RZ 3568 (# 99578) 10/01/04
Electrical Engineering 76 pages

Research Report

MoHiDoC: Modular Hierarchical Diagnosis on Chip*

Maria Gabrani and Dominique Tschopp

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
E-mail: mga@zurich.ibm.com

*Work performed by Dominique Tschopp for his EPFL Diploma Thesis while at the IBM Zurich Research Laboratory under the technical supervision of M. Gabrani.

Contents

List of Figures

List of Tables

Abstract

With ever shrinking geometries, higher density circuits and higher frequencies, soft errors in logic are expected to become a great concern for chip design, operation and maintenance in upcoming years. This kind of errors occurs randomly and unpredictably. Further, an error in a component on a System-on-Chip (SoC) could potentially affect the entire component or even the entire chip if no adequate actions are taken. Possible consequences include erroneous states, corrupted data that challenge data integrity or system failure. Mechanisms exist for detecting single or multiple bit errors in logic circuits, such as predictive coding, parity checks or code replication. Other techniques, such as ECC or time redundancy, make recovery possible. However, we argue that those methods can only deal with Single Event Upsets (SEU) locally, are expensive, particularly if implemented at each component and ignore the global system state, consequently neglecting issues such as error propagation. We suggest that a new approach should make a chip aware of its state and capable of autonomously reacting in case an error occurs.

To achieve the above, we propose a hierarchical modular framework. We first model the I/O functionality, state determination, and potential error detection, analysis and reaction of a building block (BB). Second, we map the system functionality into logical BBs; ie, a BB may correspond to a component or a part of a component. For communication between BBs we rely on a Petri Net (PN) structure. More precisely, we map the BBs to transitions in the PN and introduce token places as interfaces between BBs. As inferred above, there are additional higher layers in the PN that allow different levels of information, analysis and control. The higher layer PN(s) can correspond to a centralized system-level control block and/or to off chip higher-layer control. Again, the logical BBs of the higher layer functionality are mapped into transitions and token places are the interfaces between layers. Tokens carry the regular data and structured control data attached to it. The analysis mechanism of the BB in the different PNs uses the control data to estimate the state of the component (lower layer PN) or the system (higher layer PN) and to decide on the reaction mechanism(s) which it propagates again through the control data of the tokens. We believe that our approach offers the chip designer and tester a lot of flexibility while enabling runtime system error recovery and unified interfaces to higher layers (e.g., OS or service processor) for system maintenance.

In order to estimate the value of the framework, we develop a mathematical model of costs versus benefits, compare it to previous work and run simulations.

Chapter 1

Introduction

This introductory chapter presents this Diploma Thesis. Further we justify the need for such a work, by giving a brief overview of the problem we focus on. To do so, we start by defining what soft errors in the combinatorial logic, also called logic single events, are. We continue by emphasizing why they will become a great concern for chip design, operation and maintenance in the upcoming years. We also give some useful definitions for a clearer understanding of this report and at the end of this chapter we give a brief overview of existing error handling mechanisms at different levels of abstraction (gate level, component wide, system-wide).

1.1 Thesis

> The Goal of this Diploma Thesis is to design a chip-level system-wide framework for distributed error detection and analysis offering reaction at different layers. The framework should propose standardized structures for interfacing and communication between logical entities. The kind of errors we focus on are single event upsets in combinatorial logic. They represent an increasing threat and we argue that there is a need for a global reaction structure. We call our framework MoHiDoC, which stands for **Mo**dular **Hi**erarchical **D**iagnosis **O**n **C**hip.

1.1.1 Environment

This Diploma Thesis was written at the IBM Zuerich Research Laboratory, in the I/O Server Networking Group. The technical supervisor was Dr. Maria Gabrani and the Manager Dr. Ton Engbersen. The supervising Professor, from the Swiss Federal Institute of Technology (EPFL) in Lausanne was Prof. Paolo Ienne. This project is intended to be integrated in a larger project of Diagnosis on Chip (DoC), an ambitious research project to make

chips more autonomous.

1.1.2 Document Organization

In the next chapter (Chapter 2), we will present the MoHiDoC framework and its different features by first explaining the concepts we rely on and then introducing the different aspects of the framework. In Chapter 3 we will then present a mathematical approach to the modeling of our framework, by deriving a cost versus benefit estimation. Before concluding in Chapter 5, we compare our approach to existing approaches and apply values from the literature as well as from simulations to our mathematical model in Chapter 4.

1.2 Single Event Upsets

Soft errors in chips, also called Single Event Upsets (SEU), result from the strike of an energetic particle. NASA defines SEUs as:

> Radiation-induced errors in microelectronic circuits caused when charged particles (usually from the radiation belts or from cosmic rays) lose energy by ionizing the medium through which they pass. [NASA Thesaurus]

SEU are transient soft errors and are non-destructive. A reset or rewriting results in normal device behavior thereafter. SEUs typically appear as transient pulses in logic or support circuitry, or as bit flips in memory cells or registers. Also possible is a multiple-bit SEU, in which a single ion hits two or more bits causing simultaneous errors [EAS]. The particles causing those upsets are common in the natural space environment, ranging from neutrons and protons to large atomic nuclei (cosmic particles). In terrestrial applications the particles typically originate in the normal radioactive decay of integrated circuit packaging materials or are created by interaction between cosmic neutrons and atoms in the atmosphere. The phenomenon has been observed to be stronger in higher altitudes and/or closer to the poles.

1.2.1 Threat

When a particle strikes the combinatorial logic node, according to [KJBM98], it can create a temporary voltage disturbance at that node. If the voltage disturbance propagates to a latch and occurs near the clock edge, then the disturbed state may be loaded into the latch, causing the stored data to be incorrect just as if the latch itself were struck by a cosmic ray and changed state. In Figure 1.1, the correct state of the I input is a 0, but a momentary disturbance to the 1 level is latched and causes a SEU. In this thesis, we focus on SEU in the logic, which are called Logic SEU, abbreviated LSEU.

After an erroneous bit has been latched up, causing a LSEU, corrupted data is temporarily stored. This error will propagate out of the latch and affect other logic elements. An error in a component on a system on chip could potentially affect the entire component or even the entire chip if no adequate actions are taken. Possible consequences include erroneous states, corrupted data that challenge data integrity, or even system failure. A data integrity problem is introduced when corrupted data is sent to higher layers (e.g. application) that can cause the whole application to crash. This could be particularly damaging if critical data gets corrupted or, for instance wrong, though legal data is sent to another business.

A good example of how damaging such an LSEU could potentially be is the following: let us assume that an address is calculated in some logical circuitry to determine where an incoming network packet should be stored. Let us also assume that during the calculation, a particle strikes the logic causing an erroneous though perfectly legal address (in the sense that the address exists) to be output. Subsequently, the network packet will be stored at a wrong location, potentially overwriting sensitive data. One could imagine numerous other scenarios leading to potentially catastrophic situations.

In the subsection below, we will explain why LSEU were often if not always ignored in the past and why the chip community is slowly starting to consider them as a very serious threat.

1.2.2 Combinatorial Logic Vulnerability

LSEU have always existed, and will always exist. In the past, however, they were never considered a serious threat and were completely ignored. Not withstandingly, a lot of attention was paid to SEU in memory, as memory elements were considered much more vulnerable and critical. The situation is changing however. The chip technology tends toward increased frequency, higher density circuits and lower voltages. Those technical factors make logic

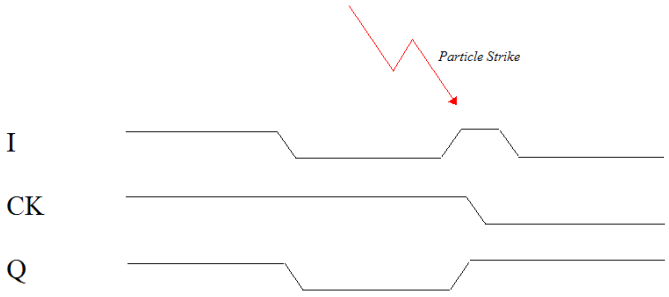


Figure 1.1: Transient disturbance causing SEU

circuits much more subject to LSEU for the following reasons:

- higher frequencies: As explained above, transient pulses can latch up if they propagate and occur close to the clock edge, consequently, higher frequencies tend to increase the risk of having such an incident
- higher density: Particles are more likely to strike the combinatorial logic in higher density circuits, cables being closer together are more likely to be hit (crosstalk).
- lower voltage: reduced nodal capacitance and reduced switching energy imply that lower charges can disturb the behavior of the circuit.

The rate of soft errors in logic is even expected to exceed that of soft errors in memory elements [BBB+97]. Recent publications [Has99, RV04] point out that threat and estimate it. Other publications present means to simulate [Sri96, MS96], and avoid or mitigate LSEU [KJBM98], by for instance hardening gates. It is obvious that in next generation chips, LSEU cannot be neglected anymore.

1.3 Definitions

In this section we give some useful definitions of terms commonly used throughout this thesis:

- *System*: designates a chip in this thesis.
- *Components*: are parts of a system-on-chip with similar or different functionalities. Typically, components are linked together with a bus. On system-on-chip handling network packets, two components could be a receive processor and a send processor. Different types of components could for instance be processor cores, bus arbiters, memory controllers, special function component etc..
- *Error Reaction*: is a generic term which encompasses all actions taken after an error was analyzed. For instance, *Error Correction* and *Error Recovery* (defined in this section) are both *Error Reactions*.
- *Error Correction*: is a type of *Error Reaction* which does not preserve the data affected by an error. Examples of such a mechanism would be repetitions of operations which went wrong or deletion of corrupted data.
- *Error Recovery*: is a type of *Error Reaction* which preserves the affected data, i.e. retrieves the correct data out of the corrupted data. An example of such a mechanism is Error Correcting Codes (ECC).

- "*Service*" *Processor* (*SP*): is an *off-chip* processor. This processor can influence the behavior of a chip it is linked to (e.g reboot).
- *Single Event Upsets* (*SEU*): are defined in Section 1.2. In the sequel, we will use the terms Single Events Upsets (SEU) and Soft Errors (SE) interchangeably. Note that LSEU stands for an SEU in the combinatorial logic (similarly LSE stands for a SE in the combinatorial logic).
- $Self - Healing\ Systems$: are capable of autonomously detecting, analyzing and reacting to errors.
- $System - on - chip$ (*SoC or SOC*): is an idea of integrating all components of a computer system into a single chip. A typical computer system consists of a number of integrated circuits that perform different tasks. These are: microprocessors, RAM, ROM, UARTs, parallel ports, DMA controller chips, etc. The recent improvements in semiconductor technology caused that VLSI integrated circuits can contain an increasingly larger number of components. These improvements allow integration of all the functions of a system in a single chip, which is being done in a number of technologies (Full-custom, Standard cell, FPGA)[WIK].

1.4 State of the Art

While error detection and recovery are well known (e.g., parity codes and ECC), they are mainly known for single event upsets (SEU) on storage circuits (e.g., memory). In logic circuit the need for error detection and correction was not considered important until recently. With the ever shrinking geometries, higher density circuits and higher frequencies, soft errors in logic are expected to become a great concern at or around the 2006 frame, as explained above. So, the area of logic SEU (LSEU) is gaining a lot of attention quickly.

In the sequel, we will first present existing gate level error mechanisms, not developed for LSEU but applicable to them in our sense. Secondly, we will introduce two component wide error handling mechanisms. Again, those approaches were not conceived to handle LSEU but as we will see in this section and in Chapter 4, there are some similarities. Finally, we will briefly explain what is done at the system level to handle errors, since our intent is to define a system-wide framework. Unfortunately, while new methods for LSEU detection and correction are developing, to our knowledge, there is no system-wide approach that: (1) detects an SEU as a system error that has not been handled locally and (2) decides how it can be dealt with in the system. In this way higher level system problems, such as data integrity, can be avoided.

1.4.1 Local Avoidance, Detection and Recovery Mechanisms

In this subsection we give a brief overview of "local" gate-level error detection and reaction mechanisms. By local we mean specific to a particular section of a logic circuit (such as for instance a pipeline stage). We list gate hardening, which is the only really LSEU specific mechanism we know of. Note that gate hardening is an avoidance mechanism, and not an error detection mechanism.

- *Cyclic Redundancy Check* (*CRC*): A cyclic redundancy check (CRC) is the result of a type of calculation made upon data, such as network traffic or computer files, in order to detect errors in transmission, operation or duplication. CRCs are calculated before and after transmission, operation or duplication, and compared to confirm that they are the same. The most widely used CRC calculations are constructed in ways such that anticipated types of errors, e.g., due to noise in transmission channels, are almost always detected. CRCs cannot, however, be safely relied upon to verify data integrity, i.e., that no changes whatsoever have occurred, since through intentional modification it is possible to cause changes that will not be detected through the use of a CRC; cryptographic hash functions can be used to verify data integrity.

- *Error Correcting Codes* (*ECC*): In information theory and coding, an error-correcting code or ECC is a code in which each data signal conforms to specific rules of construction so that departures from this construction in the received signal can generally be automatically detected and corrected. It is used in computer data storage, for example in dynamic RAM, and in data transmission. Examples include Hamming code, Reed-Solomon code, Reed-Muller code, Binary Golay code, and others. The simplest error correcting codes can correct single-bit errors (single error correction or SEC) and detect double-bit errors (double error detection or DED). Other codes can detect or correct multi-bit errors. Shannon's theorem is an important theory in error correction which describes the maximum attainable efficiency of an error-correcting scheme versus the levels of noise interference expected.

    1. If the number of errors is less than or equal to the maximum correctable threshold of the code, all errors will be corrected.
    2. Error-correcting codes require more signal elements than are necessary to convey the basic information.
    3. The two main classes of error-correcting codes are block codes and convolutional codes.

Note that in logic circuitry, the coded part of an input is passed through a check symbol generator to output the set of acceptable output codewords. This set is then compared to the coded output of the logic circuit. This type of error detection is called *concurrent error detection*, and is illustrated in Figure 1.2. [DT99, MLSS92, ZSM99] give examples of such self-checking circuits.

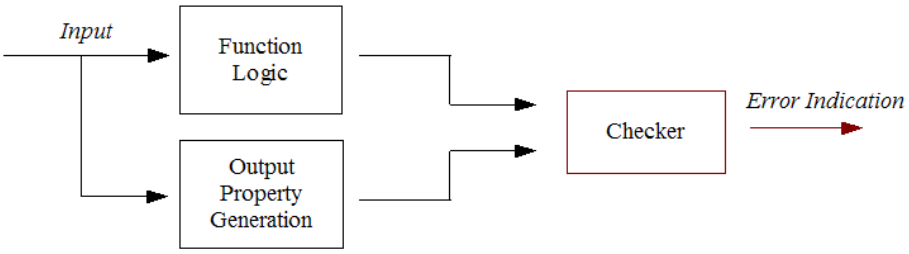


Figure 1.2: Concurrent Error Detection

- *Gate Hardening*: In order to avoid LSEU, or at least to reduce LSEU rates, one can use for instance larger gates, which are inherently less sensitive to particle strikes.

- *Parity*: In this usage, the number of '1' bits in a binary value is counted. Parity is even if there are an even number of '1' bits, and odd otherwise. Examples: the parity of the value 10111101 is even (there are 6 '1' bits); the parity of the value 01110011 is odd (there are 5 '1' bits). Parity is sometimes used for error checking due to the fact that it may be calculated easily. There are several types of parity: none, marking, even, and odd. 'None' means there is no parity calculated and a zero-bit is usually inserted (that is, the bit is present but unused or ignored). 'Marking' means that the parity bit is always a '1'. 'Even' and 'odd' parity insert '1' or '0' parity bits so that the total number of '1' is even or odd, including the parity bit. The parity bit is 'stripped off' before the data is used, thus a seven-bit character (or data value) requires eight bits to transmit or store - the seven data bits and the parity bit.

- *Redundancy*: Redundancy schemes detect errors by duplicating operations and comparing results. The duplication can be either done in space (i.e having two times the same circuit) or in time (i.e. repeating the same operation twice). If the same operation is realized three times, errors can be corrected by majority voting. By majority voting

we mean that the most frequent result is chosen (e.g. two times 1, and one time $0 \rightarrow 1$).

1.4.2 Component Specific Approaches

Component specific approaches have their scope limited by the boundaries of a component. We will pay special attention to this type of mechanisms in this thesis, in particular to Razor [EKD+03] and Diva [Aus99]. The reason for that attention is that the aforementioned techniques can be easily modeled and compared to our framework, though MoHiDoC is a system wide approach. We also believe that MoHiDoC offers a convenient way to combine miscellaneous approaches and to extend existing mechanisms. Note that those approaches were not developed for SEU, but for other types of errors. However, interesting concepts come out of them which are applicable to SEU and have inspired us. Hereunder we give a brief overview of Razor and Diva. In Chapter 4, we will make a more in-depth analysis and comparison of those techniques.

- *Diva* [Aus99]: The authors introduce dynamic verification, a novel microarchitectural technique that can significantly reduce the burden of correctness in microprocessor designs. The approach works by augmenting the commit phase of the processor pipeline with a functional checker unit. The functional checker verifies the correctness of the core processors computation, only permitting correct results to commit. Overall design cost can be dramatically reduced because designers need only verify the correctness of the checker unit. They further detail the DIVA checker architecture, a design optimized for simplicity and low cost. Using detailed timing simulation, they show that even resource-frugal DIVA checkers have little impact on core processor performance. To make the case for reduced verification costs, they argue that the DIVA checker should lend itself to functional and electrical verification better than a complex core processor. Finally, future applications that leverage dynamic verification to increase processor performance and availability are suggested.

- *Razor* [EKD+03]: Razor has been developed to detect and recover from timing (delay path) errors in a processor pipeline. The key idea of Razor is to tune the supply voltage by monitoring the error rate during circuit operation, thereby eliminating the need for voltage margins and exploiting the data dependence of circuit delay. A Razor flip-flop is introduced that double- samples pipeline stage values, once with a fast clock and again with a time-borrowing delayed clock. A metastability-tolerant comparator then validates latch values sampled with the fast clock. In the event of a timing error, a modified pipeline misspeculation recovery mechanism restores correct program state.

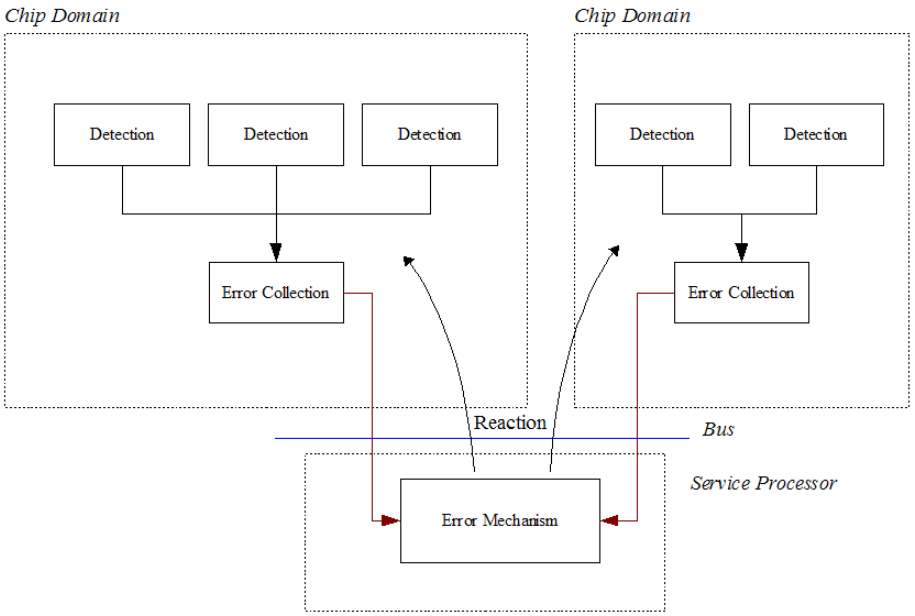


Figure 1.3: A schematic view of state of the art system wide error handling

1.4.3 System-Wide Approaches

Currently, system-wide approaches are designed for on-chip error detection and error collection and off-chip error analysis and recovery. This type of techniques are mainly implemented for errors in memory. On-chip detection mechanisms do not interact locally. When an error is detected, an error indication is passed as an output to a higher layer control unit such as a service processor. Figure 1.3 illustrates this concept. This out-off-chip higher layer control unit (e.g., Service Processor), will analyze the problem and when the error was not locally corrected and reaction is possible initiate the appropriate action. Unfortunately, since a signal has to be sent out of chip, such handling is very slow and cannot handle LSEU fast enough in most cases.

