A Fault-Aware Resource Manager for Multi-Processor System-on-Chip (MPSoC)

Bentolhoda Ghaeini
Master’s Thesis

A Fault-Aware Resource Manager for Multi-Processor System-on-Chip (MPSoC)

Bentolhoda Ghaeini

Reg Nr: LIU-ID/A-LITH-EX-A–13/055–SE
Linköping 2013

Supervisor: Petru Ion Eles
IDA, Linköpings universitet

Examiner: Petru Ion Eles
IDA, Linköpings universitet

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden
The semiconductor technology development empowers fabrication of extremely complex integrated circuits (ICs) that may contain billions of transistors. Such high integration density enables designing an entire system onto a single chip, commonly referred to as a System-on-Chip (SoC). In order to boost performance, it is increasingly common to design SoCs that contain a number of processors, so called multi-processor system-on-chips (MPSoCs).

While on one hand, recent semiconductor technologies enable fabrication of devices such as MPSoCs which provide high performance, on the other hand there is a drawback that these devices are becoming increasingly susceptible to faults. These faults may occur due to escapes from manufacturing test, aging effects or environmental impacts. When present in a system, faults may disrupt functionality and can cause incorrect system operation. Therefore, it is very important when designing systems to consider methods to tolerate potential faults. To cope with faults, there is a need of fault handling which implies automatic detection, identification and recovery from faults which may occur during the system’s operation.

This work is about the design and implementation of a fault handling methods for an MPSoC. A fault aware Resource Manager (RM) is designed and implemented to obtain correct system operation and maximize the system’s throughput in the presence of faults. The RM has the responsibility of scheduling jobs to available resources, collecting fault states from resources in the system and performing fault handling tasks, based on fault states. The RM is also employed in multiple experiments in order to study its behavior in different situations.
Abstract

The semiconductor technology development empowers fabrication of extremely complex integrated circuits (ICs) that may contain billions of transistors. Such high integration density enables designing an entire system onto a single chip, commonly referred to as a System-on-Chip (SoC). In order to boost performance, it is increasingly common to design SoCs that contain a number of processors, so called multi-processor system-on-chips (MPSoCs).

While on one hand, recent semiconductor technologies enable fabrication of devices such as MPSoCs which provide high performance, on the other hand there is a drawback that these devices are becoming increasingly susceptible to faults. These faults may occur due to escapes from manufacturing test, aging effects or environmental impacts. When present in a system, faults may disrupt functionality and can cause incorrect system operation. Therefore, it is very important when designing systems to consider methods to tolerate potential faults. To cope with faults, there is a need of fault handling which implies automatic detection, identification and recovery from faults which may occur during the system’s operation.

This work is about the design and implementation of a fault handling methods for an MPSoC. A fault aware Resource Manager (RM) is designed and implemented to obtain correct system operation and maximize the system’s throughput in the presence of faults. The RM has the responsibility of scheduling jobs to available resources, collecting fault states from resources in the system and performing fault handling tasks, based on fault states. The RM is also employed in multiple experiments in order to study its behavior in different situations.
Acknowledgments

First, I would like to thank my examiner, Professor Petru Ion Eles, for all the great help and encouragement he has given me and for providing me the possibility to complete this thesis.

I would like to thank PhD student Dimitar Nikolov for discussions and help. Thank you very much my friends in ESLAB and Ericcson; Shanai Ardi, Farrokh Ghani Zadegan, Afshin Hemmati, Leila Azari for prov reading the thesis and for the great advices.

I also need to thank my parents for all the support they have provided me over the years. Without them, I may never have gotten to where I am today.

Finally, I would like to thank my husband, Mohammad Reza for supporting me in so many ways and taking the time to listen, even when I was just complaining.
Contents

1 Introduction ........................................... 1
  1.1 Prior Work ........................................... 3
  1.2 Problem Definition ................................... 4
  1.3 Thesis Outline ....................................... 5

2 Preliminaries ........................................... 7
  2.1 Fault, Error and Failure ............................. 7
  2.2 Fault Classification, Fault Sources and Fault handling .. 9

3 Resource Manager .................................... 13
  3.1 System Model ......................................... 13
  3.2 Inputs and Outputs of the Resource Manager ........... 18
  3.3 Design of the Resource Manager ....................... 20
    3.3.1 Resource Manager Algorithm .................... 21

4 Experimental Results .................................. 29
  4.1 Experimental Setup .................................. 29
  4.2 Experimental Results ................................ 30
    4.2.1 Immediate Fault Handling, One Permanently Faulty Slave CPU .......................... 31
    4.2.2 Immediate Fault Handling, Two Permanently Faulty Slave CPUs ......................... 35
    4.2.3 Immediate Fault Handling, Three Permanently Faulty Slave CPUs ....................... 36
    4.2.4 Fault Handling After Job Completion, Two Permanently Faulty Slave CPUs ............. 38

5 Conclusion and Future Work .......................... 41
  5.1 Conclusion ........................................... 41
Chapter 1

Introduction

The development in semiconductor technologies gives opportunity to fabricate extremely complex integrated circuits (ICs) that may contain billions of transistors. The ICs enable designing an entire system onto a single chip. A System-on-Chip (SoC) is an IC that integrates all components of an entire system onto a single chip. A Multi-Processor System-on-Chip (MPSoC) is an SoC which contains multiple processors that are linked to each other by an on-chip interconnect [1]. Usage of MPSoCs is common to boost system performance.

Some MPSoCs may incorporate a number of elements such as CPUs, DSPs, accelerators, memories, etc. that are necessary for an application to run. Figure 1.1 shows a general view of an MPSoC structure.

![Figure 1.1. General view of an MPSoC structure](image)

P1, P2, ... might represent CPUs, DSPs, accelerators or other...
elements which can be considered in designing an MPSoC.

While recent semiconductor technologies enable fabrication of complex devices that provide high performance, these complex devices are more susceptible to faults because of shrinking feature sizes and lowering operating voltages. Furthermore, a fault may happen due to aging effects or environmental impacts [2]. Depending on their duration, faults are classified into transient or permanent faults. A transient fault is a fault which occurs in a component and causes the component to malfunction for a period of time, and disappears afterward. Such faults occur due to environmental impact like voltage stress, temperature stress, particles strike, etc. Unlike transient fault, a permanent fault remains in the component and does not correct with time. Permanent faults occur as a result of component failures, physical damage or design errors [3].

Assume a system which is performing a huge number of operations and is incorporated in a vital component in an airplane. In such a system a single fault may cause catastrophic consequences.

The existence of faults clarifies the need of incorporating fault handling methods to provide a system which is immune to faults [4]. Such fault handling methods should encompass the following set of tasks: 1) fault detection, i.e. identify the presence of a fault in the system, 2) fault identification, i.e. localize the defective (faulty) component, and 3) recovery actions, i.e. set of actions to restore the system to a safe state from which correct operation can be resumed. Fault handling methods are necessary to enable fault detection and provide recovery actions to mitigate the negative effect of faults. A key factor in fault handling is to correct faults as quickly as possible and keep the system in an operational state. It should be noted that fault detection and correction are best performed as close to the origin of fault as possible.

