# A Resource-Efficient Network Interface Supporting Low Latency Reconfiguration of Virtual Circuits in Time-Division Multiplexing Networks-on-Chip

Rasmus Bo Sørensen[a,*], Luca Pezzarossa[a], Martin Schoeberl[a], Jens Sparsø[a]

[a]*Department of Applied Mathematics and Computer Science*
*Technical University of Denmark*
*Kgs. Lyngby, Denmark*

**Abstract**

This paper presents a resource-efficient time-division multiplexing network interface of a network-on-chip intended for use in a multicore platform for hard real-time systems. The network-on-chip provides virtual circuits to move data between core-local on-chip memories. In such a platform, a change of the application's operating mode may require reconfiguration of virtual circuits that are setup by the network-on-chip. A unique feature of our network interface is the instantaneous reconfiguration between different time-division multiplexing schedules, containing sets of virtual circuits, without affecting virtual circuits that persist across the reconfiguration. The results show that the worst-case latency from triggering a reconfiguration until the new schedule is executing, is in the range of 300 clock cycles. Experiments show that new schedules can be transmitted from a single master to all slave nodes for a 16-core platform in between 500 and 3500 clock cycles. The results also show that the hardware cost for an FPGA implementation of our architecture is considerably smaller than other network-on-chips with similar reconfiguration functionalities, and that the worst-case time for a reconfiguration is smaller than that seen in functionally equivalent architectures.

*Corresponding author
*Email address:* `rboso@dtu.dk` (Rasmus Bo Sørensen)

*Keywords:* network-on-chip, real-time systems, reconfiguration, time-division multiplexing
*2010 MSC:* 90B18, 68M10

## 1. Introduction

Packet-switched networks-on-chip (NoCs) have become the preferred paradigm for interconnecting the many cores (processors, hardware accelerators etc.) found in today's complex application-specific multi-processor systems-on-chip [1, 2] and general-purpose chip multi-processors [3, 4].

In the multi-processor systems-on-chip domain, a significant amount of previous research has targeted the generation of application-specific NoC platforms e.g., [5, 6]. With the growing cost of developing and fabricating complex VLSI chips, application-specific platforms are only feasible for very few ultra-high-volume products. In all other cases, a cost-efficient platform must support a range of applications with related functionality. This implies that the hardware resources and the functionality they implement should be as general-purpose and generic as possible, targeting a complete application domain instead of a single application. This view is expressed in the principle *provide primitives not solutions* that is well-known and accepted in the field of computer architecture. We adopt this view, striving to avoid hardware resources for dedicated and specific functionality.

The application domain we target is real-time systems. In real-time systems, the whole architecture needs to be time-predictable to support worst-case execution time (WCET) analysis. A NoC for real-time systems needs to support guaranteed-service (GS) channels. Furthermore, many hard real-time applications have multiple modes of operation. To support applications that change between operating modes, the NoC must be able to reconfigure the virtual circuits (VCs) at run-time.

This paper proposes and evaluates a flexible and resource-efficient network interface (NI) for hard real-time systems. Our NoC implements VCs using static scheduling and time-division multiplexing (TDM). A VC provides GS channels in the form of a guaranteed minimum bandwidth and a maximum latency. Furthermore, transfer of data between an on-chip memory and the NoC is coupled with the TDM schedule so that we can give end-to-end guarantees for the movement of data from one core-local memory to another core-local memory. This architecture avoids both *physical* VC buffers in the NIs and credit-based flow control among the NIs that are found in most other NoC designs [7, 8, 9]. Moreover, the usage of TDM schedules leads to a reduced hardware complexity due to the lack of buffering in the routers and due to a static traffic arbitration.

The main contribution of the paper and a key feature of this NI is its very efficient support for mode changes. The active schedule can be switched from one TDM period to the next, without breaking the communication flow of VCs that persist across the switch. This contrasts to the Æthereal family of NoCs [10, 9], which provides similar functionality at a higher hardware cost and longer reconfiguration time.

Our NI can store multiple TDM schedules and it supports instant switching from one schedule to another, synchronously across all NIs. The last TDM period of one schedule can be followed immediately by the first TDM period of a new schedule. This allows VCs that persist across a schedule switch to be mapped to different paths, without any interference to their data flow. This again avoids the fragmentation of resources seen in the previously published solutions [10, 9], in which no changes can be made to circuits that persist across a mode change and where the set-up of a new circuit is limited to using free resources.

If the schedule tables are too small to store all necessary schedules, our NoC can transparently transmit new schedules via the standard VCs. In this way, we avoid fixed allocation of resources for schedule transmission.

The NI presented here is an extension of [11], which is part of the Argo NoC [12]. A preliminary version of the new NI was published in [13]. In the rest of the paper, we refer to the NoC that uses the new NI as the Argo 2.0 NoC. The main contributions of this paper are:

- support of instant reconfiguration of VCs;
- a more elaborate analysis of the TDM schedule distribution through the NoC;
- variable-length packets to reduce the packet header overhead, resulting in shorter schedules and/or higher bandwidth on the VCs;
- interrupt packets to support multicore operating systems;
- a more compact TDM schedule representation in the NIs, reducing the schedule memory requirements;
- analysis of the effect on the TDM period length of using GS communication for reconfiguration;
- a discussion on the scalability of the architecture.

This paper is organized into seven sections. Section 2 presents related work. Section 3 provides background on mode changes, TDM scheduling, and the Argo NoC. Section 4 presents the Argo 2.0 architecture in detail. Section 5 describes the reconfiguration method and its utilization. Section 6 evaluates the presented architecture. Section 7 concludes the paper.

## 2. Related Work

This section presents a selection of NoCs that offer GS connections and that support run-time reconfiguration of the GS provided. One approach to implementing GS connections is to use non-blocking routers in combination with mechanisms that constrain packet injection rates. These NoCs are reconfigured by resetting the parameters that regulate the packet injection rates to the new requirements.

Mango [14] uses non-blocking routers and rate-control, but only links are shared between VCs. Each end-to-end connection is allocated to a unique buffer in the output port of every router that the connection traverses and these buffers use credit-based flow control between them. The bandwidth and latency of the different connections are configured by setting priorities in the output port arbiters of the router and by bounding the injection rate at the source NI. Connections are set up and torn down by programming the crossbar switches, which is done using best effort (BE) traffic. In Mango, we can observe that the reconfiguration directly interacts with the rate control mechanism in the NIs, the crossbars, and the arbiters in the routers. In addition, the fact that GS connections are programmed using BE packets may compromise the time-predictability of performing a reconfiguration.

The NoC used in the Kalray MPPA-256 processor [15] uses flow regulation, output-buffered routers with round-robin arbitration, and no flow control. Network calculus [16] is used to determine the flow regulation parameters that constrain the packet injection rates such that buffer overflows are avoided and GS requirements are fulfilled. The Kalray NoC is configured by initializing the routing tables and injection rate limits in the NIs.

