scenarios to observe the impact of failure rates on the memory capacity degradation. The sixth section evaluates the graceful degradation in the processor array component. For that purpose, we considered two different processor arrays with three different failure rates scenarios. The experiment compares the number of operating processors in time for both processor arrays. The last section gives a summary of this chapter.

These experiments have two purposes:

- verify the correct function of the tool as described in Section 5.1 to 5.4
- use it to examine different failure scenarios in complex systems and make observations as shown in Section 5.5 and 5.6

5.1 Analysis of the Basic Tool

In order to understand and analyze the basic tool, we draw the probability density function and the distribution function considering different fault rates per day. Therefore, we generated a simple component and tied one event generator to it, with a specific failure rate λ . The simulation lifetime is 1000 days in steps of one day, and one million iterations of the experiment are enough to predict the probability of failure during this time. Figure 5.1 represents the probability density function of the same component with failure rates λ = (0.004 / 0.006 / 0.008) faults/day. We add to the graph a theoretical curve in black for λ = 0.006 faults/day to evaluate the experimental curve in red and we see that they match well.

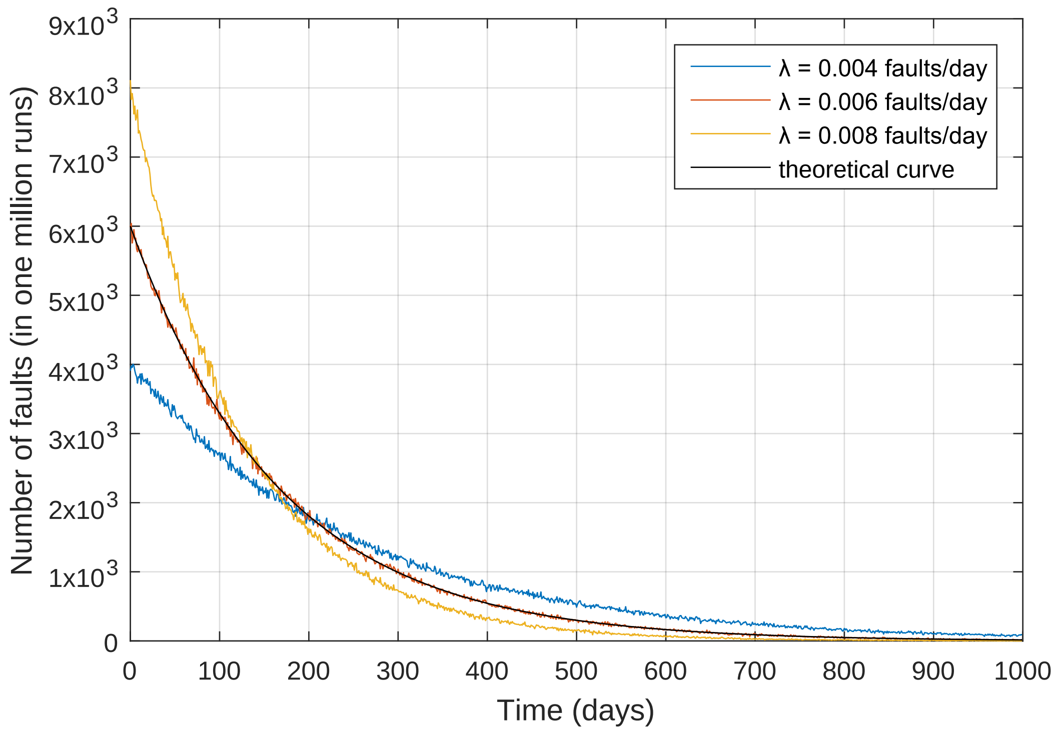


Figure 5.1: Probability density function using the Java basic version

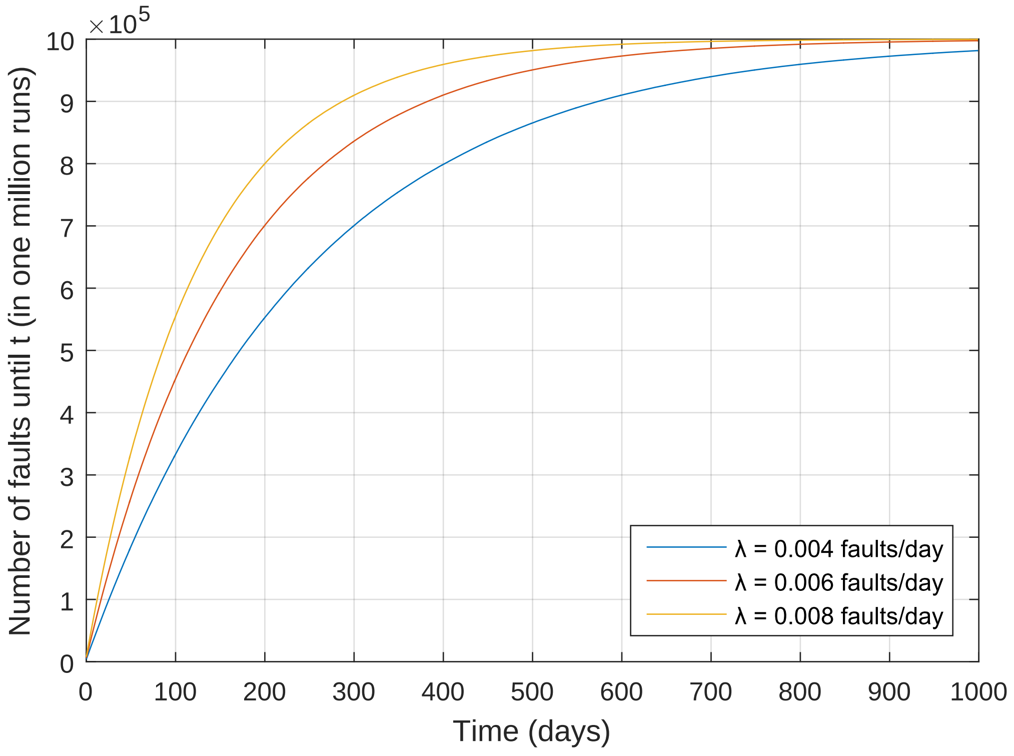


Figure 5.2: Distribution function using the Java basic version

The shape of the curves with different λ's match the definition of the probability density equation 5.1 for a component with an exponential distribution as described in [9].

$$f(t) = \lambda e^{-\lambda t} \tag{5.1}$$

Higher values of λ mean a sharper decrease toward later points in time. Thus, with high values of λ, faults cluster in early stages. On the contrary, low values of λ give a smoother distribution as the case for $\lambda = 0.004$ faults/day. Using equation 5.1 is straightforward to conclude the distribution function represented in equation 5.2 as described in [9], and draw the corresponding graph shown in Figure 5.2.

$$f(t) = 1 - e^{-\lambda t} \tag{5.2}$$

Once more, our plots matched the definition in [9]. Higher failure rates λ mean sharper rise toward later points in time. Thus, most of the one million components tested failed during their lifetime with $\lambda = 0.006$ or $\lambda = 0.008$. However, when λ $= 0.004$, we don't reach a saturation and not all the components have failed the reliability test. Please note that these failure rates λ don't reflect the reality of failure rates for the electronic components, but they were chosen for experimental purpose.

