Contents

List of Figures

Chapter 1

Introduction

Control systems for high and medium power converter systems are being developed more frequently using digital architectures instead of analog architectures. Such systems are now built with microcontrollers or microprocessors because of their effects in reducing cost. These microprocessors form a processing unit that executes the control function for such a power electronics system. These computations are also time bound, hence the processing unit has to perform in real-time. To support such digital architectures, this thesis proposes a new fault-tolerant, real-time approach to scheduling control systems for medium and high power electronics applications. The centralized computational unit for control can be replaced by a distributed embedded processing system that can mask the failure of its individual hardware components. The core of this work is the implementation of an offline real-time scheduling scheme for such a multiprocessor system. The system is made in accordance with widely-accepted principles in the industry. This work has improved the dependability of the processing units for power electronics control systems.

1.1 Control Systems in Power Electronics

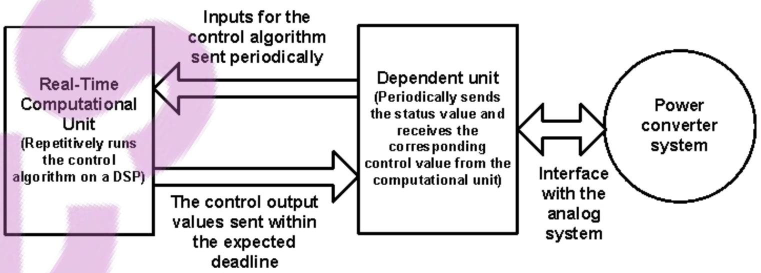

Inputs for the control algorithm sent periodically
Real-Time Computational Unit
(Repetitively runs the control algorithm on a DSP)
Dependent unit
(Periodically sends the status value and receives the corresponding control value from the computational unit)
Power converter system
Interface with the analog system
The control output values sent within the expected deadline

Figure 1.1: Modualarization of the components in a power electronics control system

The researchers at the Center for Power Electronics Systems (CPES) lab in the Bradley Department of Electrical and Computer Engineering at Virginia Tech have been working for the past several years on control systems for complex power converters. Earlier, there used to be no standardized architecture for high and medium power electronics control systems. Every new control system project would result in a different customized analog control circuit. Even though these circuits were built for similar purposes, each project required its own design and development cost because there was no standardized architecture for such systems to follow. Hence, development of such hardware systems for control proved to be costly. These systems also faced problems due to hardware failure of sub-components. The researchers at the CPES started working on the idea of digital control system for power electronics systems. They used DSP (Digital Signal Processors) for executing control algorithm for the power converters. In order to standardize the architecture they first modularized the complete system into, what we call as, the computational module and the dependent module. Figure 1.1 shows the two modules in the system. The whole system now contains 3 blocks:

- The computational unit
- The dependent unit
- The power converter system

Out of these blocks, the computational unit and the dependent unit are digital systems and the power converter system is an analog hardware system that has to be controlled. The dependent unit acts as the interface to this analog system. It samples the input analog values from the power converter system periodically, digitizes the data and sends it as input to the computational unit. The computational unit accepts these values as inputs for the control algorithm it executes. The control software produces the output within a pre-determined deadline. This output is then again sent to the dependent unit. The dependent unit converts this digital control value to analog values and feed them to the power converter system as inputs. The whole process is repeated continuously as long as the control system is in use. The control algorithm executed by the computational unit is a software emulation of the hardware control circuit that was used for control purposes. Now, for designing different control systems, the designers have to change the control software that gets executed on the computational unit instead of developing a new hardware control circuit. The 3 blocks and the inter-communication protocol between them forms the basis to develop a standardized architecture of a control system.

Milosavljevic [38] proposed an architecture for a modularized control system. It has a master board connected to other slave boards using a ring network. The objects to be controlled are separated and interfaced with the slave nodes. These slave nodes convert the analog signal into digital data and send them as input to the master board. The slave boards expect output from the master nodes within a certain deadline once an input is sent. This output value is used to control the system that is interfaced with the slave board. Thus we can see how the slave node fits into the role of dependent unit. The master board is responsible for executing the control transfer function. Thus the control circuit that used to be custom-built is modularized on the master board to form a computational unit. To reduce the dependency on the hardware control circuits as well as to have a standardized hardware architecture for the whole system, the master board is contains a Digital Signal Processor (DSP) that executes a micro-kernel. The control function is implemented using software components that run on the micro-kernel. These software components are reusable modules that on executing collectively simulates the operations of a control circuit. This means that now to develop a control system, the hardware architecture of the system need not be changed. The designer has to just reuse the software components and execute them for simulating the control algorithms that were normally effected using hardware circuits.

Francis, Guo and Singh [1, 2, 3, 5] completed the implementation of this system. Francis and Guo developed the hardware blocks and implemented the network protocol for this system. The ring network protocol, called Dual Ring PESNET (DRPESNET), is a daisy-chained, packet-switched, time-triggered network protocol. Singh implemented a real-time micro-kernel to support arbitrary control algorithms on this hardware arrangement. Thus, we have a processing unit that runs real-time control software for controlling other dependant nodes on the network. This processing unit regulates the working of the complete network and the dependant components. The control software executed on the micro-kernel is derived from the most commonly used control algorithms in power electronics. These control softwares have a dataflow architecture. Each of the reusable software component simulates the functioning of a part of the hardware control circuit. In this way, a standard architecture was developed that could be followed by the designers for their power converter control system projects. But, even now the system has not tackled the problems that the designers face because of hardware failure. Failure of the computational unit makes the whole system of no use. There was a need for an extended system that could tolerate the failure of the individual hardware components. It was this problem faced by the designers that motivated this thesis work.

1.2 Problem Statement

The existing system still has some critical drawbacks:

- The system cannot handle failure of any of its subcomponents. The whole system is centralized on a single master node for control values. Failure of this master node implies that the system is no longer useful. Hence we need a distributed system with redundant master nodes that would mask failures of its peer master nodes. The whole computational unit must be implemented as a real-time multiprocessor system that executes the same control applications even at the event of failure of any of the processor. We can conclude that the microkernel must be extended for multiprocessor support and to handle failure of peer units.

- The network protocol does not mask nodal or link failures. Francis [2] proposed a design to implement fault tolerance in the network protocol. However, there were few issues in the design that were left to be addressed. If we are developing a distributed system, the underlying network protocol should also be extended to support the new design.

- The current system has no support for the live addition of nodes once the network is started. The addresses and the configuration parameters are hard-coded. Once the network is started, there is no way that a new node could be added to it. Addition of new node could be done only after the whole system is shutdown and restarted. Hence there is a need to upgrade the current network protocol with plug and play support.

These drawbacks become the requirements used to frame the problem statement for this thesis:

```
How can we extend the design of the current dataflow real-time kernel to
support a true multiprocessor real-time system? This distributed
system should execute power electronics control applications with
fault tolerance. It should transparently mask the
failure of other processing units. The underlying ring network
protocol should also tolerate faults of individual links or nodes.
Finally, plug-and-play addition or removal of nodes within a live
network should be supported.
```

1.3 Fault Tolerant Embedded System for Power Converter Systems

This work focuses on modifying an existing micro-kernel to add fault tolerance. We shall look at the related literature and learn different paradigms that are developed for the design of such system. Fault tolerant multiprocessor scheduling for tasks with precedence constraints is a very tough problem. DARK executes applications that have dataflow architecture. Thus we have task precedence as a constraint to be satisfied while scheduling. Moreover, the network protocol also introduces the constraint of preventing an overload on the communication channel. After careful consideration of the different aspects of the system and the requirements from the final system, an offline fault tolerant scheduler was developed. This resulted in development of a new simple and static design for the local scheduler of the DARK OS. The new system was evaluated on a simulator that was developed based on the DSP emulators that were already present. From the results we found that the new computational unit made was faster as well as fault-tolerant. However, because of the large network delay, the resultant performance was found to be lot slower than what the computational unit is capable of. In spite of this performance degradation due of the network topology, the resultant execution time of the fault tolerant system is within the accpetable range. The DRPESNET protocol stack also now includes support for addition of new nodes into the live network.

1.4 Contributions of Work

The main contributions of this work are:

- A fault tolerant design and implementation of a real-time control system made of reusable and reconfigurable software components used for control in power electronic systems [1, 31].
- A fault tolerant multiprocessor scheduling strategy for hard real-time tasks with task precedence constraints and communication channel constraints.
- Plug and Play support in DRPESNET that enables hot addition of new nodes in a live, running network.

1.5 Organization of Thesis

The main body of this thesis is divided as follows.

Chapter 2 contains a survey of scheduling and fault tolerance techniques in distributed real-time systems and other topics that are relevant to the work reported in this thesis. At the end of the chapter, there is a brief explanation of the real-time system in hand.

Chapter 3 explains the approach used to tackle the problem and the fault tolerant strategy developed for the system. It explains the reasons for certain policy decisions. This chapter outlines the actual solution developed for the real-time microkernel to handle faults of the individual nodes.

Chapter 4 explains the implementation details of the whole design explained in the chapter 3. The chapter explains the kernel modifications as well as the implementation of the algorithm.

In Chapter 5, the work done in the underlying network protocol is explained in detail. It explains the design of new blocks added in the network protocol stack for the Plug and Play feature.

Chapter 6 outlines the evaluation strategy. It explains the design and the implementation of the simulator on which we evaluated our system. It details the results from the simulation. This chapter also evaluates the work on network protocol based on the VHDL simulation signals.

Finally, Chapter 7 presents the conclusions derived from this work. Chapter 7 also suggests areas of future exploration for improving the system.

Chapter 2

Background

2.1 Real-Time Computing Systems

The expectation from a real-time computing system [9] is different from other computing systems because of its emphasis on the timing constraints for generating outputs. Such a system is explained [6] as

> "A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced. "

In such systems, the output is termed as correct only if it is computed correctly and provided within the timing constraints. Such systems are installed for control in nuclear reactors, spacecrafts, robot systems, automation systems and many other applications of significance. The demand from application for the adherence to the timing constraints also varies from system to system. As we go ahead in this chapter, we would get into more relevant details.

2.1.1 Characteristics

H. Koptez [6] has defined a model for a real-time system. According to his model a real-time computer system is a part of the whole real-time system. A *controlled object* is a system that provides the input values (normally from an Analog to Digital Converter after digitizing an analog signal out of a sensor) to the real-time computing system and expects the control output in time for normal operation. An *Operator* is typically the person who uses this system. Based on this model, Koptez [6] terms a *real-time environment* as the one which comprises of the controller object and operator. In a real-time system we would find one or more processing units such as microprocessors or micro-controllers that run a control software.

The controlled object defines its state based on certain parameters. This state changes with time due to the variations in the values of the parameters that are governed by external environment. These external world parameters could be something as uncontrollable as the heat generated from a chemical reaction or the back Electro-Magnetic Force (EMF) generated by an accelerating motor or other feedback parameters from the system that needs to be controlled. The computational system holds the logic to control the values of these parameters. The purpose of the computation system is to regulate the state by giving control values to the controlled object. Hence as the state of the system changes with respect to time, these changes must be monitored by the processor. Based on the instantaneous values of these parameters, the computing system should feed the corresponding control values within the short period of time for control action to take place in time. This necessitates a periodic supply of state information to the computing system which in turn provides a corresponding regulation value within the deadline. Thus the whole system works in an endless *control loop* [6].When the computing system comprises of more than one node interconnected by a network between them, the system becomes a distributed real-time computing system. Now these set of nodes execute the control algorithm based on some resource allocation and task scheduling strategy such that they do not miss their deadlines.

Based on the perspective of consequences of missing deadlines, a real-time computing task could be classified into hard task and soft task. Buttazzo [8] explains these as followed:

> "A task is said to be hard if a completion after its deadline can cause catastrophic consequences in the system. A task is said to be soft if missing its deadline decreases the performance of the system but does not jeopardize its correct behavior."

From scheduling point of view, a task could either be periodic or be aperiodic. Buttazzo [8] explains periodic tasks as a set of tasks that are arranged for execution at constant intervals. Aperiodic tasks are also a set of tasks that are executed repeatedly but not at constant interval. In further sections of this chapter we shall get into details about the various aspects of the design of fault-tolerant real-time computing system.

2.2 Scheduling Algorithms in Real-Time Systems

A task could be either hard or soft and could be activated for execution in a periodic manner or it could get triggered as an aperiodic event. The dispatcher of the real-time operating system (RTOS) is responsible for scheduling tasks. The dispatcher normally runs an algorithm that decides the arrangement of tasks for execution with respect to time. This arrangement of task for a period of time is called the *schedule* of those tasks for that period. A real-time task is represented as (r, C, D, T). Cottet [7] explains these parameters as

> r - "release time" the point of time at which task starts its execution
> C - "worst case computation time" the maximum amount of time a task can take for execution
> D - "task relative deadline" once the task is started at time t, it should be completely executed by time t+D
> T - "task period", the period between successive activation of a task (valid only for periodic tasks)

The scheduling strategy could be executed online which means that the tasks to be triggered could be decided on the fly along with the actual real-time task computations or be offline, where the dispatcher could be made to follow certain statical order that is pre-computed before the tasks are executed. Scheduling can be preemptive, where the dispatcher need not wait for the completion of an executing task and could halt its execution for a higher priority task to execute. Optimality of a scheduling algorithm is measured in different systems based on different parameters. Typically it is measured based on the ability of a scheduling algorithm to provide feasible schedule. A schedule is termed as *feasible* if "every process starts after its release time and completes its computations before its deadline" [11]. Because of the complexity of the problem, scheduling algorithms use heuristics to compute schedule for processes. However certain systems perform exhaustive search and generate an optimal schedule. Let us first have a look at different modes of scheduling expected in real-time systems and see the different approaches for each of them.

2.2.1 Periodic Task Scheduling

Rate Monotonic (RM) Scheduling is a fixed priority scheduling algorithm. While making a scheduling decision, the task that has a higher frequency of execution is given higher priority. RM has been the most optimal of all the static priority algorithms [6]. If RM could not generate a schedule for a set of tasks, then it cannot be done by any other static scheduling algorithm. A feasible schedule is guaranteed if the processor utilization is less than 0.69. [7]

Earliest Deadline First (EDF) is a dynamic priority scheduling algorithm. At any instant of time, the task that has the nearest deadline is chosen for execution by the dispatcher. Thus this strategy has characteristics of a preemptive scheduling algorithm because an executing task can give way for a newly triggered task that has a nearer deadline. The functioning of the dispatcher is also independent of the period of the tasks, hence it could be employed even for scheduling aperiodic tasks.

Deadline monotonic scheduling is a static priority method. In this case, the relative deadline of the tasks decide the priority. The priority is "inversely proportional" [6, 7] to relative deadline D. It is an optimal algorithm. What makes this algorithm different is that it does not assume the deadline of the task to be equal to the period of the task, rather the period could even be greater than deadline. Least Laxity First is also a dynamic priority algorithm that makes scheduling decisions based on the relative laxity of tasks.

2.2.2 Aperiodic Task Scheduling

Even aperiodic tasks can have soft or hard timing constraints. As explained in section 2.2.1, the EDF algorithm could be employed in this case. This would also be a preemptive scheduling. But when the tasks are non preemptive the problem becomes NP hard [14]. There are various strategies that are used for handling aperiodic tasks. One way of handling them is schedule periodic tasks along with other periodic tasks that service these aperiodic tasks. In most cases, the dispatcher has to handle both periodic as well as aperiodic tasks. Now the problem is about handling criticality of these aperiodic tasks along with that of periodic tasks. One approach used to tackle this is by running the aperiodic tasks during the idle time in between the execution of periodic tasks. This is termed as the "background scheduling" of the aperiodic tasks. Another approach is to use a "server" (a periodic) task that is dedicated for handling requests for aperiodic tasks. The responsiveness to the requests of the aperiodic tasks is always a matter of concern. In slack stealing approach, the laxity of the periodic tasks are used for executing the aperiodic tasks without missing any deadline.

2.2.3 Scheduling Dependent Tasks

In cases where there are precedence relationships between tasks, we can still use the RM and EDF algorithm with some alterations in the parameters. In case of EDF, we have to modify the value of the release time such that it is greater than the sum of the execution times of all the tasks that precede it. For using RM algorithm, we assign each task a priority that is based on the precedence constraints and run the same algorithm based on this priority. Latest Deadline First uses a direct acyclic graph that maps the precedence relationship. At every point, from the group of nodes that has no successor or that has all its successors already traversed, the node with the latest deadline is taken and inserted in a queue. This algorithm gives an optimal schedule.

2.2.4 Multiprocessor Scheduling

In distributed real-time systems, the tasks are scheduled on multiple processors. In this case, the tasks must be assigned to a set of processors aiming a feasible schedule on each of those processors. The algorithms that we have noted until now for optimality on uniprocessor systems, may not guarantee an optimal schedule on multiprocessor systems. The problem of scheduling tasks on multiple processors is NP hard. In multiprocessor scheduling, an algorithm is also evaluated on the minimum number of processors it requires for feasible scheduling of tasks. Dhall and Liu [27] introduced the *rate-monotonic-first-fit scheduling* (RMFFS) as well as the *rate-monotonic-next-fit scheduling* (RMNFS) algorithm for scheduling real-time tasks on multiple processors. As in RM scheduling, each task is given priority based on its period of execution. The schedule decision is made at two levels. At the first level, the tasks are assigned to each processor after checking whether assigning the task to that processor would still result a feasible schedule. In RMNFS, if the task does not fit in a feasible schedule on a processor it schedules the task on a new processor and then continues allocating task on it without checking in any of the processors on which the tasks were assigned previously. In RMFFS, a task is checked for feasibility in schedule on all the processors instead of the most recent one in consideration. Multiprocessor scheduling should also take the communication cost between them. This plays an important role while assigning tasks with precedence constraints to multiple processors. Ramamritham [28] has developed a multiprocessor scheduling strategy that takes these factors into consideration.

2.2.5 Online and Offline Scheduling Decision

Scheduling decisions are taken either online or offline to the task execution. Online scheduling decides the next executing process on the run based on the tasks arrived in the ready queue for execution at that instant of time. The decision making process is done on the same processor inline to the execution of other tasks. Online scheduling thus enables dynamic scheduling decisions based on the arrival of new tasks. Offline scheduling however is run either before the system is deployed for execution or by using a dedicated hardware system for running the algorithm. In such cases the algorithm outputs a lookup table that takes different names in different systems. The scheduler refers to this table for deciding the next process for execution. Offline algorithms, especially those that generate scheduling decisions before the deployment of the system, are used in predictable systems, where the arrival of tasks could be predicted.

2.3 Dataflow Architecture

Before we get into real-time fault tolerance let us have a short insight into the features of the dataflow architecture.

Lubeck [32] pointed out user's perspective on dataflow model for parallel computing. According to him the dataflow model staves off any partitioning of the tasks by the user for parallel computing. Dataflow architecture holds an advantage over Von Neumann model when the execution of the program involves parallelism and requires synchronization of different execution threads. Ianucci [33] developed a *hybrid Von Neumann/ Dataflow* architecture and introduced the *macro dataflow* concept. In his work, he combined the features of both the models of computation.

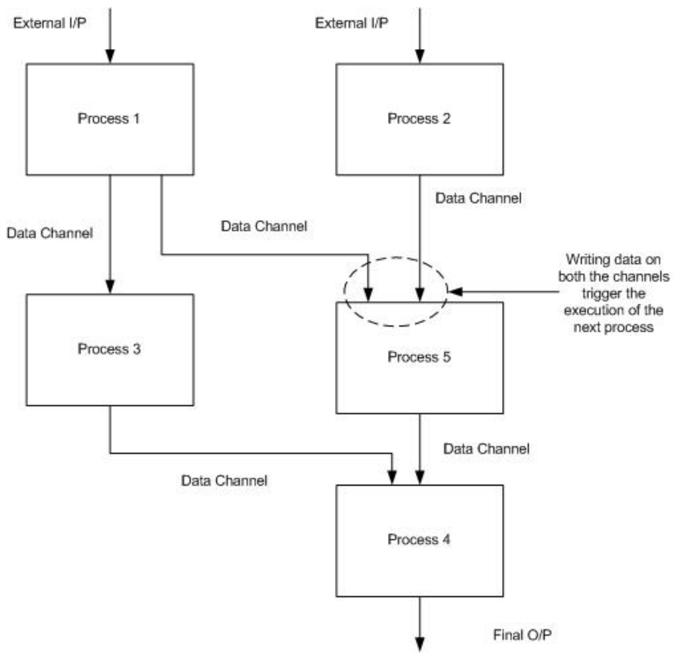


Figure 2.1: An Example Of Dataflow Graph. Note that the arrow lines represent the data channels that trigger process for execution.

A dataflow model of computation is typically represented as a directed acyclic graph as shown in Figure 2.1. In Figure 2.1, the rectangles denotes the task or process and the arrow between them denotes the data channels. A process is triggered for execution as soon as all its predecessor processes outputs data into its input data channels. Thus in this case, a process is scheduled for execution only after all its predecessor tasks are executed. A process here just means a *quantum of computing* [33]. It is also termed as an *actor*. The data channels are termed as an *arc*. An actor could execute only if all its inputs arcs contain data. In the macro dataflow graph, an actor that has its own state is called as *persistent actor* and those who do not have any state is called as a*regular actor* [36]. Singh has briefly explained the dataflow paradigm in his MS thesis [3]. A dataflow graph could be subject to any task allocation heuristics for generating a schedule with parallelism on multiple processors. As explained in Section 2.2.1, the task precedence constraint could be handled by manipulating the arrival time of the tasks to be scheduled such that the arrival time is after the completion time of all the parent tasks.

2.4 Fault Recovery in Real-Time Systems

Fault is "the cause of an error" [12]. Fault in the working of a system causes error. Error may result in non-conformance to the expected performance from the system. This gets termed as *failure*. A failure can be of different types. If a node continues working but gives erroneous outputs, we term this failure as *value failure*. If the computed output is correct but produced after the expected deadline, we term it as a *timing failure*. In real-time systems, timing failures must be avoided. A failure could also be categorized as consistent and inconsistent failure. Consistent failures are failures in which a subsystem either gives a correct output or does not respond. In such cases, if the system is not able to produce a correct output, it gives up or crashes, instead of providing the faulty output. In such cases, the processor is supposed to be have failed in a "fail-stop" manner [21]. Inconsistent failures occur when a subsystem provides false outputs. A malicious subsystem would provide conflicting outputs to the different parts of the system and cause failure in the working of the complete system. Such failures are termed as malicious failures. Let have a look at how such failures get gracefully handled.

2.4.1 Redundancy

In safety critical applications, like most of the real-time systems, redundancy is employed for better reliability. Redundancy here means repetition of presence of certain sub-components of the system with an expectation that one of these duplicated components would work perfectly in case of failure of others and the overall result from the system would not be faulty. Redundancy is employed in both hardware and software components. Most critical systems employ *space redundancy* [22], where we find replicated arrangement of hardware and software components that serves the same purpose in the system. Software components could be replicated in various manners. We can have the same processes running with parallelism at distributed locations. This is called *active replication* [23]. At the same time in certain systems we find that the process would be allocated resources and would be in the memory of the system but would not be executed. The component would be executed once the system gets a notification that the main process has failed during execution. This is called *passive replication* [23]. Active redundancy has proved to be better because of "better time properties" [10].

2.4.2 Fault-Tolerant Unit

A *Fault-Tolerant Unit (FTU)* [6] is an abstract term given for the use of space redundancy in a system. A fault-tolerant unit is used for hiding any single component failure. FTU could be made up of replicated subsystems. Even if one of the subsystems fails, the output of this fault-tolerant unit is still correct because of the utilization of the results from other redundant parts of the FTU. The kind of redundancy employed and the extent of replication used depends on the requirements from the FTU and the nature of failure of its subcomponents. For masking Byzantine failures [12], the FTU would have three times extra redundant nodes for the number of nodal failure that has to be masked. In cases where the system has to handle just the value errors, Triple Modular Redundancy [12] could be implemented in the system.

2.4.3 Time Redundancy

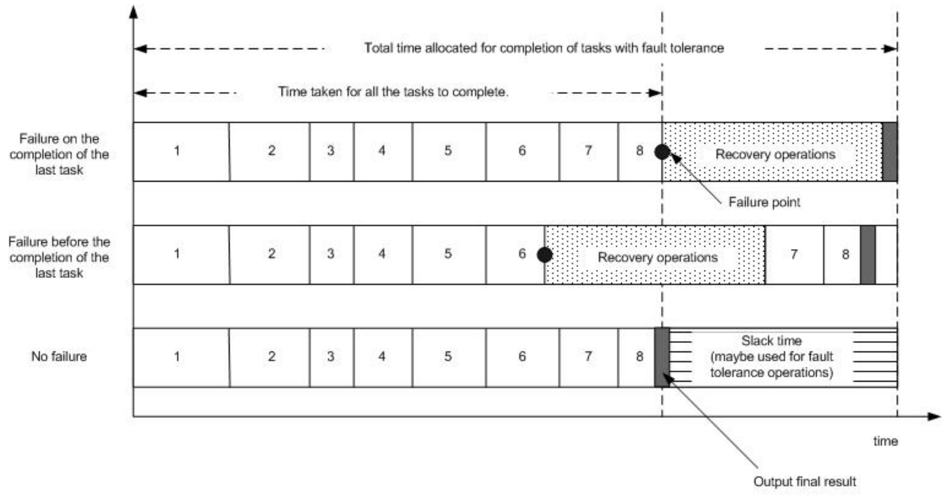

Total time allocated for completion of tasks with fault tolerance
Time taken for all the tasks to complete.
Failure on the completion of the last task
1
2
3
4
5
6
7
8
Recovery operations
Failure point
Failure before the completion of the last task
1
2
3
4
5
6
Recovery operations
7
8
No failure
1
2
3
4
5
6
7
8
Slack time (maybe used for fault tolerance operations)
time
Output final result

Figure 2.2: Time Redundancy

In order to achieve fault tolerance, many systems, on failure, perform extra operations to restore the non-faulty state. In such systems, fault tolerance requires extra time for the recovery operations. Hence the total period set for the completion of a set of tasks is greater than what it actually is. This increased time is used for recovery operations. This is how time redundancy is employed [22]. Figure 2.2 shows graphically how time redundancy could be used for fault tolerance. The extra time is for the recovery operation that needs to be done in order to attain a non-faulty state. The recovery operations could be workload redistribution [24], rollback recovery operations [25] or other kinds of recovery operations.