1.4.4 Limitations

The methods presented above have certain limitations. First, they are unfortunately often expensive to implement. This is especially true if they are implemented in every component. Second, if implemented as a local check or recovery mechanism, they lack an understanding of system-wide procedures and of the global system states. A direct consequence is that important

issues such as error propagation cannot be treated efficiently. On the other hand, high level check mechanisms lack granularity. We mean by this that a very simple error which could be corrected easily with a local error correction mechanism, could, in this case imply a chip reboot. Obviously, it would not be an adequate solution, since it would be time consuming and maybe generate some data loss or corruption.

In the next chapter, we will present the MoHiDoC framework. Our intent is that MoHiDoC should integrate miscellaneous mechanisms, offer a standardized communication structure between them, define the error handling process while reducing the costs.

Chapter 2

The MoHiDoC Framework

The state of the art techniques presented in the previous chapter were mainly designed for local error handling. Furthermore, existing system-wide approaches tend to privilege on chip error detection and collection, and off chip analysis and reaction. In the near future, those mechanisms will not be able to cope with the increasing LSEU rates, at a reasonable cost. Consequently, we propose a framework enabling integration and interfacing of distributed error detection mechanisms in order to create a self-healing system-on-chip, offering enhanced analysis and reaction capabilities. The framework is designed to be flexible, modular and hierarchical.

2.1 Concept

The framework presented herein is intended to make a chip aware of its state and capable of autonomously reacting in case an LSEU should occur. We would like to offer different options to a chip designer. The latter should have the possibility to choose a very simple and cheap implementation of the framework, in which for instance only a few critical logic parts would be checked for errors. In this case, all errors could be reported to a higher level which in turn would reboot the chip or purge the affected component(s). On the other hand, a designer willing to invest more in complexity could choose an implementation, in which at a fine granularity, many logic blocks would be checked for errors. Further, each of these blocks could locally make a diagnosis and determine whether a local reaction is possible or a higher on-chip layer should be notified. Higher layers, in turn, would have enhanced analysis and reaction capabilities. This kind of implementation of course considerably increases the self-diagnostic ability of a chip and makes reaction much more surgical. The chip would become *self-healing*.

The framework relies on a hierarchical structure. The lowest layers are incorporated in the chip, whereas higher layers could be part of a centralized component specialized in chip level error analysis and decision as well as

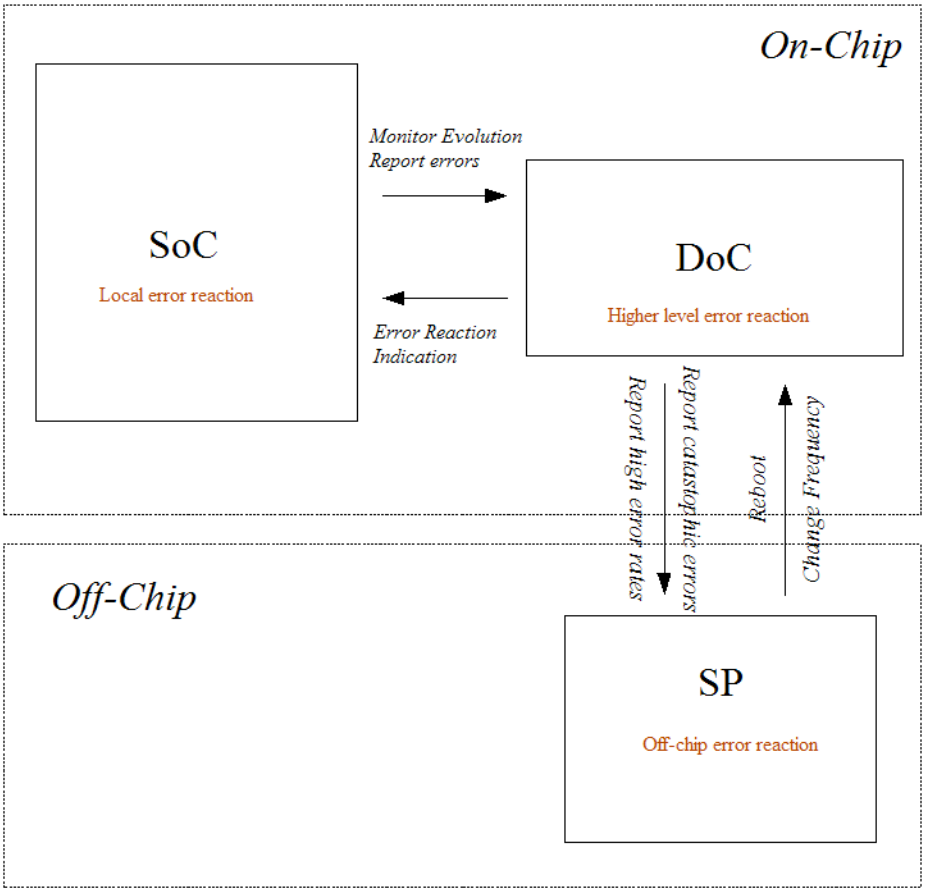


Figure 2.1: High Level View

correction. We will call this component "Diagnosis on Chip component" (DoC component).

Moreover, off-chip reaction should not be omitted (e.g. service processor SP). In case on-chip reactions is not sufficient or extreme measures are to be taken (reboot), MoHiDoC should privilege this latter option. In this thesis, we will concentrate on a three layer architecture (chip, DoC and SP). However, the model could easily be extended to n layers (n_1 on-chip and n_2 off-chip).

As Figure 2.1 shows, local reaction operations can be requested locally whereas more complex operations, often involving a larger number of components and more complex operations, can be performed on the DoC component. Finally, off-chip operations (reboot, alter frequency), can also be requested. Off-chip reaction is typically performed on a service processor (SP).

The MoHiDoC framework is first a novel way to model systems, which combines Petri Nets and logic building blocks (BB). This new model offers modularity and flexibility and allows the integration of miscellaneous mechanisms. It also allows the addition of different layers of analysis and control. Second, our framework handles errors in an original way by prioritizing local reaction when possible and allowing errors to propagate when detected to reduce costs.

The MoHiDoC Framework relies on the following novel ideas:

1. A new building block BB model.
2. The mapping of those BB to the transitions of a hierarchical Petri Net.
3. The structuring of the tokens of a PN into data and control, where control can be either detection or reaction specific.

Note that though we tried to be as specific as possible regarding the design and the implementation of the framework, some aspects are highly system dependent and up to the chip designer.

In the sequel, we will present those three aspects separately.

2.2 Modeling the Logic: Building Blocks

This section explains the first concept our MoHiDoC framework relies on, namely Building Blocks (BB). Building Blocks are a way to model and divide a system into different entities. This process allows us to work at a finer granularity, on BB with similar structures. This structure in turn makes it possible to define standard functionalities and interfaces. Later, we will also use this division into BBs to model the behavior of a component using PN, in a novel way presented in Section 2.3. This approach further offers modularity and flexibility, as a chip designer can decide on how coarse he wants his modeling to be. Below we will distinguish between so-called "Basic" and "Enhanced" BBs, depending on the functionalities of a BB. Later, we will explain how BB interact and communicate. Note that the granularity at which we define the BBs hereunder is variable and dependent on the design constraints. It could potentially go from gate level to a whole chip. However, in this thesis we consider the boundaries of chip's components.

2.2.1 Basic Building Block

The basic building block model is known in the literature [DT99], [ZSM99]. It is a "Mealy" state machine (a finite state machine whose output is a function of state transition, i.e., a function of the machine's current state and current input). Note that every logic circuit can be modeled as a Mealy State Machine. The Mealy State Machine is accompanied with on-line error detection based on error identification bits generated by the logic.

An on-line error detection mechanism is, in our case simply characterized by the number of erroneous bits it can detect. Let us call this number nb. Error identification bits are bits generated by a check symbol generator based on the input to the BB. They define a set of allowed codewords in the output of the logic under consideration. Typically, a certain predefined number of erroneous bits can be detected. A trivial example of such a scheme is parity. Based on the parity of the input, the check symbol generator defines the allowed parity of the output and detects an error if the output parity does not match the allowed set. Examples of such schemes are given in [DT99], [MLSS92], and [ZSM99], and the local error mechanisms defined in Chapter 1.

Figure 2.2 shows a basic BB. The output logic shown is the functional part of the BB and outputs results. The next state logic indicates the next state of the functional logic. The output of the output logic is passed as an output of the BB, whereas the output of the next state logic is stored in local bistable elements (latches), whose content defines the state of the building block.

We characterize BBs with a BB identification BB_id. The necessity for such an identification will become apparent in Chapter 2.4. We define the

set of all BBs as $BB = \{BB\_1, BB\_2, ..., BB\_n\}$ When an error is detected,

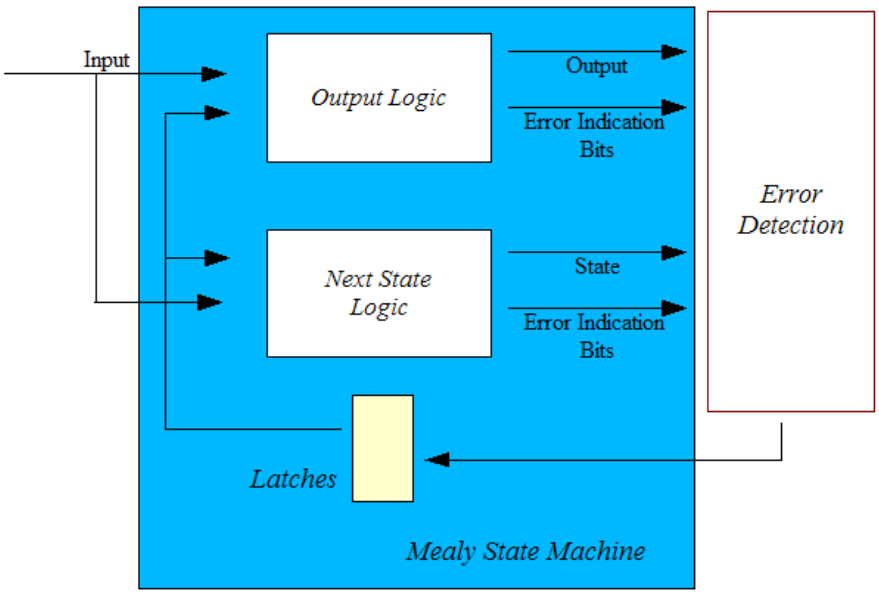


Figure 2.2: A basic building block (BB)

a description of the error, or error indication (EI) should be generated. A simple approach is to describe an error as:

$$EI = \{Type, Bits\} \tag{2.1}$$

where:

- Type: Output or State
- Bits: Number of erroneous bits

We separate error detection on next state and output logic because they might require different reactions. An error in the output logic will affect other BB but not the local state, whereas an error in the next state logic will in the cycle following it affect the local state and then, if no action is taken, affect subsequent BBs.

To handle errors, we will add analysis and reaction capabilities to BB, as explained below.

2.2.2 Enhanced Building Blocks

We enhance this building block model by adding analysis capabilities and reaction capabilities as shown in Figure 2.3

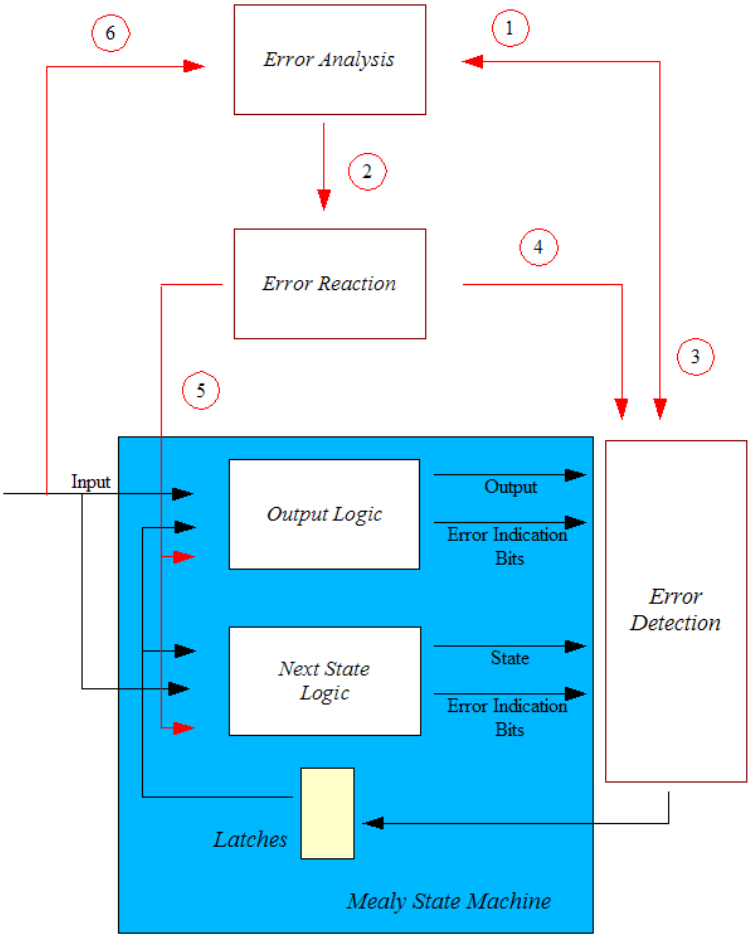


Figure 2.3: An enhanced building block (BB)

The functionality added is described hereunder (the numbering corresponds to the numbering in Figure 2.3):

1. When an error is detected using an error detection mechanism as described above, an error indication EI, as in Equation 2.1 is passed from the error detection to the error analysis. The analysis mechanism, described in the sequel (Section 2.4), can determine whether the error can be taken care of locally or not.

2. In the first case, the analysis determines the appropriate action to be taken and passes the error indication to the reaction mechanism.

3. In the second case, the error indication is passed as an output of the BB to subsequent BBs on the path, which in turn will analyze the error.

4. The reaction can be a direct recovery, in which case the corrected output is forwarded to the original destination as if nothing had happened. Such a mechanism could for instance be an error correcting code (ECC).

5. Or the initiation of some modifications in the output and next state logic to alleviate the error. For instance the operation could be repeated, under the condition that inputs were stored for repetition.

6. Inputs to the analysis can also be used to analyze errors which occurred in previous BB, since not all BB have to be enhanced.

The form in which errors are described and passed from building block to building block will be explained later. It is important however to note at this point that one can combine basic BBs and enhanced BBs. Hence, we introduce already at this point a certain flexibility in the design and costs.

Below we will give an extremely simple example of how the BB model could be applied to a logic circuit.

2.2.3 Example

Let us consider the modulo 3 counter with the truth table shown in Table 2.1. When enhanced with parity bits (on-line error identification bits), the circuit is represented in Figure 2.4. If we wanted to apply detection to this circuit, we would simply compare the parity bits (calculated by an output property generator) to the parity of the actual output. Thus, if an error had occurred in the logic (Output logic or Next State Logic), the two parities would be different and an error indication would be generated. Subsequently, if we added analysis and reaction, we could decide to repeat operations which went wrong (if we store the inputs). Figure 2.4 with detection capabilities is a basic BB, and Figure 2.4 with detection, analysis and reaction is an enhanced BB.

Input	*Present state*	*Next State* Y_1Y_2	*Output* Z_1Z2
-	S_0 (00)	S_1	00
-	S_1 (01)	S_2	01
-	S_2 (11)	S_0	10

Table 2.1: Truth Table of a modulo-3 counter

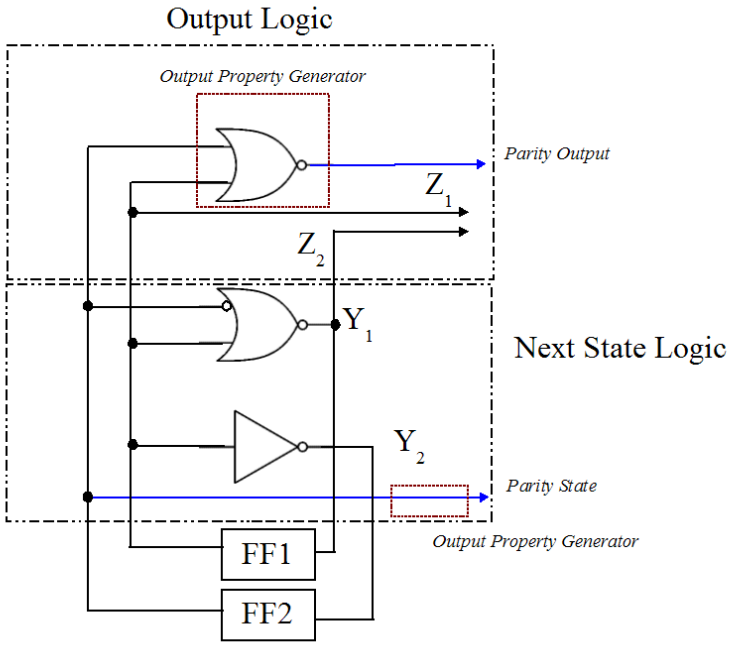


Figure 2.4: A modulo-3 counter with parity bits

2.3 Communication Structure: Petri Nets

For communications between Building Blocks, we rely on a Petri Net (PN) model. Petri Nets are a well-known formal, graphical, executable technique for the specification and analysis of concurrent, discrete-event dynamic systems. The technique is undergoing standardization. We chose PN because they offer a convenient way to describe the behavior of systems, in our case the behaviors of logical entities (e.g. components) on systems-on-chip. The use of a well known modeling technique further offers the advantage that it is easy to find related documentation and that the soundness of the approach is proved. We will start this chapter by formally defining Petri Nets, and then we will explain how we integrate them into our design.

2.3.1 Petri Nets

Formally, a PN can be described as a 4-tuple [ZB96]:

$$PN = \{P, T, A, m_0\} \tag{2.2}$$

where

- $P = \{p_1, p_2, ..., p_n\}$: a finite, nonempty set of token places (places)
- $T = \{t_1, t_2, ..., t_n\}$: a finite, nonempty set of transitions
- $A_i =\subset (p \times T)$: a set of incoming directed arcs, connecting places to transitions
- $A_o =\subset (P \times T)$: a set of outgoing directed arcs, connecting transition with places
- $A = (A_i \cup A_o)$: a set of directed arcs
- $m_0 : \{m_{p_1}, m_{p_2}, ..., m_{p_3}\}$: an initial marking function which assigns a non-negative number of *tokens* to each places of the net, $M_0 : P \rightarrow \{0, 1, ...\}$

For each place $p \in P$, its input set $Inp(p)$ contains all transitions connected to p by directed arcs, $Inp(p) = \{t \in T | (t, p) \in A\}$, while its output set is $Out(p) = \{t \in T | (p, t) \in A\}$. Input and ouput sets of transitions are defined similarly.

Let any function $m : P \rightarrow \{0, 1, ...\}$ be called a marking in a net $PN = (P, T, A, m_0)$.

A transition t is enabled by a marking m iff every input place of t is assigned at least one token by m. Every transition enabled by a marking m can fire. When a transition fires, a single token is removed (simultaneously) from each of its input places and a token is added to each of its output places.

This determines a new marking in a net, a new set of enabled transitions and so on. The set of all markings that can be derived from the initial marking m_0 is called the *set of reachable markings* of a net.

Further, we call a Petri Net with $|Inp(t)| = |Out(t)| = 1$ for all transitions $t \in T$ a *State Machine* [SL98]. We also characterize a PN as *K-Bound*, when at any time the marking at every place is smaller or equal to K, i.e. $\forall m, m_{p_i} \leq K$.

In the sequel, we will focus on $1 - Bound\ State\ Machines$ and refer to them as Petri Nets "PN". We limit ourselves to this type of Petri Nets for the sake of simplicity and also because it is easier to fit them to the building block presented in Section 2.2. Figure 2.5 shows such a PN.