When a CPU detects a fault, the system assumes the fault is transient. The usual way of dealing with a transient fault is to keep using the affected CPU [5]. To distinguish between transient and permanent faults, a fault aware system is needed which can cope with both transient and permanent faults and take the right fault handling action when a fault occurs. When a system is fault aware, it has the option to apply a proper fault handling method to tolerate the fault. A system is fault tolerant when it has the ability to respond to an unexpected fault [4].
In order to provide a fault aware system, a centralized controller can be considered. The centralized controller can be used to control the functionality of the components in an MPSoC and synchronize the components’ operations. Furthermore, a centralized controller can provide a system with necessary information to be fault aware and help the system to distinguish between transient and permanent faults.

In this thesis, designing a fault aware centralized controller called "Resource Manager (RM)" in a given MPSoC is the main focus. The RM is not only a fault aware controller but it is also capable of scheduling jobs in an MPSoC. The job scheduling and fault handling should be done in an efficient way such that it results in an increase of the system throughput. In other words, the RM has two major tasks; 1) it schedules jobs on the resources in the MPSoC, 2) it constantly monitors, keeps track of the fault occurrences in an MPSoC and decides how to manage resources when faults occur.

## 1.1 Prior Work

This section reviews prior work related to this master thesis. The prior work is divided in two parts; 1) fault handling in MPSoCs and 2) fault handling with a Resource Manager.

Fault handling is a technique which automates detection, identification and recovery from faults, with the purpose to maintain correct system operation. A list of requirements for an efficient fault handling solution is discussed in [6].

### 1. Fault handling in MPSoCs

In case of fault handling in MPSoCs several studies aim to improve fault handling in an MPSoC by employing fault tolerance techniques [7], [8]. A good example of fault handling technique for MPSoCs is introduced in [7], in which a job is executed in-synch on two processors and fault detection is achieved by comparing the state of the two processors. If the states of the two processors differ from each other, it means that a fault has occurred during the job execution and a recovery process starts by re-executing the job. The MPSoC introduced in [7] is compared with some traditional fault tolerant systems with respect to performance and reliability. The comparison illustrates that
there is always a trade-off between performance and reliability. The higher performance is achieved at a cost of reduced reliability.

2. Fault handling with a Resource Manager

A Resource Manager handles both job scheduling and fault handling decisions. This suits many fault handling solutions, such as the approach in [7], which on top of managing faults keeps track of system "health" and schedules jobs accordingly.

In the Razor approach to enable power and performance scaling [9], an extra latch is put on each flip-flop to detect and recover from delay faults. The Razor flip-flops can be seen as fault detection components for which a Resource Manager keeps the delay fault count and makes the decisions to use other components in order to mitigate the impact of faults.

Another approach worth to mention here is presented in [10]. Some structures based on a count and threshold scheme is presented in [10] in order to distinguish intermittent and permanent faults against low rate, low persistence transient faults. A single threshold scheme and a double threshold scheme are presented in the research. The study comes out with the conclusion that the practical efficiency of all the mechanisms based on threshold depends on proper settings of the threshold value. A proper range of values for the threshold is necessary in order to have an efficient system operation.

1.2 Problem Definition

In an MPSoC which consists a number of CPU, We want to design and implement a resource manager (RM). For the RM, given is a queue of jobs which should be executed by the CPUs. Also, given are faults which may occur during the jobs execution.

We want the RM to be able to:

- Control and synchronize the CPUs in the MPSoC
- Efficiently take decisions for executing a set of jobs
- Distinguish between transient and permanent fault
- Perform fault handling
1.3 Thesis Outline

The thesis is organized as follows. Chapter 2, gives a brief definition of fault, error and failure as well as an introduction to fault classifications and redundancies. Chapter 3 presents more information about the RM design and its parameters. Chapter 4 describes the experiments and achieved results of the RM. The RM is experimented with different setups in this chapter and is compared with a system which does not use the RM. Finally, Chapter 5 summarizes the thesis and discusses about some suggestions for future work.
Chapter 2

Preliminaries

In this chapter, we discuss the key terms used in the fault tolerance area. First, we start by introducing faults, errors and failures, and then we elaborate on the difference between these terms. Next, we discuss on fault classification and some common fault tolerance techniques which are used to mitigate the negative effect of faults.

2.1 Fault, Error and Failure

In this section we discuss what faults, errors and failures are. When applied to digital systems, the terms fault, error and failure have distinct definition [3].

A fault is an anomalous physical condition. Causes include mistakes in system specification or implementation, manufacturing problems, and external disturbances. A fault can be a defect inside a component in a system or an abnormal condition which may lead to an unexpected system behavior [4]. To clarify, consider a two-input OR gate as shown in Figure 2.1. The nominal operation of the OR gate which is presented in Table 2.1.

A fault in the OR gate can occur during the manufacturing process, when one of the input or output lines is unintentionally connected to power supplies, Gnd or Vcc. If such situation occurs, the line is faulty since it is defective and can cause the OR gate to function abnormally. The faulty line may not have a direct effect on the output of the OR gate thus the fault may not be visible. Assume the output of the OR gate is accidentally connected to Vcc. So, the OR gate always
generates the value 1 as the output is then independent of the values of the inputs. We can say that the OR gate is faulty but not erroneous yet. The gate produces a correct output while at least one of the inputs is 1 (observe table 2.1). Thus, the fault may not have any effect on receiving a correct output of the OR gate.

In contrast to a fault, an error is a manifestation of a fault in a system, in which the logical state of an element differs from its intended value. A fault in a system does not necessarily result in an error. An error occurs only when a fault is sensitized [3]. As an example, consider the OR gate introduced in the previous example with an output unintentionally connected to $V_{cc}$. The faulty line causes an error when both inputs of the OR gate are set to 0 and the output should be a 0, rather than a 1.

An error can spread in case of using the output of an erroneous unit as an input for other units in a system. For instance, if the erroneous OR gate in our previous example is used in a circuit and its output is used as an input for other gates in the circuit, the error propagates by using the faulty OR gate [4].

A failure occurs when a system can not recover from errors. Fail-
2.2 Fault Classification, Fault Sources and Fault handling

In general, faults can be classified according to their duration, nature, and extent. According to duration, faults are classified into [11]:

- Transient faults
- Intermittent faults
- Permanent faults

Transient fault is a fault which occurs in the system for some period of time and disappears afterward. While the fault is present in the system, it causes the system to malfunction. When the transient fault disappears the system continues with its normal operation [12].

Intermittent fault is similar to a transient fault. Hence, it causes the system to malfunction for a period of time while the fault is present. The main difference between an intermittent fault and a transient fault is that the intermittent fault appears periodically. In other words, if a fault is intermittent, the fault appears for a time and disappears, but this scenario is repeated over time [12].