2.4.4 Fault-Tolerant Real-Time Scheduling

Section 2.2.4 described scheduling of tasks in multiprocessor systems. The scheduling algorithms discussed so far assume that the processors never fail during their course of execution. The scheduling algorithm must be extended to guarantee complete execution of tasks within their hard deadlines in spite of failures in subcomponents of its processing unit. Most of these extended versions of the algorithms that are used in scheduling real-time tasks employ some form of redundancy for fault tolerance. Fault-Tolerant First Fit RM scheduling (FT-FFRMS), introduced by Berossi et.al. [23], was an extension of the First- Fit RM scheduling [27] for tolerating processor failures. The FT-RMMS employs software redundancy in FFRMS. It uses active replication as well as passive replication of tasks. On failure, it reallocates the tasks and their copies based on the task allocation policy that is decided beforehand. Studies showed that FT-FFRMS needed lesser processors than systems that simply replicated a complete multiprocessor computing system for fault tolerance. Bertossi [29] also used the first fit strategy for task assignment but used deadline monotonic scheduling policy for local scheduling with similar duplication of tasks. Stankovic [26] showed a faster and decentralized approach for fault tolerance. This work assumed migration of tasks among the processors. However, these algorithms are meant to run on multiprocessor systems with shared memory. Hence they neglect the communication latency. They also do not consider task precedence and resource constraints into account. The FT-FFRMS algorithm is run on a separate processor and it is notified about the failure first and then it takes the recovery decisions for schedule over the multiprocessor systems. Maehle [30] also introduced a dynamic fault-tolerant scheduling algorithm that takes the precedence of tasks into consideration. Girault [41] introduced a static offline fault-tolerant scheduling algorithm that can run on heterogeneous systems. It also provides a generalized solution for static scheduling of tasks on multiple processors with fault tolerance.

2.5 Network Protocol Support for Real-Time Systems

Network protocol design, for real-time systems, should be able to guarantee timely transmission of messages, especially, those meant for critical tasks. Kopetz [7] pointed out that even the OSI standard for network protocols does not consider the communication delays between two ends as an issue of interest. Hence, message scheduling is another important design issue in case of real-time systems. For timely execution of critical tasks, we need guaranteed scheduling for transferring messages deterministically within the timing constraints. Probabilistic scheduling strategy, based on the probabilistic/statistical information derived from network, could be used in cases where occasional delays in completion of hard tasks are tolerable by the system. Network protocols for real-time systems could be classified as event-triggered and time-triggered protocols. Event-triggered protocols are more useful in systems where the communication between subsystems are not predictable. Time-triggered protocols are effectively used in predictable systems where the system beforehand knows the schedule of messages and their acknowledgments in the network.

2.5.1 CAN

In controller area network (CAN), the nodes reach the bus via the CSMA-CA access [7].CAN is an event triggered protocol. Tindall [17] introduced a real-time architecture based on the CAN protocol. In CAN, the priority is deduced from its network identifier. For real-time scheduling, based on the RM algorithm, one can create network identifiers as a function of the period of the use of network by the node. Thus CAN protocol could be mapped as a predictable network protocol that could schedule messages with deterministic guarantees.

2.5.2 FIP

FIP (Factory Instrumentation Protocol) or WorldFIP is used for interconnecting sensors, actuators and processing units. FIP consists of a bus arbitrator node, which is dedicated for controlling or scheduling the communication between the other nodes in the network. FIP is a time-triggered protocol. Messages are scheduled for transmission based on the schedule that is stored as a bus arbitrator table. This table contains the instantaneous information about which nodes in the network should exchange data at any point of time. The information in this table dictates the performance of the protocol. Tovar [16] showed how RM and EDF algorithms could be used to ensure that the message exchange between the nodes could be done within deadlines. One important aspect about this protocol is that the communication is not initiated by the sender but it transfers the data when asked by the arbitrator [7]. This is not the case in CAN, where the messages are always initiated by the sender.

2.5.3 FDDI

FDDI (Fiber Distributed Data Interface) has a ring topology [18]. In this protocol, there is only one node that has the token at a time for communication with other nodes. The token is circulated in the physical order among the nodes and each nodes have fixed amount of time to transmit the message. The parameter Target Token Rotation Time is sometimes used for decisions on instantaneous allocation of medium to any node. Apart from the real-time constraints on the FDDI protocol we need to consider the fault tolerance issues in the protocol because the protocol that we are working on is also ring protocol. Sankar [39] showed how reconfiguration can be achieved on the event of failure in FDDI protocol using two rings. Chen et. al. [40] developed FDDI based fault-tolerant communication protocol that provides support for timely delivery of hard real-time messages.

2.6 Few Existant Real Time Distributed Systems

In this section, we shall have a look at few established fault-tolerant real-time systems. We shall have a look at the Spring and MARS system. We shall see some notable features that these systems have.

2.6.1 Spring

Spring [19][20] is a real-time distributed system. The system consists of redundant arrangement of a multi-processor system. The application processors execute critical tasks of the system. The system processor makes the scheduling decisions and makes temporal adjustments to avoid unpredictable delays and system overheads on application processor. The I/O subsystem interfaces with peripherals that has an unpredictable response time. The system processor takes care of time critical I/O operations.

Scheduling in RTOS The tasks in Spring system are classified as critical, essential and unessential. The critical tasks are guaranteed response at compile time. The essential tasks are scheduled online. The unessential tasks are scheduled during the idle time of the processor. In this system, a task is scheduled only if all its predecessor tasks are executed.Scheduling is implemented in the Spring RTOS at four levels.

- Every application processor runs a simple scheduler that picks task from the ready queue and marks it for execution.

- A local scheduler is run by the system processor that dynamically generates a feasible schedule for the tasks on application processors.
- The distributed scheduler tries to accommodate a locally non-guaranteed task in any other processor for feasible schedulability.
- This part of the scheduler adapts the scheduling algorithm according to the changes in instantaneous values of external parameters that affects the task schedule.

The Spring system has dedicated hardware for the purpose of scheduling. The Spring system categorizes the tasks and implements different scheduling policies for different kind of tasks. As an example, all critical tasks are guaranteed a feasible schedule beforehand while the essential tasks are scheduled online. The designers of the Spring real-time system presents us a paradigm that integrates “flexibility with predictability”. The real-time system scheduling architecture maps the different types of tasks such as the critical tasks, non essential tasks and aperiodic tasks as a predictable series of tasks for which the online scheduler finds an optimal schedule.

2.6.2 MARS

MARS stands for Maintainable Real-Time Systems [10]. It is a fault-tolerant distributed real-time system. It was designed primarily for tasks with hard deadlines. The system is designed for predictable behavior even at high loads. The architecture comprises of a cluster that contains the computational systems along with the data acquisition nodes. These are connected to each other by a high speed synchronous real-time bus. Fault tolerance is achieved by active replication at cluster level. It uses TDMA protocol on Ethernet link for timely communication. Replication of messages is employed on these links for fault tolerance.

Static priority scheduling ensures predictable behavior at overloaded conditions. The applications running on the MARS system has a dataflow architecture. The schedule for hard real-time tasks are computed offline based on the precomputed execution times of the task, message scheduling delays and communication between other tasks. The hard real-time tasks are executed according to this schedule. The soft real-time task are scheduled during idle time at lower load. Based on such design, MARS has a deterministic behavior during overload conditions [8].

2.7 Power Electronics Building Block Design

The CPES lab at Virginia Tech developed a novel technique for making control systems in power electronics [34, 35, 38]. This system eliminates the dependency on the analog circuits for control operations. This system has a real-time computing system that executes control applications. These control applications implement control algorithms that is derived from the functioning of the hardware circuit normally used in control system for power electronics.

2.7.1 Hardware design

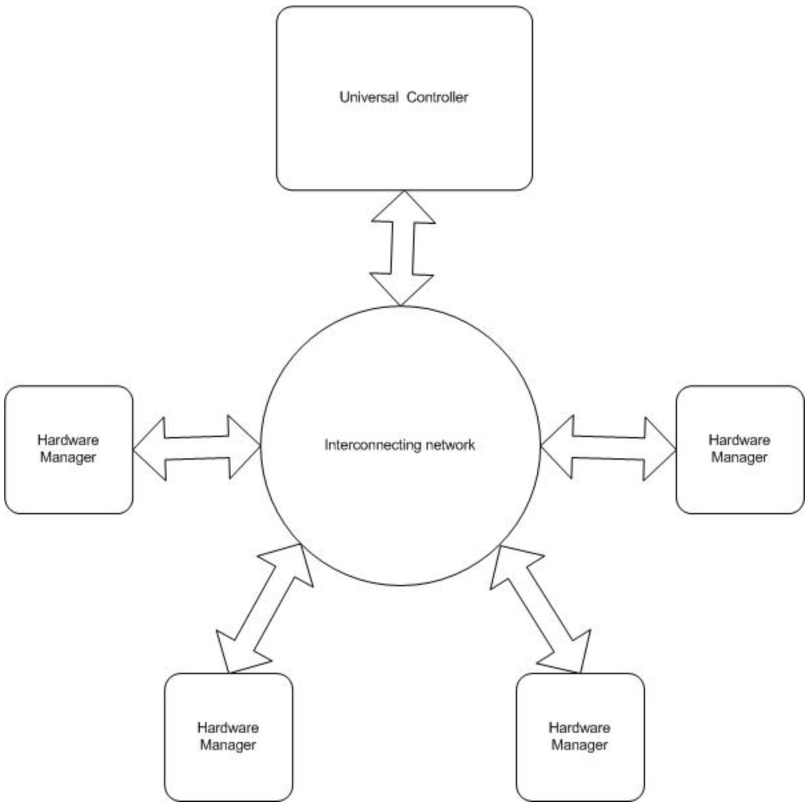


Figure 2.3: High level abstraction of the PEBB hardware components

The hardware of this system is as shown in Figure 2.3. The real-time system contains a master node called as *Universal Controller* and slave nodes by name *Hardware Manager*. From what we know after reading Section 2.1.1, Hardware Manager (HM) is the controlled object in our real-time system. It samples the input analog values from the external world, converts it into digital values and periodically feeds it to the Universal Controller (UC) and expects the control values within a specific deadline. The UC has a processor that executes a real-time microkernel. The software running on this micro-kernel provides timely control outputs to the HM's. The Universal Controller has the role of a real-time computer system. The architecture of both UC and HM is explained in MS thesis of Francis [2].The whole system is centralized on a UC for control values. This means that all the real-time computations are done on a single UC. The processor on the Universal Controller is an ADSP 21160 processor that executes the control applications on a microkernel. The processing speed of the DSP is 80-100 Mhz. This means that one cycle period of the DSP could be calculated as 12.5ns. This DSP has memory mapped interface with the FPGA memory where the network protocol stack is implemented. This interface forms the communication interface between the processing unit and the communication channel.

2.7.2 Network Protocol

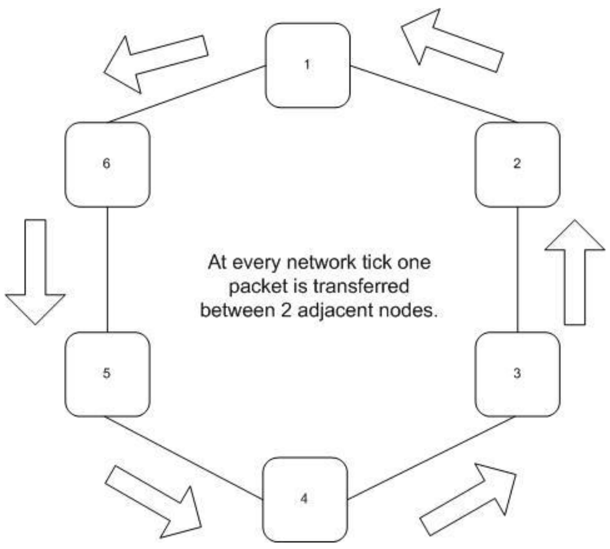

1
2
3
4
5
6
At every network tick one packet is transferred between 2 adjacent nodes.

Figure 2.4: The flow of packets in DRPESNET protocol. Each arrow denotes the transmission of a packet through the link at that instant of time.

These nodes are interconnected to each other by 2 daisy-chained fiber optic dual ring networks that run in the opposite direction. The communication between these nodes is based on Dual Ring PESNET (DRPESNET) protocol [2]. This network protocol is different from a typical token ring protocol like FDDI. In this case, the ring contains as many packets as the number of nodes in the network at an instance of time. The packets are transferred in a “lock-step” fashion [2]. Figure 2.7.2 explains this. The whole protocol is time-triggered. At every network tick, the adjacent nodes transfer a single packet among them on the primary ring. Thus a packet once put on the network is assured to reach the communication interface of the destination node within n-1 network clock ticks where n is the number of nodes in the network.

2.7.3 Communication Interface

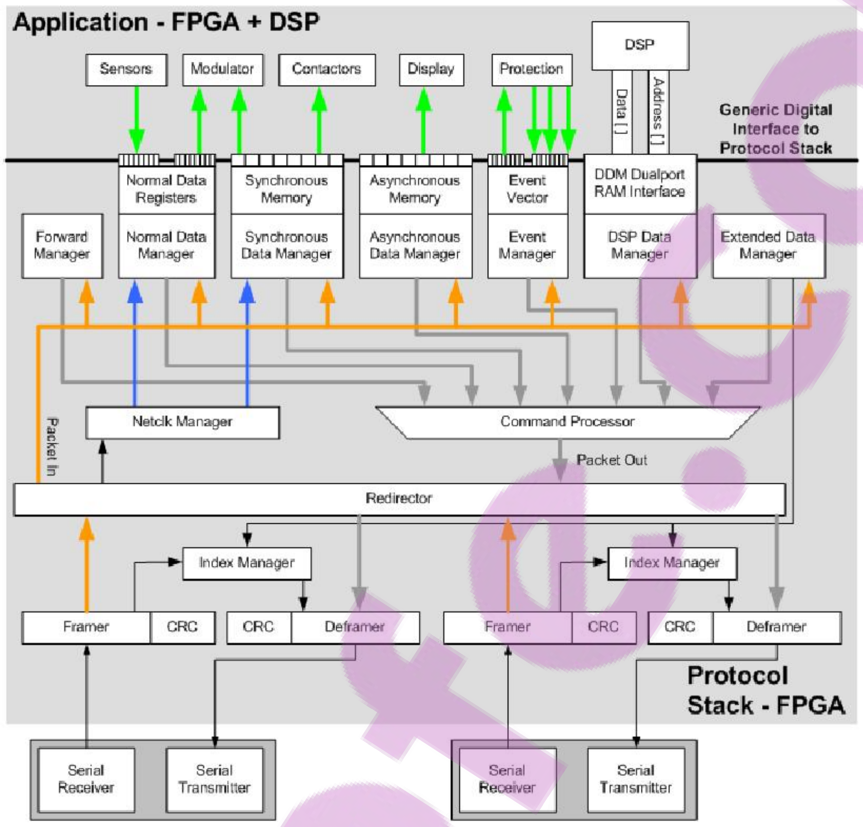


Figure 2.5: DRPESNET protocol stack implemented in FPGA. Note the point of interface between the FPGA and DSP [2].

Francis [2] developed the DRPESNET Protocol Stack in the FPGA that interfaces with the DSP. The protocol stack contains the different blocks that have their own responsibility for the respective packets received from the network. For every type of packet in the network, the protocol has a corresponding packet manager. The diagram that shows how the data manager interfaces with the DSP. The input and output buffers are directly read from or written into by the DSP. The functionalities of all other types of packet mangers are described in MS thesis of Francis [2]. These different packet managers behave according to the necessary commands coded in them. The command processor block decides whether to send this packet to the managers based on the network address in a packet. Again the command processor decides which packet to send in case of contention among the packet managers for sending the output packet messages.

Data manager serves the purpose of DSP-FPGA communication interface. The data manager consists of buffers that accepts and stores the data

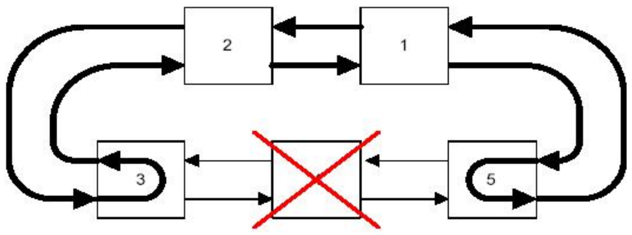


Figure 2.6: Redirection in the dual ring network protocol like DRPESNET

packets picked from the network until it is read by the DSP. When the DSP writes a packet into the network, it is kept in the buffer until it receives a null packet in the network to replace with. On receiving the null packet, the data manager sends the least recent packet in the buffer into the network.

Apart from this, Francis also introduced the concept of *redirector* in his thesis. The redirector is used in the event of failure in any of its link or its adjacent node. Figure 2.6 shows an abstraction of this redirection. The failure ring is used only on the event of a link or nodal failure. On the event of failure the packets are redirected from primary to the secondary ring at one end and at the other end the packets are routed back to the primary ring. The development of this block is explained in Chapter 5. The DRPESNET protocol has all the nodes with a predetermined static address. Since all the network addresses and the configuration parameters are hard-coded. Hence there is no way that a new node could be added to the live network. Hence there is a need to address this issue of configuration of a newly added node.

2.7.4 DARK Micro Kernel

The Dataflow Real Time Kernel (DARK) is a microkernel that executed by the DSP on the Universal Controller. Figure 2.7 shows the dataflow graph of a typical application that is run on the DARK. We have already learnt about dataflow graphs in Section 2.3. In our system, each task or actor is called as an Elementary Control Object (ECO). An ECO could be described as an abstraction of a part of the control circuit that our system is eliminating. Francis has given a very formal definition of an ECO in his Master's thesis [3]. From our perspective, an ECO is a process that does some small task when triggered by the data it requires for execution. Figure 2.7 shows a typical control application that has dataflow architecture and is executed on the DARK microkernel.

Currently, the DARK scheduler reads the ready queue and executes the first task in the queue. Every ECO gets triggered for execution (gets added to the ready queue) once all its input data channels are fed by data. This happens only when all its predecessor ECO's have finished their execution.

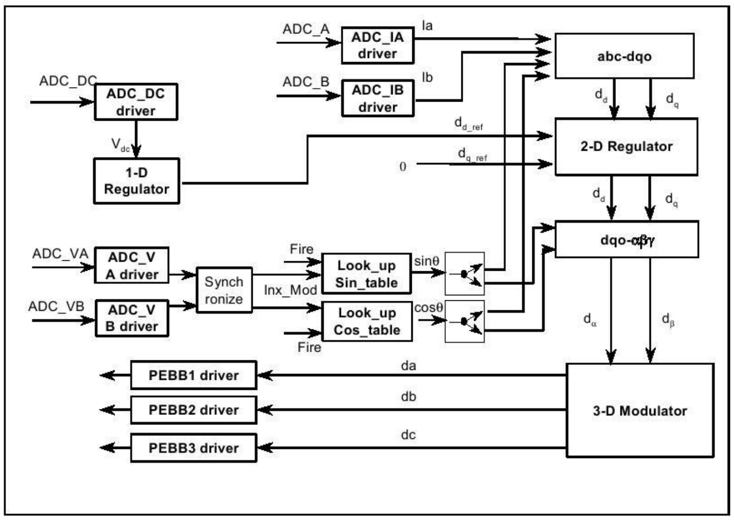


Figure 2.7: Dataflow diagram for the boost rectifier application. Each box in the diagram represents an ECO . The arrows that connects these boxes represent the data channels.

The DARK has different modes of execution. In multithreaded form of execution the ECO's run parallely. At an instant, an ECO is a part of either waiting queue, where it waits until all its input data channels are fed by data, or in the execution queue where the ECO is ready to run and is waiting for a signal from the scheduler, or the ECO could be blocked on a write or read operation. The DARK also runs on the dynamic schedule option, where an ECO is triggered for execution by the scheduler as soon as its input data channels are fed by the input data. Single threaded static schedule option triggers the ECO based on a predefined order. The results explained by Jinghong [5] shows that the context switching cost in the multithreading mode increases the switching period. Single threaded static schedule option was found to be the fastest.

2.7.5 Targeted System

To make the current system more reliable, we need a redundant computing system. There must be more than one UC in the network that is dedicated for the purpose of generating control values. As shown in Figure 2.8, a group of Universal Controllers would form a fault-tolerant unit for real-time compting system. This requires a distributed version of the DARK microkernel. This distributed version of the DARK would have a scheduler that would schedule tasks over the complete multiprocessor system such that failure of an individual UC could be masked. This scheduling policy should employ the fault-tolerant measure that was discussed in this chapter. Now, we again redefine our problem statement as *"Develop a distributed real-time*

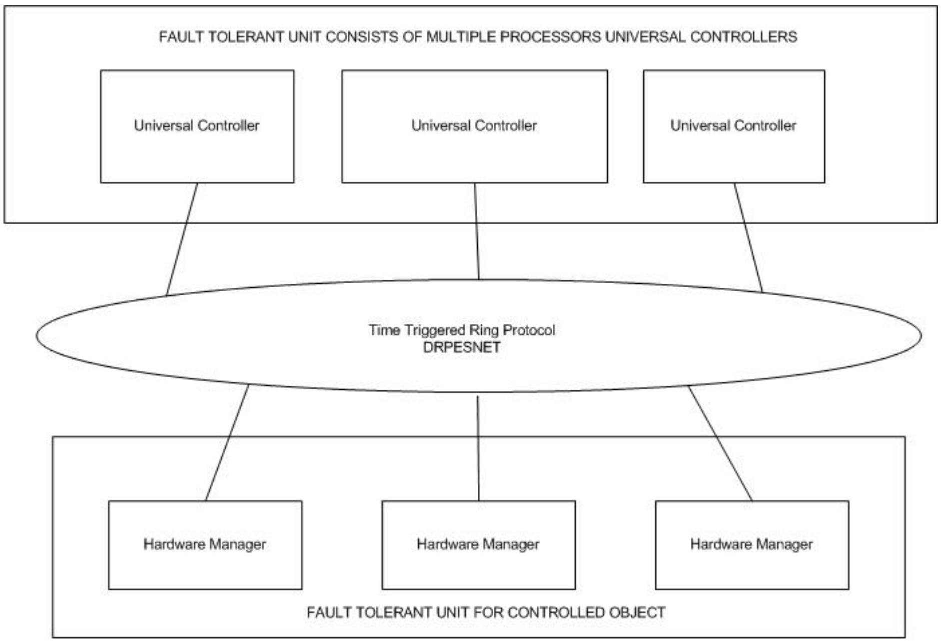

FAULT TOLERANT UNIT CONSISTS OF MULTIPLE PROCESSORS UNIVERSAL CONTROLLERS
Universal Controller
Universal Controller
Universal Controller
Time Triggered Ring Protocol
DRPESNET
Hardware Manager
Hardware Manager
Hardware Manager
FAULT TOLERANT UNIT FOR CONTROLLED OBJECT

Figure 2.8: The targeted system for fault tolerance

computing system such that the system could recover automatically from failure of individual UC. For this design an ECO scheduling policy over the multiprocessors such that they can recover from failures of individual Universal Controllers that execute them. Upgrade the underlying network protocol to provide support to such a decentralized system. This upgradation should also include the support for addition of nodes into the live network also called as the Plug and Play support."

We would again redefine the problem statement in a very detailed manner after considering the different issues that have to be tackle. In Chapter 3 we shall first evaluate our systems characteristics based on what we have seen until now. Then based on that we shall have a computer science perspective to the definition of the problem.

Chapter 3

Design

Chapter 2 presented the most important parameters and their influence in the performance of a distributed real-time system. Also, Chapter 2 provided a brief look at some of the most important approaches considered for designing such a system. We even saw how few of the successful systems have adhered to certain common principles and policy decisions. Before getting into the details of the solution described in this thesis, we shall consider the different aspects of the design. After that, we redefine the problem statement in terms of a pure scheduling problem.

3.1 Design Preamble

This section presents the key aspects of the system that affected the most important design choices.

3.1.1 Redundancy Options

The whole control function is fragmented into software components called ECO's (Elementary Control Objects). These ECO's are the basic execution units of the system. The ECO's communicate with each other through data channels. The parent ECO writes its outputs into its output data channels. The child ECO reads its input parameters from the common data channels between the parent and child. In this system, the data channels form a basic memory unit for storage of intermediate values during execution. Most of these ECO's are "stateless" entities. The term stateless because the ECO's read values from its input data-channels, do some computations and write the results into the output data channels. It is very rare that an ECO performs computations based on its previous outputs. However, ECO's that simulate circuits like regulators need the output values computed in previous switching cycle. Such ECO's need to get updated every switching cycle.

One way of doing this would be by using *approximators* that would compute the approximate value faster with simpler computations. If the task allocation can result in more slack time on a system, the slack time could be used for fault tolerance measures, in case of failure. Or we can have a pro-active non critical process for updating the state of the system, that gets scheduled during such slack time. We can decide on an acceptable switching period that is greater than the cumulative execution times of all the critical tasks and use this slack time for recovery measures on the event of failures.

In order to implement a redundant computational system, we have to replicate the different execution paths of the dataflow graph. This could be done by replicating a set of ECO's actively or passively. Passive replication implies that the set of ECO's would be triggered for execution on failure notification of the master copy of the same set of ECO's. Active replication is done by executing the same set of ECO's parallely on separate processors. We should also note that updation of the output data channel of a passive ECO with the current values eliminates the need for active replication of those ECO's on that processor. For time redundancy, adding slack time in between the execution of tasks is acceptable, if the resultant switching period from the schedule is acceptable.

In order to employ time redundancy, the induced slack time in the system should be tolerable with respect to what we discuss in 3.1.8. Software redundancy here implies replication of execution of group of ECO's. All the ECO's gets passively replicated on all the processors. There should be a judicious selection for scheduling active replicates of the ECO's.

3.1.2 Communication Delay

The communication delay between the processors is a serious issue in this system. Communication between two ECO's executing on different processors requires the following steps:

1. The parent ECO writes the output of its computation into its local output data channel.

2. The value written in the output data channel should be broke into data packets and written into FPGA memory buffers. This step is costly because the processing time required to do this operation is comparable to the execution cost of an ECO. For future explanations, we would refer such a task as a *send task.*

3. Once the value is written in the FPGA buffer, the packets have to wait until the node receives a null packet. The arrival of the null packets is completely dependent on the network traffic. Parol[4] has explained a method of assuming the value for this delay.

4. Once the data packet enters the network, there is a deterministic guarantee that the packet would arrive the destination within a value of time that equals (N-1) times of a single nodal delay. This is the case when there is no failure in the primary ring. However, to be safe we take the value of the delay caused in the event of any possible nodal or link failure. Hence, we assume the value to be 2(N-1) times single nodal delay.