IDAMC [17] is a source-routed NoC using credit-based flow control and virtual channel input buffers together to provide GS. IDAMC provides GS connections by implementing the Back Suction scheme [18], which prioritizes non-critical traffic while the critical traffic progresses to meet the deadline.

To our knowledge, details on how reconfiguration is handled in Kalray, Mango and IDAMC have not been published. However, we can safely assume that setting up a new connection must involve the initialization and modification of flow regulation parameters, and tearing down a connection must involve draining in-flight packets from the VC buffers in the NoC.

An alternative to the usage of non-blocking routers in combination with constrained packet injection rates is VC switching implemented using static scheduling and TDM. These NoCs can be reconfigured by modifying the schedule and routing tables in the NIs and/or in the routers.

The Æthereal family of NoCs [10, 9] uses TDM and static scheduling to provide GS. The original Æthereal NoC [19] supports both GS and BE traffic. The scheduling tables are in the NIs and the routing tables are in the routers. Reconfiguration is performed by writing into these tables using BE traffic. Analogously to the Mango NoC approach, using BE traffic may compromise the time-predictability of a (re)configuration. The dAElite NoC [9] focuses on multicast and overcomes this problem by introducing a separate dedicated NoC with a tree topology for the distribution of the schedule and routing information during run-time reconfiguration.

The aelite NoC [10] only supports GS traffic and it is based on source routing. This reduces significantly the high hardware cost of distributed routing and combined support for BE and GS traffic of the original Æthereal NoC. For this NoC, the routers are simple pipelined switches and both schedule tables and routing tables are in the NIs. Reconfiguration involves sending messages across the NoC using GS connections from a reconfiguration master to the schedule and routing tables that are required to change; these GS connections are reserved for this purpose only.

The original version of the Argo NoC [12] has some functional similarity with aelite. It only supports GS traffic and it also uses a TDM router with source routing. The Argo design avoids VC buffers and the credit-based flow control that account for most of the area of the NIs of the Æthereal, aelite, and dAElite range of NoCs. The Argo NoC uses a more efficient NI [11] in which the direct memory access (DMA) controllers are integrated with the TDM scheduling. The original version of the Argo NoC does not support reconfiguration.

In all the presented NoCs that uses VC switching and TDM static scheduling, the re-mapping of VCs that persist across the reconfiguration is not supported, since the reconfiguration is done incrementally (tearing down unused circuit and setting up new ones). This can lead to sub-optimal usage of resources due to fragmentation. If re-mapping of VCs is needed, the entire application must be suspended during the reconfiguration.

This paper presents a new version of the Argo architecture that implements the same functionality as the first version of Argo, while adding instantaneous reconfiguration capabilities, including re-mapping of VCs that persist across the reconfiguration.

The above coverage of related work primarily discussed architectural and implementation aspects of reconfiguration – the focus of this paper. System-level design methodology and application examples represents other important aspects of reconfiguration. Representative work on methods and algorithms for mapping tasks belonging to different modes (use-cases) to processor cores is presented in [20, 21]. A more mathematical framework for modeling the dynamic behavior of reconfigurable NoCs is developed in [22], where it is used to formulate NoC reconfiguration as dynamic optimization problem.

## 3. Background

This section provides background on the T-CREST platform, the Argo NoC, TDM scheduling, and reconfiguration for mode changes.

### *3.1. The T-CREST Multicore Platform*

T-CREST is a multicore platform to support real-time systems [23]. The vision of T-CREST is to provide a time-predictable computer architecture to enable WCET analysis. The project includes the time-predictable processor Patmos [24]. Several processors are connected to two NoCs: (1) towards the Argo NoC [12] to support message passing between processor local scratchpad memories (SPMs) and (2) to the Bluetree memory tree [25]. This memory tree connects all processors to a real-time memory controller [26] to support time-predictable access to a shared SDRAM memory.

T-CREST includes a compiler that supports the instruction set of Patmos [27]. The compiler optimizes for the WCET and interacts AbsInt's WCET analysis tool [28], which has been extended to support Patmos as well.

The T-CREST platform has been evaluated with an avionic use case [29]. Most of the T-CREST hardware and software is open-source under the industry-friendly simplified BSD license.[1]

### 3.2. Message Passing in the Argo NoC

Argo is a packet-switched and source-routed NoC that uses static allocation of network resources through TDM to provide VCs for which communication bandwidth and latency can be guaranteed.

The NoC offers message passing communication. Technically, this is implemented using DMA controllers, one per source end of every VC. A DMA controller transfers a block of data from the local memory of a processor node into the local memory of a remote node. This functionality is similar to what is seen in many other multi-core platforms including the Cell processor [30], the CompSoC platform [31], and the Epiphany processor [32]. Argo uses a very efficient NI architecture [11] in which the DMA controllers have been integrated with the TDM mechanism in the NI. This integration avoids all the buffering and flow control that is found in most NoCs. In addition, the NI hardware is dominated by area-efficient memory structures in the form of configuration tables.

### 3.3. TDM Scheduling

A parallel application on a multicore platform can be described as a set of tasks mapped to a set of processors. The steps of mapping a real-time application onto a multi-core platform and the generation of a TDM schedule for the T-CREST platform are shown in Figure 1. A set of communicating tasks can be modeled as a *task graph* (Figure 1(a)), where the vertices represent the tasks and the edges represent the communication between them.

By assigning the tasks to the processing nodes, it is possible to derive a *core communication graph* (Figure 1(b)). The assignment of tasks to processing

[1]see `https://github.com/t-crest`

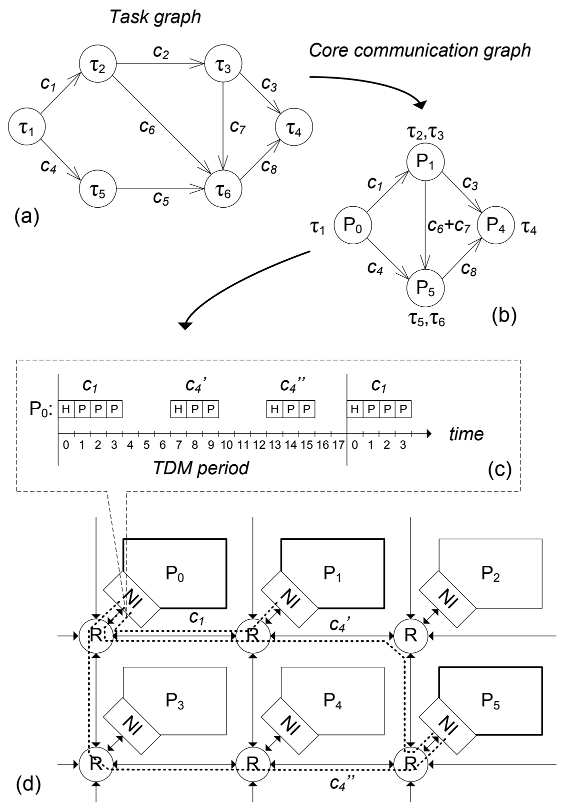

**Figure 1:** Mapping of an application onto a multi-core platform: (a) task graph for an application, (b) core communication graph, (c) TDM schedule for processor $P_0$, and (d) section of multi-core platform with possible routing for processor $P_0$.

nodes must be performed in a way that minimizes the total number of hops for traffic. For this graph, the vertices represent the processing nodes, and the edges represent the set of VCs between each pair of processing nodes.