5.2 Evaluation of the Ported Basic Modules

Once the basic modules were ported to C++ language, it was required to evaluate the basic functionality and test the integrity. At this stage, no upgrade was done to the tool functionality other than porting it to another language. We used one of the results acquired in the first simulation and tried to run the same simulation with exactly the same configuration. In the same way, the simulation lifetime is 1000 days, and one million iterations are used to draw the probability density function and the distribution function considering a failure rate of 0.008 faults/day.

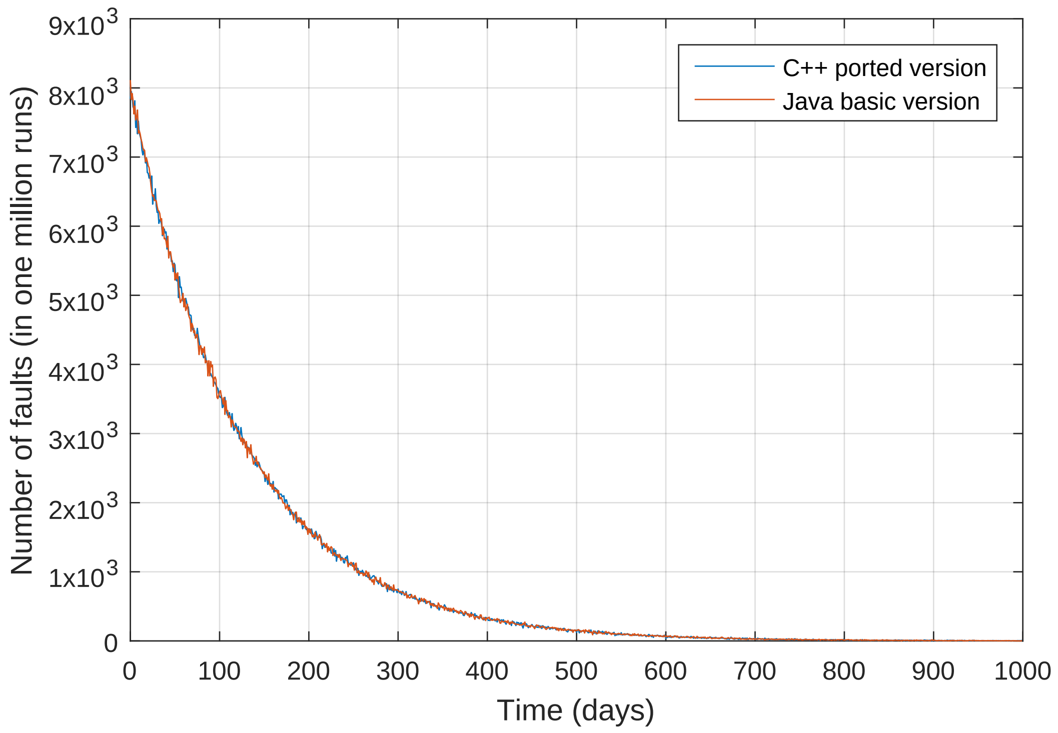


Figure 5.3: Probability density function using Java vs C++

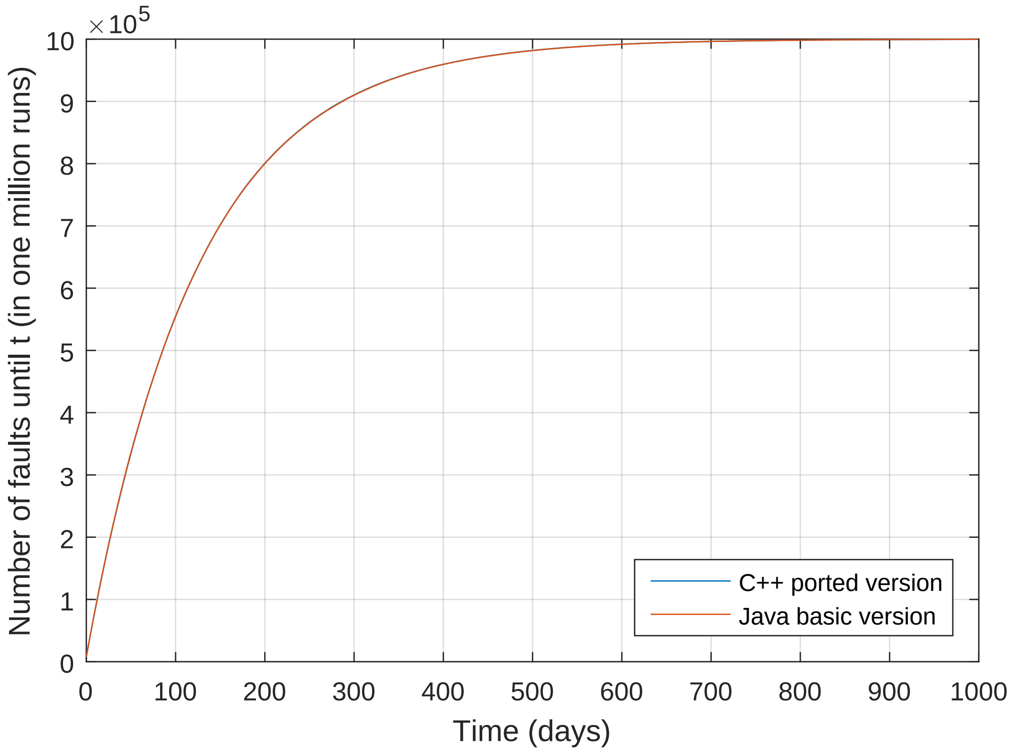


Figure 5.4: Distribution function using Java vs C++

Both Figure 5.3 and 5.4 show how matching are the results, and the curves overlap showing a successful result.

5.3 Comparison between the Basic EG and the DEG Prediction Routine

In this section we did several experiments to compare the basic EG against the DEG. For that purpose, we considered a 4-by-8 processor array component consisting of 8 processors of 4 substitutable units each. We considered the failure rate of each part to be $\lambda_{SU} = 0.0007$ faults/day without taking into consideration the faults that can occur on the interconnects. The simulation time is 1000 days and we run 100 iterations to observe the number of operating processors over time. Figure 5.5 shows a comparison between the basic EG and the DEG without taking into consideration any NBTI effect. In this case the DEG has advantage on its opponent by knowing two different dynamic pieces of information that are used to predict events more accurately and they are:

- information about the hardware configuration showing which part is working, failed or deactivated, and
- information about utilization rate of the processor array.