5. Once the data packets are written in the FPGA data buffer from the network, they have to wait until the the DSP executes a task that reads from this buffer. The wait depends on how early such a task is scheduled. There is a high possibility that the new packets are written into the buffer before the old packets are read from it. This results a buffer-overflow. The node would no longer receive any new packets from the network. This situation is equivalent to the failure of that node. To make things worse, the nature of this failure is such that the other nodes in the network are never notified about its occurrence.

6. When the DSP schedules such a task, it reads from the FPGA memory and writes them into the data channel. The cost of this task is same as the send task. We shall refer such a task as a *receive task*.

The execution time of a send task or receive task is comparable to that of other ECO's. From the scheduling point of view, these tasks also add a precedence constraint that should be satisfied while scheduling. Once a packet is in the network, the communication delay depends on the number of nodes in the network. The typical nodal delay value is 1280ns. Hence if we have a system with 20 nodes, the communication delay with failure would sum upto 49920ns. This value is high compared to the execution cost of an ECO. Hence *the communication cost could reduce the extent of the improvement in the performance when we schedule tasks with parallelism, based on the structure of the DAG.*

3.1.3 Critical Nature of Tasks

Every tasks in this system are pre-run in the emulator and the execution time of all the tasks are set as their respective worst case execution time. All tasks would complete their execution within this worst case time. Thus all tasks have a predictable behavior. All tasks are critical in this system. Once the input driver ECO takes the sensor values from the external world, the output driver ECO should write the computed control value into the network within the end of the switching cycle. All ECO's contribute to the computations for generating the output values. Hence, we cannot consider any of these tasks in the dataflow graph to be a non-critical task. However there are certain tasks that do not belong to the dataflow graphs. Tasks such as the ones for configuration of a newly added node in the network are non-critical and do not have any timing constraints. *The ECO's that are part of the dataflow graph are critical tasks and those that are responsible for the configuration of other nodes in the network are non-critical.*

3.1.4 Scheduling for Distributed Dataflow-Channels

The ECO's are data-driven. This means that they start execution as soon as the input data is written into their respective input data channels. However, for completion of the tasks on time they have to be periodically executed. When all the ECO's are executed on the same system, firing of the input driver ECO's triggers the execution of the subsequent tasks satisfying the precedence constraints of the data-flow graph based schedule. When the parent and child ECO's are allocated on separate processors, the data-channels that are used for communication between them are present on local memories of both the processors. Such a data-channel is termed as a distributed data-channel. As soon as the parent ECO writes into its output data-channel, the value written in the local data channel must be copied to the same data channel on the local memory of processor where the child ECO is executing. The schedule at each of these processors must ensure the timely arrival of data even in other processors that share the same data channels. Section 3.1.2 described the communication cost factor. This means that the schedule of the parent and child tasks on separate processors should take care that the data channel at the child end is updated before the execution the child ECO. This is an addition to the precedence constraint that we have. The parent task, the send task, the receive task and the child task must be scheduled in the order mentioned. Hence *the task assignment policy should take into account the delay for sending data on distributed data channels and schedule the dependent task only after the distributed data channel updation delays are counted.*

3.1.5 Expected Failure

Handling value failure is out of the scope of this thesis. We assume that when a node that fails, it does it in a fail-stop manner. It could be a permanent failure. When a processor is informed by the network protocol that a node is failed, it assumes that the node no longer exists. It eliminates all the prior information that it had about its existence and makes necessary changes in the (h-state) current state to adapt to the failure. Even when a node finds itself to be notified as a failed node by the network protocol, it reinitializes itself as a newly added node and waits for new configuration. This could happen when links on both sides break. The logic for the configuration process of this new node is also out of scope of this theses. *The processors in the system are fail stop processors.*

3.1.6 State Checkpointing Options

Most of the ECO's are stateless entities. The term "stateless" here means that any ECO for its current execution does not need its output computed in the previous switching cycle. Such ECO's could be reallocated on any processor without any concern about its state of computation. The ECO would invoke its operation only when it gets triggered by all its input data channels. For other ECO's like modulator, the current output is based on the current input and the output from its previous execution. Hence to reallocate the task on a new UC, we need to ensure that the ECO is updated with its previous output state before execution. Hence,, we need a mechanism for checkpointing the state of such stateful ECO's, so that the newly allocated version of the ECO's do not result in wrong outputs. *Between two switching periods, the stateful ECO's define the state of computation. There is a need to checkpoint only the state of the stateful ECO's with respect to time.*

3.1.7 Task Migration

If an ECO that is currently executing on a processor is moved to another processor for fault-tolerant scheduling reasons, we need to transfer the complete state of the ECO. The communication delay is large in this system. The state of an ECO at an instant of time is defined by the values of all the variables at that instant of time. Migration of an ECO would mean copying the stack entry for that ECO from one processor to another. Considering the high communication cost, this is a bad idea. In order to avoid the penalty of such large communication cost, we can say that *this system does not support migration of ECO's for scheduling purposes.*

3.1.8 Performance

Here, a centralized computing system is converted into a distributed one. Hence it is expected that the new system should incorporate the benefits that we get from a distributed system over a centralized one. While fault tolerance is the primary goal, one would expect the new system to be more efficient, or at least have a fault-tolerant system executing with the current performance.

Guo [5] points out the distribution of CPU cycles for various sub tasks involved in executing the concurrent processes in a graph. The graph shows how a single threaded static scheduling performs better than the dynamic schedule in multithreading mode. The context switching cost makes the difference in this case. Thus a static schedule is preferred for a better performance. In power electronics, the preferred switching frequency should be in the range 2 to 20 kHz. *The scheduler should create a feasible schedule with fault tolerance such that the resultant switching period is acceptable.*

3.1.9 Scheduling Mode

The scheduling decisions could be made online or offline. On failure notification, an online scheduler would have to allocate and deallocate the tasks based on some fault-tolerant reallocation strategy. The cost required to execute these reallocation steps is unacceptable because there are chances that the switching deadline gets missed. This motivates us to decide upon an offline scheduling strategy. We can use dedicated hardware system that makes scheduling decisions on failure notification. On failure notification, this offline scheduler would require to generate new schedules for each processor in the multiprocessor system. Now, to communicate the schedule for each processor over the network is again a costly affair. We are targetting a predictable system. We do not prefer a dedicated task scheduler, that would increase the network traffic on an unpredictable failure event. To avoid these complications, we should pre-compute the schedule for each processor, such that the resultant schedule over the multiprocessor system is tolerant to any individual processor failure. Hence, *the scheduling decisions should be taken offline and should be computed before the system is deployed for use.*

3.2 Assumptions Made for the Design.

We make the following assumptions for the design.

- All the processors have the same execution speed.
- The failure is always going to be in a fail stop manner.
- In case of distributed data channels, we assume that the packet is put into the network within a predictable amount of time.
- We assume that the reconfiguration module of the kernel shuts down the system gracefully on failure of certain vital subcomponents. These subcomponents could be the hardware managers that are responsible for the functioning of few critical ECO's. An input driver ECO is an example of a critical ECO. Blocking an input driver ECO would mean blocking the whole control algorithm from execution. Hence the reconfiguration module is expected to shut down the Universal Controllers as soon as these failures are detected.

3.3 Fault Tolerance Strategy

Section 3.2 considered the various aspects of the design. Section 3.1.9 proposes an offline scheduling strategy as a solution. Next, we describe an offline scheduler, which implements a task allocation strategy that distributes the task and its replicates among the processors.

3.3.1 Problem Statement Revisited

Schedule the tasks whose precedence constraints are represented by a directed acyclic graph G(V, E) on N number of processors. The schedule of tasks should satisfy the following constraints:

- *Employ replication such that the whole system must be tolerant of k processor faults.*
- *The schedule should take into consideration the task precedence before scheduling a task and its replicates. A child task should be scheduled after the completion time of all the replicates of the parent tasks.*
- *Any communication between two processors must ensure scheduling a send task at the source processor and a receive task at the destination processor such that the receive task is scheduled after the completion of the send task. The schedule of the receive task should also take into account the network delay after the complete execution of the send task.*
- *At no point of time the schedule should overload the communication interface of any of the nodes.*

3.3.2 Task Allocation Policy

The task allocation policy should take into consideration the following factors:

- Attain maximum parallelism possible to achieve a faster distributed computational unit.
- The communication between the ECO's should be minimum.
- The allocated tasks should also result in a feasible schedule considering the communication delay in updating the distributed data channels. This means that the allocated tasks on a processor should satisfy the schedule on that processor.
- Optimality in the number of processors should also be a matter of concern.

3.3.3 Fault-Tolerant Task Allocation Algorithm

This section explains the fault-tolerant offline scheduling algorithm. The algorithm has 2 phases of implementation:

- Clustering phase
- Scheduling phase

The clustering phase pre-processes the input before scheduling the tasks. It can also give an idea about the minimum attainable switching period for the given input. In scheduling phase the tasks are scheduled over the multiprocessor system with fault tolerance.

Input:G(V, E) a directed acyclic graph (DAG). The edges have a cost associated with it. This cost is assumed to be the worst case communication cost at the event of failure. The nodes V also have their associated weights. These weights are the worst case execution time of the corresponding ECO.
Number of processors p
Number of processor failures to be masked k
Desired Switching period. - T

Output: A system schedule table that contains scheduling information for all the processors. This table is used by each of the processors to schedule locally tasks so that the overall schedule makes the system tolerant of k processor failures.

Clustering Phase

The primary reason to have such a phase is to avoid any unwanted communication task between 2 ECOs. It is a preparatory step for the next phase. This phase processes the input graph for the effective working of the next phase. This phase dictates the order of selecting a task for scheduling. Following are the definitions of few terms that are considered before the actual scheduling of the tasks. The cost of communication between 2 nodes are assumed to be a constant. Figure 3.1 illustrates the phase operations.

List is defined here as a set of tasks arranged in an order of their precedence during execution. A list has only one path of action, which means that no node other than the source node (or the first node in the list) is dependent on the execution of a node in any other list or sequence. Thus executing a list is similar to executing a linear sequence of instructions. In Figure 3.1, the lists are (10-11), (1), (2), (3), (4), (8-9), (12), (5-6), (14-13), (7), (16-19), (20-17), (15-18).

Sequence is defined here as a linear cluster which contains a linear list of task arranged according to the precedence constraints. However a node in the list could also be dependent on the execution of node in any other sequence. In Figure 3.1, the sequences are (10-11-4-5-6-7-16-19), (8-9), (14-13), (20-17), (15-18), (1-3), (2), (12).

Dominant Sequence is the sequence of tasks that begins from one of the source ECO's and finishes into a destination output driver ECO such that the path has the highest weight (execution time) among the remaining sequences. The total cost of the dominant sequence is the minimum possible

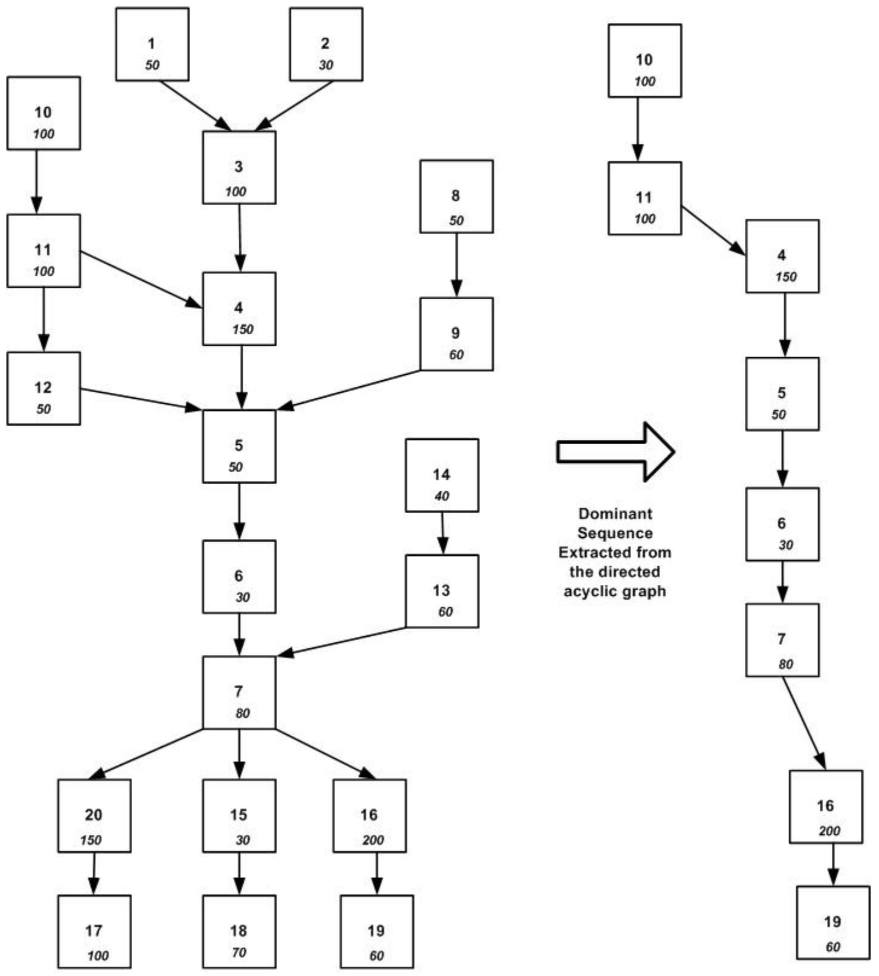


Figure 3.1: Getting the dominant sequence from the acyclic directed graph. The number written in italics shows the cost and the number at the center shows the node number.

switching period. In Figure 3.1, the sequence (10-11-4-5-6-7-16-19) is the dominant sequence.

Cluster is the set of tasks that could be in the form of one or more list or sequence. A linear cluster is same as a sequence and a non linear cluster could mean a set of lists or sequences considered together for scheduling purposes. As an example we consider the tasks (10-11-12-4-5) as cluster.

We define a communication task that has cost C. This cost is equal to the time taken for transferring data between the local memory and the FPGA memory.

Step 1: Find the dominant Sequence in the data-flow graph. Typically this dominant sequence should be made up of blocks of lists delimited by a link to other lists or sequences. The dominant sequence *DS* is found using the Dijkstra's algorithm for finding the longest path. Let *seq_list* be the list of sequences. Add *DS* to the list of sequences.

Step 2: Remove all the nodes in the dominant sequence from the graph. Repeat the following steps until all the nodes are recorded in one of the sequences.

- Find the sequence *m_seq* with maximum cost from the incomplete graph.
- Add sequence *m_seq* to the list *seq_list*.
- Remove the nodes in *m_seq* from the graph G.

Find all the lists from the dominant sequence. For every list

- Insert a communication task each at the beginning for all the communication with source
- Insert a communication task each after the task for communication with the dependent sequences.

Summation of this should be the minimum switching period possible. In the next step, we try to achieve parallelism with the schedule of the dominant sequence, In case of failure to do so we cluster the task with the current list.

Scheduling Phase

In scheduling phase, we schedule tasks and its replicates on the processors. At every step, we allocate the tasks to the processor such that a set of constraints are satisfied. However we have a scheduling problem in hand than an allocation problem. Every task that gets allocated has an associated start time for execution. Allocating a task on a processor implies that the task must be executed from the start time that is specified. We have the following constraints to satisfy before scheduling.

1. Task precedence is the most important constraint to follow. A task should be considered for scheduling only after all its parent tasks are scheduled.
2. If the child and the parent tasks are allocated on separate processors, then the child task should be scheduled at a time after considering the worst case communication cost for transferring the output of the parent task to the itself.

3. The resulting schedule should not cause any message overflow at the DSP-FPGA communication interface.

4. A replica of the task should not be scheduled on the same processor as itself.

5. The goal of the scheduling is also to minimize the time taken for the complete schedule.

Mapping the Communication into the Schedule

Communication between two ECO's on the same processor does not involve any communication cost. The cost required to read and write into the local memory is already included in the ECO execution cost. We define a task for sending the data to other processor. As explained in section 3.1.2, the cost of this process is equal to the time taken to transfer the complete data from the memory location to the FPGA memory. At the other end, we schedule enough slack time to ensure transfer of the data packet from the FPGA memory. Hence in order to schedule for the communication between two processors, we schedule the send task after the parent ECO is executed. At the receiver, we schedule the child task after adding the the communication delay to the completion time of the send task.

Tolerance for k Processor Failures

In order to make the system tolerant of k processor failures, we need to replicate a single task k+1 times, each on different processors. Hence every list must be replicated k+1 times. Scheduling a task is considered to be complete only after all its replicates are scheduled. A task is scheduled only after all the replicates of its parent tasks are scheduled. The communication between these replicates should be minimum and should ensure that the load at the communication interface is the minimum possible.

Generating the Schedule Table.

The algorithm should generate a global schedule table that has a list of tasks that a processor should execute for its given rank among the other processors. The schedule table contains the schedule of the tasks that should be followed by the processor. The algorithm should also keep track of the distributed data channels. The data-channel between two ECO's is termed as a distributed data-channel, if both the ECO's are allocated on separate processors. Such a data channel must be updated at the processor on which child ECO is allocated as soon as the parent ECO has finished execution. The data-channel is updated by transferring the output of the parent ECO to the child over the underlying network. A distributed data channel is characterized by the source and the destination task. If we say E_{AB} is a distributed data-channel, then E_{AB} represents the communication between all instances (including replicates) of the task A and B. The distributed data channel also contains the list of the send tasks and the receive tasks. We shall see in further sections, how the replicas communicate between each other on the event of failure. For now, we shall talk about scheduling the communication tasks.

Step 3: Let k the number of processor faults that the system should be tolerant of.
Initialize for this step $seq = D$ dominant sequence; where D is $m\_seq(0)$
St be the stack for sequences
Let l be the first list of *seq* scheduled on the $p_t = 0$ of *seq*.
Step 4:

a) List l = first list on the sequence *seq*

b) If List l has all its sources scheduled or if it does not have any source task, then goto step 4d

c) If List l has any of its source ECO not scheduled yet, then push the tuple (current sequence *seq*, current list l) into the *St*. Get the unscheduled source sequence and set seq as the source and goto step 4a

d) Schedule the k+1 replicates of the list such that every replicate satisfies the precedence constraints. Apart from this, the replicates should be scheduled after taking the communication cost if any into consideration. The strategy we use here for allocating the replicates is explained step by step below.

   1. For every processor, in which a replicate of the list is not allocated yet, we

      (a) Find the earliest possible time at which the replicate could be scheduled. Computation of this earliest time should be done only after including the communication cost. If the processor under consideration, does not have any of its parent list scheduled, then a communication receive task is scheduled for each source executing remotely. A communication task required to send the data is also scheduled on the processors with the source. These communication tasks are scheduled such that the receive task at the processor begins as soon as the send task at the remote processor finishes. This communication schedule ensures that the communication interface never overloads.

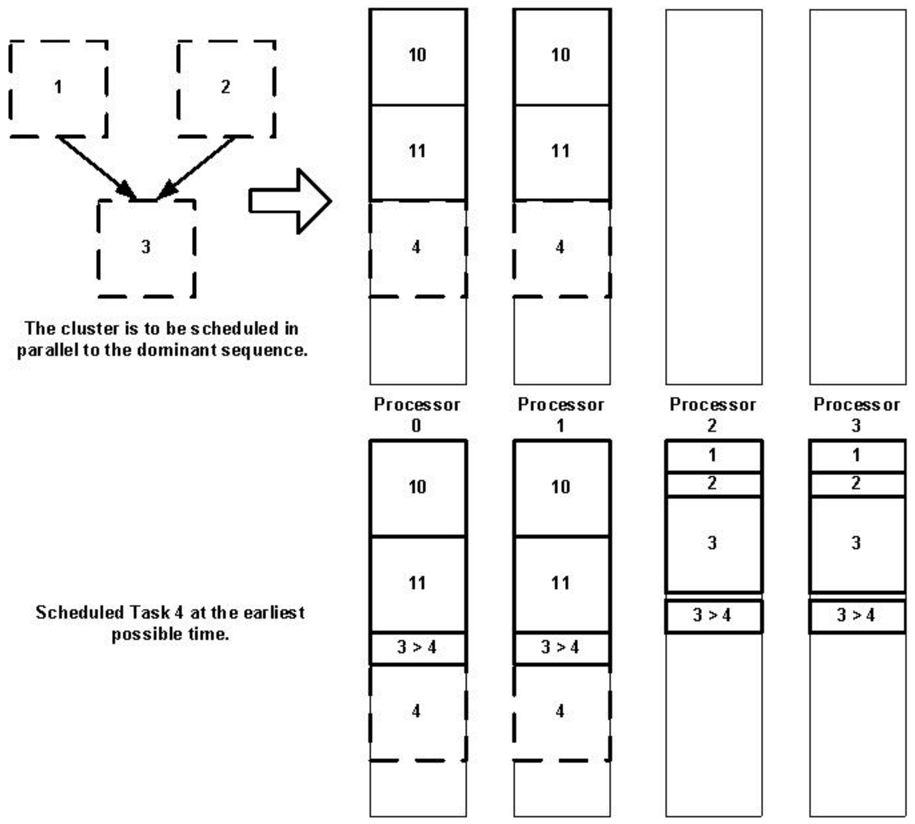

1
2
3
The cluster is to be scheduled in parallel to the dominant sequence.
Scheduled Task 4 at the earliest possible time.
10
11
4
Processor 0
Processor 1
Processor 2
Processor 3
3 > 4

Figure 3.2: The partial schedule generated by the algorithm. This shows the way parallelism is achieved.

(b) Find the processor with the earliest time from the schedule time that were found in the last step and schedule the list on that processor and eliminate the processor from future scheduling considerations for the task.

e) If there are any more list lst in the sequence seq, set $l = lst$ and goto step 4b. If there are no more lists in the sequence then pop an entry (sequence, list) from the stack. Set seq = the sequence and list l = list entry in the (sequence, list) tuple. Continue on step 4e. If the pop operation was not permitted because of stack underflow, we can infer that the algorithm has scheduled all the tasks.

Figure 3.2 shows how the partial schedule is generated with parallelism for the DAG that was shown in Figure 3.1. As shown the first list (10, 11) of the dominant sequence is selected first for scheduling. Since we have to mask a single processor failure, each task is replicated twice. Hence those tasks get replicated on the processor 0 and 1. The next list in the dominant sequence is (4). But to schedule (4), the cluster (1, 2, 3) has to be scheduled. The task 1 is scheduled first, then task 2 is scheduled and then task 3 is scheduled on processor 2 and 3. While finding the possible time for task 1, the earliest 2 times are present on the processors 2 and 3. Again for scheduling task 2, the earliest time is found on the processors 2 and 3. Task 3 is scheduled on processor 2 and 3 because its predecessor task is scheduled on the same processors and hence the schedule for the task 3 is earliest there. Now to schedule the task 4 the earliest time is found on the processor 0 and 1. This is after scheduling the communication tasks between replicates of the tasks 3 and 4. Thus we add on to the partial schedule and create a schedule table for each processor.

3.3.4 Schedule of Communication Tasks

Communication tasks are of two types. The one that is scheduled to transfer the data from the local data channel to the FPGA memory buffer is called as sender communication task. The second type is scheduled at the receiving processor. This task transfers the data packets from the FPGA memory to the local data channel and is called the receiver communication task. Hence communication between two tasks running on different processors involves scheduling of the sender task at the source processor and the receiver task at the receiving processor. These communication tasks should be scheduled such that there is minimum load on the DSP-FPGA interface. Hence the sender communication task and the receiver communication task should be scheduled such that the receiver task is scheduled just at the instant of the expected arrival time of the data from the sender task.

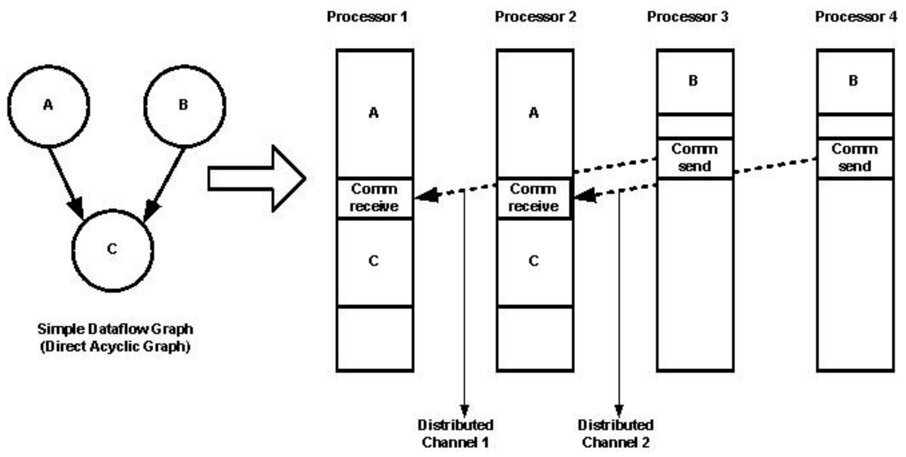


Figure 3.3: The communication tasks are scheduled for the distributed channel.

3.3.5 Distributed Data Channel Operations

To make the system fault-tolerant, we have to ensure that the destination tasks of each of the distributed data-channels gets the required input data even in the event of the failure of processors. However as explained earlier, the number of processor failures should not exceed the value of k. So in order to ensure a prompt delivery of input to all the dependent replicates, we need a scheme such that the destination tasks know whom to listen in the event of failure of processor executing its source. At the same time we should ensure that there is minimum communication load possible at the communication interface. Let us consider a simple example to see how this situation gets handled. Consider the dataflow graph that is shown in Figure 3.3. Task C is dependent on the output of the tasks A and B. In order to make the system tolerant for a single processor, we replicate each task twice as shown Figure 3.3. The 4 rectangles show respective Gantt chart on the schedule of the tasks on four processors. The replicates of Task A is scheduled on the processor rank 1(A0) and 2(A1). The replicates of the Task B is scheduled on the processor rank 3(B0) and 4(B1). As shown, the data channel between the tasks B0 and C0 as well as the one between B1 and C1 is distributed. Hence failure of processor 3 implies that the task C0 will now have to get its input from the task B1. Similarly failure of processor 4 will necessitate a distributed channel between the task C1 and B0. Thus the destination process has to take care that it receives the data from any of the source replicates.