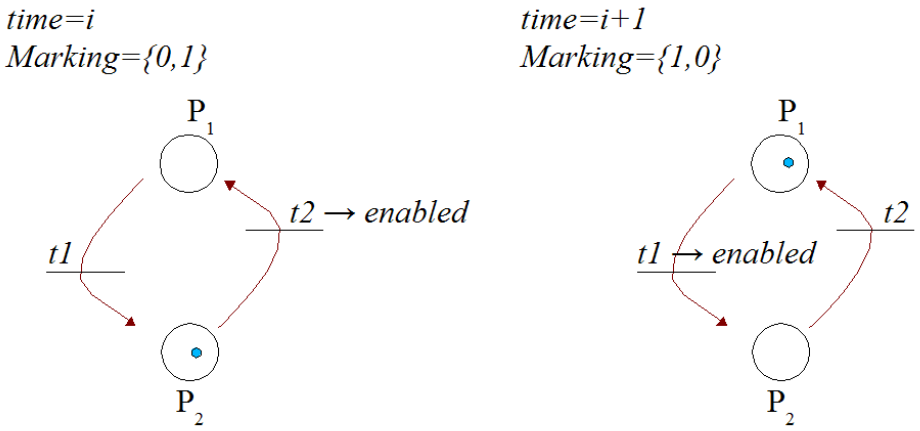


Figure 2.5: A 1-Bound State Machine

Note in addition that when used to describe a system, transitions usually model actions, and places states. Consequently, one can imagine that in Figure 2.5, at time i, the system is in state p_2, and that an action t_2 is performed resulting in a state p_1 at time $i + 1$. Actions (transitions) are considered to be atomic, i.e. they are either completed or not started.

2.3.2 Building Block Mapping

Petri Nets can model the behavior of systems, and the Building Block model is intended to divide the logic into different entities, with standardized structures but particular functionalities. We combined the two to have an architectural, modular and behavioral description of a system-on-chip. Transitions in PNs, as explained above, correspond to actions bringing a system from one state to another. We simply map BBs to the logical functional elements of the system, and then in turn map transitions of PNs to the BBs. In Figure 2.6, we show the mapping of a BB as defined in Section 2.2.

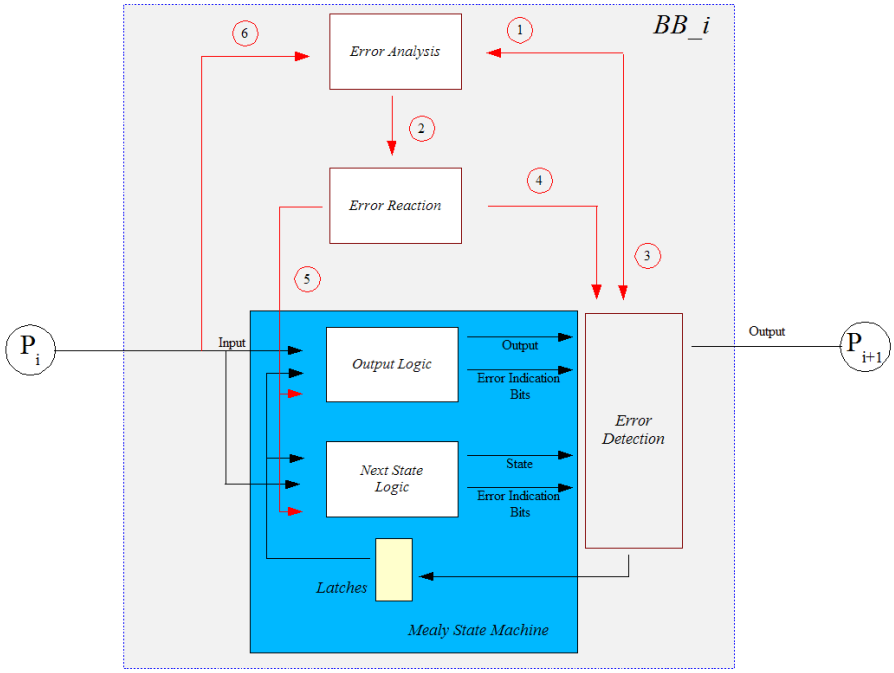


Figure 2.6: Mapping of a BB into a PN transition

Combining several BBs, we obtain a PN, corresponding to the functionality of the component or system under consideration.

Thus, we can define a function *map* linking transitions to BBs:

$$map : T \rightarrow BB \tag{2.3}$$

$$\{t_1, t_2, ..., t_n\} \rightarrow \{BB\_1, BB\_2, ..., BB\_n\} \tag{2.4}$$

Figure 2.7 illustrates a Petri Nets with incorporated BB transitions.

At this point it is important to note that there is not a unique description of a system. The granularity of the modeling and the functionalities described, both for the PN and the BB are up to the designer. This, of course increases the flexibility of the model in terms of cost and modularity.

As shown in Section 2.2, a BB is initially a "Mealy State Machine", with one input and one output (be it multidimensional). Thus, we can clearly limit ourselves to $|Inp(t)| = |Out(t)| = 1$ ($State\ Machine$). We will merge or split signals in token places in order to always fulfill this requirement.

The marking of token places, and hence the enabling of transitions, is explained in Section 2.4. However, let us say at this point that a transition is "being fired" when data is treated in the corresponding BB.

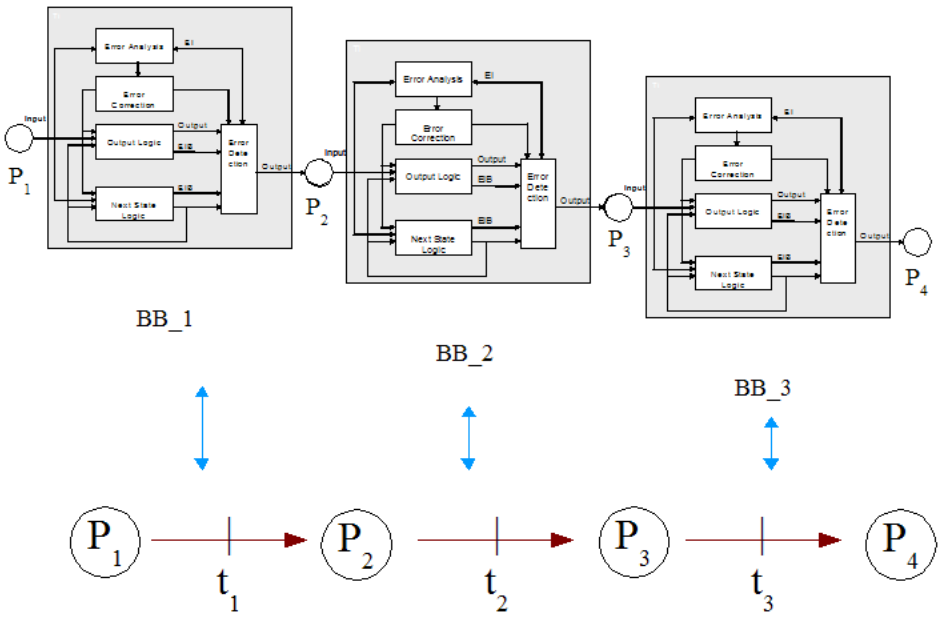


Figure 2.7: Petri net with incorporated BBs

2.3.3 Hierarchy

To add different layers of information, analysis and control, we add different layers of Petri Nets, thus forming a hierarchy of PN. Figure 2.8 illustrates the idea of Petri Net Hierarchy (PNH).

Lower level Petri Nets are refinements of higher layer PN. More precisely, this means that higher layer Petri Nets model the chip in a more abstract way. Higher layers' building blocks, as we will see in Section 2.4, are intended to instantiate higher levels of error analysis and reaction mechanisms. They can be implemented on a specialized component on-chip and/or off-chip, for instance on a service processor. Consequently, only the lowest layers of PN corresponds to the initial physical implementation. The layers above correspond to error handling mechanisms added on-top of it.

On higher layers (i.e. not the lowest physical layers), the BB model and mapping remains valid. However, as inferred above, higher layers only have analysis and reaction capabilities, since errors in the actual original implementation are detected on the chip level. Nevertheless, higher layers can, as we will see in Section 2.4, implement some statistics collection mechanisms (e.g. calculate error rate) and take appropriate actions (e.g. ask top off-chip layer to reduce the chip frequency).

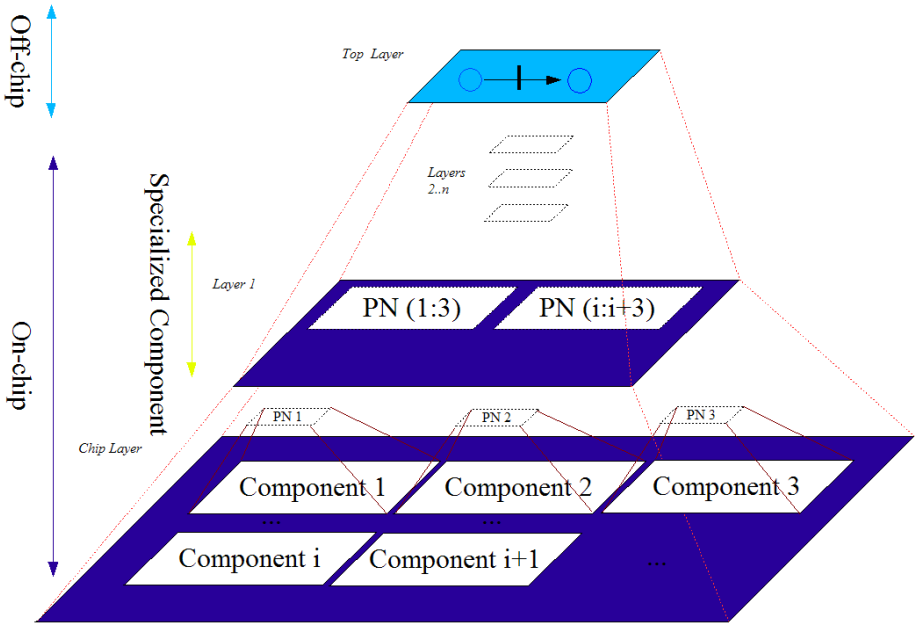


Figure 2.8: A hierarchy of Petri Nets

Mathematically, we define a refinement scheme ζ as follows [ZB96]:

$$\zeta = \{PN_0, \eta, \rho, \phi, \psi\} \tag{2.5}$$

where

- $PN_0 = \{P_0, T_0, A_0, m_0\}$: is a place/transition (initial) petri net
- $\eta = PN_1, PN_2, ..., PN_k$: is a family of place/transition (refinement) nets
- ρ: is a partial refinement function which associates elements of P_0 (place refinement) and T_0 (transition refinement) with nets from η, $\rho : P_0 \cup T_0 \rightarrow 1, ..., k$, so each place $p \in P_0$ is refined by the net $\eta_{\rho(p)}$ (if $p \in Dom(\rho)$ and each transition $t \in T_0$ is refined by $\eta_{\rho(t)}$ (if $t \in Dom(\rho)$)
- ϕ and ψ are input and output interface functions which define the interconnections between the input and output sets of a place (or transition) and its refinement determined by ρ; for each $p \in P_0$, if $p \in Dom(p)$, then $\phi(p) : T_0 \rightarrow 2^{P_{\rho(p)}}$ and $\psi(p) : T_0 \rightarrow 2^{P_{\rho(p)}}$; similarly, for each $t \in T_0$, if $t \in Dom(\rho)$, then $\phi(t) : P_0 \rightarrow 2^{T_{\rho(t)}}$ and $\psi(t) : P_0 \rightarrow 2^{T_{\rho(t)}}$

A refinement system ζ defines a Petri Net $PN = \{P, T, A, m_0\}$ such that:

- $P = \{p_i \in P_0 | p_i \notin Dom(\rho)\} \cup \left\{p_{i,j} | p_i \in Dom(\rho) \wedge p_j \in P_{\rho(p_i)}\right\}$,
- $T = \{t_i \in T_0 | t_i \notin Dom(\rho)\} \cup \left\{t_{i,j} | t_i \in Dom(\rho) \wedge t_j \in T_{\rho(t_i)}\right\}$,
- $A = \bigcup_{x_l \in P_0 \cup T_0} \left\{a_{t,i} | a_i \in A_{\rho(l)}\right\} \cup$
  $\{(t_l, p_{i,j}) | p_i \in Dom(\rho) \wedge t_l \in Inp(p_i) \wedge p_j \in \phi(p_i, t_l)\} \cup$
  $\{(p_{i,j}, t_l) | p_i \in Dom(\rho) \wedge t_l \in Out(p_i) \wedge p_j \in \psi(p_i, t_l)\} \cup$
  $\{(p_l, t_{i,j}) | t_i \in Dom(\rho) \wedge p_l \in Inp(t_i) \wedge t_j \in \phi(t_i, p_l)\} \cup$
  $\{(t_{i,j}, p_l) | t_i \in Dom(\rho) \wedge p_l \in Out(t_i) \wedge t_j \in \psi(t_i, p_l)\}$,
- $\forall (p \in P)\ m_0(p) = \begin{cases} m_{0,0}(p), & if\ p \notin Dom(\rho) \\ m_{\rho(p_i),0}(p_j), & if\ p = p_{i,j} \end{cases}$

Let us consider the example shown in Figure 2.9. It is a an example of a place refinement, where the PN shown as (1) is the initial higher level PN and the PN in (2) the refinement of place $P1$. (3) shows the complete lower layer PN. For this example, *rho* is given in Table 2.2 and ϕ and ψ in Table 2.3.

$P_0 \cup T_0$	ρ
$P1$	1
$P2$	undefined
$T1$	undefined
$T2$	undefined

Table 2.2: Refinement Function

ϕ	$T1$	$T2$	$P1$	$P2$
$P1$	$\{P1.3\}$	-	-	-

ψ	$T1$	$T2$	$P1$	$P2$
$P1$	-	$\{P1.4\}$	-	-

Table 2.3: Interface Functions

Consequently, in our model, each layer can be considered as a refinement of a higher layer. Note that a refinement is not unique, i.e different combinations of place and transition refinements could lead to the same result. We believe that it is easier for the understanding of the model one develops to only apply one type of refinement (i.e. place or transition).

In this thesis we will, without loss of generality, restrain ourselves to three layers of control (layer of PN). Table 2.4 lists those layers. However, the number of layers could be larger in other designs.

Name	*Granularity*	*type*	*Location*
Component	Local	ED/ER	On-chip[1]
DoC	Global	PD/ER	On-chip
SP	Global	ER	Off-chip

Table 2.4: PN layers

2.3.4 Token Places Interfaces

Token places are used to model interfaces both between different BBs and between different layers. Between BBs, token places can either be transparent, i.e. they have no functionality at all, or be very simple routers, i.e. they forward tokens to the correct BB, in case their are several options.

When token places are interfaces to higher layers, they could be registers, or a dedicated bus. In the first case, the register could be read regularly by an external service processor. In the second case, the dedicated bus could be used to convey information to the specialized component.

[1]ED: Error Detection, ER: Error Reaction, DoC: Diagnosis on Chip (Specialized component), SP: Service Processor, PD: Problem Detection

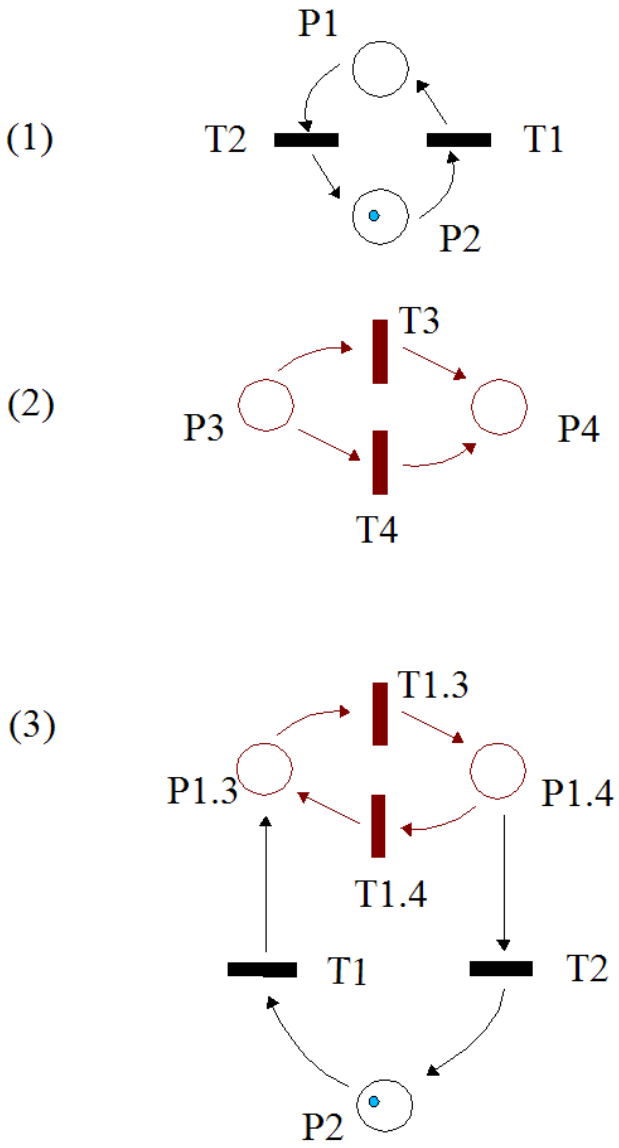


Figure 2.9: PN refinements

2.4 Conveying Information: Tokens

In addition to modeling the logic with building blocks and the communication infrastructure with Petri Nets (PN), we need means to model the data flowing through a chip. Moreover, we also need to define how building blocks on a particular path or on different layers interact and coordinate their "actions". To do so, Petri Nets actually already offer a convenient solution: tokens. In classical Petri Nets, tokens are simply black coins placed in token places to indicate in what state the system currently is. In more elaborate PN, such as fuzzy [HL95, CS92, CS91] PN and colored PN [Jen97, Jen98, KCJ98], tokens themselves carry some information, such as for instance the degree of completion of a particular task or the data type. In the MoHiDoC framework, we rely on this idea of "enhanced" tokens. Tokens are even one of the cornerstones of the approach. As inferred, they allow to model data and control information. By data, we mean regular data and by control information, information passed from building block to building block or layer to layer to coordinate error analysis and reaction mechanisms. In section 2.4.1 below, we explain in more detail how tokens are structured. Then in Subsection 2.4.3, we show how the tokens are used for analysis and reaction

2.4.1 Token Structure

Tokens carry two types of data: regular data and control data. Control data can be either analysis data or reaction data. In our model, the content, and hence the state of a token, is modified in building blocks as shown in Figure 2.10. We first define a token x, just after exiting BB i, as:

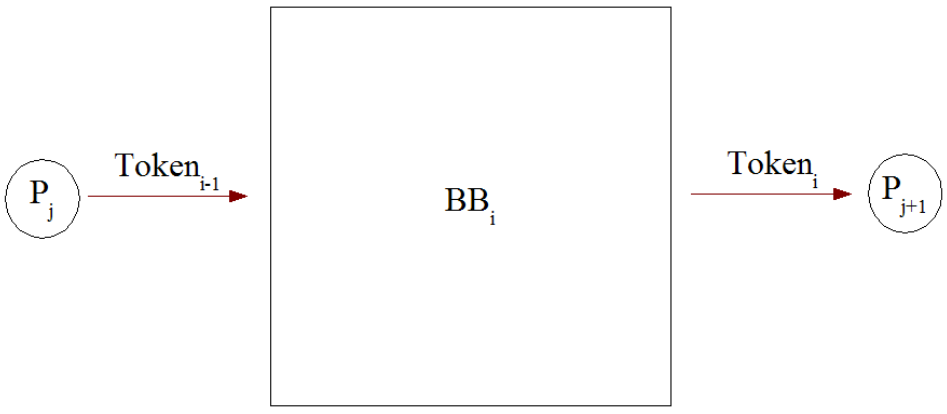


Figure 2.10: State evolution of tokens in BBs

$$x^i = \left[x^i_{data}, x^i_{control}\right] \tag{2.6}$$

Now, let us focus on the control part. The control part, as mentioned above, can be either analysis data or reaction data. The analysis data is used to describe an error, and the reaction data is used to force a BB to take a particular action to react to an error, even though the error might have occurred in another BB. To save bandwidth, they both use the same location in a token, and a phase bit indicates which type of control data is conveyed. So the control part is:

$$x^i_{control} = \{b_{phase}, x_{analysis}\ or\ x_{reaction}\} \tag{2.7}$$

We then define the structure of the analysis and reaction data as folows:

$$x^i_{analysis} = \{\tau, \beta_{correct}, nb, e_{prop}, \gamma, issuer\} \tag{2.8}$$

where

- τ: data type (I/O or State)
- $\beta_{correct}$: correct BB(s) on the path of the token
- nb: number of erroneous bits in detected error
- e_{prop}: BB(s) "contaminated by the error"
- γ: criticality of the data when the error occured
- $issuer$: BB and component where the error was detected

Furthermore,

$$x^i_{reaction} = \{\mu, data_{reaction}, executors, \pi)\} \tag{2.9}$$

where

- μ: reaction mechanism to be enabled
- $data_{reaction}$: specific data for reaction
- $executors$: BB(s) and component where the reaction data is to be read
- π: path along which the token should propagate

2.4.2 Token Creation and Propagation

Creation

Tokens are in the first place associated with a piece of information, which we will call in this section 'a packet'. This denomination is not completely random since in reality a packet could be, for instance a network packet. A packet will, in a component, be modified several times before being passed to another component or out of the chip. Each of those modifications will be done by a specific chunk of logic, which corresponds to a particular BB, as defined in Section 2.2. The idea is to conceptually map those packets to tokens. More precisely, the data part of a token corresponds to a chunk of data belonging to the same logical entity. We illustrate in Figure 2.11

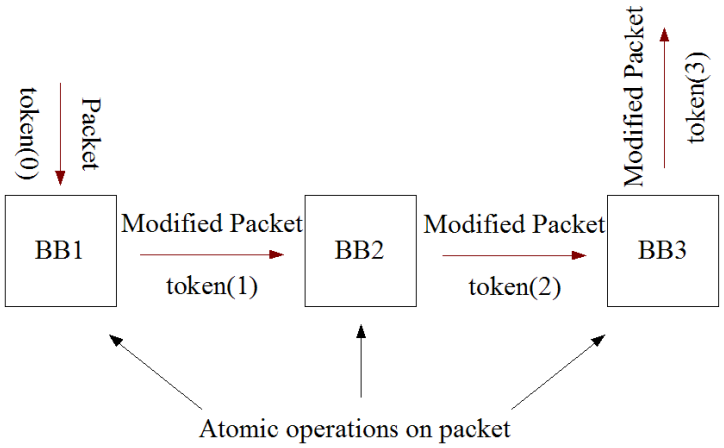


Figure 2.11: Evolution of data and token

a pipeline where a packet is modified several times and show how those packets are related to tokens. The operation within BBs could correspond to a finite state machine. Most importantly, the operations within a BB should be considered as an atomic operation bringing a packet from one state to another.