TDM scheduling shares the resources of the NoC in time between multiple VCs. The Argo NoC uses the scheduler described in [33]. This approach divides the time into TDM periods, and a period is further divided into timeslots.

The scheduler is an off-line software tool that uses the bandwidth requirements and a description of the NoC topology to compute a schedule that avoids deadlocks and collisions, and that ensures in-order arrival of packets. The static schedule is stored in the NIs of the NoC and specifies the route of each packet and the timeslot in which each packet is injected into the router. We can calculate the minimum frequency that the schedule should run at, from the bandwidth requirements and the created schedule.

Figure 1(c) shows (part of) two TDM periods for the traffic out of the processor $P_0$ (VC *c1* and *c4*). VC *c1* has been assigned four timeslots and VC *c4* has been assigned two times three timeslots which will use two different (shortest) paths through the NoC (*c4'* and *c4"*). Figure 1(d) shows the VC paths on a section of the multi-core platform. The use of multipath routing [34], as illustrated for VC *c4*, gives the scheduler more freedom and generally results in shorter schedules. The use of shortest path routing guarantee that packets arrive in order.

The length of our TDM schedule period is typically $10-100$ clock cycles as seen, for example, in Table 2 and in [33]. This is very short compared to the periods or minimum interarrival times of tasks and it is considerably shorter than the execution time of a communicating task. Moreover, the amount of data moved in a single NoC packet is smaller than the size of the sent and received messages. In other words, the scheduler does not schedule for entire messages between tasks, but for small and frequent network packets. This leads to an efficient use of the allocated bandwidth, for both periodic and non-periodic communication flows between the tasks.

Our method allows the calculation of the maximum latency of a message. This calculation also considers the maximum waiting time for the first timeslot assigned to a specific communication channel. Due to the fine granularity of the schedule, this waiting time is negligible with respect to the sending time of the entire message. Message passing between tasks is performed at a higher level in software by using the services provided by the Argo NoC [35].

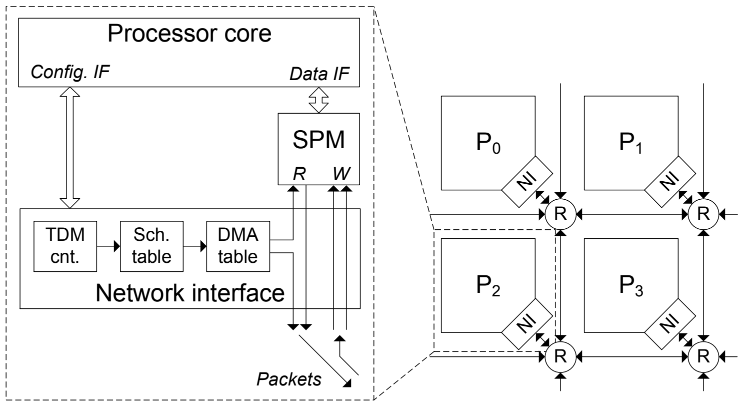

**Figure 2:** A 2-by-2 section of a multicore platform and the content of a processing tile.

## *3.4. Argo NoC Architecture*

As already mentioned, in the Argo NI architecture, the TDM-driven DMA controllers are integrated into the NI. This avoids buffering and flow control and leads to an efficient NI architecture.

Figure 2 shows a 2-by-2 section of a regular mesh topology, and the expanded tile in the figure shows the interface between the processor and the NoC as well as key elements of the NI. The processor is connected to one side of a dual-ported SPM, and the NI is connected to the other side of the SPM. The SPM populates a part of the processor's local address space and the processor sees it as a regular data SPM.

Each NI contains a *TDM counter*, that indexes into a *schedule table*, see Figure 2. The value of the TDM counter is not the current TDM slot, but it is the index of the current schedule table entry. The TDM counters in all the NIs operate synchronously and wrap around at the end of the TDM period. Each entry in the schedule table points to an entry in the *DMA table* that stores the counters and pointers corresponding to a DMA controller, and the route that a packet should follow through the network. The indexed DMA controller reads the payload data of a packet from the SPM, illustrated in Figure 2, and sends a packet. The fact that the DMA controller is activated by the TDM counter means that the DMA controller reads the data from the SPM just in time to transmit it across the network. Finally, when a packet is received at its destination NI, the payload is directly written into the SPM at the target address carried by the packet.

Like the original Argo NoC, the Argo 2.0 NoC presented in this paper does not support multicast and broadcast transmissions. This is a deliberate design decision based on the *provide primitives not solutions* principle mentioned in the Introduction. In general, hardware support for multicasting or broadcasting requires extended functionality in the routers (or even the introduction of a separate network). In addition, in a TDM-based NoC, support for multicasting or broadcasting requires scheduling slots exclusively for this purpose. Both issues represent overhead and dedication of resources to specific purposes, something we aim to avoid.

In the Argo NoC, when multicast or broadcast is needed, it is implemented by setting up dedicated VCs from the transmitter to all receivers. This avoids the cost of the dedicated resources mentioned above, and it is not as inefficient as it may first seem: the NI offers a logical DMA controller per VC, and in combination with the fine grained TDM-scheduling, all the point-to-point communications are interleaved. Thus, the latency of multiple equivalent point-to-point VCs is not necessarily longer than the latency of a multicast or broadcast.

To ensure that the collective of NIs correctly implement the global TDM schedule it is important that the TDM counters in the individual NIs operate in synchrony. In a large multi-core system-on-chip this represents a significant challenge. The Agro NOC exists in several versions including a globally synchronous version intended for FPGA prototyping and an asynchronous version that supports a globally-asynchronous locally-synchronous implementation using asynchronous routers and links, mesochronously clocked NIs and individually clocked processor cores. The former obviously requires a globally synchronous reset. The latter is preferred for an ASIC implementation and tolerate skew in both the clock signal and the reset signal. The use of asynchronous routers provides time-elasticity that allows the TDM counters in the different

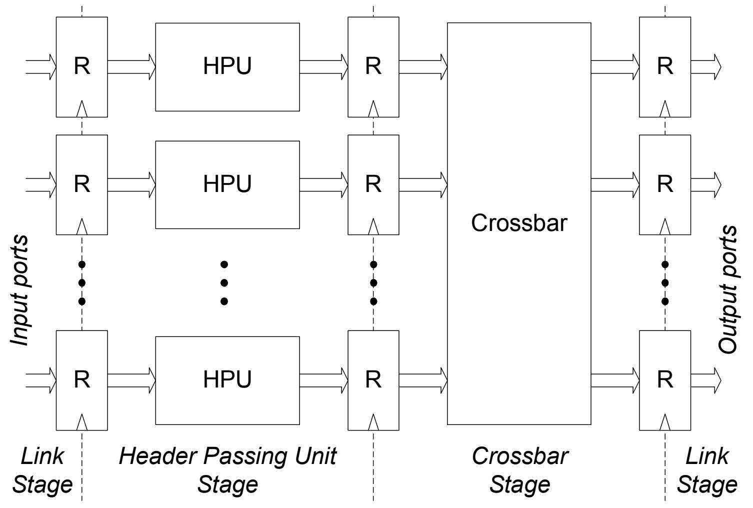

**Figure 3:** An Argo router with three pipeline stages and registers (R). The three pipeline stages are the link stage, the header passing unit (HPU) stage, and the crossbar stage.