3.3.6 Selection of Replicate

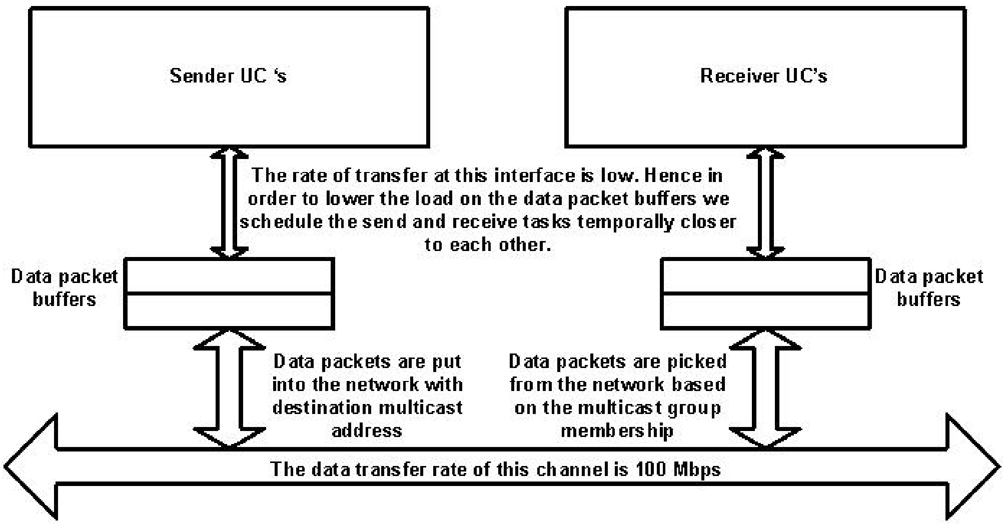
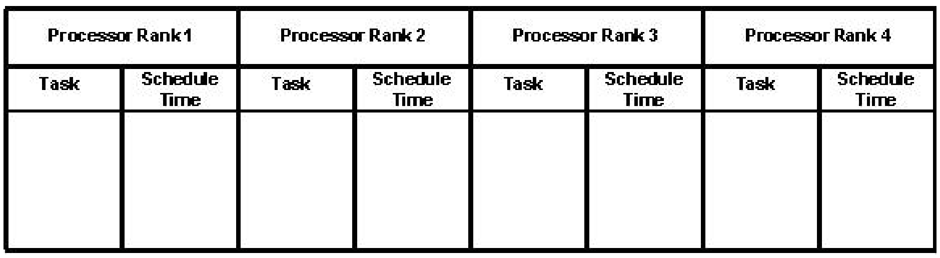

Sender UC 's
Receiver UC's
The rate of transfer at this interface is low. Hence in order to lower the load on the data packet buffers we schedule the send and receive tasks temporally closer to each other.
Data packet buffers
Data packet buffers
Data packets are put into the network with destination multicast address
Data packets are picked from the network based on the multicast group membership
The data transfer rate of this channel is 100 Mbps

Figure 3.4: The bandwidth difference at the communication interface.

In order to handle the situation, where the replicates of the source tasks have to communicate to all the replicates of the destination tasks , we need all the source tasks to send its outputs to all the dependent tasks. It would be very costly if all the the replicates of the source tasks send their outputs to the replicates of all the destination tasks. To use the communication channel effectively, we have a multicast based replica listening strategy. For every distributed channel, we assign the receive tasks a list of addresses that it should listen to. On failure notification, every node has to check whether the failed processor had any source tasks that it was listening to. Hence on every failure, the software on the processors have to select the multicast group of which it needs membership in order to execute normally such that it gets input from all the distributed data channels. To reduce the recovery operations and to make the failure of a node transparent to the local scheduler, we assign this task to the underlying network protocol. The underlying network would be notified at the startup, about the multicast group addresses of which it is a member. On failure notification the protocol stack changes its multicast group membership based on the processor that failed. This multicast based replication selection is to ensure the fastest arrival of the packets with least load on the communication interface. The underlying network protocol stack is implemented on FPGA. The multicast addresses and the group memberships are decided offline. The execution of this change in the group membership on failure could executed in a separate thread of execution. Thus the failure recovery operations as well as the time lost on such operations are minimized. Figure 3.4 gives the correct picture about the communication interface.

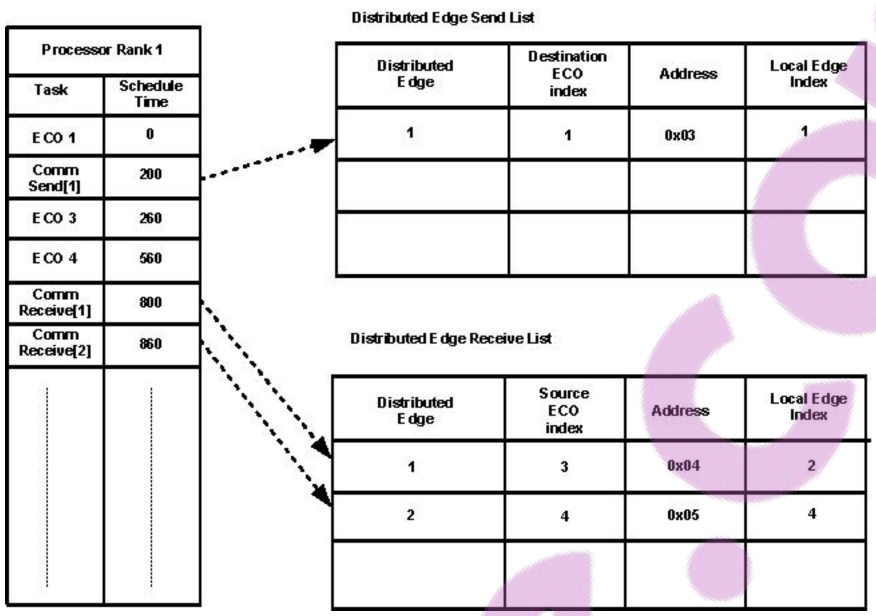
3.3.7 Global and Local Schedule



Processor Rank1		Processor Rank 2		Processor Rank 3		Processor Rank 4	
Task	Schedule Time	Task	Schedule Time	Task	Schedule Time	Task	Schedule Time

Figure 3.5: The System Schedule Table generated by the fault-tolerant task allocation algorithm

Figure 3.5 shows the system schedule table. As shown, this schedule table shows the static schedule that a processor should follow once it knows its rank. Figure 3.6 shows the local static schedule for the processor with rank 1. The local scheduler is invoked as soon as all the source data chan-



Processor Rank 1	
Task	Schedule Time
ECO 1	0
Comm Send[1]	200
ECO 3	260
ECO 4	560
Comm Receive[1]	800
Comm Receive[2]	860

Distributed Edge Send List

Distributed Edge	Destination ECO index	Address	Local Edge Index
1	1	0x03	1

Distributed Edge Receive List

Distributed Edge	Source ECO index	Address	Local Edge Index
1	3	0x04	2
2	4	0x05	4

Figure 3.6: The local schedule table for the processor rank number 1.

nels of the input driver ECO's are fed by the input data from the external hardware managers. The design of the local scheduler is simple. All it has to do is to execute the next task at the specified time according to the local schedule table. Here local schedule table is a part of the system schedule table. The startup tasks decides the rank of the processor. The rank of the processor remains the same as long as the system is deployed. The local schedule table is the schedule table that is present at the index that equals rank.

$$\text{Local schedule table} = \text{System schedule table[rank]}$$

In order to schedule a communication task, the scheduler refers to the distributed channel list that is also generated for each processor based on their ranks. Based on the entry in the distributed channel table, the communication task sends the data or receives the data from the source and destination. In the event of failure too, the local scheduler is supposed to carry out the monotonous task of picking up the next task in the table and executing it on time without any interrupt. Thus we have a simple design for local scheduler. On failure the underlying network protocol would do the necessary failure recovery operations and ensure timely arrival of inputs to the local memory.

3.3.8 Global Communication Table

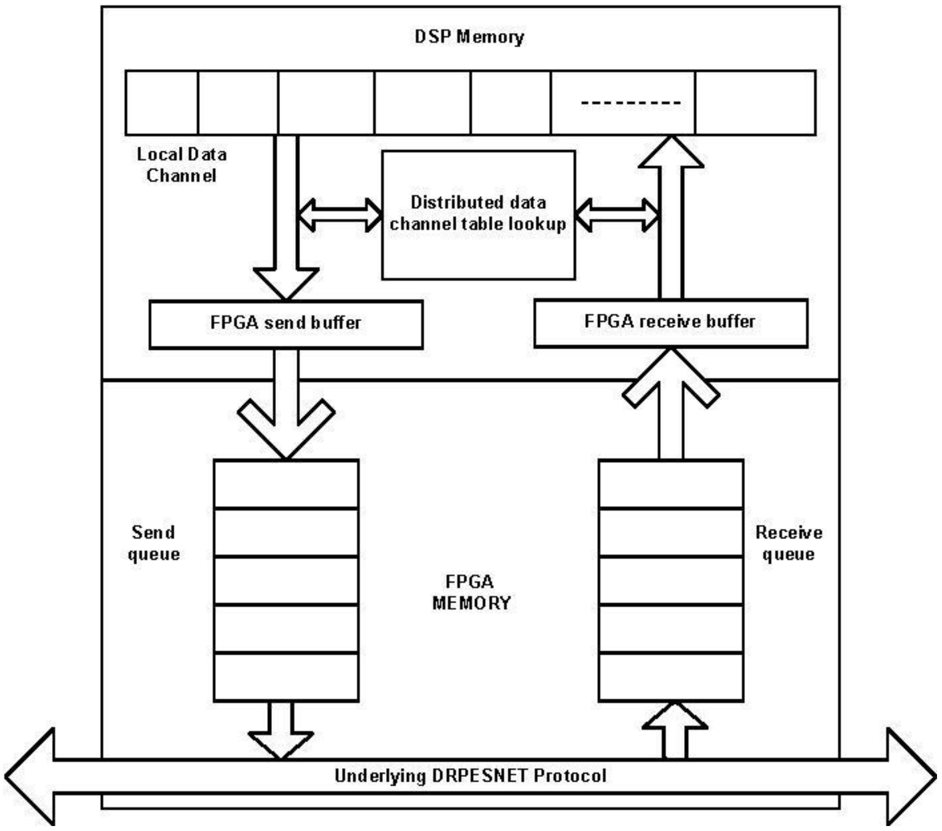


Figure 3.7: The statically scheduled communication operation based on the distributed edge table.

The communication task has two modes for functioning. A sender communication task is responsible for sending data from local data channel as per the information provided by the send distributed data channel table. The receiver communication task receives the data packet, interprets the communication data channel that has to be written and then transfers the data into the local data channel. For both the operations, the distributed channel lookup table is referred as shown in Figure 3.7. During the send operation, the DSP refers to the lookup table for address and the value of this address (which is the multicast address) remains the same. During the receive operation, the lookup operation is performed at the FPGA protocol stack. The value of the address field may change with the failure of other processors. Thus. the role of scheduler is just to ensure that the next task in the queue is executed in time.

3.3.9 Failure Recovery Operations

The failure notification from the underlying protocol is reliable. It is broadcasted within 2 complete network cycles delay. This failure notification is handled by the communication protocol stack. It updates the multicast group address that it has to listen now. The table required for this updation is written into the FPGA protocol stack during the startup. As explained earlier, the schedule of tasks and replicates on each processor are decided earlier. The failure of a processor could affect only those nodes that run tasks that are dependant on the tasks running on the failed processor. However, the schedule ensures that as long as the number of failed processors does not exceed the value of k, every dependent task would receive the output generated by its parent tasks as input. Another important aspect of this design is that the processor is never interrupted to do the recovery task.

3.3.10 Synchronization between the Universal Controllers

Since all the processors used are identical, the design assumes that the speed of the processors are the same. The local schedulers on all the processors starts scheduling tasks as soon as all the inputs for the input driver ECO's arrives on the DSP memory. Thus, the difference between the start of the schedulers on all the processors should be predictable and should be a value that is less than twice the network cycle delay. Every switching cycle starts on all the processors with a time difference among them that should be (2N-1) times single nodal delay where N is the number of nodes in the network. This delay is calculated considering the possible communication delay due to failure of a node in the ring other wise the delay would be N-1 times the single node delay. Since the network has a ring topology, the delay between two nodes remains the same. Considering the upper bound of the delay forms an appropriate assumption that will ensure synchronization among the execution of processors.

3.3.11 Features

The design explained above has the following features:

- The recovery work on failure of a processor is minimal and is limited to few read and writes of integers.
- The scheduler works on two levels. At the lowest level or the local scheduling level the scheduler design becomes the simplest possible, where the scheduler has to pick the next task in the queue and execute it at the right time.
- The fault-tolerant schedule ensures that there is minimum load on the DSP-FPGA interface.

3.3.12 Limitations

- There is no guarantee that the schedule is an optimal one.
- The system expects reliable transfer of packets from the underlying network protocol.
- The design cannot handle systems that involve multi threading and resource conflicts.
- We should also note that writing the data packets on the FPGA buffer does not actually start the transmission of data packets. The design is based on the assumption on the traffic in the communication protocol.

Chapter 4

Implementation

In this chapter, we shall get into the implementation details of the design explained in Chapter 3 . We shall first understand the offline scheduler implementation. Then we turn our attention to the modifications made in the DARK to integrate the design.

4.1 Fault-Tolerant Task Allocation

The offline scheduler results a fault-tolerant schedule to mask a specified number of processor failures. This offline scheduling algorithm is implemented in Java. The package DARKResourceAllocator contains the programs that execute the scheduling strategy. This implementation accepts an input file from the user. Based on the input values given in the input files, the scheduler outputs a schedule for each processor to follow. This global schedule is also called as the global schedule table. The program outputs a C program file *dist_config.c* that should be added into the Visual DSP project. The user has to compile and run the project for simulating one node. The local scheduler of the DARK refers to this table to schedule tasks on the DSP.

4.1.1 Input

A typical input file for the boost rectifier application is as shown in Figure 4.1.1. The input file contains the details about the dataflow graph. The input file primarily contains the execution cost of the ECO's and the data-channel details. The designer is expected to enter the following in the input file:

- the input ECO's
- the output driver ECO's

```
Input File Format

P = 4; // number of processors
F = 2; // number of processor failures to be masked
C = 300; // communication cost
d = 3; // nodal delay

// Execution cost of all the ECO's
{375,375,375,375,375,750,750,750,375,375,625,
        1250,1000,1000,3750,875,875,875};

{1,2,3,4} // source nodes
{15,16,17} // output nodes

// Datachannel description.
// <source node, destination node, number of packets>
{  {-1, 0, 1}, {0, 11, 1}, {-1, 1, 1}, .........}
```

Figure 4.1: Input File

- the data channel edges that defines the task precedence constraints as well as the communication details for each edge between 2 ECO's.

Based on these inputs, the software determines the task precedence constraints and keeps track of the data-channels that are needed for the communication between two processes. The input file thus provides the information about the dataflow structure. Apart from this the input file specifies :-

- the number of processors in the system.
- the number of non-Universal Controller nodes in the network.
- the number of processor faults that should be masked.
- the communication cost for a single packet.
- the network saturation value to be assumed.
- the nodal delay or the value of a single network tick.

4.1.2 Generated Output File

The output file generated is a C program file. This file is written after the task allocation algorithm is executed and the tables that were explained

Output File Format

```
Local_Schedule_Table_Entry table0[] =
{
        {.. }, {...} .. {...} // Schedule of tasks
}
Local_send_table0[] =
{
        {.. }, {...} .. {...} // Send Table records
}
Local_receive_table0[] =
{
        {.. }, {...} .. {...} // Receive Table records
}
// The above 3 tables are generated for each processors

Global_Send_Table global_send_table[] = { // global send table
        {6, 3, 0, 1, -1}, {}, {} .... //contains indices
        // used to refer the global communcation table
};
// global receive table
Global_Receive_Table global_receive_table[] = {
        {5, -1}, ..{},{} //contains indices
        // used to refer the global communcation table

int number_of_processors = 9;
// global communication table contains list of the
// multicast addresses that a processor is responsible
// for putting the data packet into the network.
int[7][3] Global_Comm_Table = {
        {254, 253, 252}, // processor rank 0
        {251, 250, 249},
        ....
        ....
};
```

Figure 4.2: Output File

in Chapter 3 are generated.This C program is compiled and built for all the processors. At the run time, every scheduler on the processor decides the table it has to follow. This decision is based on the rank the processor gets at the run time. On execution, the local microkernel figures out the part of the table that it is responsible to read from. Based on the processor rank the microkernel decides its section of interest in these tables and carries out the scheduling and configuration operations. Figure 4.1.2 shows a miniature version of the output file generated. The tables that are generated are the:

- schedule table
- distributed communication table
- multicast address table

4.1.3 Modeling the Problem

This section shows how the different aspects of the required solution is mapped into the implementation.

Dataflow Algorithm

The dataflow algorithms are represented by directed acyclic graphs. This graph is constructed based on the data channel details specified in the input file. The graph is a weighted graph. Each node has an execution cost of the corresponding ECO. The edges between them are denoted by the cost of the communication which is counted for now as the number of packet required for communication. This value could be easily converted into the amount of time required to transfer the data from one packet to another. The graph is stored in the form of an adjacency matrix and each edge is numbered as the number of packets that are to be sent from parent to the child ECO. However, if the parent and the child ECO are executed on the same processor, then the cost of the communication is zero.

Schedule for a Processor

A schedule for a processor is represented as an ordered list of tasks with a start time associated with each task. A task could be any of the following 3 types:

- ECO task
- Communication task
- Slack task

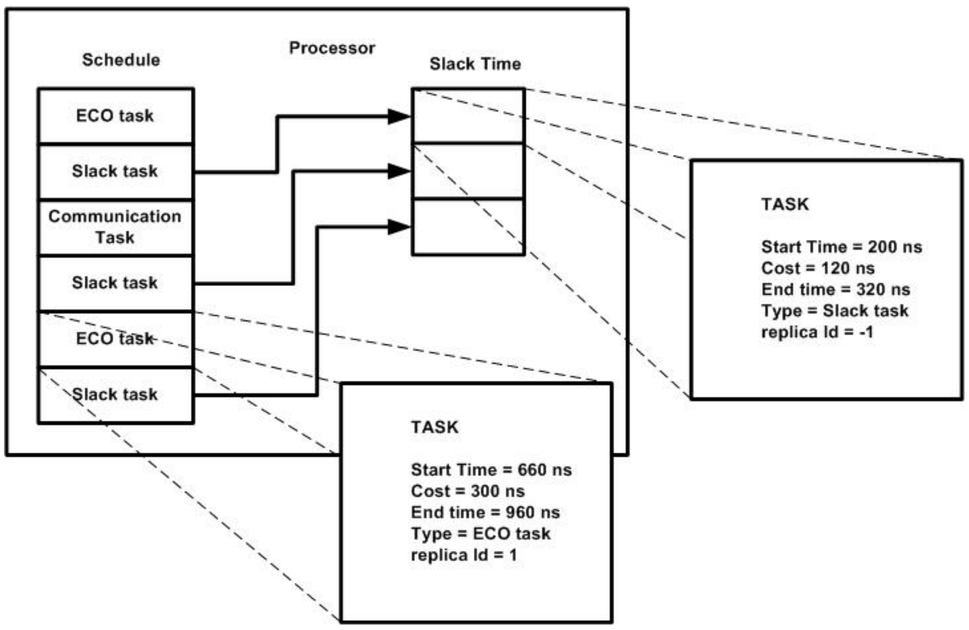


Figure 4.3: A partial schedule modelled in the implementation of processor.

An ECO task represents an ECO that is to be scheduled on the processor. A communication task represents the task that is responsible for either putting network packets into the network protocol stack or getting packets from the network protocol stack. This is typically scheduled when there is a need to send network packets from the local data channel to any remote processor. The slack time that would be present on the processor is also represented as a task. This is called as the slack task. The slack tasks is to mark possible slots on the processor that could be used by future tasks for getting scheduled on the processor. Figure 4.3 shows the implementation block diagram representing how a schedule is modelled on a processor. Every instance of the Processor object has 2 ordered lists. The first list keeps track of the schedule of the tasks on the processor. It contains a sequence of tasks that are scheduled on the processor. The second list contains all the slack tasks.

In order to schedule any task, the slack tasks are checked for availability for schedule. Figure 4.4 shows the pseudo code for scheduling a set of contiguous tasks. If the start time constraint and the cost of the task, to be scheduled, fill in the slot that is represented by a slack task, then the task gets scheduled at the specified start time. The slack task resizes itself to adjust to the newly added task. It also verifies its starting time and the cost to maintain the integrity of the schedule throughout the table. If the scheduled task had the same starting time and the cost as those of the slack task, then the slack task is eliminated. The integrity of the schedule here

```
public int scheduleTasksAfter(Task[] tasks, int fromIndex,
                                int count, int startTime)
{
   int cost = 0;
   cost = compute the total cost of the tasks

   // Get the slack task at the specified time
   // of the required cost.
   Task slackTask = this.getSlackTimeAfter(startTime, cost);

   if (slackTask != null) // if we get the slack time.
   {
      scheduleTime = startTime;
      int slackStartTime =  // start time of the slack task
      int slackEndTime = // end time of the slack task

      if (startTime > slackStartTime)
      {
          //  Create a slack task of
          //  cost = slackStartTime - startTime
          //  schedule the slack task and add it as
          //  slack time for future scheduling purpose
      }
      //  Remove the slack task
      this.slack.remove(slackTask);

      for (int i = fromIndex; i < fromIndex + count; i++)
      {
          // Schedule the task and update the start
          // time for each task
      }

      // is there any slack time after scheduling tasks.
      if (slackEndTime - time > 0)
      {
          // Create a slack task of
          //  cost = slackEndTime - time
          //  schedule the slack task and add it as
          //  slack time for future scheduling purpose
      }
   }
   return the value of the time the first task was scheduled.
}
```

Figure 4.4: Pseudo code for scheduling a task in a processor

means that every task must be scheduled at the starting time which equals the sum of the starting time of the previous task in the list and the cost of that previous task.

Communication Between the Processors

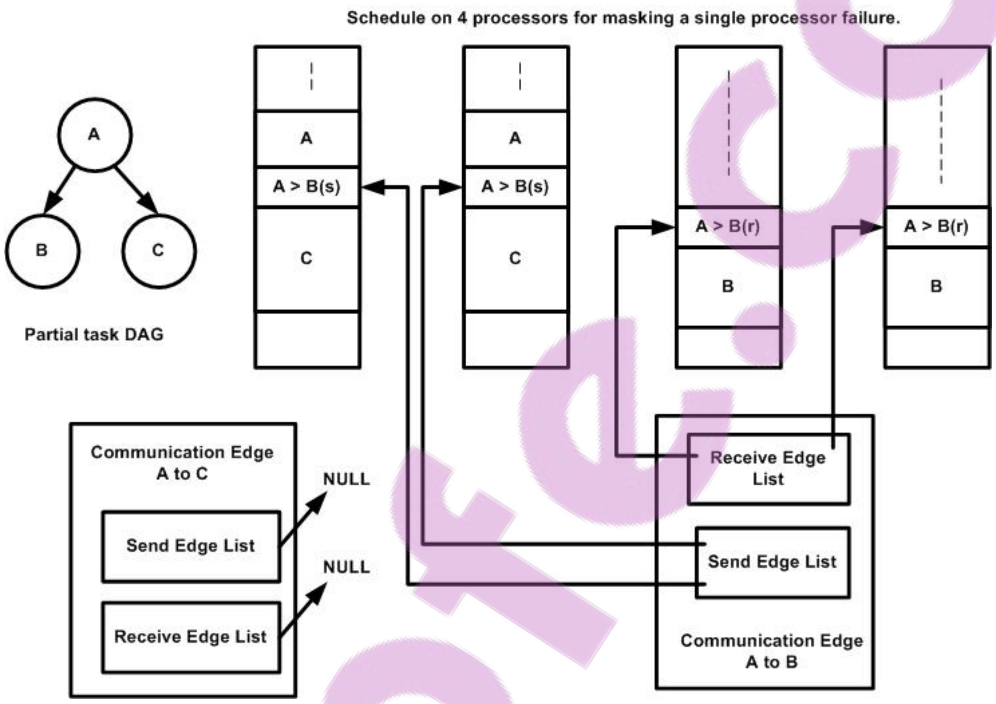


Figure 4.5: Communication Edge. As you can see the communication edge A to B keeps track of the send and receive tasks scheduled on the processors. The Communication edge A to C is not distributed.

```
public class CommunicationEdge
{
   private Task source;               // the source task
   private Task destination;          // the destination task
   private ArrayList sendEdgeList;    // list of send tasks
   private ArrayList receiveEdgeList;// list of receive tasks
   private Processor[] processors;    // list of processors
   private int cost;                  // communication cost
   ....
```

After the clustering phase, we keep track of the possible edges of the graphs that could be a distributed one. An edge is called as distributed, if there is atleast one pair of replica (parent task, child task) , that gets scheduled on separate processors. While scheduling a communication task, we relate it to the communication edge of which it is a part of. If we have a communication edge E_{AB} defined between tasks A and B, any send communication task that is scheduled for sending the output of A to B is logged with the edge E_{AB}. At the same time, the receive communication task that is scheduled to receive input for the task B also becomes a part of the communication edge E_{AB}. Once the tasks get scheduled, each communication edge keeps track of the required communication tasks. Figure 4.5 shows how the partial task DAG gets scheduled on the four processors. Every task is scheduled twice on separate processors in order mask single processor failures. Thus we can that the communication edge between A and B is distributed and it contains the send and the receive tasks. However the edge between A and C is not a distributed edge and hence there is no communication tasks between A and C. Every distributed edge is used at the end of the scheduling to generate the global multicast address table.

4.1.4 Clustering Phase

As explained in Chapter 3, the algorithm first goes through a clustering phase. This phase groups the tasks to achieve correct execution granularity in different sections of the dataflow graph. This prevents the next step of the algorithm to schedule unnecessary communication tasks.

The first step of implementation of this phase is to find the dominant sequence in the DAG. From the input files, we get the list of the source ECOs as well as the output driver ECOs. We use the Dijkstras method to find the longest path between each source and destination. This path is the dominant sequence in the DAG. We remove the nodes in the dominant sequence and then again find the next sequence with the largest execution cost. We do so until all the nodes are a part of the sequence. On doing this we get the sequences out of the DAG in descending order of their execution weights. This ordering of sequences helps getting the correct order of tasks to schedule and achieve maximum parallelism. A sequence is then broke into a number of lists. A list contains contiguous nodes, such that only the first and the last node in the list is dependant on more than one source or has more than one dependant nodes that does not belong to the list. A Task object is implemented to encapsulate the implementation that needs the list details as well as the task precedence details for the scheduling phase. Now that we have seen how a task is scheduled on a processor, how the communication between two processors are recorded and how the data from the DAG is manipulated to get the correct order for scheduling tasks, let us understand the implementation of the scheduling phase.