Permanent fault is a fault that remains in the system until the defective unit is replaced [12]. Even though, the permanent and the transient faults have distinctive meaning, in reality distinguishing them at run time is not so easy [4]. In [8] a method is presented in order to detect, isolate and recover from transient and permanent faults. This method enables system recovery by replacing the faulty elements in the system with spare elements.

There are various reasons why faults occur. A fault can occur due to a mistake in the specification or design of a system. A defective resource in the system can result in a fault. Other sources of faults can be due to different environmental conditions. For example, temperature, radiation and vibration can produce stress which can result in faults [4]. In recent semiconductor technologies, faults occur more
frequently due to shrinking feature sizes and lowering operating voltages.

To mitigate the negative effects of faults, systems have to be designed in a way to be able to handle faults. In order to handle faults, a fault-tolerant technique is needed to recover a system from a faulty state. Fault-tolerant techniques in general try to detect faults, and then perform a recovery action. Introducing redundancy is a common practice to achieve fault tolerance [13]. Redundancy is the property of having more resources than the minimum requirement for nominal system operation [14]. The main types of redundancy are:

- Hardware redundancy
- Time redundancy

**Hardware redundancy** is provided by incorporating extra hardware into the design to either detect or override the effects of a failed component. For instance, a system can have two or three processors which perform the same function instead of having a single processor. By having two processors, the failure of a single processor can be detected. By providing the system with three processors, it can be possible to override the wrong output of a single processor by using the majority output [4].

As an example for hardware redundancy, voters can be mentioned. There are several approaches for voting. Voting helps to extract correct output from a set of outputs. The simplest approach is in case of voting on outputs of N modules which are exactly the same. The voter will select the output which reflects the majority, as the correct one. Majority is determined as half of the number of modules. In this case an error can be detected and also corrected. If the number of outputs which are the same is fewer than N/2, then an error only can be detected and cannot be corrected [4]. A disadvantage with hardware redundancy is that providing extra hardware for a system increases the cost.

**Time redundancy** means the timing of the system is such that if certain tasks have to be rerun and recovery operations have to be performed, system requirements are still fulfilled [4]. By having time redundancy, a given computation can be repeated a number of times
and the results will be compared against each other to identify if a discrepancy exists [4].

Re-execution is one technique to handle faults by using time redundancy. In this technique, whenever a fault is detected in a processor, the job that has been running on that processor will be re-executed. It is important to note that re-execution as a fault handling technique is useful as long as it deals with transient faults and not permanent faults.

The reason is that if a fault is permanent, re-execution will be performed many times and each re-execution will result in a faulty output, so system overhead increases dramatically. Therefore, selecting re-execution as a fault handling method for a system could be also counted as a drawback specially if it is used when permanent faults are present. The RM designed in this thesis tries to mitigate the drawback of using re-execution as a fault handling technique when a permanent fault exists in the system. The RM assists the system to avoid performing many re-executions in case of the existence of permanent faults by introducing a threshold to limit the number of re-executions. The threshold helps to identify whether a resource in the system is affected by a transient or a permanent fault.
Chapter 3

Resource Manager

In this chapter, we present the details of the Resource Manager (RM) which we have designed in this thesis. Some information about important parameters of the RM, its inputs and outputs, and the different steps taken in the process of designing the RM are presented.

The purpose of running the RM in a system is to obtain correct system operation even in presence of faults, and to maximize the system throughput. The RM is responsible for scheduling jobs, collecting fault states and conducting fault handling for the system’s resources. One of the objectives of the RM design is to increase throughput.

A fault can happen at any time during system operation. The RM provides fault handling when a fault is detected in the system. For performing fault handling, the RM needs to monitor the processors’ operation and collect their states in order to identify whether a fault has occurred during the execution of a job.

3.1 System Model

In this section, we detail the architecture of the system which is addressed in this thesis. We present some information on how the RM observes the manifestation of faults in the system. The system which is assumed in this thesis is illustrated in Figure 3.1.

In a system which contains multiple CPUs, it is very common to have a controller which controls and synchronizes the operation of the other CPUs. Such a controller can be implemented as a piece of software that runs on one of the CPUs. The RM that we discuss in
this thesis has the role of such a controller and we assume that the RM is a piece of software that runs on one of the CPUs in the system, which we refer to as the Master CPU. The rest of the CPUs in the system we refer to as Slave CPUs (Figure 3.1) The RM synchronizes the operation of the Slave CPUs in the system.

In this thesis, we assume that the resources in the system are CPUs only. However, the same concept can be used also for systems that have other resources than CPUs. Each Slave CPU in the system has the ability of fault detection. Whenever a fault is detected in a Slave CPU, a one bit register which we refer to as 'Fault Indication Register (FIR)' (Figure 3.1), will be set to 1. So, the FIR reports the state of a corresponding Slave CPU and shows if it has detected a fault or not. The content of the FIRs can be considered as a fault indication code.

Similar to the Slave CPUs, the Master CPU is equipped with a register that keeps track of the fault states of the Slave CPUs. We refer to this register as 'Fault Checking Register (FCR)' (Figure 3.1). The size of the FCR is equal to the number of Slave CPUs that exist in the system. It means that the FCR has one bit per each Slave CPU. As it is depicted in Figure 3.1, each Slave CPU is connected to the Master CPU through a shared bus which is a dedicated bus
between FIRs and FCR. The RM observes whether a fault is detected in a Slave CPU or not by reading the contents of the FCR. The RM periodically reads the contents of the FCR which is updated according to changes in the FIRs.

As an example, consider a system similar to the one illustrated in Figure 3.1 which consists of four Slave CPUs; P1, P2, P3 and P4 and a Master CPU on which the RM is running. There is a one bit FIR in each of the Slave CPUs. Since, there are four Slave CPUs in the system, the FCR in the Master CPU is a four-bits register (one bit per each Slave CPU). We assume there are five jobs in a queue which should be assigned to the Slave CPUs by the RM in order to be executed, which are called J1, J2, J3, J4 and J5. The jobs are sorted in the queue according to their index number that is, J1 is the first job which should be fetched by the RM and be assigned to one of the Slave CPUs.

The RM starts by fetching jobs from the queue and assigning them to the available Slave CPUs. So, the RM assigns J1 to P1, J2 to P2, J3 to P3 and J4 to P4. All the Slave CPUs start to execute the jobs which are assigned to them by the RM. Whenever a Slave CPU has executed the job which has been assigned to it, it is ready to start the execution of another job. Assume P2 completes J2 while P1, P3 and P4 are still executing J1, J3 and J4 respectively. In this case, the RM fetches J5 from the queue of jobs and assigns it to P2.