NIs to be off by several clock cycles. More details on this aspect are found in [36, 37].

The Argo router is a pipelined crossbar that routes incoming packets according to the routing information contained in the packet header. Argo supports both synchronous and asynchronous router implementations.

In this paper, where focus is on reconfiguration and NI design, we assume for simplicity a synchronous implementation of the router as shown in Figure 3. However, the NoC is compatible with any of the Argo routers [38]. The router shown in Figure 3, consist of the three pipeline stages: link traversal, header passing unit (HPU), and crossbar. The header of an incoming packet is read in the HPU and, based on the route in the header, the packet is routed to the output port in the crossbar stage.

## *3.5. Reconfiguration for Mode Changes*

Finally, we provide some background on reconfiguration for mode changes, since this is the main contribution of this paper.

In a parallel application, a mode change is defined as a change in the subset of the executing software tasks during normal operation. Mode changes can be triggered as part of the normal operation of the system or in response to external events [39, p.340]. In normal operation, a mode change is triggered at a well-defined moment in the application execution. As a response to an external event, a mode change is triggered to adapt the system behavior to new environmental conditions. For example, an external alarm may require the execution of a set of tasks to manage specific situations.

Real-time multicore applications rely on the GS communication of the NoC to guarantee correct timing behavior. A single configuration of the time-predictable NoC may be unable to support all modes of operation for a given real-time application. Such applications need the time-predictable NoC to support reconfiguration of the VCs during run-time. This reconfiguration needs to be performed in bounded time to guarantee correct behavior of the tasks that continue operating across a mode change of the application.

We assume that each mode consists of a set of communicating tasks assigned to processors and that each mode has an associated core communication graph, from which the TDM scheduler can generate a schedule for the corresponding configuration.

Our main application domain is hard read-time systems where all tasks and their WCET and all modes of operation (combination of tasks) are known in advance. Therefore, all schedules can be computed ahead of time and offline. However, an interesting direction for future work would be to support more dynamic, soft real-time or mixed-critical systems. For such systems, it might not be known in advance which tasks shall run at the same time and which communication channels are needed. Our scheduler is a plain C++ program that does not need any special external tools. Therefore, we can envision porting that scheduler to the T-CREST platform and calculate new schedules online as needed. However, for our scheduler we cannot guarantee that a feasible schedule is found in bounded time. Therefore, this dynamic approach is only feasibly for non time-critical tasks.

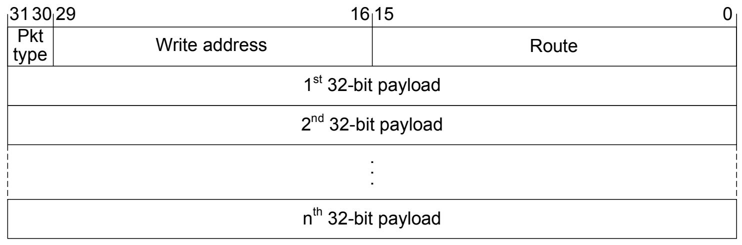

**Figure 4:** Argo 2.0 network packet format. A packet contains one 32-bit header word and $n$ 32-bit payload words. For configuration and interrupt packets, $n = 1$ and for data packets $n = 1, 2, \ldots, 15$.

## 4. Argo 2.0 Microarchitecture

Before discussing reconfiguration, we first present the basic operation and microarchitecture of the Argo 2.0 NI. Compared to the original Argo NI [11] the Argo 2.0 NI has a more elaborate microarchitecture that allows a more compact representation of a TDM schedule. The following four subsections describe the Argo 2.0 packet format, the compact schedule representation, and how the Argo 2.0 NI design transmits and receives packets.

### *4.1. Packet Format*

The microarchitecture of the Argo 2.0 NI supports three types of network packets: data packets, interrupt packets, and configuration packets. Figure 4 shows the general packet format, it contains a 32-bit header followed by $n$ 32-bit payloads. For configuration and interrupt packets, $n = 1$ and for data packets $n = 1, 2, \ldots, 15$. The variable length of data packets that allow quite long packets may be used to reduce the header overhead for VCs that require high bandwidth. The two most significant bits of the header contain the packet type. The next 14 header bits contain the write address in the target SPM where the payload data of the packet will be written. The last 16 header bits contain the route that the packet will take through the NoC.

Data packets are used to transfer regular data from the local SPM of one core to the local SPM of another core. A single DMA transfer may involve a

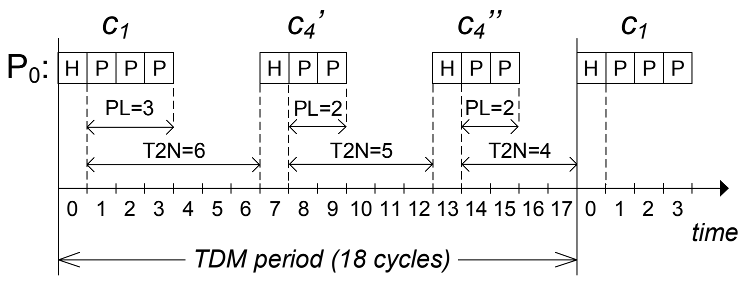

**Figure 5:** Detailed view of how the schedule in Figure 1(c)is represented in the NI. 'PL' stands for packet length.

sequence of packets sent during several consecutive TDM periods. If the sender process needs to notify the receiver when the DMA transfer is complete, the sender can mark the last packet to generate an interrupt at the destination core. We call this a local interrupt, as it is generated and processed in the processor node that receives the message.

Interrupt packets are used to generate an interrupt in a remote processor core, and this feature is needed to support multicore operating systems. When an interrupt packet arrives at the remote core it generates an interrupt. We call this a remote interrupt, as it is triggered by a remote core.

Configuration packets are used to write configuration data into the tables of a remote NI. The data of a configuration packet is written word by word into the tables of the NI.

