Mode Changes in Network-on-Chip Based Multiprocessor Platforms

Ioannis Kotleas

Kongens Lyngby 2014
IMM-M.Sc.-2014

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk IMM-M.Sc.-2014

Abstract

As the operating frequency of computer systems has reached a standstill, modern computer architectures lean towards concurrency and Multi-Processor System-on-Chip (MPSoC) solutions to increase performance. These architectures use Networks-on-Chip (NoC) to provide sufficient bandwidth for the Message Passing Interface (MPI) among the integrated processors. T-CREST is a Network-on-Chip based general-purpose time-predictable multi-processor platform for hard real-time applications. The T-CREST NoC uses static bandwidth allocation, which is constant throughout the execution. In this thesis we extend the T-CREST MPI with a mode change module that enables the reallocation of the NoC bandwidth during run-time. For this purpose we use a dedicated broadcast network with tree topology and a mode change controller, which is driven by a master processor. The designed module respects the time-predictability asset of T-CREST and manages the mode changes transparently to the tasks execution, providing the flexibility to the programmer of the general-purpose T-CREST platform to define the policy under which the mode changes are performed. The mode change extended T-CREST platform is prototyped on an FPGA and it is shown that the resource overhead is very small.

Preface

This thesis was carried out at the Department of Applied Mathematics and Computer Science (DTU Compute) at the Technical University of Denmark, in fulfilment of the requirements for acquiring a MSc in Computer Science and Engineering.

Design of Digital Systems and Embedded Systems have been of particular interest to me during my studies at DTU. When I joined the Honors Programme of the university, it was natural for me to ask from professor Jens Sparsø to be my supervisor. During my MSc studies I had the chance to get involved to the T-CREST project, focusing on the Network-on-Chip aspect of it. This thesis came as an extension to an on-going collaboration with the T-CREST group.

Lyngby, 31-July-2014

[illegible]

Ioannis Kotleas

Acknowledgements

First of all I would like to thank my supervisor professor Jens Sparsø for all of the motivating and interesting discussions we have had the last one year and a half, and for the opportunities to get involved to an on-going research project, to attend an international IEEE conference and to do special courses in co-operation with a company developing integrated circuits for the high-performance electronics market. Secondly, I want to thank the co-ordinator of the DTU MSc Computer Science and Engineering, associate professor Jørgen Villadsen for inviting me to the Honors Programme of the university. Special thanks to the T-CREST group for their help and valuable contribution. Finally, I want to thank deeply my parents, Efthimios and Dimitra, for their love and support.

Contents

Chapter 1

Introduction

This chapter starts with an introduction of Multi-Processor-Systems-on-Chip (MPSoC) and a definition of some important parameters on Hard Real-Time Systems. Then follows a general approach to Networks-on-Chip (NoC) implementing the Message Passing Interface (MPI) of MPSoCs. Subsequently, T-CREST, a time-predictable MPSoC is briefly presented and, finally, the purpose and layout of this thesis is stated.

1.1 Multi-Processor Systems-on-Chip

During the last decades a very rapid growth has been observed in the fields of computer science, semiconductor industry and integrated circuits design. As the number of transistors fitting in a single chip increased according to Moore's Law [1] and the MOSFETs scaled according to Dennard's scaling [2], the computational performance offered by the computer systems increased with a constant and very fast ratio. This has been the case until approximately 2005, when the technological advancement reached the physical limit assumed by Dennard's scaling. For chip manufacturing processes of 90nm and below the power and heat dissipation with respect to the operational frequency increase dramatically [3]. However, the amount of transistors in a single die still increases. Therefore, modern computer architectures utilize multiple cores on the same chip, managing this way to continue increasing the computational performance of the systems.

In order for the applications to take advantage of this new trend, concurrency has to be taken into account at the programming level [4] since the sequential portion of a program can limit significantly the possible speed-up of a parallel application (Amdahl's Law) [5].

1.2 Real-Time systems

Real-time systems are a special class of computer systems where the Worst Case Execution Time (WCET) of a task has to be guaranteed, otherwise catastrophic consequences may occur. For this reason, time-predictability is an important factor in such systems. The WCET analysis is the method that analyses the execution of a task taking into account both hardware and software implementations, to calculate guaranteed upper bounds on the execution time of the tasks. In order for the WCET analysis to be feasible, all of the underlying components of an architecture have to be time-predictable. During the last years researchers have been focusing on hard real-time Multi-Processor-Systems-on-Chip. In such systems the WCET analysis is a very complex task.

1.3 Networks-on-Chip

The interconnection fabric of the Message Passing Interface (MPI) between the cores of a MPSoC is very important since it can be a bottleneck to the data transactions between the processors, limiting the system performance. As MPSoCs grow and they incorporate many cores, it becomes apparent that a system bus cannot provide enough bandwidth for the MPI. Instead, modern architectures orient towards Network-on-Chip (NoC) solutions [6, 7, 8, 9].

A NoC generally consists of network adapters (the interface between the processors and the NoC), routers and links, as illustrated in Figure 1.1. The links therefore are a common multiplexed resource among the channels connecting the Intellectual Properties (IP) that are attached to the NoC.

In [10] the basic characteristics for a NoC are presented and several NoCs are classified according to these characteristics. Depending on the purpose of the platform, different architectures have been proposed. For instance, lets assume a

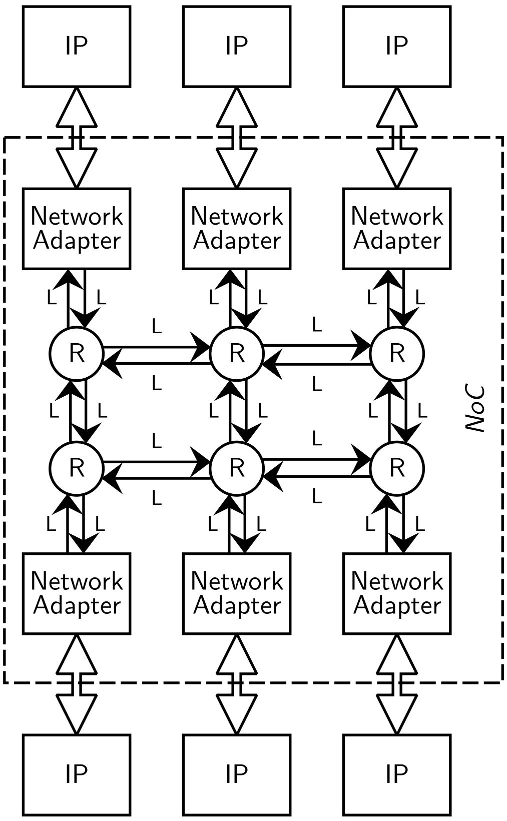

Figure 1.1: Overview of a NoC based multi-processor platform. 'IP' stands for intellectual property, 'NA' for network adapter, 'R' for routers and 'L' for links).

general-purpose application-independent platform, which must be flexible, scalable, and support a high level of parallelism. For such a system, with high packet injection rates and small packet size, a packet-switched NoC architecture, where the links are multiplexed on a packet transaction level, would deliver better results compared to a circuit-switched one [11]. In a circuit-switched NoC the path of a channel has to be first set up. Then the links of the path are used exclusively to transfer the data. When the transfer is finished, the path is torn down and the links are released to be used by other paths. On the other hand, a circuit switched NoC would suit better an application specific platform, where the requirements are precise and the NoC can be tailored to these requirements, avoiding unnecessary overhead. Examples of packet switched NoCs are the QNoC [12], XPIPES [13], SoCIN [14], SPIN [15], Tiny NoC [16], Kavaldjiev NoC [17] and Argo NoC [18]. Examples of circuit switched NoCs are the SoCBus [19], PNoC [20] and Wolkotte NoC [21].

Another important parameter is the timing organization of the system. As Systems-on-Chip grow larger, and the IPs on the chip get diverse, different clock domains must be supported. *Globally Asynchronous Locally Synchronous* (GALS) [22] system organization suggests that the IPs are locally synchronous, but possibly at different clock domains. The interface between the NoC and the IPs therefore must be a well standardised interface, supporting clock domain crossing, such as the Open Core Protocol (OCP) [23] which is used by XPIPES [13], Æthereal [24] and Argo NoC [18].

Furthermore, on large SoCs the clock distribution might be challenging. The NoC on the chip might cover distant locations on the die, and the introduced skew can reduce the operational frequency of a synchronous NoC, limiting the bandwidth and deteriorating the performance. Alternatively, mesochronous and asynchronous NoC implementations (MANGO [25], Beigne NoC [26], aelite [27] and Argo NoC [18]) can deal with this challenge, and they are a better fit to GALS architectures.

Real-time platforms, in which there must be a static and optimised upper bound to the latency of transferring a block of data, NoCs with Guaranteed Services (GS) have to be used. Time Division Multiplexing (TDM) is a common approach to avoid link contention, deadlocks and collisions. TDM is used by Nostrum [28], Æthereal [24], aelite [27], dAelite [29] and Argo NoC [18]. Best Effort (BE) NoCs focus on optimising the average case performance. Flow control, buffering of packets and arbitration may be utilized, resulting usually in larger hardware.

Finally, depending on the purpose of the interconnection of the IPs, the topology of a NoC can take different forms as grid, torus, cube, H-tree, butterfly etc. For example, Tiny NoC [16] focuses on a 3D mesh topology, while SPIN [15] is based on a fat-tree topology.

1.4 T-CREST - A time-predictable MPSoC

T-CREST is a project funded by the *Seventh Framework Programme for Research and Technological Development*[1] targeting to develop a general-purpose platform incorporating a multi-processor time-predictable system that will simplify the safety argument with respect to maximum execution time. As a multi-

[1]The *Seventh Framework Programme (FP7)* homepage can be found at http://cordis.europa.eu/fp7/home_en.html

processor system, it provides an MPI between the cores. This interface is implemented with a TDM based, packet switched, guaranteed services, asynchronous bi-torus NoC. This time-predictable NoC avoids traffic interference and provides virtual end-to-end connections. The TDM is governed by a static schedule [30] defining channels connecting the processors through the switched structure of the NoC [18, 31, 32]. The schedule and the corresponding bandwidth of the provided communication channels are generated once, when building the platform.

1.5 Mode changes of T-CREST NoC

In a real application, most of the time the running tasks will not use the assigned bandwidth of the static schedule. From a real-time point of view, over-assigning resources is a common practice as long as the guarantees are being met. Still, if the bandwidth could be re-distributed among the communication channels according to the currently running tasks' requirements, then the WCET of the tasks would be reduced, improving the performance of the system. The need for a schedule change may be driven by actual bandwidth requirements from the tasks running on the processors (starting or finishing), safety reasons like IP group isolation, or by external events (the pushing of a button, a sensor input, etc).

In this thesis we add the possibility to change the schedule of the TDM-based NoC of the time-predictable T-CREST platform, reassigning the network's bandwidth during run-time. For this purpose, we analyse the steps of a mode change, we compare these steps against the available options, we design new hardware components, we fully integrate the additional functionality both on hardware and software API level to the T-CREST platform and we calculate the latency of performing a mode change. Lastly, we verify the design and prototype it on the Xilinx ML605 FPGA board.

1.6 Thesis Layout

We have given so far a description about the environment of the thesis, together with some fundamental definitions and characteristics. The targeted platform has been introduced and the motivation and contribution has been stated. The upcoming chapters are as follows:

Chapter 2 explains the targeted platform and the aspects of it that affect the thesis project.

Chapter 3 presents mode change implementations of other NoC-based MP-SoCs and compares their applicability against the T-CREST approach.

Chapter 4 performs an analysis of the phases of a mode change. We explore the available options for every phase and make decisions regarding the specifications over a mode change. An architectural overview of the suggested extended platform is given and the phases of the mode change are allocated to architectural components.

Chapter 5 elaborates on the design of the additional hardware, together with the modifications to the existing hardware of the platform, with respect to the specifications stated earlier.

Chapter 6 presents the implementation and integration of the change mode module to the T-CREST platform and tool chain, both on hardware and software level, and the interaction of these levels is discussed.

Chapter 7 provides performance results by calculating the contribution of every mode change phase to the latency of performing a schedule change and estimates the worst case mode change latency.

Chapter 8 describes the test cases used to prove functionality and presents results regarding the correctness, the latency and the mode change module additional resources on an FPGA prototype.

Chapter 9 discusses general aspects regarding the usage of the mode change module, variations and extensions to its functionality.

Chapter 10 summarizes the thesis project, lists the contributions and suggests future works.

Finally, the Bibliography and the Appendices with the VHDL mode change module descriptions, the C software library API and the C test cases used for the verification are listed.

Chapter 2

T-CREST background

T-CREST is an open source project[1] targeting a general-purpose multi-core time-predictable platform for embedded hard real-time applications, specially designed to simplify the WCET analysis. For this purpose, all of the components of the system are independently time-predictable. The IP of the T-CREST platform is the statically scheduled, dual-issue RISC Patmos processor, which is described in [33]. In the patmos handbook [34] it is stated that for the connection of Patmos to a memory controller, I/O devices, the core-to-core NoC, and/or the memory arbiter a subset of the OCP1 [23] interface standard is used.

The MPI of the platform is implemented with the TDM-based Argo NoC [18, 31, 32] which can support bi-torus, mesh or custom topologies. Additionally to the local memories, the system provides access also to shared memory, which can be implemented both as On-Chip or Off-chip memory. In this thesis we will assume the On-Chip shared memory implementation, the access to which is managed through an arbiter. The overview of the system is given from two different perspectives, the MPI and the shared memory access perspective.

[1]The project's sources are hosted on GitHub and are available on-line at https://github.com/t-crest/

2.1 Shared memory access

As the shared memory is a common resource to all the IPs, arbitration is required to regulate the access to it. The arbiter of T-CREST utilizes the OCPburst protocol, which is described in the Patmos handbook [34], for the communication with the IPs. Details regarding the OCPburst are not relevant to the purpose of the thesis and therefore they are not being reported. The general overview of the access to the shared memory is depicted in Figure 2.1.

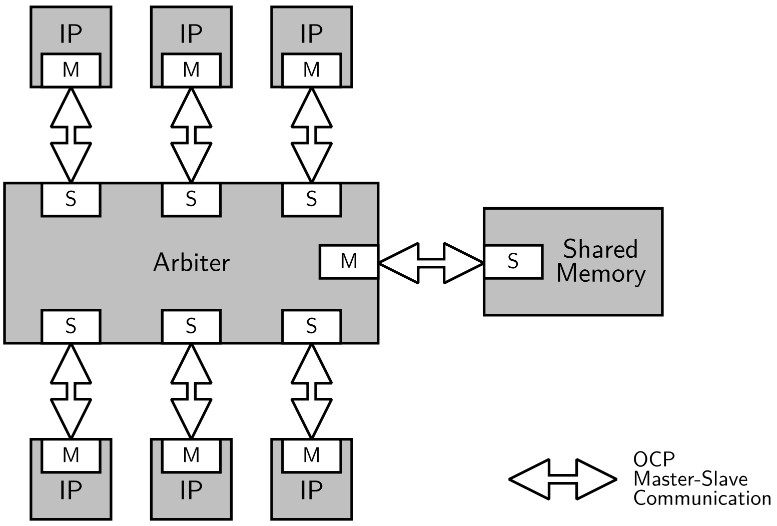

Figure 2.1: Shared memory access with Master-Slave OCPburst communications.

2.2 Message Passing Interface

In Figure 2.2 a mesh topology is illustrated focusing on a 3-by-3 area of the mesh. Each tile in the grid is a processor, together with a network adapter providing access to the NoC and a true dual port scratch pad memory, which is used as a buffer for the incoming and outgoing data transactions of the MPI.

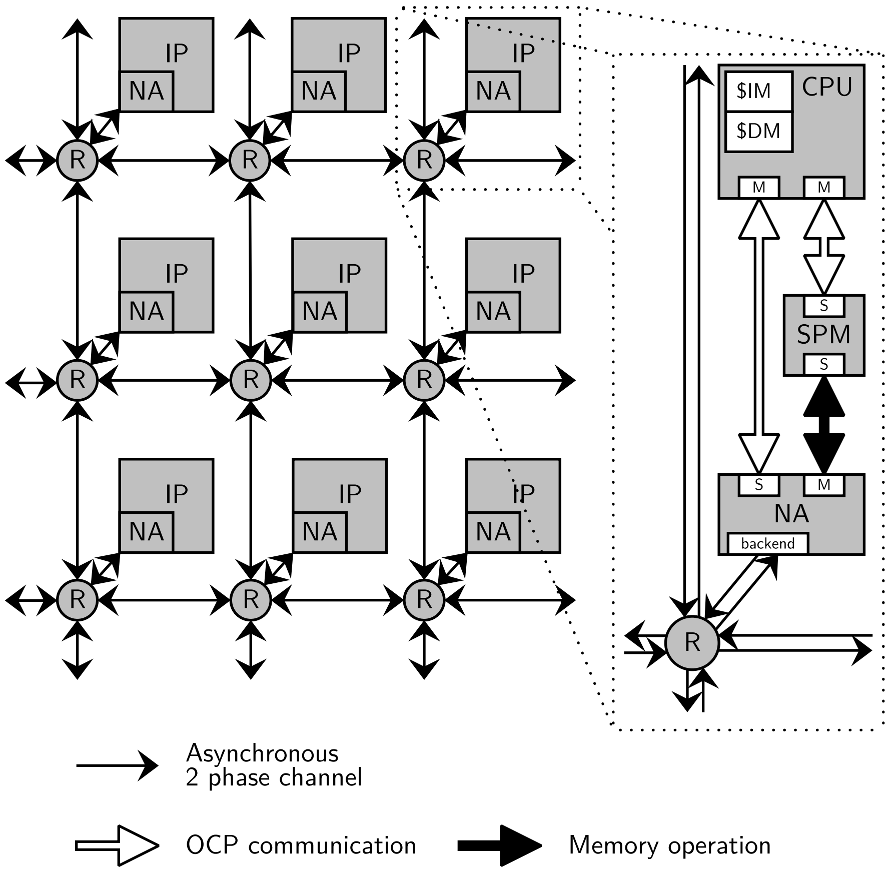

Figure 2.2: Conceptual overview of a T-CREST platform with mesh topology. 'IP' stands for intellectual property, 'NA' for network adapter, R for routers, 'L' for links, 'SPM' for scratch pad memory, 'IM' and 'DM' for instructions and data caches.

2.2.1 Timing organization and TDM

The Argo NoC of T-CREST is packet-switched and TDM-based. The time is divided into periods, and each period into slots. Driven by a static global schedule, the data to be sent are split into packets, and according to the current slot, they are injected to the NoC. Due to the TDM static scheduling, the NoC is contention free, meaning that there is no possibility of packets utilizing the same link at the same time resulting into collisions. As a result, the routers are implemented as simple pipelined crossbar switches. This requires that the network adapters have a common notion of the current slot.

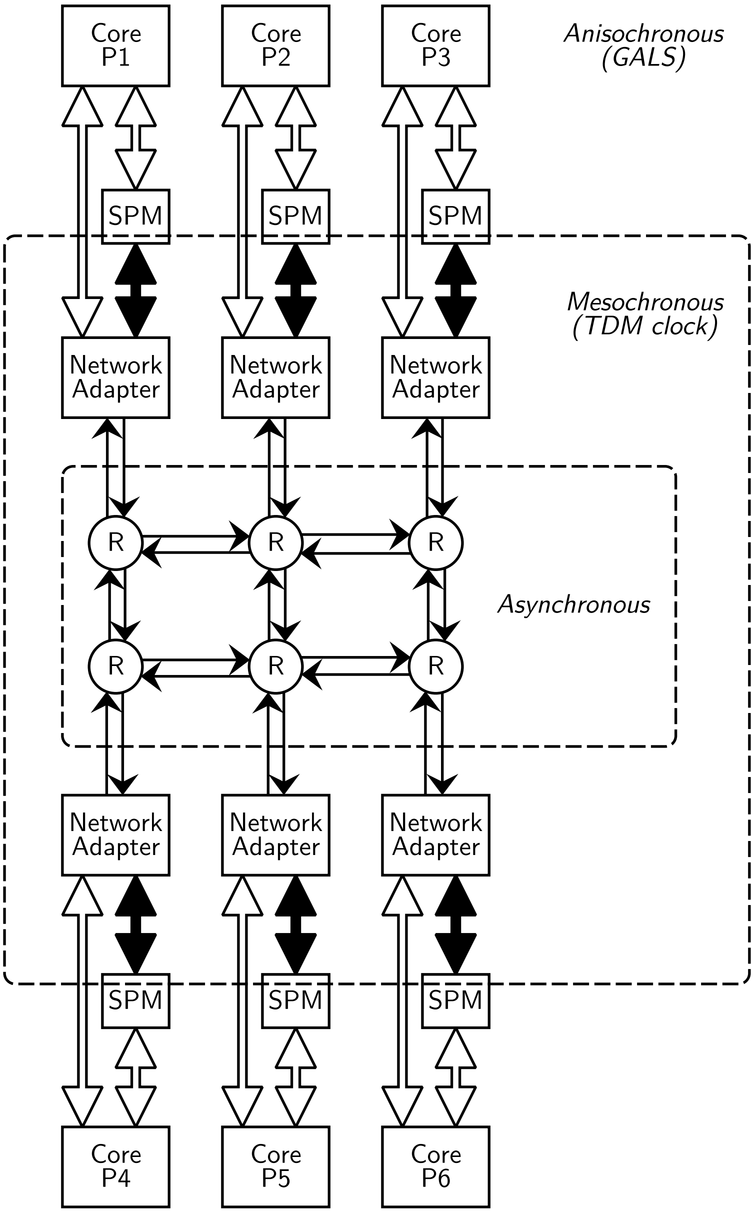

Figure 2.3: GALS system organization of Argo NoC

The Argo NoC of T-CREST provides a GALS approach to timing organization, as shown in Figure 2.3. The processors are synchronous, but they can operate at different clock domains to one another. The network adapters of the Argo NoC though are mesochronous, which means that they have the same clock, but a certain amount of skew can be tolerated among them. A slot-counter inside every adapter keeps track of the current slot in parallel to the others. The self-timed switching structure of asynchronous routers moves the data tokens utilizing the 2-phase bundled data handshaking protocol [35], absorbing at the same time the skew of the mesochronous domain.

2.2.2 Interfacing the NoC

In order for a processor to send a block of data through the NoC, a series of steps have to be taken.

1. At first, the processor has to move the block of data to send to the local SPM lying between the processor and the network adapter. The processor port to the SPM, as seen in Figure 2.2, abides with the OCPcore protocol described in the Patmos handbook [34]. The sequence of signals between the master (in this case the processor) and the slave (the SPM), in order to perform an OCPcore write or read operation is shown in Figure 2.4.

2. Then, the processor has to inform the network adapter that a block of data in the SPM has to be sent to some recipient. The communication between the processor and the network adapter is handled through the OCPio protocol described in the handbook. Moreover, a local space of addresses is defined to distinguish among the configuration operations from the processor (master) to the adapter (slave). The sequence of signals is illustrated in Figure 2.5.

2.2.3 Network adapter

The network adapter is the module that implements the static TDM schedule, and moves data through the switched router structure. It has three fundamental components, the slot counter, the slot table and the DMA table, as shown in Figure 2.6. As explained in Subsection 2.2.1, the slot counter is driven by the mesochronous TDM clock. Its value is used to index the slot table, which contains information about the current slot. The counter counts up to the last entry of the static schedule and then it is reset in order to start a new

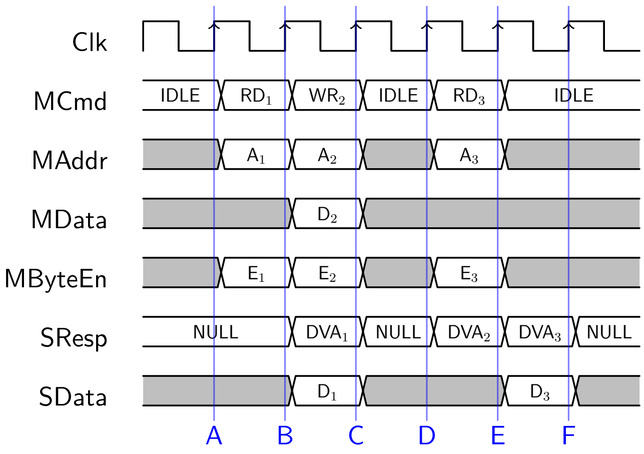

Figure 2.4: Patmos OCPcore timing diagram.

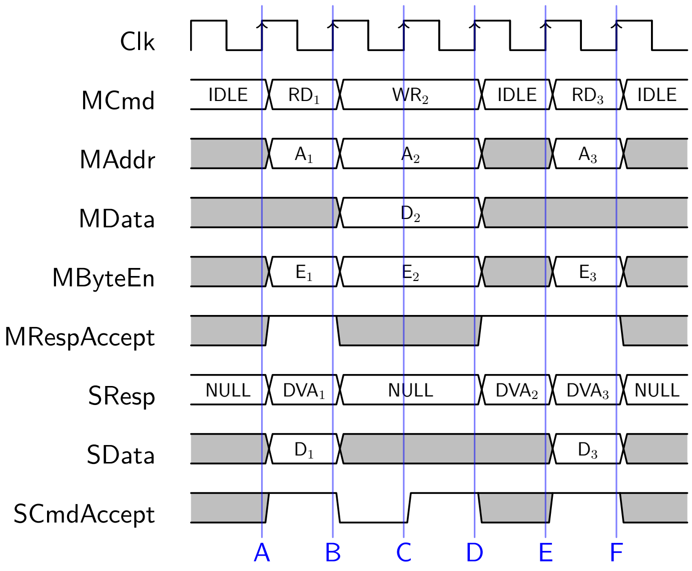

Figure 2.5: Patmos OCPio timing diagram.

schedule period. Every entry in the slot table indicates if the current slot is valid, enabling a packet sending. In such case, the entry contains also a pointer to another entry, this time in the DMA table.