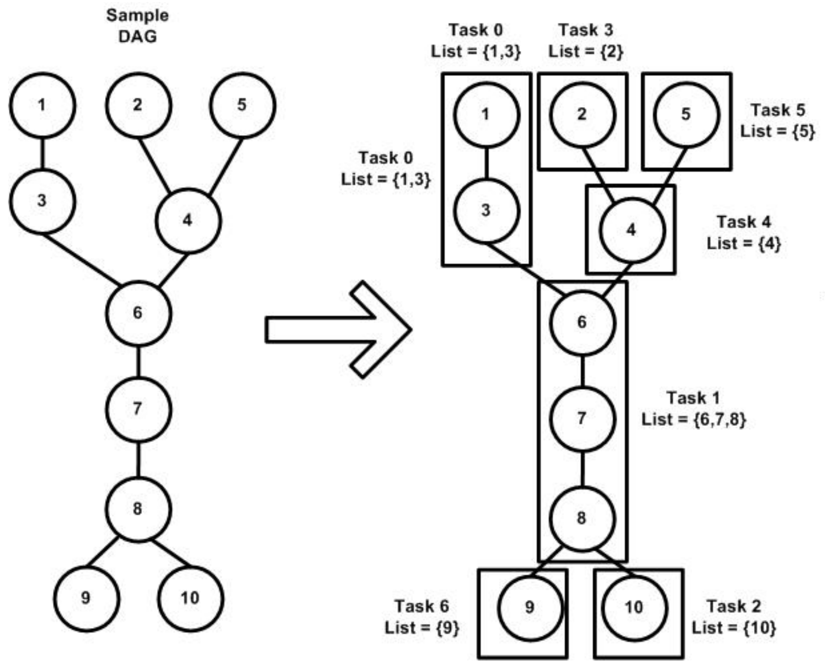


Figure 4.6: Mapping the DAG into a task flow graph. As we can see how the tasks are created out of the lists that were picked from sequences ordered in descending order of their execution cost.

4.1.5 Scheduling Phase

In this section we shall understand the implementation of the scheduling phase. Let us go step by step to understand the implementation.

Scheduling Tasks

The scheduling of tasks begin by traversal through the dominant sequence of the DAG. In the clustering phase, we transform the DAG to a graph such that a list is considered as a single task. The clustering phase does a significant job of ordering the tasks for scheduling at a time. This ordering is helpful especially in situations when more than one task is eligible for scheduling. The clustering phase ensures that the longest sequence is scheduled first. This helps in achieving better parallelism. Figure 4.1.5 shows the pseudo code that selects the task that is eligible for scheduling at every step. Scheduling a task here means scheduling the replicates of the task also. At every step, it checks whether all the parent tasks is already scheduled. If they are not, it selects each of the parent task for scheduling. Again these gets checked for precedence condition, that is whether all their respective parent tasks are already scheduled. Hence we need a stack that keeps track of the previous sequences that were getting scheduled. Since the

```
public void scheduleWithFaultTolerance()
{
   Stack sequenceStack = new ArrayList();
   Sequence s =  dominant sequence
   Task task = s.getTasks()[0];
   while (all the tasks are not scheduled)
   {
      Task[] sources = task.getSourceTasks(); // get source tasks
      // seq = get the sequence of the first source
               task that is not scheduled yet
      if (seq == null)
      // All the parent tasks are scheduled or
      // there are no parent tasks
      {
         // schedule the task with replicates
         Task[] dependents = task.getDependentTasks();
         // get the next task in sequence s
         // if such a task does not exists
         {
             // pop the sequence from the stack
             // if there is a stack underflow
             // all the tasks are scheduled
         }
       } //end if seq == null
       else // if all the sources are not scheduled
       {
          // Push the current sequence and task into the stack.
          // Assign s as seq.
       }// end else seq == null
   }// end while
}
```

Figure 4.7: The pseudo code for deciding the order of selection of task for scheduling.

first entry into the stack is always the dominant sequence, we can say that if all the tasks in the dominant sequence is scheduled then all the tasks in the DAG is scheduled. However, there are cases where sequences branch out from the dominant sequence and end at a different destination ECO. These sequences get scheduled after the dominant sequence gets scheduled. The resultant schedule is still correct. This also helps achieving parallelism.

Scheduling a Task for Fault Tolerance

Section 4.1.5 explains how the order of tasks that are to be scheduled is decided. In this section we see how the tasks are scheduled with fault tolerance and parallelism. Figure 4.1.5 shows the pseudo code that is used to schedule replicates of a task. The code keeps adding to the partial schedule generated at the intermediate steps of the algorithm. An intermediate step in the algorithm has a state of the partial schedule that is used by the future tasks to get scheduled on the processors. Scheduling every task changes the state of the schedule. This change would affect the tasks that were scheduled in the past and hence we need a validation of schedule that ensures that the tasks are scheduled at the earliest possible time. The subsequent sections explains this state of schedule and the validation operations.

Scheduling Replicates

As explained in Chapter 3, in order to make the system tolerant of k processor failures we need to replicate each task k+1 times. Once a task is selected for scheduling, we find the earliest k+1 scheduling time for the task possible on the processors and schedule the replicates on the processors. In case of tasks that do not have source task, we schedule it at the earliest possible scheduling time on the processor. However, for tasks that are dependent on the execution of other tasks, the earliest time for a task on the processor is given by the instances of CommunicationEdge object that is defined such that the current task is their destination. A task could be scheduled on a processor only after the end of execution time of all its parent tasks and the communication tasks (if any) between the parent tasks and itself. Thus at every processor the schedule time is found based on the aforesaid constraint. Then we select the first k+1 schedule times to schedule the k+1 replicates of the task.

Scheduling for Parallelism

In order to achieve parallelism, the partial schedule state that is generated every step after scheduling a task, schedules the communication tasks among the processors for future use. This enables the future tasks that get scheduled parallely to get scheduled at the earliest possible time. Figure 4.9 shows

```
private void scheduleWithReplicates(Task task)
{
   this.scheduledTasks.add(task);
   Task[] sources = task.getSourceTasks();
   Task[] dependents = task.getDependentTasks();
   // Schedule k + 1 replicates for k processor fault tolerance
   while (scheduledReplicatesCount < this.faultToleranceNumber + 1)
   {
      // Find the possible schedule time on all processors
      // on which any replica of the task is not scheduled yet

      // If the task does not have any source task,
      // schedule it as early as possible.
      // else on each of the processor the schedule time should be
      // after the completion of all the parent tasks.

      for (int i = 0; i < dependents.length; i++)
      {
        // Schedule the send tasks on the processor after the
        // scheduled task and record it with the communication
        // edge (task , dependent[i])
      }

      // add the scheduled processor to the list of
      // already scheduled processors
      scheduledReplicatesCount++;
   }

   // for each communication edge between the task and dependents
   // schedule a receive tasks on the processors
   // the task is not scheduled

   // if the task has parent tasks the validate the partial
   // schedule that has been made until now.
}
```

Figure 4.8: Pseudo code for scheduling the replicates on the processors.

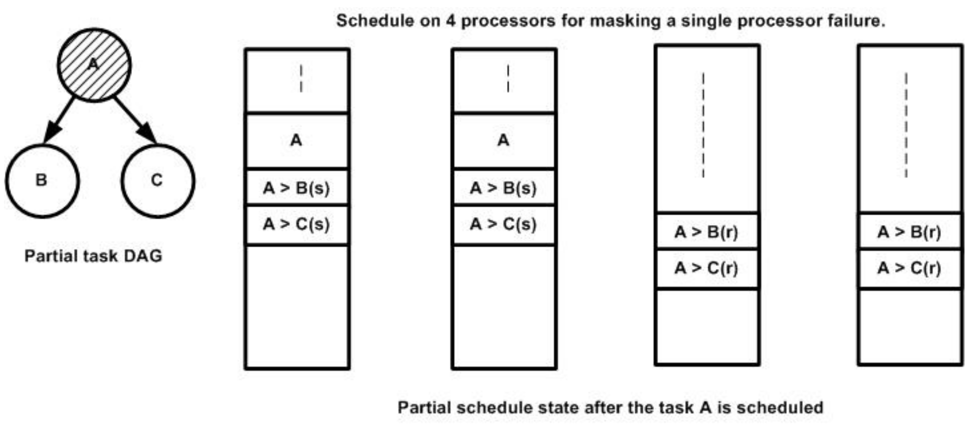

Schedule on 4 processors for masking a single processor failure.
A
B
C
Partial task DAG
A
A > B(s)
A > C(s)
A
A > B(s)
A > C(s)
A > B(r)
A > C(r)
A > B(r)
A > C(r)
Partial schedule state after the task A is scheduled

Figure 4.9: A partial schedule as soon as the task A is scheduled on the processors.

the partial schedule state after the task A gets scheduled. On the processors, where the task A is scheduled, after the completion of the task A, two send communication tasks - one from A to B and another from A to C, is scheduled. On all other processors corresponding receive tasks are scheduled at the end time of their corresponding send tasks. Hence in future the task B or C could be scheduled with an earliest possible time on the processors where the receive tasks are scheduled.

Validation of the Partial Schedule

Not all communication tasks that are scheduled for achieving parallelism proves to be useful. In some cases, when a task is scheduled, certain communication tasks becomes unwanted. This normally happens, when the communication edge between two tasks is found to be not a distributed edge or in cases where the receive tasks are scheduled on the processors where the destination task is not scheduled. Such unwanted tasks are to be eliminated from the global schedule. This neccessitated a validation function. Figure 4.1.5 shows the implementation of the validation function.

The validation function primarily reschedules all the tasks that are already scheduled. Once all the replicates of a task are scheduled, all the communication edges of which the task is a destination is enlisted. For each of those communication edges, the communication tasks are checked for validity. All unwanted communication tasks are first removed from the partial schedule generated until that point. Then all the tasks are rescheduled to check whether the task could be scheduled earlier such that it satisfies the precedence constraints. The partial schedule shown in Figure 4.9 is transformed into the one shown in Figure 4.11. The communication edge between A and C is not a distributed edge. The send tasks and receive tasks from

```
private void validateSchedule()
{
   //  For each task that is scheduled.
   for (int i = 0; i < this.scheduledTasks.size(); i++)
   {
      Task task = (Task)this.scheduledTasks.get(i);
      sources = //  source tasks.
      replicates = // replicates of the task.
      outputEdges = //communication edges with task as source
      // For each replicate of the task
      for (int j = 0; j < replicates.length; j++)
      {
        //Get the processor where the replicate is scheduled
        // If the task has no source tasks
        if (sources.length == 0)
        {
           // schedule the task as early as possible
        }
        else
        {
           // Schedule the replicate at the earliest time
           // after the end time of all its predecessor tasks
        }
      }
      // For each communication edge
      for (int j = 0; j < outputEdges.length; j++)
      {
         // Get the send and receive tasks
         for (int z = 0; z < sendTasks.length; z++)
         {
             //Schedule the send task at the earliest
             //time on the processor
         }
         int endTime = // the last end time of the send task
         for (int z = 0; z < receiveTasks.length; z++)
         {
            // Based on the last end time of the send tasks,
            // schedule the receive tasks
         }
      }
   }
}
```

Figure 4.10: The pseudo code for validating a partial schedule

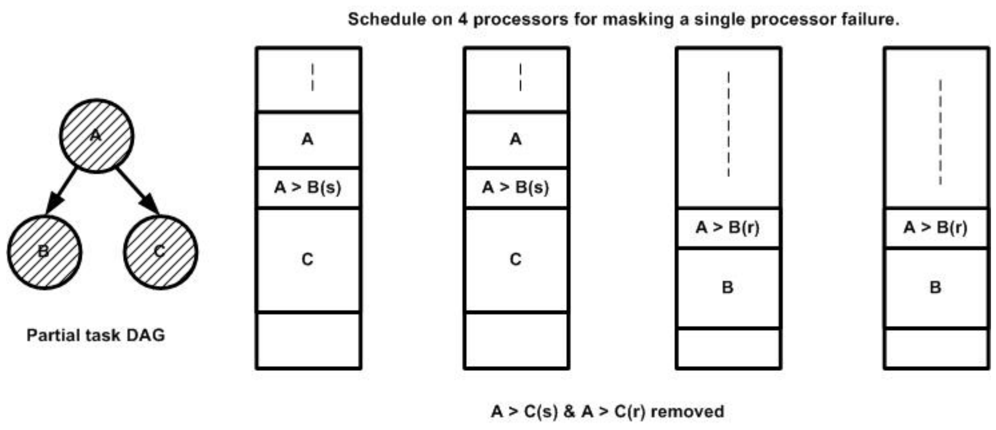

Schedule on 4 processors for masking a single processor failure.
A
B
C
Partial task DAG
A
A > B(s)
C
A
A > B(s)
C
A > B(r)
B
A > B(r)
B
A > C(s) & A > C(r) removed

Figure 4.11: The schedule on validation after scheduling tasks B and C

A to C are removed as they proved to be useless. The tasks B and C are executed parallelly at the earliest possible time. Thus the validation function minimizes the time taken by the partial schedule state.

Schedule of Communication Tasks

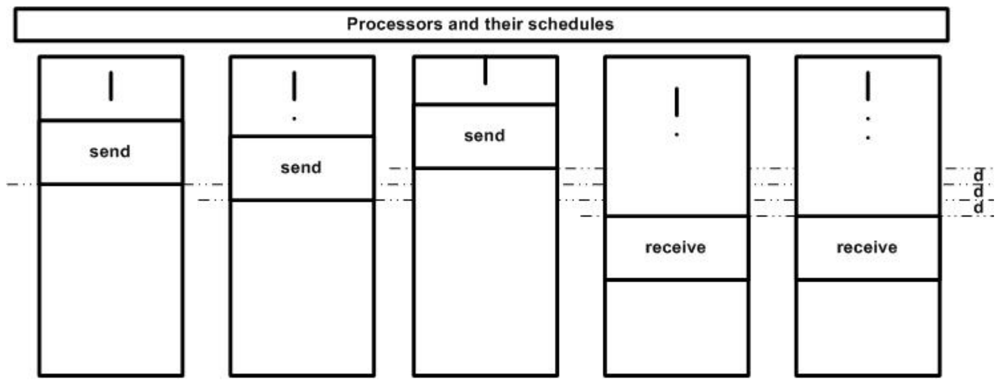


Figure 4.12: The schedule of communication tasks. Note the difference in the timings of the schedule of the send tasks among themselves. The receive tasks are scheduled at the time end time of last send + d

The communication tasks are of 2 types : send task and receive task. Until now when we talked about scheduling the send and receive tasks, we just considered that the receive tasks must be scheduled after the send tasks. However within the send tasks there is a timing constraint that has to be followed. In case of send tasks, every send task must be scheduled at least at a distance of d among themselves, where d is the time taken for a packet to traverse the 2 rings. This is because the receiver nodes could be notified with guarantee about any failure only after 2 complete traversal of the ring network. Hence, this is the delay that is needed for the failure notification to the receivers so that they can change their multicast group. All the receiver tasks are scheduled after summing up the end time value of of the last send task and the value of d. This is illustrated in Figure 4.12.

4.2 Local Scheduler

The local scheduler implementation of the DARK OS is modified to integrate the fault-tolerant offline scheduler design. The local scheduler now adopts the static schedule mode from the previous implementation. However it need not run all the ECO's. It runs the tasks that are assigned to it offline. This offline decision is made by the fault-tolerant real-time scheduling algorithm. The local scheduler first selects the schedule from the global schedule table. The design of the local scheduler is very simple. It first picks the local schedule table from the global schedule table. Then it simply has to decide whether the next task that is to be executed or not based on the time at that instant. As soon as it is time for the execution of the next task, the scheduler triggers that task for execution. Figure 4.13 shows an abstract implementation of the local scheduler.

Now the tasks that are to be executed are of different types. These are the ECO tasks, communication tasks, the configuration task and slack task. The type of the task is already specified in the local schedule table. The task that is picked from the table is checked for type and the corresponding function is executed. The structure of the table is as shown in Figure 4.13.

- In case of an ECO task, execute the ECO based on the index that is entered in the local schedule table.
- In case of a communication task, the scheduler checks whether a receive operation or a send operation is needed. From the schedule table, the scheduler gets the data channel index. Based on this value, the send and the receive operation is done on the correct data channel.
- In case of a configuration task:
  - The configuration task is scheduled offline based on the slack time present on the processors. The configuration task need to be run only if there are pending tasks for
    - Failure of a node.
    - Addition of a new node in the live network.
    - Configuration value snooped by the node.

```
#if (!MTHREADED && !DYNSCHD && FAULT_TOLERANT)
int index = 0;
for (index = 0; index < local_schedule_length; index++)
{

   TASK_TYPE type = local_schedule[index].type;
   if (type == TYPE_ECO)
   {
      // {get the current ECO index
      current_process =
             OS_All_Process[local_schedule[index].index];
      // execute the current process
   }
   else if (type == TYPE_SEND)
   {
      // Get the index of the data channel
      // invoke the packet sending module
   }
   else if (type == TYPE_RECEIVE)
   {
      // invoke the module that reads from FPGA
      // memory
   }
   else if (type == TYPE_CONFIG)
   {
       // call the configuration task
   }
   else if (type == TYPE_IDLE)
   {
        // do nothing for specified time
   }

#endif
}
```

Figure 4.13: Pseudo code for local scheduler

4.3 Communication Task

A communication task is implemented as a function that either reads or writes from the local memory and do the opposite on the FPGA memory. This task is implemented for the communication between two tasks on separate processors. As we have learnt earlier, a communication task is of 2 types : a send task and a receive task.

4.3.1 Send Communication Task

A send task is scheduled on the processor after the execution of the sender ECO. The local scheduler invokes send task with an index argument. This is used to find the data channel that the sender task has to read. Once the data channel is read, one or more data packets are created depending on the size of the data channel. The destination address of the data channel is taken from the multicast group table. The first byte tells us the datachannel index. The next byte tells the position of the data in the packet with respect to the data channel. The packet is then written to the FPGA memory. The packet is then put into the network after certain lower level processing at the protocol stack implemented in the FPGA.

4.3.2 Receive Communication Task

A receive communication task reads the data from the FPGA protocol stack. The implementation of the receive task is done by the existing module that reads from external memory and transfers the data into the local memory. The receive communication task first finds the data channel index that has to be written into. The next data it extracts is about the count of the data that has to be copied from the external FPGA memory to the local data channel.

Chapter 5

Underlying Network Protocol

When the system was centralized on a single Universal Controller, the configuration parameters of each node in the network were hard-coded. There was no provision for adding new nodes into the live network. The Universal Controller sends the pre-determined configuration parameters to each and every node. The new system is a decentralized fault-tolerant system. To enable addition of new nodes in the live network, the configuration of nodes on the network cannot be hard-coded. The underlying network must be modified to incorporate these new features. Our problem begins as soon as the system is started up. As the addresses and configuration parameters are no longer hard-coded, the nodes in the network should take the responsibilty to configure themselves based on some protocol among themselves. The protocol should react to the failure of a link or a node and start using the failure ring to mask the failure. The communication protocol must be extended to handle hot inclusion of a new node and handle its configuration operations. In the subsequent sections, we describe the solution for all these problems.

5.1 Startup and Configuration of the Network

When all the nodes in the network are started, there is a need for any one of these nodes in the network to start the process of network initialization and configuration. It is already proved in [42], that we need some information from each node for starting the network initialization process. We have the slot-id of a node as the available information at the startup. This slot-id is unique for each node. The configuration of a node is also decided based on the value of its slot-id.

The startup protocol has the values of these slot-ids as inputs for deciding the leader of the network as well as finding the configuration parameters.

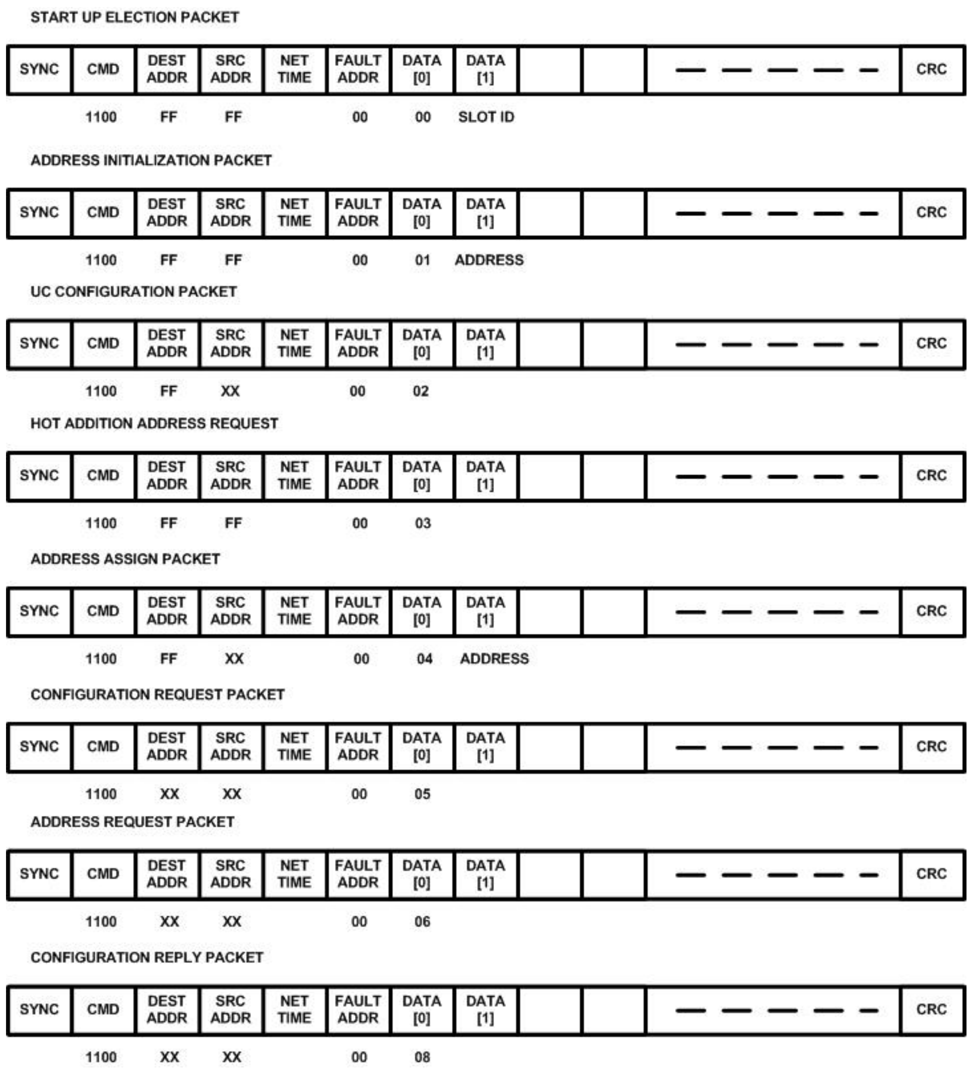

START UP ELECTION PACKET
SYNC | CMD | DEST ADDR | SRC ADDR | NET TIME | FAULT ADDR | DATA [0] | DATA [1] | CRC
1100 FF FF 00 00 SLOT ID
ADDRESS INITIALIZATION PACKET
SYNC | CMD | DEST ADDR | SRC ADDR | NET TIME | FAULT ADDR | DATA [0] | DATA [1] | CRC
1100 FF FF 00 01 ADDRESS
UC CONFIGURATION PACKET
SYNC | CMD | DEST ADDR | SRC ADDR | NET TIME | FAULT ADDR | DATA [0] | DATA [1] | CRC
1100 FF XX 00 02
HOT ADDITION ADDRESS REQUEST
SYNC | CMD | DEST ADDR | SRC ADDR | NET TIME | FAULT ADDR | DATA [0] | DATA [1] | CRC
1100 FF FF 00 03
ADDRESS ASSIGN PACKET
SYNC | CMD | DEST ADDR | SRC ADDR | NET TIME | FAULT ADDR | DATA [0] | DATA [1] | CRC
1100 FF XX 00 04 ADDRESS
CONFIGURATION REQUEST PACKET
SYNC | CMD | DEST ADDR | SRC ADDR | NET TIME | FAULT ADDR | DATA [0] | DATA [1] | CRC
1100 XX XX 00 05
ADDRESS REQUEST PACKET
SYNC | CMD | DEST ADDR | SRC ADDR | NET TIME | FAULT ADDR | DATA [0] | DATA [1] | CRC
1100 XX XX 00 06
CONFIGURATION REPLY PACKET
SYNC | CMD | DEST ADDR | SRC ADDR | NET TIME | FAULT ADDR | DATA [0] | DATA [1] | CRC
1100 XX XX 00 08

Figure 5.1: New Packets for configuration purpose.

The startup protocol necessitated using new types of network packets that are shown in figure 5.1. The configuration manager block added to the protocol stack is used for the network configuration purposes. It has the following functionalities:

- To send the startup packet. This functionality is needed in both the Universal Controller and Hardware Managers. In case if either of the 2 is added to a live network, then this packet serves the purpose of notifying the master node about self addition in to the network.
- It implements the election phase of the startup protocol to judge the leader of the network. It removes from the network any election packet with slot-ids greater than its own id that is sent by other UCs.
- In case of master node, this block sends packets in a periodic manner that look for newly added nodes. This functionality is needed only in Universal Controllers.
- In case of the newly added nodes, it responds to the packet described for requesting address from the master node.

5.1.1 Startup Protocol

Phase 1: Election Phase

In this phase all Universal Controllers run an election among themselves to elect a leader. After waiting for a predetermined time, (typically 1 sec) all the universal controllers put a packet with their own slot-id into the network. The Universal Controller with the lowest slot-id is selected as the leader for the network.

Phase 2: Network Initialization

Once elected as a leader, the Universal Controller sends a network address initializing packet that serially provides addresses to the nodes in the network. This initialization is done in 2 steps, In the first round trip initialization all the UC's get their addresses. In the next round trip initialization process all the other nodes are initialized. Once the nodes in the network are initialized, they request for the configuration from the master nodes. The configuration for a slave node is sent by the nearest Universal Controller who has the configuration parameters with it based on the slot-id. The Configuration packet that is sent by this universal controller is snooped by the remaining Universal controllers who update their information. Once all the nodes are configured, the master node signals all the nodes in the network by sending a network start packet. This marks the end of the startup process of the complete network.