Consequently, the data part of a token is created as data enters the first BB on a particular path. The control data, on the other hand, is initiated when an error is detected while a packet is handled in a BB (during the atomic operation). If the error cannot be corrected locally and there is no local analysis, the control data describing the error is initialized to:

$$x^i_{control} = \{1, \tau, 0, nb, BB\_id, \gamma, issuer\} \tag{2.10}$$

Since the error detection mechanism will be able to determine by itself the data type, the number of erroneous bits, it puts its own id as the first block of the error propagation, to set the criticality of the data and its

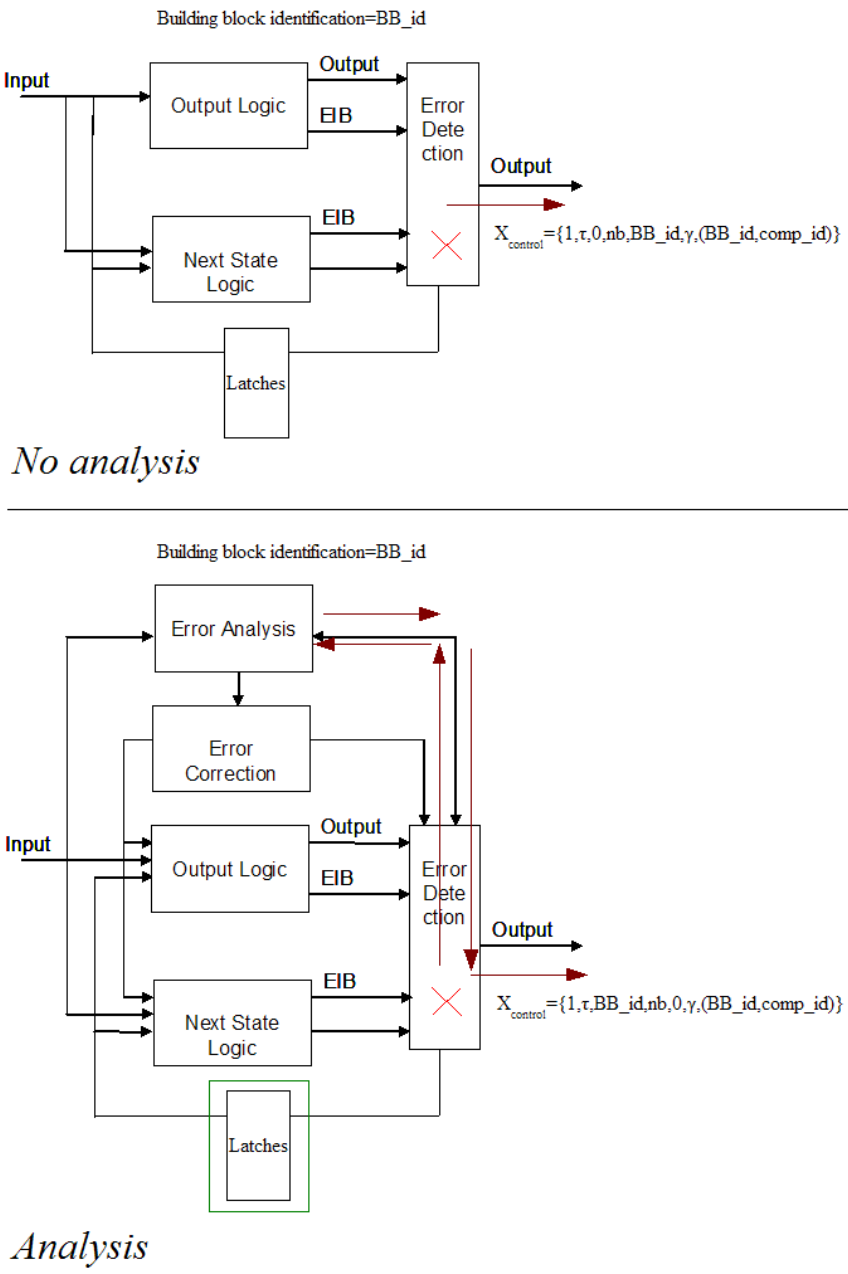


Figure 2.12: Two possible token control data initializations

own identity. If in addition to only having a detection mechanism, the BB under consideration also has analysis and recovery, the analysis will be able to determine, based on the error indication from the detection mechanism, that the error cannot be recovered locally and block the inputs of its state latches, consequently, $\beta_{correct}$ will be initialized to BB_id, and the error will not propagate, so e_{prop} is set to 0. Those modifications result in having:

$$x^i_{control} = \{1, \tau, BB\_id, nb, 0, \gamma, issuer\} \tag{2.11}$$

Figure 2.12 summarizes those two cases. Once an error was detected and no local reaction mechanism could properly handle it, a token with the control information defined above will be passed as an output of the BB. The part of this latter token can either be set to null, if a local analysis was done, or be the regular data, produced as an output of the building block in the case there was no local analysis. Note that in the second case, the data is *corrupted*, i.e though we know that an error occurred, we continued the calculations. The underlying idea is that subsequent blocks might be able to correct that error and consequently avoid loosing the data and gaining the time another reaction mechanism might have required. If the error is a "don't care" for the next BB, the error can be masked and the process of the next BB be performed as usual. This can be continued until either the error can be corrected (e.g. through assertion), or ignored (e.g. indeed in "don't care" data) or sent to a higher layer.

Propagation

After the creation of a token, this token propagates along the path required by the data part. In the case no error was detected, or an error was detected and corrected locally, no particular action is taken, and execution proceeds as if no framework were present. On the other hand, if an error was detected and could not be corrected locally, the analysis data of the control part is initialized and propagates along the same path as the data would. Every enhanced building block, i.e BB with analysis capabilities, along that path will determine whether it can correct the error, and in the positive case, do so. The first BB with analysis capabilities which is encountered by the analysis data (it can be the BB where the error occurred), blocks the input of its state latches and all subsequent BB on the block as well, thus preserving a correct state and making a potential recovery easier. Once the token reaches a token place interfacing a higher layer, if the error could not be corrected, the token is passed up one layer. In practice, this could mean that control information is written to a special register, which is read from a specialized component afterward, or passed through a dedicated bus to DoC. In Figure 2.13, we show a PN representation of a component. The transitions with a BB (gray square) are on the data path under consideration. An error is detected but not analyzed locally in the first BB. Consequently, the control

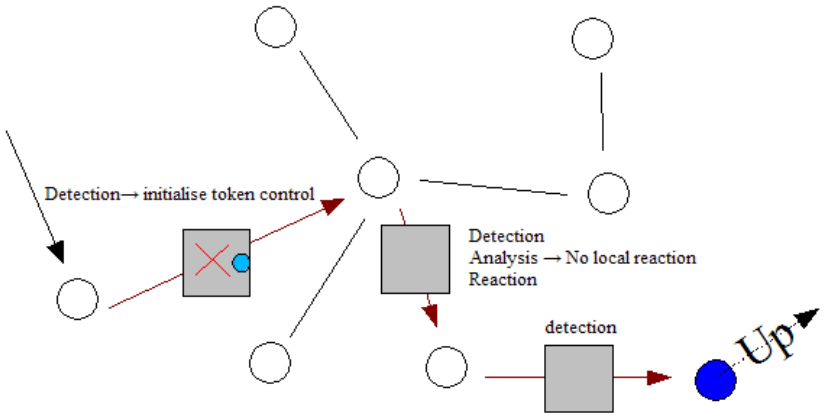


Figure 2.13: PN representation of token propagation

information is initialized. Then, the following BB analyzes the error and determines that no local reaction is possible. The state of this BB and the following ones is blocked and the data set to null. The last token place on the path is a interface to a higher layer. The token is passed up. Note that once an error was detected, the BB(s) preceding analysis add their id to the e_{prop} path parameter of the token, and those following the initial analysis to the $\beta_{correct}$ path parameter of the token in the control analysis data. This is done to facilitate higher layer analysis.

2.4.3 Analysis and Reaction

We define analysis as the process of evaluating an error indication to determine if an action can be taken locally, and if yes what it should be. Analysis can be more or less complex depending on the level of control. We will differentiate between simple on-chip building block analysis and more elaborate analysis, which could be done for instance on a specialized component on-chip or off-chip.

Simple Analysis

By simple analysis we mean analysis performed on-chip in a BB. This type of analysis applies to errors which occurred locally or in previous BBs on the same propagation path. In the latter case, the error is received as an input to the BB (as analysis data of a token). Then, the analysis should decide on two things:

1. Can the BB react to the error locally

- Yes: pass error information to Reaction

2. No: forward the token and preserve states of subsequent BBs, in order to preserve the current situation for a higher layer mechanism.

Complex Analysis

By complex analysis we mean more advanced analysis involving inputs from more components and more elaborate evaluations and decisions, resulting in more potential large scale (path specific, component wide, system wide) reactions. This type of analysis is intended to be implemented on a specialized component on-chip or on an off-chip higher level control unit such as a service processor.

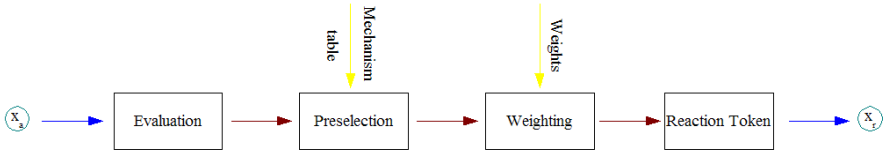


Figure 2.14: Control analysis process view

As shown in figure 2.14, the analysis process is divided into four phases:

1. The analysis data of the incoming token is evaluated in order to determine which lower layer BB are affected.

2. The applicable reaction mechanisms are preselected based on the output of the evaluation. The mechanisms are stored in a local table in a way defined below

3. The selected reaction mechanisms' costs are weighted to determine the most appropriate solution. Costs are related to the error rate and criticality of data, as explained below.

4. A token with reaction data corresponding to the selected mechanism is created and sent out to the appropriate BBs.

In this subsection we will discuss how we define analysis for an on-chip reaction on a specialized component. The description given also applies to higher layer off-chip reaction on a service processor, however, in the next subsection, we will point out some major differences.

Evaluation The evaluation is based on the $\tau, \beta_{correct}, nb, e_{prop}$ fields from the analysis data. Taking those values as an input, an error descriptor $error_{desciptor}$ is produced as an output. The purpose of that descriptor will be to allow a selection of applicable reaction mechanism. Formally, we can describe the evaluation process as:

$$evaluation : Eval(\tau, \beta_{correct}, nb, e_{prop}) \rightarrow error_{descriptor} \tag{2.12}$$

Preselection The description of available reaction mechanisms should be stored in a table for the analysis. The actual selection should be done before system bring-up. In fact, mechanisms can be configurable or adaptable during run-time (e.g. with Neural Nets). In this thesis however, we do not treat this issue. We propose a standardized syntax to define the functionalities of those mechanisms. In the preselection phase, we want to mark the mechanisms which can be applied to the error under consideration, based on what type of errors they can correct and where.

We define a reaction mechanism as a quadruplet:

$$M_i = \{BB\_list, num\_corrupted, nb, type, \lambda_{cost}\} \tag{2.13}$$

where

- BB_list: description of BB(s) for which this reaction mechanism applies
- $num\_corrupted$: number of corrupted BB(s) this mechanism can cope with
- nb: max number of corrupted bits this mechanism can cope with
- $type$: type of mechanism (recovery, correction, passup)
- λ_{cost}: Cost associated with this mechanism

The preselection module takes a list of mechanisms $L_M = M_1, M_2, ..., M_n$ and an error descriptor $error_{descriptor}$ as an input. It produces a list of applicable mechanisms L_{app} as an output. Formally, the functionality of the preselection can be written as:

$$preselection : Pre(L_M, error_{descriptor}) \rightarrow L_{app} \tag{2.14}$$

Weighting The idea underlying the weighting is that, in certain cases, the mechanism with the lowest λ_{cost} will not be the most appropriate one. For instance, if we have very critical data which should not be lost, it is definitely more appropriate to try and recover the data instead of simply skipping it and starting work on the next chunk of data, even though the second option would be less costly. Another example would be if the error

rate on a particular path becomes outrageously high. In that case it might be appropriate to send a token to a service processor which could then in turn reduce the chip frequency or reboot the chip in order to reduce the LSEU rate. Consequently, among the mechanisms in L_{app}, defined in Equation 2.14, we choose the one with the lowest weighted cost (weighting is done based on the *type*). So far, we have decided to associate three types to weights.

- $w_{correction}$: weight associated with correction mechanisms (e.g. repetition, deletion,...)
- $w_{recovery}$: weight associated with recovery mechanisms (e.g. error correcting codes,...)
- w_{passup}: weight associated with passing the token to a higher layer

We define $w_{correction}$ and $w_{recovery}$ as being proportional to the criticality of the data. If the data can be lost, and consequently appear as not being critical, $w_{recovery}$ should be high and $w_{correction}$ low. Conversely, if the data are rather critical, it should be attempted to recover the data if possible, thus $w_{recovery}$ should be low and $w_{correction}$ high. Thus, we set $w_{correction} = 1 - w_{recovery}$ and $w_{correction} = \gamma$ (criticality, as defined in Equation 2.10). Figure 2.15 shows those functions.

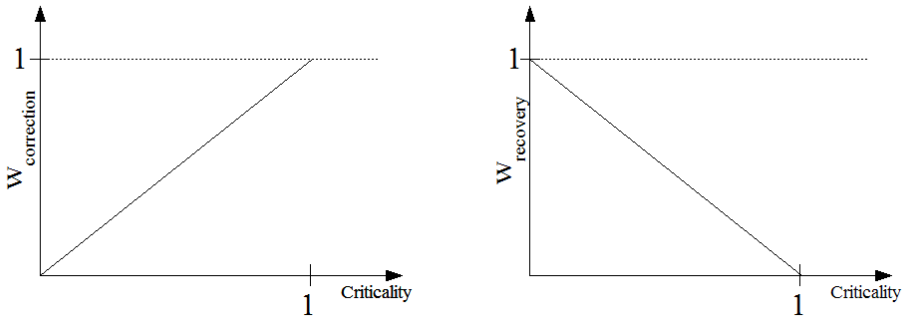


Figure 2.15: correction and recovery weights

The pass up weight w_{passup} is independent from the criticality. We base this weight on the error rate. We believe that if the error rate becomes too high, specific actions, such as reducing the frequency or rebooting should be taken. Consequently, we set $w_{passup} = e_{rate}$ ($e_{rate} = error\ rate$). Of course, this implies that some error rate calculations are done on the specialized component.

Finally, the decision about which mechanism is to be used is taken as follows:

$$M_{i,selected} = \min_i(\lambda^{corr} \times w_{correction}, \lambda^{rec} \times w_{recovery}, \lambda^{passup} \times w_{passup}) \quad (2.15)$$

where

- $\lambda^{corr} = \{\lambda_i \ with \ M_i \in L_{app} | type = correction\}$
- $\lambda^{recovery} = \{\lambda_i \ with \ M_i \in L_{app} | type = recovery\}$
- $\lambda^{passup} = \{\lambda_i \ with \ M_i \in L_{app} | type = passup\}$

Then, the selected mechanism has to be run. For that purpose, a token is created as explained below.

Token Creation The selected mechanism, as inferred earlier, could affect several BBs on-chip during the reaction process. It should be mentioned that the reaction can be directed to non-affected BBs as well, for instance when the SEU rate has increased (above a limit), DoC starts initial error detection, analysis and reaction mechanisms which were on standby for power saving reasons. Consequently, the reaction data, as defined in Equation 2.9, contains the identification of the different BBs involved in the reaction. Those BBs should be specified in the reaction mechanism's description or determined during reaction. A path signature π describes the path along which the token should propagate to reach all involved BBs.

2.4.4 Off-Chip Analysis

Off-Chip analysis is a particular type of complex analysis. The differences are the following:

1. Tokens cannot be passed to higher layers, since we are at the top level, consequently an action must be taken.

2. Since the analysis and reaction are done off-chip, actions can be taken to alter the chip's behavior, such as for instance reboot the chip or reduce the frequency.

Point (1) is straightforward, and the only difference with the complex analysis presented in subsubsection 2.4.3 is that the mechanism table cannot contain any entries of type *passup*.

Point (2) on the other hand is more interesting and is directly linked to the w_{passup} weight defined earlier. We have seen in the introduction that SEU rates are increasing at higher frequencies. Consequently, the top layer should have the possibility to momentarily reduce the chip frequency in order

to reduce the SEU rate. The structure of the complex analysis in itself is not modified, since we simply add other types of reaction mechanism.

What is most important in an SP is that it stores history log files of events happening in the chip, which can be used during maintenance/reconfiguration times.

In the next chapter, we will present a mathematical model we developed to estimate the costs and benefits of the MoHiDoC framework.

Chapter 3

Mathematical Analysis

In this chapter, our goal is to develop a mathematical model of costs vs. benefits. Our intent is to show that our approach, while offering flexibility and strong benefits, especially in terms of latency and reliability, comes at a reasonable price. Note that we talk about "system costs" and do not consider company economics (e.g. revenues and company costs).