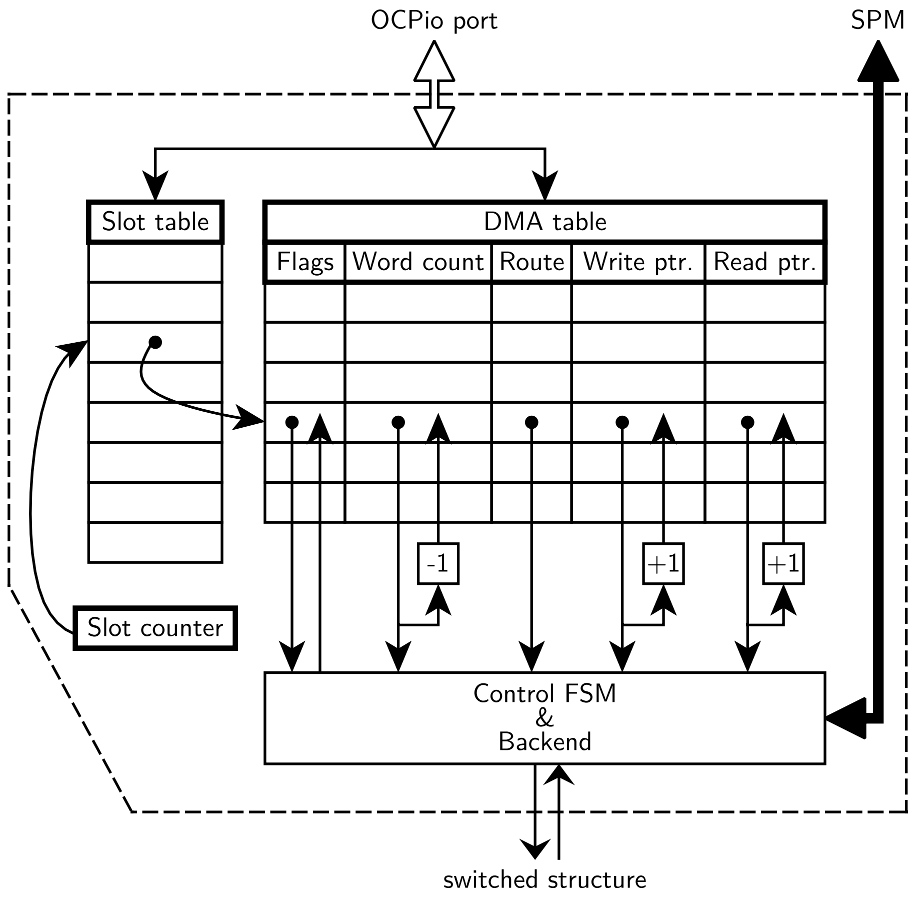

Figure 2.6: The network adapter.

Every entry of the DMA table handles the transfer of a block of data. There is one entry for every possible recipient of data through the MPI. For instance, in an N-by-N mesh, with N^2 IPs, the network adapter associated to an IP has $N^2 - 1$ entries in the DMA table, if communication to all is considered. The information stored in a DMA entry is:

- The address in the local SPM from where the next word of the block being transferred should be read

- The address in the remote SPM of the recipient where the next word to send must be written to
- The number of remaining words until the completion of transferring the block
- The routing info defining the path of routers that the packet must go through until reaching the recipient's network adapter
- Some control flags

During the system's boot phase, every processor has to perform the initial configuration of the local network adapter. Each processor copies from the local data cache memory the corresponding slot table and the routing info for every DMA entry, utilizing the OCPio port to the network adapter and the local address space. Afterwards the processors are synchronized and the execution of the tasks proceeds. This operation is done only once, during the boot phase. From that point on, all the communication between the processors and the network adapters regards setting up block transfers by reading and writing the DMA table entries.

Once a DMA block transfer has been set up, whenever there is a slot activating that DMA entry, the DMA reads a word from the SPM from the location indicated by the read pointer, a packet is built and injected to the NoC, the local read pointer, the remote write pointer and the number of remaining words in the block are updated, and if the block transfer is finished, the control flags are set accordingly.

From the reception point of view, when a packet arrives, it contains the write address and the data to be written to that address. Therefore, the network adapter simply extracts the address and the data and performs a write operation to the SPM.

The interface between the network adapter and the SPM is a synchronous read synchronous write memory interface. An internal state machine in the network adapter regulates when to perform a read and when a write operation to the SPM.

At this point it has to be mentioned that the network adapter does not support any way of signalling the completion of reception of a data block. This is resolved in software level, by extending the block to be sent by one more word, which is used as a flag to be polled by the receiver IP, indicating the completion of the block.

2.2.4 Routing and packet format

As it has been implied so far, the Argo NoC is a source routed network. The information about the path a packet is routed through is contained inside the packet itself. The packets used in Argo are fixed-size and they consist of 3 flits of 35 bits each. The first 3 bits of every flit are a prefix used to specify the type of the flit. The first flit is the header, containing the routing info and the write address to the recipient's local SPM. The remaining two flits are the payload, and combined they form a 64bit word to be written to the address indicated by the header. The structure of a packet is shown in Figure 2.7.

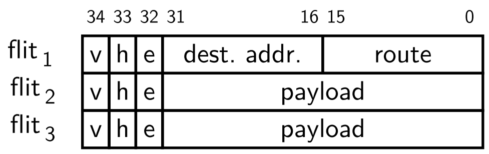

Figure 2.7: Packet format of Argo NoC.

Since the packet consists of 3 flits to be sent separately, but still as a group, the network adapter clock is three times faster than the TDM clock. For every three clock cycles the TDM advances by 1 slot. This gives enough time to the network adapter to perform all the necessary read and write operations to the SPM to accommodate for the sending and receiving of a packet during a TDM slot. These read/write operations are managed by the control state machine of the network adapter.

The path that a packet goes through consists of a series of routers and links, until the destination network adapter is reached. Both the routers and the links are pipelined to improve the network throughput.

2.2.5 Scheduling

The scheduler of T-CREST is the one described in [30]. It is a meta-heuristic scheduler which operates on task graphs of parallel applications, like the one shown in Figure 2.8. It receives as inputs the allocation of the tasks to the platform processors, the topology, the desired bandwidth for every connection and the depth of the routers and the links in terms of pipeline stages in order to generate the static global TDM schedule. During this process, meta-heuristic methods are applied to compress the schedule size.

The first note to make at this point is that the packet movement in the switched

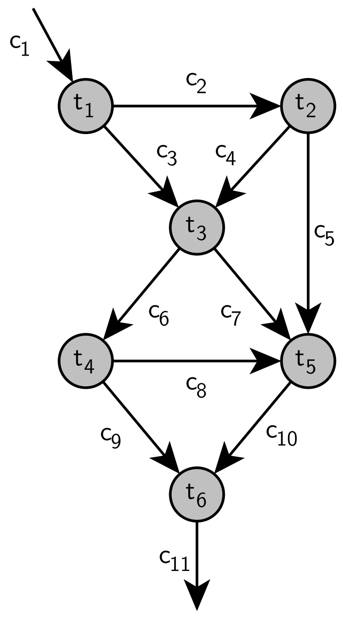

Figure 2.8: Task graph example of parallel application.

router structure is misaligned. This means that when a network adapter injects the first flit of a packet to the local router, the second or the third flit of other packets that are on their way to some destination may be entering the router as well (off course from different ports and claiming different outputs). This happens because there is no correlation between the pipeline depth of the routers and the links to the packet size. The scheduler of T-CREST takes into account this misalignment. Additionally, the scheduler, together with corresponding functionality in the network adapter, can postpone a slot by one or two clock cycles, in order to increase the flexibility and maximize the compression capability. Empty slots, where a network adapter should not send data, are also used. The information about the postponing of a slot is stored inside the slot table of the network adapter, together with the validity flag of the slot and the pointer to a DMA entry, as described in Subsection 2.2.3.

One last, but very important thing that has to be mentioned about the T-CREST static global schedule, is that it drains all the packets from the NoC at the end of every schedule period. For example, in Figure 2.9 when the period *i* starts, there are no flits of packets in the NoC, and none of the links is used to propagate data. The blue lines in this figure separate the schedule periods, while the red lines indicate the slots of the schedule period *i*. At the beginning

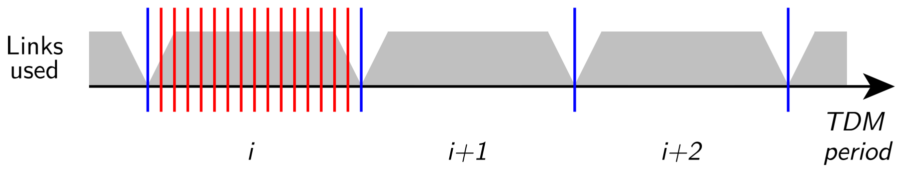

Figure 2.9: The filling and draining effect of the static schedule.

of a period, the network adapters start injecting packets from the local port of the corresponding router. As these packets proceed in the NoC and more packets are injected by the network adapters, the other links implementing the interconnection of the routers are also utilized. This utilization reaches a maximum, which is actually related to the bandwidth provided by the NoC. At the end of the schedule period, the network adapters stop injecting packets. The flits that are already in the NoC gradually arrive at their destinations and the NoC is completely drained before a new schedule period starts. This way the NoC returns to the same empty state at the end of every period.

Chapter 3

Related work

In [21] Wolkotte described a circuit switched NoC for a heterogeneous multi-tile System-on-Chip targeting multimedia applications where data streams are semi-static and with periodic behaviour. This NoC supports both Guaranteed Services (GS) and Best Effort (BE) traffic. The guaranteed services are provided by scheduling the communication streams over non-time multiplexed channels. The Best Effort traffic is handled by a separate packet switched structure and it is used also to set-up connections between the processing elements. The NoC is configured by a special node, the Central Coordination Node (CCN), which allocates tasks to the processing elements and manages the NoC. The CCN sends control packets over the BE network to the GS circuit switching routers. The reconfiguration process is not elaborated though in [21] and the usage of BE traffic does not provide guarantees on the connection set-up time.

Kavaldjiev in [17] introduced a packet switched NoC which targets mobile multimedia devices where traffic is dominated by streams. This NoC provides both GS and BE by using Virtual Channels (VC) to share the links. In every router several VCs share a physical link on a cycle-by-cycle round robin basis. Thus, all VCs equally share the bandwidth of the physical channel. A chain of VCs forms a connection. Buffers are used at the input and output of the router. For GS, the VCs are dedicated to a connection, while BE is implemented with shared VCs. The packets are source routed, which means that each packet has a header containing among others the route of VCs that it has to travel through.

The mode change in this NoC is done with special BE packets that reserve the VCs they traverse. NoC reconfiguration is managed by a central authority, the Configuration Manager, and it is done only at the beginning of an application. The mode change procedure is not explained further in [17]. Similarly to [21], the usage of BE traffic to reconfigure the NoC does not abide with the T-CREST time-predictability orientation.

SoCBus [19] is a circuit switched NoC providing dedicated end-to-end connections that can operate in the mesochronous domain. The connections are set up with special BE packets that reserve or release links as they traverse them. Arbitration is used on connection establishment, and the links are not shared between connections, leading to excessive blocking. Some discussion is given on the usage of static scheduling of communications to minimize the blocking, but no further information is provided.

PNoC [20] is also a circuit switched NoC with dedicated end-to-end connections. The connections are set-up by the nodes with a lightweight mechanism according to the authors, which involves arbitration and link request queues. No information is given on how the configuration is managed.

Æthereal [24] describes three NoC designs. All three use contention-free routing and TDM, much like the T-CREST NoC. The first NoC uses distributed routing, where the routing info is stored in the routers and the packets do not have headers. The router is also enhanced with BE logic to maximize the utilization of the links with BE traffic when there is no GS traffic. The BE part is packet switched, source routed with wormhole switching, and the packets have headers. In order to establish a connection, a special BE packet is sent by the source to the destination, which reserves time slots as it traverses through the routers. The establishment of a connection is acknowledged by the destination. Like before, no guarantees can be given on the set-up time due to the BE traffic. Moreover, the reconfiguration is not transparent to the tasks executed on the IPs, since the connections are set-up with packets from the source to the destination. The second NoC described drops the distributed routing for the GS and uses source routed packets with headers instead, with higher priority over the BE packets. Since no routing info exists in the routers, the mode change involves reconfiguration of the network adapters. To set up a connection, a mode change root process sends either special BE packets or BE packets to reserved memory addresses to the source of the connection and to the destination of the connection. The third NoC described is a NoC only with GS traffic. In this case, the mode changes are done with GS packets from the root process to the source and destination of every connection. This means though that some slots (bandwidth) have to be reserved from the root process to all of the nodes. Considering that a mode change is an infrequent and bursty operation, the reserved bandwidth is wasted and the connection set-up has big latency.

The last NoC examined, whose connection set-up mechanism is actually closer to our design, is the dAelite [29]. Like the first of the three Æthereal NoCs [24] described, it uses contention free TDM and distributed routing. However, this NoC supports only GS traffic and it focuses on multi-casting. A centralized configuration mechanism is introduced to set-up and tear-down connections with the usage of a dedicated broadcast tree network. A host IP co-operates with a configuration module, which drives the broadcast tree, managing this way the connections. ID tags and a complex protocol are used to configure the network adapters and the routers. The configuration process is transparent to the applications executed on the IPs of the NoC.

Even though the dAelite connection set-up mechanism has many qualities that are common to our aspirations, there are some fundamental differentiations to our approach. The T-CREST NoC is source routed without flow control. The set-up of a connection is a simple slot configuration at the source. For this reason, there is no need for a complex protocol. Furthermore, in dAelite the schedule size is static. On the contrary, the T-CREST scheduler [30] generates schedules of different sizes. Moreover, in dAelite the configuration is done one connection at a time. For a different allocation of the bandwidth than the initial configuration, many connections may need to be first torn down before setting up new ones. In this project we aim at a more flexible approach, where all old connections are torn down and the new ones are set up at the same time in one instant. Finally, dAelite targets a synchronous platform, whilst the Argo NoC [18] uses an asynchronous switched structure with mesochronous network adapters that can tolerate skew almost up to three clock cycles, depending on the operating frequency of the mesochronous clock.

Chapter 4

Requirements and suggested architecture

In this chapter, at first we define the mode change and we introduce some initial requirements to the mode change module. Then, the mode change is split into phases and an analysis of the available options is performed. Finally, the exact requirements for the mode change module are specified.

4.1 Mode change definition

In Section 1.5 a mode change was regarded as the reassignment of the network's bandwidth during run-time. This reassignment is necessary due to new core-to-core bandwidth requirements. In Subsection 2.2.5 it was mentioned that the TDM schedule of T-CREST is generated according to the connection requirements of the tasks of a parallel application. During execution though some tasks may finish, other tasks may start or some connection requirements may be altered. Figure 4.1 illustrates such an example using task graphs. In this example, task t_3 finishes, other tasks start (tasks t_7, t_8, t_9 and t_{10}), a new connection between tasks t_1 and t_4 is established and the connection between tasks t_2 and t_5 has updated bandwidth requirements.

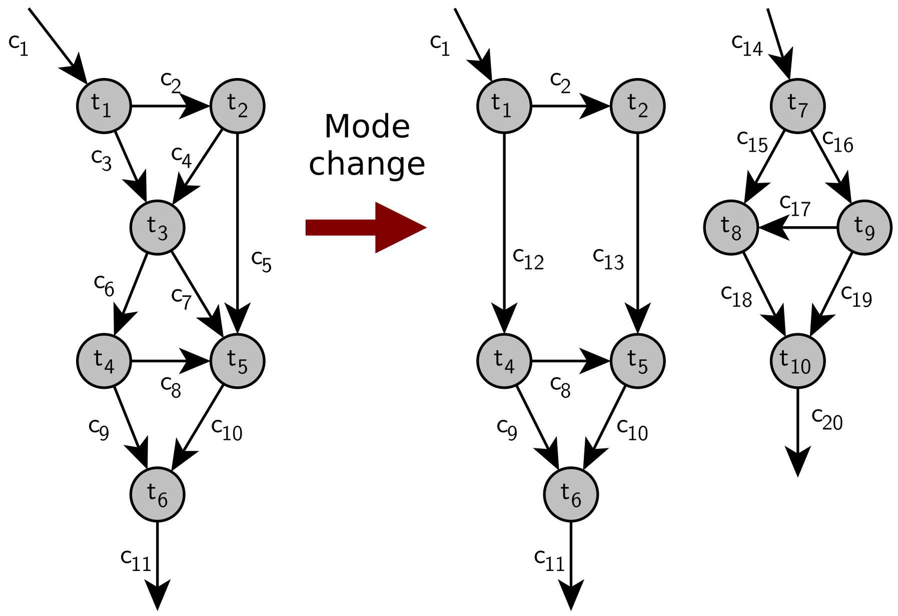

Figure 4.1: The mode change through task graph connection requirements.

4.2 Initial requirements

This class of specifications is imposed by the general T-CREST concept and the desired functionality.

Time-predictability The mode change module has to guarantee an upper bound to the execution time of applying a mode change.

Schedule flushing The packet switched Argo NoC and the T-CREST scheduler does not allow for partial reconfiguration of the schedule in general. Instead, a total schedule substitution is considered.

Transparency During a mode change, some tasks continue executing on the processors. For this reason, a mode change must be completely transparent to the applications.

Flexibility The mode change module must provide freedom to the user of the platform to configure the schedules and the application policy.

4.3 Mode change phases decomposition

To explore the requirements of a mode change, the operation is decomposed in four phases, each of which is examined independently. These phases are:

Phase 1 Accept request for a mode change

A request for a mode change is transferred to the mode change module.

Phase 2 Schedule acquisition

A schedule accommodating the pending request is accessed.

Phase 3 Fetch the new schedule

The network adapters are updated with the new schedule.

Phase 4 Apply schedule

The new schedule is applied by the network adapters.

In the following subsections we explore the available options for every mode change phase, and we indicate the chosen option with a red outline.

4.3.1 Requesting a mode change

What constitutes a request and how it is triggered is not part of this thesis. It can be run-time dependent, like the start or the finish of a task with specific bandwidth requirements or run-time independent, like an external event (push of a button, timer interrupt, etc). On the contrary, the notification of the mode change module of a pending request is important, and the matter can be approached in three ways:

a) Through the Argo NoC

In such case the mode change module must access the NoC as a regular node and some bandwidth has to be reserved to this node in the schedule. Considering that requests for a mode change are not expected to be a frequent event, this reserved bandwidth is being wasted.

b) Through a dedicated ”all-to-one” network

This network would have to be designed. As an all-to-one network, arbitration would be required in its implementation, increasing the complexity and the resource overhead.

c) Use a processor

Due to the flexibility specification stated in Section 4.2, the requests must be resolved on software level, so that the request service policy can be defined by the user of the platform. Therefore, a processor must co-operate with the mode change module. The shared memory, the I/O devices and the processor interrupts can be used this way at the will of the programmer to implement the mode change request service policy. Due to the transparency specification, this processor has to be dedicated to system level tasks and not application level tasks.

4.3.2 Schedule acquisition

Two approaches can be followed regarding the schedule acquisition. The first one is to generate the schedule when a request is given to provide for the requested bandwidth requirements. In such case all the possible combinations by the various tasks should be pre-calculated to guarantee that the requested bandwidth is feasible. At the moment the T-CREST scheduler is not designed to run as a service. For a statically scheduled TDM NoC, storing the applicable schedules instead of generating them during run-time appears to be a more suitable solution.

Regarding the location of storing the schedules, two options were considered:

a) A dedicated ROM

Since the schedules are constants and they cannot be changed, a dedicated ROM holding the schedules seems a suitable solution. Considering though the flexibility requirement, one can see that schedule handling from a software point of view would not be possible.

b) Processor accessible memory

Two options are available in this case, the shared memory and the local memory of the processors. In such case it is up to the software API and the programmer to configure the schedules storage.

4.3.3 Fetching the schedule

The first consideration regarding the fetching of a schedule is whether the control of the operation should be distributed or centralized. In a distributed control system, each processor would have to fetch a portion of the schedule to the local network adapter. This way, the transparency specification cannot be met, since the processors would have to pause their current task, regardless of the current state of execution. In a centrally controlled system, the dedicated processor for system level tasks can be in charge of the mode change.

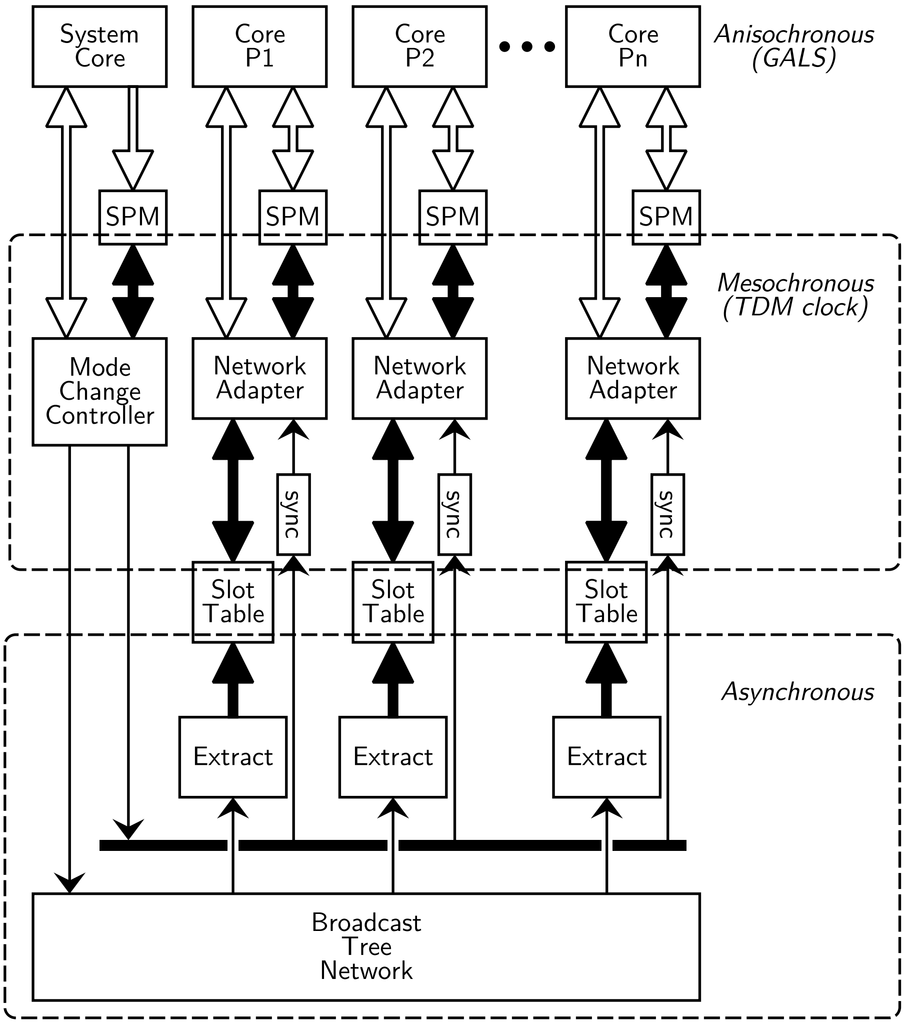

Figure 4.2: Mode change suggested architecture with timing organization.

In order to fetch the schedule, the following options were considered:

a) Use the NoC

   The dedicated processor can be attached to the NoC and use its links to transfer the new schedule to the network adapters. Special packets or address spaces would have to be used to instruct the network adapters to handle these packets for reconfiguration purposes. Some bandwidth from the control processor to all the other nodes would have to be reserved, affecting the performance of the NoC.

b) Use a dedicated broadcast tree network

   The schedule lies in a processor-accessible memory and can be read word by word. A dedicated broadcast tree that would send exactly the same information to all of the nodes, utilizing an ID tag to distinguish among them is the selected solution. Additional hardware has to be designed and used, but due to the simplicity of a broadcast tree, the resource overhead is expected to be very small.

4.3.4 Applying the new schedule

In Subsection 2.2.5 the schedule's property to return to the initial empty state at the end of every schedule period was introduced. This is the property that allows us to do a mode change transparently to the applications. The network adapters receive the new schedules and keep them inactive until a mode change is performed when the slot counter returns to 0. It is very important that all of the network adapters do the swap to the new schedule at the same (mesochronous) moment. For this reason, there must be a command triggering the swap, driven by the mode change module. Since the mode change event is a TDM-clock event, the mode change module must have a notion of the current slot.

The module can know when it is time to give the command to change mode with:

a) Flow control

   The flow control on the broadcast tree can signal the reception of the new schedule by all of the nodes.

b) Utilization of the static timing properties

> The schedule period size, the broadcast tree depth and the tolerable skew of the mesochronous clock domain are well known. Schedule period counters can be used to define a moment in the future that the mode change should take place.

4.4 Suggested architecture

For the mode change module we suggest the usage of a system task processor, which has access to the shared memory as the processors shown in Figure 2.1, but it is not attached to the MPI NoC. Instead, the system processor communicates with the mode change controller through the OCPio port of the processor.

As it can be seen in Figure 4.2, between the mode change controller and the system processor a simple dual port Scratch Pad Memory (SPM) is placed to be used as a buffer for scheduling data. The write only port is driven through the OCPcore port of the processor, while the read only port to the mode change controller is a synchronous read memory port.

The mode change controller keeps track of the current schedule period and slot, and when instructed by the processor, it manages the transfer of a schedule from the scratch pad memory to the network adapters. This transfer is done through the broadcast tree, at the leaves of which there is one schedule extractor for every network adapter.

The slot tables of the network adapters are converted to simple dual port memories. The write only port is associated to the extractor and the read only port to the network adapter. Moreover, the size of the slot table is duplicated and the table is split into two areas, to be used mutually exclusively and in an interleaved way by the extractor and the network adapter between mode changes.

From a timing organization point of view (Figure 4.2), the controller belongs to the mesochronous domain of the TDM-clock, since it needs to have the same notion of the current slot as the network adapters. To communicate data from one area of the mesochronous domain to another area of the same domain, the communication must go through the asynchronous domain, since the skew may exceed one clock cycle.

The transfer of the new schedule is done through the broadcast tree network and the extractors, which belong to the asynchronous domain. All of the communications between the controller, the broadcast tree nodes and the extractors are bundled data 2-phase asynchronous handshakes [35]. The interface from the extractors to the slot tables is a synchronous write memory interface, but the writing clock pulses are generated asynchronously.