In order to check if a fault is detected by any of the Slave CPUs, the RM periodically reads the contents of the FCR, where each of the bits in the FCR are updated according to changes in the corresponding FIRs. Here we provide an example to illustrate how the RM becomes aware that a fault has occurred in one of the Slave CPUs. Assume that the RM has assigned jobs to the Slave CPUs and no faults have occurred in any of the Slave CPUs in meanwhile. This means that none of the Slave CPUs have detected faults and therefore all the FIRs are set to 0. When the RM reads the contents of the FCR, at this time, it reads '0000' meaning that no fault has occurred in any of the Slave CPUs and therefore there is no need to perform any fault handling actions. However, if a fault occurs in P2, the fault will be detected by P2 and its FIR will be updated(set to 1). When the RM reads the FCR, it reads '0100' which means that a fault has occurred in P2 and thus it will initiate a fault handling action to handle the
fault in P2.

Testing the defective device or performing a fault-tolerance action is one of the suitable measures that can be employed on the Slave CPU which has detected a fault. Some tests can be applied to the defective Slave CPU and these tests can help to detect if a Slave CPU is permanently faulty. Initiating a functional test or a Built-In-Self-Test can be a suggestion for this purpose. The benefit of detecting that the Slave CPU is permanently faulty helps to decrease the time overhead because the system will not spend too much time on the faulty Slave CPU.

The RM has the ability of performing fault handling whenever a Slave CPU detects a fault. The fault handling method which is used in this thesis is re-execution. In this technique, whenever a fault is detected in a Slave CPU, the job that has been running on the Slave CPU will be re-executed. While the re-execution is a simple technique for fault handling, performing re-execution could be the reason to have a tremendous system overhead. Assume a system which consists of a number of Slave CPUs and one Master CPU.

The Slave CPUs are executing jobs which are assigned to them by the RM and the fault handling method which is used is re-execution. As an example, we consider a scenario where a permanent fault occurs in one of the Slave CPUs. As the Slave CPUs are equipped with fault detection mechanism, the fault will be detected by the Slave CPU and its FIR will be updated accordingly. The RM observes the fault detection when it reads the FCR and it performs fault handling (job re-execution) on the Slave CPU. The Slave CPU starts with the re-execution of the job it was executing before. However, the fault which has occurred is a permanent one, which means that the FIR of the Slave CPU will always have the value 1, pointing out that a fault is detected. Therefore, whenever the RM reads the FCR of the corresponding Slave CPU, it will try to handle the fault by forcing the Slave CPU to re-execute the job that it has been executing before. As this event would repeat over time that means that one of the Slave CPUs will be constantly busy by re-executing a job which has previously been assigned to it by the RM. This in turn causes a high time overhead for the system and it should be avoided.

For preventing this situation to happen, one solution could be removing the faulty Slave CPU affected by a permanent fault from the
system as soon as possible. Removing a Slave CPU from the system means that the RM does not assign jobs to that Slave CPU anymore. By removing this kind of Slave CPUs from the system, the time overhead caused due to unnecessary re-executions will be reduced. A drawback of this solution is the fact that the fault detected in the Slave CPU may not be a permanent fault but a transient fault.

Performing re-execution can help if the fault is transient but not if the fault is permanent. In this regard, a value called 'Re-execution Threshold (RT)' is introduced in this thesis. The RT can be used to distinguish if a fault is permanent or transient. The RM counts the number of faults detected by the Slave CPUs. If the number of faults detected by a Slave CPU exceeds the RT, the Slave CPU will be removed by the RM from the system. In other words, the RT can be used for identifying the maximum number of re-executions which a Slave CPU is allowed to perform.

Finding a proper value for the RT is important in order to have a more efficient RM. For instance, assume a system which contains a number of CPUs; one Master CPU and some Slave CPUs. The RM keeps track of the number of faults which are detected by the Slave CPUs. If the number of faults in a Slave CPU exceeds the RT, the RM decides that the detected fault is permanent and marks the Slave CPU as a faulty Slave CPU. The Slave CPU which is fault marked is no longer part of the system’s resources and the RM does not assign jobs to it anymore. A proper RT should be found for the system such that CPUs which are affected by permanent faults are removed as soon as possible, such that they do not impose a high time overhead in the system.

If the RT is set to be too high, that means the CPUs which are permanently faulty will be part of the system resources for a longer time and with this, these faulty CPUs will only introduce more overhead and thus might degrade the system performance.

On the other hand if the RT is too low, there is a rise that a CPU which has not been affected by a permanent fault is removed from the system too early. Removing a usable CPU from the system resource explicitly reduces the system performance.

If this scenario happens too often, the system will lose its resources after a while, despite the fact that some of the Slave CPUs which are removed are not permanently faulty.
Out of these discussions, we conclude that there is a need to carefully select an RT value such that the best system performance can be achieved.

### 3.2 Inputs and Outputs of the Resource Manager

The RM has some inputs which provide the necessary information for it to be able to start its operations. Figure 3.2 illustrates the general view of the inputs and the outputs of the RM.

![Figure 3.2. An overview of inputs and outputs of a RM](image)

**Inputs of the Resource Manager**

As illustrated in Figure 3.2, the inputs of the RM can be itemized as:

- Job List
- FIRs

**Job List** represents a queue of jobs. The term "Job" is used in this thesis for defining a task which should be executed in the system. Each job is provided with an execution time and a level of importance. The execution time denotes the time needed for the job to be executed. According to the level of importance, jobs are divided into two different groups:
3.2 Inputs and Outputs of the Resource Manager

- Critical Jobs
- Non-Critical Jobs

Critical jobs are those jobs whose execution is critical for the system. The more critical a job is, the higher importance it has. We prefer to execute and complete as many critical jobs as possible. The goal for the system is to reach a high number of executed critical jobs. In contrast, non-critical jobs are not so important and we do not insist on gaining high value of executed non-critical jobs. However, the higher the number of total executed jobs (critical and non-critical) the higher the system throughput is.

The jobs in the job list are not dependent and thus the order of job execution is irrelevant. The job in front will be fetched first. For each job we keep the following records: job Id, level of criticality and execution time. The job Id is a unique number for each job. A job is executed, as long as no faults are detected during its execution.

**FIR** is is one bit register in each of the Slave CPUs. The values of the FIRs is recorded into the FCR. The FCR is read by the RM at each time. Each bit of the FCR consists of a number of 0s and 1s. These values denote whether a fault has been detected in a corresponding Slave CPU.

**Outputs of the Resource Manager**

When a fault is detected in a Slave CPU, the RM makes the decision for the action which should be taken for handling the faults in the targeted Slave CPU. Based on the information provided by the inputs, the RM performs two important functions:

- Schedules and assigns jobs to the available Slave CPUs
- Supports fault handling in case of fault occurrence in the Slave CPUs

These two functions can be considered as the output of the RM. To measure the quality of the RM, we calculate the system throughput. The system throughput represents the number of jobs per time unit.
The system throughput is a good metric to evaluate the RM as both number of jobs and total time are directly dependent on how job scheduling and fault handling have been performed.