3.1 Problem

Our objective is to maximize the probability that an error is taken care of locally. We call this probability $Pr(sol)$. The motivations to maximize this value are manifold:

- Local reaction reduces latency with respect to off-chip reaction or specialized component reaction
- Local reaction reduces the number of components involved (error containment), thus offering greater granularity in reaction and reducing the risk of loosing data.
- Local reaction increases the chances that a problem specific solution is found (i.e repeat calculation instead of flushing pipeline)

Hence, we express our goal as:

$$Pr(sol) \rightarrow 1 \tag{3.1}$$

Then, $Pr(sol)$ is a function of the detection capabilities, as well as of the "effective distance" between the detection and the reaction. Consequently, its value gets smaller as this distance increases. In our approach, a solution is considered better if it is local, and the probability for a solution becomes smaller as the delay increases. If $Pr(Sol)$ goes to zero, it does not necessarily mean that no solution at all is found, but rather that no problem specific

solution was found (i.e a the reaction was not local). Based on that thought, we model the $Pr(Sol)$ as:

$$Pr(sol) = Pr(Detection\ and\ d_{opt}) = \frac{1}{\delta}e^{-d} \tag{3.2}$$

where

- $\frac{1}{\delta}$: the fraction of all transitions having detection capabilities
- d: the effective distance between detection and reaction (delay)

With the Poisson model ($Pr(d_{opt}) = e^{-d}$) for the probability of d_{opt}, we describe the probability of having a local solution (d=0), when the average effective distance is d, under the assumption that the occurence of an error is independent of the occurence of another error in the same region. According to [FP92], if we introduce a factor Γ, called the *coefficient of variation*, we can physically interpret it as a coefficient of spatial coupling between errors. In the case of the Poisson model above, we have $\Gamma = 0$ and $Pr(d_{opt}) = e^{-d}$. If Γ increases, still according to [FP92], we should use the model $Pr(d_{opt}) = (1 + \Gamma d)^{-\frac{1}{\Gamma}}$ (which is equivalent to the Poisson distribution if $\Gamma = 0$). However, below we will restrict ourselves to the first model for the sake of simplicity.

We express the effective distance d as a function of the analysis and detection capabilities and the distance (delays) between layers. Let us consider a particular path as shown in Figure 3.1. The path is an on-chip path.

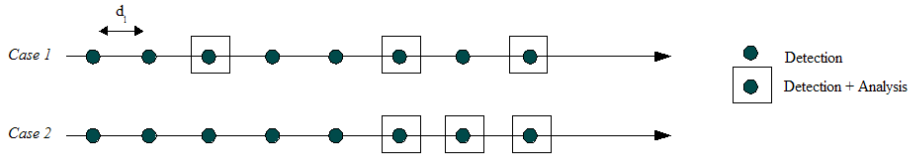


Figure 3.1: Two paths with different analysis distributions

In *Case 1*, analysis capabilities are (more or less) uniformly distributed along the path. In that particular setting, the average delay d_0 is minimized and can be written as:

$$d_0 = \frac{1}{|T|}\sum_{1}^{\left\lfloor \frac{n_\delta}{n_\alpha} \right\rfloor} id_l n_\alpha + (n_\alpha mod\ n_\delta)d_l \tag{3.3}$$

In *Case 2*, the delay is maximized and can be set to:

$$d_0 = \frac{1}{|T|}\sum_{1}^{|T|-n_\alpha} id_l \tag{3.4}$$

where:

- n_α: number of transitions with both detection and analysis/reaction
- n_δ: number of transitions with detection only
- d_l: delay from one transition to another on-chip
- $|T|$: the number of elements in the set of all transitions on-chip = # of all BBs

Let us also call the delay for going from the chip layer to the specialized component d_1, and the delay from the specialized component to the off-chip service processor d_2. Then we define the delay d as the ponderated sum of those three delays (d_0, d_1, d_2).

$$d = \frac{1}{\alpha}d_0 + \frac{1}{\beta}d_1 + \frac{1}{\gamma}d_2 \tag{3.5}$$

$$\frac{1}{\alpha} + \frac{1}{\beta} + \frac{1}{\gamma} = 1 \tag{3.6}$$

where:

- $\frac{1}{\alpha}$: probability of chip layer reaction
- $\frac{1}{\beta}$: probability of specialized component reaction
- $\frac{1}{\gamma}$: probability of off-chip reaction

This goal of ours comes at a certain cost and brings some benefits. Below we try to quantify those two values.

3.2 Cost versus Benefits

We define Ω as the ratio of costs and benefits. Straightforwardly, our intent is to minimize the value of Ω.

$$\Omega = \frac{Cost}{Benefit} = \frac{f\{C_{Area}, C_{Power}, C_{Latency}\}}{g\{B_{Latency}, B_{Reliability}\}} \tag{3.7}$$

Below, we will define the various costs and benefits separately.

3.2.1 Area

Implementing the MoHiDoC framework leads to a cost in terms of area. We make the assumptions that a transition can have either detection (A_δ), or detection, analysis and reaction (A_α), or nothing. Furthermore, we associate a certain area cost with the tokens $((A_{tok})$, i.e token places and additional bandwidth. We tried and expressed the cost as a percentage relative to the

initial area per transition. Based on that consideration, we define the cost in terms of area C_{Area} as:

$$C_{Area} = \frac{1}{|T|}\left[\sum_{i=1}^{|T|}(A_{\delta_i} + A_{\alpha_i} + A_{tok})\right] \quad (3.8)$$

$$\leq \frac{1}{|T|}[|T|\frac{1}{\alpha}[\max(A_{\alpha_i}) + \max(A_{\delta_i})] + |T|\frac{1}{\delta}\max(A_{\delta_i}) + |T|A_{tok}] \quad (3.9)$$

$$= \frac{1}{\delta}A_{\delta} + \frac{1}{\alpha}(A_{\alpha} + A_{\delta}) + A_{tok} \quad (3.10)$$

where the inequality follows from the fact that:

$$\sum_{i=1}^{n}(x_i + y_i + z_i) \leq n\ (\frac{1}{a}max(x_i) + \frac{1}{b}max(y_i) + \frac{1}{c}max(z_i)).$$

where a, b, and c are the fraction of non-zero values of x, y, and z respectively. In the third equation, for the sake of simplicity, we omit the max(). As previously:

- $\frac{1}{\alpha}$: fraction of transitions having detection, analysis and reaction capabilities
- $\frac{1}{\delta}$: fraction of transitions having detection only

Note that we considered, in Equation 3.5, that the probability of having on-chip reaction was equivalent to the fraction of analysis and reaction enabled transitions.

3.2.2 Power

Additional hardware, in addition to increasing the area costs, also leads to a larger power consumption. We assume that detection is always running, which results in a maximal additional power consumption per transition of Po_{δ}. On the other hand, analysis is only activated in case an error was detected, hence the analysis enabled transitions can either have a maximal additional consumption of Po_{sb} (standby) or Po_{α} (analysis and reaction) [FEL01, HPB02]. Note that for the sake of simplicity we omit resynchronization costs (i.e costs for switching from standby to active mode). Below we consider the particular case when N_{max} errors (LSEU) occur simultaneously. This being the case leading to the highest power consumption. Consequently, we set the power cost C_{Power} to:

$$C_{Power} = \frac{1}{|T|}\left[Pr(N_{max})Po_{\alpha} + \sum_{i=1}^{|T|-N_{max}} Po_{sb_i} + \sum_{i=1}^{|T|}(Po_{\delta_i})\right] \quad (3.11)$$

$$\leq \frac{1}{|T|}\left[|T|(\frac{1}{\alpha} + \frac{1}{\delta}Po_{\delta} + \frac{1}{\alpha}\left[(|T| - N_{max})Po_{sb} + N_{max}Po_{\alpha}Pr(N_{max})\right]\right] \quad (3.12)$$

$$= \frac{1}{\delta}Po_{\delta} + \frac{1}{\alpha}\left[Po_{\delta} + (1 - \frac{N_{max}}{|T|})Po_{sb} + \left[\frac{N_{max}}{|T|}Po_{\alpha}\right]Pr(N_{max})\right] \quad (3.13)$$

where the symbols are the same as in Equation 3.8 and next and the inequality follows from the same formula as above.

3.2.3 Latency

The MoHiDoC framework's main benefit in our sense is the small latency for reaction. Not withstandingly, the framework also adds a certain latency overhead. Below we try and quantify those two terms.

Costs

During normal operations, the framework should add as little latency as possible. We characterize the additional latency, relatively to the initial latency, simply as the following ratio:

$$C_{Latency} = \frac{t_{framework}}{t_{normal}} \tag{3.14}$$

where:

- $t_{framework}$: latency with the framework
- t_{normal}: latency without the framework

We will consider the relative increase over a long run.

Benefits

On the other hand, the framework will considerably reduce the reaction latency in case an error should occur. As explained above, our objective is to make the reaction as local as possible. Thus, we define the benefit in terms of latency as the ratio between the time necessary for a service processor reaction t^{sp}_{max} and the time necessary for a local reaction t^{local}_{max}, multiplied by the probability that the reaction is local $Pr(sol)$ (as defined in Equation 3.2).

$$B_{Latency} = Pr(sol)\frac{t^{sp}_{max}}{t^{loc}_{max}} = d_2 Pr(sol) \tag{3.15}$$

3.2.4 Reliability

Finally, we quantify the benefits in terms of reliability. Our intent is to show that our framework considerably reduces the Mean Time Between Failures (MTBF).

Definition of Reliability

Let us define the random variable **t** as the failure time of a particular item under consideration. Thus the probability of failure as a function of time is given as $Pr(\mathbf{t} \leq t) = F(t)$, which is simply the definition of the failure distribution function. We can define the "reliability", which is a probability of success in terms of $F(t)$ as:

$$R(t) = 1 - F(t) = P(\mathbf{t} > t) \tag{3.16}$$

where

- $R(t)$: Probability of surviving at time t

According to [Sho68], if we have a constant hazard (failure rate) λ, the reliability can be rewritten as:

$$R(t) = e^{-\lambda t} \approx 1 - \lambda t \ for \ \lambda t \ small \tag{3.17}$$

and furthermore, we can approximate the $MTBF$ as:

$$MTBF = \frac{1}{\lambda} \tag{3.18}$$

If we consider a system divided in several subsystems, in our case transitions, the overall reliability, under the assumptions that the subsystems are independent, is given by:

$$R_s(t) = \prod_{i=1}^{|T|} R_i(t) = e^{-\sum \lambda_i t} \tag{3.19}$$

We are interested in single event upsets in the combinatorial logic, and consequently the hazard can be set to a constant value equivalent to the SEU rate, which we will call SER (Soft Error Rate). Hence, we have:

$$\lambda = SER \tag{3.20}$$

The SER is not easy to estimate. Below we show an attemptive way to estimate that value.

SEU Rate in Combinatorial Logic

We refer to [Ngu02] and [SKBA02] for this estimation. The SER (affecting the system) is defined as the product of the real SER (all soft errors, including the ones which are not latched up), multiplied by the time derating and logical derating. This means that we consider that a SEU occurred in the combinatorial logic when a transient error in the result of a logic circuit is subsequently stored in memory elements (latches) and affects valid data. One can set:

$$SER = SER_{real} \times TD \times LD \tag{3.21}$$

where

- TD: time derating, i.e fraction of time the circuit is sensitive to SEU
- LD: logic derating, i.e probability to corrupt valid data

The LSER, which we plug-in for SER_{real} is estimated to be approximately 2% [SKBA02]. The LD can range from 0% to 80% [Ngu02]. Alternatively, we will estimate the logic derating as $1 - Pr(sol)$, with $Pr(sol)$ as defined in Equation 3.2. This latter function corresponding, as explained above, to the probability that a solution is found to a SEU.

The time derating is trickier to estimate. As in [SKBA02], we set $TD \propto frequency$. Let us consider clock cycles as shown in Figure 3.2. ω is the

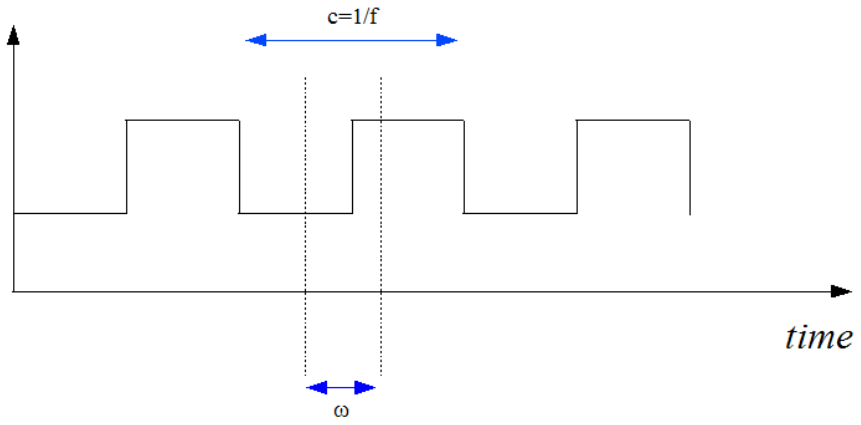


Figure 3.2: Clock cycles and latching window

latching window, i.e the time during which a pulse can be latched up. A SEU pulse can be latched up if its length l is at least as large as ω. Consequently:

$$Pr(latched) = \begin{cases} 0 & , \ l < \omega \\ \frac{l-\omega}{c} & , \ \omega \leq l \leq c + \omega \\ 1 & , \ l > c + \omega \end{cases} \tag{3.22}$$

As a function of $f = frequency$, we have $\frac{l-\omega}{c} = (l - \omega)f$. We estimate the values of ω and l as:

- $\omega \approx \frac{c}{5} = \frac{1}{5f}$
- l:$0 \leq l \leq \omega + c + x$

Note that we did not find any results for the value of l and assume that it is at most larger than $\omega + c$ by a small amount x. In [Ngu02], the TD is estimated to be 50% for most designs, which would correspond to $l = \frac{0.5+0.2}{f}$ (from equations above). However, as we will see hereunder, this parameter fortunately cancels out in our evaluation of the reliability benefits.

Benefit

As mentioned at the beginning of that subsection, the benefit in terms of reliability is pointed out through the MTBF. We define this benefit as the ratio between the new (with MoHiDoC) MTBF and the old MTBF (without MoHiDoC). So we can write:

$$B_{reliability} = \frac{MTBF_{MoHiDoC}}{MTBF_{old}} \tag{3.23}$$

$$= \frac{\lambda_{old}}{\lambda_{MoHiDoC}} \tag{3.24}$$

$$= \frac{SER_{old}}{SER_{MoHiDoC}} \tag{3.25}$$

$$= \frac{SER_{real} \times TD \times LD_{old}}{SER_{real} \times TD \times LD_{MoHiDoC}} \tag{3.26}$$

$$= \frac{LD_{old}}{LD_{MoHiDoC}} \tag{3.27}$$

where, as defined above:

- LD_{old}: can range from 0% to 80% as in the litterature [Ngu02]
- $LD_{MoHiDoC}$: $1 - Pr(sol)$, as in our model

Intuitively, one can understand this ratio as the ratio between the probability to corrupt valid data without and with MoHiDoC.

3.3 Evaluation

In this section we will evaluate the Ω cost over benefits ratio defined in Equation 3.7. We define the $f\{\}$ *and* $g\{\}$ functions in the latter equation as sums, i.e. Ω is the sum of the costs over the sum of the benefits. We consider the different costs and benefits as percentages (e.g. C_{Area} is the additional percentage of area required). Hence, we are consistent in terms of units. Thus, rewriting the aforementioned equation, we obtain:

$$\Omega = \frac{C_{Area} + C_{Power} + C_{Latency}}{B_{Latency} + B_{Reliability}} \tag{3.28}$$

where C_{Area}, C_{Power}, $C_{Latency}$, $B_{Latency}$, $B_{Reliability}$ are defined in Equations 3.10, 3.13, 3.14, 3.15, 3.27 respectively.

We plot Ω as a function of various parameters. Table 3.1 lists those different variables. We vary one of them while the others have the value listed in Table 3.2. Note that in this table we list the parameters defined in Equations 3.10, 3.13, 3.14, 3.15, 3.27.

Figure 3.3 shows the resulting graphs. As one can see, Ω is higher when all the transitions have local analysis capabilities (low α), and when the

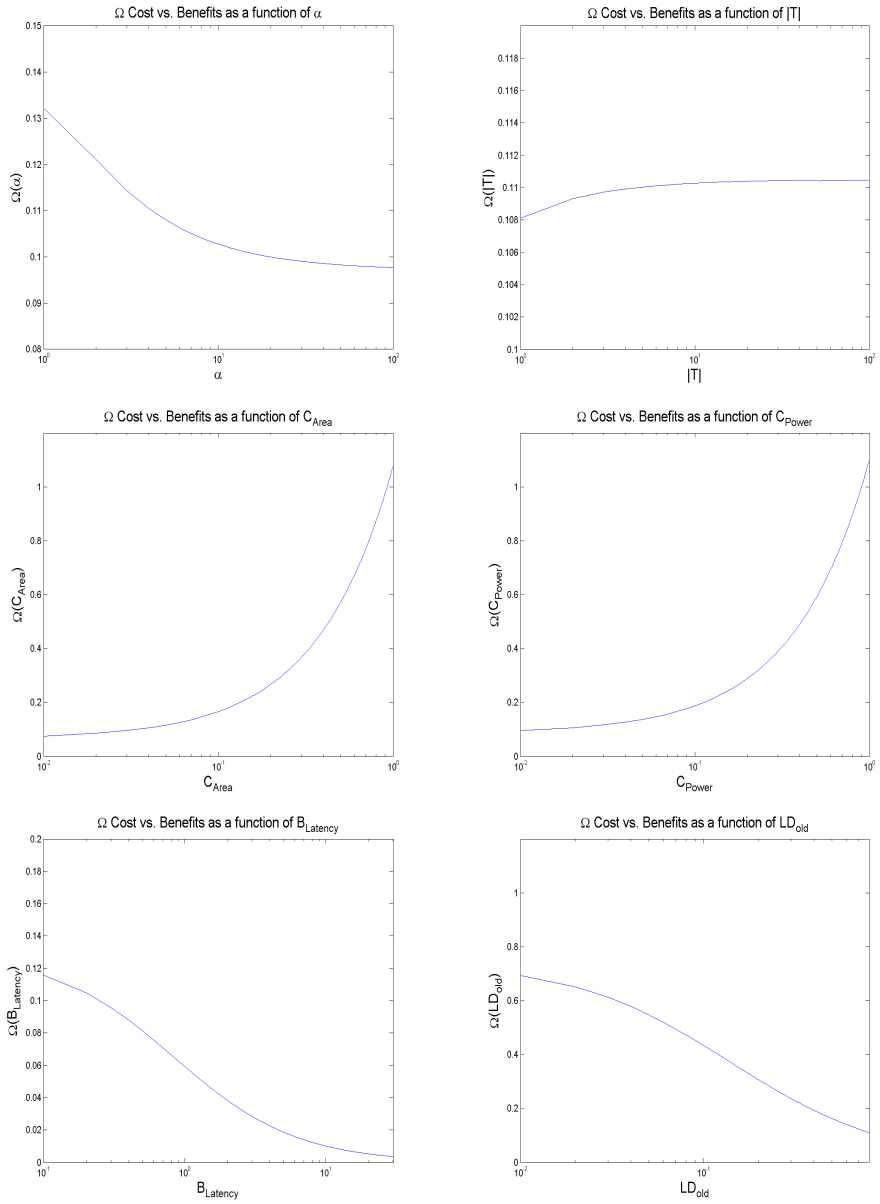


Figure 3.3: Cost vs Benefits graphs

Variable	*Range*	*Unit*
α	$[1:100]$	Fraction of transitions (BB) with analysis
$\|T\|$	$[1:100]$	Number of transitions (BBs)
C_{Area}	$[0:1]$	Additional %-tage of area
C_{Power}	$[0:1]$	Additional %-tage of consumed power
LD_{old}	$[0:0.8]$	Logic Derating
$B_{Latency}$	$[0:30]$	Relative gain in terms of Latency

Table 3.1: Varables for the evaluation of Ω

Parameter	*Value*	*Equation*
α	4	3.10,3.13,3.2
δ	1.2	3.10,3.13,3.2
A_α	0.01	3.10,3.13,3.2
A_δ	0.03	3.10,3.13,3.2
A_{tok}	0.01	3.10
Po_α	0.01	3.13
Po_{sb}	0.01	3.13
Po_δ	0.02	3.13
$\|T\|$	100	3.10,3.13,3.2
N_{max}	$\left\lceil \frac{\|T\|}{50} \right\rceil$	3.13
$Pr(N_{max})$	0.05	3.13
$C_{Latency}$	0.026	3.14
d_0	0.01	3.2
d_1	0.1	3.2
d_2	3	3.2
LD_{old}	0.8	3.27

Table 3.2: Parameters for the evaluation of Ω

number of transitions is high (high $|T|$). The interpretation for that result is that the cost of having many local analysis and reaction capabilities is slightly more important than the related benefit in our model. The second result, i.e. that the number $|T|$ should be low, tends to confirm that result as it indicates that the cost of granularity is high. Note however that both curves converge toward a constant value and that above $\alpha = 10$ and $|T| = 10$, Ω remains almost constant. Further, it is interesting to point out the fact that Ω varies very little when the C_{Area} and C_{Power} remain below 10% of the total area and power respectively. This means that implementing the framework is advantageous as long as we remain below this threshold. It is also important to notice that Ω decreases very rapidly as the $B_{Latency}$ increases. This means that if the delay to an outside processor is high, the benefit of MoHiDoc rapidly becomes important. Finally, and maybe most importantly, one can note that Ω starts decreasing dramatically when

the Logic Derating (LD) is higher than 10% (and consequently the SER increases). This result shows the usefulness of MoHiDoC in the future, when this rate will considerably increase due to the factors listed in Chapter 1.