Except for the transfer of a schedule, the controller has also to command the network adapters to apply the new schedule. Once again, as a mesochronous node to mesochronous node communication, the command has to go through the asynchronous domain. A 2-phase signal with bundled data is used to re-enter the mesochronous domain. To avoid metastability, a series of flip-flops, clocked with the clock of the network adapter is used at every adapter to synchronize the 2-phase command signal [36], as shown in Figure 4.3.

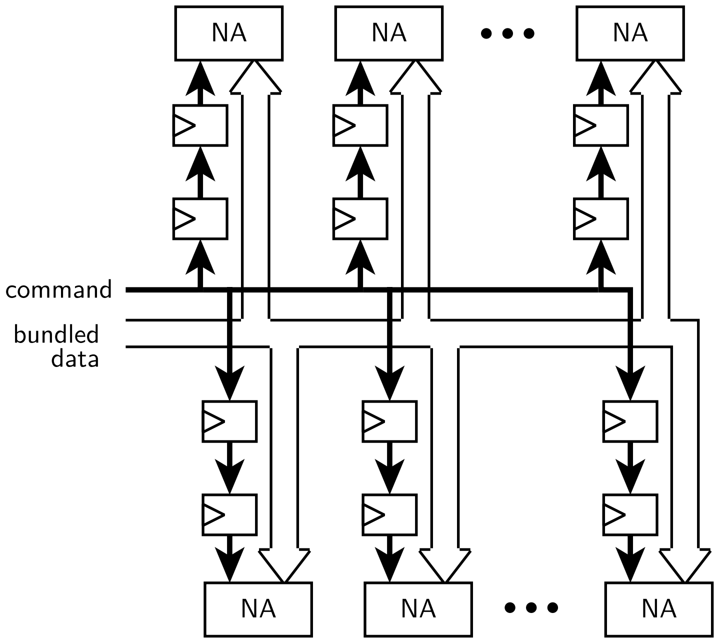

Figure 4.3: 2-phase command signal with bundled data synchronization with two flip-flops at every network adapter (NA).

4.5 Mode change phases and suggested architecture

To sum up, an allocation of the defined mode change phases in Section 4.3 is done with respect to the suggested architecture.

Phase 1 Request a mode change

A task executing on a processor writes to the shared memory setting a request flag. A routine on the system processor polls the shared memory for pending requests. Alternative, an interrupt to the system processor driven by an external event is issued. This interrupt may be associated to a mode change. The mode change request definition and handling is done on software level by the programmer.

Phase 2 Schedule acquisition

The schedules are stored either in the data cache of the system processor or in the shared memory. A decision is made on which schedule to apply as a response to the request and the selected schedule is copied to the SPM of the mode change module through the OCPcore port. The copying operation is managed through the software API of the module.

Phase 3 Fetch schedule

The system processor instructs the mode change controller through the OCPio using the software API to transfer the schedule, providing the location of the schedule in the SPM and the size of it. The mode change is therefore initiated (Figure 4.4). The controller reads and pushes the schedule in the broadcast tree, enhancing every word with an ID tag to distinguish among the recipients. The extractors receive all of the words sent, but according to the ID they extract the information related to their local network adapter. The received schedule is stored in the idle bank of the slot tables.

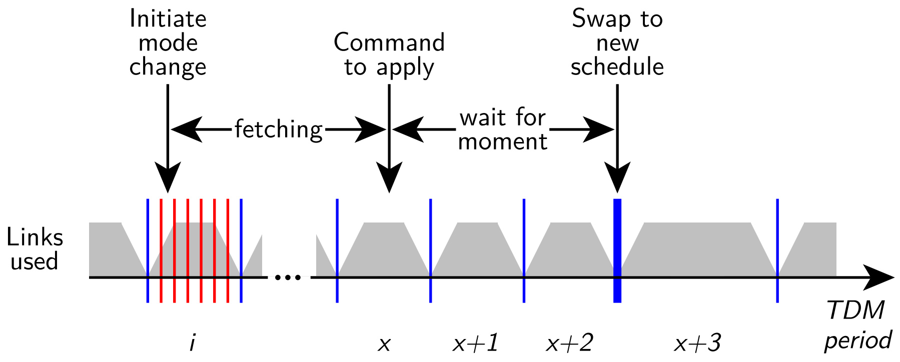

Figure 4.4: The mode change from the TDM-period time perspective.

Phase 4 Apply the schedule

Once the controller has finished pushing a schedule, it examines the current slot and period and defines a moment in the future (between two periods) that the swap must be done. The 2-phase command signal is toggled and the moment of swapping is passed as the bundled data of the command signal. The network adapters receive the command, and start comparing their local schedule period counter to the moment of swapping. When the time comes, the network adapters simply start reading from the updated bank of the slot table, taking also into account the size of the new schedule in order to wrap their slot counters to 0 accordingly. Figure 4.4 demonstrates an example where the controller, after the fetching phase, commands the network adapters to apply the new schedule at the end of the TDM-period *x+2*.

CHAPTER 5

Design

In this chapter the additional hardware that was designed and the modifications to the existing are elaborated. The new blocks are the mode change controller (Section 5.1), the broadcast tree network (Section 5.2) and the extractor (Section 5.3). The T-CREST network adapter extension is described in Section 5.4.

5.1 Mode change controller

The mode change controller is the block that manages the fetching and the application of a new schedule. The controller block, as depicted in Figure 5.1, is interfaced with an OCPio port to the system processor, a synchronous memory read-only port to the SPM, a 2-phase bundled data handshaking channel to the broadcast tree and a 2-phase bundled data channel to the network adapters. The connection of these ports is shown in Figure 4.2. The functionality of the controller is based on the mesochronous counters and three state machines managing the OCP communication to the system processor, the timing, and the schedule fetching and applying respectively.

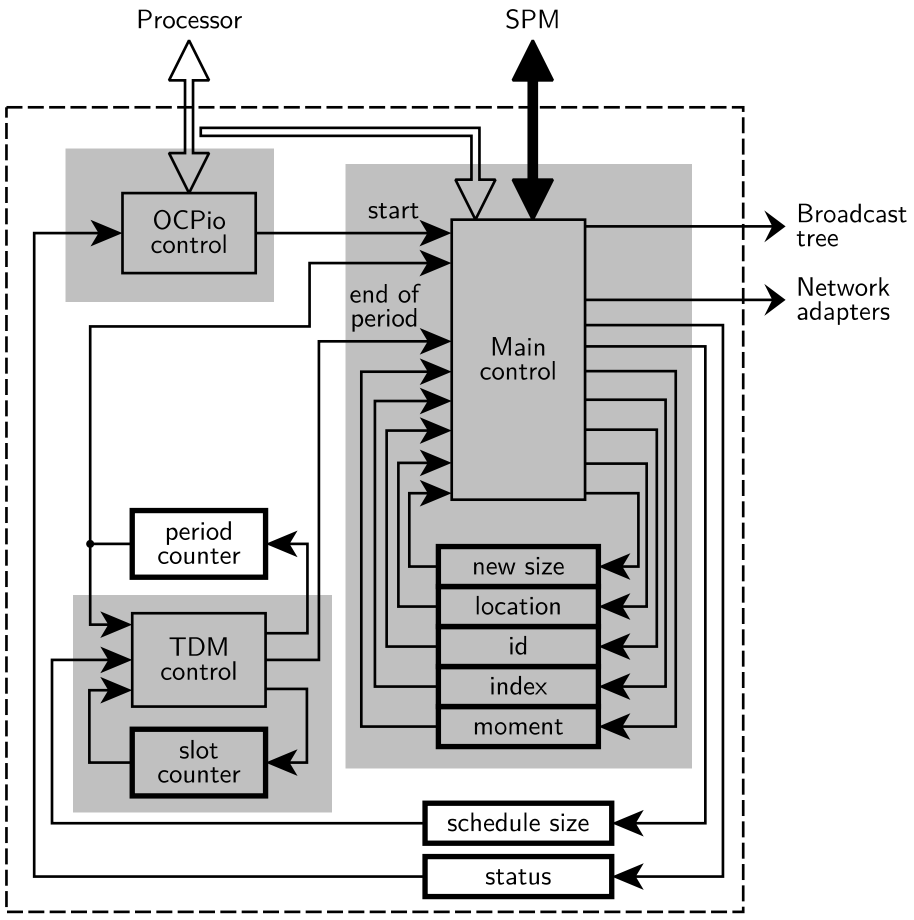

Figure 5.1: Mode change controller block diagram.

5.1.1 Processor interface FSM

The controller is mapped to the system processor address space with a single address. When the master (the system processor) reads from this address, then the status regarding the availability of the controller is given as a response through the OCPio port. On the other hand, when a write operation is done to this address, then it is perceived as an instruction to perform a mode change. The state machine handling these OCPio transactions is a Mealy machine, the ASM chart of which is shown in Figure 5.2.

The FSM has two states. When in the *idle* state, it is waiting for an OCPio

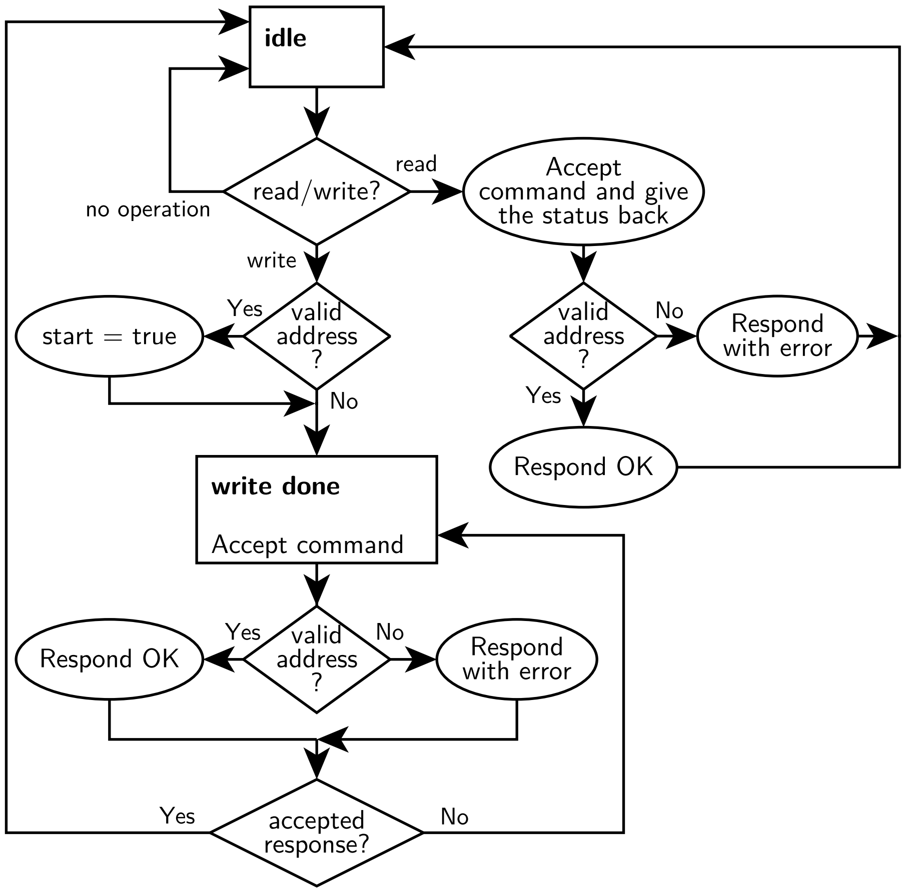

Figure 5.2: The controller's OCPio handling state machine.

command from the processor. When an OCPio command is set, if the command is a *read*, then it is handled within the same state, and the status of the controller (busy or free) is returned to the processor. In the case of a *write* OCPio command, a mode change is initiated by setting the signal *start* and the machine transits to the *write done* state, which handles the completion of the OCPio write transaction.

As it can be seen in Figure 5.1, the *start* signal triggers the main FSM. The size of the new schedule and its location in the SPM are the data of the write operation, and they are used by the main FSM.

5.1.2 TDM counters and timing FSM

The controller incorporates a slot counter that runs mesochronously and in parallel to the slot counters of the network adapters. The slot counter increments up to the size of the current schedule and then it is set back to 0 to start a new period.

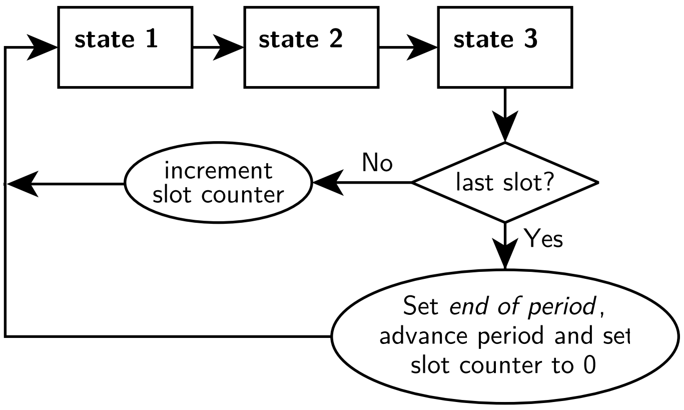
state 1
state 2
state 3
last slot?
No
increment slot counter
Yes
Set *end of period*, advance period and set slot counter to 0

Figure 5.3: Controller slot and period counter ASM chart.

Additionally, the controller has a schedule period counter to keep track of the current period. This counter is one-hot encoded and it is rotated every time the slot counter reaches the maximum value.

As explained in Subsection 2.2.4, every slot has a duration of three clock cycles of the mesochronous clock. A very simple state machine keeps track of timing for the mode change controller. It has three states and it spends only one cycle at every state. A full transition from all states is equivalent to a slot. The first two states are empty and no operation is performed. At the last state, the slot counter is enabled and if it is the last slot of the current schedule, then it is set back to 0 and the schedule period counter advances to the next period. Additionally, the *end of period* signal, which is an input to the main FSM (Figure 5.1), is set. Otherwise, the slot counter increments to the next slot. These transitions are shown in Figure 5.3. The last slot flag is the result of the comparison of the current value of the slot counter to the register holding the size of the current schedule (Figure 5.1).

5.1.3 Schedule format

Let it be a system with N processors attached to the MPI NoC and a schedule S of G slots per period to apply. Then, for every processor $P_i, i \in [1, N]$, there is a view S_i of the global schedule S, which is the information to be stored in the local slot table of the processor. Each of these views consists of G words, the slot table entries. The global schedule is the concatenation of the views in a big array of words $W_{ij}, i \in [1, N], j \in [1, G]$. The first G words contain the slot table for processor P_1, the next G words the table for P_2 and so on. An illustration of the global schedule format is given in Figure 5.4. Of course, different schedules may have different period lengths G and subsequently occupy areas of different length $(N \times G)$ in the memory.

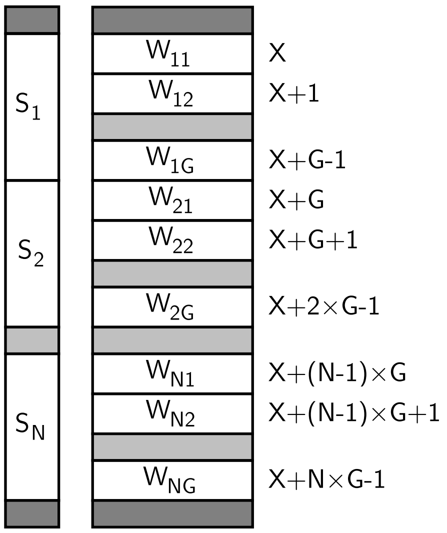

Figure 5.4: A schedule of size G for N processors in the SPM, written at location X.

5.1.4 Schedule fetch and apply - Main FSM

The basic functionality of the mode change controller is to read from the SPM the schedule to apply, push it to the broadcast tree and command the network adapters to apply the new schedule at some moment in the future. Then the controller waits until that moment arrives before being once again available to manage a new mode change. The interaction of the main FSM with its environment is shown in Figure 5.1. Figure 5.5 illustrates the ASM chart of the main FSM.

The state machine initially is at the *idle* state and the status of the controller is *free*. When an OCPio instruction to apply a new schedule is given by the system processor, the *start* signal triggers the FSM and the status of the controller is toggled to *busy*, the size of the new schedule and its location in the SPM are stored in registers and the machine transits to state *init*. The *id* of the recipient node is reset to 0 and a series of handshakes to the broadcast tree is commenced. The machine transits to state *push lead in*.

For every new schedule and for every recipient node, the first word to be sent is the size of the new schedule. Therefore, the *push lead in* state pushes to the broadcast tree the size of the new schedule. The index counter is set to 0 and the location of the first word of the global schedule is given as input to the SPMbefore incrementing. Unconditionally the machine transits to *push* state.

The SPM has synchronous read. This means that the data available at the output is the data corresponding to the location before the incrementing. The machine stays in *push* state enabling handshakes and incrementing the index counter and the location until the index counter reaches the size of the schedule, which means that the schedule view for the first recipient (S_1 in Figure 5.4) has been sent out. Then, the *node id* is incremented and the machine transits to *push lead in* state to repeat the same process for the second recipient.

The process is repeated for all of the recipients. The exit condition is given in state *push*, when the last word of the schedule view for the last recipient (W_{NG} in Figure 5.4) has been written to the handshaking channel.

Then the controller sets a moment in the future based on the current value of the period counter to perform the swap and commands the network adapters to apply the new schedule at that moment (Figure 4.4). The controller uses a fixed distance between the issuing of this command and the moment to do the swap in terms of TDM periods. This distance defines the resolution of the period counter. In the case of Figure 4.4, the distance is 2. After this, the machine transits to *wait moment* state.

The machine stays at *wait moment* state until the last clock cycle of the TDM period specified before. Then, the swap mode is disasserted and the status is restored to *free*. Moreover, the schedule size is updated with the size of the new schedule, so that the slot counter will increment up to this new value. At the same time (with a mesochronous notion of time) all of the network adapters are expected to swap to the new schedule, so that all of the slot counters will continue to operate in parallel. In Figure 4.4, at the end of TDM period $x + 2$ the new schedule is applied. This new schedule has bigger size. For this reason

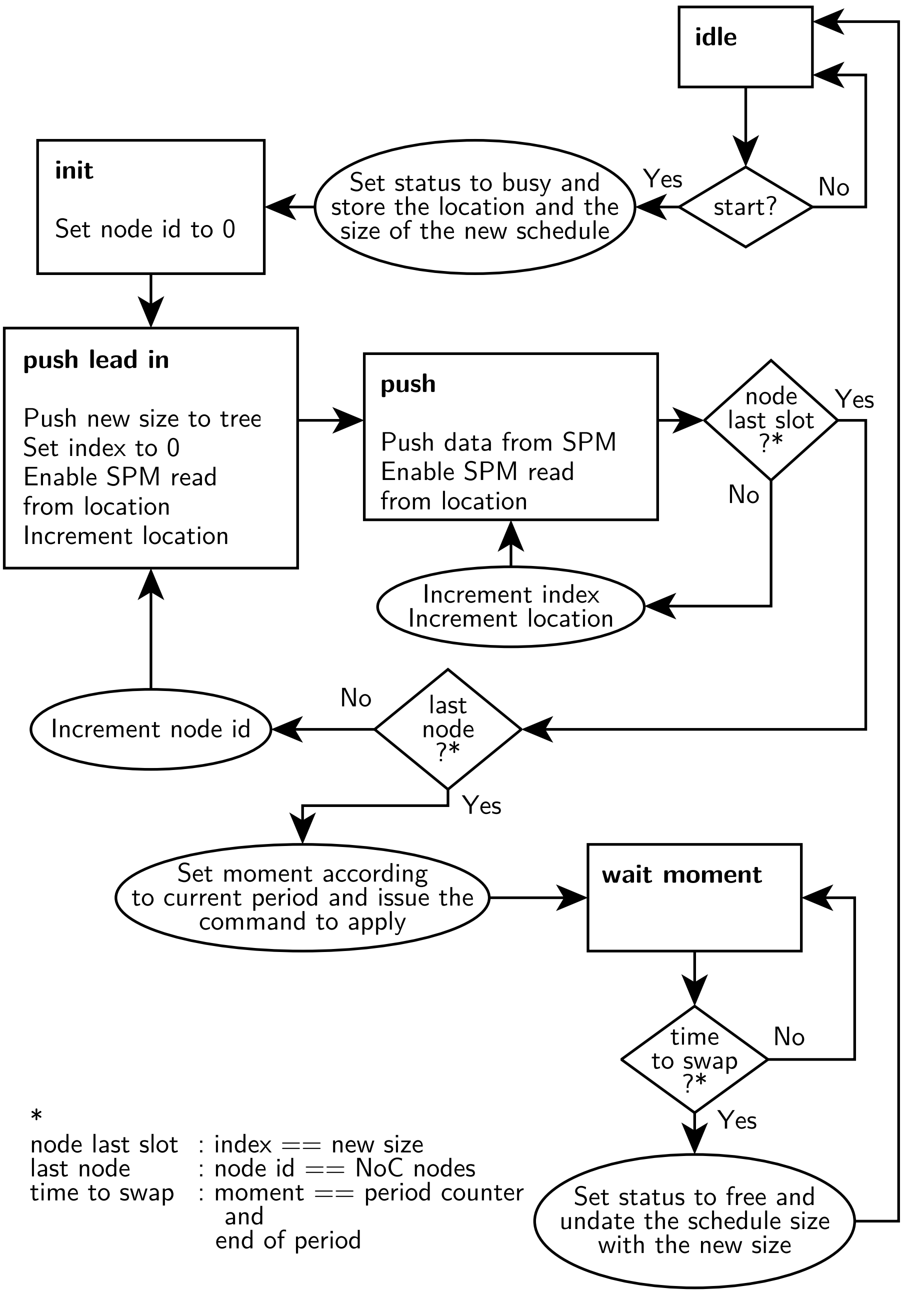
idle
start?
Yes
No
Set status to busy and store the location and the size of the new schedule
init
Set node id to 0
push lead in
Push new size to tree
Set index to 0
Enable SPM read from location
Increment location
push
Push data from SPM
Enable SPM read from location
node last slot ?*
Yes
No
Increment index
Increment location
last node ?*
No
Yes
Increment node id
Set moment according to current period and issue the command to apply
wait moment
time to swap ?*
No
Yes
Set status to free and undate the schedule size with the new size
*
node last slot : index == new size
last node : node id == NoC nodes
time to swap : moment == period counter and end of period

Figure 5.5: ASM chart of main state machine of mode change controller.

period $x+3$ is longer than the previous TDM periods. Then the machine returns to *idle* state.

5.2 Broadcast tree network

The broadcast tree operates at the asynchronous domain, utilizing handshakes between its components to propagate the information. The block interface of the broadcast tree is shown in Figure 5.6.

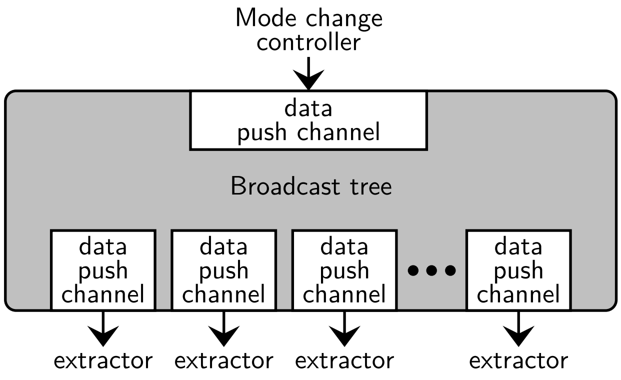

Figure 5.6: Broadcast tree block interface

It consists of simple asynchronous pipeline stages and broadcasting forks, providing a structure that transfers the data from the root of the tree to all of the leaves, without any intervention on them. Conceptually, a broadcast tree to 8 leaves with conventional asynchronous components as described in [35] is depicted in Figure 5.7.

The broadcast tree presented uses forks to three or two to reduce the fanout and inserts the same amount of pipeline stages between the root and the leaves for all of the leaves. This is not mandatory for the operation of the tree. Actually, any tree structure would be acceptable, and the links could be pipelined too. All that is important and has to be known for the analysis is the maximum path from the root of the tree to a leaf.

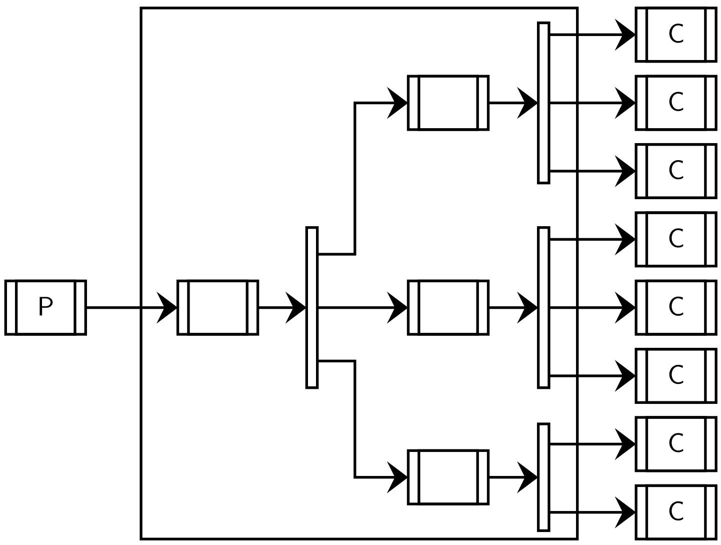

Figure 5.7: Broadcast tree to 8 leaves with conventional asynchronous latches and forks. The root of the tree is connected to a token producer (source) and the leaves to token consumers (sinks).

5.2.1 The asynchronous click element template

For the design of our asynchronous components we explore the usage of the *click elements*, a class of asynchronous components introduced in [37] that uses the 2-phase bundled data handshaking protocol and conventional flip-flops and logic gates instead of latches and C-elements.

A simple pipeline stage with the click template is illustrated in Figure 5.8. It has a *state* flip flip which toggles at every handshake driving the output request and the backward acknowledgement. When the *state* is different than the input request, then there is new data available at the input. Moreover, when the *state* is the same as the incoming acknowledgement, then the component can accept new data. These two conditions generate an event, the so-called *click* event. The logic function $state \neq a.req \wedge state = b.ack$ drives the *click* signal. When the conditions are met, the signal goes HIGH. This signal is used to toggle the *state* flip-flop. The toggling of the *state* forces the *click* signal to go back LOW, until the environment responds to the handshake and new data arrive at the input, so that a new event can occur. Therefore, the *click* signal is an asynchronous local clock, the rising edges of which clock the *state* flip-flop. The *click* signal drives also the clock of the register in the datapath, so that at the moment of the event the bundled data at the input is captured in the pipeline stage.