### 3.3 Design of the Resource Manager

In this section, we will demonstrate how the RM is designed. Before explaining the details of the RM, it is necessary to give some information about the two important blocks:

- **Device State Map**
- **System Health Map**

**Device State Map (DSM)** keeps track of the current state of each Slave CPU in the system. Each Slave CPU can have three different states which are: idle, active, and faulty. The state of all the Slave CPUs is kept in the DSM which is illustrated in Table 3.1. This table will be modified by the RM whenever the state of a Slave CPU changes.

<table>
<thead>
<tr>
<th>Slave CPUs ID</th>
<th>State of Slave CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave CPU 1</td>
<td>idle</td>
</tr>
<tr>
<td>Slave CPU 2</td>
<td>active</td>
</tr>
<tr>
<td>Slave CPU 3</td>
<td>idle</td>
</tr>
<tr>
<td>Slave CPU 4</td>
<td>faulty</td>
</tr>
<tr>
<td>Slave CPU 5</td>
<td>active</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Slave CPU n</td>
<td>faulty</td>
</tr>
</tbody>
</table>

A Slave CPU is "idle" when either no job has been yet assigned to it or it has finished execution of a previously assigned job without any fault detection. An idle Slave CPU is ready to start with execution of a new job. A Slave CPU is 'active' when a job is assigned to it. A Slave CPU is faulty when it detects more faults than it is allowed by the system. There is one row for each Slave CPU in the DSM.
System Health Map (SHM) is used to record the number of fault occurrences for each Slave CPU in the system. The SHM identifies how many faults are observed by the RM in a given Slave CPU. The number of detected fault in SHM for a Slave CPU will be reset when:

- The detected fault is a transient fault
- The Slave CPU is assigned to execute a new job

The SHM table is illustrated in Table 3.2.

<table>
<thead>
<tr>
<th>Slave CPUs ID</th>
<th>Number of faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave CPU 1</td>
<td>1</td>
</tr>
<tr>
<td>Slave CPU 2</td>
<td>3</td>
</tr>
<tr>
<td>Slave CPU 3</td>
<td>0</td>
</tr>
<tr>
<td>Slave CPU 4</td>
<td>4</td>
</tr>
<tr>
<td>Slave CPU 5</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Slave CPU n</td>
<td>n</td>
</tr>
</tbody>
</table>

The first column in Table 3.2 shows the IDs for each Slave CPU in the system. The second column captures the number of faults which are detected in the corresponding Slave CPU. There is one row for each Slave CPU in the SHM. For instance, the SHM for a system which consists of four Slave CPUs includes four rows and two columns. The RM updates this table whenever an active Slave CPU detects a fault.

### 3.3.1 Resource Manager Algorithm

In this section we present the algorithm which describes the different steps that are carried out by the RM. The algorithm is presented in Figure 3.3.

As discussed earlier, one of the main tasks of the RM is assigning jobs to the Slave CPUs. In this regard, the RM starts assigning jobs to the Slave CPUs by identifying an available Slave CPU (Figure 3.3).

The RM searches through the DSM to ensure that there is at least one non-faulty Slave CPU in the system. If the RM cannot find any
Figure 3.3. The RM algorithm
non-faulty Slave CPU, the system experiences a failure because all the Slave CPUs are fault marked. All the Slave CPUs can be fault marked if all of them have detected permanent faults.

After finding a non-faulty Slave CPU, the RM searches for the jobs in the job list file. If there are no jobs to assign to the Slave CPUs and none of the Slave CPUs are active, the RM terminates. Otherwise, a job is fetched by the RM from the job list and is assigned to the first "idle" Slave CPU. It should be noted that all the Slave CPUs initially start up in the idle state. The state of an "idle" Slave CPU changes to "active" once a job is assigned to that Slave CPU by the RM. The RM updates the DSM after assigning a job to a Slave CPU. Figure 3.4 illustrates the transitions between different states for a given Slave CPU. While executing a job the Slave CPU remains in the "active" state.

![Figure 3.4. Transition between different states for a Slave CPUs](image)

In the next phase, the RM starts performing fault observation. When a fault is detected in a Slave CPU, the FIR in the Slave CPU will be set. The value of each FIR is transferred to the Master CPU and is saved into the corresponding bit of the FCR. In this step, which is called polling, the RM periodically checks the value of the FCR in order to find an active Slave CPUs which has detected a fault.
According to Figure 3.3 two scenarios might happen after polling; 1) none of the Slave CPUs has detected a fault or 2) at least one of the Slave CPUs has detected a fault during the job execution.

In the first scenario, i.e. none of the Slave CPUs has detected a fault, the RM is done with polling and has identified that none of the active Slave CPUs in the system has detected a fault.

If a job has completed, the RM modifies the DSM. As illustrated in Figure 3.4, the state of the "active" Slave CPU changes to the "idle" when the Slave CPU completes the execution of the job that was assigned to it. The idle Slave CPU is now ready to execute a new job from the job list. If no jobs are completed the RM continues with polling.

In the second scenario, i.e. at least one of the "active" Slave CPUs has detected a fault during the job execution, the RM performs fault handling for the Slave CPU which has detected a fault. While being in the "active" state, the Slave CPU needs some time to execute the job and a fault might occur during the job execution. In this case, the RM identifies the Slave CPU which has detected a fault after finishing the polling step. The RM analyzes the content of the FCR. If an active Slave CPU has detected a fault, the RM performs fault handling for that particular Slave CPU.

Attempting to detect faults before assessment of the job completion helps to reduce the time overhead. In this case, the RM can perform fault handling earlier instead of waiting for the jobs to complete first and then check if faults have occurred. To clarify more, lets assume a system in which the RM checks for faults after job completion.

Consider the following scenario. A job is assigned to a Slave CPU and a fault occurs in the Slave CPU after it has just started executing the job. If the RM checks whether faults have occurred only after the job completes, it means that a Slave CPU has been allowed to proceed executing a job which would produce a faulty outcome and thus the Slave CPU has wasted time on executing unnecessary operations. The RM which is designed in this thesis checks for fault detection periodically and does not wait for the job to be completed. So, if the Slave CPU detects a fault immediately after starting the job, the RM interrupts the job execution and applies fault handling for the Slave CPU. It helps to save time and reduce the time overhead of fault handling.

Figure 3.5 shows how performing fault handling immediately after
a fault detection reduces the time overhead. It is assumed that the job execution in both cases starts at the same time; a0 is equal to b0. In Figure 3.5(A), a Slave CPU starts a job execution at time a0. The Slave CPU detects a fault at time a1 but for this example we assume that the fault handling should be performed upon job completion (time a2). So, the job is re-executed at time a2. Finally, the job completes at time a3. In contrast to this scenario, Figure 3.5(B) illustrates a scenario where fault handling is done immediately after fault detection. A fault is detected at time b1 and the Slave CPU re-executes the job immediately after fault detection. Finally the job completes at time b2. As it is depicted in Figure 3.5, b2 is less than a3, which justifies to use a RM that performs fault handling immediately after fault detection as this reduces the time to execute a job. (a2 - b1) is the wasted time in (A) spent due to postponing the fault handling.