## *4.2. Compact Schedule Representation*

The Æthereal family of NoCs and the original Argo NoC use a fixed 3-word packet format. In both designs the TDM counter is incremented once every 3 clock cycles, resulting in a 3 clock cycle slot. The TDM counter in these designs index directly into the schedule table, where unused entries are marked as *not valid*. Compared to these relative straightforward designs, the Argo 2.0 NI design represents the schedule in a more compact form. Argo 2.0 represent each packet with an entry in the schedule table and adds two fields to each entry of the schedule table. One of these fields specifies the number of payload words of the specific packet and the other specifies the time until the header of the next packet; an example of this is illustrated in Figure 5. In the example, the schedule period is 18 clock cycles and the schedule requires 3 entries in the schedule table. For comparison, we mention that the original Argo and the Æthereal family would require 6 entries in the schedule table to represent a schedule with a period of 18 cycles (6 TDM slots of 3 cycles each).

The incremental reconfiguration that is used by the Æthereal family requires this uncompressed representation, such that a scheduled packet can be written into the active schedule in one atomic write. Because the Argo 2.0 reconfiguration approach can instantaneously switch between two configurations, we can compress the schedule.

## *4.3. Transmitting Packets*

The transmit module of the NI, shown in Figure 6, consists of the following components: the TDM controller, the schedule table, the DMA table, the packet manager, and the reconfiguration controller. The reconfiguration controller is described in Section 5.

The NI contains several tables where pointers in one table index into the next. Its architecture and operation are best explained using an example. Figure 1(c) showed a schedule executed by the NI in processor node $P_0$ that has two outgoing VCs *c1* and *c4*. Figure 5 shows the same schedule in greater detail. The scheduler has assigned one 4-word packet per TDM period to *c1* and two 3-word packets to *c4*. The period of the TDM schedule in the example is 18 clock cycles, and in principle the TDM counter operates as a wrapping modulo-18 counter.

The example schedule requires three entries in the schedule table, one for *c1* and two for *c4*. Passing time in combination with the time-to-next (T2N) field controls how and when the TDM controller indexes/accesses the schedule table. In the example, the TDM controller will access the entry corresponding to *c1* in TDM cycle 0, the entry corresponding to *c4'* in cycle 7 and the entry

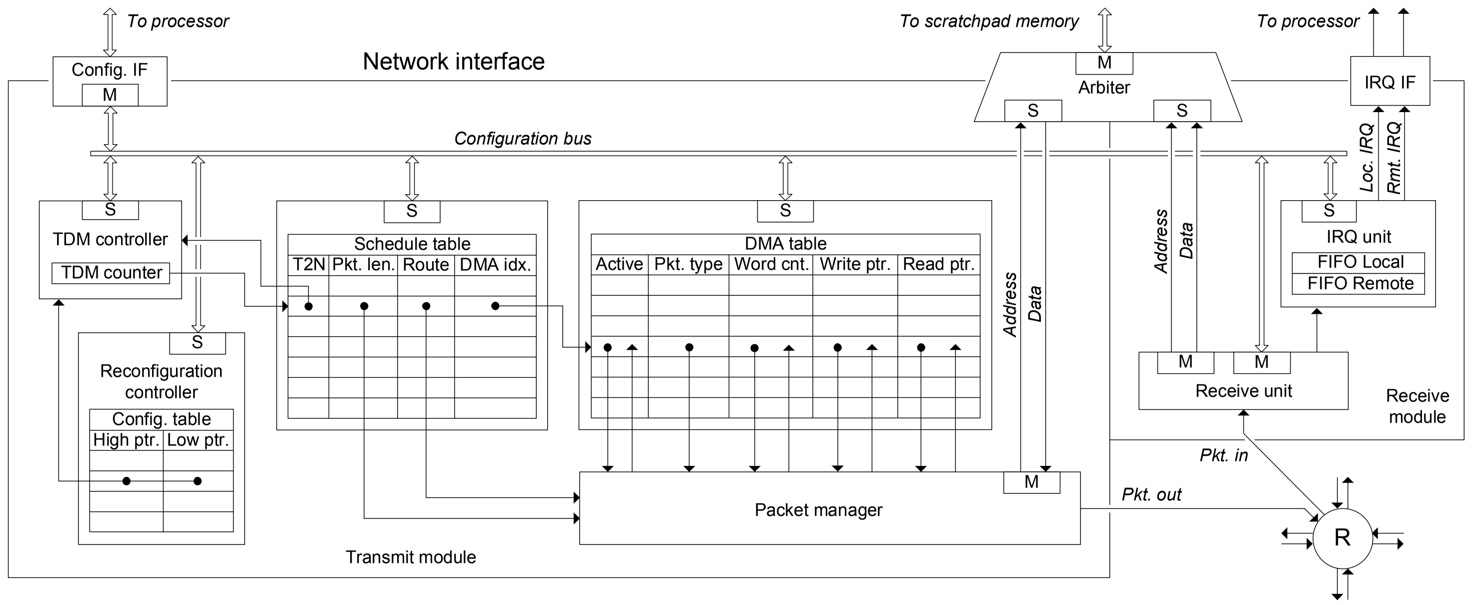

**Figure 6:** A block diagram of our NI. The block diagram is split into two parts: the transmit module and the receive module.

corresponding to *c4"* in cycle 13. As the implementation is pipelined, the table is accessed a few cycles earlier.

The schedule table can hold entries belonging to different schedules used for reconfiguration. The reconfiguration controller marks the region in the schedule table of the currently active schedule.

An entry in the schedule table contains the route of the packet that the NI is about to send. The entry also contains an index into the DMA table. Each entry in the DMA table represents a VC. Our example schedule with two VCs requires two entries in the DMA table, one for the DMA controller that pushes data across *c1* and one for the DMA controller that pushes data across *c4*. Using information from the schedule table and from the DMA table the packet manager assembles and sends out packets. The header of an outgoing packet is assembled from the packet type field of the DMA table entry, the route field of the schedule table entry and the write address field of the DMA table entry. The following payload words are read from the SPM. The packet length field *Pkt. len.* of the schedule table entry indicates the maximum number of payload words that can be appended the header word. The words are read using the read address field of the DMA table entry. During transmission of a packet the read address, the write address and the word count in the DMA table are updated.

If the data DMA transfer is marked as causing a local interrupt when it completes, the NI marks the last packet when the word count field in the DMA table entry reaches zero. When the DMA transfer is complete, the packet manager sets the active field in the DMA entry to inactive. A TDM schedule reserves slots for the different VCs and the schedule table repeatedly indexes the DMA table accordingly. If the indexed DMA table entry is inactive, no packet is transmitted in the reserved slot.

As our scheduler [33] generates schedules with shortest-path routing, all the possible paths that a packet can take through the NoC have the same number of hops. This means that packets arrive in order, and this gives the scheduler the freedom to route multiple packets belonging to the same VC along different paths. For this reason, the Argo 2.0 design places the route field into the schedule table. In the example illustrated in Figure 1 and Figure 5 this feature may be used for VC *c4* where packets *c4'* and *c4"* may be sent along different routes.

