DIPLOMARBEIT

A Resource Management Scheme for the TT-SoC Architecture

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines

Diplom - Ingenieurs

am

Institut für Technische Informatik 182/1

der

Technischen Universität Wien

unter der Leitung von

o. Univ. - Prof. Dr. Hermann Kopetz

und

Univ.Ass. Dipl.-Ing. Bernhard Huber

als verantwortlich mitwirkendem Assistenten

durch

Bernhard Weirich Bakk.techn.
Matr. - Nr. 0126357
Utendorfgasse 3/3, A–1140 Wien

Wien, im März 2008 .....

A Resource Management Scheme for the TT-SoC Architecture

The advance of computer chip manufacturing technology makes it possible to construct full-featured systems on a single chip, yielding a number of advantages. One of which is that very efficient interconnects with high data rates are possible, since each component is part of the chip.

However, there is the risk that the different components, if not properly separated, influence each other. Thus, even a low priority application might cause the operation of a highly critical real-time task to fail. To avoid such a situation the interactions between applications must be encapsulated. To this end the TTSoC architecture uses a central time-triggered on-chip interconnect. This interconnect is protected by a guard, denoted Trusted Interface Subsystem (TISS), at each micro component to guarantee the correct operation.

The objective of this thesis is the design and implementation of a resource management infrastructure for the TT-SoC architecture. Resource management is important when resources are limited, which is not unusual for embedded systems. In particular, on battery operated devices power consumption should be kept at a minimum. Dynamic resource management enables efficient usage of the resources, since they may be allocated on demand and freed when they are no longer needed. For the proposed resource management solution, the components which are involved, their mutual interfaces and the algorithms that run on the components are described and evaluated. Care is taken that the encapsulation, which is encouraged by the TT-SoC architecture, is preserved. This is achieved by dividing the system into trusted and non-trusted parts and by protecting the access to the components within the trusted part.

Ein Resourcenverwaltungssystem für die TT-SoC Architektur

Der Fortschritt der Chipfertigungstechnik ermöglicht es, ein voll funktionsfähiges System auf einem einzelnen Chip unterzubringen, wodurch sich viele Vorteile ergeben. Einer der Vorteile ist die sehr effiziente Kommunikation der Komponenten untereinander.

Es besteht jedoch die Gefahr, dass die mitunter sehr verschiedenen Komponenten, sofern sie nicht entsprechend voneinander abgegrenzt sind, sich gegenseitig beeinträchtigen. Dadurch kann selbst eine Applikation mit niedriger Priorität die Funktion einer sicherheitskritischen Echtzeitanwendung stören. Um das zu vermeiden, benötigt man deterministische und vor unbeabsichtigten Eingriffen geschützte Schnittstellen. Die TT-SoC Architektur, die auf einem zeitgesteuerten On-Chip Netz aufbaut, wurde für diese Zwecke entworfen. Zugriffe auf das On-Chip Netz werden vom sogenannten Trusted Interface Subsystem (TISS) überwacht, wodurch eine korrekte Funktionsweise gewährleistet werden kann.

Das Ziel dieser Arbeit ist der Entwurf und die Implementierung einer Infrastruktur zur Ressourcenverwaltung für die TT-SoC Architektur. Ressourcenverwaltung ist für Systeme wichtig, in denen nur wenig Ressourcen zur Verfügung stehen. Im besonderen in batteriebetriebenen Geräten muss auf eine geringe Leistungsaufnahme geachtet werden. Dynamische Ressourcenverwaltung erlaubt effiziente Nutzung der Ressourcen, da diese nur bei Bedarf angefordert werden und nach Gebrauch wieder für andere Anwendungen nutzbar sind.

In dieser Arbeit wird ein Lösungsansatz zur Ressourcenverwaltung vorgestellt. Weiters werden die dafür notwendigen Komponenten, die gegenseitigen Schnittstellen und die verwendeten Algorithmen erläutert. Es wird dabei darauf geachtet, dass "Encapsulation" (Abkapselung), eine wichtige Eigenschaft der TT-SoC Architektur, nicht beeinträchtigt wird. Dies wird dadurch erreicht, indem das System in vertrauenswürdige und nicht vertrauenswürdige Teile unterteilt wird, wobei der Zugriff auf Komponenten im vertrauenswürdigen Bereich überwacht wird.

Danksagung

Ich möchte mich an dieser Stelle bei jenen bedanken die zur Arbeit beigetragen haben. Allen voran meinem betreuenden Assistenten Bernhard Huber, der mit viel Mühe und Sorgfalt über der Entstehung der Arbeit gewacht hat. Weiters sollen meine Eltern Elisabeth und Herbert Weirich nicht unerwähnt bleiben, die mir während der gesamten Ausbildung den Rücken freigehalten haben und mir somit die Möglichkeit gaben mich voll und ganz auf das Studium zu konzentrieren. Mein Dank gilt auch meiner Freundin Nicole Fuchs, die mir den notwendigen Ausgleich zur manchmal trockenen Arbeit geboten hat.

Vielen Dank auch an jene durch die das TT-SoC Projekt entstanden ist und auch an diejenigen, die am Projekt an meiner Seite mitgewirkt haben.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

The advances in chip manufacturing technology have made it possible to integrate a whole full-featured system on a single die. And still, the hardware structures are becoming smaller, allowing to put a great amount of functionality in small chip areas. It has become feasible to build multiple distributed application subsystems on a single chip. The advantages are promising: cost savings due to reduction of the number of hardware parts and increased reliability since on-chip interconnects are less susceptible to physical stress than connections on the PCB. Furthermore, a performance gain can be expected as data exchange can be done more efficiently and finally, power dissipation is cut when using a single chip.

However, the possibilities offered by this evolution are countered by challenges to the development of such highly integrated systems. The complexity of these systems becomes insurmountable if proper measures are not taken. Development gets error-prone as there are too many issues to be addressed at the same time. The most important technique to cope with this complexity is abstraction. By relying on well specified interfaces, the main focus can be put on the actual application development. In System-on-Chips (SoCs), the interconnect between different components is becoming more and more difficult to handle since the signal delay of on-chip wires is no more negligible with current technology. This complexity can be hidden from application developers using a Network-on-Chip (NoC) infrastructure to connect all components of the system. The challenge here is to design an efficient, yet flexible network with small overhead so that the cost does not outweigh the benefits. This issue is addressed by many research projects by now and appropriate solutions have been found already. However, hardly any of the solutions can be used in the context of safety-critical real-time application, which are very demanding in terms of reliability and need the provision of guarantees on

communication properties.

The result is that the System-on-Chip solutions cannot be exploited by safety-critical real-time applications. For example, an electronic system in a car may be built of one application subsystem for electronic stability control, another for a multimedia system and a third for motor control. The classical approach is to form three distinct systems that share nothing except the battery as the common power source. With three systems this is not a big issue, but nowadays cars feature many more application subsystems than those mentioned already. Each system must be provided with the resources it needs during peak load. If the systems are combined on the same chip, resources can be shared and less resources are required in total if two systems never run at peak load at the same time. Often, the risk that one application subsystem has adverse effects on the execution of another is too big if they were put on the same chip if they are not properly separated. Thus, safety-critical real-time applications are mostly excluded from the benefits of SoC solutions.

1.2 Contribution

The context for the findings presented in this thesis is how safety-critical real-time systems can make use of chip technology and profit from SoC architectures.
For this, the Time-Triggered System-on-Chip (TT-SoC) architecture, an architecture designed for real-time safety-critical applications, is used as a basis for the design. Primary focus in this thesis is the dynamic resource management scheme that enables efficient use and sharing of resources in the SoC. It is explained how dynamic resource management can be realized even in the co-existence of application subsystems of different degree of criticality. In other words how it can be prevented that an application subsystem can negatively influence another application subsystem on the same SoC, sharing the same resources.
To prove the utility of the proposed architecture, the SoC design is then implemented on a Field-Programmable Gate-Array (FPGA) and the performance and behavior of the resource management facilities is evaluated.

1.3 Structure of this Thesis

The remainder of the thesis is organized as follows. Chapter 2 introduces the concepts on which the work is based and presents related work. In Chapter 3, the TT-SoC architecture, on which the System-on-Chip (SoC) designed in this thesis is based, is explained. The refined design of the SoC, for which resource management is implemented, is detailed in Chapter 4. Furthermore, the mechanisms working

together to make dynamic resource management possible are explained. Chapter 5 goes into the details of how the design was implemented and thoroughly describes the interfaces between the components. How the design was evaluated along with the results obtained can be found in Chapter 6. And, finally, Chapter 7 closes with the conclusions drawn from the work and what is left to do in future projects.

Chapter 2

Basic Terms and Concepts

2.1 Real-Time Systems

Real-time systems differ from conventional computer systems by having to provide results which must not only be correct in the value domain, but also have to arrive within a specified time interval (deadline). A system is called a hard real-time system if a violation of the deadline renders the result useless. In some systems this may lead to a system failure. If deadlines in a soft real-time system are missed, the quality of service is degraded (e.g. a frame is dropped). While usual systems try to optimize the average case, hard real-time systems focus on the worst case, since the timing requirements must be met in every single case. An exhaustive overview on the topic of real-time systems is available in [Kop97].

Real-time requirements do not only have an impact on the algorithms used in computations, but equally on the communication mechanisms. Especially the Medium Access Control (MAC) schemes must be devised carefully. For example, Carrier Sense Multiple Access (CSMA), which is used in Ethernet, cannot be used in real-time application without adding a flow-control mechanisms to an upper network layer. Using plain CSMA, it cannot be guaranteed that any packet arrives in time during high-load situations if access to the network is not coordinated. However, Kopetz et al. [KAGS05] have developed a variant of Ethernet called Time-Triggered Ethernet that guarantees collision free access to the Ethernet medium by synchronizing and coordinating the network nodes.

For the system described in this thesis, the Time-Division Multiple Access (TDMA) access scheme is used that works very well for real-time systems. It is explained in the next section.

2.1.1 TDMA

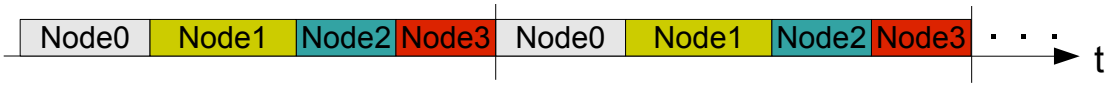


Figure 2.1: Time-Division Multiple Access (TDMA)

In TDMA, all network nodes are assigned a certain time interval where they are allowed to transmit data over the network. Each time interval is repeated periodically (cf. Figure 2.1). Obviously, these assignments must be exclusive so that in no case two nodes are ever sending simultaneously.

The big advantage of TDMA is that every node can rely on the network being available during its time share which is crucial for real-time applications. The drawback of TDMA is, that the network is much less flexible. If additional nodes are connected they must be assigned a time share before they can send messages. Furthermore, all nodes must be synchronized to a global time base for obvious reasons.

2.2 Pulsed Data Streams

Pulsed data streams are a novel communication primitive proposed by Kopetz in [Kop06]. They were developed for TDMA networks to allow the coexistence of messages with short periods and messages with long periods. The problem with messages with long periods is that their duration may be long, thus they block the communication medium for too long a time so that messages with short periods are no more possible as illustrated in Figure 2.2.

The solution to this is simple but effective: divide message “2” into multiple parts and send the individual parts over a longer time interval leaving gaps of inactivity between the parts. The parts are called “Fragments”. Similarly, divide message “1” so that it fits between the Fragments of message “2” (see Figure 2.3). These fragmented messages are called “pulsed data streams”.

Summing up the above, a pulsed data stream is a periodic message that is split up into individual fragments. The size of a fragment is determined by the underlying communication system. The length of the Pulse period is a non-positive power of 2 seconds to reduce the complexity of the communication system. The distance between two neighboring fragments is determined by the Fragment Period of a pulse, which must be a non-positive power of 2 seconds as well. The four parameters that define a pulsed data stream are the Pulse Period, the Fragment Period, the Number of Fragments and the Phase Offset (cf. Figure 2.3).

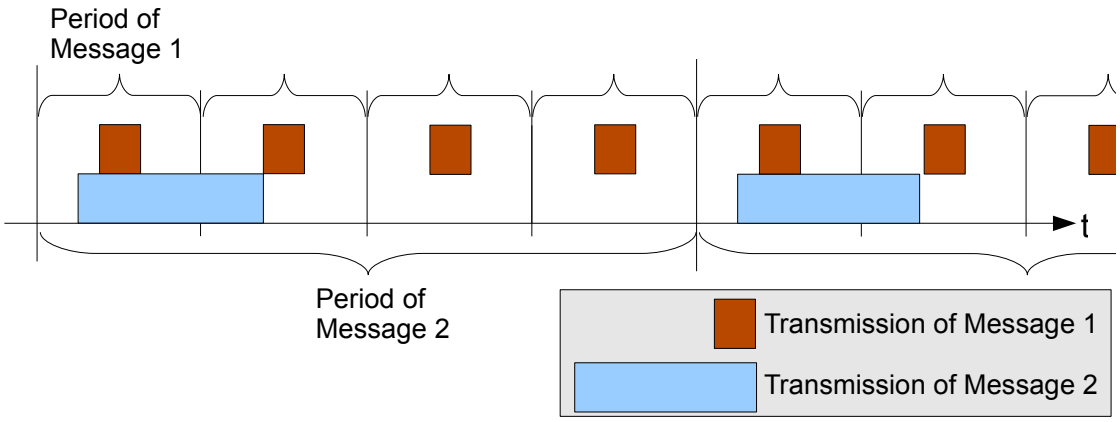


Figure 2.2: Problems of mixing long and short periods

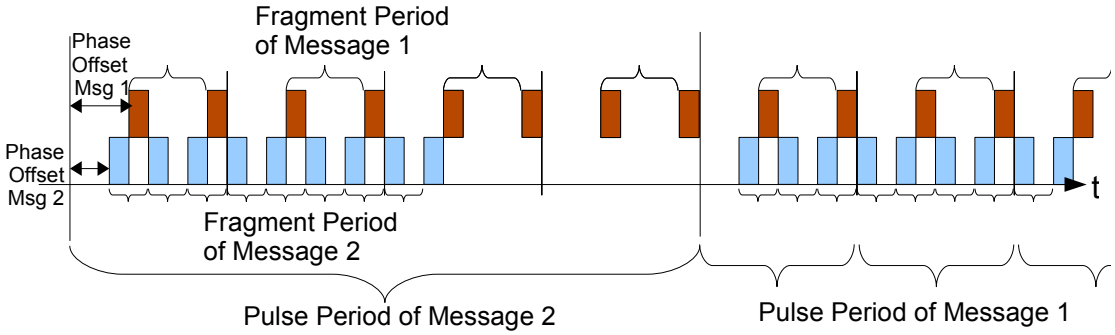


Figure 2.3: Spreading messages to pulsed data streams

Pulsed data streams follow the time-triggered paradigm which states that actions (e.g. transmission of packets) are triggered at predefined time instants. This allows very low latencies in a variety of applications, since the sending instants are defined a priori. For example, consider a control application where a sensor must be read, its data has to be sent to the control unit, which calculates the control values. The control values are then sent to the actuators to be applied. Proper alignment of both pulsed data streams makes it possible to generate a seamless flow of actions, which is not possible in event-triggered networks where transmission latencies depend on the network load.

2.3 System-on-Chip (SoC)

"The definition of an SoC is simply a chip where an entire system is designed into a single ASIC" (taken from Siewert [Sie06]). In the past, it was not possible to put all components that comprise a computer system on a single chip. So every

component resided in its own chip, connected to the others via the connections on the Printed Circuit Board (PCB). In the 1970s and 1980s, attempts were made to build integrated circuit networks that use an entire silicon wafer to host a collection of components, but these were thwart by manufacturing complexities. Since the 1990s, as technology evolved, SoC architectures are used in a number of applications. They have numerous advantages. One of which is reduced cost compared to multiple chips since only one chip must be packaged, tested and mounted to the PCB. Second, the interconnection of components within a chip is more efficient in terms of power consumption, cost and throughput. Third, modular design of the SoC is achieved using Intellectual Property (IP)-Cores. An IP-Core is a hardware description of a component that can be integrated into the chip. IP-Cores from different vendors may co-exists on the SoC. Disadvantages of the SoC is an additional complexity in the system design and that a single chip corresponds to a single fault containment region. In other words, the reliability required for safety-critical applications cannot be achieved by having three instances of a component in the same SoC for use with Triple Modular Redundancy (TMR).

2.3.1 NoC

In [LG02], Benini pointed out that for a sound SoC design, the system should be seen as a "micro-network of components". This on-chip network is called the NoC. With the advance of manufacturing technology the timing of on-chip wiring becomes more and more complex. With an NoC the complexities can be hidden from the cores in an elegant way. In [JT03], Jantsch comes to the same conclusion, that NoC are needed to "keep up with the pace of technology advances".

2.4 Message Scheduling Techniques

As already mentioned in Section 2.1, communication in real-time systems must be coordinated in order to be able to guarantee worst case end-to-end delays and to guarantee an upper bound on the jitter. This coordination is achieved basically by telling each network node at which instants in time they are allowed to send. Scheduling is the process to find correct values for the sending instants. However, it is not always trivial to find valid values for the sending instants. Messages may depend on others, have strict timing requirements or have to co-exist with message of different period lengths. A general overview on scheduling in distributed embedded systems can be found in [Kop97]. The terms relevant in the context of this thesis are explained below.

2.4.1 Online vs. Offline Scheduling

Since scheduling can be such a resource demanding task, it is often done during the design phase of the system. Then it is called *Offline Scheduling*. If communication demands can change during run-time of the system, all possible cases must be scheduled and stored in different configurations. Thus, the system can adapt online, even if it is not capable to do the scheduling itself.

The alternative to *Offline Scheduling* is *Online Scheduling*. In this case the scheduling is performed during run-time according to requests of the network nodes. The main advantage of *Online Scheduling* over *Offline Scheduling* is that it is much more flexible. The message schedule is not limited to a certain set of pre-generated schedules, but may attain any possible configuration. However, this brings in a disadvantage. There may be situations where the scheduling fails which could compromise system stability if not handled properly. Another disadvantage is that *Online Scheduling* requires processing power in the running system. Since computational resources in typical embedded systems are scarce this implies that most *Online Scheduling* algorithms are *non-optimal*. A non-optimal scheduler is a scheduler that may fail to find a solution even if a solution exists. Most online schedulers use heuristics for finding solutions which can fail in unusual cases. However, online schedulers do not require memory for storing the pre-generated schedules.

2.4.2 Basic Techniques

Full Enumeration

Full Enumeration is a simple concept: try out all possible values until a valid solution is found. It is easy to implement, but in general it has an exponential time complexity. Therefore, it is hardly suitable for *Online Scheduling*. Often, it is even impossible for *Offline Scheduling*, because the number of items to check for a solution is too large.

Branch and Bound

Branch and Bound is a technique invented for solving optimization problems. It uses the two tools *Branch* and *Bound* to explore the problem space more efficiently than plain *Full Enumeration*. Assuming that function *f(. . .)* is to be minimized, it starts with the set of solution candidates. Then, using *Branch* the set is divided to subsets ("Branches") to which tighter constraints apply. Based on these constraints the bounds for function *f(. . .)* in all branches are evaluated in the *Bound* step. Branches with a lower bound that lies above the upper bound of

any other branch obviously cannot lead to the minimal solution. Therefore, they are excluded from further investigation. *Branch* and *Bound* are repeated until a solution is found. The topic was covered comprehensively by Clausen in [Cla99].

An example of how Branch and Bound can be used effectively for combined task and message scheduling is given by Abdelzaher and Shin in [DS94].

2.5 Related Work

As it is generally accepted that NoCs are a promising approach to counter the complexity of highly integrated SoCs, much research is done on the topic.
In addition to working on different topologies and hardware realizations of NoCs, the issue of controlling an NoC by software is addressed. Without software control, resources (i.e. power consumption and network bandwidth) cannot be used efficiently.

In [ANM+05], Avasare et al. presented an algorithm that performs flow control on end-to-end channels in best effort NoCs. They propose to install a monitor application on a centralized Operating System (OS) that collects statistics of all NoC nodes. If the network is congested at a node an algorithm is used to lower the message injection rate of affected nodes. When the network load subsides the injection rate is increased again. This mechanism assures that the network can be operated just below saturation, where maximum throughput is achieved. Furthermore, based on this NoC they developed heuristics to make use of hardware that is reconfigurable at run-time which are described in [NMAM05]. The paper addresses the issue of task migration in such systems.

Radulescu et al. have developed a real-time capable, TDMA based NoC called Æthereal, which is reconfigurable at run-time. It is based on end-to-end connections which may be established or released at run-time via a single or multiple configuration ports. The configuration ports are accessed solely through the NoC itself. Active connections have a guaranteed lower bound on throughput and a guaranteed upper bound on latency. A detailed description is given in [RDP+05].

Hansson et al. propose an alternative to global reconfiguration schemes in [HCG07]. They point out that the negative effects of a global reconfiguration can be avoided if partial reconfiguration is done. For this, the system is divided into parts which may be configured individually without disrupting the other parts.

Chapter 3

Time-Triggered System-on-a-Chip Architecture

In this chapter the Time-Triggered System-on-Chip (TT-SoC) architecture, for which dynamic resource management is implemented, will be presented.

The goal of this architecture is not primarily to optimize for highest performance of the SoC, but to provide an infrastructure that may host multiple application subsystems with different levels of criticality, where no application subsystem can have a negative effect on another application subsystem even in the presence of implementation faults in one of the application subsystems.

One of the most important requirements is the provision of services for safety-critical real-time applications. The next section presents some key properties of the TT-SoC architecture. Later follows the structure of the TT-SoC architecture and the resource management facilities. The TT-SoC architecture is discussed comprehensively in the PhD theses of Christian El Salloum [Sal08] and Bernhard Huber [Hub08].

3.1 Properties / Characteristics

In this section the properties that characterize the TT-SoC architecture and differentiate it from other SoC architectures are listed.

Real-Time The support for real-time applications is a key feature of the TT-SoC architecture. It inflicts the requirement that all latencies caused by the TT-SoC are predictable.

Encapsulation This is a very powerful property, that distinguishes this particular SoC architecture from many others. Encapsulation

means that every component in the TT-SoC is fully independent from the other components from a functional point of view. In other words, a faulty component "F" cannot disturb the correct operation of components that do not depend on the results of component "F". The same is true for communication channels. Thus, encapsulation allows mixture of certified safety-critical application subsystems and untrusted application subsystems that provide additional non-critical services, without having adverse effects on the safety-critical one. Another benefit of encapsulation is that failures can be tracked down to the responsible component more easily. Furthermore it enforces composability. In short, a system features composability, if different modules may be developed independently and if the integration of further components does not affect the correct functioning of the other components in any way.
Encapsulation is achieved by the introduction of a trusted region, which components are certified to the highest required level of criticality. All configuration data relevant for the TT-SoC architectural elements which enters the trusted region is validated by trusted components to guarantee system integrity.

Support for Resource Efficiency In embedded systems resources may be scarce and must be used efficiently. Especially the power dissipation should be minimized as embedded devices are often battery powered. Resource efficiency must be reflected in the system design. The system has to be capable to adapt dynamically to changing resource demands during run-time. Examples for this are frequency scaling or complete shutdown of unneeded components, and avoidance of redundant transmissions. However, measures taken to improve the resource efficiency must not compromise the other properties of the system. Thus, 100% reliable resource management facilities are to be used.

3.2 Structure

The structure of the TT-SoC is depicted in Figure 3.1. The central element is the NoC which connects all micro components. For the NoC, the micro components can be considered as clients using the network service. From the point of view of the NoC there is no difference between the individual micro components. The interface to the NoC is identical for all micro components.

The TT-SoC architecture, however, distinguishes between micro components dedicated to TT-SoC services on the one hand and micro components that implement the application on the other hand. The dedicated micro components are

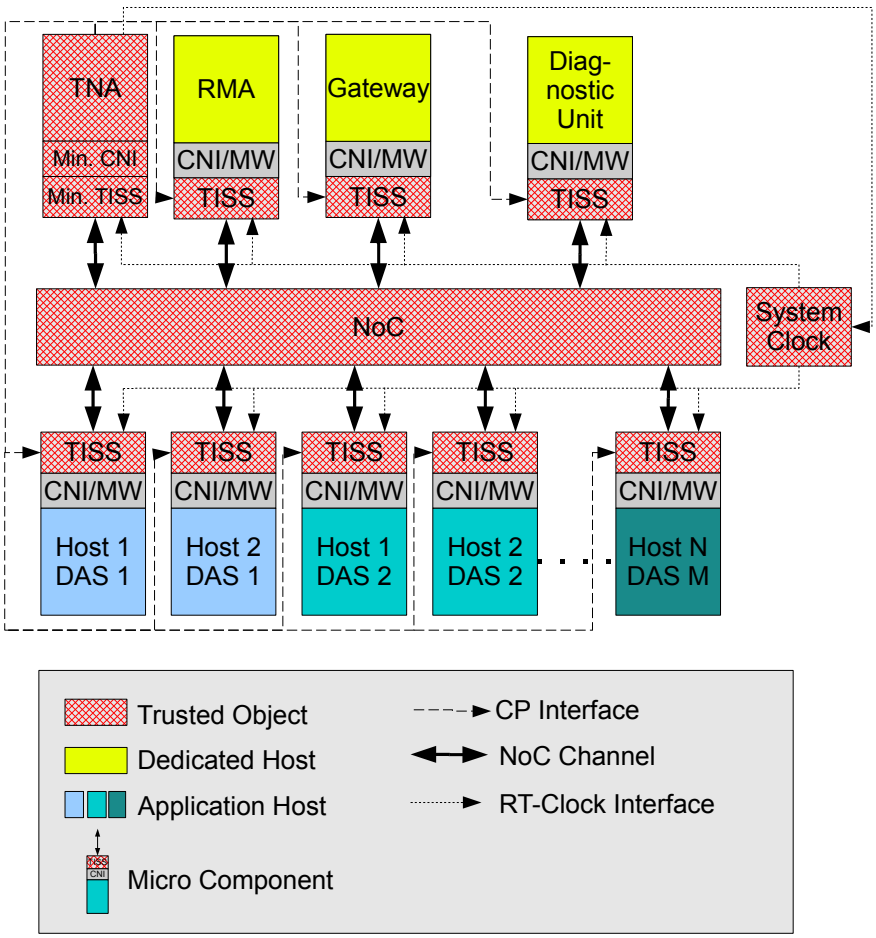


Figure 3.1: TT-SoC components overview

shown in the upper half in Figure 3.1. They comprise the Trusted Network Authority (TNA), the Resource Management Authority (RMA), the Diagnostic Unit (DU) and the Gateway. They are introduced in the sections after the micro component description. A typical micro component (as shown in Figure 3.2) is built up of three parts : the Trusted Interface Subsystem (TISS), the Communication Network Interface (CNI)/Middleware Layer and the host. They are discussed below.

3.2.1 Trusted Interface Subsystem (TISS)

The TISS is the access point of the micro components to the NoC. It has to ensure that the hosts can access the NoC exclusively during the time-slots allocated for them. Neither sending nor receiving may be permitted in all other

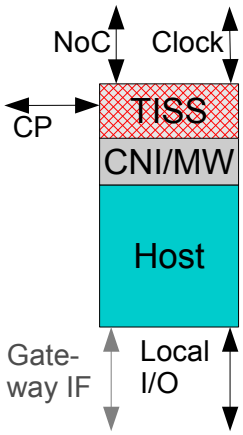


Figure 3.2: A typical micro component

time-slots. Thus, a faulty host cannot interfere with the communication of probably safety critical jobs. Furthermore, if an intruder is able to break into an unsafe micro component, he cannot eavesdrop the exchange of sensitive data between safe micro components. The active time-slots for each TISS must be configured via the Configuration and Planning (CP)-Interface of the TISS. Obviously, only a trusted component is entitled to fulfill this task. The TISS has an interface to access the Global Time Base which is compared against the preconfigured time-slots to determine the active time-slots. Since the global time is of interest to the CNI/Middleware Layer and the host, the TISS forwards the current time to the CNI/Middleware Layer.

3.2.2 CNI/Middleware Layer

The CNI/Middleware Layer provides higher level access to the NoC. Since the TISS must be trusted it has to be certified to the highest criticality level of any host in the SoC. To ease certification it is stripped down to a minimum of functionality. This lack of functions is filled by the CNI/Middleware Layer, so that access to the NoC is eased for the hosts. Depending on the host, there may be different versions of the CNI/Middleware Layer. For example, the TNA, which has to be certified as well, does not need a CNI/Middleware Layer with a rich feature set, so a minimal version suffices.

3.2.3 Host

The hosts perform the computational tasks. A host can be a processor within the SoC, an external processor or any logic circuit that can make use of the NoC. In

most cases, a host has an I/O interface to read sensor values, drive actuators or has an additional network or bus interface to connect to the outside world. However, hosts without further interfaces that are used solely for performing computational operations may be useful in some applications as well.

3.2.4 Dedicated Hosts

Dynamic resource management is accomplished mainly by the cooperation of two dedicated micro components, the Resource Management Authority (RMA) and the Trusted Network Authority (TNA). Their role is described in the following sections. Thereafter, the components Diagnostic Unit (DU) and Gateway are presented which offer additional services to the TT-SoC application subsystems.