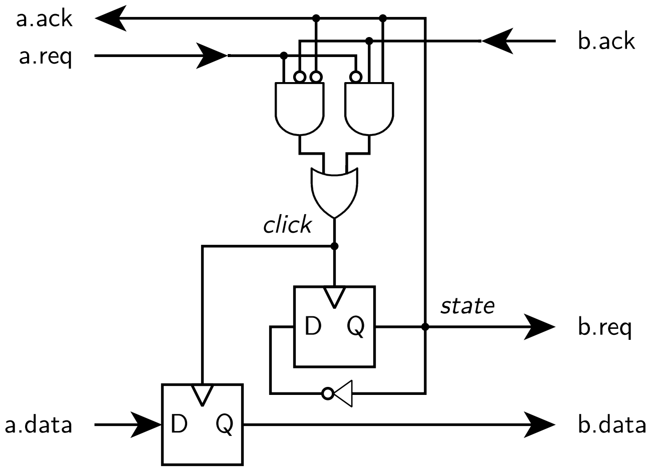

Figure 5.8: Click element pipeline stage.

5.2.2 Broadcast tree using click components

In [37] the example of a click join component that captures internally the data on a handshake event was given. Under the same guidelines we design a click fork component, the symbol and the schematic of which are given in Figure 5.9.

The *click* generating function is enhanced with more acknowledgements. The symbol and circuit provided are generic, for a fork with n outputs. The *click* function becomes therefore:

$$state \neq a.req \wedge (state = b_1.ack \wedge state = b_2.ack \wedge ... \wedge state = b_n.ack)$$

The *state* of the fork drives all of the output requests $b_i.req, i \in [1, n]$. Similarly to the join example of [37], the click fork also captures the data when a handshaking event occurs. Since the fork is used in our case to broadcast data, the data register drives all of the data outputs $b_i.data, i \in [1, n]$.

Compared to the conventional asynchronous components, from a functional point of view the click fork that captures data is equivalent to a handshaking latch followed by a fork. Considering this, the design of a broadcast tree with 8 leaves becomes as illustrated in Figure 5.10.

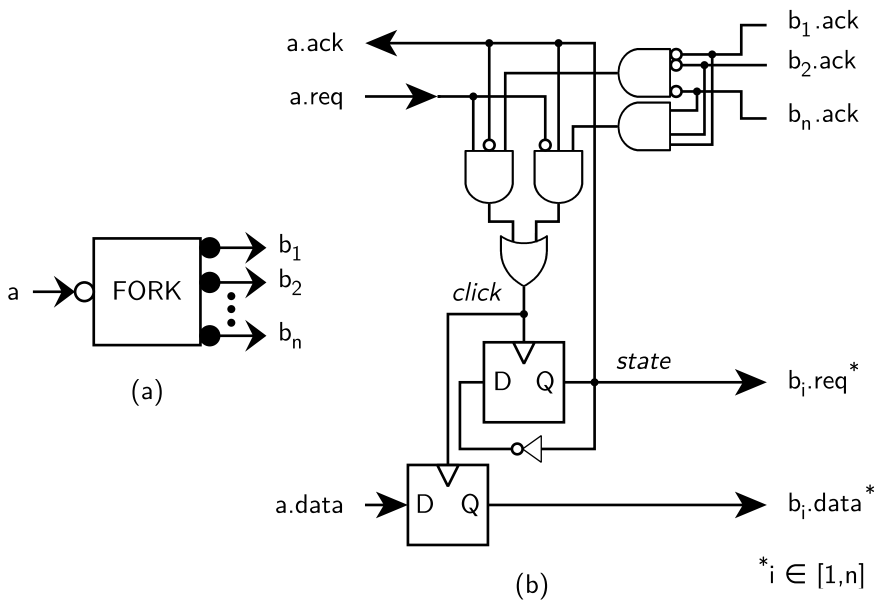

Figure 5.9: Click element fork. (a) Symbol (b) Schematic

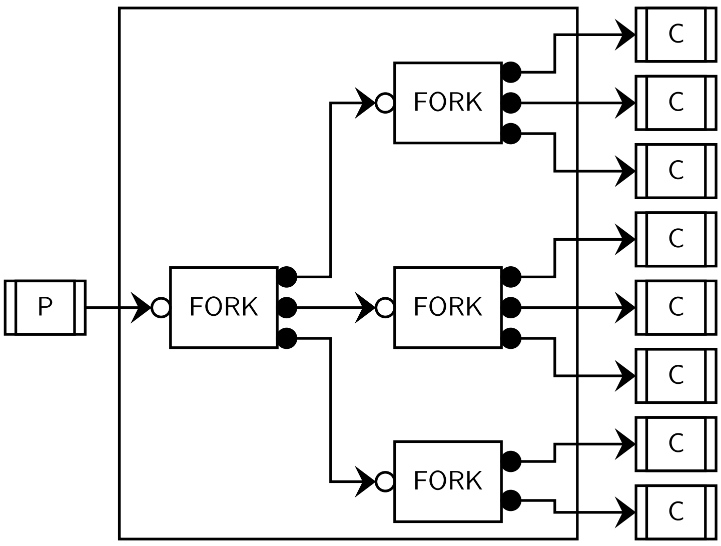

Figure 5.10: Broadcast tree to 8 leaves with click element data capturing forks. The root of the tree is connected to a token producer (source) and the leaves to token consumers (sinks).

5.3 Extractor

The extractor is an asynchronous component that consumes tokens from its input, extracts conditionally data out of the tokens and provides a synchronous memory write port at its output. The block interface is shown in Figure 5.11.

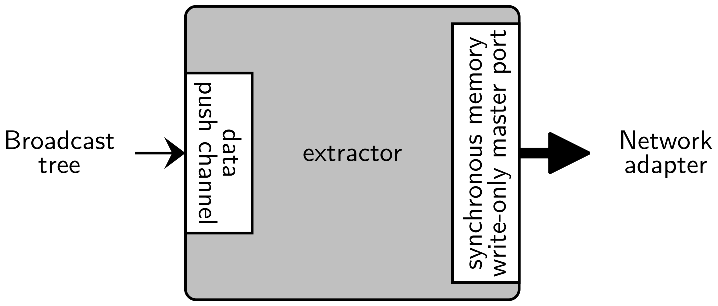

Figure 5.11: Extractor block interface.

Each extractor is associated to an ID tag. The tokens also carry ID tags. This way, according to the tag of a token the extractor can be aware of which information is relevant and acts accordingly. The action to be taken is handled by an asynchronous state machine. Therefore, the extractor can be regarded as an asynchronous consumer with an asynchronous state machine. The *click element* template was also used in the extractor design, which is illustrated in Figure 5.12.

The function generating the *click* pulse is modified to $state \neq req$ in order to accommodate for a consumer, which is a sink of tokens. This function is a simple XOR gate. The *click* pulse, together with the data at the input triggers the transitions of the state machine, which drives the synchronous memory write output port. In order to allow for the combinatorial to be stable, a matched delay is introduced at the request path, as seen in Figure 5.12.

The registers of the extractor's state machine are a register holding the size of the schedule view under extraction and a counter used for indexing.

As seen in the ASM chart of the FSM in Figure 5.13, the FSM has only two states, state *idle* and state *extract*. When in the *idle* state, the counter has the value 0 and the machine waits for a token with a matching ID tag. When the token arrives, its data is interpreted as the size of the new schedule, which is stored, and the machine transits to state *extract*, where the extraction of the new schedule takes place. For every token with matching ID that arrives, the write enable of the memory port is set to HIGH and the counter increments.

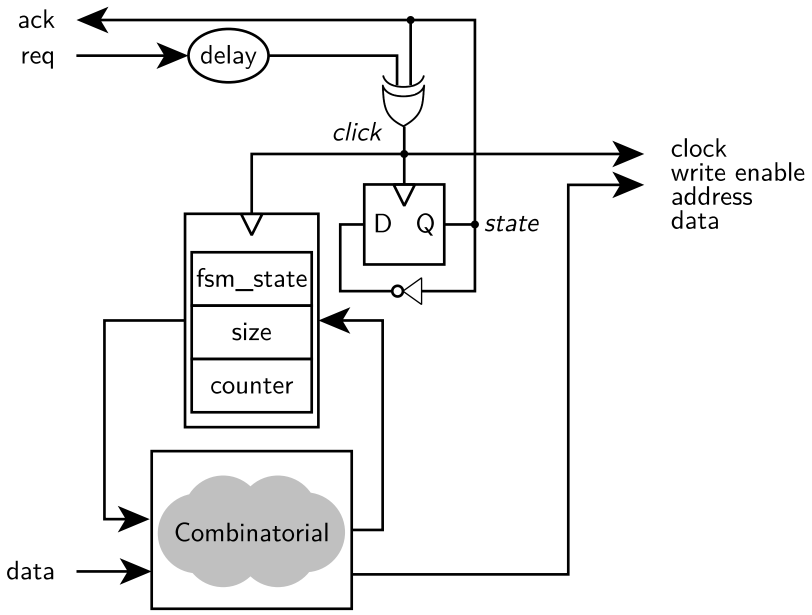

Figure 5.12: Asynchronous state machine of extractor.

The counter register and the data input are directly connected to the memory port as address and data respectively. Together with the write enable and the *click* signal as clock, they constitute the full synchronous memory write port. The machine stays in *extract* state incrementing the counter for every matching token until the counter reaches the size of the schedule. Then, the counter is set to 0 and the machine transits back to *idle* state, in order to wait for a new schedule.

From the previous it is apparent that there is no explicit signalling of a new set of data to be extracted. The machine returns to *idle* after a comparison of the indexing counter with the size register. The next matching token that will arrive must contain the size of the new set of data to be extracted.

5.4 Network adapter modifications

In order for the network adapter to cooperate with the mode change module, some modifications and additions had to be made. The modifications are related to the DMA table, the slot table and the slot counter of the adapter, while the

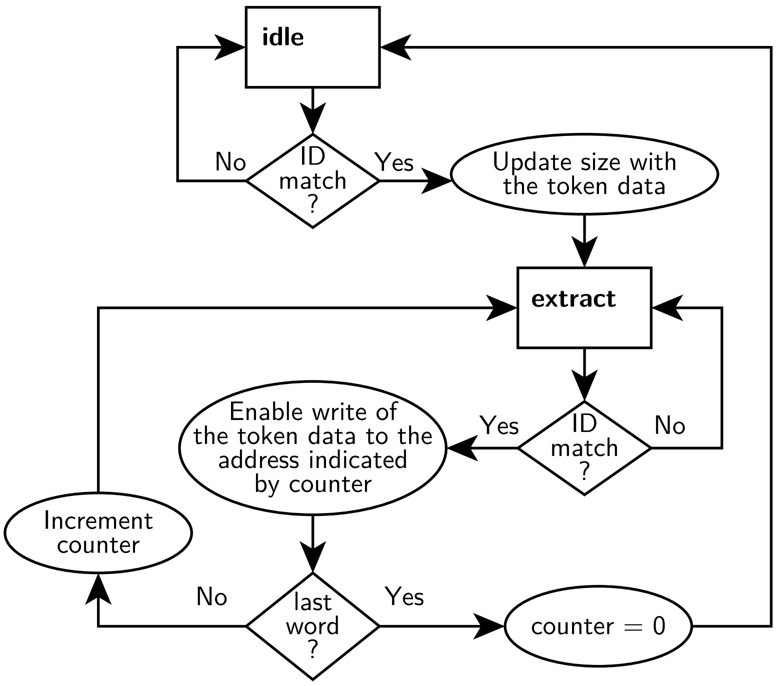
idle
ID match ?
No
Yes
Update size with the token data
extract
ID match ?
Yes
No
Enable write of the token data to the address indicated by counter
Increment counter
last word ?
No
Yes
counter = 0

Figure 5.13: ASM chart of extractor's state machine.

addition is the incorporation of the schedule period counter and the mode change logic.

5.4.1 DMA table

As mentioned in Subsection 2.2.3, the DMA table entries consist of five parts, 1) the address in the local SPM to read from, 2) the address in the remote SPM to write to, 3) the number of remaining words of the block being sent, 4) the routing information defining the path in the NoC and 5) some control flags.

The routing paths is the only DMA information that is related to the schedule and has to be altered due to a mode change. Future plans of T-CREST involve relocating this field to the slot table. Thus, even though this change does not affect the basic functionality of the mode change module, it has to be taken into account. For this reason, the route field was removed from the DMA table and moved to the slot table. As a result, a DMA entry ceases to be associated to a path, as it has been the case so far in the T-CREST NoC. Instead, a slot entry contains the route that the packet sent during that slot should follow. This allows for slots that point to the same DMA entry to navigate packets to the same recipient through different paths, increasing the flexibility of the schedules, but at the cost of additional memory space. For example, if a schedule defined x slots during a TDM period for core i to send a packet to core j, then with the previous arrangement of storing the routing information only one route would be specified in the DMA table of core i at the entry handling the sending to core j. With the new arrangement, every one of the x slots contains a route. These routes do not have to be the same as mentioned before. Still, care must be taken from behalf of the scheduler to avoid packet reordering.

5.4.2 Slot table

The slot table is converted into a simple dual port memory. So far the network adapter would write the slot table during the boot phase of the platform with information provided by the local processor through the OCPio port. This functionality is eliminated and replaced by the mode change module. A synchronous memory write-only input port is added to the interface of the network adapter and it is connected to the write port of the slot table.

In order to allow for the reception of a new schedule, while the previous one is still active, the size of the slot table is duplicated and the table is split into two areas. While one of the areas contains the active schedule and is read slot by slot at every TDM cycle, the other area can be written with a new schedule, without any interference to the active one. When the network adapter performs a swap to the new schedule, the roles of the two areas are also swapped.

As said, the size of the table is duplicated. Therefore the addresses to index the table are longer by one bit. This extra bit, which is the MSB bit of the address is used to select the area that contains the active schedule. When the MSB of the address of the read port is set to *HIGH*, then the MSB of the address of the write port is set to *LOW*. At a mode change the MSBs toggle, so that the network adapter will proceed by reading the new schedule, which becomes the active one, while the previous active area becomes the area to receive a new schedule.

5.4.3 Schedule counters and mode change

The network adapter slot counter is modified to count up to a reconfigurable value before being reset. Moreover, the adapter is enhanced with a schedule period counter, like the one of the mode change controller described in Subsection 5.1.2. As a result, the mode change module and the network adapters have the same mesochronous notion of the current slot and period.

One more port is added to the network adapter, the one giving the command to apply a new schedule at some moment in the future. The port is a 2-phase bundled data channel. The toggling of the 2-phase signal means that a new schedule must be applied. The size of the new schedule and the moment to be applied are the bundled data. Then, according to the local slot and period counter, when that moment comes, the network adapter simply toggles the MSBs of the read and write addresses of the slot table.

The 2-phase bundled data input port is not synchronized to the local clock domain. This is why the 2-phase signal has to go through a synchronizer to avoid metastability. The synchronizer is simply a series of flip-flops clocked with the local clock. The last flip-flop of the series provides the stable 2-phase signal to be used by the additional network adapter mode change logic. This technique to cross clock domains is described in [36]. With every flip-flop in the series the risk of metastability is reduced further. In our design two flip-flops were used to synchronize.

5.4.4 Summary of the new network adapter

In Figure 5.14 the modified network adapter is depicted. Comparing to Figure 2.6 presented in Subsection 2.2.3, the modifications are:

- Two extra ports have been added, one to the extractor and one to the mode change controller.
- The slot table is no longer written or read through the OCPio slave port to the processor. Instead it is written by the synchronous write-only memory input port from the extractor.
- The *route* field has been moved from the DMA table to the slot table.
- The slot table is a simple dual port memory and its size has been duplicated. The total area is split into two banks, one to be written by the extractor and one to be read by the adapter.
- A *bank select* flip flop has been added to define the active slot table memory bank.

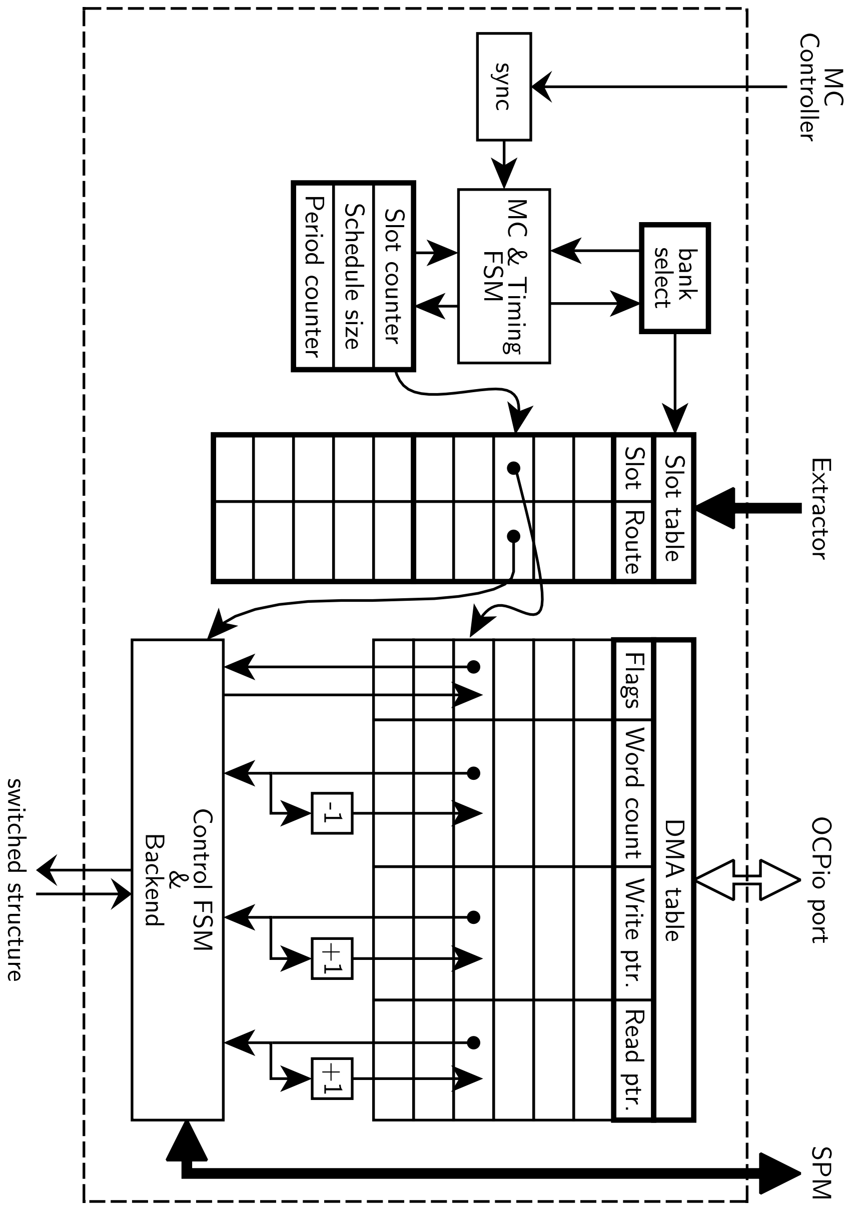

Figure 5.14: The modified network adapter.

- A reconfigurable ceiling to the slot counter before resetting was added by storing the size of the active schedule.
- The period counter was introduced.
- The mode change and timing FSM was added to handle the counters and the bank selection.
- A synchronizer was added between the mode change controller input port and the added FSM to allow for the 2-phase bundled data channel to cross the clock domain.

5.5 Static timing properties of the design

In this section some fundamental timing assumptions on which the design is based on are explained.

In an asynchronous circuit the frequency of the handshakes is defined by the slowest path. In the case of the broadcast tree and the extractor, the slowest path lies in the extractor, which incorporates a matched delay to compensate for its logic and the slot table write operation. This delay though is small, and it is a safe assumption that the extractor can operate at a higher frequency than the mesochronous clock. Therefore, the backward acknowledgement from the broadcast tree to the mode change controller is ignored. Instead, when pushing a new schedule, the controller handshakes blindly at every clock cycle, and the broadcast tree is fast enough to handle these handshakes.

In Figure 5.15 the handshakes of a path of 3 pipeline stages driven by the controller is shown. At every clock cycle the controller toggles the handshake signal and provides data A, B, C etc. Let it be that the mesochronous clock period is 2 *time units(tu)*. Stage 1 has a delay of 1.5 tu. Therefore it responds to the incoming handshake from the controller after 1.5 tu, but still the interval between the handshakes is the interval imposed by the controller. Stage 2 has a delay of 1 tu and Stage 3 delays 1.8 tu. As a result, the word A arrives at Stage 3 after 4.3 tu. In the worst case, the pipeline stages would be as slow as the controller and each word would arrive at Stage 3 after 3 clock cycles (6 tu in this example).

This way, if the path of a token in the broadcast tree has to go through B asynchronous pipeline stages, considering also the write operation from the extractor to the slot table, the maximum latency for a token from the controller to a slot table is $B + 1$ *cycles*.

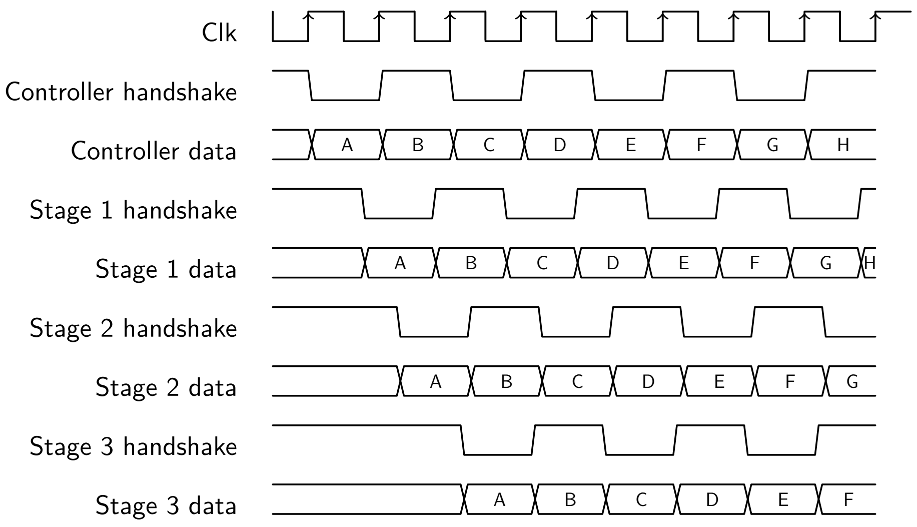

Figure 5.15: Mode change handshakes propagation.

When the controller has pushed the last word of a new schedule, it issues a command to apply the new schedule at some moment in the future. At that time the tokens still propagate in the broadcast tree. In addition, due to the mesochronous clock, there may exist some skew between the controller and the network adapters. The selected moment to apply the schedule must provide enough time for both of the pre-mentioned conditions.

T-CREST platform targets a 64 core system. For such a system the broadcast tree together with the extractors would have maximum path of 5 asynchronous pipeline stages. As a result the maximum latency for a token to be written to a slot table is 5 cycles of the mesochronous clock. Moreover, in [18] it has been shown that the T-CREST NoC, under certain conditions, can tolerate almost 3 clock cycles of skew on the mesochronous clock. As a result, when issuing a command to apply a new schedule, the selected moment has to be at least 8 cycles further.

The moment selection is done on schedule period granularity level. If no assumptions are made on the minimum size of a schedule, then the smallest possible schedule needs to be considered. A schedule must always have at least 2 slots, due to the NoC draining property. Moreover, the slot duration is 3 cycles. As a result, a schedule period is always greater or equal to 6 cycles. This means that if the schedule swap is set to be done after 2 TDM periods have elapsed, then there are always 8 cycles for the data to arrive and to provide enough time for the skew.

Moreover, setting the swap moment 2 TDM periods further also guarantees enough time for the clock domain crossing by the synchronizer at the network adapters. The additional cycles needed are as many as the flip-flops used to traverse the domain. Thus, swapping after two TDM periods is the earliest possible, which provides at any case enough time for all of the pre-mentioned conditions.

Alternatively, if the minimum size of a schedule is greater or equal to 4 slots, then the swap moment can be set to the end of the next TDM period. In order to set the swap moment even earlier, the current slot would have to be compared to the size of the running schedule. If there is enough time, the controller could command the new schedule application at the end of the current TDM period. Still, this additional functionality would be only an average case optimization that does not apply in all mode changes.

At this point, it has to be said that the swap moment selection does not have an upper bound. This means that the links of the tree may be pipelined to increase the throughput of the tree if the wires grow too long and the synchronizer may incorporate more flip-flops to decrease further the risk of metastability. In any case, all that is required is to postpone the swapping to a new schedule further in time.

Finally, due to the skew, a minimum separation has to be introduced between the swapping to a new schedule and the pushing of a new one. For example, the mode change controller may be ahead of a network adapter by 3 cycles. After doing a swap, the controller returns to *idle* state, while the network adapter has not swapped yet due to the skew. Then the controller is instructed to push a new schedule. If a token arrives to the slot table of the network adapter before the pending swap is completed, then the schedule will be corrupted. Still, the state machine of the mode change controller, after swapping, has to spend two cycles in state *idle*, one for the system processor to read that the module is free and one to initiate a new mode change procedure. Then, one more cycle is spent in state *init* to initialize the procedure. Therefore, the minimum separation of three cycles is met by default.

Chapter 6

Integration and Implementation

So far the design of the mode change module and the modifications to the T-CREST hardware platform to integrate the module have been presented. The integration though extends also to the software level. As it was specified in Section 4.2, the schedule selection and the policy to apply schedules is managed by the programmer.

6.1 Software support

A software API was developed with functions handling the system processor OCPio and OCPcore communication to the controller and the SPM of the mode change module respectively. These functions interact with the hardware through a local address space for the SPM and a single-address address space for the controller.