5.1.2 Configuration Manager

Configuration manager is included in the current DRPESNET protocol as a peer to the other packet managers. However it has the least priority compared to the other packet managers. Figure 5.2 shows the state diagram of the functioning of the configuration manager. The following are the description for each state.

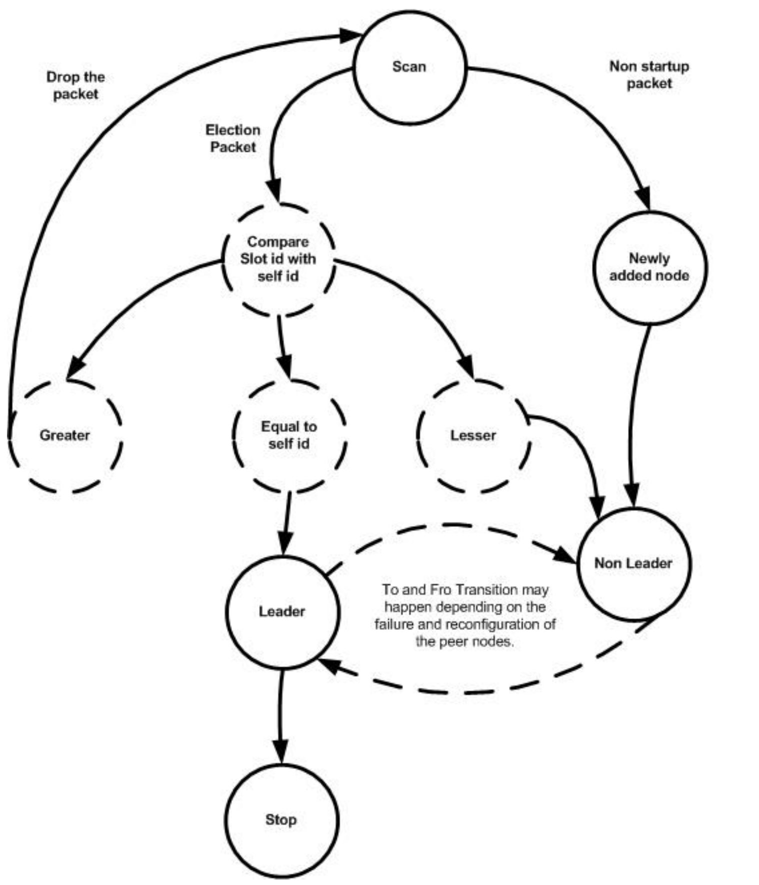

Scan
Drop the packet
Non startup packet
Election Packet
Compare Slot id with self id
Newly added node
Greater
Equal to self id
Lesser
Non Leader
Leader
To and Fro Transition may happen depending on the failure and reconfiguration of the peer nodes.
Stop

Figure 5.2: State Diagram of the Configuration Manager

States

State 1 : Start
It represents the state when the node is powered on. It initializes a timer to count 1 sec and transitions to the Scan state.
State 2 : Scan
It represents the state where the node keeps listening to the network. The future transition is based on the packet that it reads from the network. If the packet is found to be a non configuration packet that is used for normal

network operation, the node realizes that it is a newly added node to the live network. Hence it abstains from doing any startup protocol operations and stops the timer and moves to the State 4. In cases when the node realizes that the network protocol is at its startup phase, it scans each configuration packet and makes decisions based on the packet. If it receives an election packet it moves to the Comparison state.
State 3 : Comparison

a) In the comparison state, the node checks the slot-id with its own slot-id. If the slot-id is greater than the same of its own, it removes the packet from the ring. In case if the node has not sent its own election packet, it puts its own slot-id packet now and disables the timer.

b) If the slot-id is lesser than its own id then it forwards the packet. In case if the node has not put its own election packet into the network then it just disables the action because it realizes that it could not be the leader of the network and hence disables the timer.

c) When the slot-id is equal, this means that the election packet sent by self made a complete traversal throughout the ring. Since the packet was not removed from the network, the node has the smallest slot-id and it gets elected as the leader. The node then transitions itself to the State 5.

State 4 : Newly added node In this state, the node has to respond to the check packet that is sent periodically by the leader node. The node responds to the packet by sending a request for address. Once it receives the address assignment packet then it waits for the configuration parameters to be sent by the leader node. After getting the configuration parameters the configuration manager moves to the State 6.

State 5 : Leader In this state, the node has to periodically poll for new nodes that may get added to the live network. Hence at the end of the switching cycle it polls for any newly node added. It interfaces with the DSP and finds the configuration parameters for the newly added node. The leader that is elected at the startup is responsible for the initiation of the configuration of all nodes at the startup. Thus after getting elected as leader at the startup, the configuration manager initializes the ring at the startup. As you can see in the state diagram, there is a transition between the states leader and non-leader mode. This happens when one of the Universal Controller who was elected as leader fails. When the leader fails, the next potential Universal Controller gets elected as leader. This is when the configuration manager transitions its state of functioning from the non-leader state to the leader state.

State 6 : Non Leader Node The functionality in the non leader mode is minimal. In this state the configuration manager waits till the notification from the DSP to transition to the State 5 which is the Leader state. In this state, the configuration block snoops other configuration packets.

5.1.3 Network Initialization and Configuration Process at Startup

As explained before, the leader of the network is elected based on the slot-id. Once the leader is elected, it initiates the network addressing and configuration process. In the addressing phase, the Universal Controller first sends a broadcast packet that assigns addresses to the Universal Controllers only. Thus in this phase all the Universal Controller are informed about their ranks in the network. Then it assigns addresses to all the hardware managers.

In case of the hardware managers, the configuration is based on its slot-id. The hardware managers get their configuration parameters from a UC. A UC typically has the configuration information stored in them as a function of their slot-ids. For a Universal Controller, the only configuration parameter they require is their rank in the network. A hardware manager gets its configuration parameters from the nearest Universal Controller. The configuration parameter that is sent by this Universal Controller is snooped by the other Universal Controllers. At the startup, configuration is allowed for only one node at a time. This is done by the token packet that the leader puts into the network after assigning addresses to all the nodes. Once all the nodes in the network is initialized, the token is given back to the leader node and the leader node indicates the start of the switching cycle.

5.2 Failure Management

As explained before, the network consists of 2 rings. The first ring that is primarily used for communication between the nodes in the network is called the *primary ring*. The second ring is dedicated for fault tolerance purposes and is called *failure ring*. The failure ring runs in an opposite direction to the primary ring. Failure could be of two types node failure, where a node in the ring fails, and link failure where a link in the ring fails. This necessitates redirection of network packets from primary ring to failure ring and then back to primary ring as shown in figure 5.4.

This redirection process is not as straightforward as it seems. At the point of failure, where the packets are redirected from the primary ring to the failure ring, we have to take care that the actual number of packets that are lost due to the failure, are recovered. The count of packets that are lost at the primary ring are dependent on the type of failure. At the other

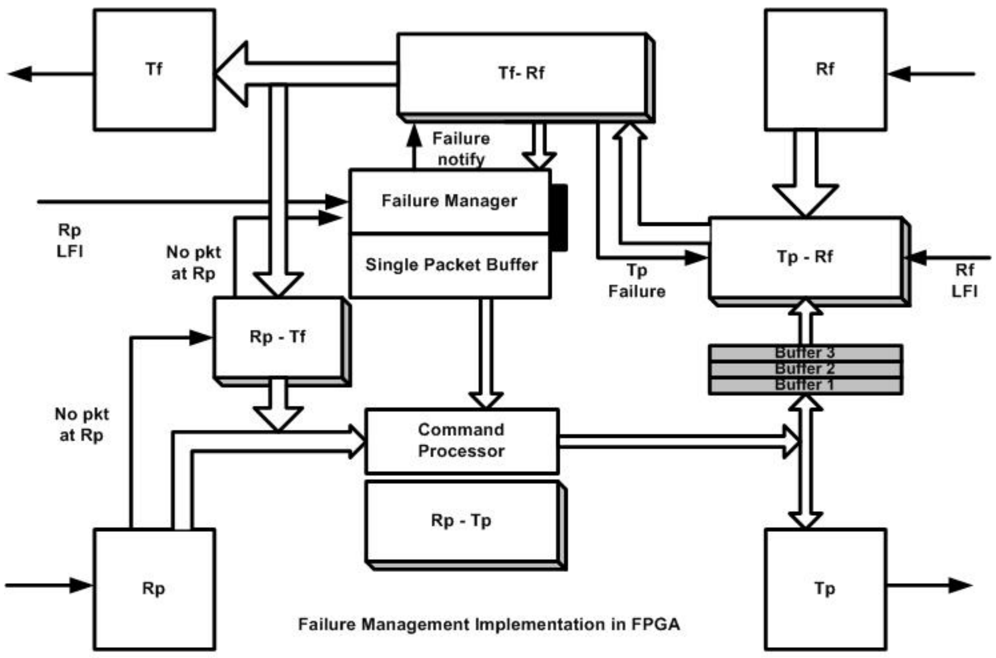


Figure 5.3: Failure Management in the DRPENSET protocol stack

end of the redirection, the packets get successfully redirected as long as the packet flow from the primary ring is ceased due to failure. As soon as the link or the node is restored, we have two streams of packets each from the primary ring and the secondary ring that are to be directed into a single stream which is the primary ring. Figure 5.3 shows the design for the failure management in the ring. In the subsequent sections, we shall understand the working of the different blocks of the diagram.

5.2.1 Tp-Rf

The Tp-Rf block redirects the packets from the failure ring to the primary ring. The redirection begins when the failure of the next node is notified by the Tf-Rf block. The other source of failure notification is the R_fLFI signal. If we get failure notification from either of the signals, we terminate the communication with the next node as if the next node has failed. If the failure notification is from the Tf-Rf block then it means that the packets sent through the transmitter link of the primary ring is not reaching the receiver link of the primary ring of the next node. If the signal LFI goes low we can infer that the next node has failed. In order to handle such cases, the Tp-Rf block has 3 packet buffers. These packet buffers keep a copy of the past 3 packets so that they can be redirected in case of faiure without any packet loss. Figure 5.6 shows the state diagram of the Tp-Rf block.

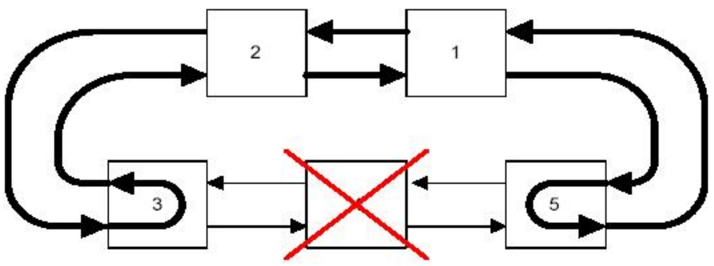
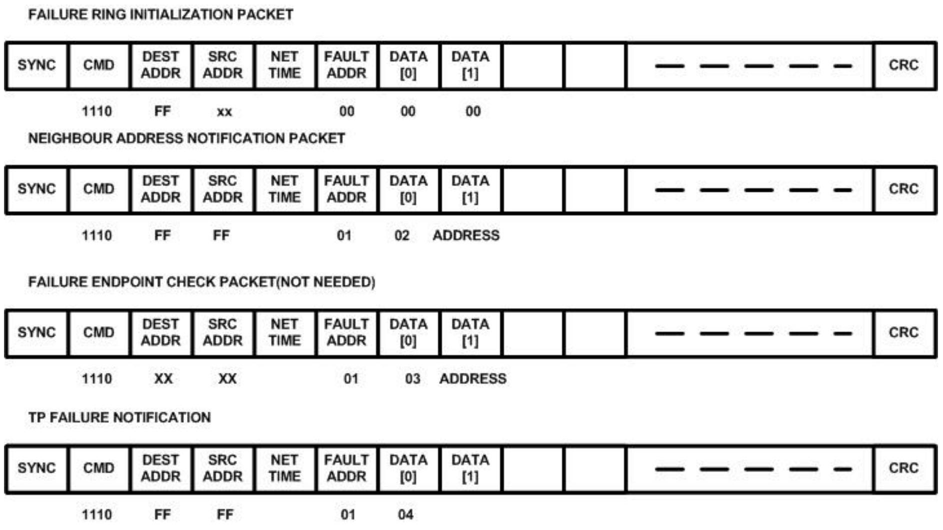


Figure 5.4: Redirection of packets between primary and failure ring



FAILURE RING INITIALIZATION PACKET

SYNC	CMD	DEST ADDR	SRC ADDR	NET TIME	FAULT ADDR	DATA [0]	DATA [1]			— — — — —	CRC
	1110	FF	xx		00	00	00				

NEIGHBOUR ADDRESS NOTIFICATION PACKET

SYNC	CMD	DEST ADDR	SRC ADDR	NET TIME	FAULT ADDR	DATA [0]	DATA [1]			— — — — —	CRC
	1110	FF	FF		01	02	ADDRESS				

FAILURE ENDPOINT CHECK PACKET(NOT NEEDED)

SYNC	CMD	DEST ADDR	SRC ADDR	NET TIME	FAULT ADDR	DATA [0]	DATA [1]			— — — — —	CRC
	1110	XX	XX		01	03	ADDRESS				

TP FAILURE NOTIFICATION

SYNC	CMD	DEST ADDR	SRC ADDR	NET TIME	FAULT ADDR	DATA [0]	DATA [1]			— — — — —	CRC
	1110	FF	FF		01	04					

Figure 5.5: Packet Structures for failure management

States

State 1 : No redirection
It represents the default state. In this state, at every network tick, a copy of every packet is inserted into the first buffer. The contents of the previous second buffer is moved to third buffer and the previous contents of the first buffer is moved to second buffer. The contents of the third buffer is discarded. When the LFI goes low, transition to state 2. If there is a failure notification from Tf-Rf go to state 3.
State 2 : Redirection from second buffer
In this state, the packets from the second buffer are directly routed into the Tf-Rf block. The packets are still forwarded the way they were in the previous state. When the LFI signal goes high, goto state 4.

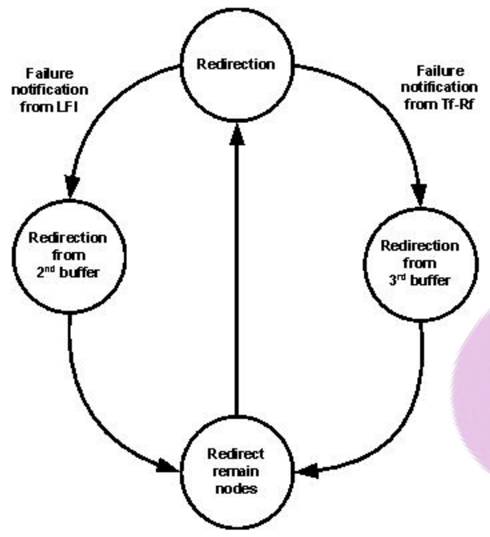

Redirection
Failure notification from LFI
Failure notification from Tf-Rf
Redirection from 2nd buffer
Redirection from 3rd buffer
Redirect remain nodes

Figure 5.6: State Diagram of TpRf block

State 3 : *Redirection from third buffer*
In this state, the packets from the third buffer are directly routed into the Tf-Rf block. The packets are still forwarded the way they were in the previous state. When the failure notifyng signal goes high then goto state 4.

State 4 : *Redirection of remaining nodes* In this state, the packet in the first buffer is marked and the previous routing to the Tf-Rf block is continued until the first packet is routed. The packets are forwarded among the buffers in the same way. Once the first packet marked on the arrival to the state is routed to the Tf-Rf block goto state 1.

5.2.2 Tf-Rf

The Tf-Rf block has similar functionality as that of the command processor [2]. However this block acts as an interface between the receiving and the transmission port of the node for failure ring. The block is also synchronized with the other modules such as Tp-Rf and Tf-Rp. This module is also responsible for routing the "wandering" packets on the failure ring back to the primary ring through the failure managers single buffer interface. We shall learn more about wandering packets in section 5.2.4. Let us have a detailed view on the functionality of the module based on the state diagram shown in figure 5.7.

The TfRf block gets initialized when the startup protocol initiates the network configuration. The working of the TfRf block is synchronized with the command processor that handles the packet flow at the primary ring.

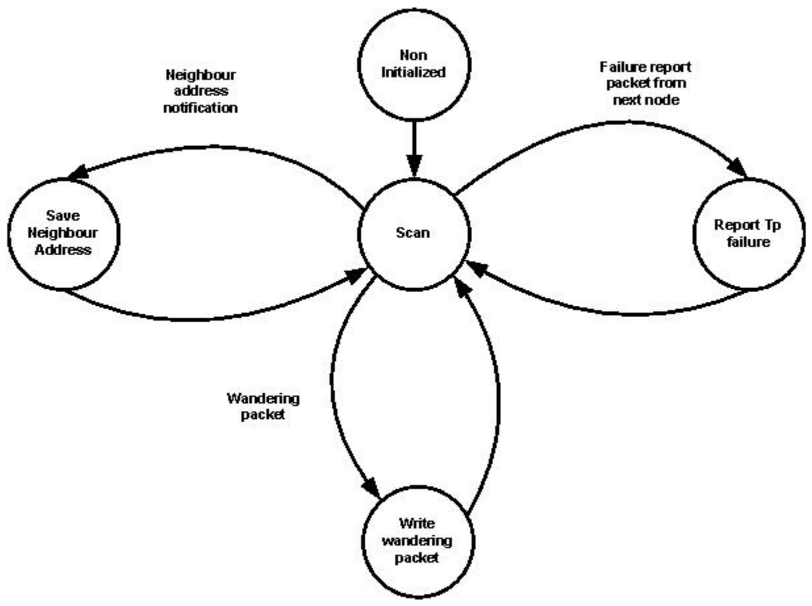


Figure 5.7: State Diagram of the Tf-Rf Block

Tf-Rf has similar functionalities as those of the command processor for the primary ring. As shown in the diagram 5.3, Tf- Rf dictates the packet transfer from the failure ring receiver port to the failure ring transmission port.
State1: Non Initialized:
This is the state of the failure ring before the startup protocol initializes the complete network ring. The elected leader node initializes the network. On this event, the state transitions to State 2.

State2: Scan:
This state represents the default working state of the block once it is initialized. In this state, the Tf-Rf forwards the packets to the blocks based on the packet types. All the transitions below are triggered as soon as the packet is received and the intermediate state transitions happen before the arrival of the next packet. On receiving the neighbor address notification packet, the working of the block transitions to the State 3. If the packet informs the failure of the primary ring transmission link, then the Tp-Rf block is signaled to start redirection of packets into the failure ring from the primary by moving to the state 4. When the packet is found to be a wandering packet then it transitions to the state 5.

State 3: Neighbor Address Record:
In this state, the Tf-Rf records the address of the adjacent node from which it gets the packet from the failure ring. Before the advent of the next packet the state reverts back to the scan state.

State 4: Report Primary Transmission Link Failure
The transmission link of the current node has failed and hence Tf-Rf signals this failure to the Rp-Tf block. Now the input is taken from the Rp-Tf block. The block now carries out the same functioning as it was in state 1, but with inputs from the Rp-Tf. Once the link with the next node is established, the state transitions to the state 1.
State 5 : Route Wandering Packet.
In this state, first the Tf-Rf makes an attempt to write the packet in the single packet failure buffer. In case if the buffer is already filled, the packet is forwarded to the next node and state of functioning is back to the state 2..

5.2.3 Rp-Tf

The Rp-Tf state is responsible to route the packets from the failure ring to the primary ring. The working of this state has only 2 states. The state of functioning toggles between these 2 state based on the value of the Rp_LFI signal.

States

State 1 : *No redirection*
When the LFI signal is high there is no failure in the previous node or link. Hence there is no need for any redirection of the outputs from the Tf-Rf block to the command processor block. When the LFI signal goes low, goto state 2.

State 2 : *Redirection*
In this state, the packets from the Tf-Rf block is redirected to the command processor block. When the LFI signal goes high, goto state 1

5.2.4 Single Failure Buffer

As mentioned before, the routing of the packets from the failure ring to the primary ring is not straightforward. We know that, on failure -

- at one end the packets are routed from the primary ring to the failure ring and
- at the other end, the packets are routed back from the failure ring to the primary ring.

The primary ring is established as soon as the failed component revives its correct working mode or gets replaced by a new subcomponent or gets

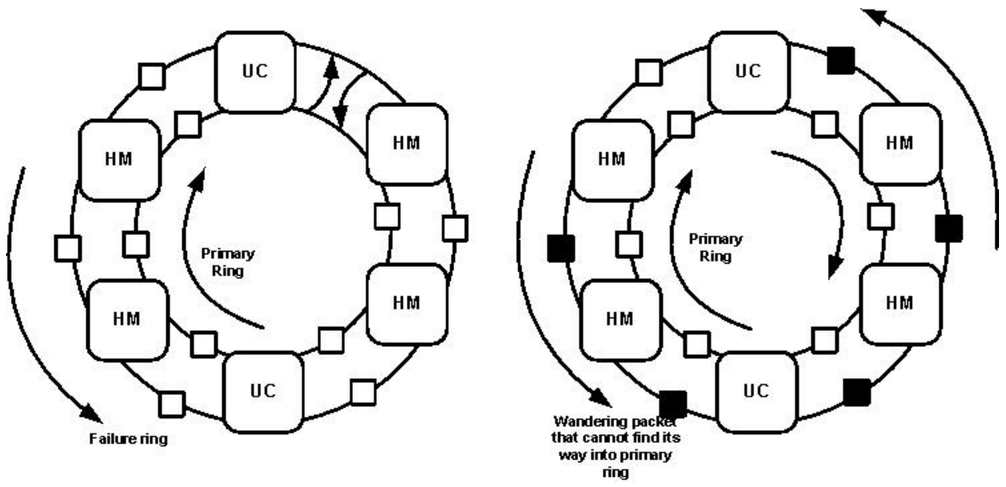


Figure 5.8: Wandering Packets on Failure Ring

removed from the network.. This transition period from the failure mode to the working mode of the network protocol is interesting. At the first end, the redirection of the packet has to be stopped. However, at the other end, simple stoppage of redirection is not enough. As shown in figure 5.8, the packets that are on the failure ring between the end points are yet to be routed to the primary ring. But the primary ring is now already holding and forwarding packets and has no scope for these packets to make way into the primary ring. These packets are termed as the *wandering packets.* The term "wandering" because these packets eventhough have a destination address but now wanders until it gets back into the primary ring. The wandering packet wanders on the failure ring till it finds an empty failure buffer that would redirect it into the network. This redirection is shown in figure 5.9.

Every node in the network has a single buffer. The data in this buffer is used when the network is in the healing phase from a failure. The role of this buffer is to hold a *wandering packet* . Wandering packets are the extra packets that are present in the failure ring that was not able to find its way back to the primary ring. As soon as a null packet is received on the network, the wandering packet replaces it in the network and the packet finds its destination on the primary ring.

5.3 Plug and Play Support

The new configuration functionality, when integrated with the failure management, provides enough support for the hot addition of new nodes in the live network. Thus the blocks that we described till now provide the infrastructure for the much required plug and play support. The configuration manager handles the configuration of the newly added node. The role of the

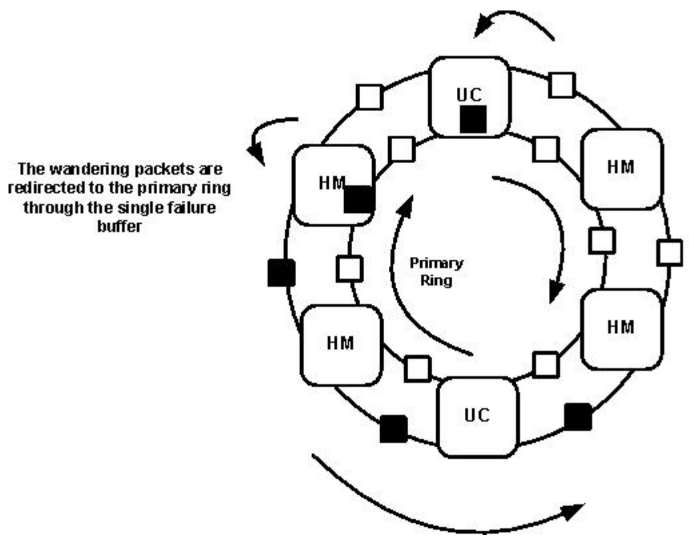

The wandering packets are redirected to the primary ring through the single failure buffer
UC
HM
HM
HM
HM
UC
Primary Ring

Figure 5.9: Redirection of the Wandering Packets using single failure buffer.

failure management is described in detail in the subsequent sections of this chapter.

5.3.1 Steps Needed to Add a Node

To add a new node in the ring, one has to break the ring first and then add the node to the ring by connecting the snapped ends to the newly added node. Since we have the adjacent nodes in the ring connected by two links (one for primary ring and other for failure ring), addition of a new node requires the user to break the 2 links between two adjacent nodes. Figure 5.10 shows an example of addition of a node into the live network. The newly added node would have 4 unconnected ports. The user has to ensure the correctness of the connections and see to it that the node has the ring and the failure ring formed by connecting the correct ports. But what we should observe here is that every new node addition is first seen by the ring protocol as a link failure. The protocol stays in the failure state with redirection happening at the adjacent nodes between which the connection was snapped. It remains in the same state until both the ring connection gets established by addition of the new node. This is where the failure management block comes into play that eliminates the possibility of packet loss due to the whole process.

5.3.2 Configuration of New Node

We already know the role of the configuration manager on addition of the new node from the discussion about the configuration manager in detail. The new node when added to the ring first expects itself to be added before

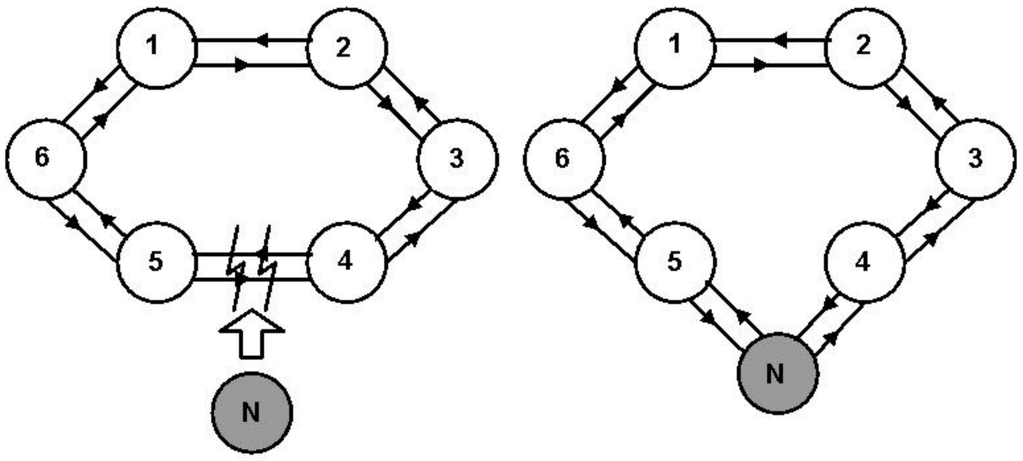


Figure 5.10: Addition of a node into the live network

the startup process. Hence it waits for pre-defined delay before it puts the packet into the network. However on forwarding few non-startup process packets, it “realizes” that its a newly added node. Hence it responds to the “scan” packet that the leader of the network sends periodically requesting for an address. Once it receives an address, the node sends a configuration request packet. The nearest Universal Controller sends the configuration parameters in response to the request packet. These configuration parameters are snooped by other Universal Controllers and the new node gets registered with its configuration parameters.