Resource Management Authority (RMA)

The RMA is the core of the dynamic resource management. It receives resource requests from other hosts via the NoC and calculates the resource allocation for the whole SoC. As the algorithms that perform this task can get very complex, it may not be reasonable to certify the RMA, thus it cannot be guaranteed to be free of design faults. Therefore, the results must be checked for validity and for syntactical correctness. A separate micro component, the TNA, is entitled to perform this delicate task. Thus, the resource allocation computed by the RMA is sent to the TNA again via the NoC. In fact, the RMA does not need an interface other than to the NoC.

The three principal resources to be managed by the RMA are: computational resources, communication resources and power. Computational resources include I/O allocation and memory allocation. The management of communication resources consists of creating a conflict-free message schedule that complies with the requirements (bandwidth, latency and phase alignments) of all micro components in the system. Finally, power management may lower clock frequencies and/or lower core voltages to reduce power dissipation significantly, when performance is not critical. The RMA may even completely disable a micro component, in which case it must be reactivated by another micro component of the same Distributed Application Subsystem (DAS).

Trusted Network Authority (TNA)

The TNA realizes together with the NoC and the TISSs the trusted part of the resource management. It has to ensure that no invalid resource allocation may ever be activated during system uptime. For this, it has to verify that the resource

allocation received from the RMA does not violate any requirements of the system. First, collisions must not exist. For example, a sending slot may not be assigned to two different hosts. And, second, all resources needed by safety-critical application subsystems or dedicated architectural elements of the TT-SoC architecture (e.g. RMA, TNA) must be provided.
If these tests are passed, the TNA is also responsible for the establishment of the resource allocation. This is done via a separate dedicated channel to the CP-interfaces of the TISSs.

The second task of the TNA is to enable clock synchronization. The ability of the TT-SoC to synchronize to the clock of another system (which is not necessarily an SoC) allows to form clusters of systems. Since the global time-base is of integral importance to a time-triggered system as the TT-SoC, a trusted component has to perform external clock synchronization. The clocks is adjusted using rate correction to avoid discontinuities in time.

Diagnostic Unit (DU)

The DU collects status and diagnosis information from different parts of the TT-SoC. In particular, abnormal operation is recorded to locate the source of faults. All structural elements of the SoC (TNA,RMA,TISSs, hosts) can report such issues. For example, the TNA reports if the verification of the resource allocation fails. The TISSs report when a hosts violates its timing specifications (e.g. queue overflow, invalid time-slot).

Gateway

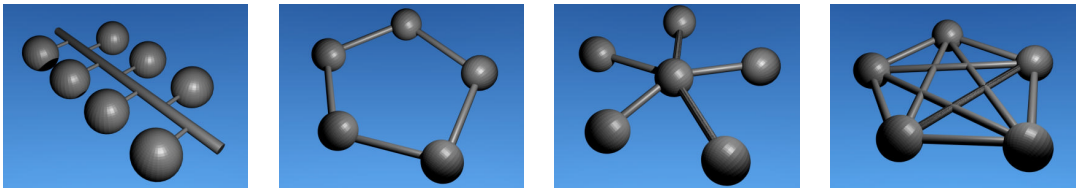
A Gateway host allows to connect the TT-SoC to other networks. In principal any network may be connected to the SoC via a gateway, but if inter-network real-time communication channels are needed, time-triggered protocols ease the interconnect and make it possible to use a minimum of buffers. Examples are TTP or Time-Triggered Ethernet. Furthermore, Gateway hosts enable interconnection of multiple TT-SoCs to form a cluster. This is necessary to construct ultra-dependable systems, which cannot be realized with only a single chip because of the relatively high soft error rate of deep submicron devices. In [Con02], Constantinescu analyzes the dependability of devices of this technology.

3.3 Network-on-Chip

The NoC provides the communication channels between the micro components. Although in an SoC it is easily possible to establish links between two micro

components outside the NoC, doing so results in loosing all benefits provided by the NoC.

The structure of the NoC is not restricted to a particular topology. Some commonly used topologies are shown in Figure 3.3.



(a) Bus (b) Ring (c) Star (d) Mesh

Figure 3.3: Commonly used network topologies

In conventional cabled networks mesh-like topologies (Figure 3.3(d)) are not always practicable. In an SoC "wires" are relatively cheap so mesh structures, which offer the highest throughput, can be realized more easily and therefore are a promising topology for NoCs.

Medium Access Control (MAC) The NoC must be real-time capable as all components of the TT-SoC. Therefore, TDMA (see Section 2.1.1), which guarantees collision-free operation, is used for all interactions with the NoC.

Communication Primitives All communication on the NoC is done using "pulsed data streams" (see Section 2.2). Pulsed data streams group multiple TDMA timeslots, called fragments, to a single message so they need not be allocated individually. The fragments are spread in such a way that long messages do not inhibit messages of very short periods.

Ports The access to the NoC from higher network layers is done via Ports. There are two different kinds of ports: Data-Link Ports (D-Ports) and Logical Ports (L-Ports). D-Ports are comparable to the ports in IP-based networks. An NoC interface has a number of D-Ports that may be assigned to individual communication channels and form the physical interface to the NoC. There may be some special ports with fixed numbers (e.g. port 0 for diagnostic data) but otherwise the assignment of port numbers to channels is arbitrary and should not be of interest to the application that is actually using the port. Applications operate on the higher level L-Ports. An L-Port identifies the semantics of the data on the channel. For instance the temperature reading of a sensor, a voltage level or the left channel of an audio stream. If an L-Port is activated, the RMA assigns

a currently unused D-Port which is then used for the transmission of the data identified by the L-Port. The advantage of this separation is that the software is independent from the low-level D-Ports. If the configuration changes, the D-Port numbers may change but the L-Port remain the same. Thus, even the relocation of the source of a channel is transparent to the receivers.

Chapter 4

Design of the Resource Management Solution

In this chapter the design of the SoC with dynamic resource management is presented. Also the resource allocation procedure and the algorithms used in the realization are explained.

4.1 Resource Management Cycle

The resource management is a permanently active periodic process. Its course of actions is depicted in Figure 4.1. In [Hub08], this resource management cycle is motivated.

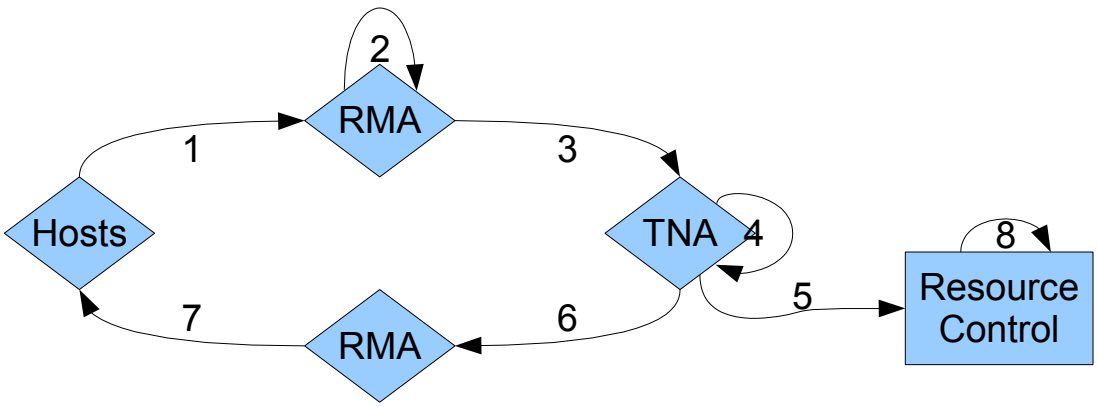


Figure 4.1: The resource management cycle

All of the micro components that dynamically allocate resources take part in the process. The other micro components may have static allocations that are always

active, or have their allocations done by another micro component of the same application subsystem.
After the completion of all eight steps a new resource allocation becomes activated. The period length of the process called reconfiguration period is a system parameter defined at system design time. It depends on the dynamics of the application and the performance of the RMA and the TNA. Furthermore, very short cycles result in a considerable "non-productive" network load, since resource management messages must be exchanged at very short intervals.

Since the TT-SoC is a time-triggered architecture for real-time systems, the send and receive instants of all messages are defined a priori. Therefore, the beginning of all steps has a fixed offset within the period, which dictates the maximum duration for all steps. Similar to the period length the instants are system parameters chosen during system design.

In the following, the steps depicted in Figure 4.1 are explained.

1^{st} **step: hosts request Resources** In the first step, the hosts have to specify the resources they currently need. Since they are not in the position to allocate or deallocate resources themselves, they send a request to the RMA.

2^{nd} **step: RMA computes resource allocation** In the second step, after the receive instant of the last resource request, the RMA calculates the new resource allocation based on the recently received requests. At that time no more resource requests can be issued for the current cycle.

3^{rd} **step: RMA transmits the resource allocation** The third step consists of sending the resource allocation to the TNA in order to have it verified and applied.

4^{th} **step: TNA verifies the resource allocation** The correctness of the results of the RMA are verified in the fourth step to make sure that no conflicts exist and that resources for critical components are allocated.

5^{th} **step: TNA writes the configuration** If the configuration proves to be correct, it is written to the configuration registers of the TISSs in step five.

6^{th} **step: TNA reports verification result** The RMA needs to know whether the allocation was correct to react on errors and to notify the hosts. The RMA may try a different allocation algorithm on a verification failure or at least output diagnostic information.

7^{th} **step: RMA notifies hosts** The hosts must be informed if their requests could be satisfied. Since there is no communication channel between the TNA and the hosts, the RMA has to notify the hosts based on the verification result from the TNA. Furthermore, it is possible that a host's resources were altered by a request from another host. In short, this step assures that every host in the system is aware of the resources it will be assigned to after the oncoming reconfiguration.

8^{th} **step: Activation of the new configuration** To avoid inconsistent intermediate configurations, the whole new allocation is activated at a predefined instant for all components.

4.2 Resource Management Authority (RMA)

The RMA implements the actual dynamic resource management. It is equipped with possibly a number of algorithms that are able to distribute the available resources to the hosts according to the resources they requested.

Tasks

Process Requests It is unpractical to realize the resource requests issued by the hosts as complete descriptions of all required resources since that would be a waste of network bandwidth. Therefore, a table of application modes, where each table entry describes the resources needed in this mode is employed. Thus, before the allocation computations can even start, the requests must be translated to detailed resource requirements.

Compute Resource Allocation After the processing, the resources can actually be assigned to the hosts.

Provision of Resource Usage Information It is not sufficient to inform the hosts if their request was successful. The knowledge how to access the resources must be included. In the implementation presented in this thesis, the hosts need to know the network ports for requested messages in order to use the network.

Interfaces

Hosts $\Rightarrow$ RMA Used to request or relinquish resources

RMA ⇒ TNA Used to transmit the resource allocation to be verified and applied.

TNA ⇒ RMA Used to receive the status of the verification.

RMA ⇒ Hosts Used to inform the hosts which resources they were assigned to and how to access them.

4.3 Trusted Network Authority (TNA)

The TNA implements the guard function to assure that no corrupted resource allocation may enter the system. This makes it possible to use dynamic resource management even in systems of applications of mixed criticality. Without the TNA, there would be the risk that a fault in the resource allocation algorithm could lead to a resource shortage in crucial components leading to catastrophic failures. Through the TNA such a scenario is made impossible as the TNA maintains a list of guaranteed resources. The TNA checks that these resources are assigned in all allocations and hinders the activation of the allocation if the check fails.

Tasks

Detect Resource Conflicts Verify that no resource allocations conflict with each other.

Detect Missing Allocations Verify that all guaranteed resources are provided.

Apply Resource Allocation Write the configuration registers to effectuate the assignment of the resources to the hosts.

Interfaces

RMA ⇒ TNA Used to receive the proposed resource allocation.

TNA ⇒ RMA Used to return the verification result.

TNA ⇒ TISS CP Used to write the resource configuration.

4.4 Scheduling Constraints

In addition to the apparent requirement that any two fragments must not be scheduled at the same phase the hardware implementation imposes three constraints.

1. The distance between two fragments must not be smaller than one bus clock tick (macrotick).
2. The distance between two fragments concerning the same host (but from different pulses) must not be smaller than MAX_EVENT_RATE. That is the time the hosts takes to switch between two pulses.
3. "Same Period Constraint": For each period a host can only process one pulse at a time (refer to Section 5.3.2 for details). This means that pulses of the same period that require attention from at least one common host may not be interleaved (i.e. fragments of one pulse cannot be placed between two fragments of the other pulse).

4.5 Scheduler for Pulsed Data Streams

Message scheduling is generally a non-trivial problem. However, with pulsed data streams the problem space becomes much larger because messages are allowed to overlap. Furthermore, while searching for a solution Constraint 3 must be satisfied and restrictions on the placement of messages defined by the system designer must be respected. This makes scheduling a very complex task. The proposed algorithm was designed to run on-chip under tight timing constraints. It cannot be expected that it is able to solve all possible scheduling problems. Nevertheless, it performs well for situations where the bus is operated well below full load.

4.5.1 Inputs

The input for the scheduling algorithm is a list of pulsed data stream definitions. Each pulse definition contains the following parameters:

Logical Port ID used to identify the pulse (irrelevant for scheduling, but must be kept for later usage).

Pulse Period determines the frequency of the pulse.

Fragment Period determines the distance between two neighboring fragments. To keep the scheduling effort manageable this value must

be specified. Alternative, possibly offline, scheduling algorithms might be able to select the fragment period dynamically since in many applications the actual fragment period is of little importance as long as the desired number of fragments is delivered within the pulse period.

Pulse Length the number of fragments. This attribute along with the fragment period determines the **Pulse Duration** which is the time interval between the start of the transmission of the first fragment and the end of the transmission of the last fragment. Figure 4.2 shows pulsed data streams in the first row and their duration in the second row.

Set of involved hosts is a bitfield that selects the Hosts acting as sender or receiver of the pulse. It is required in order to account for "Constraint 3".

Lower Bound the system designer can use this field to specify that a pulse must be scheduled after a certain phase.

Upper Bound the system designer can use this field to specify that a pulse must be scheduled before a certain phase. "Lower Bound" and "Upper Bound" control the possible phase of the first fragment. If their values are equal, the phase offset of the pulse is predefined and will not be modified by the algorithm.

4.5.2 Preparations

The algorithm inserts one message after the other into the schedule. The more messages that were scheduled the harder it gets to find places for the remaining messages. So the pulses are ordered with those most difficult to schedule first.

The ordering is based directly on the input values mentioned in the last section, but it has proved useful to calculate one further value for use in the sorting process. It models the constraint that pulses of the same period that have a common host must not overlap. If we consider pulses in this relation they can be regarded as blocks rather then pulses. Figure 4.2 illustrates this. It can be seen that there is much space for the fragments of a pulse in the first line but only very small blocks can fit in between the others. For a given period and a set of hosts the length of the blocks determines if it is easy to fit all blocks into the period. So the sort-index is the ratio length of all blocks to period length. The procedure that calculates the values can be found in Section 4.6, Listing 4.1

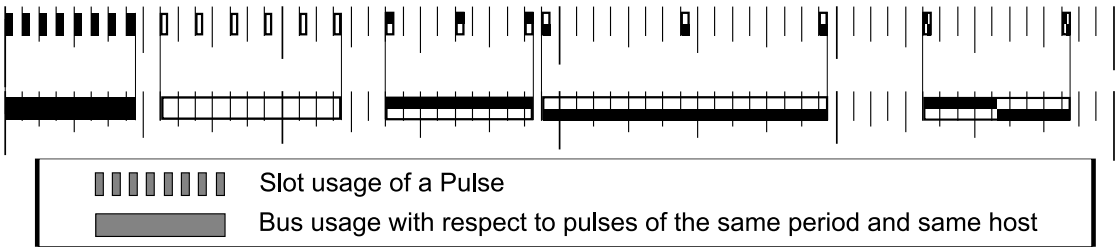


Figure 4.2: Pulses of the same period and at least one common host

4.5.3 Scheduling order

The pulses are sorted according to the following criteria:

1 Ascending fragment period
Pulses with shorter fragment periods fit harder between the fragments of other pulses, so they are scheduled first.

2 Ascending pulse period
Pulses with shorter pulse periods occur more often so it is more difficult to find a space for all occurrences.

3 Ascending upper bound
To avoid that a pulse without an upper bound takes a valid place for a restricted pulse, most restricted pulses are scheduled first.

4 Ascending lower bound
Although a high lower bound is more restrictive than a low lower bound, pulses with low lower bounds are preferred. This is because the pulse schedule is built up from left to right. Any gaps in the structure must be filled with so-called placeholders. So a pulse with a low lower bound could enable scheduling of a pulse with a high lower bound without use of a placeholder.

5 Descending sort-index
As explained in Section 4.5.2, pulses that compete with many other pulses of the same host are prioritized. Note that at this decision level the pulses have the same pulse period.

6 Descending fragment count
The last criterion is simply the pulse length since longer pulses are more difficult to schedule.

4.5.4 Algorithm Properties

The dynamic resource allocation relies on a scheduling algorithm that arranges the pulses so that all bandwidth and timing requirements can be satisfied. Since the computational effort of the scheduling algorithm is rather low, it is possible to do the calculations online during system operation. Compared to static offline scheduling it is the more flexible and less memory intensive solution.

The scheduling algorithm developed for the resource management builds up the schedule in a tree structure. The nodes of the tree are the pulses that were already assigned a phase. The leaves represent phase offsets that can be used by yet unscheduled pulses. The leaves impose restrictions to pulses that may be scheduled at the respective phase.

The algorithm starts with the single *Leaf A* that represents all slots (as shown later on in Figure 4.5). It has no restrictions except that the phase offset is zero. The first pulse is set at this location. If the lower bound of the first pulse lies above "0" a placeholder that starts at phase "0" must be inserted. More on placeholders follows later. The pulse becomes the root node and its leaves represent the fragments remaining free. The second pulse will be checked against the leaves until an appropriate leaf (=sequence of slots) is found. The order in which the leafs are tested is not arbitrary. First it is attempted to place the pulse at more restrictive leaves. If a leaf is found the pulse is inserted as a new node. If the leaf offered slots that were not used by the pulse these slots will be pointed to by the leaves of the new pulse node. Restrictions of the replaced leaf are inherited to the newly appearing leaves.

The following Section demonstrates the construction of the tree by means of an example execution.

4.5.5 Example Execution

For a better understanding an example execution is presented here. To simplify the demonstration we assume that the network is able to transmit one fragment at each time instant in the 64-bit time format. In other words one fragment every $2^{-32}s$ or 2^{32} fragments per second. We refer to those time instants as *timeslots* or short *slots* to be consistent with TDMA terminology. The number of slots of a period is simply its length divided by the slot length.

$$\frac{2^{-X}}{2^{-32}} = 2^{32-X}$$

Table 4.1 lists the properties of the pulses used in the example. Bounded pulses are discussed later on. The pulses are already ordered according to the properties mentioned above.

Number	Period (Length in slots)	Fragment Period (Length in slots)	Fragment Count	Duration in slots
Pulse 1	2^{-27} (32)	2^{-30} (4)	3	9
Pulse 2	2^{-26} (64)	2^{-29} (8)	2	9
Pulse 3	2^{-26} (64)	2^{-28} (16)	3	33
Pulse 4	2^{-23} (512)	2^{-26} (64)	2	65

Table 4.1: Pulses to be Scheduled in the Example Execution

The pulses listed in Table 4.2 are not scheduled in the example execution but they are used in discussions to demonstrate the kinds of pulses that may be placed at a specific location. They are labeled with special characters.

Number	Period (Length in slots)	Fragment Period (Length in slots)	Fragment Count	Duration in slots
*Pulse **	2^{-28} (16)	- (-)	1	1
Pulse #	2^{-25} (128)	2^{-27} (32)	3	65

Table 4.2: Pulses for Demonstrations

Figures

At each step the tree and slot allocation is illustrated. The elements building up a tree diagram are explained in Figure 4.3. The meaning of the relations will become clear later on.

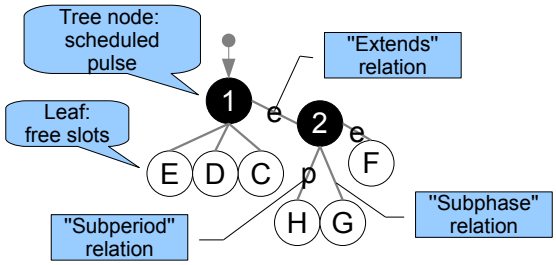


Figure 4.3: Elements of a tree diagram

The slot allocation diagrams show all slots of the longest period appearing so far. For each slot it can be seen which pulse or which leaf it belongs to. Figure 4.4 explains the different elements of those diagrams.

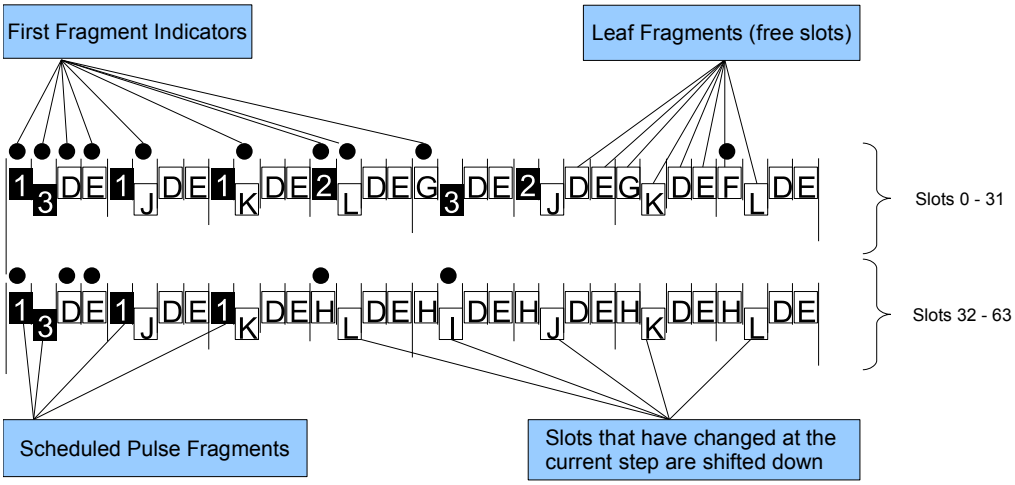
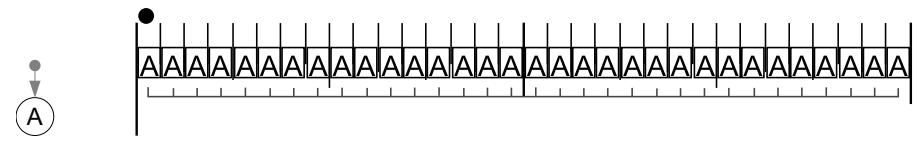


Figure 4.4: Elements of a slot allocation diagram

Initial State

The algorithm starts with the tree depicted in Figure 4.5(a). To the right of the tree, Figure 4.5(b) shows one cycle of Period 2^{-27} which is 32 slots long. It can be seen that *Leaf A* covers all phases. This means that *Leaf A* has no restrictions at all and any pulse may be scheduled there.



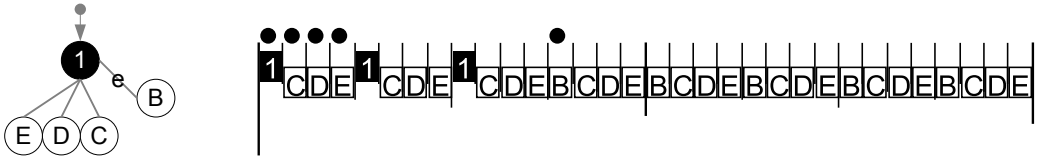
(a) Tree

(b) Slot Allocation

Figure 4.5: Initial tree and slot reservation

Pulse 1

Pulse 1 replaces the only *Leaf A*. See Figure 4.6 for the new situation. Since *Pulse 1* does not use all slots provided by *Leaf A* the new leaves *B*, *C*, *D* and *E* are created. The reason why we need 4 leaves is that the slots of a single leaf must be describable in the form of a pulsed data stream. So the tree structure reflects the structure of pulsed data streams which makes it easy to check if a pulse fits into the slots of a certain leaf.



(a) Tree (b) Slot Allocation

Figure 4.6: Slot allocation and message tree after *Pulse 1*

Leaf B Take a look at *Leaf B*. It covers the space left free because *Pulse 1* has only 3 fragments but 8 would fit into the 32 slots that make up one cycle of period 2^{-27} . This type of leaf is called "extension" because the slots are the extension of the fragments of *Pulse 1*. Note that in the tree figures (e.g. Figure 4.6(a)) "extension" leaves are connected to the right side of the pulse node and marked with an "e".

In Figure 4.6(b) we can see the pulse structure of *Leaf B*. It is 5 fragments long with period 2^{-18} and fragment period 2^{-21} , starts in slot 12 and has a duration of 17 slots.

Basically, three different types of pulses may replace *Leaf B*:

- Pulses which pulse period is longer than or equal to 32 slots, which fragment period is longer than or equal to 4 slots and which have a duration of maximal 17 slots.
  The next pulse in our example, *Pulse 2*, satisfies those requirements and will actually be scheduled at *Leaf B*'s position.

- Pulses with a pulse period of 16 slots and only one fragment. Note that the fragment period for single fragment pulses is meaningless. This case is illustrated in Figure 4.7.
  Cases like this are not handled by the algorithm, which means that even if *Pulse 2* possessed these properties it would not be scheduled there. The reason is that the scheduling of the succeeding pulses would be more complex.

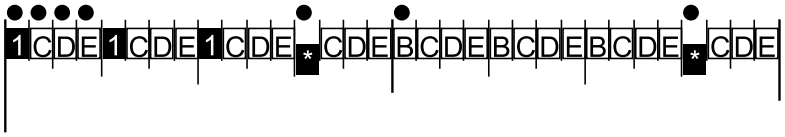


Figure 4.7: *Pulse **: short pulse period

- Pulses having a **fragment** period that is longer than or equal to 32 slots. These pulses are called "orthogonal" to *Pulse 1* because their

fragments are orthogonal to the fragments of *Pulse 1* if the period cycles are arranged in rows.
This arrangement is exemplified in Figure 4.8. Each row represents one cycle of period 2^{-18} . All four rows together form one cycle of period 2^{-16} (4x32 = 128 slots), the pulse period of *Pulse #*.

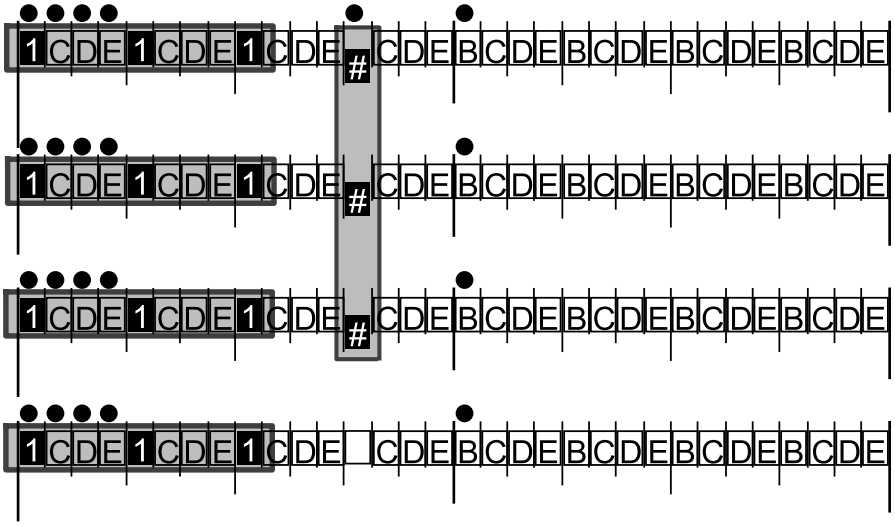


Figure 4.8: Orthogonal *Pulse #*

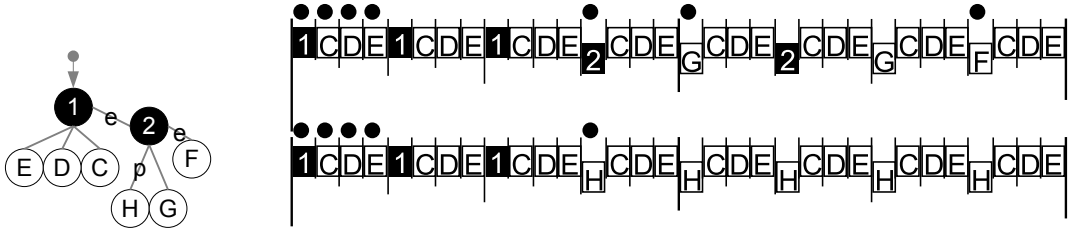
Leaves *C, D, E* The leaves *C*, *D* and *E* fill the gaps between the fragments of *Pulse 1*. They all have similar properties, only the phases (1, 2, 3) are different. They appear because the fragment period of *Pulse 1* does not cover all phases. They are called "subphase"-leaves.