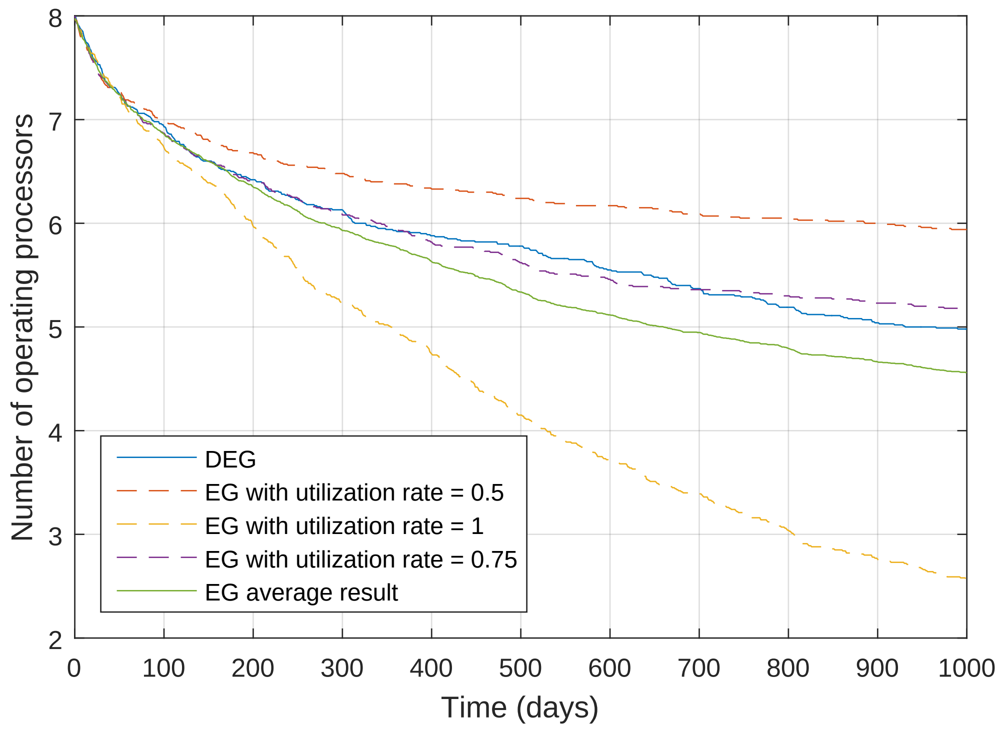


Figure 5.5: Processor array evaluation using the basic EG vs DEG

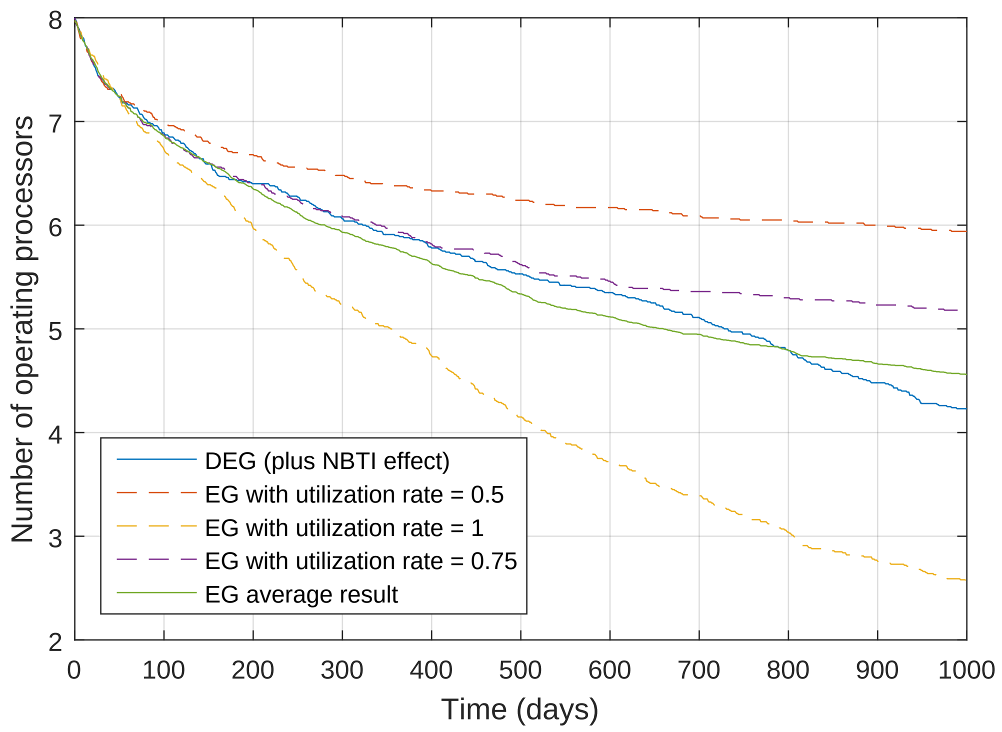


Figure 5.6: Processor array evaluation using the basic EG vs DEG and taking into consideration the NBTI effect

Figure 5.6 shows how the shape of the DEG curve differs even more when taking into consideration the NBTI effect. In this case the DEG predicts events using four dynamic pieces of information that the basic EG doesn't use and these parameters are:

- information about the hardware configuration showing which part is working, failed or deactivated,
- utilization rate of the processor array,
- the steady-state-temperature of the processor array, and
- the NBTI duty cycle which along with the previous parameter is used to calculate the failure rate of the NBTI generator.

Regarding the basic EG, these pieces of information cannot be updated at runtime. However, we allowed it to determine a specific value for the utilization rate of the processor array in the beginning of the simulation and we vary it to be 0.5, 0.75, or 1.0. These values are chosen to represent the spectrum used by the DEG, in which the utilization rate varies between 0.5 and 1.0. Furthermore, we plot the EG average result for the three different utilization rates.

5.4 NBTI Calibration Process and Results

In this section we will discuss NBTI calibration process for the probabilistic model defined in Section 4.3. Furthermore, we will explore the NBTI impact on a heterogeneous component.

Our model is presented in Section 4.3 and illustrated in Figure 4.5. The challenge is to find the corresponding exponential function describing $\lambda_{NBTI} = f(\Delta V_{th})$ and to define the threshold point A and B. Accordingly, we managed to calibrate our high-level predictive model for NBTI degradation using a reference result of lower-level tools [20]. The calibration means determining the parameters of the exponential function (coefficient, base, exponent) and the two thresholds. This calibration serves to integrate the NBTI effect in a particular scenario for a specific component.