The developed software has a special way of managing the memory space of the SPM of the mode change module. On hardware level a maximum size for a schedule is defined. Then, we selected to set the size of the mode change SPM to fit two maximum-sized schedules. The software API splits the SPM into two banks. When a mode change is to be performed, the function *apply_schedule* is called, which:

1. copies the schedule to one bank of the SPM,
2. waits until the controller is free,
3. instructs the controller to fetch and apply the new schedule, providing the size and the location in the SPM as arguments.

After that, if another schedule needs to be applied, then the function may be called again, and the other bank of the SPM will be used to copy the new schedule. This way, by using two areas in the SPM in an interleaved way, some latency of doing a mode change is hidden.

Other strategies of utilizing the SPM and applying new schedules may be used, simply by modifying or enhancing the software API.

Since the NoC mode change is handled by the mode change module, the initialization of the network adapters by the local cores during the boot phase was removed.

6.2 T-CREST tool-chain integration

T-CREST provides a tool-chain to generate customizable platforms. The parameters to be customized are the size of the system, the topology, the link depth of the switched structure, the type of the IPs, the shared memory type, the application to be loaded, etc. The configuration parameters of the platform to be exported are described through xml files, which are parsed by python scripts exporting VHDL and Verilog hardware descriptions.

The platform generation process, except for generating descriptions according to the configuration parameters, it also incorporates the compilation of the user application and the generation of the binary code to be written in the local caches of the processors.

6.2.1 Generic generation of broadcast tree network structure

In order for the designed module to be integrated to this dynamic generation process, the module was described in VHDL using generic coding techniques to support systems of any size. Depending on the number of processors that are attached to the NoC, the structure of the broadcast tree network is generated accordingly to provide as many leaves as these processors. The generic broadcast tree is generated under two fundamental guidelines. All of the routes from the tree root to the leaves traverse the same number of pipeline stages, and only forks to 2 or 3 stages are used. Since all of the paths are of equal length, the tree is organised in levels, where a level is the order of a pipeline stage in the path from the root to the leaves. Each level contains parallel nodes implemented with click-element forks with capture. The number of levels, the number of nodes per level and the type of every node (fork to 2 or 3) is generated using functions and generic VHDL descriptions.

6.2.2 Tool-chain extensions and modifications

To allow for a complete integration to T-CREST, the configurable parameters in the xml files were enhanced with support for the mode change module. Additionally, the python scripts that parse the parameters and generate platform hardware descriptions were modified and extended to incorporate conditionally (under the guidance of the parameters) the mode change module. The extensions involve the instantiation of the generic mode change module in the platform hierarchy, while the modifications affect the component selection, the interfaces of the generated components and the interconnection of the instances. For example, if the mode change module is enabled, then the modified network adapter has to be used instead, which has an extended interface. This extended interface has to be taken into account in the platform hierarchy and be interconnected accordingly. As a result, the mode change has become a T-CREST platform configuration parameter.

Chapter 7

Latency estimation

The first thing to consider when doing the latency estimation is what constitutes the latency in a mode change. The presented design allows the programmer to define the schedule application policy and the management of the mode change SPM. Nevertheless, the software API accompanying the module provides a certain way of accessing the SPM, as described in Section 6.1. The estimation done is based on this approach. Similarly, estimations can be done for other approaches. So, we define as mode change latency the time from the moment that a decision to apply a schedule has been taken to the moment that the schedule is actually applied. This includes the schedule acquisition, fetch and application phases of the mode change.

Before proceeding, some fundamental assumptions have to be made. Even though T-CREST platform has a GALS timing organization, at the time no clock domain crossing is supported between the cores and the network adapters. Instead, the same clock is used for all. This is the reason why the read/write operations through the OCP ports are considered to have a duration of 1 cycle. Furthermore, the schedules are considered to be stored in the local data cache of the system processor. Lastly, the schedule swap moment is set to be after $P = 2$ full schedule periods in the future.

Having said the above, and considering as size of a schedule always the maximum size, the latency of a mode change can be divided in four steps:

Step 1 Schedule acquisition latency

It is the time required to copy the new schedule from the data cache of the system processor to the mode change SPM. The interval between OCPcore writes from the processor is 8 cycles due to index incrementing in software. If loop unrolling is used, then this interval can be reduced. The copying time is at most $size_{max} \times NODES \times 8\ cycles$.

Step 2 Waiting period

The controller may be occupied by a previous mode change operation. Totally, a mode change operation, once started, may last:

i 1 *cycle* for the initialization,

ii $NODES \times (size_{max} + 1)\ cycles$ for the pushing out of the new schedule and

iii $(P + 1) \times size_{max} \times 3 - 1\ cycles\ \ (periods \times \frac{slots}{period} \times \frac{cycles}{slot} - cycles)$, which means at least P periods until the swapping to the new schedule is completed.

Nevertheless, at the worst case that a previous mode change is ongoing, still a part of this operation is parallel to the schedule acquisition. The waiting period so is at most the difference of the sum of *i*, *ii* and *iii* from the schedule acquisition latency that was calculated in Step 1. Taking into account that a schedule cannot have size less than 2, that a system cannot have less than 2 processors attached to the NoC, and setting $P \leq 2$, we can derive that the worst case acquisition latency is always greater than the worst case waiting time. So, the waiting time is not considered in the worst case mode change latency. Still, 10 *cycles* are required for the processor to read that the mode change controller is free and instruct a mode change through the OCPio port.

Step 3 Schedule fetching latency

Once the instruction has been given, 2 *cycles* are needed to start and initialize the mode change, and $NODES \times (size_{max} + 1)\ cycles$ additionally for the fetching.

Step 4 Schedule apply latency

After the fetching has finished, at the worst case the swapping to the new schedule will happen after $3 \times (P + 1) \times size_{max} - 1\ cycles$.

Summing all of the above partial sums we get that the total worst case latency of a mode change, under the assumed configuration, lasts:

$$\begin{aligned} T_{WClatency} = 8 \times NODES \times size_{max} + NODES \times (size_{max} + 1) + \\ 3 \times (P + 1) \times size_{max} + 11\ cycles \end{aligned} \tag{7.1}$$

The calculation was done assuming always schedules of maximum size. For a given system, the size of the biggest schedule should be used to determine the worst case latency.

Chapter 8

Verification with test cases and results

In this chapter we present the test cases that were used to verify the functionality of the design, along with some results regarding the latency of a mode change and the resources used by the FPGA prototype.

For the verification, two perspectives have to be considered. The first perspective is related to the active communication channels. When doing a mode change, some virtual end-to-end circuits connecting cores may be disabled, while others are activated. Furthermore, a communication channel may remain open after a mode change, possibly with different bandwidth or paths to the destination. The second perspective is about the correctness of the transferred blocks. The mode changes are performed transparently to the execution of the tasks. Due to a mode change, a channel may change paths, or it may be disabled and enabled later on with another mode change. In any case, the block transfers must be correct. It is the programmer's responsibility to maintain the required connections open.

Two applications were developed to verify the design under the two perspectives. The applications define a set of tasks that use the MPI NoC to exchange blocks while the system processor performs mode changes within a static set of schedules. The serial port of the system processor is used to monitor correctness reports by the tasks.

For the developed test cases a 4 core T-CREST platform with 2x2 bitorus topology for the MPI NoC was generated. The core with ID 0 was used as the system processor, and therefore it was disconnected from the MPI NoC and connected to the mode change module instead.

Lastly, when in simulation, the T-CREST platform simulation model forces faster clock ticks to the serial port to speed up the simulation, which otherwise would be infeasible.

Table 8.1: Static set of global schedules for test cases. Twelve schedules are used. The size and the open core-to-core connections of every schedule are shown in the second and the third column respectively.

Schedule index	Size (slots/period)	Active communication channels
1	5	Core 1 → Core 2 , Core 2 → Core 3 Core 3 → Core 1 , Core 1 → Core 3 Core 2 → Core 1 , Core 3 → Core 2
2	4	None-to-none
3	5	Core 1 → Core 2 , Core 2 → Core 3 Core 3 → Core 1 , Core 1 → Core 3 Core 2 → Core 1 , Core 3 → Core 2
4	3	Core 1 → Core 2
5	3	Core 2 → Core 3
6	3	Core 3 → Core 1
7	3	Core 1 → Core 3
8	3	Core 2 → Core 1
9	3	Core 3 → Core 2
10	3	Core 1 → Core 2 , Core 2 → Core 1
11	3	Core 2 → Core 3 , Core 3 → Core 2
12	3	Core 3 → Core 1 , Core 1 → Core 3

8.1 Static set of schedules

A set of 12 global schedules was written to select among them for the mode changes of the test cases. The schedules activate different combinations of communication channels and utilize different routes and slot allocations. All of the schedules are compressed, with equal bandwidth for every active channel and one slot per period for every recipient. Table 8.1 presents the sizes of the schedules and the enabled communication channels per schedule. The schedules with indices 1 and 3 differ as they use different routes through the NoC for the connections.

8.2 Test case 1: Open communication channels verification

The purpose of this test case is to monitor the enabled communication channels while performing mode changes. For this reason an application was written. In this application, every core of the NoC is in an infinite loop, at every iteration of which the process attempts to set up a block transfer to every other NoC-attached core and checks on the reception of incoming blocks. There is neither MPI synchronization between the senders and the receivers, nor verification of the contents of the blocks being transferred. All that is important in this test case is to demonstrate that blocks are being transferred, and monitor on which channels the transfers are done.

T-CREST NoC is supported with a set of functions to handle the DMAs through the OCPio port and instantiate block transfers on software level. The function to instantiate a block sending is a blocking function, which will not return until the DMA handling the transfer is available and set up. This functionality does not fit the purpose of the first test case. For this reason, a modified function was used which instead of doing, it attempts a DMA set up returning a boolean flag reporting if the set up was successful.

As mentioned before, the serial port of the system processor, which is core 0, is used to send out reports. Whenever a process receives a block, it notifies the system processor about the reception and the ID of the sender. This notification is done through the shared memory. A set of semaphores is defined and stored in the shared memory, one for every NoC-attached core. The semaphores are set by the core processes respectively when receiving a block and cleared by the system processor when processed. From the moment that a block is received by a core until the printing out of the corresponding report by the system processor and the clearing of the core semaphore, the core process is blocked. In the meanwhile, other blocks may arrive, the reception of which is being ignored. After all, the purpose of the application is to demonstrate just that traffic exists, and report on the active channels.

The system processor, except for managing the reports through the serial port, it also performs mode changes. The schedules used are the schedules with index

Table 8.2: Simulation console output result of test case 1.

Time (nsec)	Serial port
987,5	Schedule 4
9187,5	1to2
12137,5	1to2
15137,5	1to2
18087,5	1to2
21037,5	1to2
24037,5	1to2
26987,5	1to2
29937,5	1to2
32250,0	Schedule 5
37187,5	1to2
38887,5	2to3
41837,5	2to3
44837,5	2to3
47787,5	2to3
50737,5	2to3
53687,5	2to3
56637,5	2to3
59637,5	2to3
62587,5	2to3
65537,5	2to3
67600,0	Schedule 6
72787,5	2to3
74437,5	3to1
77437,5	3to1
80437,5	3to1
83437,5	3to1
86437,5	3to1
89437,5	3to1
92437,5	3to1
95437,5	3to1
98437,5	3to1
101437,5	3to1
103200,0	Schedule 7
107937,5	3to1
109837,5	1to3
112737,5	1to3
115637,5	1to3
118537,5	1to3
121437,5	1to3
124337,5	1to3
127237,5	1to3
130137,5	1to3
133037,5	1to3
135937,5	1to3
138000,0	Schedule 8
143337,5	1to3
144937,5	2to1
147987,5	2to1
151037,5	2to1
153937,5	2to1
156987,5	2to1
160037,5	2to1
162937,5	2to1
165987,5	2to1
169037,5	2to1
172087,5	2to1
173850,0	Schedule 9
178787,5	2to1
180487,5	3to2
183737,5	3to2
186737,5	3to2
189637,5	3to2
192637,5	3to2
195637,5	3to2
198537,5	3to2
201537,5	3to2
204537,5	3to2
207437,5	3to2
209800,0	Schedule 10
215487,5	3to2
217537,5	2to1
219237,5	1to2
222787,5	1to2
224787,5	2to1
227837,5	2to1
229487,5	1to2
232737,5	1to2
234737,5	2to1
237787,5	2to1
239437,5	1to2
242887,5	1to2
244887,5	2to1
247937,5	2to1
249587,5	1to2
252687,5	1to2
254687,5	2to1
256650,0	Schedule 11
266737,5	2to1
268437,5	1to2
270437,5	2to3
273287,5	3to2
276837,5	3to2
278487,5	2to3
281637,5	2to3
283487,5	3to2
286787,5	3to2
288437,5	2to3
291737,5	2to3
293687,5	3to2
296937,5	3to2
298587,5	2to3
301737,5	2to3
303587,5	3to2
306787,5	3to2
308437,5	2to3
310850,0	Schedule 12
321037,5	3to2
322887,5	2to3
324487,5	3to1
327187,5	1to3
330837,5	1to3
332437,5	3to1
335837,5	3to1
337737,5	1to3
340737,5	1to3
342487,5	3to1
345737,5	3to1
347637,5	1to3
350887,5	1to3
352487,5	3to1
355637,5	3to1
357487,5	1to3
360787,5	1to3
362387,5	3to1
364550,0	Schedule 1
375187,5	3to1
376887,5	1to2
378587,5	1to3
380487,5	2to1
382987,5	3to2
385487,5	2to3
387987,5	3to1
390487,5	1to2
392987,5	1to3
395387,5	2to1
397987,5	3to2
400487,5	2to3
402987,5	3to1
405487,5	1to2
407987,5	1to3
410387,5	2to1
412987,5	3to2
415487,5	2to3
417987,5	3to1
420487,5	1to2
422987,5	1to3

4, 5, 6, 7, 8, 9, 10, 11, 12 and 1, as defined in Table 8.1, and they are applied in a cyclic fashion and with some delay between the mode changes. Every time the mode is changed, the index of the new schedule is also reported through the serial port.

The application was executed on the simulation model of the platform using Modelsim SE and the output of the serial port, together with timing annotation, is shown in Table 8.2. Comparing to the enabled communication channels per schedule, which are presented in Table 8.1, we can see that after a mode change, the open channels match the ones defined by the active schedule.

8.3 Test case 2: Correctness of functionality

The second test case examines the correctness of the block transfers when mode changes are performed transparently to the tasks executed on the cores of the NoC. A series of block transfers between the cores is done and then a validation function compares the received blocks to the expected ones. If the comparison is successful and the blocks are correct, then the system processor prints out from the serial port a report of correctness. Otherwise, an error is reported. The block transfers and the validation is repeated infinitely.

Similarly to the first task, the system processor manages the mode changes and the reports to the serial port. The schedules used in this test case are schedules 1, 4, 3 and 2 as presented in Table 8.1. Once again the shared memory is used to pass reports through semaphores from the tasks to the system processor. This time though, the reports are about the success or the failure of the validation of the transferred blocks.

Every process, except of course from the system process on processor 0, has 2 different blocks to send and two phases. The first block is associated to phase A and the second to phase B. The sending blocks of every process are different to one another. Each phase is divided in three steps:

1. At first, two block transfers are set up to both of the other two NoC processors. Both of the transfers send the same block. In Figure 8.1, at step 1 of phase A, processor 1 (P1) sets up the transfer of block b_{1A} to processors 2 (P2) and 3 (P3). P2 sets up the transfer of b_{2A} to P1 and P3, and P3 the transfer of b_{3A} to P1 and P2. Then, every process receives two blocks from the other two processors.

2. The received blocks are sent out again, without any change, but not to

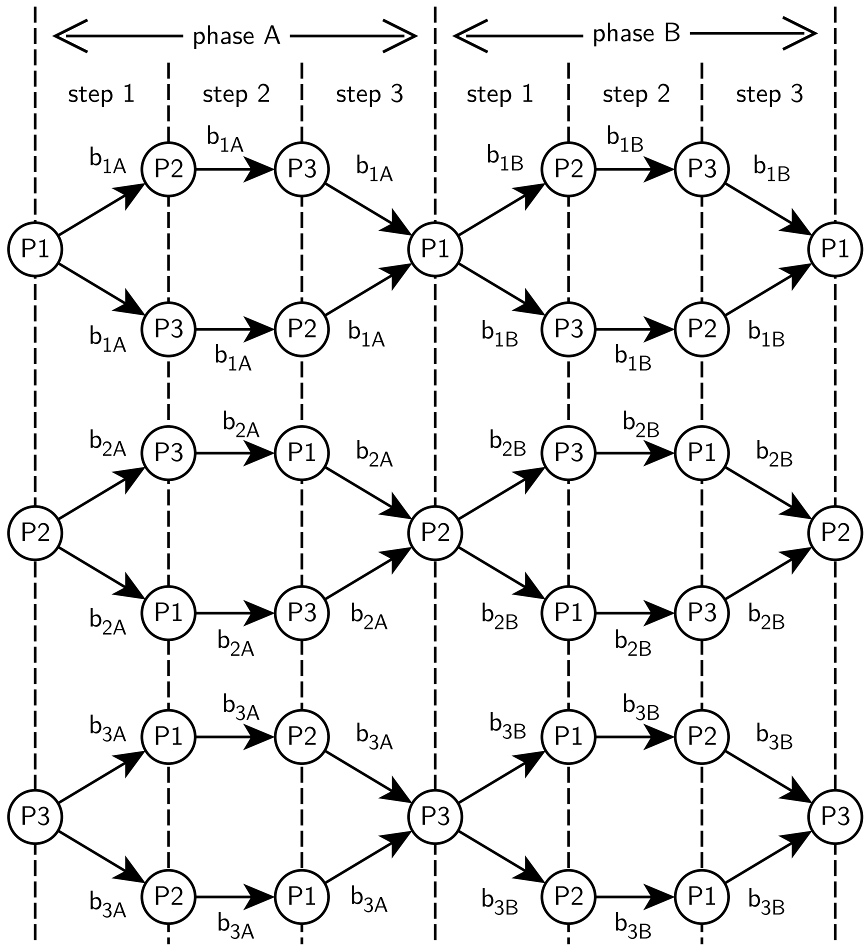

Figure 8.1: Test case 2 phases and block transfers diagram.

the processor that they came from. For example, at the end of step 1 P1 receives from P2 and P3. The block received from P2 is sent to P3 and the one received from P3 is sent to P2. The same applies for all NoC cores. Then, once again every process receives two blocks from the other two processors.

3. The third step is identical to the second one, with the addition of the correctness checking. All NoC processors send out the received blocks with the same policy and then they receive two blocks. Following the trace of the previous steps, it can be seen that at the end of the third step, each NoC processor receives two blocks that are expected to be the

same as the block that was sent out at step 1. Therefore, the received blocks are compared against it, and the boolean result of the comparison is passed to the semaphore related to the processor, so that the system processor can print out the corresponding report.

Table 8.3: Simulation console output result of test case 2.

Time (nsec)	Serial port	Time (nsec)	Serial port	Time (nsec)	Serial port
3387,5	Schedule 1	195687,5	Core 2 :OK	391537,5	Core 1 :OK
21837,5	Core 1 :OK	212037,5	Core 2 :OK	410337,5	Core 2 :OK
24587,5	Core 2 :OK	214787,5	Core 3 :OK	413087,5	Core 3 :OK
27187,5	Core 3 :OK	217387,5	Core 1 :OK	415687,5	Core 1 :OK
46087,5	Core 1 :OK	236287,5	Core 2 :OK	433387,5	Schedule 4
48837,5	Core 2 :OK	239037,5	Core 3 :OK	436387,5	Core 1 :OK
51437,5	Core 3 :OK	241637,5	Core 1 :OK	439737,5	Core 2 :OK
70387,5	Core 1 :OK	259737,5	Schedule 2	445137,5	Core 3 :OK
73137,5	Core 2 :OK	262737,5	Core 1 :OK	522287,5	Schedule 3
75737,5	Core 3 :OK	266087,5	Core 2 :OK	538037,5	Core 3 :OK
88337,5	Schedule 4	271487,5	Core 3 :OK	540787,5	Core 1 :OK
93837,5	Core 2 :OK	348725,0	Schedule 1	543387,5	Core 2 :OK
96587,5	Core 3 :OK	364437,5	Core 3 :OK	559737,5	Core 2 :OK
100187,5	Core 1 :OK	367187,5	Core 1 :OK	562487,5	Core 3 :OK
174537,5	Schedule 3	369787,5	Core 2 :OK	565087,5	Core 1 :OK
190337,5	Core 3 :OK	386187,5	Core 2 :OK	583987,5	Core 2 :OK
193087,5	Core 1 :OK	388937,5	Core 3 :OK	586737,5	Core 3 :OK

The second phase of the application is identical to the first one, but the blocks sent out at step 1 are different than the ones of the first phase. The reason for this is to exercise better the block transfers through the NoC. The two phases are interleaved in an infinite loop of execution.

The application, for proper execution, expects an equal bandwidth from all cores to all cores. Still, if some connection is not enabled by the schedule, then no failure is expected. Instead, the process that waits on that channel will be blocked in a polling loop, waiting for the reception. Subsequently, that process will not set up other block transfers and therefore other processes will be also blocked waiting for reception, much like a domino effect. When the bandwidth is restored by a mode change, the execution proceeds normally.

The application was executed on the simulation model of the platform under Modelsim SE and the serial port output of the system processor is shown in Table 8.3. As it can be seen in the table, when schedules 1 and 3 are active, which provide all-to-all communication, the blocks are being transferred continuously and correctly. When a mode change activates schedule 4 or 2, which are schedules that do not enable all the required communication channels, the execution is blocked. The hysteresis that can be seen, being the successful block transfer cycle after a mode change to an inadequate schedule, is due to the validating function which is executed at the end of step 3, introducing some delay between the actual reception and the printing of the report. Table 8.3 verifies that all of the block transfers are successful.

8.4 Mode change swap demonstration on Modelsim

In this section a mode change swap to a new schedule from the perspective of the network adapters is given through the waveforms generated by the simulation on Modelsim SE.

Some skew was introduced between the network adapters and the mode change controller. The controller and the network adapter of processor 1 are at the same phase. The adapter of processor two is delayed by one clock cycle and the adapter of processor 3 by two clock cycles. The skew can be seen in Figure 8.2 by observing the phase difference between the slot counters.

As shown in Figure 8.2, at first the state of the controller transits to state *wait_disassert*, and the 2-phase swap command signal is toggled. Since the current period, according to the period counter of the controller, is the period *100* (one-hot encoding), the moment to swap is set at least two periods later and therefore at the end of period *010* (*100* rotated left twice). The network adapters receive the command but do not act until the end of period *010*. Due to the skew, each network adapter has its own notion of the current slot and period. At the end of period *010* Figure 8.2 shows how every network adapter swaps to the new schedule, which lies in the other bank of the slot table, simply by toggling their bank selection flag.

8.5 Latency results

As explained in chapter 7, the latency is partially dependent on the mode change policy selected by the programmer. The following analysis is based on the mode change policy of the provided software API that supports the design. At first, we do an estimation based on the steps presented in chapter 7, but considering the actual sizes of the active and the new schedule of the case studied and not the maximum. Then, we compare this upper bound estimation to the reported

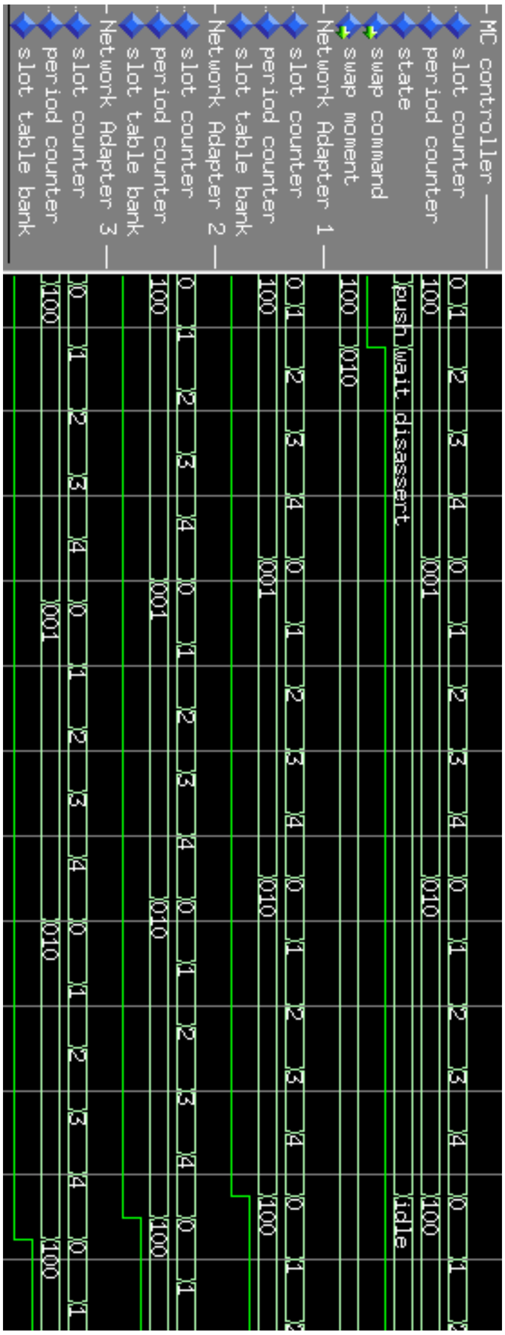

Figure 8.2: Modelsim simulation waveforms of a mode change swap.

one by the simulation, and finally we show that it is smaller than the worst case latency according to the formula of Chapter 7.

The case that is analysed is that the active schedule has 10 slots and a mode change to a schedule of 5 slots has been requested. The system has 4 cores attached to the NoC. Then the estimation for every step is as follows:

Schedule acquisition

$$5\ slot\ words\ \times\ 4\ cores\ \times\ 8\ \frac{cycles}{word}\ =\ 160\ cycles \tag{8.1}$$

Mode change instruction

$$10\ cycles \tag{8.2}$$

Schedule fetching

$$2\ cycles +\ 4\ cores\ \times (5\ slot\ words\ +\ 1\ size\ word)\ \times\ 1\ \frac{cycle}{word}\ =\ 26\ cycles \tag{8.3}$$

Apply schedule

$$3\ \frac{cycles}{slot}\ \times\ 3\ periods\ \times\ 10\ \frac{slots}{period}\ - 1\ cycle\ =\ 89\ cycles \tag{8.4}$$

The total of the above estimation is 285 *cycles*. The simulation of the above mode change resulted into 280 *cycles*. The difference is due to the fact that the formula of the schedule application latency takes into account the worst case scenario that the mode change command is given after a new period has just started. In such case, the current period has to elapse, plus two more periods as defined by the policy, before swapping to the new schedule.