The big difference to *Leaf B* is that no fragments of *Pulse 1* can collide with a pulse that is to be scheduled at one of the "subphase"-leaves. The consequence is that there is no restriction on the pulse period and the pulse duration as opposed to *Leaf B*. The only requirement is that the fragment period must be longer than or equal to 2^{-30} (4 slots) which is guaranteed by the ordering of the pulses (cf. Section 4.5.3).

Pulse 2

Pulse 2 is first checked against *Leaf B*. It is more restrictive than the other leaves so it should be used if possible so that the other leaves are saved for more difficult cases.

Pulse 2 actually fits at *Leaf B* and replaces *Leaf B* in the tree. The pulse period and the fragment period of *Pulse 2* are longer than those of *Pulse 1* and the pulse is shorter than 20 slots, so not all slots offered by *Leaf B* are used. These slots are covered by creating new leaves as children of *Pulse 2*. These leaves are named *F*, *G* and *H*. The new tree and slot allocation can be found in Figure 4.9. The pulse period of *Pulse 2* is the double of the pulse period of *Pulse 1*. So the pulse occurs only once in two cycles of *Pulse 1*'s period.



(a) Tree (b) Slot Allocation

Figure 4.9: Slot allocation and message tree after *Pulse 2*

Leaf F Similar to the "extension" *Leaf B* of *Pulse 1*, *Leaf F* is the "extension" leaf of *Pulse 2*. However, the phase of *Leaf F* is very near to the end of period 2^{-19} so only pulses with one fragment or "orthogonal" pulses (cf. Figure 4.8) can fit at its place. The pulse period of such one fragment pulses must be at least 64 slots long (the pulse period of *Pulse 2*). Likewise the fragment period of "orthogonal" pulses must not be shorter than 64 slots.

Leaf G *Pulse 2* has only one "subphase" leaf *G*. Unlike the "subphase" leaves of *Pulse 1* (*C*, *D* and *E*) *Leaf G* restricts both duration and pulse periods, because it inherits the properties of *Leaf B*.

Note: the number of "subphase" leaves calculates as:

$$\frac{fragment\ period\ length\ of\ Pulse\ 2}{fragment\ period\ length\ of\ Pulse\ 1} - 1.$$

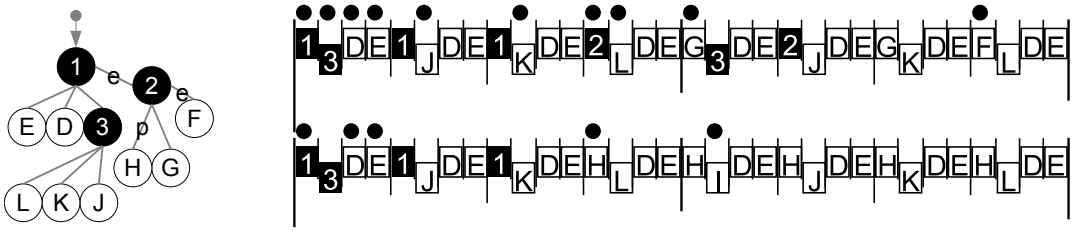
Leaf H The first "subperiod" leaf is *Leaf H*. It comes to existence because *Pulse 2* occurs only in every second cycle of the dominating period. So the slots of *Leaf B* in the unused other cycle have to be covered by a new leaf: *Leaf H*.

Note: the number of "subperiod" leaves calculates as:

$$\frac{period\ length\ of\ Pulse\ 2}{period\ length\ of\ Pulse\ 1} - 1.$$

Pulse 3

The duration of *Pulse 3* (33 slots) is too long for leaves *F*, *G* and *H*. So *Leaf C* is chosen for the placement of *Pulse 3*. Figure 4.10 shows the updated tree and slot allocation.



(a) Tree (b) Slot Allocation

Figure 4.10: Slot allocation and message tree after *Pulse 3*

The leaves of *Pulse 3* are obtained like those of *Pulse 1* only the different fragment period of *Leaf C* must be taken into account. The result is the "extending" *Leaf I* and three "subphase" leaves *J*, *K* and *L*.

Pulse 4

The last pulse in this example is very easy to fit since its **fragment** period is longer than the **pulse** periods of the other pulses. So it fits even at the most restrictive *Leaf F* because it is "orthogonal" to the prior pulses. This can be seen in Figure 4.11.

Like every pulse that does not use up the whole period length, *Pulse 4* has an "extending" *Leaf M*. Furthermore it has a "subphase" *Leaf N*.

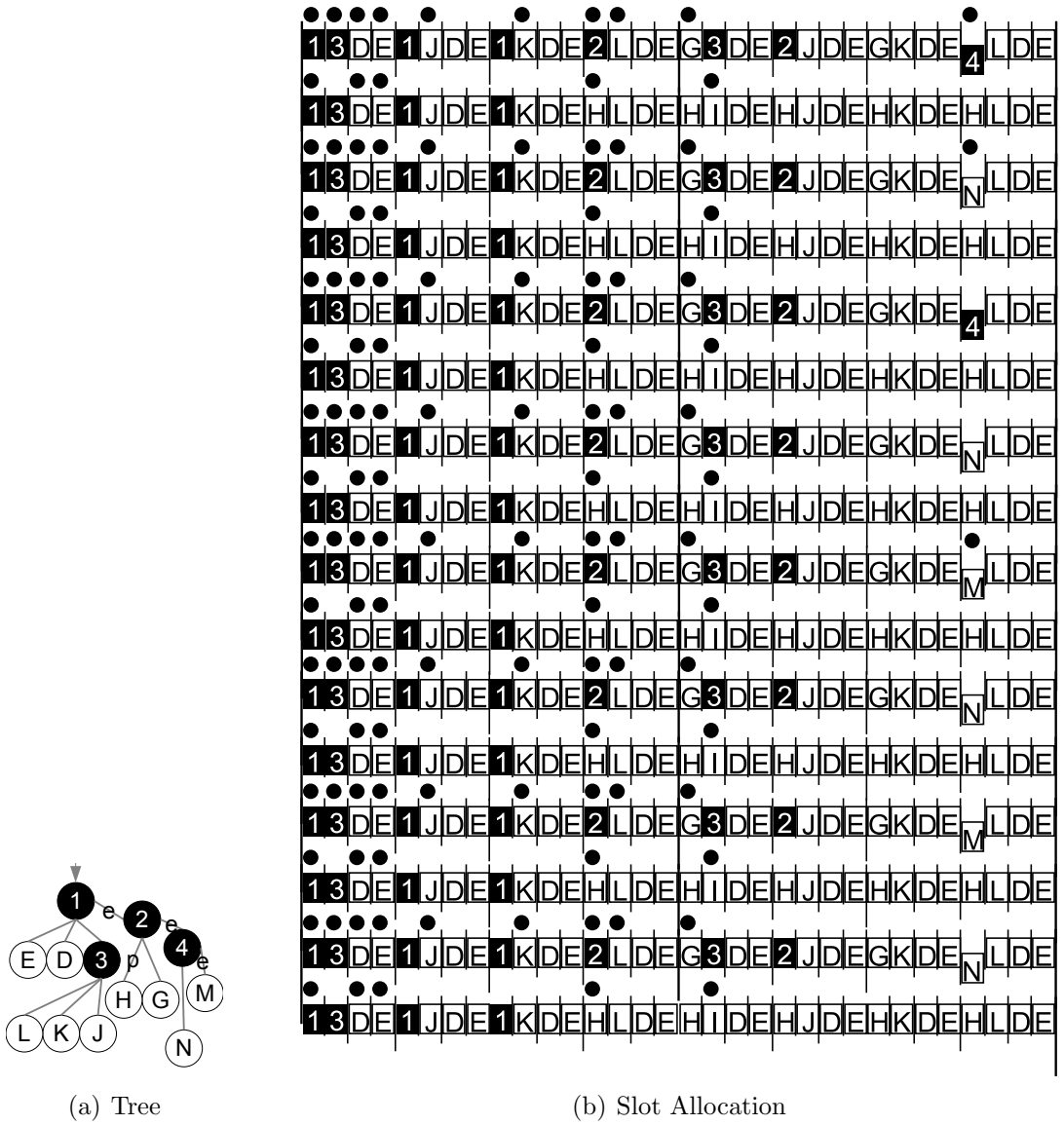
4.5.6 Restricted Pulses

Upper Bounds

When moving down the tree the phase offsets increase. Figure 4.12 illustrates this for our example. The tree from Figure 4.11(a) was redrawn but now the node elements are labeled with the phase offsets.

The reason is trivial. When a leaf is replaced, the position of new leaves ("extending","subphase" or "subperiod") is always after the first fragment of the original leaf.

This property is used for scheduling of pulses restricted by an upper bound. To find a location for such a pulse the tree is traversed as usual but if a node with a



(a) Tree (b) Slot Allocation

Figure 4.11: Slot allocation and message tree after *Pulse 4*

phase offset higher than the upper bound is reached the search in that branch of the tree is aborted.

Lower Bounds

The lower bound is checked when an appropriate leaf is found. If the lower bound is higher than the phase of the leaf the pulse cannot be placed immediately there. But a placeholder may be inserted and the pulse can be placed at its "extending"

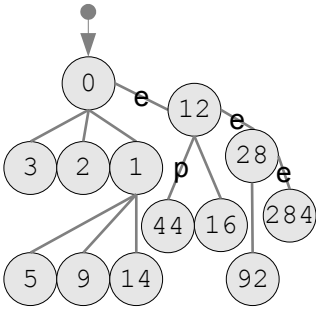
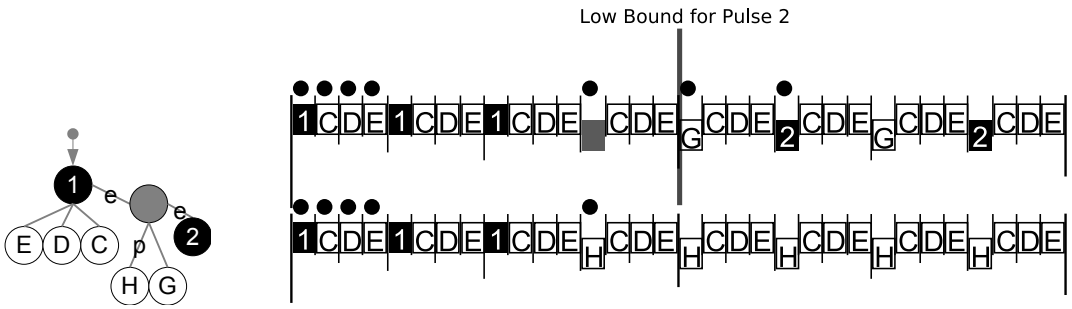


Figure 4.12: Phase offsets of the tree elements

leaf.

Assume that *Pulse 2* of our example is bounded within $16 \leq phase < 64$. Then, it cannot be placed at phase 12 as before. But if we put a placeholder at phase 12 we may use its "extending" leaf for *Pulse 2*. The outcome is found in Figure 4.13.



(a) Tree (b) Slot Allocation

Figure 4.13: Use of a placeholder

This time, the "subphase" and "subperiod" leaves are child nodes of the placeholder because its fragment period and pulse period are equal to those of *Pulse 2*. *Pulse 2* does not even have an "extending" leaf because there is no more space left in the dominating period.

For pulses with lower bounds the tree is traversed normally but if a suitable leaf for the pulse is found which violates the lower bound, the leaf is recorded and the traversal is continued. If there is a leaf that satisfies the lower bound the pulse is scheduled there and the placeholder solution is discarded. If not, the first placeholder solution in the traversal is taken and subsequent solutions are ignored.

4.5.7 Constraint 3

So far Constraint 3 (see Section 4.4) was not taken into consideration. It demands that two pulses with the same pulse period and at least one common host may not be interleaved.

Constraint 3 necessitates an extra check before the phase offset of a pulse can be set. To be able to efficiently perform this check an ordered set of the previously scheduled pulses is required for each period. The data structure proposed for this purpose are Red-Black Trees which guarantee a worst case time complexity of $\mathcal{O}(\log n)$ for both insertion and search of elements. Red-Black Trees are balanced binary trees where nodes are colored either red or black. The colors are subject to two rules, which are also called balance conditions:

1. A red node has a black parent.

2. Every path from the root to a leaf contains the same number of black nodes.

Due to these conditions the longest path from the root to a leaf is at most twice as long as the shortest path from the root to another leaf. Red-Black trees are less strictly balanced than AVL-Trees. Therefore, insertion and deletion of nodes is generally faster with Red-Black trees, but searching is generally faster with AVL Trees.
More on Red-Black Trees and an implementation example can be found in [Hin99].

If there is a conflicting pulse with the same period and at least one common host which would overlap, it is attempted to place the pulse after the conflicting pulse. This process may be repeated until the end of the period is reached or the upper bound of the pulse is exceeded. However, if the pulse fits after the conflicting pulse(s) a placeholder may be inserted so that the pulse may be placed there. But similar as with the lower bound, the placeholder possibility is recorded and only exploited if no other position can be found.

4.5.8 Static pulses

As mentioned above, the phase offset of a pulse can be static. Especially pulses to or from a gateway connecting to another time-triggered network (e.g. Time-Triggered Ethernet [KAGS05]) can take advantage of static pulses to reduce memory requirements for buffering and to preserve real-time characteristics.

It is difficult to integrate static pulses into the dynamic tree structure employed by the algorithm. The problem is that static pulses must be placed first so that its

slots are not taken by other pulses. However the algorithm relies on the ordering of the pulses by ascending fragment period.

In the solution presented here placeholders are used to reserve the slots of static pulses. Usually the placeholders require more bandwidth than the represented static pulse, because a part of the slots must be reserved for preservation of the tree structure. Therefore, static pulses should be avoided if possible.

For example lets take the pulse configuration from Section 4.5.5, Table 4.1. This time assume that *Pulse 2* is static with phase offset 9. Figure 4.14 shows how *Pulse 2* is represented in the tree structure.

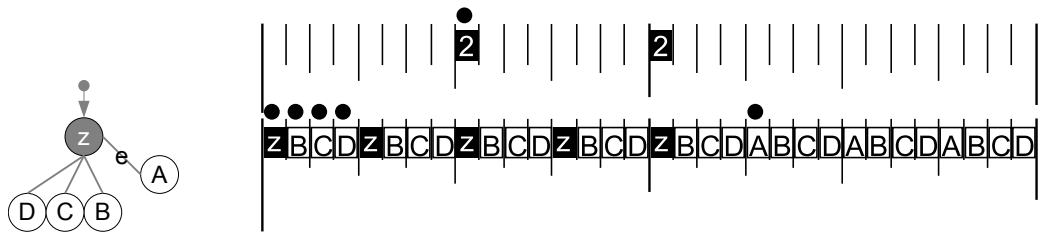
(a) Tree (b) Slot Allocation

Figure 4.14: Integration of a static pulse with phase offset 9

Pulse x is required for the root and *Pulse y* actually reserves the fragments of the static pulse shown in the upper line.

The scheduling of the succeeding *Pulse 1* is restricted because *Pulse y*'s fragment period is longer than that of *Pulse 1*. So the child nodes of *Pulse y* cannot be used for *Pulse 1*. However, leaves *A*, *B* and *C* can be used.

If we assume a fixed phase offset of 8 for *Pulse 2*, the situation depicted in Figure 4.15 is reached. This time only one placeholder *z* is needed, but with a higher fragment period. Otherwise there would be no place available where *Pulse 1* may be put.



(a) Tree (b) Slot Allocation

Figure 4.15: Integration of a static pulse with phase offset 8

4.6 Listings

This section includes the code listing for the preparation prior to scheduling from Section 4.5.2.

```
1  analyzePeriods(PulseType pulses[1..n])
2  {
3    foreach(period p)
4    {
5      load[1..number_of_hosts] = 0;
6      // calculate load on individual hosts
7      // for the current period value:
8      foreach(pulse m of period p)
9      {
10       foreach(host h)
11       {
12         if h sends m or h receives m then
13           load[h] = load[h] + m.length
14       }
15     }
16     // the sum of the load on concerned hosts
17     // is the sort-index
18     foreach(pulse m of period p)
19     {
20       m.sort_index = 0
21       foreach(host h)
22       {
23         if h sends m or h receives m then
24           m.sort_index = m.sort_index + load[h]
25       }
26     }
27   }
28 }
```

Listing 4.1: Computation of the usage of the pulse periods for use in sorting

4.7 Verification

The most challenging task of the resource allocation verification is to verify the conflict free network operation. The verification is done by checking each pair of pulses for a collision. The structure of pulsed data streams complicates collision detection. It is not possible to decide whether two pulses collide only by comparing start and end times of the transmission. The next Section explains which tests are necessary to find out whether two pulses conflict.

4.7.1 Verification Tests

Figure 4.16 shows the flow-chart of the verification algorithm.

A description of the individual tests follows:

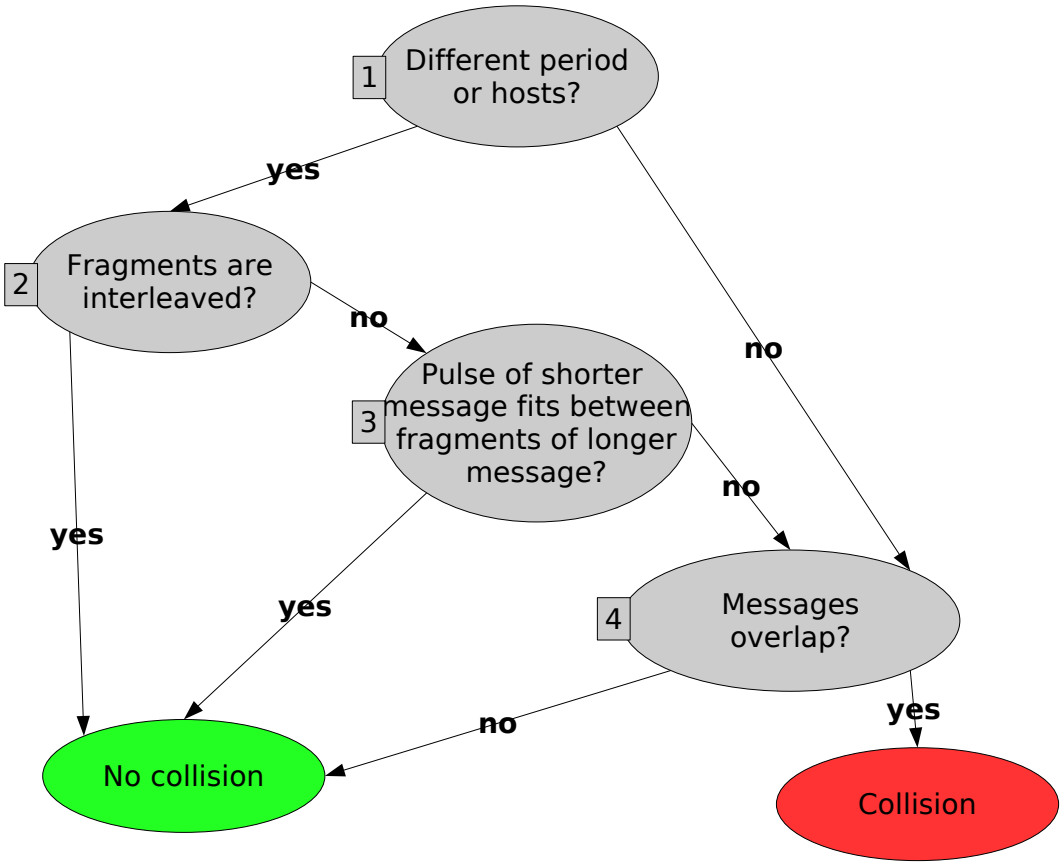


Figure 4.16: Collision detection between two pulses

1 First, it must be tested if "Constraint 3"(Section 4.4) applies in which case the next two tests are skipped, because the two pulses must not overlap.

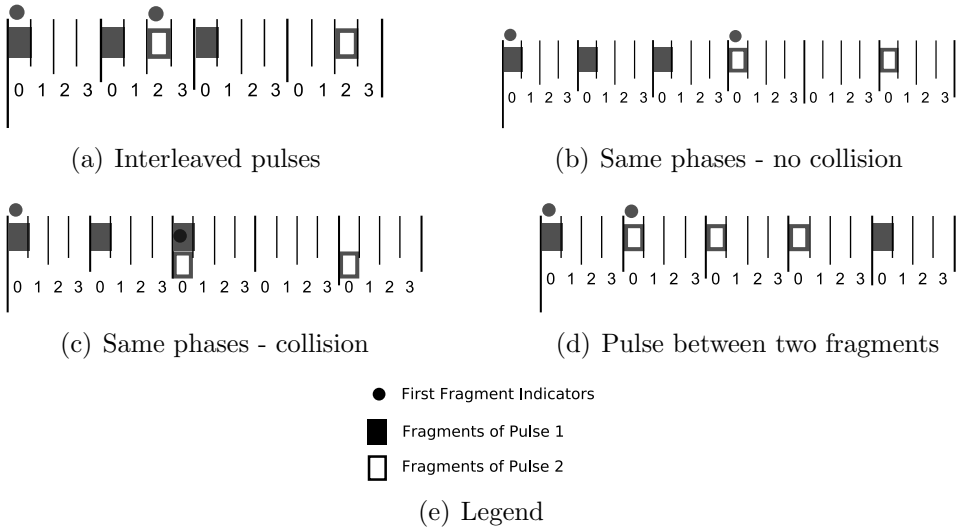
2 If the pulses may overlap it is checked whether the pulses are aligned such that a collision is impossible. The test is performed by projecting the phase offset of both pulses into the smaller fragment period. For instance, in Figure 4.17(a) the projected phase offset of all fragments of "Pulse 1" is 0. The fragments of "Pulse 2" project to phase 2. So the test is passed and there cannot be a collision.

   In Figures 4.17(b) and 4.17(c) both pulses are projected to the same phase. In the first case there is no collision while in the second there is one. So further tests are required.

3 If the pulses may overlap another situation (depicted in Figure 4.17(d)) must be checked for. The whole "Pulse 2" fits between two fragments

of "Pulse 1". The projected phases are the same so test 2 fails. Test 4 ensures that the pulses do not overlap thus fail for the given scenario. So test 3 is required to detect cases as the one shown in Figure 4.17(d) as correct.

4 This is the most complex test. Its steps are illustrated in Figure 4.7.1. To find out if the pulses overlap, the start phase of the pulse with longer period length is projected into the shorter period (Figure 4.18(b)). Then, the ending phases of both pulses are calculated by adding the duration of the respective pulse (Figure 4.18(c)). The ending phases may lie outside the period, but from this representation it is easy to tell whether the pulses overlap. Obviously, the ending phase of the first pulse must be before the starting phase of the second pulse (marked "a" in Figure 4.18(c)). Also, the ending phase of the second pulse must be before the start phase of the next occurrence of the first pulse (marked "b" in Figure 4.18(c)).



(a) Interleaved pulses

(b) Same phases - no collision

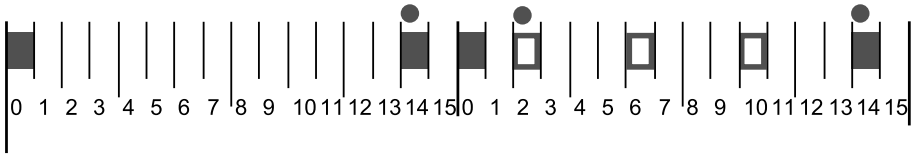
(c) Same phases - collision

(d) Pulse between two fragments

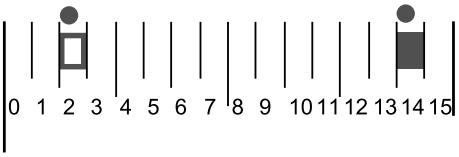
(e) Legend

Figure 4.17: Possible pulse relations

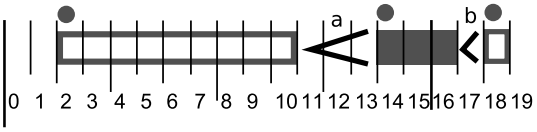
This algorithm is independent of the scheduling algorithm. On the one hand this is important if a failure causes the scheduling algorithm to work out of specification. On the other hand different scheduling algorithms can be used without modification of the TNA.



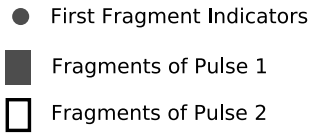
(a) Pulse 1 and Pulse 2



(b) Projection of Starting Fragments



(c) Calculation of Ending Phases and Comparision



(d) Legend

Figure 4.18: Steps performed during test 4

Chapter 5

Implementation

5.1 Environment

Since FPGAs enable testing of hardware prototypes, the SoC components are custom logic cores written in VHDL. An FPGA makes it possible to synthesize logic circuits described in a hardware description language like VHDL. Different hardware approaches can be tested very easily. When the hardware design is stable and a large volume of this type of hardware is needed the design may be realized as an Application Specific Integrated Circuit (ASIC). ASICs are optimized to the very function they have to perform and thus offer more performance than FPGAs.

5.1.1 Hardware

As hardware platform for the SoC the Nios II Development Board Stratix II Edition was chosen. The heart of the board is the Stratix II EP2S60 FPGA chip which hosts the whole TT-SoC. The board provides the resources necessary to build an instance of the TT-SoC architecture with 8 micro components. A detailed description of the board can be found in [Alt07b]. Table 5.1 sums up the most relevant features for the implementation provided by the board. Figure 5.1 gives a picture of the board.

Static RAM	2 MB	
Dynamic RAM	32 MB	
Logic Elements	48352	
Memory Bits	2544192	
I/O	8 LEDs,	2 7-Seg digits, 4 buttons, 1 two line LCD

Table 5.1: Stratix II EP2S60 features

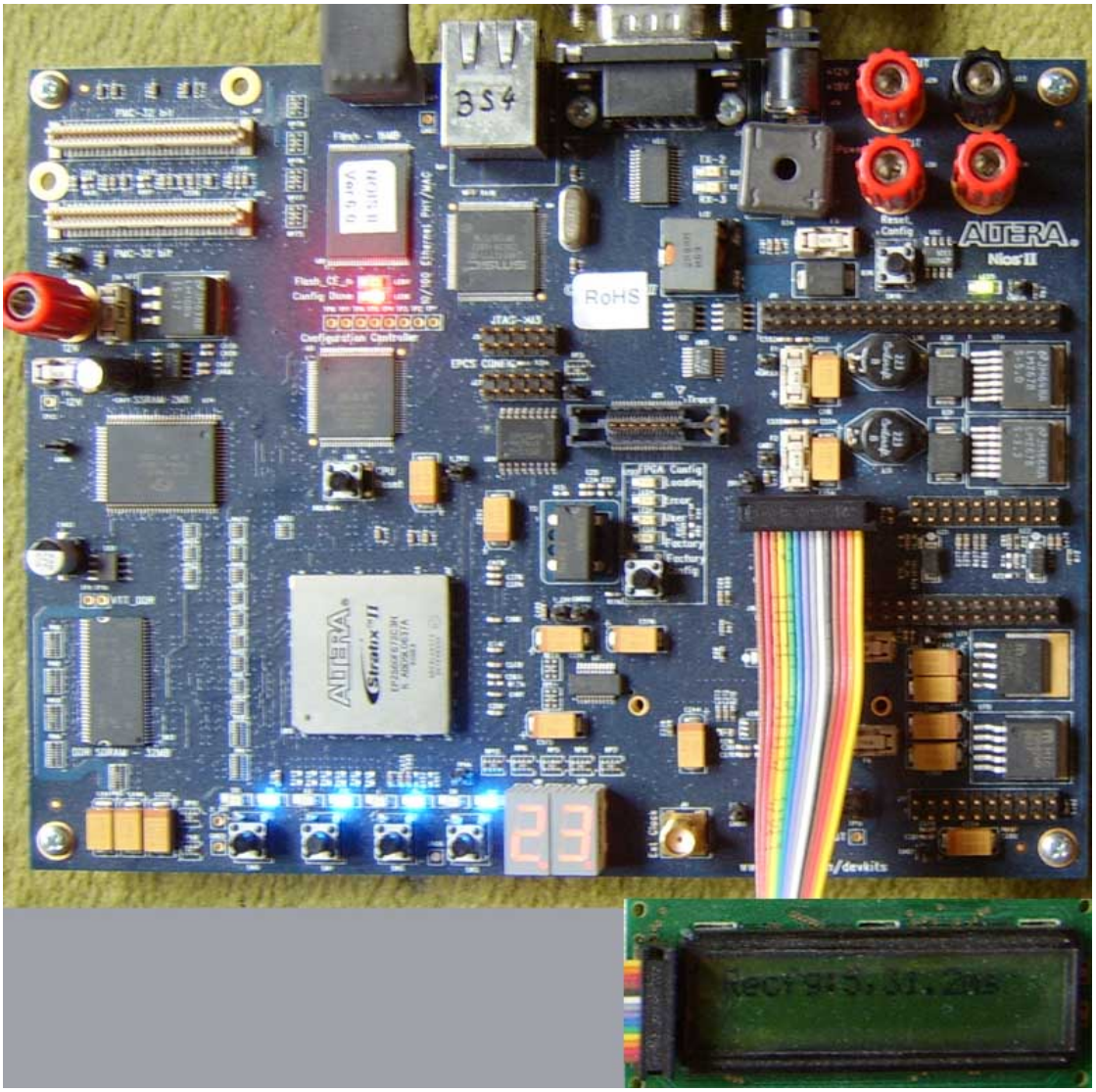


Figure 5.1: The Nios II Development Board, Stratix II (EP2S60) RoHS

5.1.2 System-on-Chip Implementation

In Figure 5.2 the different hardware modules that form the SoC and their interconnections are illustrated. The core was generated using Altera's SOPC Builder and consists of 8 Nios processors along with modules driving the peripherals. Cores (RMA and application hosts) access the NoC via the Transport Layer (TL) Interface. For best portability and extensibility this interface conforms to the Open Core Protocol (OCP) specification, a comprehensive, bus-independent and configurable interface between Intellectual Property (IP) cores and on-chip communication subsystems. The specification is available in [OCP05]. The Nios processors

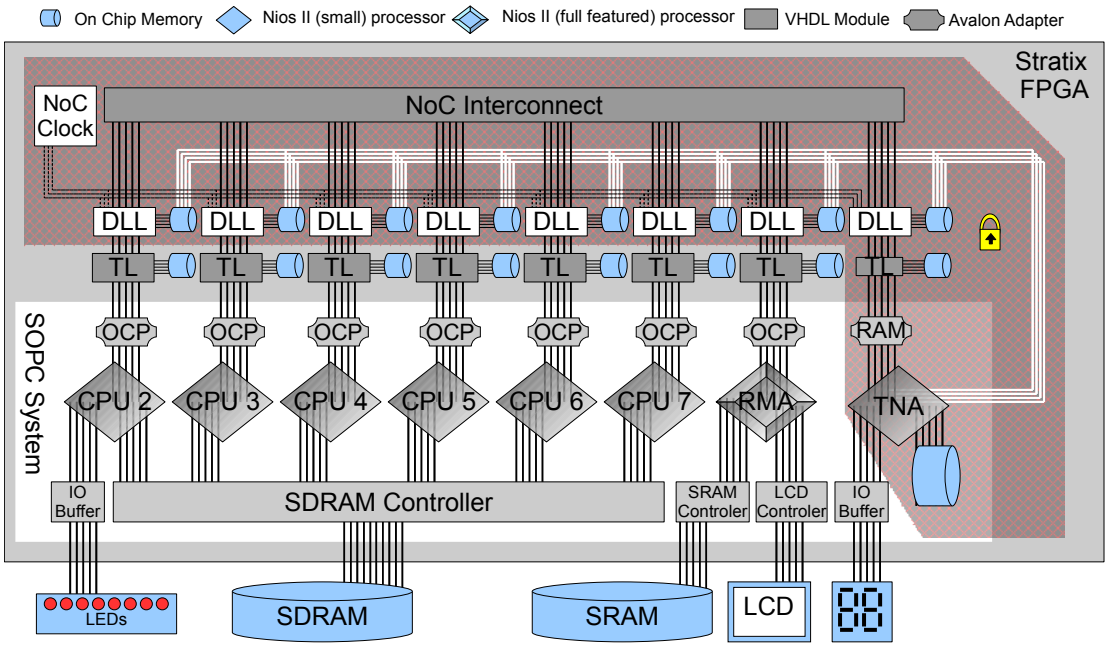


Figure 5.2: Hardware modules overview

provide only the Altera specific Avalon interface ([Alt07a]) which necessitate OCP-to-Avalon adapters between CPUs and TLs. The TL allows high-level access to the NoC. It buffers the message data and forwards it to the Data Link Layer (DLL). More detailed information about the TL can be found in Section 5.3.3.