The reference low-level tool in our case uses Monte Carlo simulations that treat each transistor independently. That means that a fault on one transistor has no effect on other transistors. Regarding the experiment setup, we based it on a homogeneous component of 300 million transistors. We considered that each NBTI generator is responsible to predict errors on a specific area of the component containing several transistors. Thus, when an NBTI generator produces a fault, certain areas of our component are shut down. As a consequence, we project the faults we observe on the active area of the component to the whole area.

The main goal of this experiment is to showcase a methodology for calibrating our high-level predictive model for NBTI degradation, using as reference results of lower-level tools by obtaining as close results as possible when having the same scenarios for the NBTI effect. The values shown in Figure 4.5 are results of the calibration.

Scenario	**Reference**	**DEG**	**Unit**	**Deviation**
T=363K, α = 0.75, t = 3years	0.8229	0.804	faults/3E+8 transistors	-2.3%
T=383K, α = 0.75, t = 3years	1.08	1.273	faults/3E+8 transistors	+17.8%
T=403K, α = 0.75, t = 3years	1.5	1.542	faults/3E+8 transistors	+2.8%
T=363K, α = 0.95, t = 3years	0.835	1.502	faults/3E+8 transistors	+79.8%

Table 5.1: NBTI calibration results

Table 5.1 contains the calibration results for the NBTI effect with respect to the reference low-level simulation under different scenarios. We ran 250 simulations as this is the number of simulations in the reference experiment and draw a mean value for each scenario. When varying the temperature to be 363K, 383K, and 403K under a fixed NBTI duty cycle $\alpha_{NBTI} = 0.75$, we could calibrate the DEG to have reasonable deviation from low-level results. However, with high NBTI duty cycle $\alpha_{NBTI} = 0.95$, we get a significant difference between the reference tool and the DEG. The result of the reference tool considering the same temperature $T = 363K$ and an operating time of $t = 3$ years shows that the number of faults per 300 millions transistors for $\alpha = 0.75$ is **0.8229** and for $\alpha = 0.95$ is **0.835**. These numbers are very close. However, according to the literature on which we based our models implementation [25][28], there is an exponential dependence between degradation and duty cycle, so we think that this result in the reference simulation is unexpected and not in line with this trend.

To sum up our goal is not to compare the tools and define which one is more accurate, rather than showing the possibility that our tool could be calibrated to produce results comparable to lower-level tools which utilize much more fine-grain information.

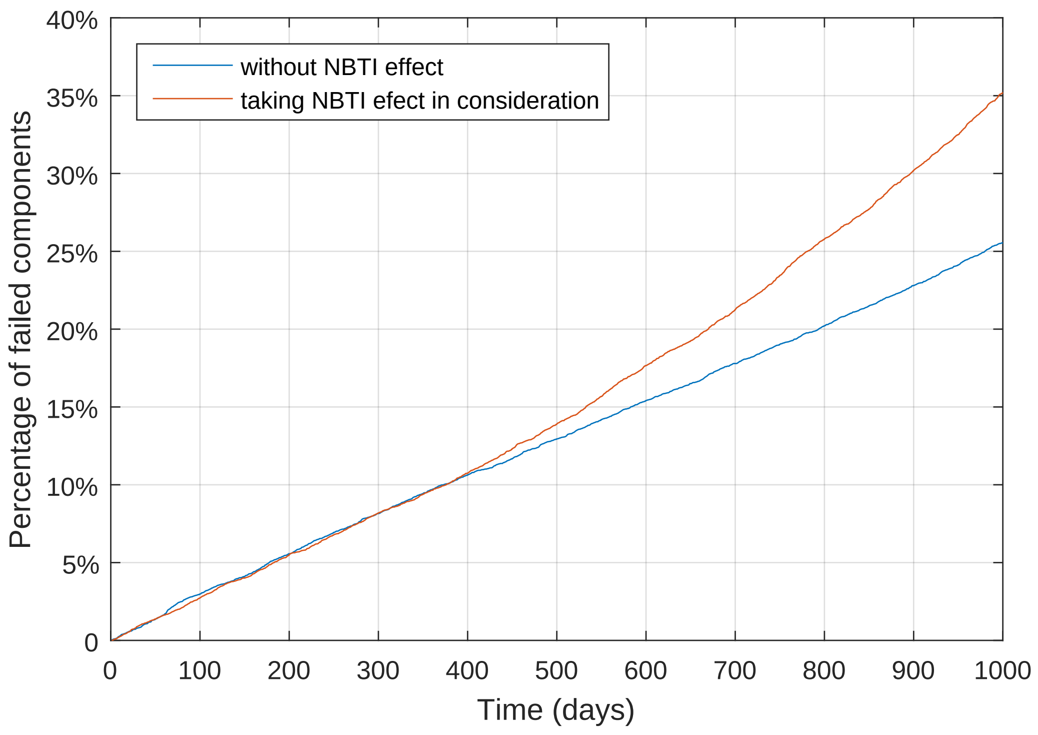


Figure 5.7: NBTI impact on Heterogeneous component

To explore the NBTI impact, we considered an heterogeneous component with a floating point unit, integer unit and a cache divided into 8 regions 512KB each as the one discussed in Section 3.3 plus the circuitry. The utilization rates for each part were randomized and we run the simulation for 10,000 components and see how many components fail after 1000 days. We considered two different scenarios: taking or not taking into account the faults that the NBTI generators could produce. The results are shown in Figure 5.7. The graph shows that after almost 3 years the NBTI effect will result in 10% more components failing. Furthermore, with NBTI not only we have more failures, but they start becoming more later in time, indicating the impact of the aging effect.

5.5 HMC Evaluation

To study the capacity degradation over time of the 3-D stacked DRAM Hybrid Memory Cube (HMC), we considered running two different scenarios. One with normal failure rates $\lambda_{nomral(i)}$ for the different parts and one with more pessimistic failure rates $\lambda_{pessimistic(i)}$. The considered failure rates for both scenarios are shown in Table 5.2.