Figure 3.5. (A) fault handling after reaching the end of job. (B) fault handling before reaching the end of job

Next we discuss how fault handling is performed. Figure 3.6 illustrates how the RM which we have designed in this thesis performs fault handling. The first step in order to perform the fault handling is identifying if the number of detected faults for the Slave CPU exceeds the RT. As it is discussed earlier, the RT can be used for identifying the maximum number of re-executions which a Slave CPU is allowed
to perform. When a fault is detected in a Slave CPU, the RM updates the SHM and the number of detected faults for the corresponding Slave CPU is incremented by one. The RM keeps the number of detected faults for each Slave CPU in the SHM in order to be able to compare that number with the RT.

Figure 3.6. Fault handling algorithm

Figure 3.6 illustrates that there are two possibilities after comparing the number of detected faults with the RT; 1) The number of detected fault in the Slave CPU is less than the RT or 2) The number of detected fault in the Slave CPU exceeds the RT.

1. If the number of the detected faults does not exceed the RT the RM takes different decision based on whether a) the job is critical or
b) the job is non-critical.

a) If the job is critical it means that the job is important to be executed. The critical job will be re-executed on the same Slave CPU on which it was running before. The reason for executing the job on the same Slave CPU is because of the difficulty in distinguishing between transient and permanent faults. When a fault is detected in a Slave CPU, it is not clear if the fault is transient or permanent. In such a situation, the first assumption is that the fault is transient and it will not occur again when re-executing the same job on the same Slave CPU. This assumption is because of the preference in keeping the Slave CPU in the system until making sure that the fault is a permanent one.

b) If the job is non-critical so it means that the job has low priority to be executed. In this case, the job simply will be dropped by the RM and the Slave CPU is marked idle. The RM updates the DSM after changing the state of the Slave CPU to "idle".

2. There is another scenario that can happen; the number of the total faults has exceeded the RT. In this case, as illustrated in Figure 3.4, the RM changes the state of the Slave CPU to 'faulty' and the Slave CPU is no longer considered as a part of the system resources. If the faulty Slave CPU was executing a critical job, the job should be re-executed again but on another Slave CPU. So the RM keeps the job and returns it to the queue of the jobs. The job will be assigned by the RM to the first available (idle) Slave CPU in the system. The situation could be handled simply if the job is a non-critical job. In this case, the RM drops the job and that job will not be assigned to another Slave CPU for re-execution.

As it is illustrated in Figure 3.6, these two scenario were the two possible outcomes when performing fault handling. When fault handling is over, the RM proceeds with the following steps. It modifies the state of the Slave CPU which has detected a fault. If the state is "faulty", the Slave CPU is known as a permanently faulty Slave CPU and no job will be assigned to it by the RM anymore. Otherwise, if the state is "idle", the Slave CPU is ready to execute a new job from the job queue.
Chapter 4

Experimental Results

In this chapter we present some experimental results obtained by running the RM simulator in an MPSoC for different scenarios. The MPSoC consists of four Slave CPUs and one Master CPU on which the RM is running.

4.1 Experimental Setup

The RM which is simulated in this chapter is running on a Master CPU in an MPSoC. The MPSoC consists of one Master CPU and four Slave CPUs called P1, P2, P3 and P4.

Each Slave CPU is assigned for a job execution by the RM. The queue of jobs is simulated as a file called job list file. Each of the jobs in the job list file is provided with ID, level of criticality (critical or non-critical) and execution time. There is no priority assumed for the jobs in the job list file. The jobs in the job list file are independent so the order of job execution is irrelevant.

A job is critical or non-critical according to how much important it is for the system to be executed. The RM is more sensitive to critical jobs. If a Slave CPU detects a fault during execution of a critical job, the RM performs fault handling and re-execution is applied for the job. In case of a fault occurrence during execution of a non-critical job, the RM drops the job and the job is removed from the job list file. For the experiments in this chapter, we have only considered critical jobs. The job list file in all the experiments consists of 50 critical jobs. The execution time for each of the jobs varies in the range [50,100]
To simulate the values of FIRs and FCR, we use another file called fault list. Each line of the fault list file consists of a number of 0s or 1s which are selected randomly with the probability of 1%. The value 1 indicates that the corresponding Slave CPU has detected a fault during the job execution and the value 0 shows that no fault is detected at this time. Each value is corresponding to the individual FIRs. Since there are four Slave CPU in the MPSoC, each line of the fault list file consists of four values. As an example, consider a line in the fault list file which is "1 0 0 0". This line shows that P1 has detected a fault and no faults are detected by P2, P3 and P4 at this time.

The RM simulator retrieves one line from the fault list file to simulate the values saved into the FCR in the polling step. We assume the polling is the only step which consumes time, so time progresses by one time unit only when the RM simulator enters the polling step. It means that after the polling step one additional time unit has elapsed since the jobs were assigned to the Slave CPUs. Hence if a number of time units equal to the given execution time of a job have elapsed and no fault is detected, we say that the job has been successfully completed.

4.2 Experimental Results

In this section we present experimental results for four different scenarios which are listed below.

In the first three scenarios, the RM performs fault handling immediately after fault detection. In the first experiment one of the Slave CPUs is affected by a permanent fault. In the second experiment, two Slave CPUs are affected by a permanent fault and in the third experiment three Slave CPUs are affected by a permanent fault. By studying these three scenarios, we want to learn how much the total execution time for a number of critical jobs is affected when the system loses its resources.

In the last scenario, the RM performs fault handling after the job completion. By this experiment, we want to study what is the difference in total execution time for the same number of critical jobs when
applying the these two RMs in an MPSoc; 1) the RM which performs fault handling immediately after fault detection and 2) the RM which performs fault handling after the job completes. Furthermore, we will see how much the RM is successful to distinguish between transient and permanent faults.

For all the experiments, the RT varies in the range $[10,5000]$, fault generation rate of 1% is considered and the job list file includes 50 critical jobs.

The three scenarios in which the RM performs fault handling instantly are as follow:

### 4.2.1 Immediate Fault Handling, One Permanently Faulty Slave CPU

In the first experiment we assume that there is a single permanent fault in one of the Slave CPUs. Figure 4.1, illustrates the total time consumed by the RM for different RT values in case of executing all 50 critical jobs in the job list file.

![Figure 4.1. Immediate fault handling, one permanently faulty Slave CPU, RT ∈ [10, 5000]](image-url)
Figure 4.1 shows that the execution time changes in the range of (1500,2500) when the RT changes between 10 and 1900. The total time increases linearly with RT when the RT is higher than 1900. So, by selecting a RT value in range of [10,1900] all critical jobs in the job list file are executed and total execution time is less than selecting a RT value higher than 1900.