The DLL (Section 5.3.2) is the entry point to the trusted area (in the figure marked by the lock) which can be assumed to be free of design errors. It grants access to the network only for the pulses pre-configured by the TNA. Each DLL has access to the Global Time Base of the SoC to coordinate network access. The NoC clock maintains the 64-bit wide chip-global time. For instance, if a host needs timestamps to measure intervals or simply wants to read the current clock it may obtain a snapshot of the time via the 32-bit DLL interface.

The NoC interconnect is a kind of demultiplexer that selects the data from the sending DLL and distributes it to all DLLs. However, only those that were configured to receive the corresponding pulse forward the data to the associated TL.

The TNA has very limited communication requirements. It has to receive one pulse from the RMA that carries the resource allocation. And, after checking the resource allocation and possibly reconfiguring the TISSs, it has to emit a pulse (in fact just a single fragment) to signal success or failure to the RMA. Therefore, a downscaled version of the TL suffices and is by the way easier to certify. This TL has a RAM interface which is accessed via an Avalon-to-RAM adapter since

Nios processors do provide only an Avalon interface. The DLL is the same as for application hosts and the RMA. In addition to the network access the TNA has direct access to the configuration memory of all DLLs. This is indicated in Figure 5.2 using the white bus lines.
The TNA has its own 18 KBytes on-chip memory because the TNA program size is only about 10 KBytes. The FPGA memories are dual-port capable, so the instruction and data bus operate on different ports to allow fast parallel access on both busses.

The RMA uses the static RAM for storage of program code and data, because it offers very good performance and on-chip memory is saved for the network layers namely DLL and TL.
Since memory is a scarce resource and the test application needs not run under real-time constraints the dynamic RAM is shared by the application hosts. Thus, performance on the hosts is degraded, but valuable on-chip FPGA memory blocks and the on-board SRAM are saved for RMA, TNA and the NoC so the dynamic resource management can be done efficiently.

Table 5.2 summarizes the FPGA resource usage.

	DL	TL	TL Small	Nios	Nios RMA	Total	(%)
ALUTs	1800	1200	85	870	1783	38294	(75%)
ALMs	1300	760	227	577	1151	24127	(55%)
Memory KB	16384	71936	16384	9216	69504	939776	(37%)

Table 5.2: FPGA resource usage

5.2 Time and Durations

5.2.1 Time format

Time on the SoC is represented as 64-bit values similar to the Network Time Protocol (NTP) time format (Figure 5.3). The difference is that NTP is per definition based on Coordinated Universal Time (UTC) which is not a requirement in this implementation. Thus, if no external time synchronization is performed, the seconds are counted from the system start. The horizon is approximately 136 years which should suffice for embedded applications. If larger values are needed they may be handled by application software. The theoretic granularity of the format is 232.8 ps. However, the achievable resolution of the on-chip clock implementation is only 14.9 ns. Therefore the last six bits (denoted “dead bits”) are always set to “0”.

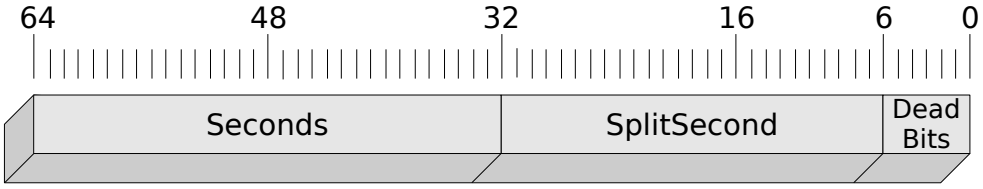


Figure 5.3: 64-bit time representation

5.2.2 Periods

Many mechanisms on the SoC (especially networking) have periodic behavior. To reduce complexity of the calculations and operations periods are restricted to negative powers of 2 of a second. This allows to encode the period lengths in 5-bit integers. See Table 5.3 for the values and meanings.

0	1	s	**8**	3.91	ms	**16**	15.26	us	**24**	59.6	ns
1	500	ms	**9**	1.95	ms	**17**	7.63	us	**25**	29.8	ns
2	250	ms	**10**	976.56	ms	**18**	3.81	us	**26**	14.9	ns
3	125	ms	**11**	488.28	us	**19**	1.91	us	**27**	7.45	ns
4	62.5	ms	**12**	244.14	us	**20**	953.67	ns	**28**	3.73	ns
5	31.25	ms	**13**	122.07	us	**21**	476.84	ns	**29**	1.86	ns
6	15.62	ms	**14**	61.04	ns	**22**	238.42	ns	**30**	0.93	ns
7	7.81	ms	**15**	30.52	ns	**23**	119.21	ns	**31**	0.47	ns

Table 5.3: Theoretical Period Durations

5.2.3 Phase Offset Alignment

A periodic action is defined by its period (one of the 32 period values) and the phase offset. The bit length of the phase offset depends on the period length. For period 0, all 26 bits are significant (excluding "dead bits"), period 2 is 4 times shorter and thus needs only 24 bits. Phase offsets are stored in 32-bit words. There are two ways how to store the offset values, which are compared in Figure 5.4:

Left-Aligned

The MSB of the 32-bit word is the MSB of the offset value. This representation does not depend on the time format. The 32-bit value can be interpreted as the fraction of the period. For example an MSB of "1" and all others bits "0" denotes the phase exactly in the middle of a period cycle, for any period length. This

representation can be used to specify relations between phases independent of the period length.

Right-Aligned

The LSB of the 32-bit word is the LSB of the offset value (without dead bits). This representation is closely related to the time-format. Offset values of different periods are directly comparable. For internal calculation such as pulse scheduling this representation is better suited.

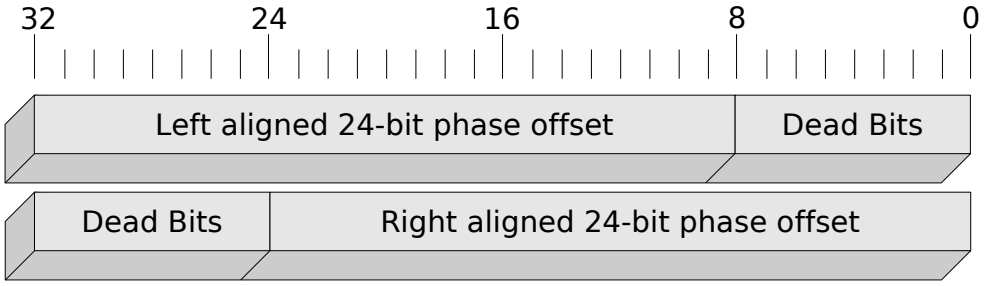


Figure 5.4: Left vs. right alignment

5.3 Network-on-Chip Implementation

The NoC for which dynamic resource management is developed, is based on the collaborative work of Gerhard Engleder and Roman Seiger. Their master theses focus on the NoC design. Gerhard Engleder has designed the system clock (see Section 5.3.1) and the DLL (more in Section 5.3.2). Roman Seiger researched the TL (refer to Section 5.3.3). Both layers were implemented and tested on Altera Cyclone II Development Boards.

In this implementation the NoC is realized as a single channel bus which means that at one instant at most one fragment is permitted to be in transmission. The basic communication type are pulsed data streams because they make it possible to interleave the transmission of messages of very long periods and messages of very short periods.

The bus structure, because of its simplicity, is a good starting point for research on the topic of dynamic resource management. Later, when the design has proved its validity, more complex topologies can be investigated.

5.3.1 Clock and Timing

On Cyclone II and Stratix II FPGA technology the NoC modules (TL and DLL) can be run at a clock frequency of 75 MHz (a period length of 13.33ns). This frequency drives all components of the SoC, except for the DDR-RAM controller that needs to be run at 100 MHz.

The clock module maintains the global time that uses the binary SoC time format with 6 dead bits. The resulting frequency of the clock is 67.11 MHz (2^{26} Hz) which corresponds to a clock resolution of 14.9 ns. A tick of this clock is called a Microtick. The NoC needs 4 Microticks (equals one Macrotick) to process a fragment of a message. So, the maximum frequency on the network is 16.77 MHz (59.6 ns). Since a fragment has a size of 128 bits the achievable transfer rate totals 5.59 Gbit.

5.3.2 Data Link Layer (DLL)

The DLL provides low level access to the preconfigured pulsed data streams of the NoC. It signals the higher layer when a fragment has arrived or when a fragment can be sent, which port the fragment belongs to and the number of the fragment within the pulse. Furthermore, it relays the data of the fragments on the assigned time-slots. The DLL does not use a buffer, instead it logically connects the output wires from the TL to the input wires of the NoC interconnect on the appropriate time-slots. The full description of the DLL can be found in the Gerhard Engleder's master thesis [Eng07].

To better understand the constraints imposed on the scheduling of the pulses, which have been presented in Section 4.4 and the background behind the TISS Configuration interface described in Section 5.5.8, the implementation of the DLL is briefly explained here.

A hardware module that reads the properties of the first pulse of a period and waits until the real-time clock reaches the phase offset of the pulse exists for each of the possible pulse periods. At this point the receive or send operation for the first fragment is triggered. Then, the fragment period length is added to the phase offset to obtain the time of the second fragment where the next action is taken. This is repeated until the last fragment was handled. Afterward, the properties of the next pulse in the list are retrieved. Since this process is periodic, the pulse list needs to be stored in a cyclic list.

It is obvious that this system cannot handle overlapping pulses on the same DLL and that the pulse lists must be ordered. In addition, due to the relatively high hardware cost of these modules only 10 out of the 32 period lengths defined in Section 5.2 are implemented. Choosing these 10 periods is in the responsibility of

the TT-SoC system designer. The values are declared as constants in an VHDL source file where they are easily accessible. Thus, changing them requires to rebuild the FPGA image.

Table 5.4 lists the periods used throughout the development of the dynamic resource management system along with a description of what they were needed for.

Period	Length	Used for
0	1 s	Early tests
1	500 ms	
2	250 ms	
4	62.5 ms	Reconfiguration periods
5	31.25 ms	
6	15.62 ms	
10	976.56 ms	More advanced timing tests
15	30.52 ns	
17	7.63 us	
19	1.91 us	

Table 5.4: Implemented Period Durations

5.3.3 Transport Layer (TL)

The function of the TL is to provide an interface to the NoC that can be used conveniently by a host implemented on a microprocessor. The TL buffers all message data and is capable of raising interrupts on network events. Via the TL the configuration memory of the DLL can be accessed read-only by the host. Furthermore, the TL effects the system mode settings. For example it asserts the processor reset signal if the watchdog period expires without an update of the host's lifesign.

Apart from the network and resource management functions already mentioned, the TL offers access to the real-time clock. It can be configured for a periodic or single-shot timer interrupt using the global time-base accessible via the DLL. Many more network features that are too specific in this context and implementation details can be found in Roman Seiger's master thesis [Sei07].

5.3.4 Pulse Selection

The RMA comprises several *resource agents*, each responsible for managing the application mode requests of one particular DAS. Thus, after reception, the ap-

plication mode requests from the jobs of a DAS are forwarded to the respective resource agent.

For each job of the DAS the resource agent calculates or looks up in a table the 12-bit wide *activation-bitfield* of the active pulse groups. Each bit represents a particular group of pulses, e.g. the group of audio streams to possibly four speakers, that have to be scheduled if the bit is set. The pulse definition (see Section 5.5.2) contains a similar 12-bit *membership-bitfield* to determine to which group or groups the pulse belongs to. If in the *activation-bitfield* any of the groups that a pulse belongs to is set the RMA will schedule the pulse. This is the case if the bitwise AND of both bitfields is non-zero.

Guaranteed pulses are always active. Since the RMA is not realized as a trusted component (otherwise its functionality could be directly moved to the TNA) it does not make sense to allow deactivation of *guaranteed pulses*. The deactivation could lead to the problem that while the pulse is inactive its time-slots are taken by another pulse and reactivation of the pulse is no more possible, which must not happen for guaranteed pulses.

Of course pulses of this type can be realized as sporadic pulses, so that data is only sent (and thus power is only consumed) if necessary, but the bandwidth must remain reserved during the whole uptime of the system.

5.4 Reconfiguration Timing

The reconfiguration process is a periodic real-time task that consists of 8 steps. The time window of each step is illustrated in Figure 5.5. The detailed values can be found in Table 5.5. Steps that are highlighted in gray are network transfers.

The largest parts are occupied by computational steps. The network transfers take only a small part of the time and can be shortened further, in case the fragment period is decreased. The presented solution is a tradeoff between using not too much bandwidth but still providing enough time for the calculations.

Since the processor of the TNA is less powerful than the one of the RMA, the time reserved for verification is comparable to that reserved for the much more complex task of schedule generation.

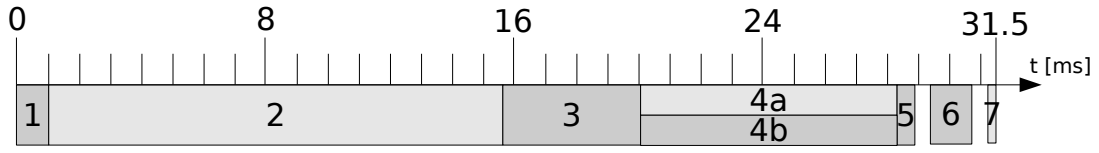


Figure 5.5: Reconfiguration timing

#	Action	Start [ms]	End [ms]	Duration [ms]
1	Application Mode Requests	0.00	0.97	0.97
2	Resource Allocation	0.97	15.63	14.66
3	Forward Resource Allocation	15.63	20.02	4.39
4a	Verification and config write	15.63	28.32	12.69
4b	Transmission D-Port/L-Port mapping	15.63	28.32	12.69
5	Verification Result	28.32	28.80	0.48
6	Config-change Notification	29.30	30.76	1.46
7	Config switch	31.49	31.50	0.01

Table 5.5: Timing of the reconfiguration steps

5.5 Interfaces

This section contains detailed descriptions of the interfaces required for resource management. After an overview of the entire resource management process in the following section, the individual interfaces are described in detail.

5.5.1 Overview

Figure 5.6 gives an overview of the architectural elements and their interfaces that participate in the resource management process.

The following list gives a short introduction to each of the interfaces:

1a The *pulse definition* represents a database comprising all pulses (except the *guaranteed pulses*) that are transmitted via the NoC. Out of this information, the RMA constructs the pulse schedule (see Section 5.5.2).

1b Similar to 1a, but for *guaranteed pulses*, which must also be provided to the TNA to enable the TNA to check the existence of all *guaranteed pulses* in the pulse schedule and that their properties are correct.

2 Via the *Host-to-RMA* interface, mode requests from the jobs inform the RMA which resources are required (see Section 5.5.3).

3 Since the RMA is not a trusted component, the resource allocation must be sent to the TNA for verification. This is done via the *RMA-to-TNA* interface (see Section 5.5.6).

4 During the verification of the schedule the RMA sends the proposed mapping of L-Ports to D-Ports via the *RMA-to-Host* interface to the

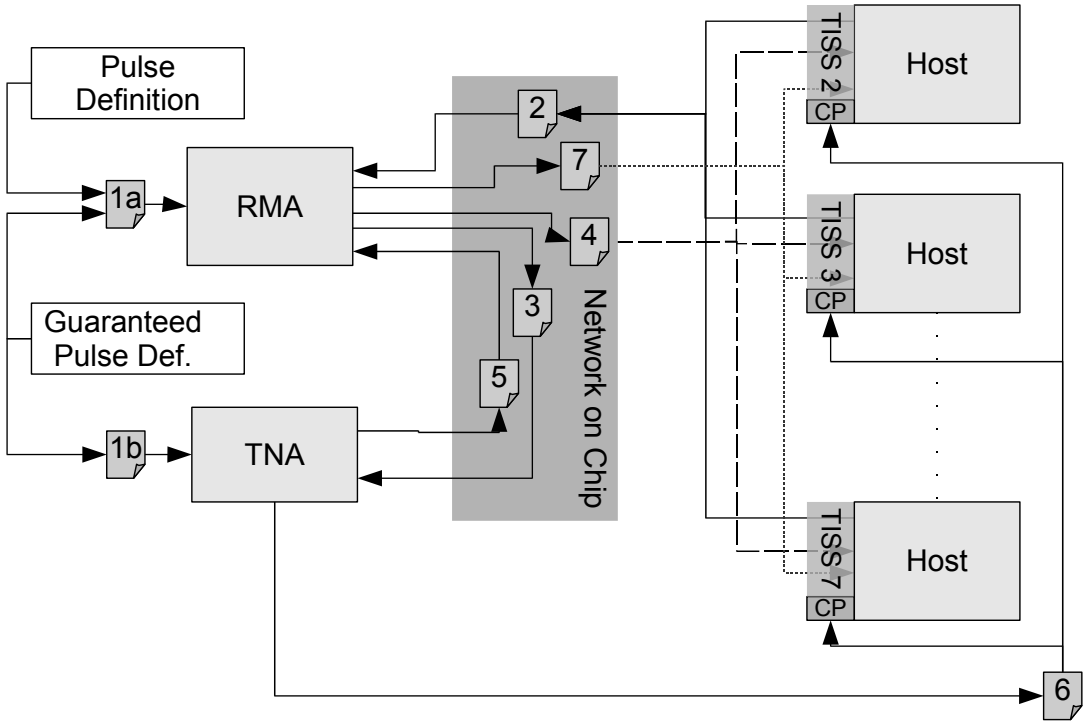


Figure 5.6: Interfaces overview

hosts so that they can prepare the new port setup. However, if the resource allocation turns out to be invalid the mapping loses its validity (see Section 5.5.5).

5 The TNA uses the *TNA-to-RMA* interface to send back the verification result to the RMA, which is exploited by the RMA to notify the hosts (see Section 5.5.7).

6 In case the verification turns out to be successful, the TNA reconfigures the TISSs via the *TNA-to-TISS* interface (see Section 5.5.8).

7 As a last step, the *RMA-to-Host* interface is exploited by the RMA to inform the hosts of the upcoming configuration (see Section 5.5.3).

5.5.2 Pulsed Data Stream Definition Layout

As explained in the beginning of Chapter 4 all possible pulsed data streams have to be defined in advance. For each micro component the pulses are defined in a list of pulse definition entries shown in Figure 5.7. The pulse with ID "-1" signals the

end of the list. The list cannot change during run-time, but the subset of active pulses is configurable as described in Section 5.3.4.

There are guaranteed pulses that must exist during the whole runtime of the system. They are required for the flawless operation of the system. Unlike the other pulses, these guaranteed pulses cannot be activated or deactivated. In order to specify a pulse as a guaranteed pulse, the `G-flag` has to be set to "1".

A pulse is defined by its period and phase offset, its fragment period and its length (i.e. the number of fragments). These data is stored in the fields `Period`, `FragPeriod`, and `PulseLen`. Note that out of optimization purposes `PulseLen` is given as the number of fragments reduced by one. The period lengths and the phase offset are encoded as described in Section 5.2. For the pulse period only the 10 period values implemented by the DLL (cf. Section 5.3.2) are valid. For the fragment period all of the 32 period lengths are valid as long as all fragments fit into the period length. This is expressed in the following inequality:

$$\texttt{PulseLen} * 2^{-FragPeriod} < 2^{-PulsePeriod}$$

During normal operation the phase offset is calculated by the RMA and needs not be specified. However, in order to be able to write the startup configuration, the TNA needs to know the phase offset to set up the NoC. So, for any guaranteed pulse a phase offset value must provided in the field `InitialPhase`.

The range for the phase offset of any message is restricted to the period length of the pulse. This range can be further restricted using the fields `LowBound` and `UppBound`. For example two pulses can be restricted in such a way that is it assured that one is finished before the other starts. `LowBound` gives the minimum phase offset of the first fragment while `UppBound` gives the maximum phase offset of the first fragment. For feasible scheduling the scheduling margin `UppBound` − `LowBound` should not be set below the time distance between the fragments (i.e. the fragment period length). The only exception to this are static pulses where `UppBound` − `LowBound` = 0. In this case the phase offset is fixed and cannot be changed by the RMA. Static pulses must be treated specially by the algorithm and need reservation of otherwise free sending slots thus require more bandwidth. The reason for this was explained in the presentation of the scheduling algorithm in Section 4.5.8.
If no restrictions apply `LowBound` = "0″ and all bits of `UppBound` are "1".

The meaning of the `BitField` field depends on the flag `G`. If the pulse is a *guaranteed pulse* (`G`=1) each bit of the field represents a micro component of the system. The position of the bit corresponds to the ID of the TISS of the micro component. Each micro component, for which the respective bit is set, is receiver or sender of the pulse. Since each micro component has its own outgoing pulse list, the sending micro component can be identified.

If the pulse is not guaranteed (`G`=0), `BitField` serves the decision of whether the pulse has to be scheduled in the current configuration or not as explained in Section 5.3.4.

To identify the pulse, the logical port identifier (`LogicalPort`) of the sender job is included in the pulse definition. The logical port ID has a hierarchical structure (Figure 5.8) similar to the application hierarchy. At the highest level the DAS ID identifies the Application Subsystem, the job ID identifies the sending job within the DAS and the Port ID distinguishes between the different pulses of a job.

Using the data described above, the RMA generates a conflict free schedule for the pulses and assigns L-Ports to D-Ports. At the TNA the schedule is received and checked for collisions. The TNA has the data of all guaranteed pulses and checks whether the parameters are correct and whether the bounds `LowBound` and `UppBound` are not violated.

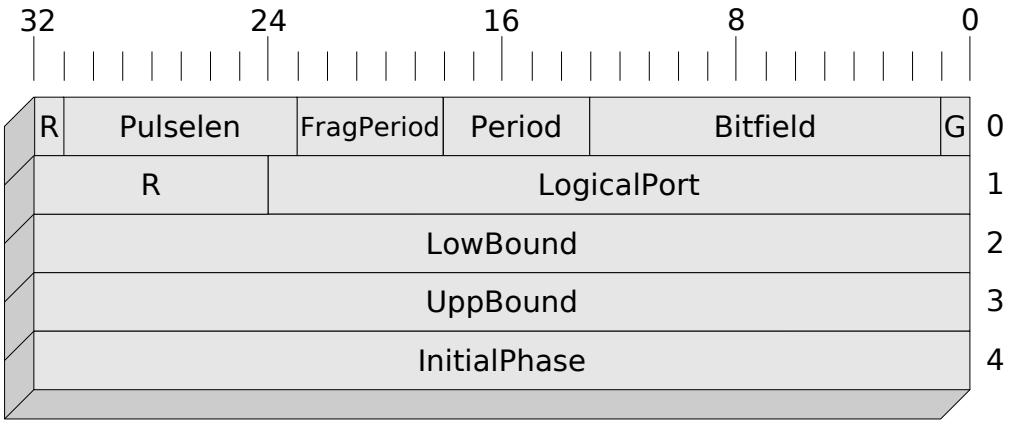


Figure 5.7: Pulsed data stream definition layout

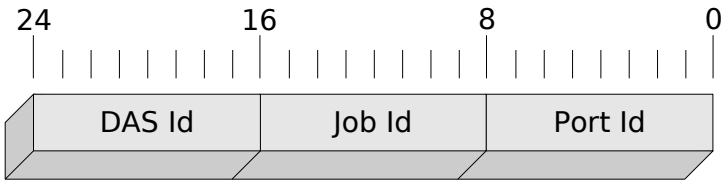


Figure 5.8: Logical port identifier

Constraints

The initial phase offsets of *guaranteed pulses* must be conflict-free. Otherwise, the transmission of *guaranteed pulses* might fail.

`G`	1-bit	If $G = 1$ the pulse is guaranteed and is therefore always scheduled.
`Bitfield`	12-bit	If $G = 0$ the pulse will only be scheduled if the bitwise AND-operation of $Bitfield$ and $Hostmode$ is non-zero. If $G = 1$ the pulse will be scheduled for the micro components given in $Bitfield$.
`Period`	5-bit	One of the 32 possible values for the period of pulsed data streams.
`FragPeriod`	5-bit	One of the 32 possible values for the fragment period of pulsed data streams.
`PulseLen`	8-bit	Index of last fragment of the pulsed data stream (i.e. length of the pulse - 1).
`LogicalPort`	24-bit	Value of the logical port id (L-Port ID) assigned to the pulsed data stream.
`LowBound`	32-bit	Left-aligned, the earliest phase where the first fragment may be scheduled.
`UppBound`	32-bit	Left-aligned, the latest phase where the first fragment may be scheduled.
`R`		Reserved

Table 5.6: Pulsed data stream definition attributes

In addition, a micro component cannot handle more than one pulse of the same pulse period at the same time. Thus, these messages must not possess a configuration of `UppBound` and `LowBound` which would result in their overlapping. For the same reason the sum of the durations of the pulses of the same period of one micro component must not exceed the length of the pulse period, because they can only be scheduled one after the other.

Depending on the DLL parameters, only a subset of the 32 possible values for the pulse periods is implemented. In configurations comprising pulses with unsupported period lengths, those pulses are ignored by the TISS. The value of the fragment period is restricted by the design of the TISS to 23, since the shortest fragment period achievable by the TISSs is $2^{-23}s$.

5.5.3 Micro Component-to-RMA Interface

This interface separates the architectural elements for resource management from the application. Via this interface, jobs can issue resource requests to the RMA. The other way round, this interface is exploited by the RMA to inform the jobs which resources will be made available for them.

The jobs transmit an identification of the application mode at which they have to

run. The RMA looks up in a local table to identify the resources that are required for a specific application mode.

Regardless of whether a micro component has requested a mode change or not, the RMA sends to the job the number of the actually active application mode. On the one hand, this is used to signal a resource allocation problem: If the actual mode differs from the requested mode, the job is able to detect that his resource request has been declined. On the other hand, it enables a job to change the mode of another job within the same DAS. For instance in order to realize a "controller" job within a DAS, which is responsible for allocating the resources for all jobs if necessary.