Part	Baseline λ [faults/day]	Pessimistic λ [faults/day]
Bank	$100 \cdot 24 \cdot 10^{-9}$	$150 \cdot 24 \cdot 10^{-9}$
TSV	$35 \cdot 24 \cdot 10^{-9}$	$50 \cdot 24 \cdot 10^{-9}$
Link	$40 \cdot 24 \cdot 10^{-9}$	$60 \cdot 24 \cdot 10^{-9}$
Router	$400 \cdot 24 \cdot 10^{-9}$	$600 \cdot 24 \cdot 10^{-9}$
Controller	$50 \cdot 24 \cdot 10^{-9}$	$100 \cdot 24 \cdot 10^{-9}$

Table 5.2: Fault rates used in the HMC simulations

The capacity of the HMC is determined by the total fault free operating banks. The size of each bank is bounded to the current technology being 16 MB. Each vault contains 2 banks in each layer. Thus there are 16 banks in each vault as shown in Figure 3.12. As a HMC is composed by 32 vaults, the total capacity in the fault-free case is 16 MB·16·32 = 8 GB. With a number of 100 iterations and a simulation life time of 10 years in steps of one month we deduced the mean value of the capacity for each month during the simulation. At any specific time in order to know the capacity of the HMC it is enough to be aware of the number of fault free operating banks. Thus,

$$Capacity = \frac{No.\ of\ fault\ free\ banks}{total\ No.\ of\ banks} \cdot 8GB \tag{5.3}$$

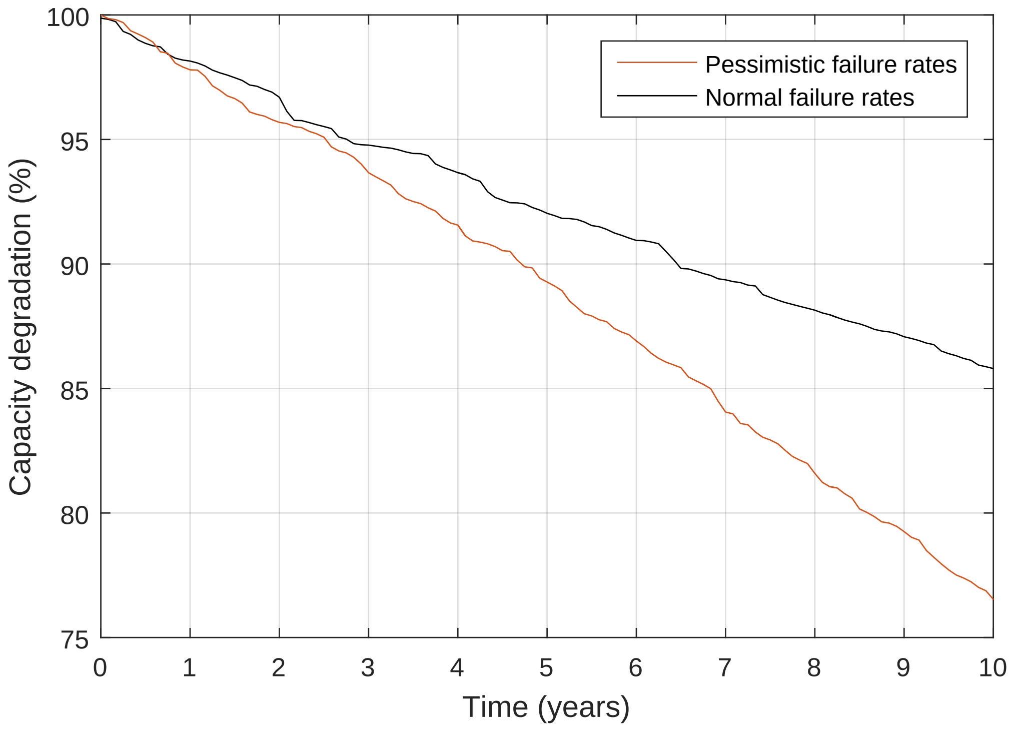


Figure 5.8: HMC capacity degradation in time

Figure 5.8 illustrates the capacity degradation percentage considering the two scenarios presented in 5.2. With the normal failure rates for the different parts, the HMC remaining capacity after 10 years is 85.8% $\approx$ 6.86 GB. However, for the pessimistic scenario, the remaining capacity after 10 years is 76.53% $\approx$ 6.12 GB. Furthermore, considering the pessimistic failure rates, the red curve has a sharp decreasing slope whereas for the normal failure rates, the blue curve has smoother decreasing slope.

5.6 Processor Array Evaluation

In this section we will evaluate the graceful degradation algorithm of the processor array component. To do that we considered two different processor arrays to compare:

- 4-by-8 processor array consisting of 8 processors of 4 substitutable units each, and
- 8-by-7 processor array consisting of 7 processors of 8 substitutable units each

The reconfigurability incurs some area penalty, which grows with the number of SUs per processor, because we have more interconnection elements between these substitutable units as shown in Figure 3.8. This is why the coarser-grained 4-by-8 processor array has more processors (eight) than the finer-grain, which has seven. Furthermore, we set the failure rates of each substitutable unit in such a way that the total failure rates of the two processor arrays are equal.

Scenarios	**4-by-8 processor array**	**8-by-7 processor array**
baseline λ_{part} (faults/day)	0.0007	0.0004
scaled down λ_{part} (faults/day)	0.00042	0.00024
scaled up λ_{part} (faults/day)	0.000933333	0.000533333

Table 5.3: Fault rates used in the processor array simulations

To evaluate these two processor arrays, we ran 100 simulations considering the scenarios shown in Table 5.3 and observe the number of operating processors over time.

Figure 5.9 shows the processor array degradation using the baseline failure rates whereas Figure 5.10 shows the degradation with failure rates scaled down by a factor of 3/5 compared to the baseline scenario, and Figure 5.11 uses scaled up failure rates by a factor of 4/3 compared to the baseline ones. Note that for the purpose of these simulations, we did not consider the failure of the interconnects between the columns as described in Section 3.5. The trend of all the graphs is similar disregarding the failure rates. We could clearly see that after a specific point in time, the 8-by-7 processor array has higher number of operating processors than the 4-by-8 processor array. This flipping point is inversely proportional with respect to the failure rates of the parts.