We expect to see a direct relation between the total execution time and the RT, i.e. the total execution time for a higher RT should be more than total execution time for a lower RT. For example, assume two situations; first, the RT is set to 100 and second, the RT is set to 200. When the RT is 100, it means the RM lets each Slave CPU to perform up to 100 re-executions. After that the Slave CPU is identified as a faulty one and is removed from the system resources. In contrast, each Slave CPU is allowed to perform up to 200 re-executions when the RT is set to 200.

When the RT is higher, it means that a faulty resource will be kept in the system for a longer time. Thus, due to the fact that a faulty resource is present and this resource will not contribute in improving the system throughout, having the faulty resource will just increase the total execution time. For the example we presented, when the RT is set to 200, it has 200 t.u. before the faulty CPU is removed, while for the case when the RT is set to 100, it has 100 t.u. before the faulty CPU is removed. Thus having a faulty resource present in the system will unnecessarily slow down the system as more time will be wasted on unsuccessful re-executions scheduled on the faulty CPU, i.e. the total execution time will be increased.

For a Slave CPU which is affected by a permanent fault, the total time needed for performing 200 re-executions will be larger than the total time for performing 100 re-executions. While the previous discussion justifies the claim that the total execution time increases with the RT, the experimental results presented in 4.1 shows that this claim is not always correct. For example in Figure 4.1, the total execution time for an RT value set to 3210 is larger than the total execution time for an RT value set to 3530. The reason for existence of these kinds of glitches could be because of differences in the pattern of faults in the fault list file for each Slave CPU. A lower RT may be the reason to remove a Slave CPU from the system earlier compared to when a higher RT is used
4.2 Experimental Results

in the system.

When the RT is low, a resource might be fault marked due to that a number of transient faults have occurred in the resource. Observe that in such scenario the RM would remove an available resource from the system which can later be the reason to increase the total execution time.

An example is provided in order to explain the impact of the RT on the total execution time. In this example, it is assumed that the system was including four Slave CPUs. One of the Slave CPUs is fault marked as it was affected by permanent fault. So, the system has three Slave CPUs which can start job execution. All the three Slave CPUs which are idle at the starting point at time zero. Table 4.1 shows the fault pattern for the time unit zero and time interval [0,6] t.u.

Table 4.1. Fault pattern for each of the Slave CPUs

<table>
<thead>
<tr>
<th>time</th>
<th>FIR Slave CPU1</th>
<th>FIR Slave CPU2</th>
<th>FIR Slave CPU3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 t.u</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 t.u</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 t.u</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3 t.u</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4 t.u</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5 t.u</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6 t.u</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1 shows that Slave CPU1 detects a fault during the first and the second time unit. The Slave CPU2 detects a fault during the third time unit and Slave CPU3 is fault-free during the entire interval. Furthermore, we assume there are 4 critical jobs in the job list file and each of them is associated with an execution time as it is illustrated in Table 4.2.

Table 4.3 presents how the jobs are assigned to the Slave CPUs by the RM, while having the RT set to 4. Since all the Slave CPUs are idle at starting point i.e. at time zero, job1 is assigned to Slave CPU1, job2 is assigned to CPU2 and job3 is assigned to Slave CPU3 by the RM. Slave CPU1 detects a fault during the first time unit. Since the job is a critical job and the number of detected faults by the Slave CPU does not exceed the RT, job1 will be re-executed on Slave
Experimental Results

Table 4.2. Jobs ID and their execution time

<table>
<thead>
<tr>
<th>Job ID</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

CPU1. Slave CPU2 and Slave CPU3 have not detected any faults at this time so they will continue to execute the jobs which are assigned to them. During the second time unit, Slave CPU1 detects another fault. The number of detected faults in Slave CPU1 is equal to 2 and is still less than the RT which is 4. Therefore, Slave CPU1 re-executes job1 again. In meanwhile, Slave CPU2 and Slave CPU3 continue with the execution of job2 and job3 respectively. When two time units have elapsed, since job2 was assigned to CPU2 and no errors have occurred in CPU2, job2 is completed successfully and Slave CPU2 is available to execute another job. So job4 will be assigned to Slave CPU2 by the RM. During the third time unit, a fault is detected by Slave CPU2 and re-execution will be performed for fault handling. Slave CPU2 and Slave CPU3 complete the job execution at time unit 4 and will be idle while Slave CPU1 is still active executing job1. Finally, job1 completes at time unit 5 and Slave CPU1 will be idle afterward. As can be seen from Table 4.3, the total execution time for executing job1, job2, job3 and job4 is 5 time units for the given fault pattern (Table 4.1).

Table 4.3. execution of jobs by the Slave CPUs, RT = 4 for the fault pattern presented in Table 4.1

<table>
<thead>
<tr>
<th>Time Unit</th>
<th>Slave CPU1</th>
<th>Slave CPU2</th>
<th>Slave CPU3</th>
<th>FCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 t.u</td>
<td>No job</td>
<td>No job</td>
<td>No job</td>
<td>0 0 0</td>
</tr>
<tr>
<td>1 t.u</td>
<td>job1</td>
<td>job2</td>
<td>job3</td>
<td>1 0 0</td>
</tr>
<tr>
<td>2 t.u</td>
<td>job1</td>
<td>job2</td>
<td>job3</td>
<td>1 0 0</td>
</tr>
<tr>
<td>3 t.u</td>
<td>job1</td>
<td>job4</td>
<td>job3</td>
<td>0 1 0</td>
</tr>
<tr>
<td>4 t.u</td>
<td>job1</td>
<td>job4</td>
<td>job3</td>
<td>0 0 0</td>
</tr>
<tr>
<td>5 t.u</td>
<td>job1</td>
<td>No job</td>
<td>No job</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>

The previous example is repeated in Table 4.4 with the RT equals
to 2. In this case, Slave CPU1 will be fault marked by the RM faulty at the end of the second time unit. So, job1 will be transferred to Slave CPU2 at time unit 3 in order to be executed. Slave CPU2 detects a fault at time unit 3 while executing job1 and will therefore need to perform re-execution. Slave CPU3 completes job3 and job4 will be assigned to the Slave CPU by the RM. Table 4.4 shows the total time for executing all the jobs by the Slave CPUs. As can be seen from Table 4.4, the total time to execute all the critical jobs from the job list file under a fault pattern as described in Table 4.1 and the RT = 2, is 6 time units.