To program a TDM schedule into the NIs, information must be written into the TDM controller, the schedule table, and the reconfiguration controller of every NI. This can be done by the local processors or by a remote master processor sending out configuration packets as explained in the next subsection. The entries in the DMA table can be written and read by the local processor.

### *4.4. Receiving Packets*

The receive module shown in Figure 6 consists of two blocks: the receive unit and the interrupt (IRQ) unit. The receive unit processes incoming packets depending on the packet type. Incoming data packets carry the target address as part of the header and the data payload is written directly into the SPM as it is being received. For each packet, the receive unit increments the target address for each write into the SPM. If the data packet is the last packet of a DMA transfer, the target address of the last word is written into the IRQ FIFO for local interrupts.

If the received packet is a configuration packet, the data payload is written into one of the NI tables in the transmit module. The data structures in these blocks are mapped into a private address space of the NI and the address of the configuration packet header points into this address space.

If the received packet is an interrupt packet, the data payload is written into the SPM and the target address is written into the remote interrupt FIFO. The IRQ unit contains two FIFO queues that store interrupts. One queue is for external interrupts communicated using the interrupt packet format. The other is for local interrupts that are generated when the last packet belonging to a message is received.

The transmit and receive modules share one port to the SPM. To allow sustained and concurrent 32-bit reads and writes, the SPM uses a double width read/write port. The associated buffering and arbitration is implemented by the SPM arbiter.

The data payload of incoming packets is written directly to its target address. Therefore, there is no need for buffers or flow control in the NI, or for extra DMA controllers in the processor to copy the received data out of the NI. This makes the area of the receive module very small.

## 5. Reconfiguration

This section describes how we support mode changes by reconfiguration. Firstly, we present the underlying observations and ideas, and introduce the architectural features supporting reconfiguration. Secondly, we discuss several ways in which the reconfiguration mechanism can be used by an application requiring mode changes.

### *5.1. Key Observations and Ideas*

As we target domain-specific platforms that support a multitude of applications, our primary concern is to avoid adding resources that are specialized for one use. Therefore, we decided to use the available NoC for reconfiguration commands and transmission of schedules. This contrasts with a dedicated (re)configuration network, as for example used in dAElite [9]. Given a fixed amount of hardware resources for the NoC, a dedicated reconfiguration NoC establishes a static split of bandwidth between regular traffic and configuration traffic. We prefer to use all hardware resources to provide as much total bandwidth as possible, leaving it to the application programmer to allocate bandwidth for schedule transmission and regular traffic.

In Argo 2.0, this implies that the VCs dedicated for reconfiguration commands and possible transmission of new schedules must be set up alongside the VCs that are used for transmission of regular data; for example, a VC for reconfiguration from a master core to each of the other cores in the platform. As seen in the results section, the addition of VCs for configuration often has little impact on the TDM schedule period of an application.

As previously mentioned, Argo 2.0 does not natively support any form of multicast and broadcast transmission, since this kind of traffic can be implemented using multiple unicast transmission. The VCs added from the reconfiguration master to all the other nodes of the network and used to send the configuration commands is a clear example of this form of broadcast transmission.

An eventual support to multicast and broadcast traffic with dedicated VCs scheduled over a TDM period would not lead to any relevant speed-up in the reconfiguration time, since an entire TDM period is the minimum amount of time required to complete a broadcast transmission, which is the same as using multiple unicast VCs.

As mentioned earlier, reconfiguration of a NoC typically requires accessing and modifying the state in the NoC, as well as flushing the VCs that are torn down and some initialization of VCs that are set up. In this respect, the Argo 2.0 NI has three characteristics that both individually and in combination greatly simplify reconfiguration.

Firstly, the combination of TDM and source routing means that the routers are simple, pipelined switches, without any buffers, flow control, or arbitration. A router does not preserve any state when switching between VCs. For this reason, reconfiguration does not involve the routers; only the NIs.

Secondly, Argo 2.0 avoids VC buffers in the NIs and credit-based flow control among these buffers. Therefore, Argo 2.0 does not need to flush VCs and initialize credits counters when new connections are set up.

Thirdly, end-to-end transmission of packets, in which incoming packets are written directly into the destination SPM, in combination with the way the scheduler maps VCs to timeslots in the TDM schedule, means that the network is conceptually empty at the end of each TDM period. This opens for the very interesting perspective of instantaneously switching from one TDM schedule to another in a way that is fully transparent to VCs that persist across the reconfiguration. These circuits can even be re-mapped to different TDM slots and different (shortest path) routes. This feature avoids the fragmentation of resources that is seen in NoCs where VCs are torn down and created on an incremental basis. This ability to switch from one TDM schedule to another can be used to support reconfiguration and mode changes in several ways, as described at the end of this section.

## *5.2. Reconfiguration Controller*

To support reconfiguration, we add a reconfiguration controller to the NI and connect the receive unit to the configuration bus, as seen in Figure 6. Connecting the receive unit to the configuration bus allows the receive unit to write incoming configuration packets into all the tables connected to the configuration bus. The schedule table may hold several different schedules, each spanning a range of entries. Each range is represented by a pair of pointers, high and low, that are stored in the configuration table of the reconfiguration controller. A reconfiguration simply requires that the TDM counter is set to the start entry of the new schedule when the TDM counter reaches the end of the current schedule.

A master invokes a reconfiguration of the NoC by sending a reconfiguration packet to the reconfiguration controller of all the slave NIs, announcing that they must switch to the new schedule. This packet contains two parameters: the index of the reconfiguration table entry that holds the high and low pointers for the new schedule and the number of the TDM period after which the new schedule should be used.

The reconfiguration process is illustrated in Figure 7. When the master issues the reconfiguration command in TDM period $i$, the NI transmit configuration packets to all the slave reconfiguration controllers in period $i + 1$. Due to pipelining in the routers and in the NIs, packets sent during a TDM period arrive in a time window that is phase-shifted by some clock cycles. The time shift of the beginning of a TDM period corresponds to the pipeline depth in a shortest possible path between two NIs. In the implementation assumed in this paper this is 6 cycles. To minimize the period of a TDM schedule and avoid wasting bandwidth, the scheduler is constrained to allow a similar phase-shift