Since both messages (*Job-to-RMA* and *RMA-to-Job*) contain the application mode number their structure (as depicted in Figure 5.9) is very similar. However, the message from the RMA contains an additional `SysMode` field that carries information about the system mode to which the micro component will be set. The system mode cannot be affected directly by the micro components, but is based on application modes and the system state. The system mode consists of one bit for turning a micro component *on* or *off* and the period of the watchdog timer. More about the system mode follows in the next section.

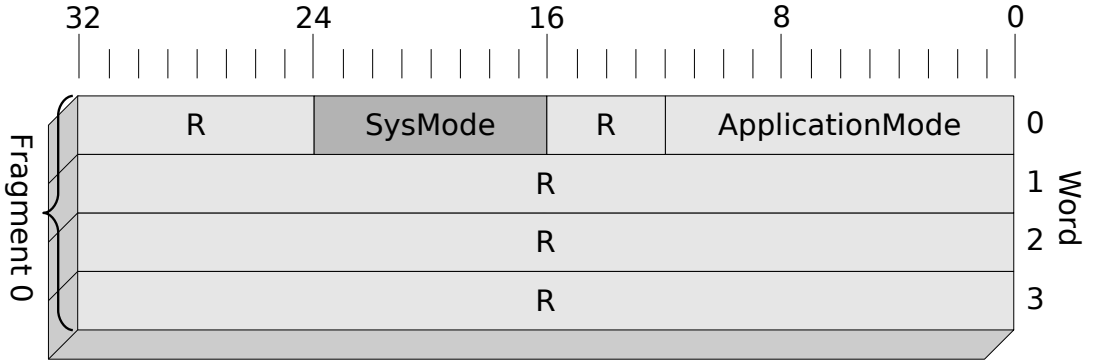


Figure 5.9: Micro component-to-RMA interface

`ApplicationMode`	12-bit	Number that identifies the application mode of the application (either by the job or assigned by the RMA).
`SysMode`	8-bit	Return message from the RMA only: describes the system mode after the next reconfiguration instant, for details see Figure 5.10.

Table 5.7: Micro Component-to-RMA Interface Attributes

5.5.4 System Mode Information

Currently, the system mode contains 2 types of information (cf. Figure 5.10): the service level (one bit for modes "on" and "off") and the watchdog period. The watchdog period determines the maximum time interval between two lifesigns before the host is reset. The lifesigns must be issued from the host via the TL. If the time between the sending of two lifesigns is exceeded, it is assumed that a failure at the host has occurred, which potentially can be resolved by performing a reset of the host.

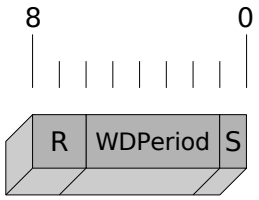


Figure 5.10: System mode byte

`S`	1-bit	Service level of the host. The service level is restricted to values "0" (micro component switched off) and "1" (micro component turned on).
`WDPeriod`	5-bit	Frequency of the life-sign for the watchdog timer. The value "0x1F" disables the watchdog functionality.

Table 5.8: System Mode Attributes

5.5.5 Port Mapping

The assignment of D-Ports to L-Ports is not statically defined at design time of a micro component. This allows dynamic and thus efficient assignment of D-Port numbers to a particular pulsed data stream by the RMA. The transmission of the mapping information to the micro components is achieved by separate pulses from the RMA to each of the individual micro components. In these pulses, the L-Port IDs along with the D-Port numbers are listed (cf. Figure 5.11).

The first two bytes of the logical port IDs (`DAS ID` and `Job ID`) are constant for all pulses of the same micro component so they are transmitted only once in the first word (DAS ID and job ID).

Each of the remaining 15 words contains two pulse entries totaling in 30 such entries. Currently, the design of the system is restricted to 62 pulses, it is a fair

assumption that a single micro component deals with at most half of the possible pulses in the system.

The entries are numbered from pulse 0 (`P0`) to pulse 29 (`P29`). One entry consists of two bytes. The low order byte `PXL-Port` together with DAS ID and job ID form the L-Port identifier presented in Figure 5.8. The high order byte `PXD-Port` gives the Data Link Layer Port where the pulse data can be accessed.

The separation of the pulse containing the port mapping from the RMA response pulse to the jobs is done in order to achieve a more efficient resource usage. Since the L-Port/D-Port mappings for all micro components is of considerable size, the gap in the network usage during the verification activities performed by the TNA is perfectly suited for their transmission.

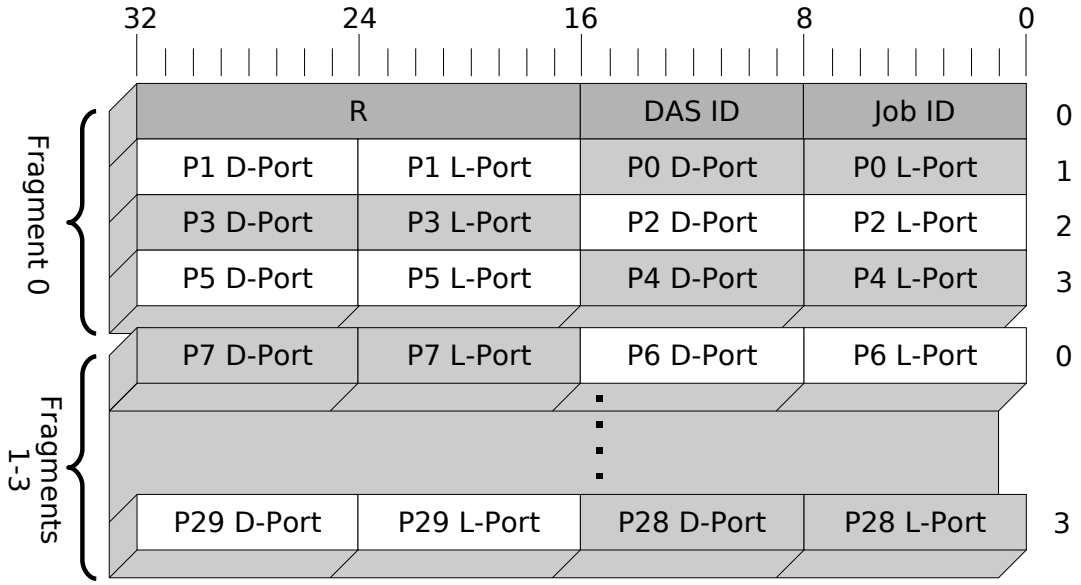


Figure 5.11: Port mapping

`DAS ID`	8-bit	ID of the DAS the job belongs to.
`Job ID`	8-bit	ID of the job within the DAS.
`PXL-Port`	8-bit	L-Port of the X^{th} pulsed data stream.
`PXD-Port`	8-bit	D-Port of the X^{th} pulsed data stream.

Table 5.9: Port mapping

5.5.6 RMA-to-TNA Interface

After having gathered the application mode requests of all DASs, the RMA calculates the resource allocation for the individual micro components. The main task

is to assign the phase offsets of the activated pulses. In addition, the System Mode of the hosts is determined. This information has to be transmitted to the TNA where it will be checked for validity. The layout of the pulse that is sent from the RMA to the TNA is depicted in Figure 5.12.

In fragment 0 of this pulse, the fields `RecfgPhase` and `RecfgPeriod` determine the instant when the next reconfiguration activity is performed (i.e. the point in time when the configuration that is currently transmitted to the TNA becomes active). Usually, these values remain constant throughout the entire uptime of the system, since they represent system parameters that are determined at design time.

The TNA has to acknowledge the validity of the resource allocation so that the RMA can inform the micro components about the success of their resource requests. The acknowledgment or reject message is automatically disseminated by the TNA after a well-specified delay, which represents the amount of the time that is granted to the TNA software to verify the configuration. If the TNA software crashes or fails to meet the deadline, the RMA might receive an old acknowledge message (since communication between RMA and TNA is established by a guaranteed periodic time-triggered pulsed data stream). To avoid that the RMA mistakes an old message as valid the field `RequestID` is used. The TNA has to perform a mathematical operation on this value and responds with the result. Due to this value the RMA can decide, whether the TNA has correctly performed this operation.

The `SysModeX` fields determine the system mode for each micro component. The field possesses the same syntax as the SysMode attribute in the *RMA-to-Job* response message as described in Section 5.5.4.

The following fragment is reserved for future use. For instance, it can be expected that the SysMode fields are extended if more advanced power scaling mechanisms are supported by the hardware of the SoC. For this purpose, this reserved fragment can be exploited.

The calculated schedule of all pulses is specified in the following fragments (fragments 2-63). Each fragment holds a single pulse description, so a maximum of 62 active pulses is possible. Up to 20 pulsed data streams are used for configuration purposes. At least 42 pulses (that is an average of 7 outgoing pulses for each micro component hosting a job) are free for application usage.

The temporal characteristics of a pulsed data stream are specified in the fields `Period`, `FragPeriod`, `PulseLen`, and `Phase`. The `SendMC` field states the ID of the micro component sending the particular pulse. In addition, the TNA has to know which micro components are involved in the reception of the pulsed data stream and at which D-Port they expect the data for use in the verification process. For instance, this information is necessary, if the additional constraints defined in the subsection "Constraints" of Section 5.5.2 (e.g. two pulses with the same

pulse period must not be interleaved) are fulfilled by the communication schedule. The fields `D-Port0` to `D-Port7` serve this purpose. Each entry holds the D-Port number at the corresponding micro component at which the data of the pulse is stored. To disable the reception of the pulse at a certain host, the flags `F0` to `F7` may be used. Setting flag `FX` to "1" invalidates the value of `D-PortX` and disconnects host$X$ from the pulse. The advantage of using a zero active flag here is that if a port is valid, the 8-bit value composed of `D-Port` and `F` can be used by the TNA without the need to toggle the Most Significant Bit (MSB). The data link layer supports port numbers from "0" to "118". See TISS CP Interface (Section 5.5.8) for details.

The only field not yet discussed is `Guaranteed Index`. It is the index to the pulse in the table of *guaranteed pulses*. The field is used by the TNA to check the properties of *guaranteed pulses* and to check whether all *guaranteed pulses* have been scheduled.

For the verification algorithm within the TNA, the ordering of the pulses is of high importance. The pulses have to be ordered by ascending period values (i.e. descending period cycle lengths). Pulses of the same period must be ordered by ascending phases. This requirement substantially simplifies the TNA. Furthermore, this ordering is already provided by the scheduling algorithm.

`RecfgPhase`	32-bit	Left-aligned, the phase offset at which future configuration changes take place.
`RecfgPeriod`	5-bit	The periodicity of the reconfiguration cycle.
`RequestID`	16-bit	The identification of the request used to match the response from the TNA to the actual request.
`SysModeX`	8-bit	The new system mode of micro component with ID X. See Figure 5.10 for the details.
`Phase`	32-bit	Right-aligned phase of the first fragment of the pulse.
`Period`	5-bit	Period of the pulsed data stream.
`FragPeriod`	5-bit	Fragment period of the pulsed data stream.
`SendMC`	3-bit	The index of the host that is the sender of the pulsed data stream.
`PulseLen`	8-bit	Index of the last fragment of the pulsed data stream (i.e. length of the pulse - 1).
`D-PortY`	7-bit	D-Port number at micro component with ID Y.
`FZ`	1-bit	Equals "0" if `D-PortZ` is valid or "1" if micro component with ID Z does not receive the pulse.
`Guaranteed Index`	6-bit	Index into the table of *guaranteed pulses* or -1 if the pulse is not a guaranteed pulse.

Table 5.10: RMA-to-TNA Interface Attributes

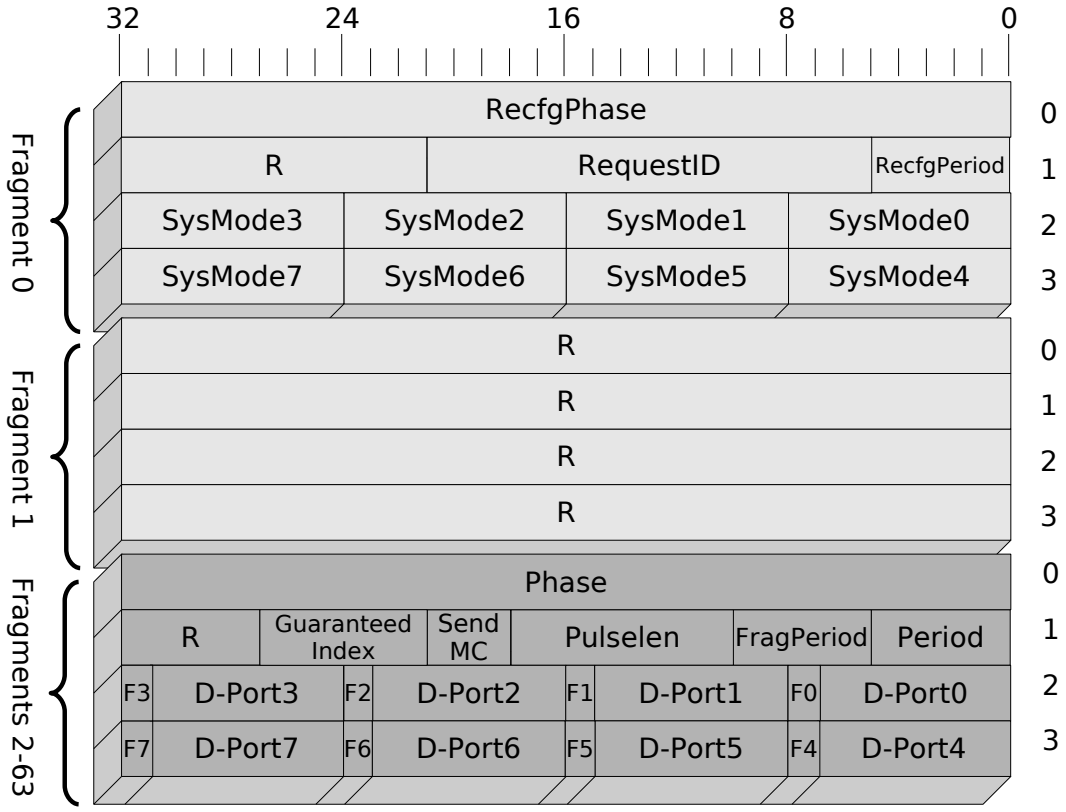


Figure 5.12: RMA-to-TNA interface

5.5.7 TNA-to-RMA Interface

After the verification of the resource allocation, the TNA disseminates its response back to the RMA. In case the proposed resource allocation is valid, the TNA returns an acknowledge message to the RMA and updates the TISSs via their CP interfaces. Otherwise the detected violation of the constraints is reported to the RMA. The layout of the response is described in Figure 5.13. The error codes reporting the detection of a violation of the schedule follow in Table 5.12.

5.5.8 TNA-to-TISS Interface

The TNA configures the micro components via the CP interface of the TISS. This interface is separated from the NoC in order to simplify the access, which enables the TNA to directly alter the configuration of the micro components independently of ongoing communication on the NoC.

This interface is realized as a memory interface with an 11-bit address bus and

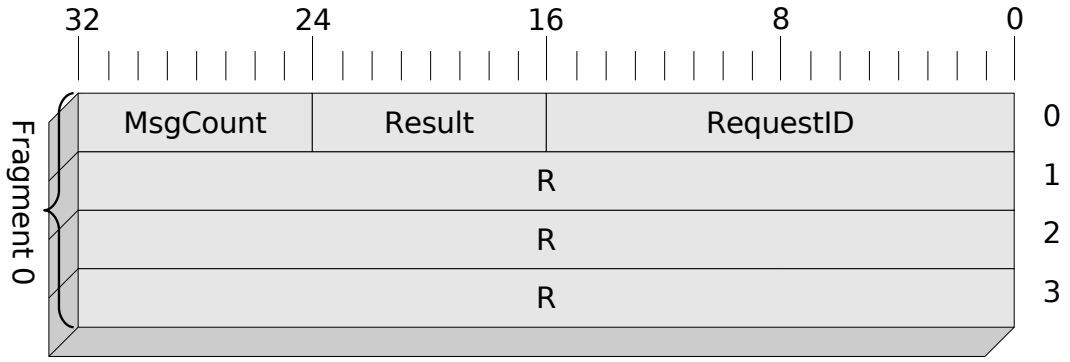


Figure 5.13: TNA-to-RMA interface

`MsgCount`	8-bit	If Result="0" (no error) this field indicates the number of active pulses in the current configuration. The value is undefined otherwise.
`Result`	8-bit	Indicates whether the verification was successful. In case a violation was found, the problem source is indicated. The values and their meanings are listed in Table 5.12
`RequestID`	16-bit	The updated value that has been disseminated from the RMA in order to identify a particular reconfiguration request.

Table 5.11: TNA-to-RMA interface attributes

a 32-bit data bus. The 3 most significant bits of the address are used to address one of the 8 TISSs in the system. The remaining 8 bits allow for addressing the 256 words of the memory interface of each TISS (cf. Figure 5.14).

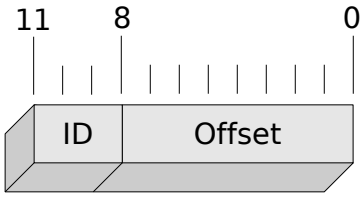


Figure 5.14: TISS address

In order to allow for updating the TISS configuration during operation of the SoC, the configuration memory block is realized in each TISS as a shadow buffer, i.e. the configuration memory is doubled. Each write access of the TNA operates solely on the inactive shadowed configuration block. The last step when writing the configuration memory is to update the valid flag `V` to "1". This signals the TISS

0	OK	Verification successful, no violations found.
1	COLLISION	At least one pulse cannot be scheduled at the given phase because resources (e.g. bus) are not available.
2	BAD ORDER PERIOD	The messages are not ordered as required by the verification algorithm of the TNA (descending period length).
3	BAD ORDER PHASE	The messages are not ordered as required by the verification algorithm of the TNA (ascending phase offset).
4	MISSING	At least one pulsed data stream that is defined as guaranteed pulsed data stream is missing in the schedule.
5	DUPLICATE	The L-Port/D-Port mapping contains two pulses allocated to the same D-Port.
6	MISMATCH	At least one of the guaranteed pulsed data streams has invalid (temporal) properties.

Table 5.12: TNA verification result values

`ID`	3-bit	Identification of the TISS (by means of the ID of the micro component the TISS belongs to).
`Offset`	8-bit	Word offset in the configuration memory of the TISS.

Table 5.13: TISS address structure

that a reconfiguration of the entire micro component is requested. At the next reconfiguration instant (which is determined by `RecfgPeriod` and `RecfgPhase` of the active configuration block) the TISS switches the configuration blocks and starts working with the new configuration. Future write accesses operate on the old (now inactive) block. Furthermore the TISS toggles the read-only active flag `A` so that it can be observed that the change was performed.

In addition to the fields containing configuration management data `A`, `V`, `RecfgPeriod`, `RecfgPhase` the TISS configuration contains the operation mode (`S`) of the host and the configuration of the watchdog service `WDPeriod` (cf. Figure 5.10).

The remainder of the configuration memory is reserved for holding the Message Descriptor List (MEDL) of the micro component. The MEDL is a list of all pulsed data streams sent or received by the particular micro component.

The data structures of these pulse configurations are realized as cyclic linked lists. For each of the 32 periods there is a list of pulses of that period. The pointers (cf. `PointerX` in Figure 5.15) give the word offset of the first list element. The

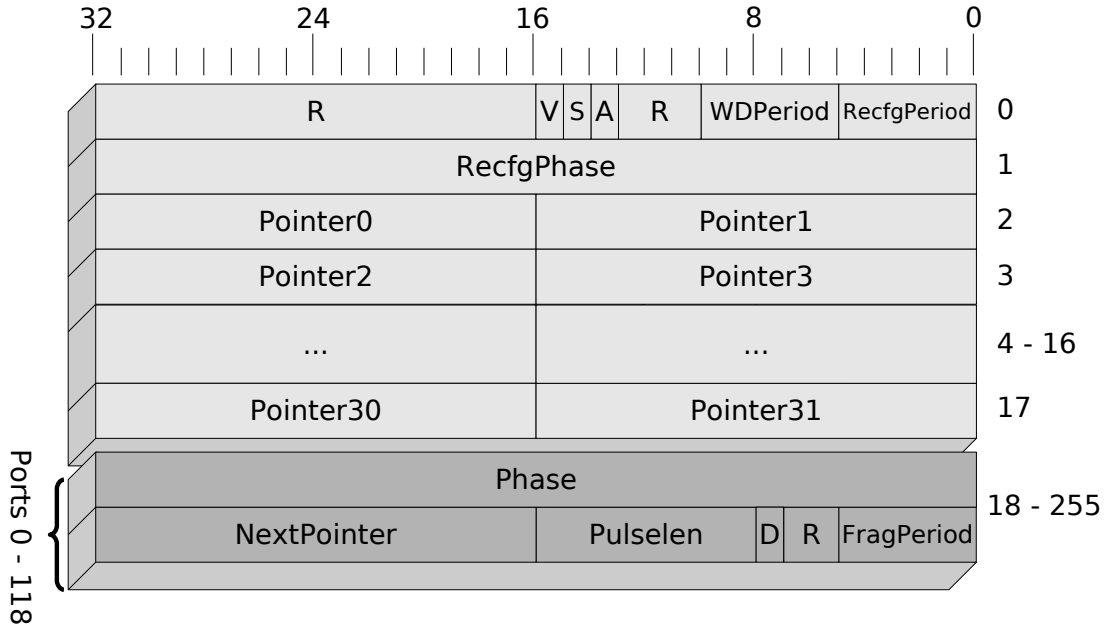


Figure 5.15: TNA-to-TISS interface

reserved value "0" represents an empty list if no pulse of the respective period exists. Within a list, it is important that the list elements are sorted by ascending phase offsets.

Each list element contains the description of the pulsed data stream (`Phase`, `FragPeriod`, `PulseLen` and `D`) and the pointer to the next list element. The last list elements points back to the first element. The memory-offset of the pulse relates to the D-Port number in the following way:

$$port_{dl} = \frac{offset-18}{2}$$

The D-Port number is important for the identification of a pulsed data stream at the higher levels of the TISS such as the TL.

5.5.9 Network-on-Chip Interfaces

So far, only the high-level interfaces directly related to resource management have been discussed. Figure 5.16 gives an overview of the low-level NoC interfaces. The details are not in the scope of this document. Here is a short summary:

1 The interface between the full-featured Transport Layer (TL) and the

RecfgPeriod	5-bit	Period of future reconfiguration cycles, usually a system parameter that remains constant.
WDPeriod	5-bit	Frequency of the life-sign for the watchdog timer. The value "0x1F" disables the watchdog functionality.
A	1-bit	Active bit, read-only, reflects which of the two configuration blocks is active.
S	1-bit	Service level of the host (on/off).
V	1-bit	Valid bit, must be written to "1" to finalize configuration updates.
RecfgPhase	32-bit	Left-aligned, the phase at which future configuration changes take place, usually a constant system parameter.
Pointer*X*	16-bit	Offset to the pulsed data stream of period $2^{-X}s$ with the minimum phase (the first element in a linked list) or "0" if there is no such pulsed data stream.
Phase	32-bit	Left-aligned instant of the first fragment of the pulse.
FragPeriod	5-bit	Fragment period of the pulsed data stream (time between 2 succeeding fragments).
D		Direction bit, if D="1" the pulsed data stream is outgoing, if D="0" it is incoming.
PulseLen	8-bit	Index of the last fragment.
NextPointer	16-bit	Points to the offset of the succeeding pulsed data stream or the offset of the first pulsed data stream if this is the last one in this period (linked list of pulsed data streams).

Table 5.14: TNA-to-TISS interface attributes

hosts is a memory-mapped OCP interface. A more detailed description is available in Roman Seiger's master thesis [Sei07].

2 The interface between the minimal TNA Transport Layer and the TNA is a simple memory-mapped interface. Since this TL was developed especially for the TNA its description is found in this document in Section A.1.

3, 4 The interfaces regarding the DLL are thoroughly explained in Gerhard Engleder's master thesis ([Eng07]).

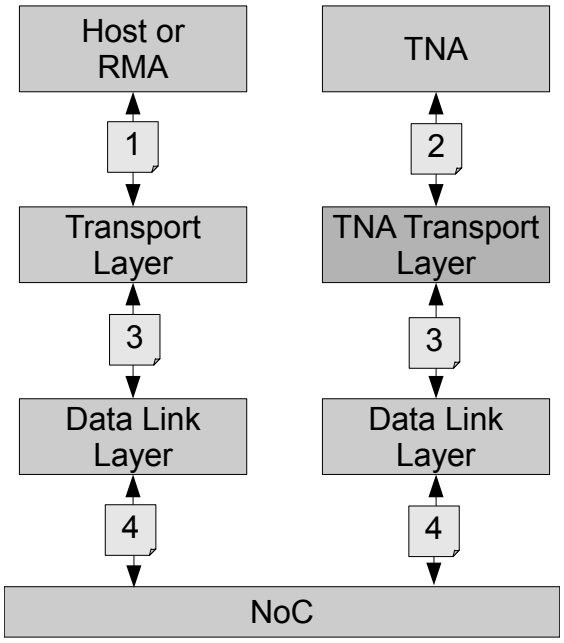


Figure 5.16: NoC interfaces

5.6 RMA/TNA Implementation

This section details the implementation of the two major components of the resource management system: the RMA and the TNA. First, the hardware is described and then the details of the software implementation follow.

5.6.1 Hardware

The software of the two main components responsible for the resource management runs on dedicated Nios II processors. To meet the high computational requirements of the RMA the most powerful Nios II was chosen, while the TNA manages with less costly hardware. More about Nios II processors is available in [Alt07c]). Table 5.15 gives the basic properties of the processors.

5.6.2 RMA Software

The program flow of the RMA is cyclic as depicted in Figure 5.17.

1 During initialization the Transport Layer is set up for the ports to and from the TNA and the micro components. The initial application mode is applied.

	RMA	TNA
Nios II Type	fast	economy
Clock Speed	75 MHz	75 MHz
Instruction Cache	4096 Byte	-
Data Cache	2048 Byte	-
Pipeline	6 stages	1 stage
Features	Hardware multiplier, dynamic branch prediction	-

Table 5.15: RMA / TNA processor properties

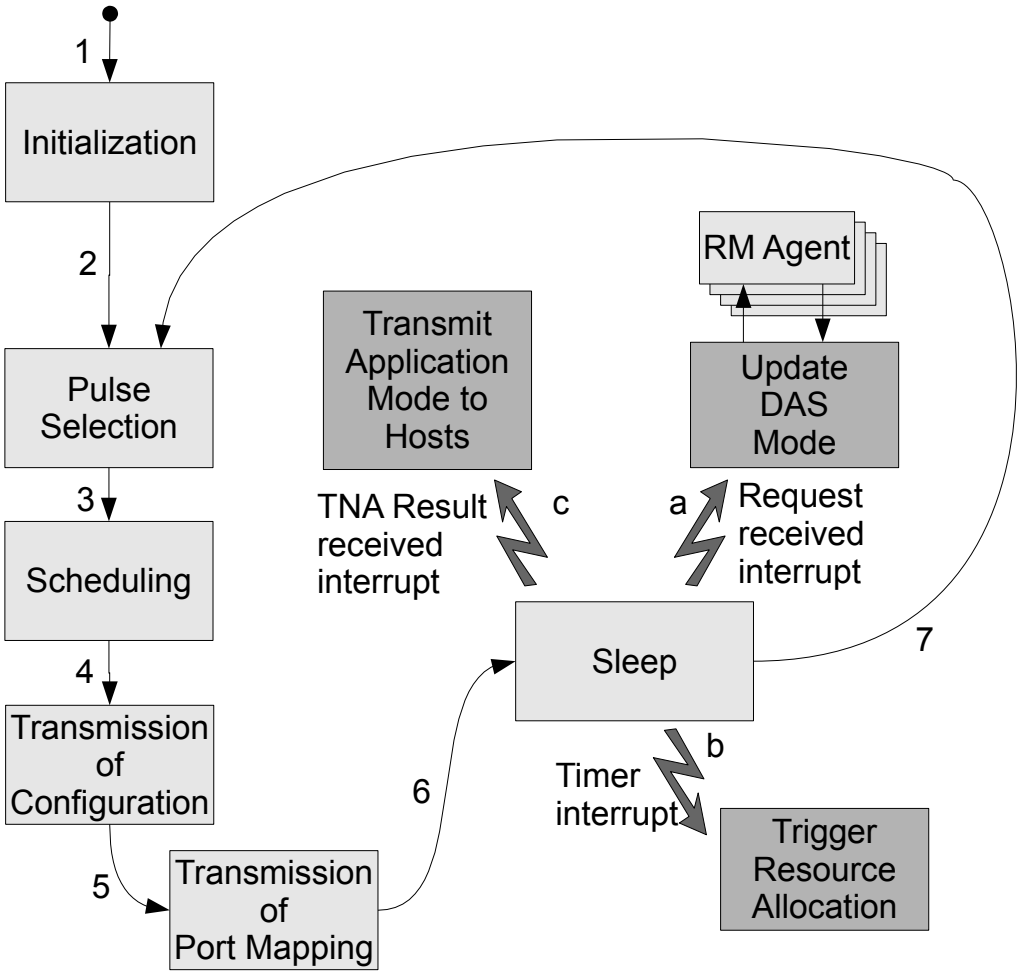


Figure 5.17: RMA program flow

2 The parameters of the pulses for the current application mode are loaded.

3 The resource allocation is calculated by the scheduling algorithm.

4 The resource allocation is put into the NoC buffer to be sent to the TNA.

5 The mapping of L-Ports to D-Ports is put into the NoC buffer to be sent to the micro components.

6 All upcoming events are triggered from outside the processor. Therefore they are handled in Interrupt Service Routines (ISRs) which allows the processor to be put in sleep mode[1]. The main disadvantage of ISRs, which is unpredictability, does not apply here since the interrupts are time-triggered and thus easily predictable. The small overhead of the ISR is negligible.