Table 4.4. execution of jobs by the Slave CPUs, RT = 2 for the fault pattern presented in Table 4.1

<table>
<thead>
<tr>
<th>Time Unit</th>
<th>Slave CPU1</th>
<th>Slave CPU2</th>
<th>Slave CPU3</th>
<th>FCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 t.u</td>
<td>No job</td>
<td>No job</td>
<td>No job</td>
<td>000</td>
</tr>
<tr>
<td>1 t.u</td>
<td>job1</td>
<td>job2</td>
<td>job3</td>
<td>100</td>
</tr>
<tr>
<td>2 t.u</td>
<td>job1</td>
<td>job2</td>
<td>job3</td>
<td>100</td>
</tr>
<tr>
<td>3 t.u</td>
<td>faulty</td>
<td>job1</td>
<td>job3</td>
<td>010</td>
</tr>
<tr>
<td>4 t.u</td>
<td>faulty</td>
<td>job1</td>
<td>job3</td>
<td>000</td>
</tr>
<tr>
<td>5 t.u</td>
<td>faulty</td>
<td>job1</td>
<td>job4</td>
<td>000</td>
</tr>
<tr>
<td>6 t.u</td>
<td>faulty</td>
<td>job1</td>
<td>No job</td>
<td>000</td>
</tr>
</tbody>
</table>

With the example we have justified that the fault pattern can have an influence on the total execution time.

### 4.2.2 Immediate Fault Handling, Two Permanently Faulty Slave CPUs

For the second experiment, we assume that two of the Slave CPUs are affected by permanent faults. Figure 4.2 illustrates the total execution time for different RT values. As in the previous scenario the job list file consists of 50 critical jobs.

From Figure 4.2, it can be observed that the total execution time for the RM changes between 2500 and 3500 time units when the RT varies in the range between 10 and 2890. For an RT value higher than 2890, total execution time increases with a linear relation with the RT. While in general the trend is that the total execution time is lower for the lower RT values, from figure 4.2 one can notice that...
Figure 4.2. Immediate fault handling, two permanently faulty Slave CPUs, $RT \in (10, 5000)$

there are some fluctuations in the graph. The main reason for having these fluctuations is explained previously with the example elaborated in Table 4.3 and Table 4.4.

4.2.3 Immediate Fault Handling, Three Permanently Faulty Slave CPUs

For the third experiment, we assume three Slave CPUs in the system are affected by a permanent fault. Figure 4.3 illustrates the total execution time for different RT values while all 50 critical jobs in the job list file are executed.

Figure 4.3 illustrates the total execution time versus RT for different RT values in the range of $[10,5000]$. A wider range of RT is considered for this experiment, shown in Figure 4.4. The total execution time for all critical jobs in the job list file for a RT value in the range of $[10,5000]$ is less than the total execution time for the same jobs for a RT value higher than 5000.

Figure 4.4 illustrates that for RT values more that 5000, the total execution time has a linear trend and it grows fast. The three experiments above have been studied in the case of employing the RM
4.2 Experimental Results

Figure 4.3. Immediate fault handling, three permanently faulty Slave CPUs, $RT \in [10, 5000]$

Figure 4.4. Immediate fault handling, three permanently faulty Slave CPUs, $RT \in [5000, 15000]$
which performs fault handling when a Slave CPU detects instantly a fault and does not wait until the end of job execution.

### 4.2.4 Fault Handling After Job Completion, Two Permanently Faulty Slave CPUs

In a second set of experiments, the system’s behavior is evaluated in case of executing the fifty critical jobs while the RM checks for fault detection and performs fault handling upon job completion. It is assumed that two Slave CPUs are affected by a permanent fault. The result of this experiment is shown in Figure 4.5. The thickness of curve in Figure 4.5 shows that there is a lot of fluctuations in the total execution time but the whole trend is consistent.

![Figure 4.5.](image)

**Figure 4.5.** Fault handling after job completion, two permanently faulty Slave CPUs, $\text{RT} \in [10, 5000]$

As can be seen in Figure 4.5, the total execution time which is spent to execute all fifty critical jobs in the job list file is increasing dramatically compared to the total execution time while the RM detects faults and performs fault handling instantly, i.e. does not wait until job is completed as illustrated in Figure 4.2. For example, the total
execution time when the RT value is set to 5000, is slightly more than 5000 t.u. when the system is employing the RM which performs fault
detection and handling instantly. However, if the RM perform fault
detection and fault handling upon job completion, the total execution
time is more than 450000 t.u.

The RM designed in this thesis is needed for a system in order to
perform fault handling at system level. By using this RM it is possible
to handle faults immediately upon detection. The RM also helps to
cope with transient and permanent faults by using the RT. The RT
helps to decide if a detected fault should be considered a transient
or permanent fault. Using the RM helps to reduce time overhead
since the system will not spend too much time for a faulty Slave CPU.
According to the Figure 4.1, Figure 4.2 and Figure 4.3, a range of RT
can be selected such that all critical jobs in the job list file are executed
without a huge impact on the total execution time. Observation from
the above (and similar) experiments, can help MPSoC designers in
choosing a suitable fault handling strategy.
Chapter 5

Conclusion and Future Work

5.1 Conclusion

Development in the semiconductor technologies makes it possible to increase the complexity of multi-processor System-on-Chips (MPSoCs). These complex systems are more susceptible to faults. In this regard, fault tolerance becomes a big challenge because of the rapid development in the semiconductor technologies and the growing rate of the fault occurrences in complex systems. Fault tolerance provides correct operation of a system even in presence of faults.

Having a fault tolerant system and performing fault handling introduces overhead. In this thesis, we have designed and implemented a fault aware Resource Manager (RM) for an MPSoC. The purpose of the RM is to obtain correct system operation, maximize the system’s throughput and reduce the time overhead.

The RM schedules jobs to the Slave CPUs in an MPSoC, keeps track of fault occurrences and performs fault handling to mitigate the negative effects of faults. The RM is designed such that it can cope with transient and permanent faults. A range of RT can be selected such that all critical jobs in the queue are executed without a huge impact on total execution time.
5.2 Future Work

In this section we provide some directions on how this research can be improved. In particular we have the following two ideas:

1. Using fault recovery with checkpointing.
2. Using a dynamic RT instead of a static RT which is used in this thesis.

To improve the performance of the RM an improved fault handling techniques could be considered; One of the techniques could be Rollback Recovery with Checkpointing. In this technique, each job is divided into a number of execution segments (ESs) and a checkpoint is placed between each ES. If at the end of an ES a fault is detected, the ES will be re-executed. Otherwise, a check point is saved in a memory and the processor continues execution with the following ES [1]. The benefit is that only a single ES is re-executed instead of re-executing the entire job.

For the RM presented in this thesis we have used a static RT value which does not change during system operation. One improvement might be applying a dynamic RT instead of the static RT in the RM. If a dynamic RT is defined, the RM is able to decrease or increase the threshold by a specific pattern when it is needed. A RM with a dynamic RT might be able to detect a Slave CPU which is defected by a permanent fault earlier than a RM with a static RT. So, the system won’t waste too much time by assigning jobs to a permanently faulty Slave CPU. Hence, by using a dynamic RT, time needed for detecting a faulty Slave CPU will be shorter.
Bibliography