If the schedule of 10 slots that was considered is the biggest schedule for this system, then the worst case latency, derived by a mode change from a schedule of 10 slots to another schedule of 10 slots is 465 *cycles*, which is bigger than the latency reported by the simulation.

If the maximum schedule size is considered to be 256 slots, then the worst case latency of a mode change in this system with 4 NoC-attached cores would be 11535 *cycles*. However, due to the T-CREST scheduler property to compress the generated schedules, the schedule sizes are kept to smaller values.

Finally, it has to be commented that the latency of the schedule acquisition is the factor that contributes more to the total sum. If loop unrolling when writing the schedule to the SPM or other policies are used, like utilizing a larger SPM, so that more global schedules can be kept in it, then the schedule acquisition latency could be reduced significantly, or even eliminated.

8.6 FPGA prototyping

The mode change extended T-CREST platform was successfully prototyped on the Xilinx ML605 FPGA-board. Special techniques were invoked to prototype asynchronous components on the FPGA, which typically targets synchronous designs.

8.6.1 Asynchronous components FPGA implementation

The T-CREST platform operates partially in the asynchronous domain, since the switched structure of the NoC is self-timed. Moreover, the designed mode change module passes through the asynchronous domain also. The asynchronous components that need special treatment on an FPGA are the matched delays that are used to guarantee bundling constraints (needed to satisfy setup and hold time requirements) and the Muller C-gates used by the routers of the NoC, and their FPGA prototyping has been elaborated in [38]. Since the mode change module uses just matched delays, we will focus on that. For more details refer to [38].

The VHDL *after* statement is not synthesizable and Xilinx ISE does not support constraints for minimum delay of a path. Therefore, we implement the delay as an array of Look Up Tables. The introduced delay grows with the length of the array. The logic function of each LUT is an AND gate, but since both inputs are connected to the same signal, each LUT behaves as a simple delay (Figure 8.3). Behavioural VHDL statements cannot be used to describe the array because they are being optimised out during synthesis. Instead, the matched delay is described structurally, as an array of instances of LUT components from the Xilinx library, the contents of which are hard-written and not inferred by a logic function.

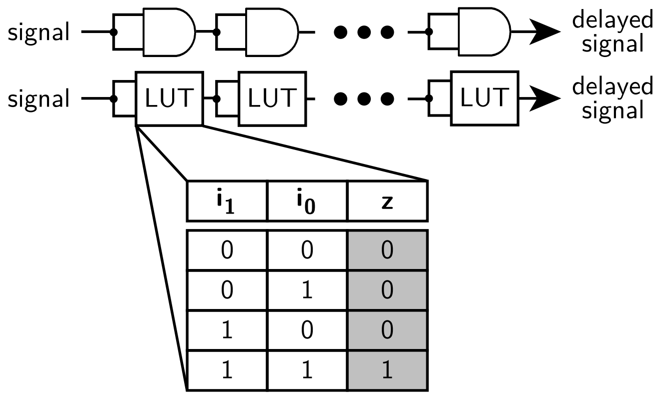

Figure 8.3: Matched delay as an array of Look Up Tables.

8.6.2 Resources report

The hierarchical ML605 FPGA-board resources utilization by a mode change extended T-CREST platform with a total of 4 processors is presented in Listing 8.1. As mentioned in the listing, for every column two numbers are reported, the first being the resources used solely at the hierarchical level of the instance and the second being the resources used totally by the instance from its hierarchical level and down.

Listing 8.1: T-CREST platform with mode change module resources utilization by hierarchy on Xilinx ML605 FPGA board.

```
+----------------------------------------------------------------------------------+
| Module             | Slices*   | Slice     | LUTs      | LUTRAM  | BRAM   |
|                    |           | Reg       |           |         | FIFO   |
+---------------------------------------------------------------------------- +
| top                | 2/13838   | 4/23182   | 4/27750   | 1/489   | 0/326  |
|  cmp               | 0/13728   | 0/22978   | 0/27588   | 0/488   | 0/70   |
|   arbiter          | 48/48     | 6/6       | 69/69     | 0/0     | 0/0    |
|   mode_change      | 0/108     | 0/205     | 0/172     | 0/0     | 0/0    |
|    controller      | 35/35     | 61/61     | 63/63     | 0/0     | 0/0    |
|    extractor 0     | 9/12      | 18/18     | 15/18     | 0/0     | 0/0    |
|    extractor 1     | 9/12      | 18/18     | 19/22     | 0/0     | 0/0    |
|    extractor 2     | 9/12      | 18/18     | 15/18     | 0/0     | 0/0    |
|    extractor 3     | 9/12      | 18/18     | 19/22     | 0/0     | 0/0    |
|    tree            | 0/25      | 0/72      | 0/29      | 0/0     | 0/0    |
|   noc              | 0/2868    | 0/4641    | 0/3500    | 0/0     | 0/4    |
|    node 0          | 1/583     | 1/882     | 1/680     | 0/0     | 0/1    |
|    node 1          | 1/744     | 1/1253    | 1/960     | 0/0     | 0/1    |
|     adapter        | 158/243   | 322/564   | 325/401   | 0/0     | 0/1    |
|      dma_table     | 21/85     | 2/242     | 24/76     | 0/0     | 0/0    |
|      slt_table     | 0/0       | 0/0       | 0/0       | 0/0     | 1/1    |
|     node 2         | 1/772     | 1/1253    | 1/926     | 0/0     | 0/1     |
|     node 3         | 1/769     | 1/1253    | 1/934     | 0/0     | 0/1     |
|    processor 0     | 45/2651   | 1/4552    | 56/5930   | 0/128   | 0/10    |
|    processor 1     | 45/2579   | 1/4520    | 56/5935   | 0/120   | 0/10    |
|    processor 2     | 45/2738   | 1/4520    | 56/5918   | 0/120   | 0/10    |
|    processor 3     | 49/2635   | 1/4520    | 56/5937   | 0/120   | 0/10    |
|    mc spm          | 2/2       | 2/2       | 1/1       | 0/0     | 0/2     |
|    noc spm 1       | 30/30     | 4/4       | 42/42     | 0/0     | 0/8     |
|    noc spm 2       | 39/39     | 4/4       | 42/42     | 0/0     | 0/8     |
|    noc spm 3       | 30/30     | 4/4       | 42/42     | 0/0     | 0/8     |
|   shared memory    | 17/108    | 20/200    | 28/158    | 0/0     | 0/256   |
+----------------------------------------------------------------------------+

* Slices can be packed with basic elements from multiple
    hierarchies.
  Therefore, a slice will be counted in every hierarchical module
  that each of its packed basic elements belong to.
** For each column, there are two numbers reported <A>/<B>.
   <A> is the number of elements that belong to that specific
    hierarchical module.
   <B> is the total number of elements from that hierarchical
    module and any lower level
   hierarchical modules below.
*** The LUTRAM column counts all LUTs used as memory including RAM,
     ROM, and shift registers.
```

It is clear that the mode change module in total is significantly smaller than a single network adapter. The important overhead is the system processor, which is managing the mode changes, together with the serial port. Configuration, user interacting and monitoring services are system services that are generally needed and can be combined with the mode change management in the system processor. Therefore we do not consider the system processor as a mode change extension overhead.

Chapter 9

Discussion

So far we have described the design and the implementation of the mode change module and its integration to the T-CREST platform, allowing for changing the schedule of the TDM NoC, reassigning the bandwidth during execution, with respect to the time-predictability and transparency specifications. The system processor, under some user defined mode change request policy, instructs the designed module to execute the change, which is then done within a static time bound, the calculation for which has been provided.

9.1 Alternative utilization of mode change module

The worst case latency that was calculated is dependent on the size of the platform and the biggest schedule supported. The fact that the latency increases with the number of processors attached to the NoC could lead to cases that the mode change cannot be performed as fast as the user of the platform would like it to be. After all, how much latency can be tolerated for a mode change is an application specific parameter. Still, in such case the designed module gives to the programmer the flexibility to handle the usage of the mode change SPM in a different way by modifying the provided software API. For instance, the

SPM could be increased to fit more than two schedules, if not all of the global schedules of the application. Then, the schedule acquisition latency could be transferred to a system booting phase, after which the latency of a mode change would not take into account the schedule acquisition latency.

9.2 Latency vs resources trade-off

Another way to minimize the latency would be to introduce a distributed mode change control system, where a control unit and a mode change SPM would exist for every network adapter, increasing further the resources required. At this point one should wonder how often is a mode change expected and if the trade-off between latency and resources is worth of making such a decision. T-CREST is a general-purpose time-predictable platform for hard real-time applications. Under this perspective, the designed module, which uses minimum resources, moves the mode change policy to the programming level and provides timing guarantees, is well suited for this platform.

9.3 Mode change under a real-time OS

Another issue to be discussed is how a request for a mode change can be triggered. As already mentioned in Subsection 4.3.1, a mode change can be the result of run-time dependent or independent events, like tasks starting, finishing or requesting for bandwidth according to their execution phase, external interrupts, timer events or simply the push of a button. From the above it is apparent that a mechanism to handle the requests is needed. It is up to the user of the platform to implement this mechanism according to the needs of his application. However, if a real-time operating system was ran on the platform, then the mode change could become part of it, having the O/S allocating tasks to processors and managing the bandwidth of the MPI interface.

9.4 Schedule generation as a system service

Another perspective to be considered is having the scheduler run as a system service. Then, the global schedules would not be stored but instead generated during run-time according to the current bandwidth requirements. In such case, the scheduler service must guarantee that it can always generate a schedule meeting the bandwidth requirements, and the latency of the schedule generation must be bound. The designed mode change module is compatible with this option too.

9.5 FPGA prototyping of asynchronous circuits

Implementing asynchronous circuits on FPGA boards is particularly challenging, since the synthesizer is a tool oriented to synchronous design for synchronous platforms. Actually, the only way to be done is by misusing the tools and manipulating the synthesis process with manual constraints and directives on signal level. Obtaining timing closure becomes infeasible, and the only way to reach a result is to iterate between synthesizing and performing post Place & Route simulation until the proper constraints have been set. This process becomes even more challenging when similar techniques to clock gating are used in the asynchronous domain, as it has been the case for the switched structure of the T-CREST NoC.

Chapter 10

Conclusion

The purpose of this thesis has been the enhancement of T-CREST platform, a Network-on-Chip based multi-processor platform, with the ability to change the mode of operation of the TDM NoC, reassigning the Message Passing Interface bandwidth between the cores with respect to the time-predictability asset of T-CREST.

Some default specifications were set regarding the desired operation of the mode changes. T-CREST is a general-purpose platform that the user of which must be able to configure and adjust to his needs. From this point of view, the mode change module that was designed is flexible and configurable from the programming level, allowing the end-users to define their schedules and their mode change policy. Moreover, the transparency of a mode change to the tasks executed on the processors was a fundamental specification. The designed module does not interfere at all with the tasks' execution, which continues uninterruptedly during a mode change. Special focus was given to analyse the phases of a mode change and examine them separately, comparing their requirements to the available options by T-CREST in order to identify the necessary modifications. Our module utilizes the existing hardware where possible and minimizes the additional resources.

Due to the Globally Asynchronous Locally Synchronous timing organization of T-CREST platform and the mesochronous domain of the TDM NoC network adapters that allows for skew between them, we explored asynchronous handshaking techniques to propagate data in the asynchronous space of the GALS organization, along with the *Click element* template [37] to design the asynchronous components that served our design.

The theoretical analysis, which was based on the static timing properties of the design, provided a bound to the worst case execution time of a mode change. The module was integrated and tested on T-CREST with two applications exercising the transparent bandwidth re-assignment while blocks are being successfully exchanged among the tasks. The correctness of functionality under skewed operation and the latency estimation were verified by simulation.

The contribution of this thesis project can be summarized to the following points:

- Design and implementation of mode change module for the TDM NoC based multi-processor time-predictable T-CREST platform
- Design and implementation of the asynchronous broadcast fork and state machine components using the click element template
- Integration of the mode change module to T-CREST hardware platform
- Modification and extension of the T-CREST network adapter to incorporate the mode change functionality
- Development of mode change supporting software API
- Two applications exercising the Message Passing Interface of T-CREST platform
- Integration of mode change module and software API to T-CREST toolchain
- FPGA prototyping of mode change extended T-CREST platform

As future work, the generation of global schedules as a service executed during run-time is an aspect that can be explored. Already in [39] an online allocation of TDM slots is described for contention-free routing. Furthermore, the development of a real-time operating system that will manage the allocation of tasks to the processors and the bandwidth offered to them by the MPI is a very interesting perspective.

Bibliography

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[2] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, Oct 1974.

[3] M. Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *Solid-State Circuits Society Newsletter, IEEE*, 12(1):11–13, Winter 2007.

[4] H Sutter. A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):16–23, 2005.

[5] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[6] William J. Dally and Brian Towles. Route Packets, Not Wires: On-chip Inteconnection Networks. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 684–689, New York, NY, USA, 2001. ACM.

[7] Luca Benini and Giovanni De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, January 2002.

[8] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli. Addressing the System-on-a-chip Interconnect Woes Through Communication-based Design. In *Proceedings of the 38th*

Annual Design Automation Conference, DAC '01, pages 667–672, New York, NY, USA, 2001. ACM.

[9] S. Kumar, A Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A Hemani. A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 105–112, 2002.

[10] Ankur Agarwal, Cyril Iskander, and Ravi Shankar. Survey of network on chip (NoC) architectures & contributions. *Journal of Engineering, Computing and Architecture*, 3(1), 2009.

[11] Shaoteng Liu, Axel Jantsch, and Zhonghai Lu. Analysis and Evaluation of Circuit Switched NoC and Packet Switched NoC. In *Proceedings of the 2013 Euromicro Conference on Digital System Design*, DSD '13, pages 21–28, Washington, DC, USA, 2013. IEEE Computer Society.

[12] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS Architecture and Design Process for Network on Chip. *Journal of Systems Architecture*, 50(2-3):105–128, February 2004.

[13] Davide Bertozzi and Luca Benini. Xpipes: A network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2):18–31, 2004.

[14] Cesar Albenes Zeferino and Altamiro Amadeu Susin. SoCIN: A Parametric and Scalable Network-on-Chip. In *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design*, SBCCI '03, pages 169–174, Washington, DC, USA, Sept 2003. IEEE Computer Society.

[15] Adrijean Adriahantenaina, Herve Charlery, Alain Greiner, Laurent Mortiez, and Cesar Albenes Zeferino. SPIN: A Scalable, Packet Switched, On-Chip Micro-Network. In *Proceedings of the Conference on Design, Automation and Test in Europe: Designers' Forum - Volume 2*, DATE '03, pages 70–73, Washington, DC, USA, 2003. IEEE Computer Society.

[16] César Marcon, Ramon Fernandes, Rodrigo Cataldo, Fernando Grando, Thais Webber, Ana Benso, and Letícia B. Poehls. Tiny NoC: A 3D Mesh Topology with Router Channel Optimization for Area and Latency Minimization. In *Proceedings of the 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, VLSID '14, pages 228–233, Washington, DC, USA, 2014. IEEE Computer Society.

[17] Nikolay Kavaldjiev, Gerard J. M. Smit, Pierre G. Jansen, and Pascal T. Wolkotte. A Virtual Channel Network-on-Chip for GT and BE Traffic. In *Proceedings of the IEEE Computer Society Annual Symposium on Emerging*

VLSI Technologies and Architectures, ISVLSI '06, Washington, DC, USA, 2006. IEEE Computer Society.

[18] E. Kasapaki and J. Sparsø. Argo: A Time-Elastic Time-Division-Multiplexed NOC using Asynchronous Routers. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 45–52. IEEE Computer Society Press, 2014.

[19] Daniel Wiklund and Dake Liu. SoCBUS: Switched Network on Chip for Hard Real Time Embedded Systems. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, page 8 pp., Washington, DC, USA, 2003. IEEE Computer Society.

[20] C. Hilton and B. Nelson. PNoC: a flexible circuit-switched NoC for FPGA-based systems. *Computers and Digital Techniques, IEE Proceedings -*, 153(3):181–188, May 2006.

[21] Pascal T. Wolkotte, Gerard J. M. Smit, Gerard K. Rauwerda, and Lodewijk T. Smit. An Energy-Efficient Reconfigurable Circuit-Switched Network-on-Chip. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3 - Volume 04*, IPDPS '05, page 155a, Washington, DC, USA, 2005. IEEE Computer Society.

[22] Daniel Marcos Chapiro. *Globally-asynchronous Locally-synchronous Systems (Performance, Reliability, Digital)*. PhD thesis, Stanford, CA, USA, 1985. AAI8506166.

[23] OCP International Partnership. Open Core Protocol specification, release 3.0, 2009. http://www.ocpip.org.

[24] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: Concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414–421, Sept 2005.

[25] T. Bjerregaard. *The MANGO clockless network-on-chip: Concepts and implementation*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2005. Series IMM-PHD-2005-153. Supervised by Assoc. Prof. Jens Sparsø, IMM.

[26] E. Beigne, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin. An Asynchronous NOC Architecture Providing Low Latency Service and Its Multi-Level Design Framework. In *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, ASYNC '05, pages 54–63, Washington, DC, USA, 2005. IEEE Computer Society.

[27] Andreas Hansson, Mahesh Subburaman, and Kees Goossens. aelite: A Flit-Synchronous Network on Chip with Composable and Predictable Services. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 250–255, 3001 Leuven, Belgium, 2009. European Design and Automation Association.

[28] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed Bandwidth Using Looped Containers in Temporally Disjoint Networks Within the Nostrum Network on Chip. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '04, pages 890–895, Washington, DC, USA, 2004. IEEE Computer Society.

[29] Radu Andrei Stefan, Anca Molnos, and Kees Goossens. dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-Up. *IEEE Transactions on Computers*, 63(3):583–594, March 2014.

[30] Rasmus Bo Sørensen, Jens Sparsø, Mark Ruvald Pedersen, and Jaspur Højgaard. A Metaheuristic Scheduler for Time Division Multiplexed Networks-on-Chip. In *IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2014.

[31] J. Sparsø, E. Kasapaki, and M. Schoeberl. An Area-efficient Network Interface for a TDM-based Network-on-Chip. In *Proc. Design Automation and Test in Europe (DATE)*, pages 1044–1047, 2013.

[32] E. Kasapaki, J. Sparsø, R.B. Sørensen, and K.W.G. Goossens. Router Designs for an Asynchronous Time-Division-Multiplexed Network-on-Chip. In *Proc. of Euromicro Conference on Digital System Design (DSD)*, pages 319–326, September 2013.

[33] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[34] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos Reference Handbook. Available online at `http://patmos.compute.dtu.dk/patmos_handbook.pdf`, 2014.

[35] J. Sparsø and S. Furber. *Principles of asynchronous circuit design : a systems perspective.* Kluwer Academic Publishers, 2001.

[36] Pong P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability.* Wiley-IEEE Press, 2006.

[37] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click elements: An implementation style for data-driven compilation. In *Proceedings of the 2010 IEEE Symposium on Asynchronous Circuits and Systems*, ASYNC '10, pages 3–14, Washington, DC, USA, 2010. IEEE Computer Society.

[38] T-CREST deliverable D3.6. ”FPGA implementation of self-timed NOC”. Available online at http://www.t-crest.org/page/results.

[39] Radu Stefan, Ashkan Beyranvand Nejad, and Kees Goossens. Online allocation for contention-free-routing NoCs. In *Proceedings of the 2012 Interconnection Network Architecture: On-Chip, Multi-Chip Workshop*, INA-OCMC '12, pages 13–16, New York, NY, USA, 2012. ACM.

VHDL code of the mode change module

Listing 1: File: mc_defs.vhd

```
-------------------------------------------------------------------
-- Definitions package
--
-- Author: Ioannis Kotleas
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use work.config.all;
use work.noc_defs.all;

package mc_defs is
  -----------------------------------------------------------------
  -- Constants for schedule and counter sizes
  constant MAX_SCHEDULE_SIZE  : integer := 256;
  constant BOOT_SCHEDULE_SIZE : integer := 10;
  constant SLOT_CNT_SIZE      : integer := integer(ceil(log2(real(
    MAX_SCHEDULE_SIZE))));

  -- The schedule periods counter is one-hot encoded. The size
  -- defines the amount of periods to wait before swapping to the
  -- new schedule. A size of 3 gives two periods interval.
  constant PERIOD_CNT_SIZE : integer := 3;

  -----------------------------------------------------------------
  -- Mode change SDP memory that holds the schedule
  constant MC_SPM_SIZE : integer := 2 * MAX_SCHEDULE_SIZE * NODES;
  constant MC_SPM_DATA : integer := DMA_IND_WIDTH + 3 + BANK2_W;
  constant MC_SPM_ADDR : integer := integer(ceil(log2(real(
    MC_SPM_SIZE))));

  -- Read-only port of MC SDP memory
  type mc_mem_m is record
    address : std_logic_vector(MC_SPM_ADDR - 1 downto 0);
  end record mc_mem_m;

  type mc_mem_s is record
    data : std_logic_vector(MC_SPM_DATA - 1 downto 0);
  end record mc_mem_s;

  -----------------------------------------------------------------
  -- Subtype definitions for data propagated on the broadcast
  -- tree
  subtype mc_word_t is std_logic_vector(MC_SPM_DATA - 1 downto 0);
  subtype node_id_t is unsigned(integer(ceil(log2(real(NODES)))) -
    1 downto 0);

  -----------------------------------------------------------------
  -- Data handshake channel records to propagate a schedule through
  -- the tree
  type mc_data_channel_f is record
```

```
    req  : std_logic;
    id   : node_id_t;
    data : mc_word_t;
  end record mc_data_channel_f;

  type mc_data_channel_b is record
    ack : std_logic;
  end record mc_data_channel_b;

  -----------------------------------------------------------------
  -- Swap channel record. No acknowledgment used.
  type mc_swap_channel_f is record
    req    : std_logic;
    moment : std_logic_vector(PERIOD_CNT_SIZE - 1 downto 0);
  end record mc_swap_channel_f;

  -----------------------------------------------------------------
  -- Combined data and swap channels to reduce wiring
  type mc_channels_f is record
    data_channel : mc_data_channel_f;
    swap_channel : mc_swap_channel_f;
  end record mc_channels_f;

  type mc_channels_b is record
    data_channel : mc_data_channel_b;
  end record mc_channels_b;

  -----------------------------------------------------------------
  -- Unconstrained arrays of combined records. To be used for
  -- dynamic wiring.
  type mc_array_f is array (natural range <>) of mc_channels_f;
  type mc_array_b is array (natural range <>) of mc_channels_b;

  -----------------------------------------------------------------
  -- Interface between controller command bus and network adapters
  type mc_join_channel_f is record
    req    : std_logic;
    size   : unsigned(SLOT_CNT_SIZE - 1 downto 0);
    moment : std_logic_vector(PERIOD_CNT_SIZE - 1 downto 0);
  end record mc_join_channel_f;

  type slot_table_write_interface is record
    wen   : std_logic;
    waddr : std_logic_vector(SLOT_CNT_SIZE downto 0);
    wdata : mc_word_t;
    wclk  : std_logic;
  end record slot_table_write_interface;

  type mc_to_ni is record
    swap_channel : mc_join_channel_f;
    slot_write   : slot_table_write_interface;
  end record mc_to_ni;

  -----------------------------------------------------------------
  -- Unconstrained array of mc/adapter interfaces. To
```

```vhdl
  -- be used for dynamic wiring.
  type mc_to_ni_a is array (natural range <>) of mc_to_ni;

  ------------------------------------------------------------------
  -- Broadcast tree unconstrained parameters definitions. Used to
  -- generate a tree of total leaves amount provided as a generic.
  type tree_level is array (0 to 1) of integer;
  type tree_sizes is array (natural range <>) of tree_level;

  function calc_levels(nods : natural) return natural;
  function calc_tree_levels(nods : integer) return tree_sizes;
  function calc_connections_of_level(sizes : tree_sizes; level :
    integer) return integer;
  function calc_total_connections(sizes : tree_sizes; levels :
    natural) return integer;
  function calc_connections_of_previous_levels(sizes : tree_sizes;
    level : integer) return integer;

end package mc_defs;

package body mc_defs is
  function calc_levels(nods : natural) return natural is
    variable result : integer := 0;
    variable nodes  : natural := nods;
  begin
    while nodes > 1 loop
      nodes  := natural(ceil(real(nodes) / real(3)));
      result := result + 1;
    end loop;
    return result;
  end function;

  function calc_tree_levels(nods : integer) return tree_sizes is
    constant levels       : natural := calc_levels(nods);
    variable result       : tree_sizes(levels - 1 downto 0);
    variable temp, u_nods : integer := nods;
  begin
    for level in 0 to levels - 1 loop
      temp              := u_nods;
      u_nods            := 0;
      result(level)(0)  := 0;
      result(level)(1)  := 0;
      while temp > 4 loop
        result(level)(0)  := result(level)(0) + 1;
        temp              := temp - 3;
        u_nods            := u_nods + 1;
      end loop;
      if temp = 3 then
        result(level)(0)  := result(level)(0) + 1;
        u_nods            := u_nods + 1;
      elsif temp = 4 then
        result(level)(1)  := 2;
        u_nods            := u_nods + 2;
      elsif temp = 2 then
        result(level)(1)  := 1;
        u_nods           := u_nods + 1;
      end if;
    end loop;
    return result;
  end function;

  function calc_connections_of_level(sizes : tree_sizes; level :
    integer) return integer is
    variable result : integer := 0;
  begin
    result := sizes(level)(0) * 3 + sizes(level)(1) * 2;
    return result;
  end function;

  function calc_connections_of_previous_levels(sizes : tree_sizes;
    level : integer) return integer is
    variable result : integer := 0;
  begin
    for I in 0 to level - 1 loop
      result := result + calc_connections_of_level(sizes, I);
    end loop;
    return result;
  end function;

  function calc_total_connections(sizes : tree_sizes; levels :
    natural) return integer is
    variable result : integer := 0;
  begin
    for I in 0 to levels - 1 loop
      result := result + calc_connections_of_level(sizes, I);
    end loop;
    return result;
  end function;

end mc_defs;
```