The paragraph above revealed some of the inherent tradeoffs of our cost versus benefits model. The rules of thumb one can extrapolate from this evaluation are the following:

- The cost of having high granularity for analysis decreases rapidly and becomes almost constant for less than $\frac{1}{10}$ of transitions with such capabilities. However, the variation in Ω is small.
- Similarly, a very low granularity in terms of number of transitions leads to a slightly lower Ω.
- If the additional power consumption and area costs remain below 10%, Ω does not increase significantly. This means that the costs outpace the benefits only above that threshold.
- The higher the SER, the higher the benefit of implementing the cost versus benefits model, even if the LD is as small as 1%.

In the following chapter, we will plug-in values from simulations and literature into the model developed in this chapter in order to quantify the added value of our framework.

Chapter 4

Results

In this chapter, we will first show how the MoHiDoC framework can be used to model known component specific error reaction approaches. Then, we will integrate values from simulation and the known approaches into our mathematical model from the previous chapter.

4.1 Comparison to Razor and Diva

In this section, we will show how existing component specific error handling mechanisms can be modeled using our BB model and point out the main differences. It is important to note that those techniques (Diva and Razor) were not developed for LSEU. Nevertheless, we believe that it is useful to relate our theoretical approach to previous existing approaches which were tested and implemented. This process adds credibility to MoHiDoC by showing that similar approaches can be implemented in practice at a reasonable cost. Moreover, MoHiDoC provides means to further expand those mechanisms and to link together different type of mechanisms.

4.1.1 Diva

The DIVA (Dynamic Implementation Verification Architecture) [Aus99] fits nicely to the BB model, as shown in Figure 4.1. According to the DIVA architecture, the instruction, result and input of the DIVA core (normal core before commit) are input to the checker. The checker recalculates the result (EX) and re-performs the register and memory accesses (RD) and compares (CMP) and checks (CHK) the new results and inputs with the DIVA results (2a) and inputs (2b). It also checks whether the timer for the instruction has expired (2c). If there is a problem it performs a reaction (2a-c). Otherwise it commits the results of the DIVA core (3).

When the error is in the result, it recovers the error by committing the new result (4). When the error is in the memory/register communication,

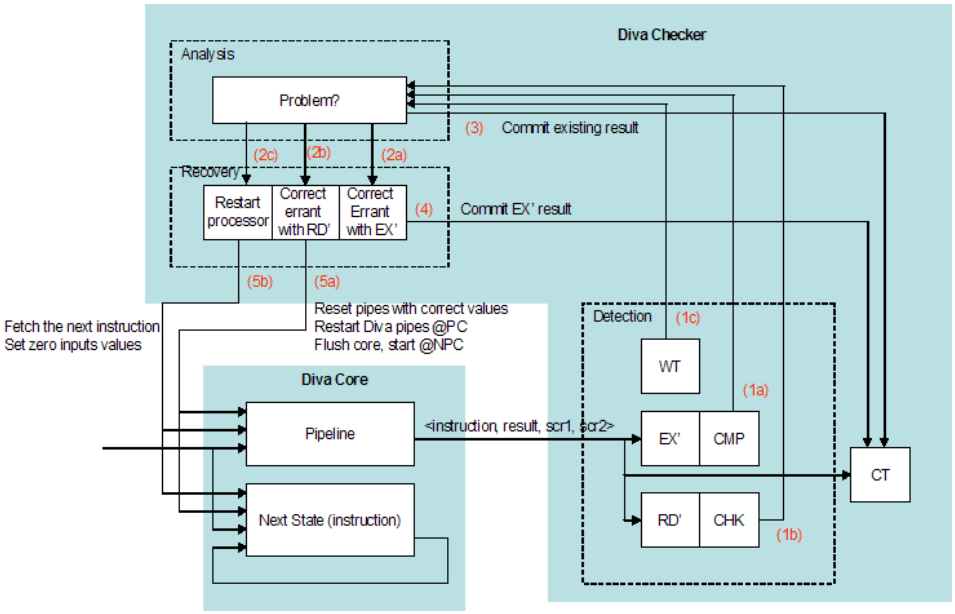


Figure 4.1: MoHiDoC and Diva

it resets the DIVA pipes with the new values and restarts the DIVA core at the next instruction (5a).

When the problem is in the timing (the DIVA core is deadlock) it restarts the processor with zero values at the next instruction. The processor will give a wrong result, which will be corrected in the next checking of the checker. As one can see, Diva is a subset of our BB model. The main differences of DIVA and our BB model is that:

1. The next state (instruction) logic has no error detection correction, because the authors believe that the instruction can be fetched out of order without a problem.

2. There is no input (6), and that is because the DIVA architecture is considered isolated and no control input from previous logic is considered. At the same time the input is not saved.

3. There is no control information output, for the same reason as above.

Note that the Diva approach is designed at the pipeline level.

4.1.2 Razor

The Razor [EKD+03] approach, shown in Figure 4.2 has been developed to detect and recover from timing (delay path) errors in a processor pipeline. In the bottom of the figure we show its representation through the BB model.

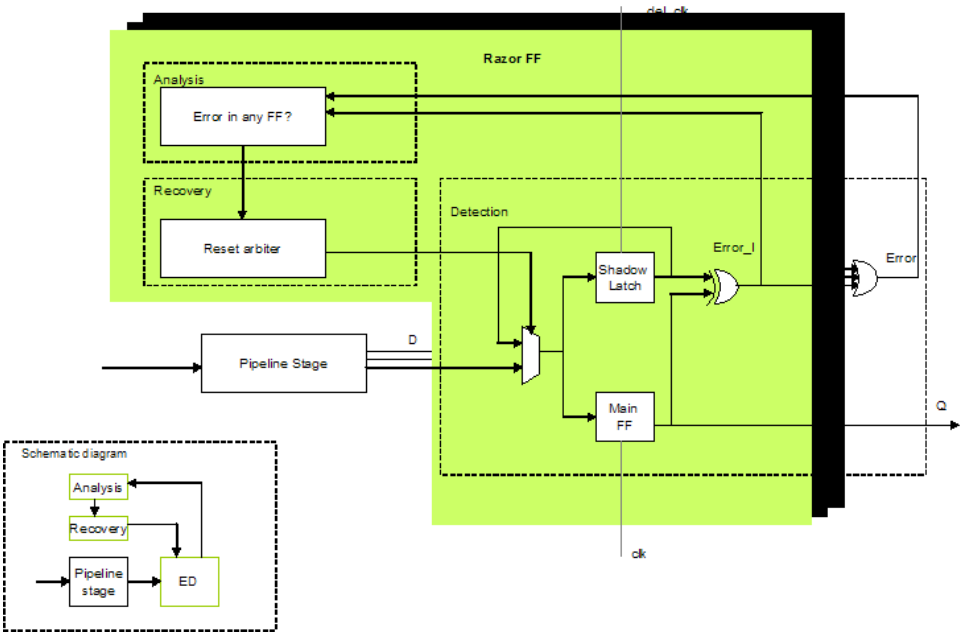


Figure 4.2: MoHiDoC and Razor

The error detection logic consists of a Main FF and a Shadow Latch. The shadow latch is clocked with a delayed clock. The output of a pipeline stage (e.g., instruction fetch (IF), instruction decode (ID), etc) is stored in the Main FF. In case there is a delayed response from the pipeline stage this is latched by the shadow latch and the comparator circuit will indicate error. The error leads to a change in the arbiter in the input of the Razor FF which will lead to get as input the output of the shadow latch. Thus with a cycle delay the output of the Razor FF has the right data. The local errors are ORed in order to ensure that the data are restored in all Razor FFs accordingly.

In Figure 4.3 we show the actual Razor reaction pipeline. With dotted-dashed (1) arrows we indicate the standard Razor reaction, as it is described on the previous page. When an (timing) error is identified in the pipeline the Razor pipeline reaction mechanism is performing the next steps:

1. The next stage is annulled through a bubble input in the next Razor FF (dashed lines “2”). This bubble can be correlated to the control information send from one transition to the next. Only though the bubble information does not change the actual processing but just annuls its results.

2. The next current and forwards stages are flushed (dotted arrows “3”).

The idea is that the instruction to which the timing error occurred is re-

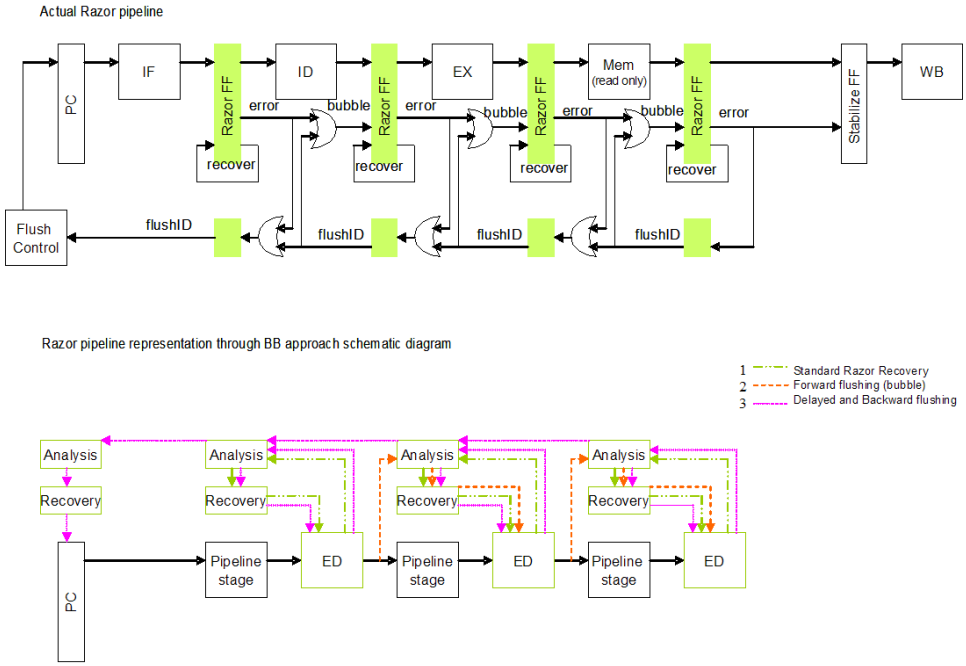


Figure 4.3: MoHiDoC and the Razor Pipeline

covered and finished, but all subsequent already started in the pipeline are flushed and the pipeline starts anew from the one next to the error occurring one.

Note that the Razor approach is designed at the gate level and applied at the pipeline level.

4.2 Simulation

In order to evaluate certain parameters of our costs versus benefits models (Chapter 3) and the effect of our framework on a real system, we developed a probabilistic simulation model of an InfiniBand (IB) Host Channel Adapter (HCA). An overview of that model is given in Appendix A. Hereunder we explain our experimental setup and present the results we obtained.

4.2.1 Setup

We implemented a particular scenario on top of our IB HCA simulation model. In this subsection we will describe that scenario and in the next subsection we will present the practical results we obtained.

We chose to implement the following scenario (all names refer to terms used in the Appendix):

1. An address translation (AT) is requested by the the Receive Process (RP) of an HCA from the Control Unit (CU).

2. The CU returns the requested address (the entry could have been cached or requested from Memory) to the RP. During the transfer between the CU and the RP, a particle strikes and corrupts the data. Practically, the data is randomly marked as being correct or corrupted (nb bits, with a higher probability for lower number of bits). One can set the average delay between LSEUs.

3. The RP treats the received address as if it were correct all the time and stores the data from the IBin Buffer at this location in memory. If data marked as corrupted is stored, we consider that the data integrity was challenged.

Furthermore, we also implemented some simple reaction mechanisms. After the dedicated bus between the CU and the RP and the address processing in the RP, we implemented simple detection and analysis capabilities, as well as on a specialized component on-chip. For the sake of simplicity, we decided that the local reaction mechanisms could recover up to nb (1 in our case) corrupted bits, and else pass the error to the specialized component. The specialized component reaction is a repetition of the address translation. In Figure 4.4 we illustrate that scenario and show the BB and PN representation. Note that we consider a delay of 1 cycle for on-chip reaction in the logic (local reaction) and 10 cycles for specialized component reaction. Unfortunately, we did not find any benchmark values for those delays. Our estimation is that local reaction can be done in the same clock cycle and specialized component reaction should add less than 10% of delay.

4.2.2 Practical Results

We first run the simulation for 1,000,000 cycles without having the framework enabled in order to evaluate the MTBF for a certain LSEU rate (*test* 1). Then we run the simulation again for 1,000,000 cycles, this time with the framework enabled and estimated the cost in terms of latency for a long run (*test* 2). Note that this cost only corresponds to a particular scenario, and could be different in other parts of the chip. Unfortunately, there was not enough time to develop a complete simulation environment and consequently we limited ourselves to an example case. The results are exposed below and incorporated in the mathematical model in the next Section (4.3). Note that we used the setup described in Table A.2.2.

Test 1

In this first test run (framework off), we obtained the following results:

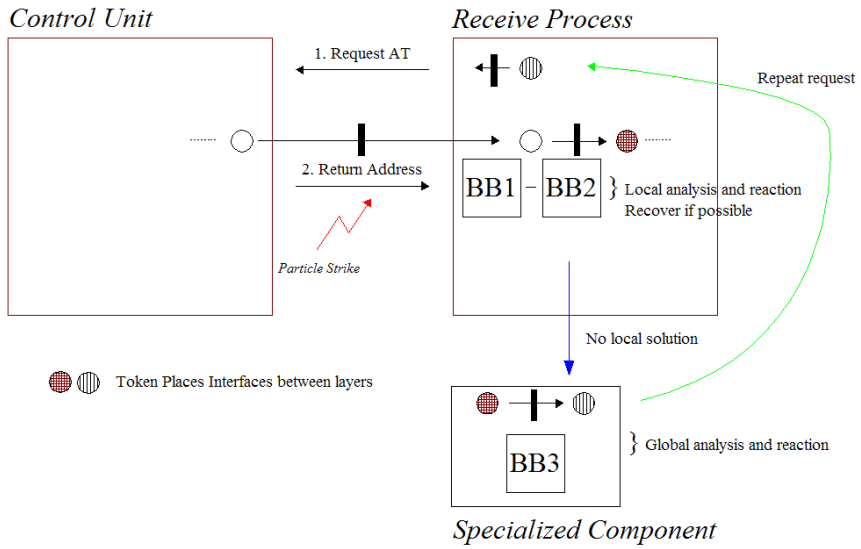


Figure 4.4: Illustration of the LSEU Scenario

1. Average Time to receive a packet (time in RP): 108 cycles
2. Number of packets treated: 8389 (100% of packets received)
3. Average size: 1321 Bytes
4. Average Time to transmit a packet (until in IBout Buffer): 100 cycles
5. Number of packets transmitted: 9942
6. Average size: 767 Bytes
7. Mean Time Between Failure (MTBF): 68729 cycles
8. Percentage of packets affected: ≈6.9% (575 on rx side)
9. Total number of errors injected: 1978 (≈ 0.002 errors/cycle)

Test 2

In the second test run (framework on), we obtained the following results:

1. Average Time to receive a packet (time in RP): 108 cycles
2. Number of packets treated: 8390 (100% of packets received)
3. Average size: 1321 Bytes

4. Average Time to transmit a packet (until in IBout Buffer): 100 cycles
5. Number of packets transmitted: 9956
6. Average size: 769 Bytes
7. Mean Time Between Failure (MTBF): Infinity
8. Errors corrected in DoC (more than 1 bit): 24
9. Total number of errors injected: 2019 (≈ 0.002 errors/cycle)

One can notice that the average delay per packet in the receive side processing is identical over a long run whether the framework is running or not. We explain it by the fact that the processing time per packet is variable, determined by the random model of our simulation (see Appendix A), and that only a small number of packets are affected by errors, resulting in a negligible overhead over a long run.

On the other hand, if we consider the delay per corrupted packet, the average delay is approximately 2.8 cycles, which represents $\frac{2.8}{108} = 0.026 = 2.6\%$. Below we will consider this particular value. Table 4.1 exposes the values which we obtained from our simulation model and we will use in the following Section.

Benefit/Cost	*Value*
Latency $C_{Latency}$	2.6%
MTBF $MTBF_{old}$	68729 cycles

Table 4.1: Costs obtained from the simulation model

4.3 Correlation to the Mathematical Model

In this section, we will plug in values from the simulation (subsection 4.2.2) and from the literature into the mathematical model derived in Chapter 3.

Our simulation model obviously did not allow us to estimate all the costs involved in a complete "costs versus benefits" estimation. The parameters we obtained are listed in Table 4.1. On the other hand, we obtained some parameters from the literature, i.e from the Diva and Razor Papers. Below, in Table 4.2, we list the parameters we extracted from various publications. Note that for Diva ([Aus99]), we use values in [WA01](from the same author) which describes the implementation costs of *dynamic verification schemes*, such as DIVA. Unfortunately, not all parameters were available.