Thus the development of the failure management and the startup protocol together effected the plug and play support in the protocol.

Chapter 6

Evaluation

This chapter presents the evaluation of the new fault-tolerant design of the system. When only one Universal Controller executed the DARK OS in the complete system, the whole processing unit was simulated by a Visual DSP emulator. All the performance measurement was done using a Visual DSP emulator. But now we need an evaluation of the distributed system running on the DRPESNET network protocol. This requirement necessitated the development of multiple Visual DSP emulator based simulation system. The chapter first explains the architecture of this simulation system and explains how the evaluation strategy is implemented in the simulator. It also has the explanation of the parameters based on which we evaluate the system. We checked our new fault-tolerant system on the simulator for the Boost Rectifier application. We were able to find that the fault-tolerant design allowed running the system more efficiently than the previous version in certain cases. We also evaluated schedule generated by the scheduling algorithm and then tested the schedule by injecting failure in the system. In the last section, we observe the simulation signals of few important blocks in the protocol stack that were explained in the previous chapter.

6.1 Distributed DARK Simulator

The Simulator is based on multiple instances of the Visual DSP++ emulator with an ADSP 21160 debug session. Visual DSP++ provides a COM object with Automation API. This API could be used by external Windows application to interface with the software. In order to simulate a network with many Universal Controllers, we use this Automation API and interface it with our network simulator. The network simulator consists of two types of components, which are the network simulation server and the UC simulator driver. The UC simulation drivers are client programs that interface with the Visual DSP emulator. The simulation server program is a server program that keeps track of the outputs from different simulation clients.

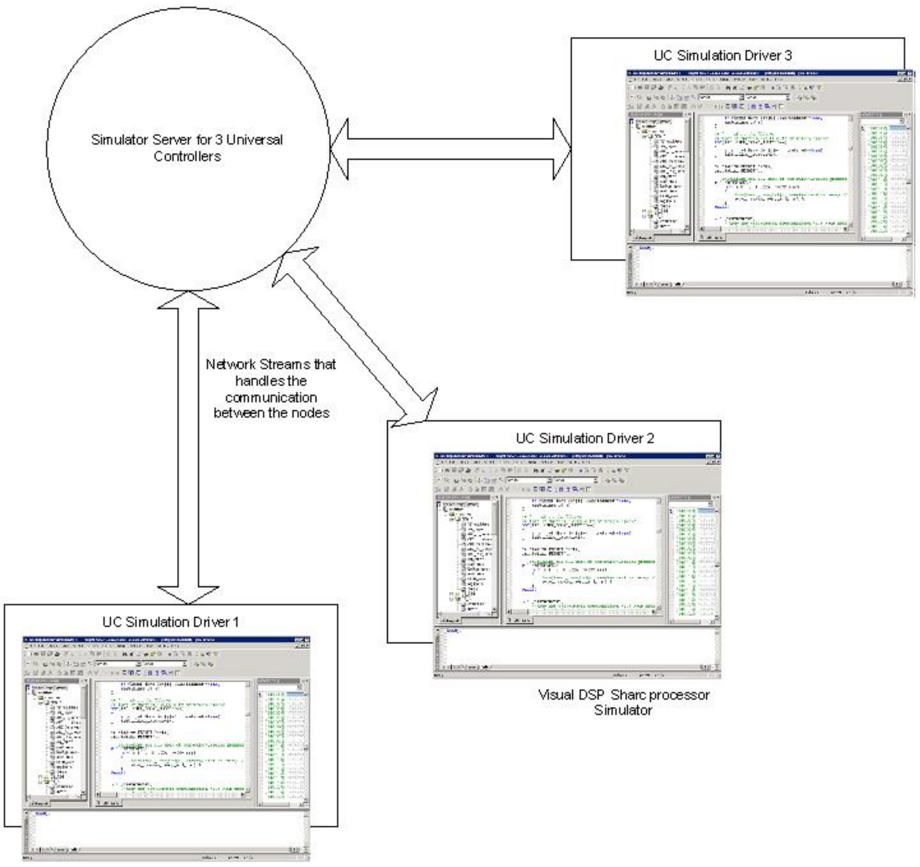

Simulator Server for 3 Universal Controllers
UC Simulation Driver 3
Network Streams that handles the communication between the nodes
UC Simulation Driver 2
UC Simulation Driver 1
Visual DSP Sharc processor Simulator

Figure 6.1: Architecture of the simulator

It knows the topology of the network and hence schedules the messages between the nodes.

6.1.1 DRPESNET protocol Simulation

Thus in the simulator, a Visual DSP emulator is mapped as a processor executing the DARK OS. The complete multiprocessor system is a C# implementation. This software simulates the following:

1. DRPESNET network protocol
2. Behavior of certain nodes like the hardware managers.
3. Failure in the network.
4. The DSP-FPGA interface operations

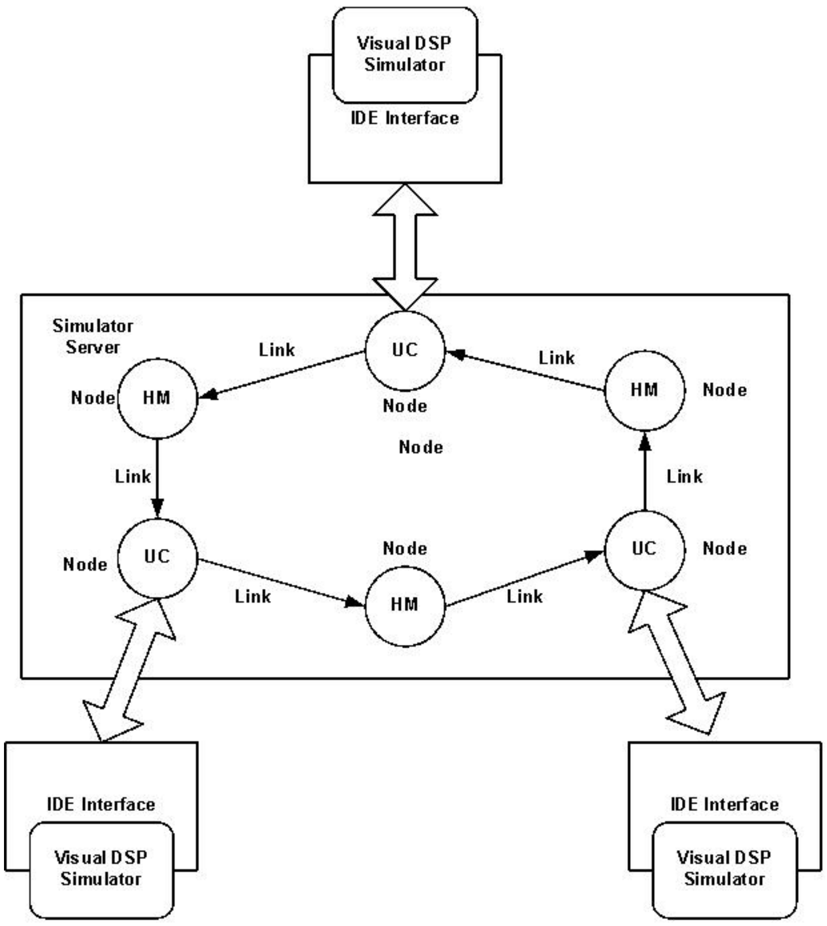


Figure 6.2: Implementation block diagram of the DARK simulator.

The simulator has a client-server architecture. At the server, a single process handles the synchronization and the message exchange between the nodes. This means that the server is responsible for the communication among the nodes and thus creates a single-threaded control flow among the different components of the system. A client is used to simulate the operations undertaken in a Universal Controller. The clients interface with the server and the Visual DSP emulator. The emulator runs the DARK code and the control application on the clients. Thus we have each client simulating a universal controller. The hardware managers are simulated inside the server process. The whole simulation architecture is shown in figure 6.2. In the subsequent sections, we shall see how the different parts of this simulator are implemented.

The implementation of the whole protocol is single threaded. Every packet in the ring moves between a single pair of nodes at a single network tick. Once the input message is given to each node in the network, every node in the network decides upon its output by the next network tick. This feature of the DRPESNET protocol enables a single threaded implementation. The time-triggered nature of the protocol also enables keeping track of the global time of the system. The protocol implementation internally updates the global time after every time-triggered packet transfer. The simulator uses this global time to trigger the failure of a node at the specified time as well as to halt the simulation once it runs for a duration over the stipulated time.

At every network tick, a node decides the output message that has to be transferred to the next node. This output message could be based on the transfer function that the node has. Typically the transfer function primarily has 2 functions. It either forwards the input message to its output port or it replaces the input packet with its own output packet or a null packet. Every node is provided with respective inputs. This input message is the output of the previous node that was generated in the previous network tick. Based on the input message, the node decides the output message that is to be sent to its output link. For all nodes, other than the Universal Controllers, the decision is made locally. Any decision that is made in the same process or execution thread that the server is running is called as a local decision. For a Universal Controller, processing a message implies sending the message to the client. At the client side, the message is read and the DSP-FPGA simulation block decides the output to be sent. These periodic messages helps synchronizing the global time recorded on both server as well as the clients.

6.1.2 Universal Controller Simulation

A Universal Controller simulation is integrated into the complete simulator using the Automation API of the Visual DSP. This API enables outside applications to interface with the simulator. The Automation API provides

COM objects that could be instantiated in the simulator and used for purposes like:

- Starting a session on the ADSP processor.
- Loading the required project in the simulator.
- Starting the code execution.
- Enabling and disabling the interrupts and watchpoints.
- Reading values from memory and DSP registers.
- Writing values into the DSP registers and local memory.
- Halt the execution of the processor.
- Perform some menu operations on the simulator IDE window and so on.

To simulate more than one Universal Controller, we run the DARK microkernel using Visual DSP on separate computers in a network. The simulator follows the protocol implementation and schedules the messages among the nodes in the network regardless of the type of node. The interface between the simulation of DARK execution in Visual DSP IDE and the DRPESNET protocol is implemented by implementing an IDEInterface object. This IDEInterface object is responsible for doing aforesaid listed tasks. The IDEInterface object is responsible for getting data packets from the local memory of the DSP as well as writing the data packets into the local memory.

The IDEInterface acts as an interface between the client process and the Visual DSP. This client process is executed in synchronization with the DRPESNET protocol implementation in the server process. The synchronization is achieved by an socket connection between the server and client process. Every message transfer between the client and the server process is done at the processMessage call done in the Universal Controller implementation in the server process. Thus the global time is kept consistent server as well as the client processes. The synchronization between the client process and the DARK OS execution is done by setting a watchpoint on the CPU cycle count register on the DSP. This means that at every processor cycle, the Visual DSP halts because of the watchpoint. The IDE interface keeps track of the global time of the DSP. The DSP is put back into the execution mode as soon as the global time of the client process exceeds the global time of the DSP. Thus the simulator achieves synchronization in all the parts of the simulator at the lowest resolution possible. At the processor level, the synchronization is achieved at the resolution of a single CPU cycle. At the client process, the synchronization is achieved at every network tick.

6.1.3 DSP-FPGA interface simulation

The IDEInterface object handles the interface between the DSP and client process. The implementation of the FPGA protocol stack is divided between the Universal Controller implementation on both server and the client process. The input buffer for the data packets are kept at the client side and the output buffer is implemented on the server side. The server side implements the command processor logic of all the nodes in the network. At the client side, the destination addresses of the packets are checked. If the address belongs to the nodes multicast group, then the packet is added to the input buffer. On failure notification from a peer node, the client program changes the membership of the multicast group that it is listening to.

6.1.4 Hardware Manager Simulation

A hardware manager is completely simulated in the server process. A hardware manager simply puts the input feed for the Universal Controllers and then expects the control value by the specified switching time. It logs the time at which it sends the input values and then receives the output values. Thus we can use these timestamps to evaluate the schedule in the Universal Controllers. It accepts the message that is addressed to itself. If it has to put a message into the network it waits for a null message and puts them into the network.

6.2 Evaluation Strategy

We carry out our evaluation operations based on the infrastructure developed till now. There are two important aspects to be measured in the final system

- Performance.
- Fault tolerance aspect.

We shall first see the evaluation of the schedules generated by the offline scheduling algorithm. We see how the generated schedule is affected by the different parameters and how the designer could deploy this system. We run our scheduling algorithm for boost rectifier application. We pick up one of the schedules generated and check that schedule for fault tolerance by injecting nodal failures in the simulator. The first part of the evaluation would demonstrate the effectiveness of the scheduling strategy in terms of attaining the switching period. Section 6.2.1 demonstrates the performance of the new scheduling system. Section 6.2.3 explains how a generated schedule is fault-tolerant. This is verified by the simulator. The simulation of the dependent hardware managers decides the validity of the schedule. The evaluation of the multiprocessor schedule is done by keeping a measure of the timeliness of the arrival of the control output values.

6.2.1 Performance

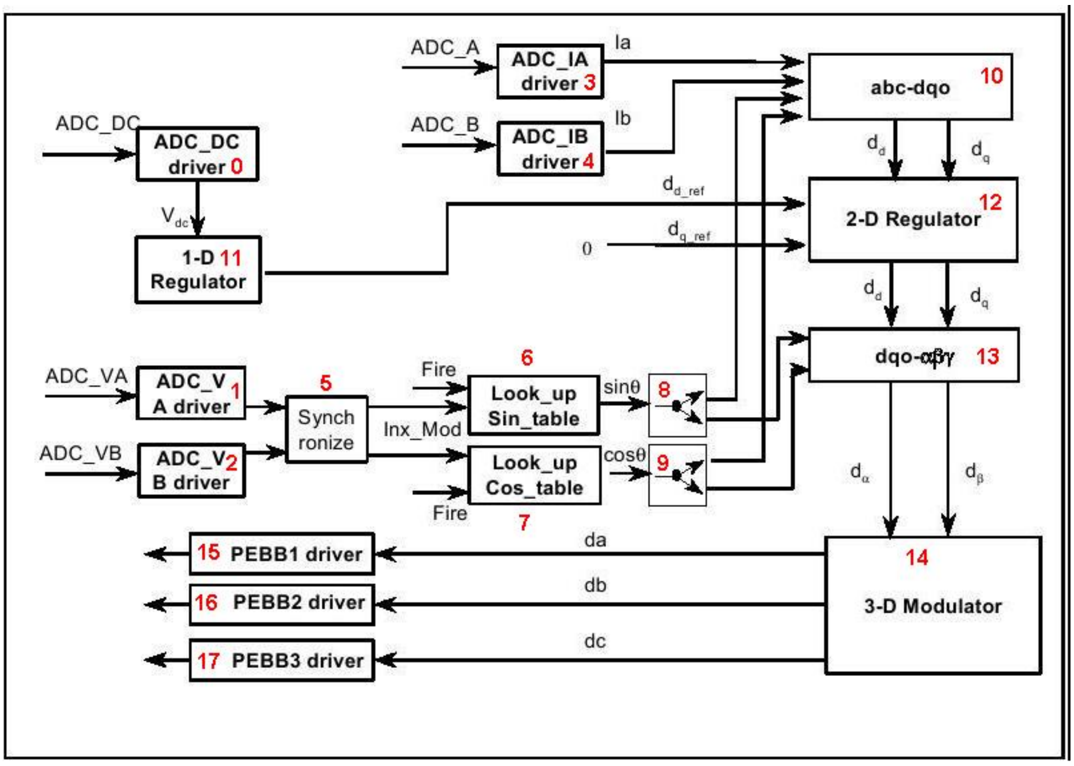


Figure 6.3: Boost Rectifier. Note the id assigned to each ECO.

The performance of the schedule is measured from the schedules generated by the offline scheduling strategy. The longest of the maximum time taken by each processor is assumed to be the switching period. This final value of the switching period is dependant on three factors:

- number of processors (n)
- number of faults to be masked (k)
- communication delay

As the number of processors increases, the algorithm achieves more parallelism in the execution of the sub-components of the application. Hence the total execution time on each of the processors decreases resulting in a faster switching period. One should also note that achieving parallelism involves communication between the processors. The resulting communication delay also plays a role in deciding how effective the benefits of the parallel execution could be. For any value of k, the system has to execute (k+1) copies of the set of tasks. Hence if we increase the value of k, we need to execute more copies of the tasks. Now the whole multiprocessor system has more number of tasks for execution. Moreover, each replicate of the task have to obey the overall precedence constraints. Hence as we increase the value of k, the switching period should tend to increase. We ran our tests for the boost rectifier application. We observed the results by varying 3 parameters, which are the number of processors(n), the number of faults to be masked(k) and the communication delay of the DRPESNET protocol.

Figure 6.3 shows the dataflow diagram of the boost rectifier application. A task that takes 30 CPU cycles for execution has a cost of 30 x 12.5 = 375ns. The cost of the dataflow graph is decided from the values measured in the Visual DSP emulator. The communication delay is computed based on the rate at which the data is transferred by the FPGA protocol stack. There is no assumption on the position of a node in the ring. Thus there is no way that we can have a correct estimate on the time taken to send data from one node to another. However, there is a deterministic guarantee for a packet to reach its destination in the ring by (n-1) times single nodal delay. This means that the data would reach within the round-trip time of the ring network. When there is a failure in the network, the delay because of the traversal through the failure ring adds another (n-1) times single nodal delay time for the packet to reach its destination. So in the worst case, a packet is deterministically sent to its destination within twice the roundtrip time of the network. A single nodal delay is assumed to be 1280ns. We executed the scheduling algorithm for the application on a number of processors with different values for k (k is the number of processor faults that the system should mask). This is the core of the evaluation, where we check whether the resultant schedules are acceptable. We vary the communication delay to show the difference that a faster communication system between the processors can make. The resultant scheduling frequencies are acceptable if they are in the range 2kHz-20kHz. A ring network with n number of processors and three hardware managers was considered for generating a schedule. We vary the value of k from zero to three. At k = 0, the system is not fault-tolerant but it is scheduled on a multiprocessor system.

The graphs in figures 6.4 and 6.5 show an interesting trend.

- The switching frequency decreases inspite of achieving parallelism with multiple processors.
- Using a 10 times faster network results in faster switching frequencies.
- The switching frequency achieved is in the limits of the required switching frequency in the field of power electronics.

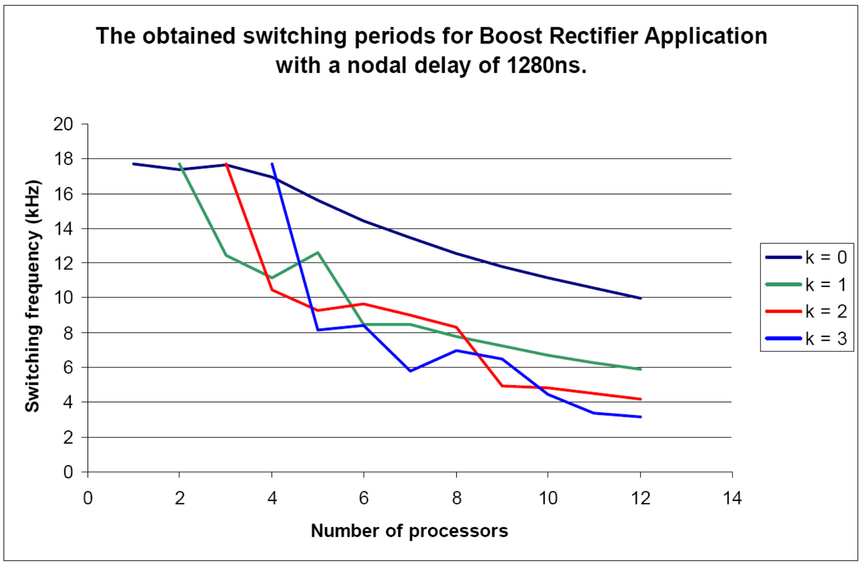


Figure 6.4: The results obtained by the fault-tolerant version of the Boost Rectifier application

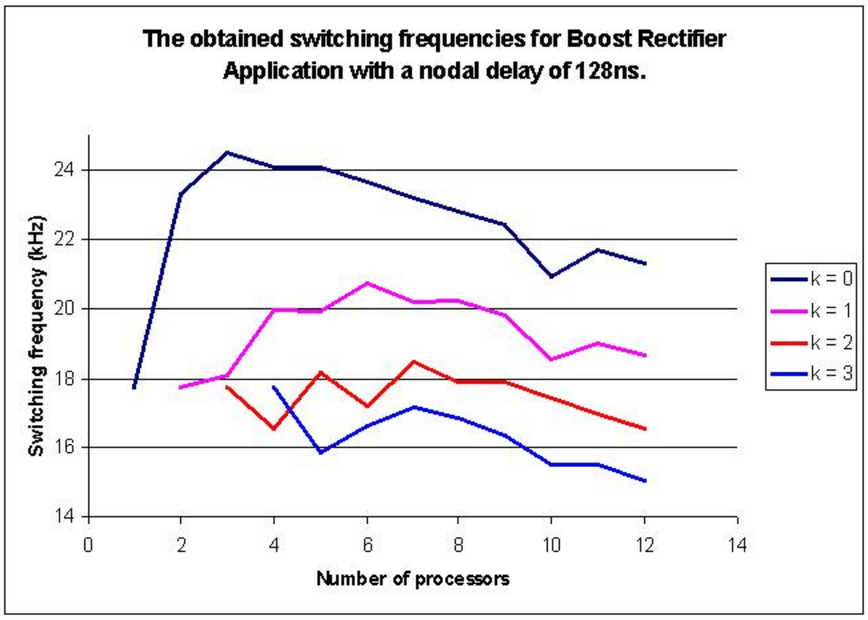


Figure 6.5: The switching frequency achieved with a faster communication protocol shows that the computation part is optimized enough to achieve better switching frequency with fault tolerance.

Figure 6.4 depicts the results on the actual system. We find that the highest achievable frequency is same as executing all the ECO's on the single processor and duplicating the whole system. This is attained at round 18kHz. As we increase the number of processors in the system, we find that the switching frequency decreases. When we use 12 processors in the system, this frequency reduces to around 3kHz in order to mask failure of 3 processors. This lets our expectations down. We expected the switching frequency to increase as we increased the number of processors in the system. Figure 6.5 shows the reason why. We reduced the single nodal delay by one tenth of the existing value. As the graph shows. now the overall switching frequencies are increased. In case of k = 0 and 1, we can attain switching frequencies greater than 20kHz. From this we can surely infer that the computational system, which now comprises of multiple processors, has definitely improved in performance. This improvement is an extra advantage that we have gained in this fault-tolerant system. The large round trip time is responsible for increasing the switching period. This is evident if we compare both the graphs. They show how decreasing the communication delay results in lower switching periods. The ring topology increases the communication cost with addition of new nodes. As our algorithm tries to achieve better parallelism, the increasing communication cost, lets down the improvement in performance if any. This is happening to the extent that having such a task allocation policy becomes meaningless and a k fault-tolerant system requires you to run k+1 identical copies on each of the k+1 nodes. However, looking at the requirements, we have a faster computational system that is fault-tolerant. The performance of this system is however dictated by the communication delay involved.

6.2.2 Example Fault Tolerant Schedule

Figure 6.6 shows a fault-tolerant schedule on a 4 processor system that can mask a single processor failure. The tasks that showed in green are receive tasks and the tasks that are shown in blue are send tasks. The unlabeled blocks denote the idle time on each processor. The label of a block denotes the ECO-number that is executed. Multiple numbers in a block denotes the ECO's that were grouped after first phase of the algorithm. We have to understand few important factors from figure. The DAG structure of the boost rectifier application is not a perfect example that we should be evaluating the system on. From the schedule, we can see the large slack time on the processors 3 and 4 while the other two processors are executing the ECO's {12, 13, 14}. The execution cost of these ECO's sum upto 14490ns. This value is more than 25% of the total cost for executing the application on a single processor. Thus this application does not get the best out of the task allocation strategy we have. Another factor that we have to see is the large idle time in the processor before the execution of the tasks {12, 13, 14}.