   The following events are handled sequentially by ISRs. The first and the third are network events, the second is a timer interrupt:

   a **Host request received:**
   This is the first phase of the reconfiguration. Any incoming host application mode request is forwarded to the Resource Agent of the respective DAS. The Resource Agent is a function that takes the application mode request of any of its jobs and returns the new system mode for all jobs of the DAS.

   b **Schedule generation:**
   It cannot be assured that the last job actually disseminates an application mode request, so the resource allocation calculations are triggered by a separate timer interrupt and not after evaluating the last request. The TL is configured to raise the interrupt shortly after the last possible reception of the last request message. The ISR simply sets a flag so that the main procedure starts with the calculations instead of going to sleep again.

   c **TNA result received:**
   The hosts must be informed of the successful mode change, so the mode values are put in the network memory to be sent to the hosts. This is the last task of the RMA in the reconfiguration cycle.

7 The reconfiguration cycle resumes at step 2.

[1]The Nios II does not support sleep modes, but the design does not target the Nios architecture in particular

Data Structures

Pulsed Data Stream (PSD) Objects The algorithm mainly works with PSD objects. Each of these objects represents a pulsed data stream and holds the following data:

`Period`: Pulse Period

`FragPeriod`: Fragment Period

`PulseLen`: Fragment Count

`LowBound`: Minimum allowed phase

`UppBound`: Maximum allowed phase

`Phase`: Phase offset within the pulse period. The Phase is assigned when the scheduling algorithm has found a valid phase for the pulsed data stream. `Phase` is bounded by `LowBound` and `UppBound`.

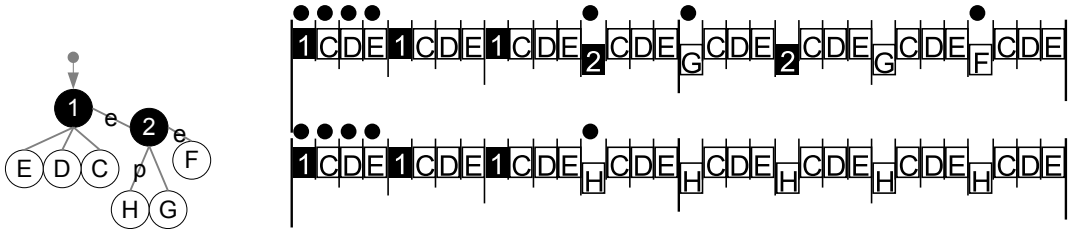
All fields except `Phase` are initialized from the pulse definitions (see Section 5.5.2). All PSD objects are stored in an array which is the input for the scheduler.

Phase Hash Table As soon as a valid phase has been found for a pulse, its `Phase` field is updated and it is added to the *Phase Hash Table*. The *Phase Hash Table* allows to quickly (i.e. in nearly constant time) locate the pulse with a specific `Phase`. It uses the Modulo operation as hashing function and linked lists to handle hashing collisions. The size of the *Phase Hash Table* may be adjusted, more than twice the number of defined PSDs is recommended. In any case the size should be a prime number to reduce hash collisions to a minimum.

***Period List*s** In addition to being added to the *Phase Hash Table*, a pulse is added to the list corresponding to its `Period` when its `Phase` is assigned. 32 lists exist, one for each of the 32 possible periods.
The PSDs in these *Period List*s are sorted by ascending `Phase`. Firstly, they are used while checking the "Same Period Constraint". Secondly, they aid in writing out the pulse data for the TNA in the correct order. They are realized using STL `map`s [SL95].

The Tree In Section 4.5, the tree built up by already scheduled pulses was introduced. In the implementation this tree does not explicitly show up because the pulse objects do not reference their child objects or leaves directly. Instead, a pulse object carries the necessary information to compute the `Phase`s of child nodes, which can be looked up using the *Phase Hash Table*.

Child Relationships Before presenting the implementation, a short recapitulation of the relations between parent and child nodes is given here that may be useful to understand the internals of the algorithm. Figure 5.18 illustrates the relations in graphical form. Its elements were explained in paragraph "Figures" in Section 4.5.5.



(a) Tree (b) Slot Allocation

Figure 5.18: Child relationships

The list of relationships follows:

Extension The fragments of the child pulse are aligned with the fragments of the parent pulse. In Figure 5.18(b), this is true for *Pulse 1* → *Pulse 2* as well as *Pulse 2* → *Leaf F*. An extension pulse can itself be extended by a further extension pulse. This way, a chain of extension pulses can be formed. Obviously, the chain cannot be longer than the period length of the first pulse (in our case *Pulse 1*). Therefore, this period is called **Dominating Period**. But other restrictions apply as well: both the fragment period and the pulse period of the extending pulse must be longer than the respective period of the parent pulse. Disrespecting these restrictions risks collisions with Subphase and/or Subperiod pulses.

Subphase For pulses in this relationship the first fragment of the child pulse is in between the first two fragments of the parent pulse and has an offset from the first fragment of the parent pulse that is a multiple of the fragment period length of its own parent.
In Figure 5.18(b), this is true for *Pulse 1* → leaves *C*, *D*, *E* and *Pulse 2* → *Leaf G*. The dominating period for these pulses is the dominating period for the parent pulse. Consider *Leaf B* and its parent *Pulse 1*. As *Pulse 1* is the first pulse, it is not dominated by any pulse, so the same applies to *Leaf B*. *Leaf G* on the contrary, is the child of *Pulse 2*, which is dominated by the period of *Pulse 1*. Looking at Figure 5.18(b), it can be seen that the fragments of *Leaf B* are not interrupted, while the fragments of *Leaf G* are interrupted by the next occurrence of *Pulse 1*, which confirms the above postulations.

Subperiod For pulses in this relationship the first fragment of the child pulse has an offset from the first fragment of the parent pulse that is a multiple of the period length of its own parent. In Figure 5.18(b), only the relation *Pulse 2* → *Leaf H* matches this description. *Pulse 1* is 32 slots long and *Leaf H*'s first fragment is 32 slots after *Pulse 2*'s first fragment. The restrictions for subperiod pulses can be seen in Figure 5.18(b). They are similar to those for *Pulse 2* (extending relation), only the pulse period must be longer than the pulse period of *Pulse 2* instead of the pulse period of *Pulse 1*.

Scheduling Algorithm

The scheduling algorithm has to traverse the tree of scheduled pulses to find a location for the next pulse. Thus, it is implemented as a recursive function with the following parameters:

phase identifies the location in the tree.

nextPulse the next pulse to be scheduled.

dominatingPeriod in this subtree, no pulse may exceed the bounds of this period, otherwise it collides with a prior pulse.

minFragmentSubperiod in this subtree, the algorithm must not exploit subphases shorter than this period.

minPeriodSubperiod in this subtree, the algorithm must not exploit subperiods shorter than this period.

First, the procedure checks whether `Phase` P is occupied with a lookup in the *Phase Hash Table*. If it is not occupied, it checks whether the "Same Period Constraint" is violated if `nextPulse` is scheduled at P. If this is not the case either, the pulse is scheduled at P, its `Phase` field is updated and it is added to both the *Phase Hash Table* and the *Period List*. The function returns *True* to indicate success and the abortion of the traversal.

Otherwise, if `Phase` is occupied by a pulse called `occPulse` here, its children are enumerated and the procedure is called for them. This is demonstrated in Listing 5.1.

Child Phases In this phase, the child relations presented in Paragraph "Child Relationships" are investigated to find a place for `nextPulse`.

First of all, if `occPulse` is the first pulse in a chain of extension relations, its period is the dominating period and must be handed down in the tree. Otherwise, the dominating period has already been set and remains untouched. The check is performed in Line 2.

The first possible child is the "extending" phase just after `occPulse`, its phase is calculated in Line 7. This phase may only be used if the period of `nextPulse` is longer than the period of `occPulse`.

Then, "subphases" and "subperiods" are tried. Both are handled using the function *doSubphases*, but with different parameters. The *doSubphases* procedure starts with a given phase. It adds the duration of the `SubPeriod` and calls `schedule` for the result. Then the duration is added a second time and `schedule` is called again. The process continues until the duration of `MainPeriod` is reached. See Figure 5.19 for an example. The behavior of *doSubphases* relates to the definition of subphase and subperiod children from Paragraph "Child Relationships". For subphases of *Pulse 2* (Figure 5.18), it starts with phase 12, and adds the period length of the fragment period of *Pulse 1* to get to phase 16, which is the position of subphase *Leaf G*. It then proceeds to phase 20 which is the second fragment of *Pulse 2*, so the enumeration is stopped. The subperiods of *Pulse 2* are found similarly: this time the pulse period of *Pulse 1* is added, so phase 44 (*Leaf H*) is reached in the first step. The enumeration stops at the second step as the next instance of *Pulse 2* is encountered.

There are different cases to consider. If `occPulse` is the first in the chain (e.g. *Pulse 1* in Figure 5.18), it has only "subphase" children. So one call to *doSubphases* is required (Line 17). In this case, the Fragment Period of `occPulse` serves as the parameter `MainPeriod`, because the time-slots in between the fragments will be used. The Sub Period is dictated by the caller of `schedule` (the parent of `occPulse`). It may be that every time-slot is still free (e.g. subphases of *Pulse 1*), but it is also possible that only a portion of the time-slots can still be used (e.g. subphases of *Pulse 2*). The restrictions for children must be specified when calling *doSubphases*. In this case, only the fragment period is important to avoid collisions with siblings. The fragment period is passed as parameter `minFragmentSubperiod`.

If `occPulse` itself is "extending" another pulse (e.g. *Pulse 2*), it can have both "subphase" and "subperiod" children. However, as with the "extending" children, they may not be used if the period of `nextPulse` is too short. The elaboration of "subphases" (Line 17) works similar to the case presented above. The difference is that more restrictions apply. First, the dominating period must be respected. Second, the pulse period of children cannot be shorter than the pulse

period of `occPulse`.

For "subperiods", `occPulse` determines the parameters `MainPeriod= occPulse.period` while `SubPeriod` takes the value handed down from the parent of `occPulse`. The fragment period restrictions from the parent of `occPulse` apply for subperiod pulses, so the parameter `minFragmentSubperiod` remains constant. However, the pulse periods are dictated by `occPulse`, so `minPeriodSubperiod=occPulse.period` for subsequent nodes.

```
1
2  if(period == MAX_PERIOD)
3          newDomPeriod = occPulse.period;
4  else
5    newDomPeriod = period;
6
7  newPhase = phase + PERIOD_LEN(occPulse.fperiod)
8                       * occPulse.PulseLen;
9
10 if(nxtPulse.period >= occPulse.period)
11 {
12   schedule(newDomperiod, newPhase, occPulse.fperiod, occPulse.period);
13 }
14
15 if(period == MAX_PERIOD)
16 {
17   doSubphases(occPulse.fperiod, minFragmentSubperiod, phase,
18               MAX_PERIOD, occPulse.fperiod, minPeriodSubperiod);
19 }
20 else
21 {
22   if(nxtPulse.period >= occPulse.period)
23   {
24     doSubphases(occPulse.fperiod, minFragmentSubperiod, phase,
25                 newDomPeriod, occPulse.fperiod, occPulse.period);
26
27     doSubphases(occPulse.period, minPeriodSubperiod, phase,
28                 newDomPeriod, minFragmentSubperiod, occPulse.period);
29   }
30 }
```

Listing 5.1: Enumeration of Child-"Phases" and Invocation of the Schedule Procedure

Checking the "Same Period Constraint" After a possible location has been found for `nextPulse`, it must be verified that no other pulses with the same period and common hosts overlap with the pulse. For this, the *Period List* of the period of `nextPulse` is searched for such pulses. Remember that *Period List* is sorted by the starting phase of the contained pulses.

The first part is easy. Verify that all pulses which begin within the start and end phase of `nextPulse` do not have a host in common with `nextPulse`. For this,

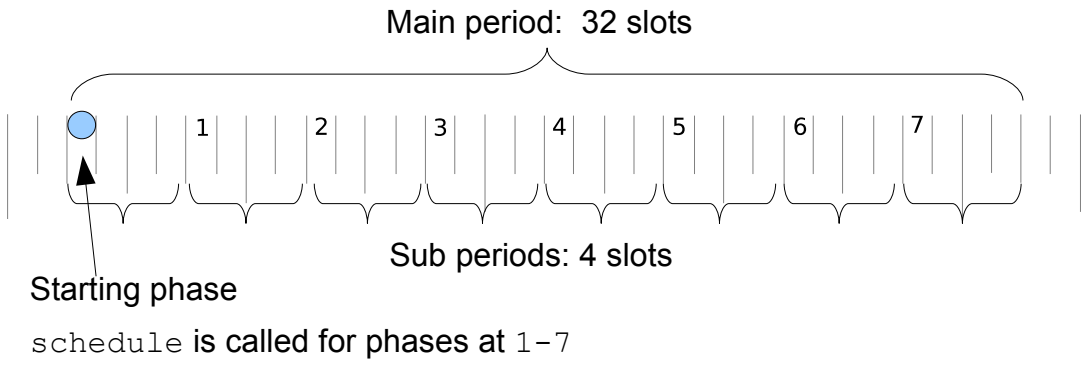


Figure 5.19: Example of doSubphases with MainPeriod=2^{-18} (32 slots) and SubPeriod=2^{-21} (4 slots)

Period List can be iterated from the start phase to the end phase of `nextPulse`. If there is a pulse with a common host another location must be found.

The second part is a bit tricky. Pulses that start before `nextPulse`, but end after the beginning of `nextPulse` must be investigated. A second list similar to *Period List*, but sorted by the ending phases, would not help. Pulses that start before `nextPulse` and end after `nextPulse` would go undetected (cf. Figure 5.20).

Figure 5.20: `nextPulse` is enveloped by an already scheduled pulse

Furthermore, it is not possible to go back only to the first pulse that has a host in common with `nextPulse`. This is illustrated in Figure 5.21. Here, we go back from `nextPulse` to find `oldPulse2`, which has Host 3 in common with `nextPulse` and ends well before `nextPulse`. But there also is `oldPulse1`, which has Host 1 in common with `nextPulse` and overlaps `nextPulse`, which is not allowed.

Thus, the only possible strategy is to continue until all hosts participating in `nextPulse` have been found in a pulse prior to `nextPulse`. Alternatively, if the number of pulses is not very high, the check may be started right from the very first pulse in the period instead.

Figure 5.21: Trap when checking the "Same Period Constraint"

5.6.3 TNA Software

The TNA program flow is illustrated in Figure 5.22.

1 During initialization the network interrupt handler is set up, the interrupt is activated and the real-time clock module is initialized. In case there is no external clock for synchronization, the clock simply starts at zero. Furthermore, the initial pulse configuration is loaded.

2 The current pulse and system mode configuration is written to the TISSs. Initially the configuration contains only the guaranteed pulses. Note that the configuration is not activated immediately.

3 Finally, the configuration is marked "valid". This informs the TISSs that a configuration switch must be performed at the defined reconfiguration phase. Since there is no way to signal all TISSs at once, they must be signaled sequentially. Care must be taken that this step is completed for all TISSs before the reconfiguration phase. If the setup of some TISSs is incomplete at the reconfiguration phase, the configuration becomes inconsistent and "guaranteed" pulses may fail. To avoid such a situation, this step is omitted if the reconfiguration phase is too close by.

4 While the RMA processes the host requests, the TNA can be put in sleep mode. However, it is not necessary to wait until the whole allocation message has been received, so the reception of a fragment wakes up the TNA. This way the idle time can be reduced.

5 Each fragment carries the description of a pulse. So each newly received pulse description is checked for collisions with previously received descriptions.

6 If no collisions between pulses were found other properties of the resource allocation are checked. For example it must be checked that the service levels of RMA and TNA equal "1". If more fine-grained service level settings are supported, it must be checked if the service levels are high enough that the timing requirements of RMA and TNA are met.

7 The verification result is put into the message buffer to be sent to the RMA, which in turn forwards it to the hosts. This must happen before the new configuration becomes active so that the hosts can prepare for the new configuration.

8a If the resource allocation is valid the reconfiguration process is started.

8b Otherwise nothing happens and the TNA waits for the next allocation message.

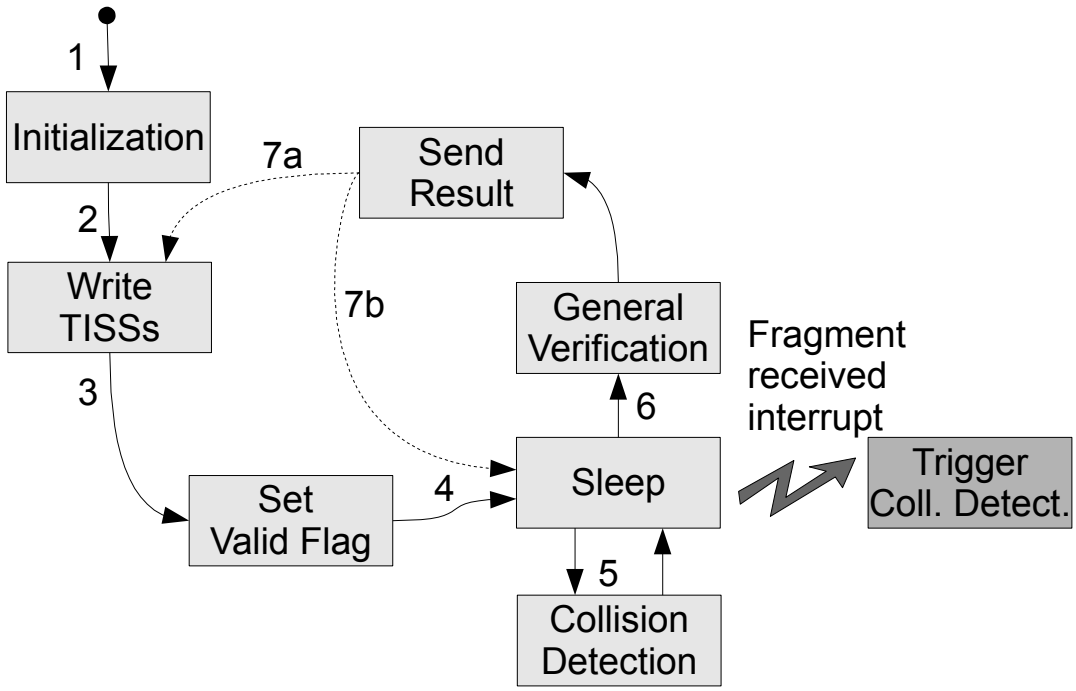


Figure 5.22: TNA program flow

Verification Algorithm

Recall that the TNA has to verify three items.

A Pulses are primarily ordered first by ascending pulse period and then by ascending phase.

B All *guaranteed pulses* are scheduled and their properties are correct.

C No collisions exist in the pulse schedule.

A The first item is easy to solve. For every received pulse definition (except the first one) it is checked whether the period value is higher than the one of the previous pulse. If they are equal the phase of the new pulse must be higher, otherwise the ordering requirement is violated.

B Thanks to the `Guaranteed Index` field the second item is trivial as well. On reception of a pulse definition the `Guaranteed Index` field is checked. If its value is not "-1" a lookup of the pulse definition of the corresponding *guaranteed pulse* is done, which is then compared to the received pulse. The `Phase` must be within `LowBound` and `UppBound`. All other values (`Period`, `FragPeriod` `D-Port`*`0-7`*, `F`*`0-7`* and `SendMC`) must match exactly. Otherwise a `MISMATCH` (cf. Table 5.12) error is generated.
To detect missing *guaranteed pulses*, a bitfield is used. There is one bit for each *guaranteed pulse*. The bitfield is initialized to zeros. For each matching *guaranteed pulse*, a bit is set to "1". In the end all bits must be "1" otherwise the `MISSING` error is returned. In the current implementation less than 64 *guaranteed pulses* are supported. Therefore two 32-bit words suffice to store the bitfield.

C As pointed out during the presentation of the verification algorithm in Section 4.7.1, to find out whether a pulse schedule is conflict-free each pair of pulses in the schedule is checked for a collision. This results in $\frac{n^2-n}{2}$ checks for n pulses. In Section 4.7.1, the 4 tests required for each such check were introduced. Here, the implementation of each test is described.

First, it should be noted that the data structure used by the algorithm to represent a pulse is identical to the definition of the pulses in the RMA-to-TNA interface (cf. Section 5.5.6). Thus, no extra processing is required when data is retrieved from the NoC input buffer. The structure is repeated in Figure 5.23 for convenience.

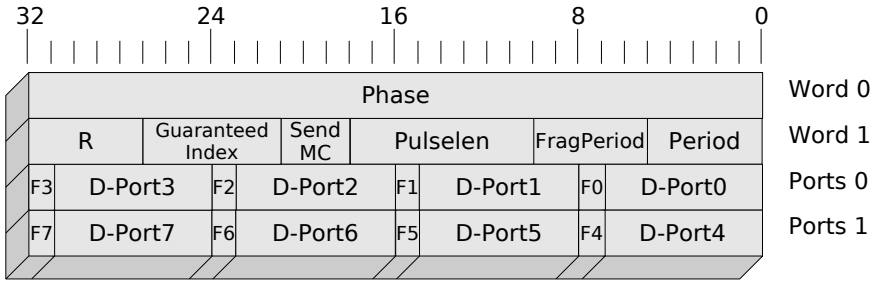


Figure 5.23: Data structure for pulsed data streams in the TNA

In the following description of the individual test conditions as well as in the code listings, the pulse with shorter Fragment Period will be referred to as *SFP* while the pulse with longer Fragment Period will be called *LFP*. Analogous, *SPP* and *LPP* will be used to indicate the pulses with shorter or longer Pulse Period. Note that *SFP* and *SPP* are the same pulse in most cases, but exceptions are possible.

Test 1 Test whether the "Same Period Constraint" applies. First, the `Period` values of both pulses are compared. If unequal the test succeeds. Otherwise it must be tested whether the pulses have a host in common. This test can be efficiently performed by combining the `D-Port` and `F` fields to the two 32-bit words `Ports0` and `Ports1` (see Figure 5.23). This reduces the test to the expression found in Listing 5.2. Recall that flags `F0-7` are "0" if the port is connected. The bitwise OR of the `F` flags evaluates to "0" if the port is active in both pulses which means that the pulses have a common host. The bitwise OR is performed for `Ports0` and `Ports1`. Then a bitwise AND is used to preserve any zeros already detected. In the final result all flags are "1" in case the pulses have no host in common.

```
1 boolean have_common_host(PSD p1, p2)
2 {
3   return ((p1.Port0 OR p2.Port0) AND (p1.Port1 OR p2.Port1)
4           AND 0x80808080) <> 0x80808080
5 }
```

Listing 5.2: Test for common host of two pulses

Test 2 Test whether the fragments of the two pulses are interleaved. The phases of both pulses are projected to the shorter Fragment Period. If the projected phases are not equal, it is impossible that two fragments of the pulses are in the same time-slot and the test succeeds.
Since all period lengths are a power of 2, the projection can be done using a bitwise AND operation. `PeriodLength-1` gives the mask for the AND operation. Example values can be found in Table 5.16.

```
1 boolean interleaved(PSD sfp, lfp)
2 {
3   fbitmask = PERIOD_LEN(sfp) - 1;
4   return ((sfp.phase AND fbitmask) <> (lfp.phase AND fbitmask));
5 }
```

Listing 5.3: Test for pulse interleaving

Test 3 Test whether one of the pulses fits between two fragments of the other pulse. The code is shown in Listing 5.4. For this test, the phase of the *SFP* is first aligned with the phase of the *LFP* (Line 3) and then projected to the fragment period of the *LFP* (Line 4). Thus, a check whether the duration of the *SFP* surpasses the Fragment Period Length of the *LFP* suffices to decide the test.

Period	Length Bitmask
0	10000000000000000000000000000000 11111111111111111111111111111111
1	10000000000000000000000000000000 11111111111111111111111111111111
30	100 11
31	10 1

Table 5.16: Period values and masks

```
1  boolean fitbetween(PSD sfp, lfp)
2  {
3      projected = (sfp.phase - lfp.phase)
4          AND (PERIOD_LEN(lfp.)-1);
5
6      return (projected + MSG_DURATION(sfp) < PERIOD_LEN(lfp));
7  }
```

Listing 5.4: Test for fitting of a pulse fits between two fragments of another pulse

Test 4 Verify that the pulses do not overlap. See Listing 5.5 for the code. Using the same technique as in test 3, the phase of the *LPP* is aligned with and projected to the pulse period of the *SPP*. After that, all that is left to do is to check whether the *LPP* actually starts after the end of *SPP* and that *LPP* ends before the next instance of *SPP*.

```
1  boolean overlap(PSD spp, lpp)
2  {
3      project = (lpp.phase - spp.phase) AND (PERIOD_LEN(spp.period));
4
5      if(project < MSG_DURATION(spp)) return 0;
6
7      if(project + MSG_DURATION(lpp) > PERIOD_LEN(spp.period)) return 0;
8      else return 1;
9  }
```

Listing 5.5: Test for pulse overlapping

Chapter 6

Results

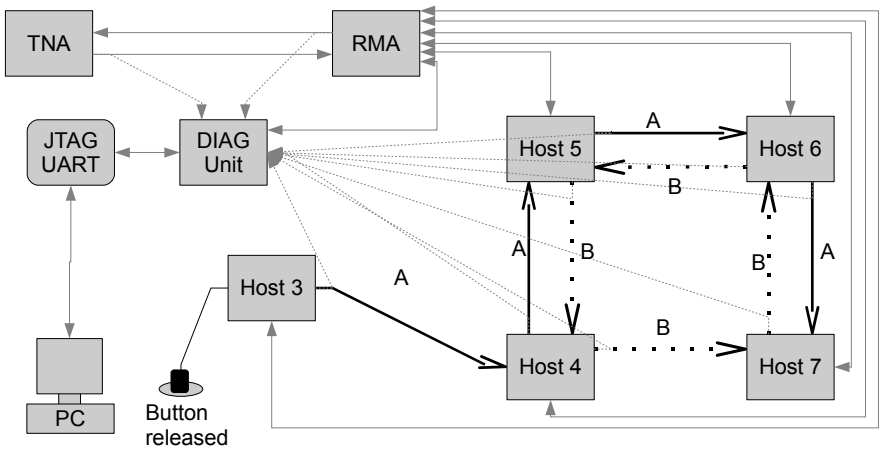
This chapter presents an evaluation of the performance of dynamic resource management in the TT-SoC described in this thesis. The main part is concerned with the evaluation of properties of the scheduling algorithm that is the heart of the resource management system.

6.1 General Function

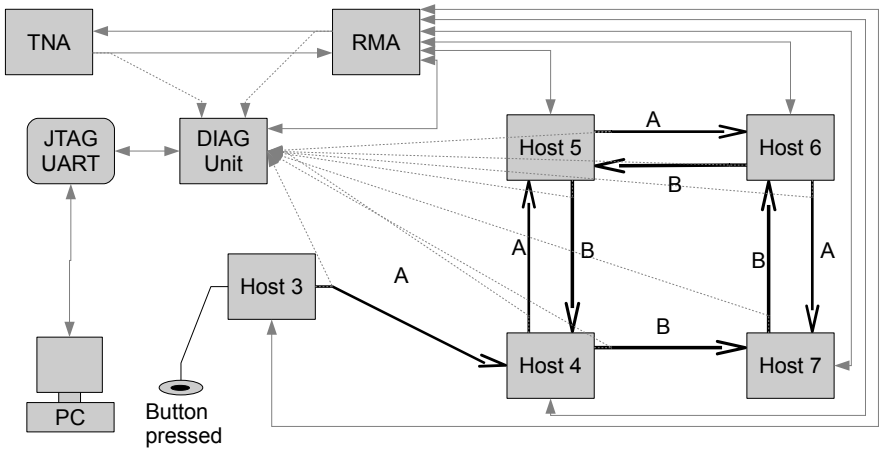
The implementation as presented in Chapter 5 was tested on the Stratix FPGA Board with a small test application. The test application is kept rather simple. Its structure can be found in Figure 6.1. The pulses defined for this application are listed in Table 6.1.

There is one push-button that is connected to Host 3. The state of the button is sent using pulses marked with an "A". The state is first sent to Host 4 which forwards it to Host 5 until it reaches Host 7 after 4 hops. Depending on the state of the button the hosts send requests to the RMA to either activate or deactivate their outgoing pulse marked with a "B". The pulses of type "B" carry an integer that is incremented at each host. If the button is pressed this integer constantly increases with each period, otherwise it remains constant which is also the case if a single host in the ring is non-functional. All messages are also received at the Diagnostic Unit (DU), where the system is supervised. The state of the hosts and the current value of the "B" pulses is observed via the JTAG UART console on the PC.