We estimate the C_{Area} for Razor as being between 1 and 10%. Hence we obtain costs for Razor and Diva of:

$$Cost_{Diva} \approx 0.105 \tag{4.1}$$

$$Cost_{Razor} \approx 0.063 : 0.163 \ (error\ case) \tag{4.2}$$

Approach	*Parameter*	*Value*	*Reference*
Diva	C_{Area}	6%	[WA01]
”	C_{Power}	1.5%	[WA01]
”	$C_{Latency}$	3%	[Aus99]
Razor	C_{Power}	3.1% (error free)	[EKD+03]
”	C_{Power}	4.1% (error)	[EKD+03]
”	$C_{Latency}$	≈ 0[1]	[EKD+03]

Table 4.2: Costs obtained from the literature

For both approaches, we set $B_{Latency} = 0$, since their implementation does not add any benefit in terms of latency. Again, the “reliability” benefit is tricky to estimate. For the sake of comparison, we set the $SER_{MoHiDoC}$ in Equation 3.25 to one, since it is impossible to quantify for Razor and Diva, and use Equation 3.21 to define $SER_{old} = SER_{real} \times LD \times TD$. As before,

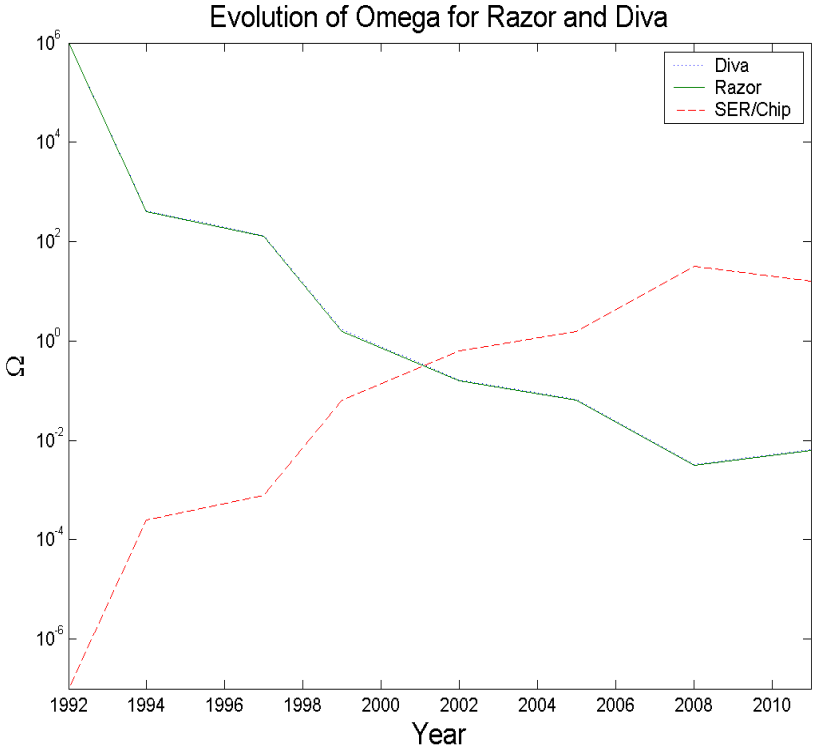


Figure 4.5: Evolution of Ω for Razor and Diva

[1]We assume that this value is averaged over a long run and that similarly to what happens in our model, the overhead costs are negligible in that case. In case an error occurs, pipeline operations are delayed one cycle, resulting in a relative cost of $\frac{7}{6} \approx 1.2\%$

we define $TD = 0.5$ and arbitrarily set $LD = 0.5$. For SER_{real}, we use the values presented in [SKBA02] for SER/Chip in the logic and obtained through simulation. In Figure 4.5, we present the evolution from 1992 until 2011 (Diva and Razor almost identical). One can see that Ω decreases by nine orders of magnitude in less than twenty years. This result infers that whatever approach is taken, as expected the importance of dealing with SEU will become increasingly high.

If we assume similar costs for MoHiDoC in terms of area, power and latency, we immediately see that in order to have better performances than the two approaches above, MoHiDoC should either considerably reduce the logic derating and/or offer a great benefit in terms of latency. The simulation has shown us that $C_{Latency}$ is very similar to that of Diva and Razor.

While it is difficult to gauge the other terms without implementing an actual MoHiDoC framework on a real chip, we feel that those objectives should be achievable for at least two reasons:

- First, MoHiDoC allows the integration of the above mentioned framework at no or very small cost while in addition offering several levels of control and miscellaneous reaction mechanisms, thus increasing the overall reliability.
- Second, MoHiDoC allows error propagation, hence it is possible to save hardware for analysis and reaction in less critical parts of the chip.

In the next chapter we will conclude by pointing out advantages MoHiDoC could potentially bring and explaining how this work is integrated in a larger scale project for autonomous computing.

Chapter 5

Conclusions

In this final chapter we will first try and summarize the benefits and advantages the MoHiDoC framework offers. Then,we discuss future work and related projects. Finally, we will conclude on a personal note.

5.1 Advantages

We believe that the MoHiDoC framework can offer many benefits to designers willing to implement it. We enumerate a few below.

1. The method leads to a standardized interface and mechanism for reporting problems and getting them out to a higher-level management authority, which has always been very difficult.

2. While the method is designed for on-chip autonomic error detection, analysis and reaction, it can be adapted to different logic layers of control both on and off-chip, and thus it can be used to organize different hierarchies of control and different levels of information.

3. The method produces a DoC architecture that is flexible, lightweight, and easy to implement by subsystem implementers.

4. The actual problem detection and reaction functionality may have to be implemented by subsystem designers; fact that allows flexibility but also future proof, since new methods can be adopted.

5. The method allows also flexibility in the cost, since it is up to the system designer to decide where he will place the different control components (detection, analysis and reaction) and how he will link them together.

6. While the method was designed for LSEU, it is valid for all sorts of errors. Currently chip verification, code debug during system bring-up

and field problem analysis are extremely time-consuming. The method provides an architected framework that may reduce significantly this time.

7. The method enhances the first error data capture mechanisms and enables the "self healing system approach".

8. The method assists in optimizing the design point regarding performance, chip size, and costs.

5.2 Future Work and Related Projects

As mentioned at the very beginning of this report, MoHiDoC is part of a much larger project, namely the "Diagnosis-on-Chip" (DoC) project at IBM. The DoC framework is intended to make chips:

- *reliability aware*: capable of autonomously reducing the error rates.
- *performance aware*: capable of autonomously optimizing their own performance.
- *power aware*: capable of autonomously optimizing their power consumption.

We think that integration of the MoHiDoC framework into DoC will allow to reduce costs and to link features, such as temporal frequency alteration mentioned in Chapter 2.

This thesis is just the initial step for the MoHiDoC framework and obviously a lot of work needs to be done. The project was voluntarily focused on a high level of abstraction, to make it as generic as possible. However, in future steps, attention should be paid to a particular implementation. The approach should also be made more proactive, i.e one could work on ways to reduce LSEU rates before they become too high. Further, more precise communication protocols between BBs need to be defined and critical issues, such as time dependencies (e.g. synchronization of regular and control data in tokens) and token dependencies (e.g. what happens if two token must be handled simultaneously in a BB) need to taken care of. The correlation to the DoC framework also needs to be specified and special attention should be paid to creating specifications for the implementation. One also needs to consider the case when false alarms should occur, and define error propagation schemes.

Concerning the cost versus benefits model, in future work one should also consider other costs such as for instance the area costs for additional I/O. Note also that we defined the probability of a solution ($\frac{1}{\delta}e^{-d}$) based on our intuition and optimization consideration, however, a sound proof of this formula and others should be derived.

Finally, and certainly most importantly, a large scale gate-level simulation environment should be developed in order to evaluate the value of the approach and validate the theoretical approach on a real system.

5.3 Personal Conclusion

Writing this Diploma Thesis at the IBM Zuerich Research Laboratory was an extremely fruitful experience. I could work on a very challenging, new and interesting problem. Further, I had the chance to be selected to present my work at a conference (IBM Academy of Technology Study on System Effects of Bit Error Rate Trend) and to modestly contribute to the preparation of a disclosure for my work, to test its patentability.

Appendix A

InfiniBand Host Channel Adapter Simulation Model

In order to obtain practical simulation results, we developed a Matlab Simulink Model of a InfiniBand (IB) Host Channel Adapter (HCA). With respect to time, it was an important part of this diploma work. In this Appendix, we will first very briefly give an overview of the InfiniBand architecture and protocol, as well as of HCAs. Then, in Section A.2, we will explain how we modeled the behavior of an IB HCA.

A.1 Infini-Band Host Channel Adapters

The InfiniBand Architecture Specification [IBA02] describes a first order interconnect technology for interconnecting processor nodes and I/O nodes to form a system area network. The architecture is independent of the host operating system (OS) and processor platform. Figure A.1 illustrates an IBA System Area Network. In this section we will describe the InfiniBand send/receive procedure briefly and explain the functionality of HCAs (of which we developed a simulation model) shown on Figure A.1. However, for a very detailed presentation, please refer to [IBA02].

A.1.1 InfiniBand Send/Receive Procedure

In Figure A.2, we show both the communication stack of IB [IBA02] and a schematic view of the IB Send/Receive procedure [Sch03].

We are considering two consumers on *node* 1 and *node* 2 (for instance two processor nodes with HCAs on Figure A.1), communicating with the IB send/receive procedure. Note that there could very well be several consumers on every node. The two consumers under consideration each have a Queue Pair (QP), consisting of a Send Queue (SQ) and a Receive Queue (RQ), as well as a Completion Queue (CQ) in their respective memory

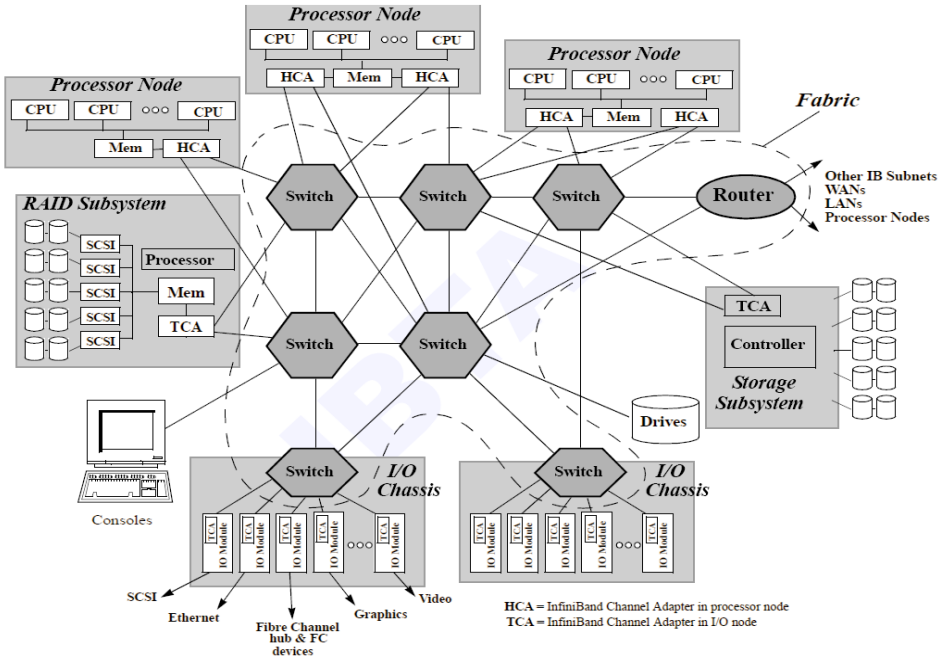


Figure A.1: IBA System Area Network

spaces. Let us assume that the consumer on *node* 1 wants to send a message to the consumer on *node* 2. The steps below correspond to the circled numbers in the lower part of Figure A.2.

1. In *node* 1, the consumer places a Work Queue Element (WQE) in its SQ. The WQE contains a list of virtual addresses corresponding to the locations where the message to be sent is stored.

2. Similarly, the consumer on *node* 2 places a WQE in its RQ. The WQE contains a list of virtual addresses where the message which will be received should be stored. The size of the WQE is agreed upon during an initialization phase not described here.

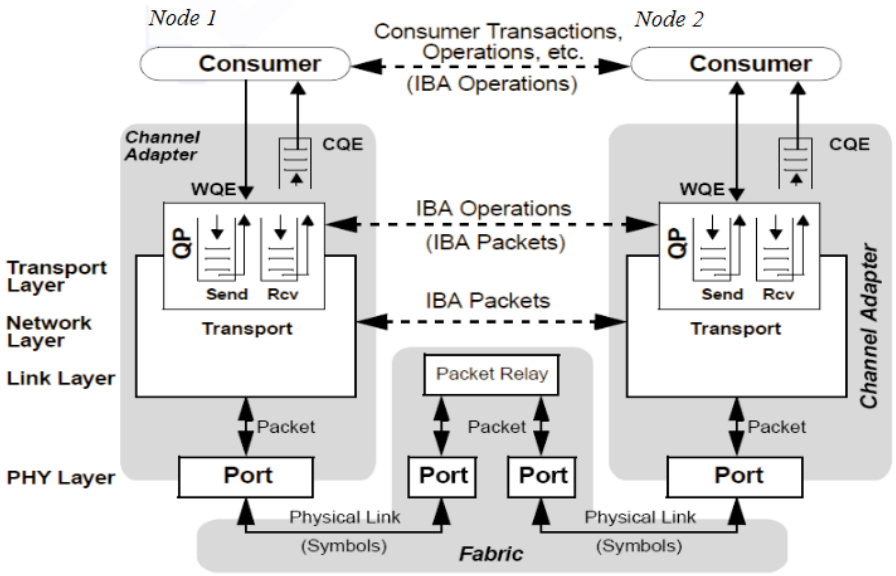
3. The hardware in *node* 1 (HCA) fetches the WQE, translates the virtual addresses and retrieves the corresponding message from memory.

4. Subsequently, the message is divided into several network packets and transmitted over the network. Note that packets are sent out in order on the network but that WQEs may be executed concurrently.

5. On *node* 2, the message is reassembled and stored in the locations described by the virtual addresses in the WQE placed in the RQ earlier.

6. Once the entire message is received, a Completion Queue Element (CQE) is placed in *node* 2's CQ.

Communication Stack



Send\Receive

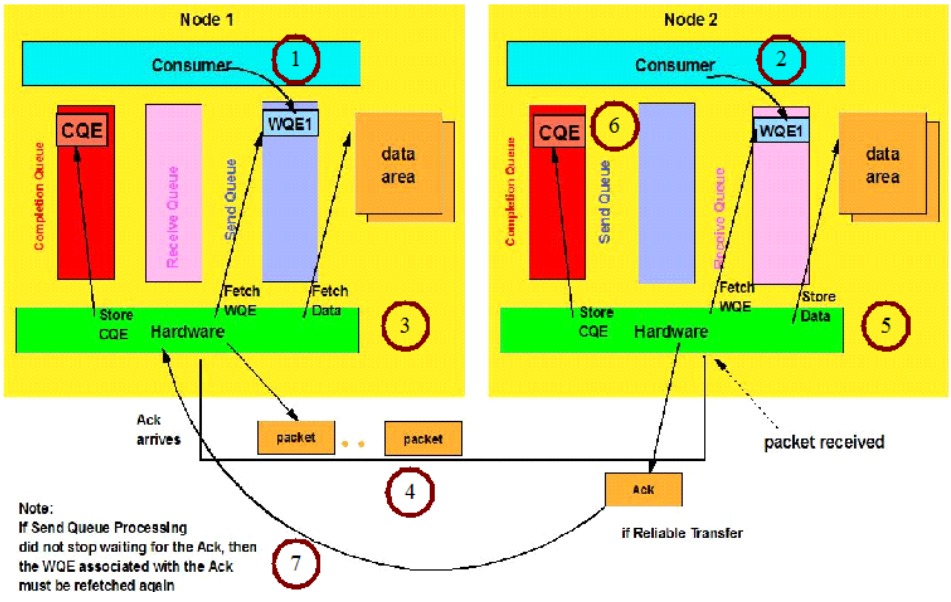


Figure A.2: IB Communication Stack and Send/Receive Procedure

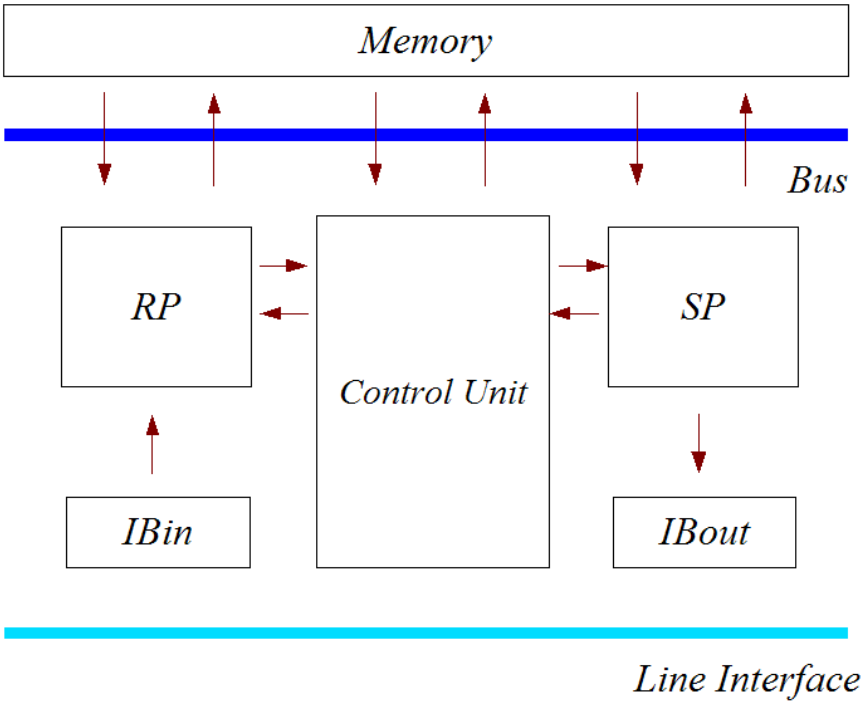


Figure A.3: Architectural view of the simulation model

7. If the transfer is "reliable", an acknowledgment is sent back to *node* 1, where a CQE is also placed in the CQ.

Below, we focus on the hardware part of the nodes, i.e. the HCA.

A.1.2 Host Channel Adapters

The HCA can be schematically subdivided in five blocks as shown in Figure A.3. In our simulation model, described later in this Appendix, we use a similar level of abstraction. Each block has several functionalities, which correspond to the tasks of a part of the chip. Hereunder, we will enumerate those different components and very briefly expose their tasks. For a detailed description, please refer to A.1.

- *IBin*: This component is responsible for getting packets from the line and storing them in a Buffer. Packets are dispatched to the RP in FIFO order, with the major exception that up to a fixed configurable number of packets from the same queue pair can preempt other packets. A packet is dispatched to the RP when the RP is done with his

previous work, and the completed packet is removed. There can be several virtual lanes (priorities) implemented (refer to A.1).

- *Receive Process* (*RP*): This component's role is to orchestrate the operations on incoming packets. When it obtains a new command from the IBin, it first checks whether this packet belongs to the same queue pair as the previous packet. If no, a request for the Queue Pair Context (QPC) of this packet is passed to the control unit, else, or once the control unit returned the QPC, we are ready to fetch a WQE from this queue pair. As mentioned previously, there can be several data segments per WQE. Thus, a WQE is not always fetched in memory. Subsequently, address translation is performed. The latter can be omitted as one data segment can be assigned to more than one packet. Address translation is a request to the control unit to translate a virtual address contained in a WQE. When we know where to store the data, we are ready to store it in memory. The actual data packet is transfered directly from the buffer in VLin to memory and this buffer freed accordingly. When a packet is the last packet of a WQE, a post CQE request is passed to the control unit after the transfer. Simultaneously, an acknowledgment packet WQE is posted to memory. Note that there can be several RPs.

- *Control Unit* (*CU*): The control unit is accessed both by the RP and the SP. First, it handles QPC and address translation requests (in the Context Unit (CTU) and Address Unit (AU) respectively). The QPC or address translation entry for a particular request can be either on chip or off chip, depending on whether it is cached or not. In the first case it can be returned very quickly, whereas in the latter case, it must be fetched from memory. The CU is also accessed by RP when a CQE is to be posted. Similarly to the QPC, the CQE can be either on chip or off chip and in the second case must access memory to retrieve it. Finally, two more elements are modeled inside the control unit, namely the Scheduler Subunit (SU) and the ack subunit (ACU). The SU's task is to dispatch new queue pairs to the SP. The ACU posts a WQE in memory when an acknowledgment is to be sent.

- *Send Process* (*SP*): This component is RP's pendent on the transmit side. When it receives a new queue pair from the SU, it first fetches the corresponding QPC. Then, fetches a new WQE. Once the first WQE for this QP is obtained, address translation is done if necessary, since there can be several packets per data segment. After this operation, data is fetched from memory. If data is not fetched from memory, an acknowledgment can be generated locally. Data or acknowledgments are transferred to the IB out buffer when space is available. Once the entire packet is put to this buffer, a corresponding header is created

and put in the header buffer when space is available. Only after this last operation can the SP start working on a new task. Note that there can be several SPs.

- *IBout*: This components handles the dispatching of packets to the lines, which it fetches packets from the header buffer.

A.2 Matlab Simulink Model

In this section we will describe how we implemented a Matlab Simulink simulation model of an IB HCA, like the one presented above. First we will explain for what purpose we developed that model and the assumptions we made. Then we will describe the implementation.