Listing 2: File: mc_controller.vhd

```vhdl
---------------------------------------------------------------------
-- Mode change controller
--
-- Author: Ioannis Kotleas
---------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use work.mc_defs.all;
use ieee.numeric_std.all;
use work.config.all;
use work.noc_defs.all;
use work.ocp.all;

entity mc_controller is
  port(
    clk             : in  std_logic;
    reset           : in  std_logic;

    -- SDP Memory interface
    spm_in          : in  mc_mem_s;
    spm_out         : out mc_mem_m;

    -- Configuration OCP interface
    proc_in         : in  ocp_io_m;
    proc_out        : out ocp_io_s;

    --Broadcast tree interface
    push_schedule_f : out mc_channels_f;
    push_schedule_b : in  mc_channels_b
  );
end mc_controller;

architecture rtl of mc_controller is
  -- Pushing schedule to broadcast tree
  signal push_handshake        : std_logic;
  signal enable_push_handshake : boolean;
  alias push_data              : std_logic_vector(MC_SPM_DATA - 1
    downto 0) is push_schedule_f.data_channel.data;

  -- Index of data to read from memory
  signal location              : unsigned(MC_SPM_ADDR - 1 downto 0)
    ;
  signal next_location         : unsigned(MC_SPM_ADDR - 1 downto 0)
    ;
  -- Recipient network adapter id
  signal next_node_id, node_id : node_id_t;
  -- Words per recipient counter
  signal index_cnt             : unsigned(SLOT_CNT_SIZE - 1 downto
    0);
  signal next_index_cnt        : unsigned(SLOT_CNT_SIZE - 1 downto
    0);

  -- Swap command flip flop - 2 phase signal
  signal swap_2ph : std_logic;

  -- Mode change module state
  signal toggle_occupied : boolean;
  signal occupied        : std_logic;

  --To be written/driven by the OCP-IO:
  alias schedule_location   : std_logic_vector(MC_SPM_ADDR - 1
    downto 0) is proc_in.MData(MC_SPM_ADDR + 16 - 1 downto 16);
  signal schedule_max_index : std_logic_vector(SLOT_CNT_SIZE - 1
    downto 0);
  signal start              : boolean;

  -- Time to swap schedule
  signal moment : std_logic_vector(PERIOD_CNT_SIZE - 1 downto 0);

  -- Slot counter
  signal slt_cnt        : unsigned(SLOT_CNT_SIZE - 1 downto 0);
  signal next_slt_cnt   : unsigned(SLOT_CNT_SIZE - 1 downto 0);
  signal enable_slt_cnt : boolean;
  signal max_slt_cnt    : unsigned(SLOT_CNT_SIZE - 1 downto 0);

  -- Schedule period counter one-hot encoded
  signal period_cnt        : std_logic_vector(PERIOD_CNT_SIZE - 1
    downto 0);
  signal enable_period_cnt : boolean;

  -- Swap control signals               : boolean;
  signal assert_swap, disassert_swap : boolean;

  -- Push schedule state machine
  type states is (idle, init, push_lead_in, push, wait_disassert);
  signal state, next_state : states;
  -- Regulate OCP-IO state machine
  type ocp_io_states is (idle, write_done);
  signal io_state, next_io_state : ocp_io_states;
  -- Slot and period counter state machine
  type cnt_states is (s1, s2, s3);
  signal cnt_state, next_cnt_state : cnt_states;

begin
  push_schedule_f.data_channel.req    <= push_handshake;
  push_schedule_f.data_channel.id     <= node_id;
  push_schedule_f.swap_channel.req    <= swap_2ph;
  push_schedule_f.swap_channel.moment <= moment;

  -- OCP-IO state machine. When the processor asks to read the
  -- pre-specified address ST_MASK the state (occupied or not)
  -- of the module is returned. When the same address is written
  -- by the processor, it is interpreted as an "apply the new
  -- schedule" command. The location of the schedule in the
  -- mode change memory and the size of the new schedule are
  -- given as data of the OCP-IO write command.
  ocp_io_fsm : process(io_state, proc_in, occupied)
```

```
    variable ocp_read, ocp_write           : boolean;
    variable accepted_resp, valid_command : boolean;
  begin
    -- Variables
    ocp_read              := proc_in.MCmd = "010";
    ocp_write             := proc_in.MCmd = "001";
    accepted_resp         := proc_in.MRespAccept = '1';
    valid_command         := proc_in.MAddr(OCP_ADDR_WIDTH - 1 downto
    OCP_ADDR_WIDTH - ADDR_MASK_W) = ST_MASK;
    -- Defaults
    next_io_state         <= io_state;
    proc_out.SData        <= (others => '0');
    proc_out.SResp        <= OCP_RESP_NULL;
    proc_out.SCmdAccept   <= '0';
    start                 <= false;
    case io_state is
      when idle =>
        if ocp_write then
          if valid_command then
            start <= true;
          end if;
          next_io_state <= write_done;
        end if;
        if ocp_read then
          proc_out.SCmdAccept <= '1';
          proc_out.SData(0)   <= occupied;
          if valid_command then
            proc_out.SResp <= OCP_RESP_DVA;
          else
            proc_out.SResp <= OCP_RESP_ERR;
          end if;
        end if;
      when write_done =>
        proc_out.SCmdAccept <= '1';
        if valid_command then
          proc_out.SResp <= OCP_RESP_DVA;
        else
          proc_out.SResp <= OCP_RESP_ERR;
        end if;
        if accepted_resp then
          next_io_state <= idle;
        end if;
    end case;
  end process ocp_io_fsm;

  -- The module maintains a slot counter and a schedule period
  -- counter running mesochronously in parallel to the network
  -- adapters' counters.
  cnt_fsm : process(cnt_state, max_slt_cnt, slt_cnt)
    variable last_slot : boolean;
  begin
    --Variables
    last_slot           := max_slt_cnt = slt_cnt;
    --Defaults
    next_slt_cnt        <= slt_cnt + 1;
    enable_slt_cnt    <= false;
    enable_period_cnt <= false;

    case cnt_state is
      when s1 =>
        next_cnt_state <= s2;
      when s2 =>
        next_cnt_state <= s3;
      when s3 =>
        next_cnt_state <= s1;
        enable_slt_cnt <= true;
        if last_slot then
          next_slt_cnt      <= (others => '0');
          enable_period_cnt <= true;
        end if;
    end case;
  end process;

  -- Protect addra from overflowing.
  spm_out.address <= std_logic_vector(location) when location <
    MC_SPM_SIZE else (others => '0');

  -- State machine that regulates the pushing of a new schedule,
  -- waits synchronization with the period counter, asserts a swap
  -- schedule state and waits until the schedule is applied and
  -- the swap state is disasserted.
  fsm : process(state, start, schedule_location, schedule_max_index
    , location, index_cnt, node_id, enable_period_cnt,
    disassert_swap, spm_in.data, moment, period_cnt)
    variable node_last_slot, last_node, time_to_swap : boolean;
  begin
    -- Variables
    node_last_slot := index_cnt = unsigned(schedule_max_index);
    last_node      := node_id = NODES - 1;
    time_to_swap   := moment = period_cnt and enable_period_cnt;

    -- Defaults
    next_state            <= state;
    push_data             <= spm_in.data;
    next_location         <= location;
    next_index_cnt        <= index_cnt;
    next_node_id          <= node_id;
    enable_push_handshake <= false;
    toggle_occupied       <= false;
    assert_swap           <= false;
    disassert_swap        <= false;
    case state is
      when idle =>
        if start then
          next_state      <= init;
          toggle_occupied <= true;
          next_location   <= unsigned(schedule_location);
        end if;
      when init =>
        next_node_id          <= (others => '0');
```

```vhdl
        enable_push_handshake  <= true;
        next_state             <= push_lead_in;
      when push_lead_in =>
        --data for pending handshake
        push_data                                <= (others => '0');
        push_data(SLOT_CNT_SIZE - 1 downto 0)   <= schedule_max_index
    ;
        next_index_cnt                           <= (others => '0');
        next_location                            <= location + 1;
        enable_push_handshake                    <= true;
        next_state                               <= push;
      when push =>
        if node_last_slot then
          if last_node then
            next_state   <= wait_disassert;
            assert_swap  <= true;
          else
            enable_push_handshake  <= true;
            next_node_id           <= node_id + 1;
            next_state             <= push_lead_in;
          end if;
        else
          enable_push_handshake  <= true;
          next_location          <= location + 1;
          next_index_cnt         <= index_cnt + 1;
        end if;
      when wait_disassert =>
        if time_to_swap then
          disassert_swap   <= true;
          next_state       <= idle;
          toggle_occupied  <= true;
        end if;
    end case;
  end process fsm;

  registers : process(reset, clk)
  begin
    if reset = '1' then
      push_handshake  <= '0';
      index_cnt       <= (others => '0');
      location        <= to_unsigned(33, location'length);
      node_id         <= (others => '0');
      state           <= idle;
      slt_cnt         <= (others => '0');
      max_slt_cnt     <= to_unsigned(BOOT_SCHEDULE_SIZE - 1,
    max_slt_cnt'length);
      for I in 1 to PERIOD_CNT_SIZE - 1 loop
        period_cnt(I)  <= '0';
      end loop;
      period_cnt(0)       <= '1';
      cnt_state           <= s1;
      moment              <= (others => '0');
      swap_2ph            <= '1';
      occupied            <= '0';
      schedule_max_index  <= (others => '0');
      io_state           <= idle;

    elsif rising_edge(clk) then
      index_cnt <= next_index_cnt;
      location  <= next_location;
      state     <= next_state;
      node_id   <= next_node_id;
      cnt_state <= next_cnt_state;
      io_state  <= next_io_state;
      if disassert_swap then
        max_slt_cnt <= unsigned(schedule_max_index);
      end if;
      if enable_push_handshake then
        push_handshake <= not push_handshake;
      end if;
      if enable_slt_cnt then
        slt_cnt <= next_slt_cnt;
      end if;
      if assert_swap then
        for I in 0 to PERIOD_CNT_SIZE - 2 loop
          moment(I) <= period_cnt(I + 1);
        end loop;
        moment(PERIOD_CNT_SIZE - 1) <= period_cnt(0);
        swap_2ph                    <= not swap_2ph;
      end if;
      if enable_period_cnt then
        for I in 1 to PERIOD_CNT_SIZE - 1 loop
          period_cnt(I) <= period_cnt(I - 1);
        end loop;
        period_cnt(0) <= period_cnt(PERIOD_CNT_SIZE - 1);
      end if;
      if toggle_occupied then
        occupied <= not occupied;
      end if;
      if start then
        schedule_max_index <= proc_in.MData(SLOT_CNT_SIZE - 1 downto 0);
      end if;
    end if;
  end process registers;

end rtl;
```

Listing 3: File: mc_broadcast_tree_node.vhd

```vhdl
----------------------------------------------------------------------
-- Mode change broadcast tree node
--
-- It consists of two forks. The fork size is given as a generic.
-- Data channel fork:
--    Asynchronous buffered fork, implemented with click element
--    template.
-- Swap channel fork:
--    Just wiring propagating the input to all outputs. No
--    acknowledgment used. Mono-directional channel. No buffering.
--
-- Author: Ioannis Kotleas
----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use work.mc_defs.all;

entity mc_broadcast_tree_node is
  generic(SIZE              : integer;
          MATCHED_DELAY_REQ : natural);
  port(
    reset : std_logic;
    in_f  : in  mc_channels_f;
    in_b  : out mc_channels_b;
    out_f : out mc_array_f(SIZE - 1 downto 0);
    out_b : in  mc_array_b(SIZE - 1 downto 0)
  );
end mc_broadcast_tree_node;

architecture rtl of mc_broadcast_tree_node is
  -- 1 to SIZE fork  - handshake and data signals
  -- Control
  signal data_click : std_logic;
  signal data_state : std_logic;
  signal data_acks  : std_logic_vector(SIZE - 1 downto 0);
  signal del_req    : std_logic;
  -- Data
  signal id         : node_id_t;
  signal data       : mc_word_t;

begin

  -- Connect all inputs and outputs
  in_b.data_channel.ack <= data_state;

  out_req : for J in 0 to SIZE - 1 generate
    out_f(J).data_channel.req     <= data_state;
    out_f(J).data_channel.data    <= data;
    out_f(J).data_channel.id      <= id;
    out_f(j).swap_channel.req     <= in_f.swap_channel.req;
    out_f(j).swap_channel.moment  <= in_f.swap_channel.moment;
    data_acks(J)                  <= out_b(J).data_channel.ack;
  end generate;

  -- Matched delay for setup time violations
  delay : entity work.matched_delay
    generic map(MATCHED_DELAY_REQ)
    port map(in_f.data_channel.req, del_req);

  -- Click generating process
  data_click_proc : process(del_req, data_acks, data_state)
    variable set : boolean;
  begin
    set := data_state /= del_req;
    for I in 0 to SIZE - 1 loop
      set := set and data_state = data_acks(I);
    end loop;
    if set then
      data_click <= '1';
    else
      data_click <= '0';
    end if;
  end process;

  -- Registers
  data_reg : process(reset, data_click)
  begin
    if reset = '1' then
      data_state <= '0';
      id         <= (others => '0');
      data       <= (others => '0');
    elsif rising_edge(data_click) then
      data_state <= not data_state after 2 ns;
      id         <= in_f.data_channel.id;
      data       <= in_f.data_channel.data;
    end if;
  end process;

end rtl;
```

Listing 4: File: mc_broadcast_tree_level.vhd

```vhdl
------------------------------------------------------------------
-- Mode change broadcast tree level
--
-- It is a group of nodes. It supports 2 kinds of nodes, 1to3 nodes
-- and 1to2 nodes. The nodes of a level stand in parallel and have
-- no connections to one another. The level has as many inputs
-- (input array size) as the amount of nodes and as many outputs
-- (output array size) as the sum of the outputs of the nodes.
--
-- Click element pipeline stage template used
--
-- Author: Ioannis Kotleas
------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use work.mc_defs.all;

entity mc_broadcast_tree_level is
  generic(THREES             : integer;
          TWOS               : integer;
          MATCHED_DELAY_REQ  : natural);
  port(
    reset : in  std_logic;
    in_f  : in  mc_array_f(THREES + TWOS - 1 downto 0);
    in_b  : out mc_array_b(THREES + TWOS - 1 downto 0);
    out_f : out mc_array_f(THREES * 3 + TWOS * 2 - 1 downto 0);
    out_b : in  mc_array_b(THREES * 3 + TWOS * 2 - 1 downto 0)
  );
end mc_broadcast_tree_level;

architecture structural of mc_broadcast_tree_level is
  component mc_broadcast_tree_node is
    generic(SIZE              : integer;
            MATCHED_DELAY_REQ : natural);
    port(
      reset : std_logic;
      in_f  : in  mc_channels_f;
      in_b  : out mc_channels_b;
      out_f : out mc_array_f(SIZE - 1 downto 0);
      out_b : in  mc_array_b(SIZE - 1 downto 0)
    );
  end component;

begin
  node_1to3 : for I in 1 to THREES generate
    n3 : mc_broadcast_tree_node
      generic map(3, MATCHED_DELAY_REQ)
      port map(
        reset => reset,
        in_f  => in_f(I - 1),
        in_b  => in_b(I - 1),
        out_f => out_f(I * 3 - 1 downto (I - 1) * 3),
        out_b => out_b(I * 3 - 1 downto (I - 1) * 3)
      );
  end generate node_1to3;

  node_1to2 : for I in THREES + 1 to THREES + TWOS generate
    n2 : mc_broadcast_tree_node
      generic map(2, MATCHED_DELAY_REQ)
      port map(
        reset => reset,
        in_f  => in_f(I - 1),
        in_b  => in_b(I - 1),
        out_f => out_f(THREES * 3 + (I - THREES) * 2 - 1 downto
    THREES * 3 + (I - THREES - 1) * 2),
        out_b => out_b(THREES * 3 + (I - THREES) * 2 - 1 downto
    THREES * 3 + (I - THREES - 1) * 2)
      );
  end generate node_1to2;

end structural;
```

Listing 5: File: mc_broadcast_tree.vhd

```vhdl
-------------------------------------------------------------------
-- Broadcast tree. All paths of equal length.
-- Dynamically generated by the amount of leaves.
--
-- Author: Ioannis Kotleas
-------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use work.mc_defs.all;
use work.delays.all;

entity mc_broadcast_tree is
  generic(leaves : natural);
  port(
    reset : in  std_logic;
    in_f  : in  mc_channels_f;
    in_b  : out mc_channels_b;
    out_f : out mc_array_f(leaves - 1 downto 0);
    out_b : in  mc_array_b(leaves - 1 downto 0)
  );
end mc_broadcast_tree;

architecture structural of mc_broadcast_tree is
  -- Calculate how many levels the tree must be consisted of
  constant levels      : natural     := calc_levels(leaves);
  -- For every tree level, calculate how many forks to three
  -- and how many forks to two the level is consisted of
  constant level_sizes : tree_sizes := calc_tree_levels(leaves);

  -- A level is a group of buffered forks to three and forks to two
  component mc_broadcast_tree_level is
    generic(THREES             : integer;
            TWOS               : integer;
            MATCHED_DELAY_REQ  : natural);
    port(
      reset : in  std_logic;
      in_f  : in  mc_array_f(THREES + TWOS - 1 downto 0);
      in_b  : out mc_array_b(THREES + TWOS - 1 downto 0);
      out_f : out mc_array_f(THREES * 3 + TWOS * 2 - 1 downto 0);
      out_b : in  mc_array_b(THREES * 3 + TWOS * 2 - 1 downto 0)
    );
  end component;

  -- Dynamic wiring. Calculate the total connections among the
  -- tree levels and the schedule extractors and declare record
  -- array.
  signal interconnect_f : mc_array_f(calc_total_connections(
    level_sizes, levels) - 1 downto 0);
  signal interconnect_b : mc_array_b(calc_total_connections(
    level_sizes, levels) - 1 downto 0);

begin
  out_f                                <= interconnect_f(leaves - 1
    downto 0);
  interconnect_b(leaves - 1 downto 0) <= out_b;

  -- For every level calculate how many are the inputs and how many
  -- the outputs, perform some indexing calculations for the
  -- dynamic wiring, and instantiate the tree level with parameters
  -- the amount of forks to three and forks to two the level is
  -- consisted of.
  level : for I in 0 to levels - 1 generate
    not_top : if I < levels - 1 generate
      level : block
        constant input_connections  : integer :=
    calc_connections_of_level(level_sizes, I + 1);
        constant output_connections : integer :=
    calc_connections_of_level(level_sizes, I);
        constant already_connected  : integer :=
    calc_connections_of_previous_levels(level_sizes, I);
        signal level_in_f           : mc_array_f(input_connections
    - 1 downto 0);
        signal level_in_b           : mc_array_b(input_connections
    - 1 downto 0);
        signal level_out_f          : mc_array_f(output_connections
     - 1 downto 0);
        signal level_out_b          : mc_array_b(output_connections
     - 1 downto 0);

      begin
        interconnect_f(already_connected + output_connections - 1
    downto already_connected) <= level_out_f;

        interconnect_b(already_connected + output_connections +
    input_connections - 1 downto already_connected +
    output_connections) <= level_in_b;

        level_out_b <= interconnect_b(already_connected +
    output_connections - 1 downto already_connected);
        level_in_f  <= interconnect_f(already_connected +
    output_connections + input_connections - 1 downto
    already_connected + output_connections);

        l : mc_broadcast_tree_level
          generic map(level_sizes(I)(0), level_sizes(I)(1),
    link_req_delay)
          port map(
            reset => reset,
            in_f  => level_in_f,
            in_b  => level_in_b,
            out_f => level_out_f,
            out_b => level_out_b
          );

      end block level;
    end generate not_top;

    top : if I = levels - 1 generate
```

```vhdl
      level : block
        constant input_connections  : integer := 1;
        constant output_connections : integer := calc_connections_of_level(level_sizes, I);
        constant already_connected  : integer := calc_connections_of_previous_levels(level_sizes, I);
        signal level_in_f           : mc_array_f(input_connections - 1 downto 0);
        signal level_in_b           : mc_array_b(input_connections - 1 downto 0);
        signal level_out_f          : mc_array_f(output_connections - 1 downto 0);
        signal level_out_b          : mc_array_b(output_connections - 1 downto 0);

      begin
        interconnect_f(already_connected + output_connections - 1 downto already_connected) <= level_out_f;

        level_out_b   <= interconnect_b(already_connected + output_connections - 1 downto already_connected);
        level_in_f(0) <= in_f;
        in_b          <= level_in_b(0);

        l : mc_broadcast_tree_level
          generic map(level_sizes(I)(0), level_sizes(I)(1), link_req_delay)
          port map(
            reset => reset,
            in_f  => level_in_f,
            in_b  => level_in_b,
            out_f => level_out_f,
            out_b => level_out_b
          );

      end block level;
    end generate;
  end generate level;

end structural;
```

Listing 6: File: mc_extractor.vhd

```
-------------------------------------------------------------------
-- Asynchronous 2-phase schedule extraction module
-- Click element template used
-- Author: Ioannis Kotleas
-------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use work.mc_defs.all;
use ieee.numeric_std.all;

entity mc_extractor is
  generic(ID                : natural;
          MATCHED_REQ_DELAY : natural);
  port(
    reset : std_logic;
    -- Interface to the tree
    in_f   : in  mc_channels_f;
    in_b   : out mc_channels_b;
    -- Interface to network adapter
    out_f : out mc_to_ni;
    bank  : in  std_logic
  );
end mc_extractor;

architecture behavioural of mc_extractor is

  -- Click handshaking signals
  signal click : std_logic;
  signal state : std_logic;

  -- Matched delay signals
  signal delay_req : std_logic;

  -- Indexing signals
  signal counter, ncounter : unsigned(SLOT_CNT_SIZE - 1 downto 0);
  signal size              : unsigned(SLOT_CNT_SIZE - 1 downto 0);
  type states is (idle, extract);
  signal fsm_state, n_fsm_state : states;

  -- Register enable
  signal enable_size, enable_counter, enable_fsm : boolean;

begin
  -- Enable module when the ID matches the receptor's ID

  -- The size is not sent explicitly. The first word of a new
  -- schedule is the size. The last_slot flag is used to indicate
  -- when it is time to reset the counter and receive a new size.


  -- Backward acknowledgment to data channel. The extractor is
  -- a token consumer for the data channel.
  in_b.data_channel.ack <= state;

  -- Enhance the mono-directional swap command channel with the
  -- size of the new schedule and pass on to the network adapter
  out_f.swap_channel.req    <= in_f.swap_channel.req;
  out_f.swap_channel.size   <= size;
  out_f.swap_channel.moment <= in_f.swap_channel.moment;

  -- Slot table write interface
  -- The MSB of the write address is driven by the input bank
  -- from the network adapter. Therefore the adapter chooses
  -- the current write memory bank.

  out_f.slot_write.waddr <= bank & std_logic_vector(counter);
  out_f.slot_write.wdata <= in_f.data_channel.data;
  out_f.slot_write.wclk  <= click;

  -- Matched delay for the combinatorial of this component
  delay : entity work.matched_delay
    generic map(MATCHED_REQ_DELAY)
    port map(in_f.data_channel.req, delay_req);

  -- Click generating process
  clock : process(delay_req, state)
    variable set : boolean;
  begin
    set := state /= delay_req;
    if set then
      click <= '1';
    else
      click <= '0';
    end if;
  end process;

  fsm : process(fsm_state, in_f.data_channel.id, counter, size)
    variable enable, last_slot : boolean;
  begin
    -- Variables
    enable                := in_f.data_channel.id = ID;
    last_slot             := counter = size;
    -- Defaults
    n_fsm_state           <= idle;
    enable_size           <= false;
    enable_counter        <= false;
    enable_fsm            <= false;
    out_f.slot_write.wen  <= '0';
    ncounter              <= counter + 1;
    case fsm_state is
      when idle =>
        if enable then
          enable_size <= true;
          enable_fsm  <= true;
          n_fsm_state <= extract;
        end if;
      when extract =>
        if enable then
```

```
          out_f.slot_write.wen <= '1';
          enable_counter       <= true;
          if last_slot then
            ncounter    <= (others => '0');
            enable_fsm  <= true;
            n_fsm_state <= idle;
          end if;
        end if;
    end case;
  end process fsm;

  -- Registers with register enable
  reg : process(reset, click)
  begin
    if reset = '1' then
      state     <= '0';
      counter   <= (others => '0');
      size      <= (others => '1');
      fsm_state <= idle;
    elsif rising_edge(click) then
      state <= not state;
      if enable_fsm then
        fsm_state <= n_fsm_state;
      end if;
      if enable_size then
        size <= unsigned(in_f.data_channel.data(SLOT_CNT_SIZE - 1
    downto 0));
      end if;
      if enable_counter then
        counter <= ncounter;
      end if;
    end if;
  end process;
end behavioural;
```