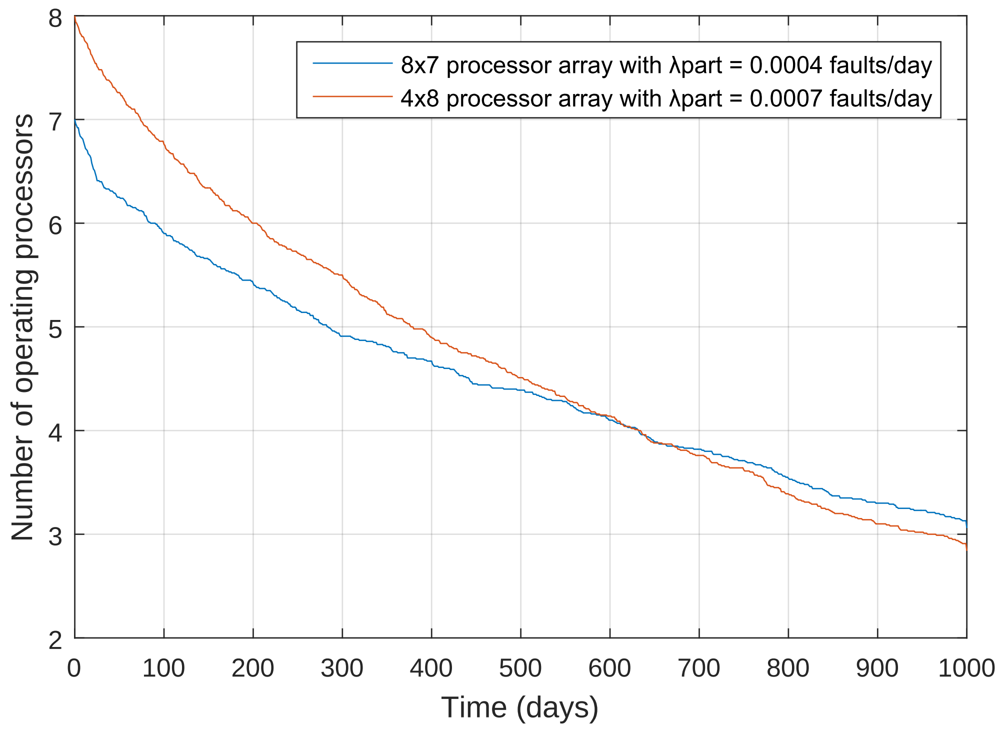


Figure 5.9: Processor array degradation with baseline failure rates

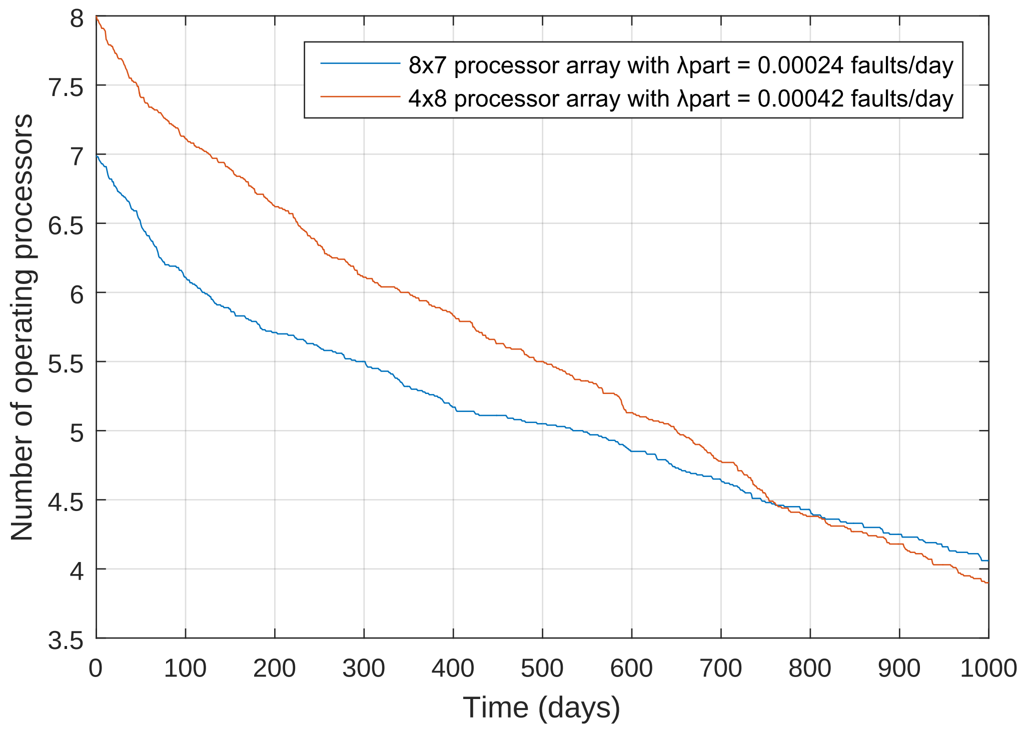


Figure 5.10: Processor array degradation with scaled down failure rates

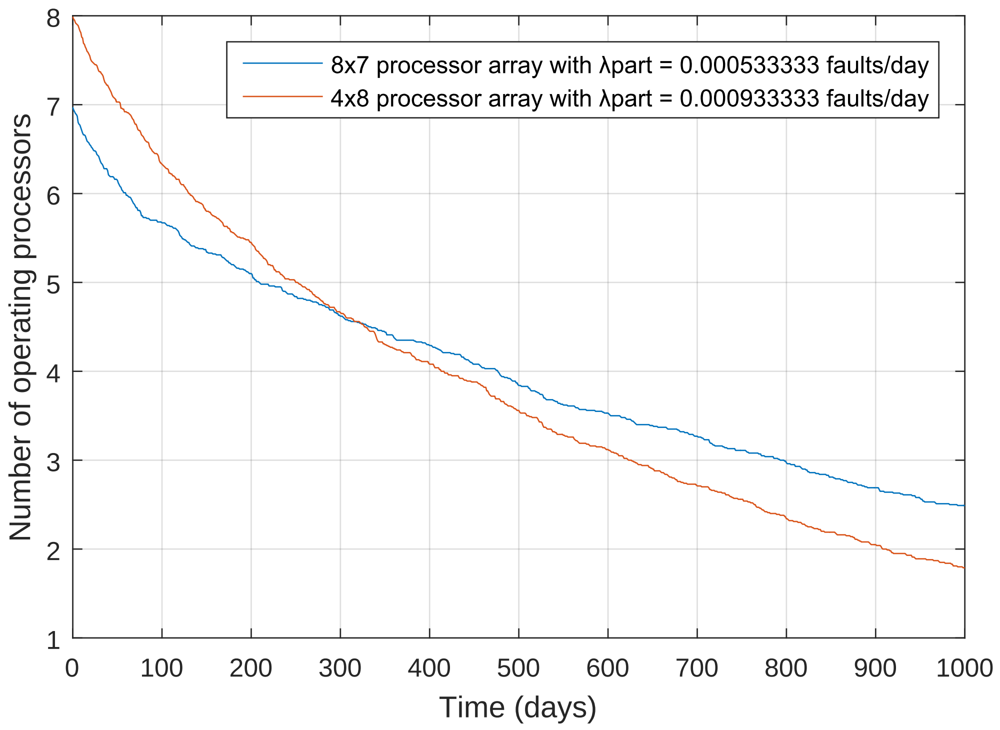


Figure 5.11: Processor array degradation with scaled up failure rates

We can conclude that the 8-by-7 processor array is preferable to operate under long period as this architecture provides more processors for higher operating duration when the failure rates are high. If the failure rates were lower, the 4-by-8 processor array would be preferable even for longer times. However if the mission of this processor array is short, the 4-by-8 architecture has almost one extra processor than the other architecture considering the different tested failure rates.