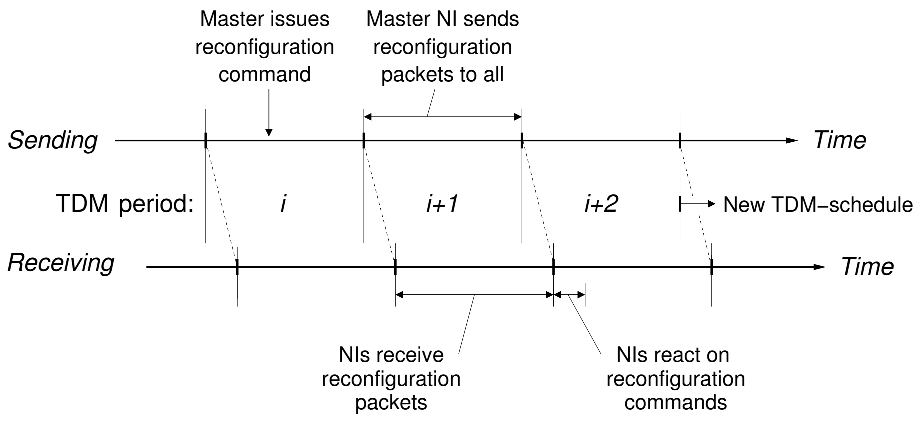
Master issues reconfiguration command
Master NI sends reconfiguration packets to all
Sending
Time
TDM period:
i
i+1
i+2
New TDM–schedule
Receiving
Time
NIs receive reconfiguration packets
NIs react on reconfiguration commands

**Figure 7:** Time diagram illustrating the reconfiguration of the NoC.

at the end of the TDM period. In this way, the flushing of the network at the end of a TDM period and the filling at the beginning of the next TDM period is overlapped. Because of this, and because an NI need two clock-cycles to process the reconfiguration command, a reconfiguration request issued by the master processor in TDM period $i$ can take effect at the earliest from TDM period $i + 3$. This argument assumes that the period of the TDM schedule is longer than the $6 + 2$ cycles. In the rare case where the schedule is shorter it can be unrolled two or more times within a TDM period.

All the tables that contain configuration data in the NI are connected to the receive unit through the configuration bus. The receive unit writes the incoming NoC configuration packets into these tables. Therefore, we can also use the NoC to transmit new schedules from the master core to the slave cores by sending the schedules using configuration packets. This transmission is transparent to the slave core.

## *5.3. Using the Reconfiguration Features*

The reconfiguration mechanism described above can be used to implement reconfiguration in several ways when an application requests a reconfiguration:

1. In cases where the schedule table and the DMA table have sufficient capacity to store all possible configurations, these can be loaded into the

NIs when the platform is booted. In this way, a master only needs to send reconfiguration requests to the NIs, and this method has the lowest reconfiguration latency.

2. Another approach is first to transmit the new schedule and then send a reconfiguration request. As seen in the next section, the time required to first distribute and then activate a new schedule is relatively short and comparable to the reconfiguration seen in other NoCs.
3. A hybrid of the above two methods is also possible, and is preferred if 1) is not possible. This hybrid divides the mode change graph into components, such that each component can be mapped into a statically sized portion of the schedule table. The mode change graph is divided by cutting either the least likely mode transitions or the mode transitions with the longest timing requirements. In this way, the reconfiguration master can switch rapidly between the schedules of one group. Switching between groups of schedules will include the transmission of the new group.
4. It is also possible to use the incremental approach [9], by tearing down and setting up individual circuits by writing into the live schedule. This approach requires a non-compacted schedule, where T2N and Pkt. len. are 3 and empty slots are represented by an invalid schedule table entry. As mentioned earlier, this method can suffer from fragmentation in the schedule tables.

Not all of these procedures are feasible for all applications, but the best solution is to use as much of the schedule table as possible. In general, this reduces the worst-case reconfiguration time.

## 6. Evaluation

This section evaluates the proposed architecture in terms of six criteria: (i) the TDM period extension due to statically allocating VCs for reconfiguration, (ii) the impact of variable-length packets on the schedule period, (iii) the storage size of the schedule in the schedule table, (iv) the worst-case reconfiguration

| Benchmark | Baseline | w/ VCs for config. | |
|---|---|---|---|
| | (cc) | (cc) | (%) |
| FFT-1024 | 63 | 78 | 24 |
| Fpppp | 120 | 120 | 0 |
| RS-dec | 90 | 92 | 2 |
| RS-enc | 84 | 86 | 2 |
| H264-720p | 90 | 92 | 2 |
| Robot | 171 | 171 | 0 |
| Sparse | 27 | 30 | 11 |
| All2all | 54 | 75 | 39 |

**Table 1:** TDM period in clock cycles (cc) and overhead (%) relative to the baseline, when VCs are used for configuration.

time, (v) the worst-case schedule transmission time, (vi) and the hardware cost and maximum operating frequency of the NI. Each criterion is evaluated in one of the following subsections. For the evaluation, we use a 4-by-4 platform with a bi-torus network and 3-stage pipelined routers. We use the MCSL benchmark suite [40] and an All2all schedule as communication patterns for the evaluation. For space reasons, we leave out the H264-1080p benchmark, as its communication pattern is identical to that of H264-720p.

## *6.1. Virtual Circuits For Configuration*

This subsection evaluates the TDM period extension due to statically allocating VCs for configuration packets in the application-specific schedules from the MCSL benchmark and an All2all schedule. In the All2all schedule, each core communicates with equal bandwidth to all other cores. For the MCSL benchmarks, the traffic patterns are mapped to the cores of an homogeneous platforms of square dimension. Then, the bandwidth requirements between the cores of the platform are extracted and used to generate the proper TDM schedule, as explained in Subsection 3.3.

Table 1 shows the TDM period of the baseline schedules without VCs for configuration and of the baseline schedules plus the added VCs for configuration. The VCs for configuration connect the master core to all slave cores in the platform.

When the traffic patterns from the MCSL benchmarks are mapped to the cores of the platform, we choose the core with the smallest outgoing bandwidth as the reconfiguration master core. We believe that this mimics a real application best, since in most cases the reconfiguration master would not have a high communication load.

A TDM schedule involving a master core that sends a single configuration packet to each slave core in the platform has a lower input/output (I/O) bound, on the TDM period, of two words per slave core. This is because the master needs to transmit for two clock cycles per slave core. For a 16 core platform, this lower I/O bound on the TDM period is 2 words per packet times 15 slave cores, in total 30 clock cycles.

The *Sparse* benchmark in Table 1 reaches this I/O bound. Table 1 shows that the TDM periods of some benchmarks are only increased by two clock cycles when VCs for configuration are added. These two slots are the two words of the configuration packet from the master to the node that has the highest incoming bandwidth requirements. The TDM periods of the *All2all* and *FFT-1024* benchmarks are increased considerably, because all cores have high outgoing bandwidths.

## *6.2. Variable-Length Packets*

This subsection evaluates the TDM period reduction of the MCSL benchmarks, when allowing variable-length packets. We let the scheduler route fewer packets with more payload words, such that the number of payload bytes during one TDM period is the same as without variable-length packets. Table 2 shows the TDM periods with VCs for configuration and packet lengths between 3 and 16 words, which is one header word plus 1 to 15 payload words.

| Benchmark | w/ VCs for config. | Var.-len. pkt. | |
|---|---|---|---|
| | (cc) | (cc) | (%) |
| FFT-1024 | 78 | 74 | 5 |
| Fpppp | 120 | 95 | 21 |
| RS-dec | 92 | 77 | 16 |
| RS-enc | 86 | 73 | 15 |
| H264-720p | 92 | 78 | 15 |
| Robot | 171 | 127 | 26 |
| Sparse | 30 | 30 | 0 |
| All2all | 75 | 75 | 0 |

**Table 2:** TDM period in clock cycles (cc) and reduction (%) relative to the schedules with VCs for configuration when variable-length packets are allowed.