After pressing and holding the button, the DU reports that the RMA scheduled all 31 pulses from Table 6.1 and the success of the verification by the TNA. As expected, the integer value starts counting. Upon release of the button the 4 pulses of type "B" are no longer scheduled and 27 pulses remain and the integer counter



(a)



(b)

Figure 6.1: Test application overview

stops.

Furthermore the DU outputs the timing properties of the actions performed by RMA and TNA. The actions along with the execution time values are listed in Table 6.2 and are explained below.

Pulse Set Generation Before being able to construct a schedule, the RMA goes through the definitions of the pulses and collects all pulses that were requested by the hosts to a list for the PSD scheduler.

Schedule With the list of active pulses, the RMA invokes the scheduling algorithm, that assigns the phases for all pulses. Furthermore it stores the results to the output buffer for transmission to the TNA.

#	Function
1-6	Host to RMA requests
7	RMA to TNA config
8	TNA to RMA result
9-14	RMA to Host result
15-20	RMA to Host port mapping
21-22	DU control pulses
23-27	Type “A” pulses
28-31	Type “B” pulses

Table 6.1: Pulses in the Test Application

Cache-Flush The RMA CPU uses a data cache. Therefore, when the output buffer is completely written, some data may be left in the cache. The cache-flush assures that the output buffer is up-to-date.

Verify Overtime The actual time the TNA required for verification is not available since the verification algorithm starts running when the first fragment is received and pauses while waiting for the next fragment. The time given here denotes how much time after the last received fragment is required for the completion of the verification.

RMA Slack This is how much time to the RMA deadline was left. The deadline is the sending instant of the first fragment of the configuration to the TNA. It is one parameter of the reconfiguration timing which is determined in the design phase. If the deadline is missed, the reconfiguration process is delayed by one cycle of the reconfiguration period. The reconfiguration timing was discussed in Section 5.4.

TNA Slack This is how much time to the TNA deadline was left. The deadline is the sending instant of the result of the verification to the RMA. It is part of the reconfiguration timing parameters. If the deadline is missed, there is the risk that the TNA is outpaced by the RMA. If the RMA constantly sends new allocations and the TNA fails to verify them in time, no reconfiguration takes places at all.

6.2 In-Depth Tests

In this section the algorithmic functions that make up the dynamic resource management system are evaluated. This concerns mainly the scheduling and verification procedures. The different tests try to point out the achievable performance

Step	Time [ms] for 31 pulses (button pressed)	Time [ms] for 27 pulses (button released)
Pulse Set Generation	0.1305	0.1195
Schedule	3.0131	2.4867
Cache Flush	0.0025	0.0024
RMA Total	3.1461	2.6086
RMA Slack	8.5661	9.1036
Verify Overtime	0.1617	0.0817
Write TISSs	0.3445	0.3121
TNA Total	0.5062	0.3938
TNA Slack	10.1750	10.5315

Table 6.2: Execution Times for the Test Application

from different points of view. The first section explains the reference marks used in the evaluation of the scheduling algorithm derived from the fundamental limits of the system. Section 6.2.2 gives details about time and space complexity of the functions, Section 6.2.3 and Section 6.2.4 discuss the resource efficiency of the scheduling algorithm. Finally some ideas of possible improvements are presented in Section 6.2.5.

6.2.1 Fundamental Limits

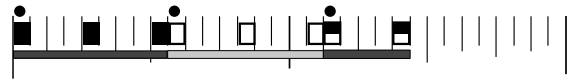
To be able to judge the capability of the scheduling algorithm to use time-slots efficiently, reference marks are required to which the results can be compared. The best reference mark is, in fact, an optimal scheduler. An optimal scheduler always finds a solution provided it exists. Thus, the degree of "non-optimality" of the *PSD Scheduler* is the portion of pulse sets that can be scheduled by the optimal scheduler but not by the *PSD Scheduler*. With PSDs however, it is difficult to build an optimal scheduler since PSDs may be interleaved which leads to a huge number of possible arrangements. Therefore, in this analysis, the limits imposed by the resources were chosen for evaluation of the *PSD Scheduler*. The limited resources are on the one hand the number of available time-slots on the bus and on the other hand the "Same Period Constraint" (see Section 4.4).

Limit 1: Time-Slots The network implementation presented in the previous chapter is able to transmit $2^{23} = 8388608$ fragments per second. The same number is assumed for the tests, although during this theoretic evaluation any power of 2 could be used for this system parameter. The sum of all fragments of all pulses can not exceed this value.

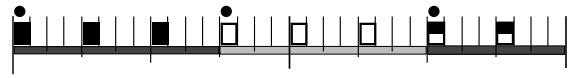
Limit 2a: Same Period Constraint Since two pulses of the same period can not be handled by a single host at the same time, they must not be interleaved, instead they have to be scheduled one after the other. This means that the summed duration of pulses of the same period with a common host must not exceed the duration of the period.

Limit 2b Limit 2a can only be reached if the pulses are packed together tightly. It requires that the first fragment of a pulse resides in the time-slot after the last fragment of the previous pulse as shown in Figure 6.2(a). It is very unlikely that the scheduling algorithm is able to pack pulses this way. A more realistic arrangement can be seen in Figure 6.2(b). This is how the scheduling algorithm would arrange the three pulses if only those three were in the set. So Limit 2 is split in a part "a" which is the limit possible if an optimal scheduler was used and a part "b" which is the Limit expected from our scheduler.

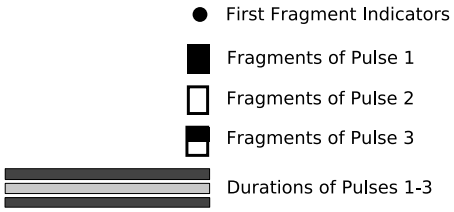
The calculation of Limit 2b is similar to Limit 2a, but the duration of a pulse is prolonged. This is illustrated by the differently sized grey bars in Figures 6.2(a) and 6.2(b). In Figure 6.2(a) the duration of a pulse ends just after its last fragment. It is calculated as $FragmentPeriodLength * (FragmentCount - 1) + TimeSlotLength$. In Figure 6.2(b) the duration of a pulse includes also the rest of the fragment period begun by the last fragment. It is calculated as $FragmentPeriodLength * FragmentCount$.



(a) Improbable pulse arrangement: 9 free slots in this period



(b) Realistic pulse arrangement: 0 free slots in this period



(c) Legend

Figure 6.2: Arrangements of pulses within a period

6.2.2 Test 1: Pulse Set Size

Scheduling Algorithm

In the first test the impact of the number of pulses on the execution time was observed. For this the test set was built up from many small pulses with different periods from different hosts so that a large number of pulses is possible.

The basic test set consists of 32 pulses listed in Table 6.3. For test sets with more than 32 pulses the pulses of the basic test set are used multiple times.

Any two pulses in the basic set differ in period length or do not have common hosts, so the "Same Period Constraint" does not apply to any such pair. However, it does apply when multiple instances of the basic test set are used. But then, the restrictions imposed by the constraint are very low in relation to the number of pulses.

No	Period (Length in slots)	Frag. Period (Length in slots)	Frag. Count	Duration in slots	Hosts	Slots per sec
1	2^{-6} (131072)	2^{-13} (1024)	5	4097	1,2	320
2	2^{-6} (131072)	2^{-13} (1024)	5	4097	3,4	320
3	2^{-6} (131072)	2^{-13} (1024)	5	4097	5,6	320
4	2^{-6} (131072)	2^{-13} (1024)	5	4097	7,8	320
5	2^{-7} (65536)	2^{-14} (512)	5	2049	1,2	640
6	2^{-7} (65536)	2^{-14} (512)	5	2049	3,4	640
7	2^{-7} (65536)	2^{-14} (512)	5	2049	5,6	640
8	2^{-7} (65536)	2^{-14} (512)	5	2049	7,8	640
...						
29	2^{-13} (1024)	2^{-20} (8)	5	33	1,2	40960
30	2^{-13} (1024)	2^{-20} (8)	5	33	3,4	40960
31	2^{-13} (1024)	2^{-20} (8)	5	33	5,6	40960
32	2^{-13} (1024)	2^{-20} (8)	5	33	7,8	40960

Table 6.3: Pulses of the Basic Test Set

Limit 1 Summing up the number of slots per second for all pulses gives the bus usage of the test set. For our 32 pulse test set this sum is: $326400 = 5 * 4 * (2^6 + 2^7 + 2^8 + ... + 2^{13})$. The division of available slots by used slots tells us how many instances of the pulses in the test set can be scheduled successfully: $\frac{8388608}{326400} = 25.7$. Multiplied by the size of the test set the value for Limit 1 is obtained as 822 pulses.

Limit 2a To calculate Limit 2a, each period is filled with as many pulses as possible, one pulse tightly packed to the other. The number of pulses that fit into

a certain period is given by: $\lfloor \frac{PeriodLength}{PulseDuration} \rfloor$. As mentioned before each pulse of the 32 pulse test set counts for a different period. So the limit is calculated separately for each pulse and then summed up. Due to rounding down, the maximum number of pulses equals 31 for all periods. Therefore, Limit 2a is $32 * 31 = 992$.

Limit 2b For Limit 2b the pulse duration is modified. The value then calculates as: $\lfloor \frac{PeriodLength}{FragmentCount * FragmentPeriodLength} \rfloor$, which equals 25 for all 32 types in the basic test set. Therefore, Limit 2a is $32 * 25 = 800$.

Results Two aspects are the most important in the evaluation. On the one hand, how close to the theoretical limit is the scheduling algorithm able to successfully operate, and on the other hand, how does the number of pulses affect execution time. Both questions can be answered by looking at Figure 6.3. The size of the pulse set is plotted against the x-axis. The grayed regions mark the pulse sets where not all of the pulses could be scheduled. We would expect that there is one last schedulable pulse set and that all larger sets fail. In other words that there is one continuous grey region. This is not the case, however. The reason is that if we append a pulse with a short fragment period it will be treated very early and may have severe effects on the tree structure. In Figure 6.4 this effect is illustrated. The first set that fails has 709 pulses. Its tree structure is shown on the left-hand side (Figure 6.4(a)). On the right-hand side (Figure 6.4(b)) the set with 718 pulses is shown. This set is the first of the long successful series after the first grey bar.

The same effect is responsible for the jumps in the execution time. The largest jump is 44.8 ms between the sets of size 605 and size 606 (for details see Table 6.4). The corresponding tree structures can be found in Figures 6.5(a) and 6.5(b). In the tree with 605 pulses, there are many chains that reach down to the middle, while the tree with 606 pulses is broad, and only some chains reach as far down. Long chains mean that for each node of the chain, the chain had to be traversed in the scheduling algorithm, which obviously takes more time than if the tree spreads into short branches.

Set Size	Execution time	Limit 1 (%)	Limit 2a (%)	Limit 2b (%)
605	481.726	27.54	38.7	25.8
606	437.606	27.05	38.7	25.8
708	637.749	14.45	25.8	10.2

Table 6.4: Properties of remarkable Executions

Limits The first failure occurs with the test set containing 709 pulses. Table 6.5 shows how close to the limits the algorithm has been successful before failing.

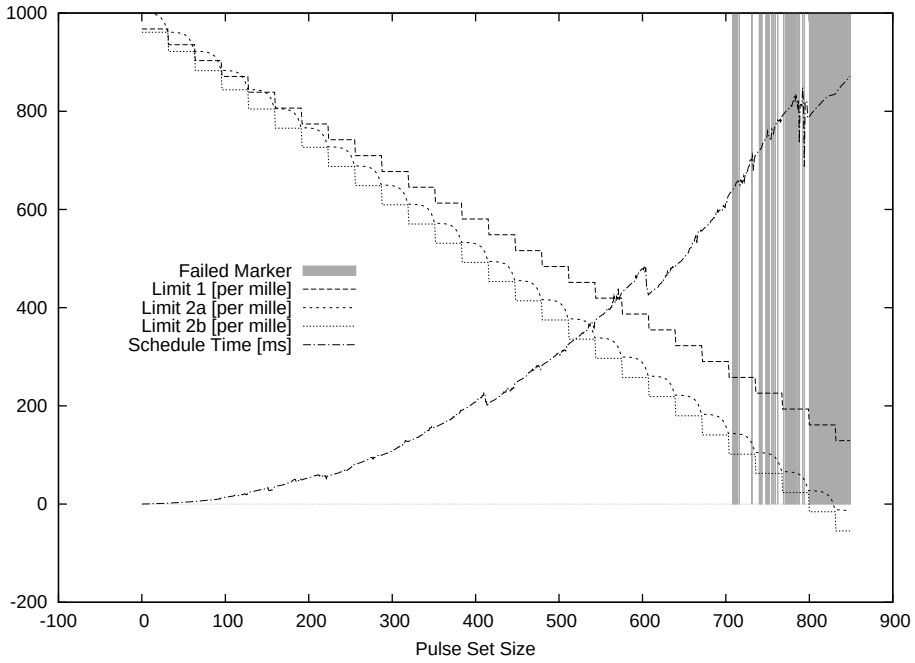
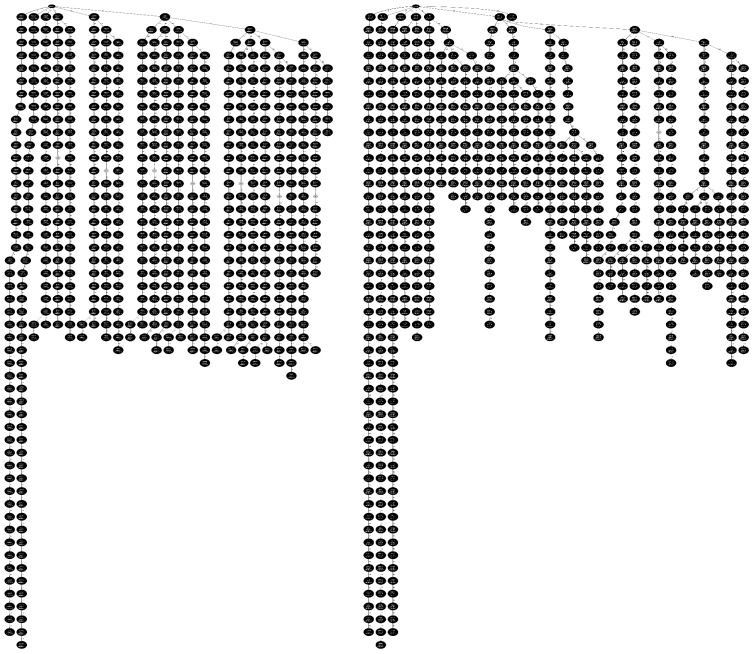


Figure 6.3: Execution-time of the scheduling procedure

(a) 709 pulses : unschedulable (b) 718 pulses : schedulable

Figure 6.4: Effect of the tree structure on schedulability

The results are quite well. However, the primary focus of this test were the time requirements for large pulse sets.



(a) 605 pulses : 481.726ms (b) 606 pulses : 437.606ms

Figure 6.5: Effect of tree structures on the execution-time

Limit 1	85.55%
Limit 2a	74.2%
Limit 2b	89.8%

Table 6.5: Limits Test 1

Time Complexity From Figure 6.3 we conclude that the time complexity of the algorithm is $\mathcal{O}(n^2)$. The result is not very surprising as the algorithm tries "difficult" locations first. Mostly, it travels through almost all nodes before finding a suitable location.

Space Complexity The memory required by the algorithm is linear to the number of pulses ($\mathcal{O}(n)$). This is because the algorithm does not save the leaves of the pulse trees in memory, it only stores the nodes and derives the leaves from the node's properties whenever required.

Scheduling Algorithm Helper Functions

Figure 6.6 shows the execution-time of the helper functions for increasing pulse set sizes.

Here is a short summary of their purpose:

Sorting Sorts the pulses according to the criteria listed in Section 4.5.3 before they can be scheduled.

Store Stores the pulse schedule to the transmission buffer.

Cleanup Releases temporary memory allocated during scheduling.

Analyze Calculates an indicator value used for sorting as described in Section 4.5.2.

Compared to the actual scheduling algorithm (Figure 6.3) the impact of those functions on the total execution-time is insignificant. Except for the sorting they are executed in linear time.

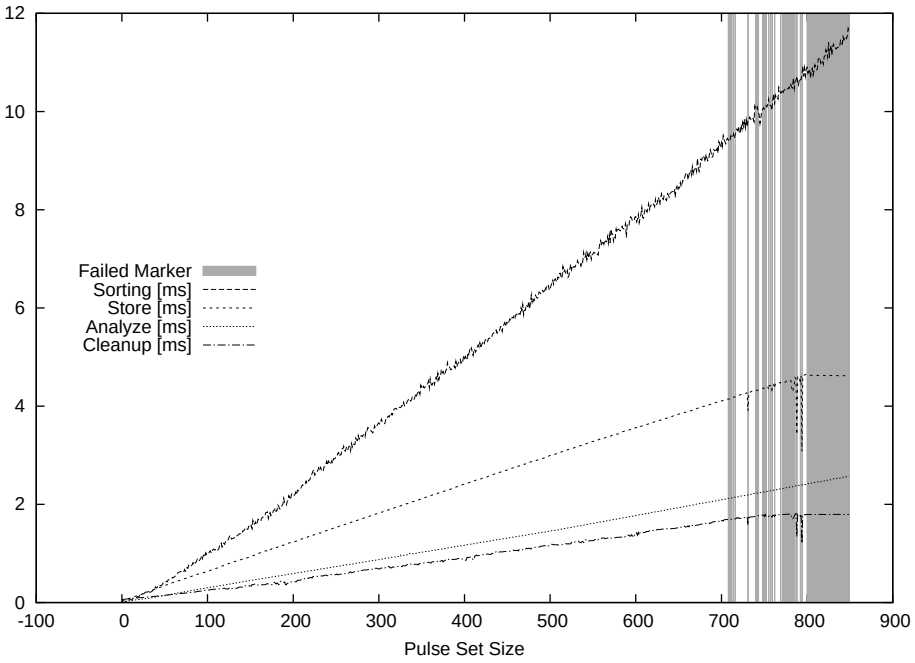


Figure 6.6: Execution-time of helper functions

Verification Algorithm

The execution-time of the verification algorithm is plotted in Figure 6.7 for all three versions of the Nios II processor. For comparison the graph of the scheduling time is shown as well.

The time-complexity of the verification algorithm is $\mathcal{O}(n^2)$. This is because every pair of pulses is checked for collisions. So, $\frac{n(n-1)}{2}$ checks are performed. This is confirmed by Figure 6.7 where the graphs look like quadratic functions.

There are no jumps in the graphs of the verification algorithm because it does not depend on the tree structure, which is the source of the jumps in the scheduling time graph. In fact, the verification algorithm ignores the fact that a tree structure is used for scheduling.

The difference in the execution time between the Nios architectures is a constant factor. The medium Nios is 3.4 times faster than the tiny Nios and the fast Nios Nios is almost 4 times faster than the tiny Nios and 1.17 times faster than the Medium Nios.

The scheduling algorithm was run on a fast Nios processor. Compared to the verification on a fast Nios, scheduling is about 4 times slower than verification.

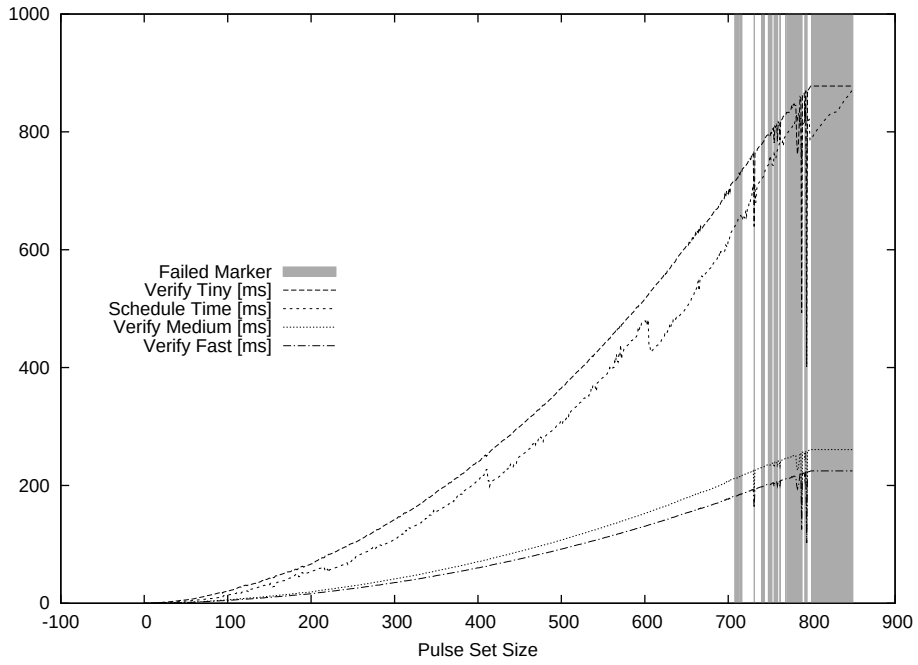


Figure 6.7: Execution-time of the verification algorithm on different Nios II architectures

6.2.3 Test 2: Limit 1

This test evaluates the performance of the scheduling algorithm with respect to the theoretical limit 1. In other words it is tested how much of the bandwidth can be used for the application and how much bandwidth should remain free to

be sure that a pulse set can be scheduled.

For this test random test sets are generated on which the scheduling procedure is executed. In this test the "Same Period Constraint" (Limit 2) is ignored to put the focus specifically on Limit 1.

Description of Test 2

The test is started with a single pulse in the test set. If the scheduling procedure succeeds, another pulse is added to the set and scheduling is retried. The process is repeated until the scheduling fails for the first time. Then, the percentage of theoretically free time-slots is calculated similar to the calculation of the Limit 1 values in Test 1.

The Test is repeated 100,000 times with different random seeds in order to get meaningful results.

Random Test Sets

Test sets are generated by pseudo-random number generators. A systematic approach to test set generation was not taken to reduce coherency between the pulses in the test sets. With systematic approaches there is the risk of testing only some kinds of relations between pulses. However, random tests may also miss very rare problematic cases.

For random number generation the Boost library [Daw04] implementations were used. Uniform random numbers are generated using the Mersenne Twister MT19937 [MN98]. Non-uniformly distributed random variables are obtained by applying transformations on the uniform random variable.

All properties that define a pulsed data stream (cf. Section 2.2) must be determined by random numbers. Remember that period durations are expressed as negative powers of two. The randomized properties are:

Pulse Period The pulse period is a uniformly distributed random number in the interval [0, 15]; that is the range $30.52\mu s$ to $1s$

Fragment Period The success of the algorithm depends heavily on the relationship between pulse period and fragment period values. The algorithm performs best if the fragment period is the pulse period plus a constant offset for all pulses in a pulse set. It performs worst if pulses with long pulse periods have very short fragment periods and vice-versa. However, the worst case is not very realistic, but

can occur if the fragment period is chosen using a uniformly distributed random value.

In order to obtain meaningful results the test was done three times (2a, 2b, 2c) with different policies for the choice of the Fragment Period.

a) Constant Factor The Fragment Period value is the Pulse Period increased by 5 (that is 2^5 times shorter than the Pulse Period). As explained above this yields the best results.

b) Normal Distribution The Fragment Period is a random number that is normally distributed around $PulsePeriod+5$. See Table 6.6 for the parameters of the normal distribution. Good results are expected for this Test.

c) Uniform Distribution The Fragment Period is a random number that is uniformly distributed on the closed interval [PulsePeriod+2, 20]. Average results can be expected.

Mean (μ)	Pulse Period + 5
Std. Dev. (σ)	2

Table 6.6: Parameters for the Normal Distribution used for the Fragment Period

Pulse Length (Number of Fragments) The maximum number of fragments of a pulse is determined by the factor

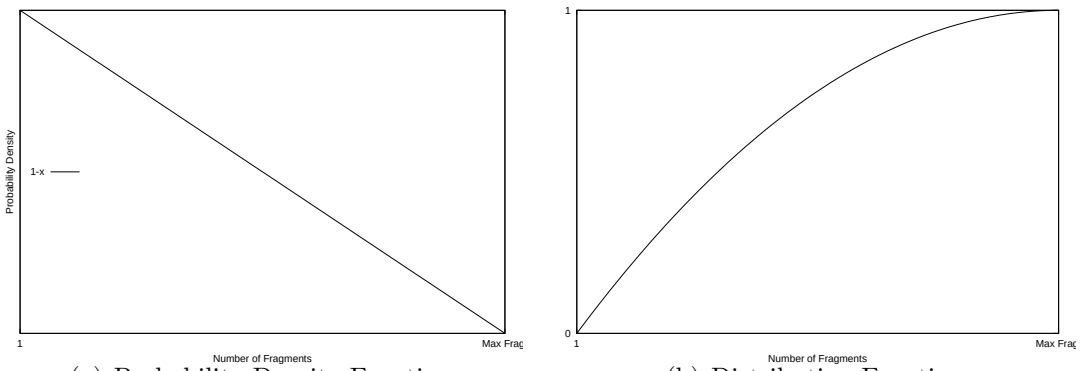
$$\frac{PulsePeriod}{FragmentPeriod}.$$

However, it is unlikely that the maximum number is used for a pulse because it disallows any other pulses of the same period on the same host ("Same Period Constraint"). Thus, the random variable for the number of fragments was modeled using a descending probability density function, which makes very high numbers unlikely. The probability density function is plotted in Figure 6.8(a), the distribution function in Figure 6.8(b).

Results of Test 2

The result of each test run is the percentage of free time-slots where the scheduling algorithm could no longer find a valid schedule.

The percentage scale was divided to classes of 1%. For each test run the count of the respective class was increased. The resulting histogram after 100,000 runs is



(a) Probability Density Function (b) Distribution Function

Figure 6.8: Probability functions for the number of fragments

shown in Figure 6.9. The height of the bars indicate the portion of test runs that failed when confronted with the percentage of free resources shown on the x-axis but succeeded when more resources were free.

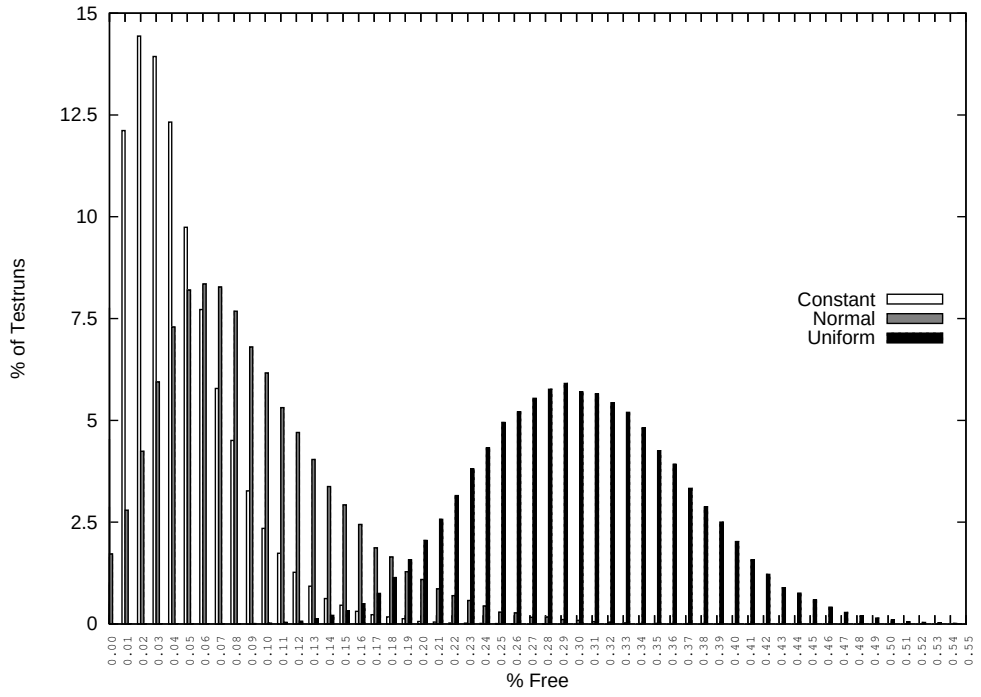


Figure 6.9: Histogram of the results of test 2

The result is not surprising. When choosing the Fragment Period with constant offset from the Pulse Period more than half of the test runs failed only if there were less than 3% free time-slots corresponding to a bus load of more than 97%. The observed worst case was 30% (cf. Table 6.7). The test-runs form a high peak

at the left hand-side which means that there are not many test-runs that fail with much resources still free.
With normally distributed Fragment Period values, the result is still quite good, the worst case of 41% is acceptable. The peak is broader, but still centered at a good resource percentage value.
With the rather unrealistic uniformly distributed Fragment Period values quite much bandwidth is wasted but it is still acceptable for applications that do not heavily depend on the NoC. The peak is centered around 29% and reaches into percentages of more than half.

Fragment Period Policy	Worst % Free	Best 10% Quantile
Constant Offset	30%	0%
Normal Distribution	41%	2%
Uniform Distribution	58%	21%

Table 6.7: Worst case and best case results of test 2

Figure 6.10 shows the cumulative sum of the classes representing the distribution functions. From these functions, the quantiles can be read out. The y-axis displays the portion of test-runs that could successfully schedule test sets where the percentage of resources given on the x-axis was free.