A.2.1 Goal and Assumptions

Our goal was to develop a simulation model of an HCA at a high level of abstraction. We wanted to simulate the global behavior in order to have, over a long run, occurrences of numerous different scenarios. Not withstandingly, we tried to keep the model simple, since we are not interested here in performance evaluation nor gate level interactions. We also wanted to be able, later on, to plug an SEU scenario (see Chapter 4) on top of the model, and analyze the relative increase in terms of latency, as well as to be able to estimate parameters such as the Mean Time Between Failure (MTBF). Consequently, we made a series of assumptions explained below:

- *Operating State*: Our goal was to model a high traffic, though not saturated operating state. This means that there is always work available for the chip, and that the chip can cope with this amount of work, without rejecting more than a few packets or entering a critical state. We also modeled the traffic as being bursty, i.e the rate of arrival is not steady. Furthermore, off-chip memory is not considered to be a critical issue in our case, and consequently it is considered to be unlimited. More precisely, this implies that we are not concerned by memory handling of any kind. In our model, this boils down to saying that there is always enough space to store packets, and that the time necessary for a memory operation, be it a read or write operation, is proportional to the size of the data and limited by the operations of other components.

- *InfiniBand*: On the Infini-Band level, we decided that there are always WQEs available, thus respecting the aforementioned high traffic condition. The number of queue pairs on the send and receive side are free parameters. Both on the receive and transmit sides, there can be several WQEs per queue pair, several data segments per WQE, and

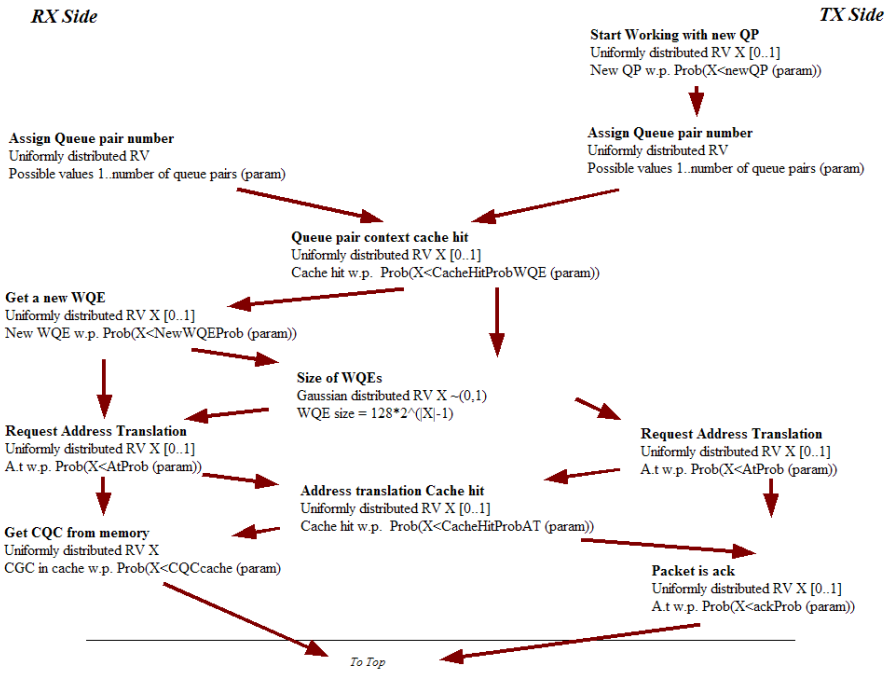


Figure A.4: Probabilistic Elements

several data packets per data segment. However there is no strict relation between those elements, but a probabilistic relation. For instance, once we are working on a certain queue pair on the receive side, there is a certain probability (configurable) that we fetch a new WQE, and within this WQE a certain probability (configurable) that we start a new data segment and consequently require a new address translation.

- *Host Channel Adapter*: We only consider one RP and one SP. Each of them only works with one port and one virtual lane (VL). The speed of the link is assumed to be 48Gbits/sec and the speed of the Memory Bus (Bus in Figure A.3) 2B@3GHz ≈32B@250MHz.

- *Probabilistic Behavior*: As inferred above, the behavior of the simulation model is highly probabilistic. Indeed, as explained at the beginning of this subsection, we are mainly interested in the overall behavior. Below we list the probabilistic elements in the model and an illustration is shown in Figure A.4:

    1. *Queue pair numbers in IBin and SU*: The number of queue pairs on the receive and transmit sides are determined by a free parameter. A queue pair number is randomly assigned to a

packet. The queue pair numbers are uniformly distributed between 1 and the number of queue pairs.

2. *Cache hits in the CTU*: The cache hit probability can be set as a parameter. The higher the cache probability, the greater the chance that we do not have to access memory to fetch the queue pair context.
3. *Cache hits in the AU*: The cache hit probability can be set as a parameter. The higher the cache probability, the greater the chance that we do not have to access memory to fetch the address translation table.
4. *Sizes of rx and tx WQEs*: We assume that WQEs are small. Consequently, the size of a transmit or receive WQE (not for acknowledgments) is gaussian distributed N (0,1) which results on approximately 68% of 128B WQEs, 27% of 256B WQEs 4% of 512B WQEs etc... up to 4KB.
5. *Address Translation*: Both in the transmit and receive side, address translation is done with a certain probability. This probability can be defined as a parameter.
6. *Acknowledgment tx*: A packet can be an acknowledgment with a certain probability. This probability is a parameter but should remain low. In case a packet is an acknowledgment, no data is fetched in memory by SP, but rather a small packet generated.
7. *New WQE in RP*: A new WQE is fetched with a certain probability to model the fact that we can work on several WQEs in a queue pair. Probability defined as a parameter.
8. *Start working on a new QP in SP*: same as previous item.
9. *Fetch CQC*: CQC fetched with a certain probability defined as parameter.

In the next subsection, we will explain how we implemented the simulation model with the assumptions above.

A.2.2 Implementation

The simulation model was implemented using Matlab Simulink, mainly because this program allows users to easily model modular systems, and offers, as well a great freedom in terms of implementation.

This section explains how a packet is treated and processed on the receive and the transmit sides. Further, we also detail here the mechanisms used to implement complex operations, such as for instance the access to the I/O Bus, and how delays are introduced to mimic the behavior of other components. The following subsubsections each treat a particular aspect of the

system. The last subsection presents certain numerical values corresponding to a real IBM chip and used during our simulations.

Receive and store packets

To receive and store packets from the line interface, the following steps are modeled:

1. Packets are simply numbers representing sizes in Bytes and are read from a text file. The text files used are traces benchmark for network traffic [TRA]. However, as the MTUs differ for IB HCAs, the packet sizes are multiplied by a constant (2.66) in order to have a maximum size of 4K. The minimum size is fixed to 256K, for the purpose of avoiding small unlikely packets. The traces used model traffic with bursts, which is what we intended to have. The number of simulation cycles necessary to read a packet from the file is proportional to the size of the packet, the relation is shown below.

$$Cycles = \frac{(IFG + Size)[Bytes] * 8}{LineRate[bits]} \tag{A.1}$$

   IFG stands for Inter Frame Gap, and is a useful parameter to regulate the intensity of the traffic. It can be set arbitrarily and its presence is justified by the fact that in a real chip, as mentioned in the subsection A.1.2, several RPs share the traffic load.

2. The IBin Buffer, intended to store incoming packets, has 544 locations, each containing up to 64Bytes of data. A free list points to the free locations. When a packet is received, we first randomly assign a queue pair number to this packet, as explained previously. Then the free list is checked to verify whether a sufficient number of buffers is available to store the packet. If there are not, the packet is dropped, else, a command number, corresponding to the number of the first location belonging to the aforementioned packet, is associated with the packet. In our model, the size of the packet and the number of buffers necessary to store it are only written in the first location of the VLin Buffer. Beside the free list and the Vlin Buffer, two other data structures, more precisely linked lists, play an important role in the storing procedure. First, the data table links buffers corresponding to the same packet, thus, when storing a packet, the different corresponding buffers (which need not be contiguous), are linked in this table and the free list is updated accordingly. Second, the command table links commands, i.e packets. This second table ensures that packets are ordered as they were received. Note that only one packet can be received at a time.

3. The unload to the RP is modeled as a FIFO queue of depth eight called “dispatch queue”. One packet is sent to the RP when the RP

done signal is received. Packets are dispatched in FIFO order, except when one or several packets from the same queue pair as the previously dispatched packet are present. Those packets have priority over other packets, and are also treated in FIFO order among themselves. To avoid starving out other queue pairs, the number of packets from the same queue pair which can be dispatched back-to-back is limited to five in our case. Five is an arbitrary value. Next we will describe how a particular packet is treated once its associated command is dispatched to the RP.

Handling a Packet

Above we explain how a packet is received, stored and dispatched to the RP. Hereunder, we shall explain what happens next:

1. A packet dispatched to the RP is simply a number corresponding to a position in the IB Buffer (the position of the first location corresponding to a particular packet = command for this packet), and the queue pair number associated with it. The RP first checks whether the Queue Pair (QP) of the current command is the same as the one of the previous command. In case it should be different, the Queue Pair Context (QPC) for this new QP must be fetched. The new QPC context cannot be fetched directly by the RP, rather, a request must be issued to the Control Unit (CU). The request is simply model as a 1 bit signal, '1' representing a request, which can last at most 1 cycle, and conversly '0' no request. Once in the CU, the request is handled in the following way: either the QPC is on chip, in which case a similar signal is sent back to the RP in the next cycle, or it is not. In the latter case, a request is to be issued to the memory. After a certain delay, the answer from the memory will also be a 1 bit signal set to '1' during one cycle. Only then will the CU answer to the RP. The cache hits are modeled probabilistically. The access to the memory through the Bus and the delays of the memory itself are explained below as they are also relevant for the transmit side and other operations on the receive side. From this point on, we will merely talk about "access to memory" and detail it later. In case the QP should be the same, this step can be skipped as it would be superfluous to refetch the same QPC.

2. Next, a new WQE should be fetched from Memory in case the previous WQE is exhausted. Again, the behavior is stochastic and interactions with the memory done on the basis of a one bit signal. The need for a new WQE is however always calculated one command in advance. This way, it is easy to model the fact that a packet is the last one belonging to a certain WQE, which is helpful later on for posting CQEs.

3. Following this step, address translation must be done if we started a new data segment. A request is issued to the CU, in the form of a one bit signal as before, and similarly to the QPC case, the address translation entry (ATE) can be either on chip or off chip. The behavior is exactly the same as for the QPC. Thus, either a answer is immediately returned to the RP, or the CU must wait for an answer from memory.

4. After address translation or if we are still in the same data segment, we are ready to transfer the data from the VLin Buffer to the memory. The size of the data packet, as we explained above, is stored in the IBin Buffer location corresponding to the command number, and the locations theoretically filled by this packet are linked in the data table. Consequently, a packet of a certain size is transferred to memory, and accordingly we free the tables and unlock these locations in the free list. Once the packet is transferred, the RP done signal can be sent to the IBin. The RP done signal is, for the sake of simplicity, an integer corresponding to the queue pair number of the treated packet. This trick makes it easier to dispatch packets from the same QP.

5. Moreover, there can be an additional step. A CQE and a WQE for an acknowledgment are to be posted if we are finished working on a WQE. Those two events are modeled as requests to the CU. A CQE has a CQC, which is to be loaded from memory if not in the local cache. The mechanism is the same as for the QPC. Then a request (1 bit) to post a CQE is sent to memory. Similarly, a 1 bit request to post a WQE is also sent.

Memory Access

Memory is accessed through the Bus by different components. However, only one component can access memory at a time. We chose to enforce this rule in a distributed manner, and not using an arbiter placed in a Bus component. Hence, every component accessing the Bus will determine by itself how long it has to wait until it can start transferring data and how long it will take to transfer this data. More precisely, a global variable, common to all components, stores the time until which the Bus is busy. To model the access to the bus, we distinguish between an access to the memory controller and an access to store data. In the first case, we say that the time until the operation can start is defined as follows:

$$CyclesToSend = (BusArb + 1) + BusBusyTill - (TCur + 1) \qquad \text{(A.2)}$$

whereas in the second case:

$$del = (delay - 1) * s * SRdAcc + f * FRdAcc + 1 + 1; \qquad \text{(A.3)}$$

or

$$del = (delay - 1) * s * SWrAcc + f * FWrAcc + 1 + 1; \quad (A.4)$$

and then

$$del = del + BusBusyTill - tCur + 1; \quad (A.5)$$

In A.2, BusArb corresponds to the Bus Arbitration time. In equations A.3 and A.4, the delay is

$$delay = \frac{SizeOfData[Bytes] * 8}{BusWidth[bits]} \quad (A.6)$$

The first equation is used for memory reads and the second equation for memory writes. In both cases there is a different execution time for first and subsequent ("second") accesses. This is modeled by the SWrAcc, FWrAcc, SRdAcc and FRdAcc variables. f and s take boolean values and are set to 1 if there is a first or/and second access.

Transmit Side

Finally, we are going to explain what happens on the transmit side. Many steps are identical to those previously described. For the sake of briefness and clarity, we will simply point out the differences and the similarities to the receive side:

1. The first step on the transmit (tx) side is that the Scheduler Subunit (SU) in the CU randomly chooses a new queue pair for the transmit side and passes this value to the SP (as an integer).

2. The next steps are exactly identical to step 1 (QPC fetching) to step 3 of A.2.2 "Handling a Packet".

3. The main difference is how messages are transfered from memory, after address translation is performed in step 2 above. There are two possibilities at this point. First, a very small packet (of arbitrary size 1 Byte) can be generated with a very low probability to simulate the fact that there are acknowledgments sent in the case of reliable transfers. The other possibility is that a new packet is fetched in memory. Since there is limited space in the output buffer, the SP is blocked until sufficient space is available in the IBout Buffer (8 slots of 512KB, even a 1 Byte packets requires one slot). When the space is available, the data is transferred to the IBout Buffer (one packet could require several slots) and a header pointing to the slots of this packet is created and queued in the header buffer in the IBout component. Headers are then in turn dispatched to the line interface which sends out the packets. Consecutively, the buffers (IBout and Header) are freed.

Sizes

Elements	*Size in Bytes*
WQE	128 (68%),256(27%),512(4%),...,4K(0%)
QPC	256
CQC	40
CQE	64
ATE	128

Frequencies

Location	*Frequency*
Chip	250 Mhz
Line	48Gbps
Bus	2B@3GHz ≈32B@250MHz

Probabilities

Numbering as in Subsection A.2.1	*Probabilisties*
1 *Queue pair numbers in IBin and SU*	8
2 *Cache hits in the CTU*	80%
3 *Cache hits in the AU*	80%
4 *Sizes of rx and tx WQEs*	same as above
5 *Address Translation*	80%
6 *Acknowledgment tx*	80%
7 *New WQE in RP*	1%
8 *Start working on a new QP in SP*	100%
9 *Fetch CQC*	1%

Table A.1: Size of elements

4. Once a packet was treated in the SP, there are two possibilities. Either we start again at step 1, to simulate the fact that the SU could request the SP to work on a new QP, or start again at step 2 to simulate the fact that we continue working on the same QP.

Parameters

This section summarizes the parameters in the model. Table A.1 contains various values for parameters in a typical simulation setup. In addition, incoming packets can be between 256 and 4KB in size and sent packets between 256 and 4KB in size, except acknowledgment which are modeled as 1B packets (and are not fetched from memory).

Bibliography

[Aus99] Todd M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 196–207. IEEE Computer Society, 1999.

[BBB+97] S. Buchner, M. Baze, D. Brown, D. McMorrow, and J. Melinger. Comparison of error rates in combinatorial and sequential logic. *IEEE transaction on nuclear science*, 44:2209–2216, 1997.

[CS91] T. Cao and A.C. Sanderson. Task sequence planning using fuzzy petri nets. In *Conference Proceedings 1991 IEEE International Conference on Systems, Man, and Cybernetics. Decision Aiding for Complex Systems*, volume 1, pages 349–354. IEEE, 1991.

[CS92] T. Cao and A.C. Sanderson. Sensor-based error recovery for robotic task sequences using fuzzy petri nets. In *Proceedings. 1992 IEEE International Conference on Robotics And Automation*, volume 2, pages 1063–1069, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[DT99] Debaleena Das and Nur A. Touba. Weight-based codes and their application to concurrent error detection of multilevel circuits. In *17TH IEEE VLSI Test Symposium*, page 370, 1999.

[EAS] http://www.eas.asu.edu/ holbert/eee460/see.html.

[EKD+03] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Toan Pham, Rajeev Rao, Conrad Ziesler, David Blaauw, Todd Austin, and Trevor Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Micro Conference*, December 2003.

[FEL01] Xiaobo Fan, Carla Ellis, and Alvin Lebeck. Memory controller policies for dram power management. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 129–134. ACM Press, 2001.

[FP92] Albert V. Ferris-Prabhu. On the assumptions contained in semiconductor yield model. *IEEE Trans. on Computer-Aided Design*, 11(8), 1992.

[Has99] K. Joe Hass. Probabilistic estimates of upset caused by single event transients. In *8th NASA Symposium on VLSI Design*, 1999.

[HL95] Ammar H.H. and Yu L. Fuzzy marking petri nets: concepts and definition. In *Proceedings of the 1995 IEEE international symposium on Intelligent Control*, pages 291–297, 1995.

[HPB02] T. Heath, E. Pinheiro, and R. Bianchini. Application-supported device management for energy and performance. In *Proceedings of Workshop on Power-Aware Computer Systems PACS'02*, February 2002.

[IBA02] *InfiniBand^{TM} Architecture Specification Release 1.1*, 2002.

[Jen97] Kurt Jensen. A brief introduction to coloured petri nets. In *Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 203–208. Springer-Verlag, 1997.

[Jen98] K. Jensen. An introduction to the practical use of coloured petri nets. *Lecture Notes in Computer Science: Lectures on Petri Nets II: Applications*, 1492, 1998.

[KCJ98] Lars M. Kristensen, Soren Christensen, and Kurt Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer: Special section on coloured Petri nets*, 2(2):98–132, 1998. available at http://sttt.cs.uni-dortmund.de/.

[KJBM98] Hass K.J., Gambles J.W., Walker B., and Zampaglione M. Mitigating single event upsets from combinatorial logic. In *7th NASA Symposium on VLSI Design*, 1998.

[MLSS92] Nicolaidis M., Jien-Chung Lo, Rao ST.R.N., and Thanawastien S. An sfs berger check prediction alu and its application to self-checking processor designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1992.

[MS96] P. C. Murley and G. R. Srinivasan. Soft-error monte carlo modeling program, semm. *IBM J. Res. Dev.*, 40(1):109–118, 1996.

[Ngu02] H. Nguyen. A systematic approach to ser estimation and solutions. In *Second Workshop on Evaluating and Architecting System dependabilitY (EASY)*, 2002.

[RV04] M. Sonza Reorda and M. Violante. Efficient analysis of single event transients. *J. Syst. Archit.*, 50(5):239–246, 2004.

[Sch03] Thomas Schlipf. Hca architecture education, 2003.

[Sho68] M. L. Shooman. *Probabilistic Reliability: An Engineering Approach*. McGraw-Hill Book Company, 1968.

[SKBA02] P. Shivakumar, M. Kistlerand S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *International Conference on Dependable Systems and Networks (DSN'02)*, 2002.

[SL98] V. R. L. Shen and F. P. Lai. Requirements specification and analysis of digital systems using fuzzy and marked petri nets. *IEEE Trans. on Systems, Man, and Cybernetics, part B*, 28(5):748–754, 1998.

[Sri96] G. R. Srinivasan. Modeling the cosmic-ray-induced soft-error rate in integrated circuits: an overview. *IBM J. Res. Dev.*, 40(1):77–89, 1996.

[TRA] http://pma.nlanr.net/traces/.

[WA01] Chris Weaver and Todd M. Austin. A fault tolerant approach to microprocessor design. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 411–420. IEEE Computer Society, 2001.

[WIK] http://en.wikipedia.org/.

[ZB96] W. M. Zuberek and I. Bluemke. Hierarchies of place/transition refinements in petri nets. In *Proc. 5th IEEE Int. Conf. on Emerging Technologies and Factory Automation, 18-21 November 1996, Kauai, Hawaii*, pages 355–360, 1996.

[ZSM99] Chaohuang Zeng, Nirmal Saxena, and Edward J. McCluskey. Finite state machine synthesis with concurrent error detection. In *International Test Conference 1999 (ITC'99)*, page 672, 1999.