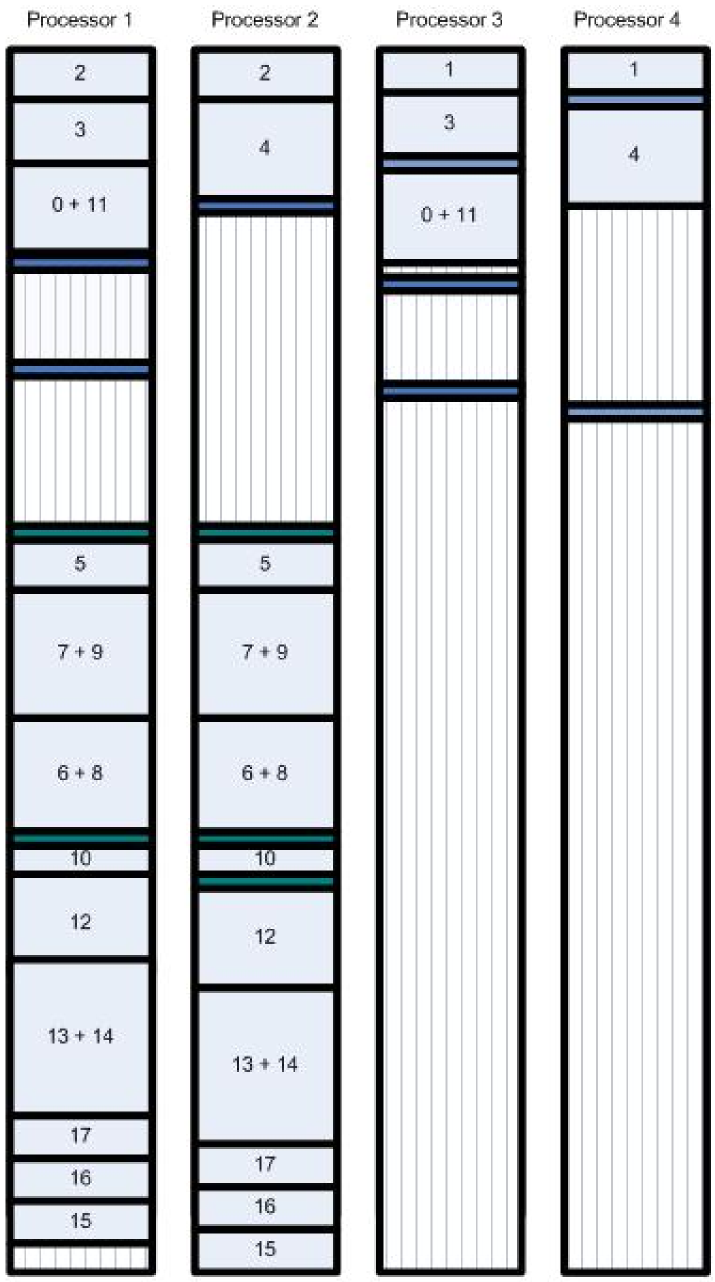


Figure 6.6: Example schedule for boost rectifier application

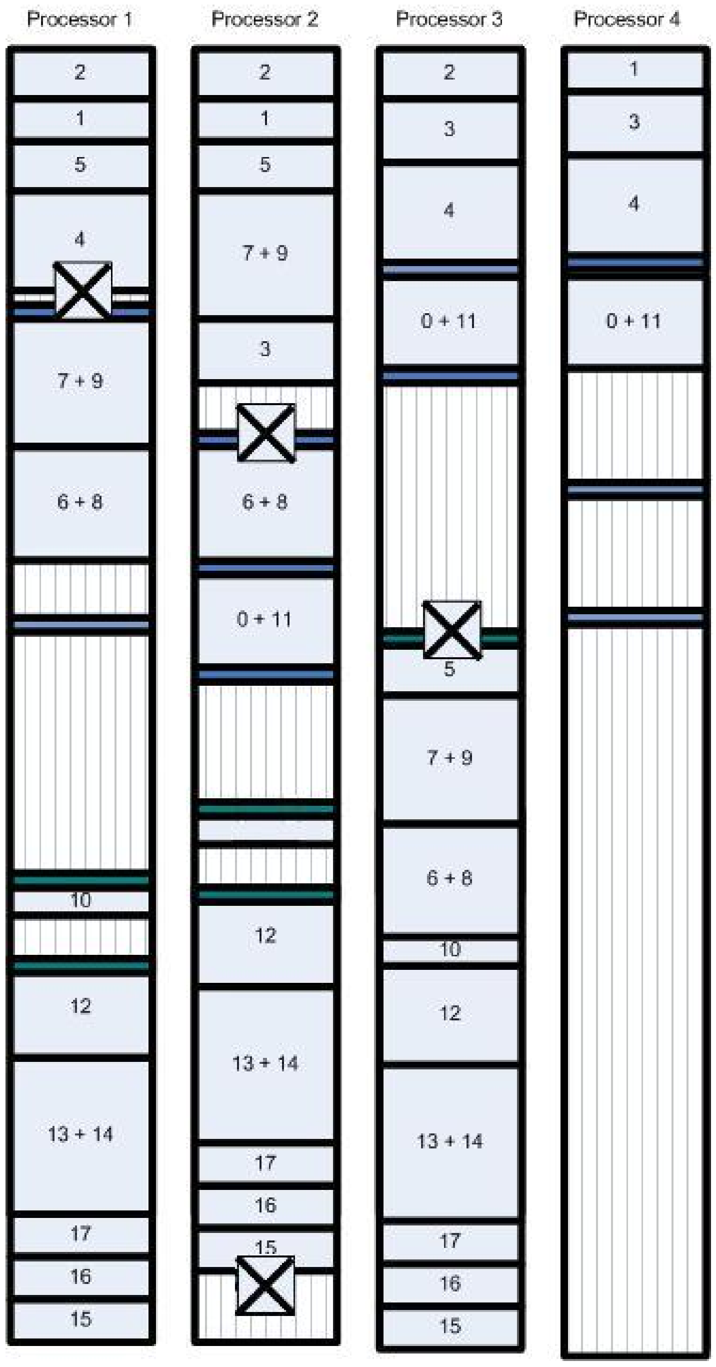


Figure 6.7: Injected faults in the schedule for a 4 processor system that can mask 2 processor failures

This is because of the large communicaiton cost that is present between the send and receive communication tasks. The algorithm also decides against running tasks {14,15,16} in parallel to avoid a late startup of those tasks after communication.

6.2.3 Fault Tolerance Aspect

Now that we have evaluated the performance of the new system, we shall now check the fault tolerance of the the schedule. We now inject failure at certain important points. For example, failing the processor, at the instant when the task is scheduled to be on its communication interface or failing the processor in between of execution of a task. failing a processor as soon as all the tasks are executed for that switching cycle and so on. The points that we are going to inject faults are

- In between the execution of an ECO
- In between the execution of a send communication task
- At the instant the receive task has started execution
- At the instant the output driver ECO is executing

Figure 6.7 shows the schedule on 4 processors. This schedule makes the system tolerant of a single processor failure. The crosses on figure shows the time at which the fault was injected and the processor that failed. We then check whether the hardware manager got the expected results in time. We found that the hardware manager got the expected outputs in time. Thus the schedule generated was a fault-tolerant schedule.

6.3 DRPESNET Block Simulation

In this section, we observe the simulation signals of the various blocks of the DRPESNET protocol stack. In the chapter 5, we saw the design details as well as the state diagrams of operations of the different blocks that are added to the DRPESNET protocol stack. In this section, we shall observe the simulation signals of the working of few of the important blocks that we have discussed. The VHDL implementation of the blocks are tested based on the test benches simulated by the ModelSim software. We are going to see the simulation of 2 most important blocks here - the configuration manager and the TfRf block.

6.3.1 Configuration Manager

Configuration Manager has two modes of operation - leader mode and the non-leader mode. Figures 6.8 and 6.9 show the simulation signals for the

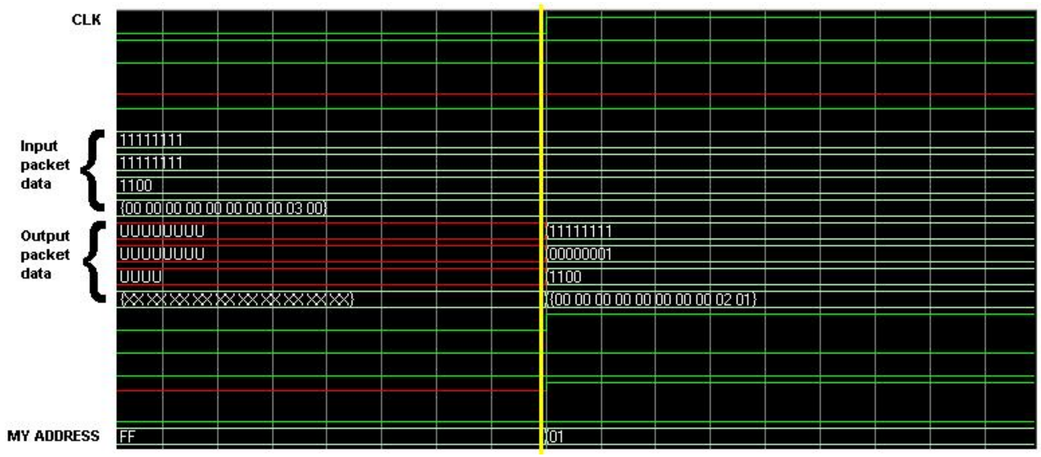


Figure 6.8: Leader mode simulation

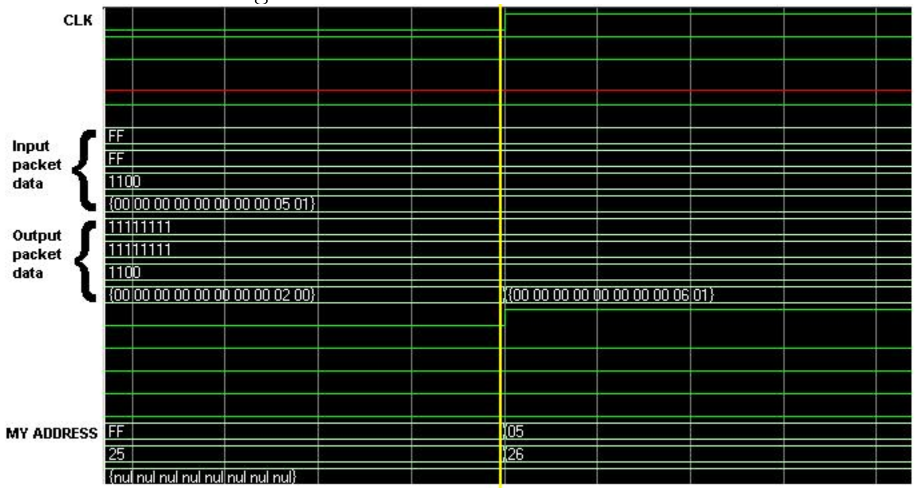


Figure 6.9: Non-leader mode simulation

implementation of the block. Figure 6.8 shows the leader mode of operation and 6.9 shows the non-leader mode of operation. In all the diagrams, the yellow line signals the positive trigger of the clock where the expected state transition occurs.

In the leader mode, we can see that as soon as the configuration manager gets the election packet with its own slot id, it signals the other blocks that the node is the leader of the network. As we can see, it sets its own address as 0x01. It puts the address initializing packet in the network. This is shown by the changed value of the output packet at the positive trigger of the clock. Note that the value of the address field is set as 0x02 for the next Universal Controller to accept. This packet initializes the addresses of all the UC's in the network.

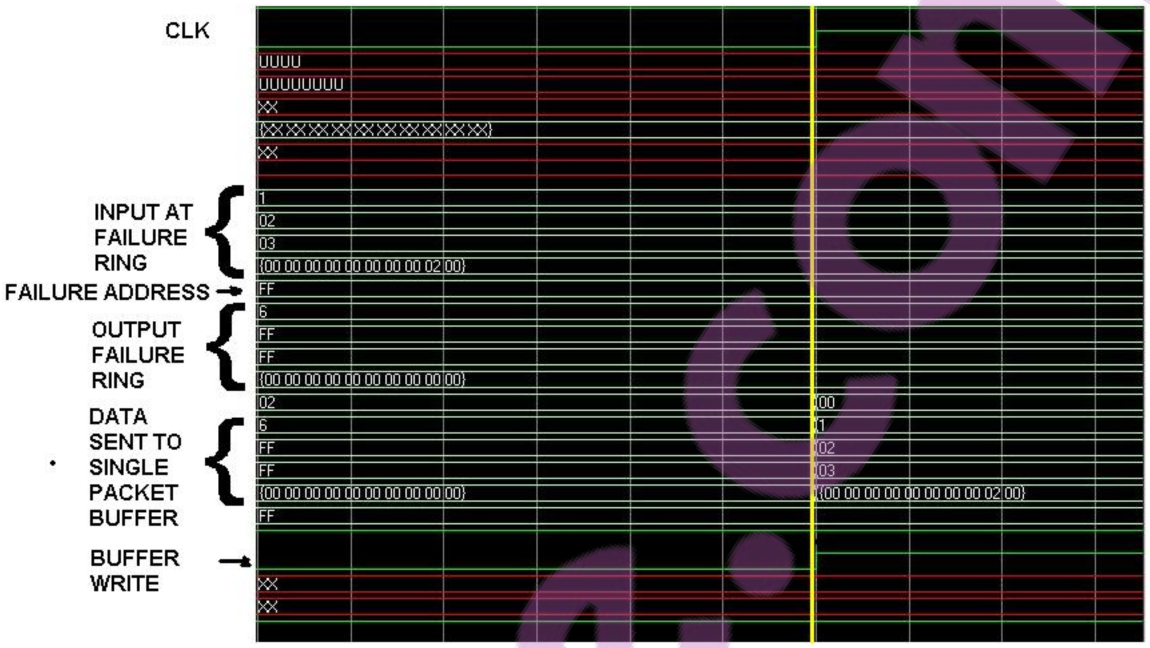


Figure 6.10: Simulation of TfRf block

In the non leader mode we can see how the node accepts the address packet accepts the address in packet as its own address and then forwards the address initialization packet with an address greater than the accepted address by 1. Figure 6.9 shows that the node accepts its address as 0x05 as its own and puts 0x06 in the network. One should also note that this value of the address is equal to the value of the rank of the UC.

6.3.2 Tf-Rf

The Tf-Rf block is responsible for transferring packets that arrive at the failure ring. It is also responsible for routing the wandering packets into the primary ring. Figure 6.10 shows the simulation signals for this operation. The Tf-Rf block realizes that the packet is to be redirected when the value of the *fault address* field is 0xFF. On finding that the value of the field is 0xFF, the Tf-Rf block tries to move this packet to the single packet buffer. This single packet buffer signals Tf-Rf about its availability by the *BUFFER_RDY* signal. As shown in the diagram, when the value of the *fault address* is 0xFF, at the next positive trigger of the clock, the output to the transmitter link of the failure ring is a null packet. The data is routed to the failure buffer by transferring the "heal" packet to the single packet buffer. The signal *buffer_write* indicates to the single packet buffer that a packet is to be written.

Chapter 7

Conclusion And Future Work

7.1 Conclusion

The drawbacks of existing approaches to PEBB-based power electronics control, which lack support for fault tolerance, motivated the work in this thesis. Eventhough CPES researchers had a standardized architecture for controlling power covnverter systems, the system still was not able to recover from hardware failures. The system was centralized on a single real-time computational unit. Hence failure of this unit lets down the complete system. Hence we needed to make this system fault-tolerant. This work involved developing a hard real-time fault-tolerant system. In order to develop such a system, the DARK microkernel was extended with one more mode of operation, which is the fault-tolerant distributed mode. This mode was designed for running redundant Universal Controllers. To realize these new features, we had to tackle a fault-tolerant real-time scheduling problem. Taking into consideration, the architecture of the system, we concluded that for a system that has a very limited processing capacity and with very high communication cost, the scheduling has to be offline with the schedules generated before the system is deployed for execution. For the same reasons, this system cannot have a dedicated hardware that takes scheduling decisions.

Just as literature has suggested, fault-tolerant scheduling of hard real time tasks with precedence constraints is a hard problem. In this system, the dataflow architecture ensured that there would not be any resource conflicts. The nature of the tasks are such that each of them had a predictable completion time. With such nature of tasks, the problem has charecteristics similar to a bin packing problem. The algorithm developed is a greedy algorithm that tries to achieve as much parallelism as possible with the dominant sequence and schedule every task as early as possible. The algorithm has also considered load at the communication interface. In this system, a multicast based replica listening strategy was employed that minimized delay caused by the bandwidth difference at different levels of communication. This multicast based replica listening manipulates the bandwidth at the different levels of the communication found in such systems.

Chapter 6 shows how we achieved a fault-tolerant system. Along with the fault tolerance, the algorithm has promised a faster computational unit if it is deployed on a faster communication system. Even with this high communication cost, the resultant schedule were within the expected period of time. We can say that this work has reduced the time required by the processing unit of the real time computation unit. However, because of the large network delay, the improvement in performance obtained by achieving parallelism is nullified. The schedule generated also has a zero recovery time. The local scheduler design is as trivial as picking up the current task and executing it. Thus in the end we have a faster and fault-tolerant computational unit for control in power converter systems.

7.2 Future Work

7.2.1 Better Parallelism in Power Eletronics Control Applications

The current results that we found using Boost Rectifier Applications can be still improved in a significant manner, if the application was designed with better parallelism. Since we could not achieve completen parallelism, there were unnecessary idle time in the processor schedules. A major section of the dominant path had no other tasks that could be executed in parallel. Hence, if the application designer could design the application with better parallelism our offline algorithm could generate even faster switching frequencies.

7.2.2 Implementing the Data Channels on the FPGA

High communication cost is a drawback of this system. The most important cause of this high cost is that every communication operation involves copy operation of the data between the local memory and the FPGA memory on both the processors. The data channels in the system is known offline hence the datachannels could be written in the FPGA block. If we implement all the datachannels in the FPGA, the DSP can read the data from the FPGA using the memory mapped interface. At the same time the protocol stack could be made smart enough to route the data packets to the correct datachannels. This would reduce the communication cost at the DSP by a considerable amoun of time (typically 600ns). However all this is subject to the question whether available FPGA memory is large enough.

7.2.3 Improving the Network Protocol

The network protocol has to be improved to ensure reliability of data transfer as well as to provide deterministic guarantee for any distributed data channel operations. The communication interface of this system is at the point where the DSP reads or writes the data into the data buffers in FPGA. The determinism should be provided at this level. Hence the expected arrival time of the data packet should not be based on some mathematical assumption on the network traffic. This could be done by having a temporal reservation strategy for each node, where every node is ensured that the packet put into the buffer would reach the destination within a deterministic time. Another strategy could be to have the protocol also synchronized with the schedule of tasks at different processors. To ensure reliabiliy, the ring network should be replicated more and there must be some routing protocol that ensures timely arrival of the data packets in case of failure.

Bibliography

[1] Jinghong Guo, Stephen H. Edwards, and Dushan Borojevic. Implementing dataflow-based control software for power electronics. In Proceedings of the IEEE 9th Workshop on Computers in Power Electronics (COMPEL), 2002.

[2] Gerald Francis, A Synchronous Distributed Digital Control Architecture for High Power Converters, MS Thesis, Dept of Electrical & computer Engineering, Virginia Tech, 2004.

[3] K. Singh, Design and Evaluation of an Embedded Realtime Micro-kernel, MS Thesis, Dept. of Computer Science, Virginia Tech, 2002. Available on-line at: ¡http://web-cat.cs.vt.edu/PEBB/publications.php¿.

[4] Parool Mody, Supporting Transparent Dataflow Messaging in Distributed Power Electronics Control Systems, MS Thesis, Dept. of Computer Science, Virginia Tech, 2003.

[5] Jinghong Guo, Distributed, Modular, Open Control Architecture for Power Conversion Systems, PhD dissertation , Dept. of Computer Science, Virginia Tech, 2005.

[6] H. Kopetz, Real-time systems, design principles for distributed embedded applications, Kluwer academic publishers, ISBN: 0-306-47055-1

[7] Scheduling in Real-Time Systems,Francis Cottet, Joelle Delacroix, Claude Kaiser, Zoubir Mammeri ISBN: 0-470-84766-2

[8] G. C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer Academic Publishers, 2000.

[9] K.G. Shin and P. Ramanathan, Real-Time Computing: A New Discipline of Computer Science and Engineering,Proc. IEEE, Vol. 82, No. 1, Jan. 1994, pp. 6-24.

[10] Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., and Zainlinger, R. 1989. Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. IEEE Micro 9, 1 (Jan. 1989), 25-40. DOI= http://dx.doi.org/10.1109/40.16792

[11] Xu, J.; Parnas, D.L., "On satisfying timing constraints in hard-real-time systems," Software Engineering, IEEE Transactions on , vol.19, no.1pp.70-84, Jan 1993

[12] Tanenbaum, A. S. and Steen, M. V. 2001 Distributed Systems: Principles and Paradigms. 1st. Prentice Hall PTR.

[13] H. Kopetz and G. Grunsteidl. TTP - a Time-Triggered Protocol for Fault-Tolerant Real-TimeSystems. In Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing, pages 524–533, Toulouse, France, June 1993. IEEE

[14] Jeffay, K.; Stanat, D.F.; Martel, C.U., "On non-preemptive scheduling of period and sporadic tasks," Real-Time Systems Symposium, 1991. Proceedings., Twelfth , pp.129-139, 4-6 Dec 1991

[15] Stankovic, J. A. and Ramamritham, K. 1991. The Spring Kernel: A New Paradigm for Real-Time Systems. IEEE Softw. 8, 3 (May. 1991), 62-72. DOI= http://dx.doi.org/10.1109/52.8894

[16] E. Tovar, F. Vasques, and L. M. Pinho, Engineering real-time applications with WorldFIP: Analysis and tools, in Proc. SICICA 2000 Buenos Aires, Argentina, 2000, pp. 297-302.

[17] .K. W. Tindell, H. Hansson, and A. J. Wellings, Analyzing real-time communications: Controller Area Network (CAN), Proc. Real-Time Systems Symp., pp. 259-263, Dec. 1994.

[18] Ross, F.E., "An overview of FDDI: the fiber distributed data interface," Selected Areas in Communications, IEEE Journal on , vol.7, no.7pp.1043-1051, Sep 1989

[19] Stankovic, J. A. and Ramamritham, K. 1989. The Spring kernel: a new paradigm for real-time operating systems. SIGOPS Oper. Syst. Rev. 23, 3 (Jul. 1989), 54-71.

[20] Stankovic, J. A. and Ramamritham, K. 1991. The Spring Kernel: A New Paradigm for Real-Time Systems. IEEE Softw. 8, 3 (May. 1991), 62-72

[21] Schlichting, R. D. and Schneider, F. B. 1983. Fail-stop processors: an approach to designing fault-tolerant computing systems. ACM Trans. Comput. Syst. 1, 3 (Aug. 1983), 222-238. DOI= http://doi.acm.org/10.1145/357369.357371

[22] F. Gaertner and H. Volzer. Redundancy in space in fault tolerant systems. Technical Report TUD-BS-2000-06, Department of Computer Science, TU - Darmstadt, 2000.

[23] Bertossi, A.A.; Mancini, L.V.; Rossini, F., "Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems," Parallel and Distributed Systems, IEEE Transactions on , vol.10, no.9, pp.934-945, Sep 1999

[24] S. Balaji, L. Jenkins, L.M. Patnaik, and P.S. Goel, Workload Redistribution for Fault-tolerance in a Hard Real-Time Distributed Computing System, Proc. IEEE Fault-tolerance Computing Symp. (FTCS-19), pp. 366-383, 1989.

[25] Lee, Y. and Shin, K. G. 1982. Rollback propagation detection and performance evaluation of FTMR2Ma fault-tolerant multiprocessor. In Proceedings of the 9th Annual Symposium on Computer Architecture (Austin, Texas, United States, April 26 - 29, 1982). International Conference on Computer Architecture. IEEE Computer Society Press, Los Alamitos, CA, 171-180.

[26] Stankovic, J.A., "Decentralized decision-making for task reallocation in a hard real-time system," Computers, IEEE Transactions on , vol.38, no.3pp.341-355, Mar 1989

[27] On a Real-Time Scheduling Problem, Sudarshan K. Dhall; C. L. Liu Operations Research, Vol. 26, No. 1, Feb., 1978, pp. 127-140.

[28] Krithi Ramamritham, "Allocation and Scheduling of Precedence-Related Periodic Tasks," IEEE Transactions on Parallel and Distributed Systems, vol. 06, no. 4, pp. 412-420, Apr., 1995

[29] Alan A. Bertossi, Andrea Fusiello, Luigi Vincenzo Mancini, "Fault-Tolerant Deadline-Monotonic Algorithm for Scheduling Hard-Real-Time Tasks," ipps, p. 133, 11th International Parallel Processing Symposium (IPPS '97), 1997

[30] E. Maehle, F. Markus: Fault-Tolerant Dynamic Task Scheduling Based on Dataflow Graphs, Proc. IPPS'97, Workshop on Fault-Tolerant and Distributed Systems, Switzerland 1997.

[31] J. Guo, S. Edwards, and D. Boroyevich. Elementary control objects: Toward a dataflow architecture for power electronics control software." In Proc. IEEE 33rd Annual Power Electronics Specialists Conf. (PESC 2001), June 2002.

[32] Lubeck, O.M., "A user's view of dataflow architectures," Compcon Spring '90. 'Intellectual Leverage'. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference. , vol., no.pp.84-87, 26 Feb-2 Mar 1990

[33] Iannucci, R. A. 1988. Toward a dataflow/von Neumann hybrid architecture. SIGARCH Comput. Archit. News 16, 2 (May. 1988), 131-140. DOI= http://doi.acm.org/10.1145/633625.52416

[34] Celanovic, I.; Celanovic, N.; Milosavljevic, I.; Boroyevich, D.; Cooley, R., "A new control architecture for future distributed power electronics systems," Power Electronics Specialists Conference, 2000. PESC 00. 2000 IEEE 31st Annual , vol.1, no.pp.113-118 vol.1, 2000

[35] Celanovic, Ivan, A Distributed Digital Control Architecture for Power Electronics Systems, Thesis, Virginia Polytechnic Institute and State University, May 2000

[36] Anh Nguyen-Tuong; Grimshaw, A.S.; Hyett, M., "Exploiting dataflow for fault-tolerance in a wide-area parallel system," Reliable Distributed Systems, 1996. Proceedings., 15th Symposium on , vol., no.pp.2-11, 23-25 Oct 1996

[37] J. Francis, J. Guo, and S.H. Edwards. Protocol Design of Dual Ring PESNet (DRPESNet), CPES 2002 Power Electronics Seminar, Virginia Tech, Blacksburg, VA, 2002. Available on-line at: ¡http://web-cat.cs.vt.edu/PEBB/CPES02-Francis.pdf¿.

[38] Milosavljevic, I.; Zhihong Ye; Boroyevich, D.; Holton, C., "Analysis of converter operation with phase-leg control in daisy-chained or ring-type structure," Power Electronics Specialists Conference, 1999. PESC 99. 30th Annual IEEE , vol.2, no.pp.1216-1221 vol.2, 1999

[39] Sankar, R.; Yang, Y.Y., "An automatic failure isolation and reconfiguration methodology for fiber distributed data interface (FDDI)," Communications, 1992. ICC 92, Conference record, SUPERCOMM/ICC '92, Discovering a New World of Communications. IEEE International Conference on , vol., no.pp.186-190 vol.1, 14-18 Jun 1992

[40] Biao Chen; Kamat, S.; Wei Zhao, "Fault-tolerant real-time communication in FDDI-based networks," Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE , pp.141-150

[41] Girault, A.; Lavarenne, C.; Sighireanu, M.; Sorel, Y., "Fault-tolerant static scheduling for real-time distributed embedded systems," Distributed Computing Systems, 2001. 21st International Conference on. pp.695-698, Apr 2001

[42] Yamashita, M. and Kameda, T. 1988. Computing on an anonymous network. In Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (Toronto, Ontario, Canada, August 15 - 17, 1988). PODC '88. ACM Press, New York, NY, 117-130. DOI= http://doi.acm.org/10.1145/62546.62568