The fact that the 8-by-7 architecture is better for long period operation comes from the flexibility that this type of processor array provides. Indeed, this architecture has more columns and each permanent fault removes a smaller part of the system. Furthermore, having higher number of parts, the reconfiguration algorithm will easier find a way to build processors from unused parts.

Summing up, both the failure rates and the maximum lifetime in the experiment are indicative (the rates, in fact, are too high), but the bottom line is that for every failure rate (depending on the technology) the tool can be used to perform a similar analysis and motivate a decision about which of the reconfiguration granularities is better for a particular mission time.

5.7 Chapter Summary

In this chapter we verified that the tool works as intended by evaluating the ported basic modules, by comparing the basic EG fault prediction process versus the DEG one, and by exploring the NBTI calibration and impact.

Furthermore, we used the DEG to make useful observations about the graceful degradation of two complex systems, processor array and HMC.

In the next chapter we will draw conclusions based on these experiments and the rest of the thesis.

6 Summary and Conclusions

Integrated circuit device scaling is causing reliability problems. However, reconfigurable components along with run-time support give the opportunity to deal with permanent faults by degrading one or more system aspects instead of failing [1] [2] [3]. In this manner graceful degradation assures a recovery mode and extends the lifetime of the system. Another serious problem in modern CMOS technology is the oxide wearout mechanisms that occur at the transistor level [7]. One such aging mechanism is the effect of Negative-Bias Temperature Instability (NBTI) and it is considered as one of the major reliability concerns of any component and should be taken into account in realistic reliability studies. Our goal was to upgrade an existing basic tool that generates faults and integrate it in closed-loop with the rest of the experimental setup to study reliability of degradable systems. Furthermore, we incorporate the most important aging effect NBTI in the tool and integrate the duty cycle parameter in our methods that predict events to get more precise results that reflect reality.

This Chapter will summarize and conclude the thesis and will lastly open up new horizons for future researchers. Section 6.1 summarizes the previous chapters, while Section 6.2 discusses the main findings and contributions. Finally Section 6.3 list the possible future prospects.

6.1 Thesis Summary

Chapter 1 reveals the motivation that comes from existing works on runtime optimization algorithms for the management of a gracefully degradable adaptive MPSoC [1] [2] [3]. The problem was to implement a tool that works in closed loop with the experimental setup in order to inject faults dynamically, and update parameters every time the configuration changes. In this Chapter we set the thesis objectives and mention the limitations.

In Chapter 2 we focus on presenting the theoretical background and define notions from literature related to our work. Furthermore we discuss relevant works related to the lifetime extension strategies and give a definition of the NBTI effect.

Chapter 3 presents the models for various degradable components in order to set them up in the dynamic event generator DEG and study their reliability. We give a brief description for each component and list the possible faults that can occur on it during its lifetime. Moreover, we define what are the degradable aspects whenever a fault occurs and reveal the possible degraded modes.

Chapter 4 focuses on the design and implementation of the DEG tool. The main decisions in the design are justified and how we managed to upgrade the tool to operate in closed loop and predict events dynamically. Since a reconfiguration module is not available for every scenario we studied, we also implemented a mock version of the reconfiguration module, to be used for debugging and verification of our tool, and in the cases such a complete module was not available. Regarding the NBTI implementation, we show how we managed to go from transistor level to logic circuit level and reveal the design decisions made. Moreover, we discussed the design and implementation of all the reliability models defined in Chapter 3.

Chapter 5 consists of the experiments done to evaluate the implemented DEG tool. We analyzed the basic EG tool, then we compared the ported DEG modules with the basic tool and showed that they match. Then we compared the basic EG with the DEG showing how dynamic parameters affect the faults prediction routine. Furthermore, we showed how robust and flexible our tool is by calibrating it to another low-level NBTI prediction method and studying the NBTI impact on an heterogeneous component. Then we studied the HMC capacity in time considering two different scenarios. Finally, we evaluate the processor array graceful degradation in time using two different types of processor arrays.

6.2 Thesis contributions

In this section we highlight the contributions of this work and go back to the thesis objectives stated in Section 1.2.1 to show how we achieved what we set out to do. The results are listed as follow:

- The upgrade of the basic tool to work in closed-loop with the rest of the experimental setup optimized the tool and made it more accurate when predicting faults using different reliability models. As the duty cycle and NBTI parameters received from the reconfigurations module change during time, they gave a higher precision when estimating the different failure rates of the generators.
- The variety of the sophisticated fault models implemented in the DEG reflects the generality of this tool and shows how easy it is to study the reliability of any adaptive degradable system and draw results and conclusions. However, in this work we study a limited number of various components after designing their fault models and defining their graceful degradation modes.
- The DEG is flexible as we can study some degradable aspects of the system in time, and draw results. For instance we used the tool to study the degradation capacity of the HMC in time, and we observed that the remaining capacity after 10 years is **85.8%** for the normal failure rates.
- We used the DEG tool to evaluate the graceful degradation of the processor array and we observed that for a given fault rate and mission time, the tool can be used to decide which reconfiguration granularity is best.
- The incorporation of the NBTI aging effect in our high-level DEG tool, shows the possibility that we can calibrate the DEG with the use of experiments done with low-level tools. Note that we developed a predictive model that doesn't need low-level information but of course is not as precise as the lower-level models. This incorporation opens the door in front of future upgrades of the

DEG by adding other wear-out effects in order to get higher precision in the fault prediction method.

6.3 Possible areas for future research

This study has arrived to the implementation of a robust and flexible tool (DEG) that predicts faults dynamically. This tool is an upgraded version of its predecessor the basic event generator. The possible areas for future research are the following:

- Further generalization of the tool, allowing for easy extension to components others than the ones presented in this thesis.
- The DEG could be even further upgraded by incorporating other oxide wear-out effects like Time-Dependent Dielectric Breakdown (TDDB) or Hot Carriers that can improve the faults prediction.
- The NBTI module could be calibrated by using as reference results from any sophisticated low-level tools. We managed to do the calibrating to be applied to a specific component. A refinement of the NBTI predictive model and better (more generic) calibration, implies that the tool can be used for any component without effort.

Bibliography

[1] S. Tzilis, I. Sourdis, Chalmers University of Technology, Department of Computer Science and Engineering, Computer Engineering (Chalmers), Institutionen för data- och informationsteknik, Datorteknik (Chalmers) and Chalmers tekniska högskola. A runtime manager for gracefully degrading SoCs. 2014, . DOI: 10.1109/DFT.2014.6962106.

[2] S. Tzilis. Graceful degradation of adaptive multiprocessor systems on a chip. 128.2015.