Listing 7: File: mc_module.vhd

```vhdl
---------------------------------------------------------------------
-- Mode change module.
-- Dynamically generated by the amount of leaves.
--
-- Author: Ioannis Kotleas
---------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use work.mc_defs.all;
use work.ocp.all;
use work.delays.all;

entity mc_module is
  generic(leaves : natural);
  port(
    reset     : in  std_logic;
    clk       : in  std_logic;

    -- SDP Memory interface
    spm_in    : in  mc_mem_s;
    spm_out   : out mc_mem_m;

    -- Configuration OCP interface
    proc_in   : in  ocp_io_m;
    proc_out  : out ocp_io_s;

    -- Interface to NI
    ni_out    : out mc_to_ni_a(leaves - 1 downto 0);
    ni_in     : in  std_logic_vector(leaves - 1 downto 0)
  );
end mc_module;

architecture structural of mc_module is

  -- Mesochronous mode change controller.
  component mc_controller is
    port(
      clk             : in  std_logic;
      reset           : in  std_logic;
      spm_in          : in  mc_mem_s;
      spm_out         : out mc_mem_m;
      proc_in         : in  ocp_io_m;
      proc_out        : out ocp_io_s;
      push_schedule_f : out mc_channels_f;
      push_schedule_b : in  mc_channels_b
    );
  end component mc_controller;

  -- Schedule extracting module. Based on the ID it decides which
  -- data to keep and which to ignore
  component mc_extractor is
    generic(ID                : natural;
            MATCHED_REQ_DELAY : integer);
    port(
      reset : std_logic;
      in_f  : in  mc_channels_f;
      in_b  : out mc_channels_b;
      out_f : out mc_to_ni;
      bank  : in  std_logic
    );
  end component mc_extractor;

  component mc_broadcast_tree is
    generic(leaves : natural);
    port(
      reset : in  std_logic;
      in_f  : in  mc_channels_f;
      in_b  : out mc_channels_b;
      out_f : out mc_array_f(leaves - 1 downto 0);
      out_b : in  mc_array_b(leaves - 1 downto 0)
    );
  end component mc_broadcast_tree;

  signal controller_handshake_f : mc_channels_f;
  signal controller_handshake_b : mc_channels_b;
  signal tree_handshake_f       : mc_array_f(leaves - 1 downto 0);
  signal tree_handshake_b       : mc_array_b(leaves - 1 downto 0);

begin
  controller : mc_controller
    port map(
      clk             => clk,
      reset           => reset,
      spm_in          => spm_in,
      spm_out         => spm_out,
      proc_in         => proc_in,
      proc_out        => proc_out,
      push_schedule_f => controller_handshake_f,
      push_schedule_b => controller_handshake_b
    );

  tree : mc_broadcast_tree
    generic map(
      leaves => leaves
    )
    port map(
      reset => reset,
      in_f  => controller_handshake_f,
      in_b  => controller_handshake_b,
      out_f => tree_handshake_f,
      out_b => tree_handshake_b
    );

  -- Generate a schedule extractor for every leaf
  extractor : for I in 0 to leaves - 1 generate
    e : mc_extractor
      generic map(I, schedule_receptor_req_delay)
      port map(
        reset  => reset,
        in_f   => tree_handshake_f(I),
        in_b   => tree_handshake_b(I),
        out_f  => ni_out(I),
        bank   => ni_in(I)
      );
  end generate;
end structural;
```

Software support

Listing 8: File: `schedules.h`

```c
/*
  Author: Ioannis Kotleas

  Contains: The schedules to be applied
*/

#ifndef _SCHEDULES_H_
#define _SCHEDULES_H_


//The maximun schedule size supported in terms of slots
const unsigned MAX_SCHEDULE_SIZE = 256;

//The number of nodes on the NoC
const int NOC_CORES_NUM = 4;

//The size in terms of slots of the specified schedules
const int S1 = 5;
const int S2 = 4;
const int S3 = 5;
const int S4 = 3;
const int S5 = 3;
const int S6 = 3;
const int S7 = 3;
const int S8 = 3;
const int S9 = 3;
const int S10 = 3;
const int S11 = 3;
const int S12 = 3;
const int SCHEDULE_SIZES[] =
                        {S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12};

//The starting index of every schedule in the SCHEDULES array
const int SCHEDULE_LOCATIONS[] = {
  0,
  S1*NOC_CORES_NUM,
  (S1+S2)*NOC_CORES_NUM,
  (S1+S2+S3)*NOC_CORES_NUM,
  (S1+S2+S3+S4)*NOC_CORES_NUM,
  (S1+S2+S3+S4+S5)*NOC_CORES_NUM,
  (S1+S2+S3+S4+S5+S6)*NOC_CORES_NUM,
  (S1+S2+S3+S4+S5+S6+S7)*NOC_CORES_NUM,
  (S1+S2+S3+S4+S5+S6+S7+S8)*NOC_CORES_NUM,
  (S1+S2+S3+S4+S5+S6+S7+S8+S9)*NOC_CORES_NUM,
  (S1+S2+S3+S4+S5+S6+S7+S8+S9+S10)*NOC_CORES_NUM,
  (S1+S2+S3+S4+S5+S6+S7+S8+S9+S10+S11)*NOC_CORES_NUM
};

//The schedules all together in an array
const int SCHEDULES[] = {

//Schedule 1 - all to all basic
  //Node 0
```

```
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,0
    x00<<16|0x0000,
  //Node 1
0x18<<16|0x0034,0x00<<16|0x0000,0x1e<<16|0x0008,0x04<<16|0x0000,0
    x04<<16|0x0000,
  //Node 2
0x14<<16|0x0034,0x1e<<16|0x000d,0x00<<16|0x0000,0x08<<16|0x0000,0
    x08<<16|0x0000,
  //Node 3
0x00<<16|0x0000,0x1a<<16|0x000d,0x16<<16|0x0008,0x0c<<16|0x0000,0
    x0c<<16|0x0000,

//Schedule 2 - none to none
  //Node 0
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 1
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 2
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 3
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,

//Schedule 3 - alternative all2all with different routes
  //Node 0
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,0
    x00<<16|0x0000,
  //Node 1
0x18<<16|0x001e,0x00<<16|0x0000,0x1e<<16|0x0002,0x04<<16|0x0000,0
    x04<<16|0x0000,
  //Node 2
0x14<<16|0x001e,0x1e<<16|0x0007,0x00<<16|0x0000,0x08<<16|0x0000,0
    x08<<16|0x0000,
  //Node 3
0x00<<16|0x0000,0x1a<<16|0x0007,0x16<<16|0x0002,0x0c<<16|0x0000,0
    x0c<<16|0x0000,

//Schedule 4 - Node 1 to node 2
  //Node 0
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 1
0x18<<16|0x0034,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 2
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 3
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,

//Schedule 5 - Node 2 to node 3
  //Node 0
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 1
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 2
0x1e<<16|0x000d,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 3
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,

//Schedule 6 - Node 3 to node 1
  //Node 0
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 1
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 2
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 3
0x16<<16|0x0008,0x00<<16|0x0000,0x00<<16|0x0000,

//Schedule 7 - Node 1 to node 3
  //Node 0
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 1
0x1e<<16|0x0008,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 2
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 3
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,

//Schedule 8 - Node 2 to node 1
  //Node 0
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 1
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 2
0x14<<16|0x0034,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 3
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,

//Schedule 9 - Node 3 to node 2
  //Node 0
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 1
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 2
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 3
0x1a<<16|0x000d,0x00<<16|0x0000,0x00<<16|0x0000,

//Schedule 10 - Node 1 to node 2 and node 2 to node 1
  //Node 0
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 1
0x18<<16|0x0034,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 2
0x14<<16|0x0034,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 3
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,

//Schedule 11 - Node 2 to node 3 and node 3 to node 2
  //Node 0
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 1
```

```
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 2
0x1e<<16|0x000d,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 3
0x1a<<16|0x000d,0x00<<16|0x0000,0x00<<16|0x0000,

//Schedule 12 - Node 3 to node 1 and node 1 to node 3
  //Node 0
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 1
0x1e<<16|0x0008,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 2
0x00<<16|0x0000,0x00<<16|0x0000,0x00<<16|0x0000,
  //Node 3
0x16<<16|0x0008,0x00<<16|0x0000,0x00<<16|0x0000,
};

#endif /* _SCHEDULES_H_ */
```

Listing 9: File: mc.h

```c
/*
  Author: Ioannis Kotleas

  Contains: The software support of the mode change module
*/

#ifndef _MC_H_
#define _MC_H_

/*
  Apply one of the schedules defined in schedules.h.
  The schedule must be provided by its index. Therefore
  Schedule 1 has index 0, schedule 2 has index 1 and so on.
*/

void apply_schedule(unsigned schedule);

#endif /* _MC_H_ */
```

Listing 10: File: `mc.c`

```c
/*
  Author: Ioannis Kotleas

  Contains: The software support of the mode change module
*/

#include <machine/patmos.h>
#include <machine/spm.h>
#include "mc.h"
#include "schedules.h"
#include "libnoc/noc.h"

/// The address of the mode change module status
#define MC_STATUS_ADDRESS  NOC_ST_BASE
/// The base address of the MC SPM
#define MC_SPM_BASE NOC_SPM_BASE

/*
  The MC SPM is split in two areas,
  each of which can hold a full schedule of maximum size. Each
  new schedule is written to a different area than the previous.
  So, the first schedule goes to the first area, the second to the
  second, the third to the first area again and so on. This way the
  worst case time to apply a schedule is reduced, since we do not
  have to wait until the previous schedule is actually applied and
  the mode change module is free to copy a new schedule to the
  memory.
*/
unsigned active_location = 0;
const unsigned ALTERNATIVE_LOCATION = MAX_SCHEDULE_SIZE *
    NOC_CORES_NUM;

/*
  Utilize the OCP-IO port of the processor (IODEV) to read the
  status of the mode change module and start a new schedule
*/
static struct mc_interface
{
    volatile int _IODEV *status;
} mc_interface = {
    MC_STATUS_ADDRESS
};

/*
  Function that utilizes the OCP-CORE port of the processor (_SPM)
  to copy the schedule whose index is provided as argument to the
  location of the MC SPM specified by the second  argument
*/
void write_schedule(unsigned sched, unsigned location){
  unsigned total = SCHEDULE_SIZES[sched]*NOC_CORES_NUM;
  unsigned sched_loc = SCHEDULE_LOCATIONS[sched];
  _SPM int *target_loc = ((_SPM int *) MC_SPM_BASE)+location;
  for(unsigned i=0; i < total ; i++){
    *(target_loc+i) = SCHEDULES[sched_loc+i];
  }
}

/*
  Read from the OCP-IO the status of the mode change module and
  return 1 if it is free
*/
unsigned mc_is_free(){
  return *(mc_interface.status) == 0 ? 1 : 0;
}

/*
  Command the mode change module to apply the schedule that
  lies in the memory. The size of the schedule and its location
  in the memory are provided as arguments
*/
void apply_sched_command(unsigned size, unsigned location){
  *(mc_interface.status) = (location << 16) | (size-1);
}

/*
  Apply a schedule
  This function copies the selected schedule to the MC SPM,
  then waits until the module is free, commands the
  application of the new schedule and updates the active location
  to which the next schedule may be written.
*/
void apply_schedule(unsigned schedule){
  write_schedule(schedule, active_location);
  while(!mc_is_free());
  apply_sched_command(SCHEDULE_SIZES[schedule],active_location);
  active_location = (active_location==0)?ALTERNATIVE_LOCATION:0;
}
```

C applications for the test cases

Listing 11: File: app1.c

```
/*
  Author: Ioannis Kotleas

  Purpose of application: Demonstrate the effect of a schedule
                          change to the communication channels
                          independently. The correctness is
                          verified with application app2.

  Functionality: Each processor, except for processor 0, sends
                 to the other processors, except processor 0,
                 blocks repeatedly and as soon as possible,
                 without any synchronization. Similarly, the
                 processors perform non-blocking poll of the SPM
                 to receive blocks from the others. Whenever
                 a block is received, a message is printed out
                 on the serial port of processor 0.
                 The application does not verify neither the
                 contents of the received blocks, nor the
                 reception of all the incoming blocks. On the
                 contrary, it verifies that a communication
                 channel is open and blocks can be received.
*/

#include <machine/spm.h>
#include <machine/patmos.h>
#include "libnoc/noc.h"
#include "patio.h"
#include "bootable.h"
#include "mc.h"
#include "app_util.h"

const int NOC_MASTER = 0;

#define DW(X) (((X)+7)/8)
struct monitor_t {
    volatile char done[3];
    volatile char result[3];
};

#define monitor ((_UNCACHED struct monitor_t *)0x00010000)

static void master(void);
static void function(void);
int noc_send_attempt(   int dst_id,
                        volatile void _SPM *dst,
                        volatile void _SPM *src,
                        size_t len);

/*//////////////////////////////////////////////////////////////////
// Main
//////////////////////////////////////////////////////////////////*/

int main() {
  if(get_cpuid() == 0) {
    monitor->done[0]=0;
    monitor->done[1]=0;
    monitor->done[2]=0;
    master();
  } else {
    function();
  }
  return 0;
}

/*/////////////////////////////////////////////////////////////////
// Mode change function running on processor 0
/////////////////////////////////////////////////////////////////*/

static void master(void) {
  unsigned schedules_to_apply[]={4,5,6,7,8,9,10,11,12,1};
  const unsigned amount = 10;
  const unsigned steps = 20;
  while(1){
    for(int i=0;i<amount;i++){
      WRITE("Apply schedule ",15);
      print_uart_num(schedules_to_apply[i]);
      WRITE("\n",1);
      apply_schedule(schedules_to_apply[i]-1);
      //Insert delay between applying a new schedule
      for(int k=0;k<steps;k++){
        for(int j=0;j<3;j++){
          if(monitor->done[j]){
            print_uart_num(monitor->result[j]);
            WRITE("to",2);
            print_uart_num(j+1);
            WRITE("\n",1);
            monitor->done[j]=0;
          }
        }
      }
    }
  }
  return;
}

/*/////////////////////////////////////////////////////////////////
// Application process running on processors 1, 2 and 3
/////////////////////////////////////////////////////////////////*/

static void function(void) {

  const unsigned block_size = 32;
  /*Processors 1, 2 and 3 are exchanging blocks. Each processor
    sends blocks to the other two, namely processor A and
    processor B. Even though it does not serve the purpose
    of the application, two sending blocks per recipient processor
    are declared for debugging purposes*/
  volatile _SPM unsigned char *sblock_a1;
  sblock_a1 = (volatile _SPM unsigned char *) NOC_SPM_BASE;
  volatile _SPM unsigned char *sblock_a2 = sblock_a1+block_size;
  volatile _SPM unsigned char *sblock_b1 = sblock_a2+block_size;
  volatile _SPM unsigned char *sblock_b2 = sblock_b1+block_size;

  //Receiving block locations on the SPM
  volatile _SPM unsigned char *rblock_a = sblock_b2+block_size;
  volatile _SPM unsigned char *rblock_b = rblock_a+block_size;

  /*Flags to manage which block to send and what is the expected
    block per communication channel. Used for debugging.*/
  unsigned psa =0,psb=0,pra=0,prb = 0;

  //Define the recipient processors ids
  int a_id,b_id;
  a_id = (get_cpuid()==3)? 1 : get_cpuid()+1;
  b_id = (get_cpuid()==1)? 3 : get_cpuid()-1;

  /*Initialize the blocks to send with data depending on the
    recipient's id. Not used in this application to verify.
  */
  for(int i=0;i<(block_size-1);i++){
    *(sblock_a1+i) = a_id+i;
    *(sblock_a2+i) = (a_id+i)<<1;
    *(sblock_b1+i) = b_id+i;
    *(sblock_b2+i) = (b_id+i)<<1;
  }

  /*Set the last entry of the sending block to all HIGH. The
    recipient polls this entry in order to be notified of the
    reception.
  */
  *(sblock_a1+block_size-1) = 0xFF;
  *(sblock_a2+block_size-1) = 0xFF;
  *(sblock_b1+block_size-1) = 0xFF;
  *(sblock_b2+block_size-1) = 0xFF;

  while(1){

    if(noc_send_attempt(
      a_id,
      rblock_a,
      (psa==0)?sblock_a1:sblock_a2,
      block_size)
    ) psa=(psa==0)?1:0;

    if(noc_send_attempt(
      b_id,
      rblock_b,
      (psb==0)?sblock_b1:sblock_b2,
      block_size)
    ) psb=(psb==0)?1:0;

    /*The completion of a block is signaled by the last entry of
      the block being all HIGH. If the block is complete, clear the
      last entry, so that it is set HIGH again by the next
      reception and print on the serial port of processor 0 a
      message indicating the reception. The serial port is slow.
      While printing, more than one blocks may arrive in the
      meanwhile. Still, the purpose of this app is to show that the
      communication channel is open, not how many blocks arrived.
    */

    if((*(rblock_a+block_size-1))==0xFF){
      monitor->result[get_cpuid()-1] = b_id;
      monitor->done[get_cpuid()-1] = 1;
      while(monitor->done[get_cpuid()-1]){;}
      *(rblock_a+block_size-1)=0;
      pra = (pra==0)?1:0;
    }
    if((*(rblock_b+block_size-1))==0xFF){
      monitor->result[get_cpuid()-1] = a_id;
      monitor->done[get_cpuid()-1] = 1;
      while(monitor->done[get_cpuid()-1]){;}
      *(rblock_b+block_size-1)=0;
      prb = (prb==0)?1:0;
    }
  }
  return;
}

/*
  Modified noc_send function from libnoc library to perform
  non-blocking send attempt. If it succeeds to initialize the
  sending, it returns 1, otherwise 0.
*/

int noc_send_attempt(   int dst_id,
                        volatile void _SPM *dst,
                        volatile void _SPM *src,
                        size_t len) {
  unsigned wp = (char *)dst - (char *)NOC_SPM_BASE;
  unsigned rp = (char *)src - (char *)NOC_SPM_BASE;
  return noc_dma(dst_id, DW(wp), DW(rp), DW(len));
}
```

Listing 12: File: app2.c

```
/*
  Author: Ioannis Kotleas

  Purpose of application: Demonstrate transparency and correctness
                          of schedule changes

  Functionality: At first each processor, except for processor 0,
                 sends to the other processors a block. Therefore
                 each processor receives two blocks from the other
                 two processors. Then each processor re-sends the
                 blocks that it received. For example: Processor 1
                 receives block A from processor 2 and block B from
                 processor 3. Then processor 1 sends block A to
                 processor 3 and block B to processor 2. The same
                 is repeated one more time. As a result, a block
                 originating from processor 1 does the trip
                 1->2->3->1, while the second block from processor
                 1 does the trip 1->3->2->1. The same happens for
                 all the processors.
                 1->2->3->1
                 2->3->1->2
                 3->1->2->3
                 1->3->2->1
                 2->1->3->2
                 3->2->1->3
                 When the round trip finishes each processor checks
                 the received blocks against the ones sent
                 originally to verify the correctness.

                 Blocking polling is used to check if a block has
                 arrived and blocking send is used to send a block,
                 meaning that if the DMA is not available, the
                 noc_send function will halt until the DMA is
                 available.

                 In the meanwhile the schedule changes multiple
                 times. As a result the execution is blocked if
                 the applied schedule does not provide the required
                 communication channels. Nevertheless, when the
                 bandwidth is granted the execution proceeds.
*/

#include <machine/spm.h>
#include <machine/patmos.h>
#include "libnoc/noc.h"
#include "patio.h"
#include "bootable.h"
#include "mc.h"
#include "app_util.h"

static void master(void);
static void function(void);
static char proof( volatile _SPM unsigned char *a,
                        volatile _SPM unsigned char *b,
                        volatile _SPM unsigned char *c,
                        unsigned size);

struct monitor_t {
     volatile char done[3];
     volatile char result[3];
};

const int NOC_MASTER = 0;

#define monitor ((_UNCACHED struct monitor_t *)0x00010000)

/*//////////////////////////////////////////////////////////////////
// Main application
//////////////////////////////////////////////////////////////////*/

int main() {
  if(get_cpuid() == 0) {
    monitor->done[0]=0;
    monitor->done[1]=0;
    monitor->done[2]=0;
    master();
  } else {
    function();
  }
  return 0;
}

/*//////////////////////////////////////////////////////////////////
// Mode change function running on processor 0
//////////////////////////////////////////////////////////////////*/

static void master(void) {
  unsigned schedules_to_apply[]={1,4,3,2};
  const unsigned amount = 4;
  const unsigned steps = 10;
  while(1){
    for(int i=0;i<amount;i++){
      apply_schedule(schedules_to_apply[i]-1);
      WRITE("Apply schedule ",15);
      print_uart_num(schedules_to_apply[i]);
      WRITE("\n",1);

      //Insert delay between applying a new schedule

      for(int k=0;k<steps;k++){
        for(int j=0;j<3;j++){
          if(monitor->done[j]){
            WRITE("Core ",5);
            print_uart_num(j+1);
            if(monitor->result[j]){
              WRITE(" :OK\n",5);
            }
            else{
              WRITE(" :ERROR\n",8);
            }
            monitor->done[j]=0;
          }
          else{
            delay(24);
          }
        }
      }
    }
  }
  return;
}

/*//////////////////////////////////////////////////////////////////
// Application process running on processors 1, 2 and 3
//////////////////////////////////////////////////////////////////*/

static void function(void) {
  const unsigned block_size = 32;
  /*Processors 1, 2 and 3 are exchanging blocks. Each processor
    sends a block to the other two. Then it receives blocks which
    are resent. The send-receive pair is repeated three times until
    a processor receives the original block. Then it will send a
    different block. Therefore each processor has 2 original
    sending blocks to send in an interleaved manner between the
    3-step phases of the procedure*/
  volatile _SPM unsigned char *sblock1;
  sblock1 = (volatile _SPM unsigned char *) NOC_SPM_BASE;
  volatile _SPM unsigned char *sblock2 = sblock1+block_size;

  //Receiving block locations on SPM
  //3 steps * 2 processors = 6 blocks
  volatile _SPM unsigned char *rblock1 = sblock2+block_size;
  volatile _SPM unsigned char *rblock2 = rblock1+block_size;
  volatile _SPM unsigned char *rblock3 = rblock2+block_size;
  volatile _SPM unsigned char *rblock4 = rblock3+block_size;
  volatile _SPM unsigned char *rblock5 = rblock4+block_size;
  volatile _SPM unsigned char *rblock6 = rblock5+block_size;

  //The phase of the procedure defining the block to send
  unsigned phase = 0;

  //Define the recipient processors ids
  int rcv_id1,rcv_id2;
  rcv_id1 = (get_cpuid()==3)? 1 : get_cpuid()+1;
  rcv_id2 = (get_cpuid()==1)? 3 : get_cpuid()-1;

  //Boolean flag for validation result
  char valid;

  /*Initialize the blocks to send with data depending on the
    sender's id. Used to verify correctness.
  */
  for(int i=0;i<(block_size-1);i++){
    *(sblock1+i) = get_cpuid()+i;
    *(sblock2+i) = 2*(get_cpuid()+i);
  }

  /*Set the last entry of the sending block to all HIGH. The
    recipient polls this entry in order to be notified of the
    reception.*/
  *(sblock1+block_size-1) = 0xFF;
  *(sblock2+block_size-1) = 0xFF;

  while(1){

    //Step 1 - send original block to both recipients
    noc_send(rcv_id1,rblock1,phase==0?sblock1:sblock2,block_size);
    noc_send(rcv_id2,rblock2,phase==0?sblock1:sblock2,block_size);

    *(rblock3+block_size-1)=0;
    *(rblock4+block_size-1)=0;

    //Receive blocks from others
    while((*(rblock1+block_size-1))!=0xFF){;}
    while((*(rblock2+block_size-1))!=0xFF){;}

    //Step 2 - re-send the received blocks
    noc_send(rcv_id1, rblock3, rblock1, block_size);
    noc_send(rcv_id2, rblock4, rblock2, block_size);

    //Receive blocks from others
    while((*(rblock3+block_size-1))!=0xFF){;}
    while((*(rblock4+block_size-1))!=0xFF){;}

    //Step 3 - re-send the received blocks
    noc_send(rcv_id1, rblock5, rblock3, block_size);
    noc_send(rcv_id2, rblock6, rblock4, block_size);

    *(rblock1+block_size-1)=0;
    *(rblock2+block_size-1)=0;

    //Receive blocks from others
    while((*(rblock5+block_size-1))!=0xFF){;}
    while((*(rblock6+block_size-1))!=0xFF){;}

    *(rblock5+block_size-1)=0;
    *(rblock6+block_size-1)=0;
    //Both of the received blocks are expected to be the same
    //as the one sent in step 1
    valid = proof(phase==0?sblock1:sblock2,rblock5,rblock6,
    block_size-1);


    monitor->result[get_cpuid()-1] = valid;
    monitor->done[get_cpuid()-1] = 1;
    //Wait until monitoring message has been processed
    while(monitor->done[get_cpuid()-1]){;}


    //Change phase so that the data written to the SPMs is
    //different every time
    phase = (phase==0)?1:0;
  }
  return;
}

/*
  Function comparing blocks and returning the result of the
  comparison
*/
static char proof( volatile _SPM unsigned char *a,
                   volatile _SPM unsigned char *b,
                   volatile _SPM unsigned char *c,
                   unsigned size){
  char flag = 1;
  for(int i=0;i<size;i++){
    flag = ((*(a+i))==(*(b+i)) && (*(a+i))==(*(c+i)))?flag:0;
  }
  if (flag){
    return 1;
  }
  else{
    return 0;
  }
}
```

Listing 13: File: `app_util.h`

```c
/*
  Author: Ioannis Kotleas

  Contains: Common utilities for apps
*/

#ifndef _APP_UTIL_H_
#define _APP_UTIL_H_

/*
  Dummy function to print a number in [0,12] to the serial port
*/
void print_uart_num(int n);

/*
  Custom delay function, specially factored not to be optimized
  out by the compiler. Used to introduce a sufficient time interval
  before applying another schedule.
*/
void delay(int steps);

#endif //_APP_UTIL_H_
```

Listing 14: File: `app_util.c`

```c
/*
  Author: Ioannis Kotleas

  Contains: Common utilities between apps
*/

#include "patio.h"
#include "mc.h"
#include "app_util.h"

/*
  Dummy function to print a number in [0,12] to the serial port
*/
void print_uart_num(int n){
  switch(n){
    case 0:
      WRITE("0",1);
      break;
    case 1:
      WRITE("1",1);
      break;
    case 2:
      WRITE("2",1);
      break;
    case 3:
      WRITE("3",1);
      break;
    case 4:
      WRITE("4",1);
      break;
    case 5:
      WRITE("5",1);
      break;
    case 6:
      WRITE("6",1);
      break;
    case 7:
      WRITE("7",1);
      break;
    case 8:
      WRITE("8",1);
      break;
    case 9:
      WRITE("9",1);
      break;
    case 10:
      WRITE("10",2);
      break;
    case 11:
      WRITE("11",2);
      break;
    case 12:
      WRITE("12",2);
```

```
        break;
  }
}

/*
  Custom delay function, specially factored not to be optimized
  out by the compiler. Used to introduce a sufficient time interval
  before applying another schedule.
*/
void delay(int steps){
  int wait=0;
  for(int i=0;i<steps;i++){
    wait=(wait+i) >>1;
  }
  print_uart_num(wait+13);
}
```