Following the horizontal line at 0.9 we come across the intersections at 0.085, 0.16 and 0.385 which means that 90% of the test-runs succeed when given 8.5% free resources for constant offset fragment periods, or 16% free resources for normally distributed fragment periods or 38.5% free resources for uniformly distributed fragment periods.

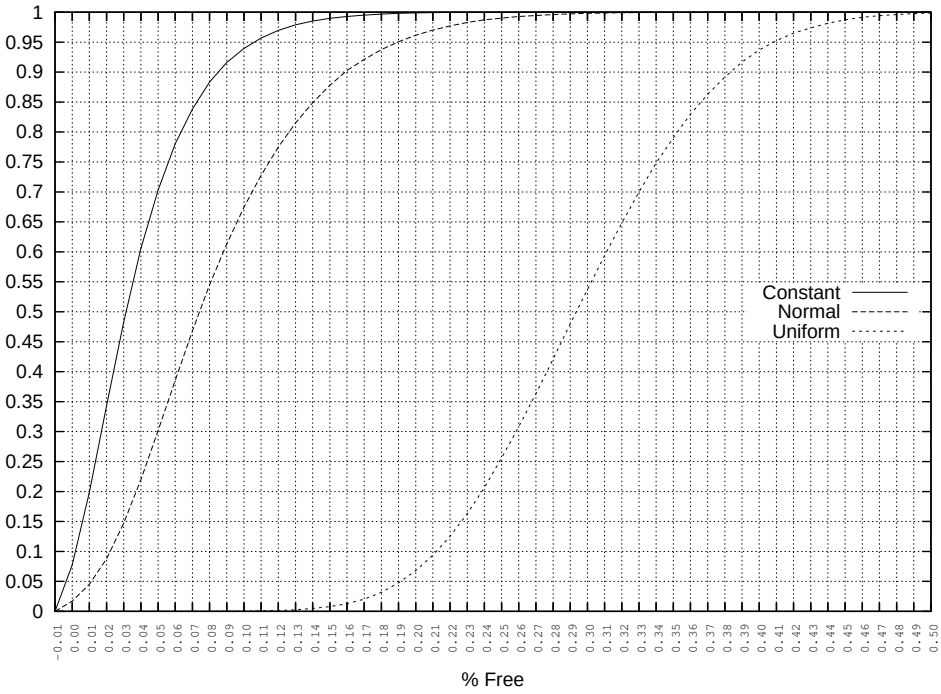


Figure 6.10: Distribution functions of the results of test 2

6.2.4 Test 3: Limit 2

This test is similar to the second Test but this time the "Same Period Constraint" is no longer ignored. The Test aims to observe how tightly the algorithm can pack pulse blocks of the same period.

Like in Test 2 random test sets are used for the analysis and each test run starts with a set of one pulse which grows until no schedule can be found. The properties of the random test sets are equal to those in Test 2.

Results of Test 3

Limit 2 is divided in a part "a" and a part "b" as explained in Section 6.2.1. The results for both parts are plotted as histograms (Figure 6.11 and Figure 6.12) analogous to those in Test 2. The worst case values are summed up in Tables 6.8 for part "a" and Table 6.9 for "b".

In both histograms there are gaps in between the white bars. These gaps come from the reduced granularity if the Fragment Period is always $\frac{1}{32}$ of the Pulse Period. Adding 1 Fragment to a pulse adds $\frac{1}{32}th$ of the period (3.125%) to the Limit.

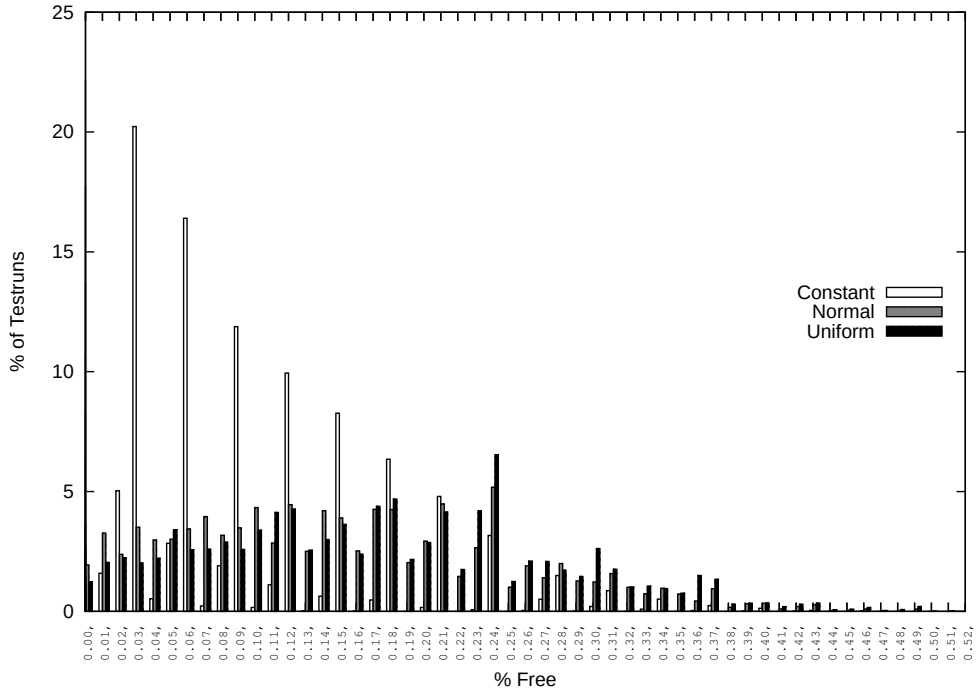


Figure 6.11: Histogram of the results of test 3a

The histograms do not show the regularity of the histograms from Test 2. They do not look like bell curves as in the previous histogram. This is because the scheduling algorithm implements a structured approach to utilize time-slots efficiently, but it lacks a strategy that is capable of arranging pulses within a period equally well.

The results are not as good as those of Test 2. The worst case percentages (see Tables 6.8 and 6.9) are quite high. There is more uncertainty about how efficiently the resources can be used and a higher percentage of the resources is wasted.

Again, the distribution functions were plotted as well. They can be found in Figures 6.13 and 6.14. These functions make apparent that the Fragment Period policies have very small influence on the "Same Period Constraint". The individual functions do not differ as much as those in Figure 6.10 especially in the more realistic Limit 2b (Figure 6.14).

Fragment Period Policy	Worst % Free	Best 10% Quantile
Constant Offset	40%	2.5%
Normal Distribution	45%	2%
Uniform Distribution	48%	2%

Table 6.8: Results of test 3a

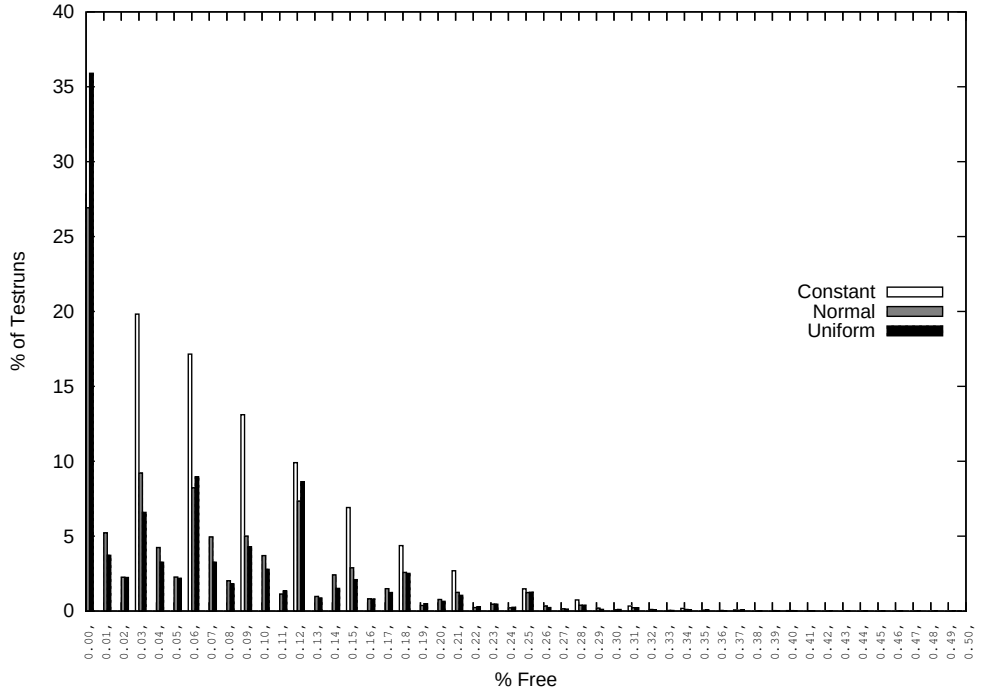


Figure 6.12: Histogram of the results of test 3b

Fragment Period Policy	Worst % Free	Best 10% Quantile
Constant Offset	49%	2%
Normal Distribution	59%	3%
Uniform Distribution	62%	4.5%

Table 6.9: Results of test 3b

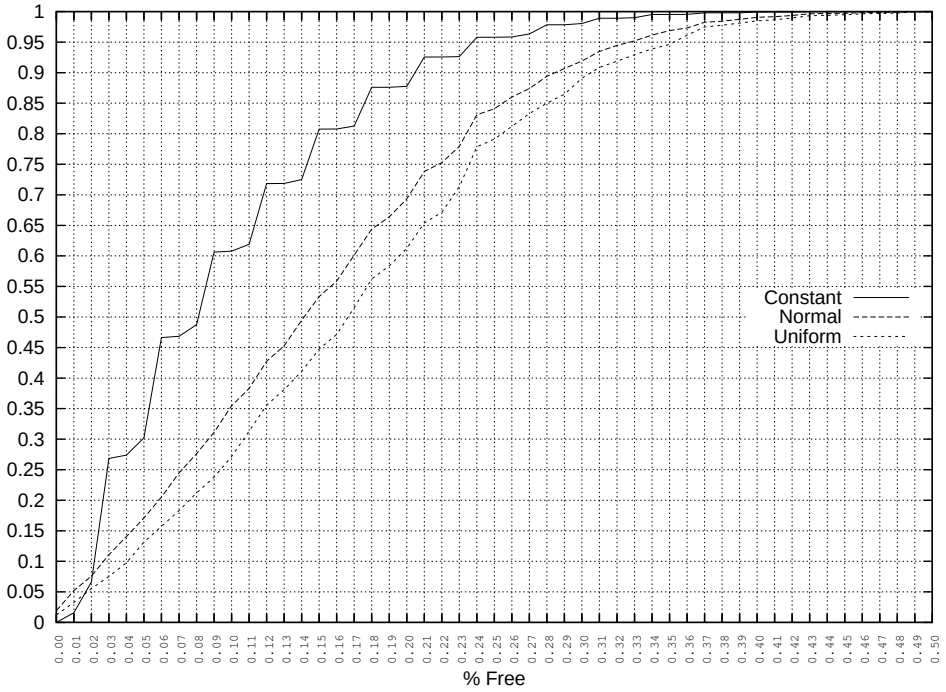


Figure 6.13: Distribution functions of the results of test 3a

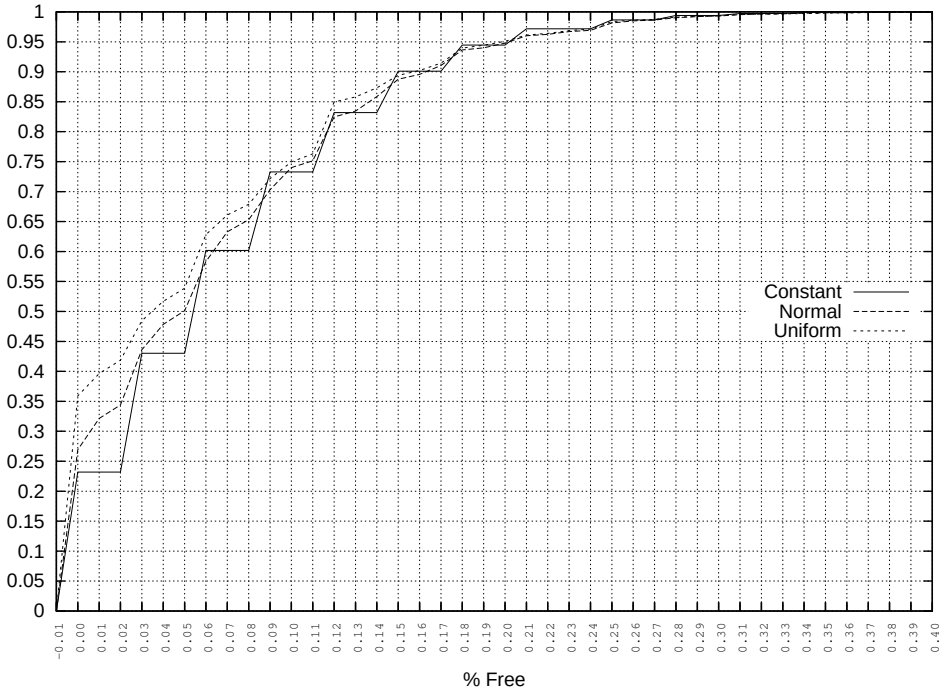


Figure 6.14: Distribution functions of the results of test 3b

6.2.5 Possible Improvements

The overall performance of the algorithm is good, but there is still potential for improvements.

One approach to reduce execution time would be to try to avoid that the algorithm has to traverse mostly the whole tree of scheduled pulses for each pulse that is to be scheduled. A possible solution could try to schedule more than one pulse in one traversal. Another possibility is to store the leaves rather than the nodes, so that they can be accessed directly.

The verification algorithm is kept simple to avoid bugs that may compromise the trusted behavior of the TNA. However, if performance is critical a more sophisticated verifier could exploit the fact that the pulse list from the RMA is sorted.

The greatest issue however, is that the algorithm does not perform very well with respect to Limit 2. The situation could be improved by extending the analysis phase before the actual scheduling. During analysis it is evaluated how much time of each period for a particular host is taken by the pulses. This information is used only to prioritize pulses of "congested" periods. An extended analyzer could arrange the pulses for each period in advance and put phase constraints on the pulses so that the scheduler keeps the arrangement. This way, inconvenient arrangements can be avoided. However, such an analyzer is quite complex as one pulse appears in the periods of multiple hosts. It is therefore not possible to focus on a single period at a time.

Chapter 7

Conclusion

7.1 Summary

The TT-SoC architecture, as presented throughout this thesis, has the potential to make SoC solutions attractive to safety-critical real-time applications. The architecture provides encapsulation, a property which is important for the independent development of different application subsystems. On the one hand, it allows the seamless integration of a new subsystem into an existing SoC design and, on the other hand, it prevents fault propagation over the boundaries of application subsystems. The architecture facilitates on-chip diagnosis through monitoring and reporting and may be operated in conjunction with other systems using gateway nodes.

In this thesis, a resource management scheme for the TT-SoC architecture was designed and implemented. It is the first FPGA implementation of the TT-SoC architecture and serves for the demonstration of the capabilities of the architecture. The resource management enables efficient use of resources (i.e. power, bandwidth, computational resources, ...) through resource sharing. A host in an SoC component is only allocated the resources it needs at the moment. They can be freed as soon as they are no longer required.

The establishment of a trusted region prevents resource conflicts in the running system at all times. All configuration values entering the trusted region are validated by the components of the trusted region (the TNA and the TISSs). The validation occurs not only in the value domain but also in time domain, which is especially important for network resources.

These mechanisms ensure that a misbehaving host cannot have adverse effects on the correct operation of hosts in different application subsystems. Thus, the required quality of service for application subsystems of high level of criticality can

be guaranteed even in the presence of uncertified application subsystems on the very same chip.
The support for resource efficiency is a key feature of the architecture and was the primary motivation for this thesis. A solution for the components involved in the resource management, namely the RMA and the TNA, was designed and implemented on an FPGA. The design of the RMA comprises the interfaces to and from application hosts, which is based on application modes to save network bandwidth, the interface to the TNA and the scheduling algorithm for the on-chip network.
The scheduling algorithm demonstrates how *pulsed data streams*, the communication primitive used in the TT-SoC architecture, can be handled efficiently using trees. It may inspire algorithms developed for ensuing implementations of the TT-SoC architecture.

For the TNA, the interface to the RMA was defined and the CP interface to the TISS was described. Furthermore, a simple and efficient algorithm for verification of the schedule of *pulse data streams* was explained, which is independent of the particular scheduling algorithm used in the generation.

The resource management cycle, the periodic process that enables dynamic allocation of resources, was explicated. The sequence of operations and their timing was presented for the implementation.

The implementation of the TT-SoC architecture was tested on current FPGA hardware and the performance was measured. The run-time of the algorithms and the quality of the results with respect to fundamental limits was surveyed.

The results showed that the goals claimed by the TT-SoC architecture (encapsulation, resource efficiency) are achievable with current technology. Even the early FPGA prototype delivered a performance that is adequate for today's applications. The efficiency of the online scheduling and verification algorithms allowed short reconfiguration cycles (32.5 ms) for quick adaptation to dynamic environments. It may be concluded that the TT-SoC architecture is fit for practical purposes.

7.2 Outlook

The implementation presented in this thesis is an FPGA prototype for the TT-SoC architecture. Its purpose was to elevate the capabilities of the architecture and to detect possibilities for improvements before more complex implementations were to be approached.

The main drawback of the presented implementation is that a bus is used for message exchange. The choice was made to reduce complexity and to be able to focus on the TT-SoC architecture as a whole. As a first improvement an alternate

version of the bottom layer of the NoC was started that is based on a switched network using a mesh topology.
Furthermore, it was observed that 128-bit long fragments limit the achievable maximum frequency too much (in this implementation: $f_{max} = 75MHz$). Smaller fragments at a higher speed are likely to achieve similar performance at less resource cost.
Both changes will require a modification of the resource management solution, in particular scheduling and verification of the communication resources. However, the concept and mechanisms of the resource management cycle remain valid.

The prototype implementation is a proof of concept of the TT-SoC architecture. The key properties claimed for TT-SoC (real-time support, encapsulation, support for resource efficiency, abstraction) were established that demonstrate the potential of the architecture. As such, with further research and refinements, it can be expected that the TT-SoC will gain importance for real-time applications, especially when subsystems of differing degree of criticality need to be joined in an SoC.

Appendix A

External Interfaces

A.1 TNA-TL Interface

The interface from the TNA to the TL is a simple memory interface with a 32-bit data bus, a 16-bit address bus and an interrupt request signal. Only 11 of the 16 bit of the address bus are significant the others provide address space for extensions. The most significant bits 15 and 14 in the address choose between control register or pulse data access (cf. Figure A.1).

If `R`="0" the pulse data memory is addressed. The pulse data memory buffers the data for the incoming pulse and the outgoing pulse. Remember that this TL cannot handle more than those two pulses. Data written to this area is sent via the outgoing pulse, incoming data can be retrieved by reading from the area. The buffer sizes are limited to the very basic requirements. The incoming buffer is 512 words (128 fragments) long to hold the resource allocation message from the RMA. The outgoing buffer comprises only 4 words (1 fragment) because the verification result is very short.

If `R`="1" the control registers are addressed. The bit `C` further differentiates between clock registers (`C`="1") and TL registers (`C`="0"). The clock registers are used to initialize the NoC clock. Those registers enable synchronization with an external clock and must be used to initialize the clock on startup even if no external clock is present. Details about their usage is available in [Eng07].

The resulting memory structure of the TNA-TL interface is illustrated in Figure A.2. The flags `M, I, N` can be reset by writing a logical "1" to their location.

`Fragment`	7-bit	Gives the index of the last received fragment. "0" is the first fragment.
`M`	1-bit	Modified flag. Is set if the content of the incoming buffer has changed.
`I`	1-bit	Interrupt flag. Is set upon reception of a fragment. Reflects also the state of the interrupt signal.
`N`	1-bit	New pulse flag. Is set upon reception of the first fragment.
`TIMEL`	32-bit	Access to the split second part of the NoC real-time clock. When `TIMEL` is read the upper 32-bit of the time stamp are buffered and can be retrieved by reading `TIMEH`. This allows to read a consistent 64-bit value.
`TIMEH`	32-bit	Access to the second part of the NoC real-time clock.

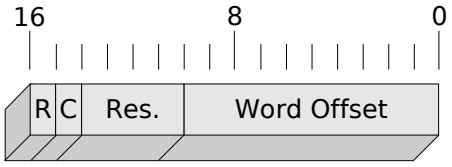


Figure A.1: TNA address bus

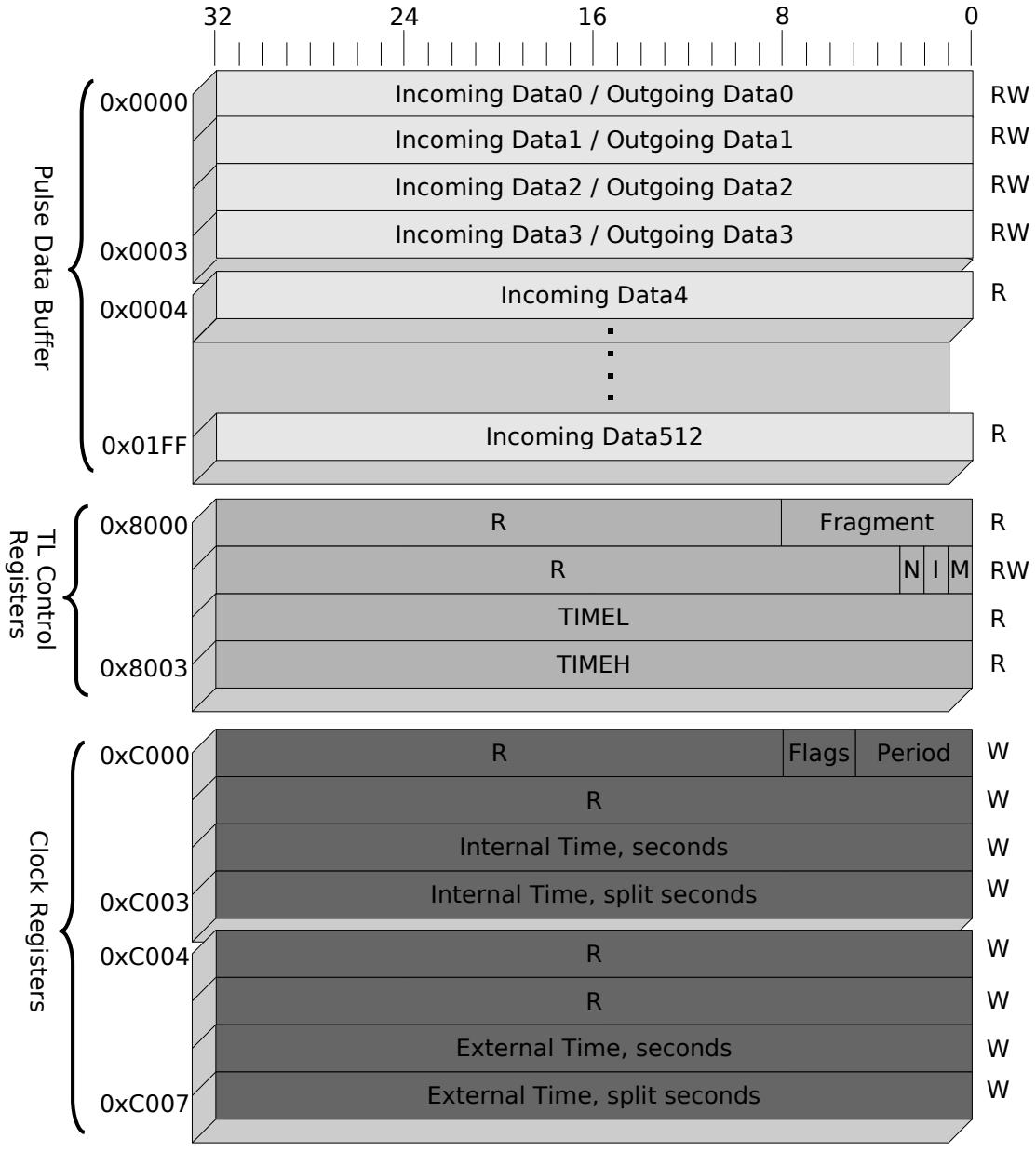


Figure A.2: TNA-TL memory interface layout

Appendix B

Acronyms and Abbreviations

ASIC Application Specific Integrated Circuit

CNI Communication Network Interface

CP Configuration and Planning

CPU Central Processing Unit

CSMA Carrier Sense Multiple Access

DAS Distributed Application Subsystem

DLL Data Link Layer

D-Port Data-Link Port

DSP Digital Signal Processor

DU Diagnostic Unit

FPGA Field-Programmable Gate-Array

IP Intellectual Property

ISR Interrupt Service Routine

L-Port Logical Port

LSB Least Significant Bit

MAC Medium Access Control

MEDL Message Descriptor List

MSB Most Significant Bit

NoC Network-on-Chip

NTP Network Time Protocol

OCP Open Core Protocol

OS Operating System

PCB Printed Circuit Board

PLL Phase-Locked Loop

PSD Pulsed Data Stream

RAM Random Access Memory

RMA Resource Management Authority

SoC System-on-Chip

TDMA Time-Division Multiple Access

TISS Trusted Interface Subsystem

TL Transport Layer

TMR Triple Modular Redundancy

TNA Trusted Network Authority

TT-SoC Time-Triggered System-on-Chip

UTC Coordinated Universal Time

VHDL Very High Speed Integrated Circuit Hardware Description Language

References

[Alt07a] Altera Corporation. *Avalon Memory-Mapped Interface Specification 3.3*, May 2007. 43

[Alt07b] Altera Corporation. *Nios Development Board Stratix II Edition Reference Manual*, May 2007. 41

[Alt07c] Altera Corporation. *Nios II Processor Reference Handbook*, May 2007. 65

[ANM+05] Prabhat Avasare, Vincent Nollet, Jean-Yves Mignolet, Diederik Verkest, and Henk Corporaal. Centralized end-to-end flow control in a best-effort network-on-chip. In *ACM Conference on Embedded Software (EMSOFT) 2005*, pages 17–20, Jersey City, 1 2005. ACM. 10

[Cla99] Jens Clausen. Branch and bound algorithms - principles and examples. http://citeseer.ist.psu.edu/683497.html, March 1999. 10

[Con02] Cristian Constantinescu. Impact of deep submicron technology on dependability of vlsi circuits. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 205–209, Washington, DC, USA, 2002. IEEE Computer Society. 16

[Daw04] Beman Dawes. Boost for visual C++ developers. *Visual Studio Technical Articles*, May 2004. 90

[DS94] M. DiNatale and J. Stankovic. Dynamic end-to-end guarantees in distributed real-time systems. In *Real-Time Systems Symposium*, 1994. 10

[Eng07] Gerhard Engleder. Time-triggered network-on-a-chip. Master's thesis, Vienna University of Technology, 2007. 47, 64, 103

[HCG07] Andreas Hansson, Martijn Coenen, and Kees Goossens. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *DATE '07: Proceedings of the conference on*

Design, automation and test in Europe, pages 954–959, San Jose, CA, USA, 2007. EDA Consortium. 10

[Hin99] Ralf Hinze. Constructing red-black trees. In *Proceedings of Workshop on Algorithmic Aspects of Advanced Programming Languages*, May 1999. 35

[Hub08] Bernhard Huber. *Resource Management in an Integrated Time-Triggered Architecture*. PhD thesis, Vienna University of Technology, Institute of Computer Engineering, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2008. 11, 19

[JT03] Axel Jantsch and Hannu Tenhunen. *Will Networks On Chip Close The Productivity Gap?* Springer US, 2003. 8

[KAGS05] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The time-triggered ethernet (tte) design. *8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), Seattle, Washington*, May. 2005. 5, 35

[Kop97] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997. 5, 8

[Kop06] Hermann Kopetz. Pulsed data streams. Technical report, Institute of Computer Engineering, Vienna University of Technology, Vienna, Austria, Febuary 2006. 6

[LG02] B. Luca and D. Giovanni. Networks on chips: A new paradigm for system on chip design. In *Proceedings of the DATE'02 Conference*, 2002. 8

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998. 90

[NMAM05] Vincent Nollet, Theodore Marescaux, Prabhat Avasare, and Jean-Yves Mignolet. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Proceedings of the DATE'05 Conference*, pages 234–239, Munich, 3 2005. 10

[OCP05] OCP International Partnership Association, Inc. *Open Core Protocol Specification 2.1*, 2005. 42

[RDP+05] Andrei Radulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees G. W. Goossens. An efficient on-chip ni offering guaranteed services, shared-memory abstraction, and flexible network configuration. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(1):4–17, 2005. 10

[Sal08] Christian El Salloum. *Resource Management in an Integrated Time-Triggered Architecture*. PhD thesis, Vienna University of Technology, Institute of Computer Engineering, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2008. 11

[Sei07] Roman Seiger. Design and implementation of an extendible interface subsystem in the time-triggered system-on-a-chip architecture. Master's thesis, Vienna University of Technology, 2007. 48, 64

[Sie06] Samuel Siewert. Soc prognostication. *SoC Drawer Series*, May 2006. 7

[SL95] Alexander Stepanov and Meng Lee. The standard template library. Technical report, HP Laboratories, October 1995. 68