[3] S. Tzilis, I. Sourdis, V. Vasilikos, D. Rodopoulos, D. Soudris, Runtime Management of Adaptive MPSoCs for Graceful Degradation. to be published in CASES 2016, Pittsburgh, October 2-7 2016

[4] M. Glaß, M. Lukasiewycz, C. Haubelt and J. Teich. Incorporating graceful degradation into embedded system design. 2009, .

[5] M. Imai, T. Nagai and T. Nanya. Pair and swap: An approach to graceful degradation for dependable chip multiprocessors. 2010, . DOI: 10.1109/DSNW.2010.5542608.

[6] R. Rodrigues and S. Kundu. On graceful degradation of chip multiprocessors in presence of faults via flexible pooling of critical execution units. 2011, . DOI: 10.1109/IOLTS.2011.5993813.

[7] V. G. Rao and H. Mahmoodi. Analysis of reliability of flip-flops under transistor aging effects in nano-scale CMOS technology. 2011, . DOI: 10.1109/ICCD.2011.6081439.

[8] G. Smaragdos, D. A. Khan, I. Sourdis, C. Strydis, A. Malek and S. Tzilis. A dependable coarse-grain reconfigurable multicore array. 2014, . DOI: 10.1109/IPDPSW.2014.20.

[9] N. Storey. Safety-Critical Computer Systems 1996.

[10] V. Vasilikos, G. Smaragdos, C. Strydis, I. Sourdis, Chalmers University of Technology, Department of Computer Science and Engineering, Computer Engineering (Chalmers), Institutionen för data- och informationsteknik, Datorteknik (Chalmers) and Chalmers tekniska högskola. Heuristic search for adaptive, defect-tolerant multiprocessor arrays. ACM Transactions on Embedded Computing Systems (TECS) 12(1s), pp. 1-23. 2013. . DOI: 10.1145/2435227.2435240.

[11] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. 2012, . DOI: 10.1109/VLSIT.2012.6242474.

[12] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron and M. R. Stan. HotSpot: A compact thermal modeling methodology for early-

stage VLSI design. IEEE Transactions on very Large Scale Integration (VLSI) Systems 14(5), pp. 501-513. 2006. . DOI: 10.1109/TVLSI.2006.876103.

[13] C. Bolchini, M. Carminati, M. Gribaudo and A. Miele. A lightweight and open-source framework for the lifetime estimation of multicore systems. 2014, . DOI: 10.1109/ICCD.2014.6974677.

[14] S. Feng, S. Gupta, A. Ansari and S. Mahlke. "Maestro: Orchestrating lifetime reliability in chip multiprocessors," in Anonymous 2010, . DOI: 10.1007/978-3-642-11515-8_15.

[15] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. IEEE Micro 25(6), pp. 10-16. 2005. . DOI: 10.1109/MM.2005.110.

[16] T. Saridakis Design patterns for graceful degradation. ; 2009.

[17] W. Wang, S. Yang, S. Bhardwaj, R. Vattikonda, S. Vrudhula, F. Liu and Y. Cao. The impact of NBTI on the performance of combinational and sequential circuits. 2007, . DOI: 10.1145/1278480.1278573.

[18] W. Wang, S. Yang, S. Bhardwaj, S. Vrudhula, F. Liu and Y. Cao. The impact of NBTI effect on combinational circuit: Modeling, simulation, and analysis. IEEE Transactions on very Large Scale Integration (VLSI) Systems 18(2), pp. 173-183. 2010. . DOI: 10.1109/TVLSI.2008.2008810.

[19] N. H. E. Weste and D. M. Harris. Integrated Circuit Design (4th, Global ed.) 2011.

[20] Testing degradation due to NBTI, internal CE division document.

[21] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari and J. Torrellas. VARIUS: A model of process variation and resulting timing errors for microarchitects. IEEE Transactions on Semiconductor Manufacturing 21(1), pp. 3-13. 2008. . DOI: 10.1109/TSM.2007.913186.

[22] K. Agarwal and S. Nassif. Characterizing process variation in nanometer CMOS. 2007, . DOI: 10.1145/1278480.1278582.

[23] N. Kimizuka, T. Yamamoto, T. Mogami, K. Yamaguchi, K. Imai and T. Horiuchi. The impact of bias temperature instability for direct-tunneling ultra-thin gate oxide on MOSFET scaling. 1999, . DOI: 10.1109/VLSIT.1999.799346.

[24] D. K. Schroder and J. A. Babcock. Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing. Journal of Applied Physics 94(1), pp. 1. 2003. . DOI: 10.1063/1.1567461.

[25] S. Bhardwaj, W. Wang, R. Vattikonda, Y. Cao and S. Vrudhula. Predictive modeling of the NBTI effect for reliable design. 2006, . DOI: 10.1109/CICC.2006.320885.

[26] S. Kumar, C. Kim and S. Sapatnekar. Impact of NBTI on SRAM read stability and design for reliability. 2006, . DOI: 10.1109/ISQED.2006.73.

[27] W. Wang, V. Reddy, A. T. Krishnan, R. Vattikonda, S. Krishnan and Y. Cao. Compact modeling and simulation of circuit reliability for 65-nm CMOS technology. IEEE Transactions on Device and Materials Reliability 7(4), pp. 509-517. 2007. . DOI: 10.1109/TDMR.2007.910130.

[28] D. Zoni and W. Fornaciari. A sensor-less NBTI mitigation methodology for NoC architectures. 2012, . DOI: 10.1109/SOCC.2012.6398329.

[29] B. W. Johnson. Design and Analysis of Fault-Tolerant Digital Systems 1989.

A Appendix

A.1 User Guide

To facilitate the use of the DEG, we introduce a configuration module named "config.cpp" containing all of the simulation parameters that could be modified by the user, and serves as initial configuration for the system. This file contains the following variables:

- **g_total_components** is the total number of components.
- **g_BAL, g_FREQ, g_MEM, g_FLOAT_INT, g_PROC_ARRAY, g_HMC** represent how many balanced, frequency scaling, memory, float_integer, processor array, and HMC components respectively we have in a specific simulation.
- **g_sim_length** defines the simulation time in number of steps.
- **g_fileIndex_limit** defines the number of iterations we want to run in one simulation with the same initial configuration.
- **g_print_option** is a display parameter; whenever it is set to 1 the output will be printed in console, otherwise in files.

Furthermore, for every reliability model there is a set of parameters that assign the fault rates for all the generators plus the corresponding generic parameters. For example for the balanced component with homogeneous parts there is a variable named *g_bal_modules* that assigns the number of modules to be implemented for this kind of components. Similarly for the m-by-n processor array, m and n could be preset by the user using *g_m_array* and *g_n_array* variables.