In Table 2 we see two benchmarks, Fpppp and Robot, where the reduction is considerable, 21% and 26%, respectively. Considering the communication patterns of the Fpppp and Robot benchmarks, we see that they have a few cores that are involved in most of the communication. These few cores communicate through VCs that require a high bandwidth, so reducing the header overhead of these VCs causes this more than 20% reduction in TDM period. The variable-length packets reduce the TDM period of most benchmarks, except for the sparse and All2all benchmarks.

### *6.3. Schedule Storage Size*

This subsection evaluates the size of the schedules in the schedule table and the size of the DMA controllers in the DMA table. The minimum and maximum number of bytes that it takes to store the schedule in the schedule table of one node in the platform is shown in Table 3, for each MCSL benchmark and an All2all schedule. The sum of the maximum schedule table sizes of all the MCSL benchmarks and the All2all schedule is 696 bytes for one node. This is an upper bound on the schedule table size that is required to store all the schedules at

| Benchmark | Sched. tbl. (Byte) | | DMA tbl. (Byte) | | # VC | |
|---|---|---|---|---|---|---|
| | min | max | min | max | min | max |
| FFT-1024 | 52 | 108 | 74 | 152 | 13 | 27 |
| Fpppp | 56 | 108 | 74 | 147 | 13 | 26 |
| RS-dec | 24 | 76 | 23 | 102 | 4 | 18 |
| RS-enc | 20 | 68 | 6 | 90 | 1 | 16 |
| H264-720p | 20 | 72 | 6 | 90 | 1 | 16 |
| Robot | 32 | 84 | 6 | 85 | 1 | 15 |
| Sparse | 8 | 60 | 6 | 85 | 1 | 15 |
| All2all | 60 | 120 | 85 | 169 | 15 | 30 |

**Table 3:** The minimum and maximum number of bytes of storage in the schedule table and in the DMA table of one node, and the minimum and maximum number of outgoing VCs in one node.

the same time. For further studies, we assume that a schedule table of 1 KB is enough to keep all the schedules of most applications in the schedule table at the same time, avoiding the need to transmit a new schedule from the master core to all the slave cores through the NoC.

The minimum and maximum number of bytes that are required in the DMA table of one node in the platform to execute the DMA controllers are shown in Table 3. The required number of bytes in the DMA table is the number of VCs multiplied by the width of the DMA table. The width of the DMA table mainly depends on the read pointer, the write pointer and the word count. The numbers that are shown in Table 3 are for a case where 14 bits are used for the three fields, which enables an SPM of 64 KB to be used, and this is also what the current packet format supports. Each entry in the DMA table represent an outgoing VC. Therefore, the memory requirements for the DMA tables of

| Benchmark | Original (entries) | Argo 2.0 (entries) | Reduction (%) |
|---|---|---|---|
| FFT-1024 | 21 | 15 | 28.6 |
| Fpppp | 40 | 16 | 60.0 |
| RS-dec | 30 | 8 | 73.3 |
| RS-enc | 28 | 6 | 78.6 |
| H264-720p | 30 | 7 | 76.7 |
| Robot | 57 | 10 | 82.5 |
| Sparse | 9 | 4 | 55.6 |
| All2all | 18 | 16 | 11.1 |

**Table 4:** The number of schedule table entries in each core of the original Argo version (Original.) and the average number of entries in Argo 2.0.

each schedule can be overlapped by the VCs that persist across reconfigurations between the schedules of an application.

In the original version of Argo and in Argo 2.0, the number of entries in the DMA tables of each node is the same, since it is determined by the application. Therefore, we only compare the number of schedule table entries in each node of the original version of Argo against the number of entries in Argo 2.0, in table 4. The average reduction in the schedule table entries of each node is 58 %, this improvement is due to the new and more efficient architecture of the Argo 2.0 NI.

## *6.4. Worst-case Reconfiguration Time*

This subsection gives an overview of how to calculate the worst-case reconfiguration time $T_{\text{recon}}$ of a new schedule $C_{\text{new}}$. $T_{\text{recon}}$ depends only on the currently executing schedule $C_{\text{curr}}$. From Figure 7 we see that:

$$T_{\text{recon}} = 3 \cdot P_{\text{curr}} \qquad (1)$$

| $C_{\text{curr}}$ | $T_{\text{recon}}$ |
|---|---|
| FFT-1024 | 222 |
| Fpppp | 285 |
| RS-dec | 231 |
| RS-enc | 219 |
| H264-720p | 234 |
| Robot | 381 |
| Sparse | 90 |
| All2all | 225 |

**Table 5:** The worst-case reconfiguration time $T_{\text{recon}}$, starting from each benchmark as the current schedule.

Where, $P_{\text{curr}}$ is the TDM period of $C_{\text{curr}}$. We calculate $T_{\text{recon}}$ of the MCSL benchmark and an All2all schedule, shown in Table 5. An application programmer needs to add the software overhead of setting up DMA transfers and triggering a reconfiguration request to the numbers in the table.

For the benchmarks presented in Table 5, the $T_{\text{recon}}$ is between 135 and 381 clock cycles, depending on the current benchmark. For the maximum number of entries shown in Table 3 and assuming a schedule table of 256 entries (1 KB), it is possible to store of the order of 10 schedules. We believe that in most cases this schedule table size is sufficient to store the schedules associated with all the modes of an application.

The reconfiguration mechanism in Æthereal and dAElite are different from our approach. With both NoCs, VCs are teared down and setup individually. This requires 246 and 60 clock cycles respectively [9]. With Argo 2.0 we can switch the complete schedule, which is between 135 and 381 clock cycles with our benchmarks. To compare those numbers let us assume we want to switch from benchmark RS-dec to benchmark RS-enc. RS-dec has a maximum of 18 outgoing VCs and RS-enc a maximum of 16 outgoing VCs per node. Let us assume that 2/3 (12) of the VCs stay the same in the two modes. Therefore, we need to tear down 6 VCs and setup 4 VCs. On dAElite this would require $(6 + 4) \cdot 60 = 600$ clock cycles. In Argo 2.0 this reconfiguration from RS-dec to RS-enc needs 279 clock cycles. If any two modes differ by more than a handful of VCs, the reconfiguration time of Argo 2.0 is in general shorter than the reconfiguration time of Æthereal and comparable to or less than that of dAElite.

### *6.5. Worst-case Schedule Transmission Time*

This subsection gives an overview of how to calculate the schedule transmission time $T_{\text{st}}$ of a new schedule $C_{\text{new}}$ that is not stored in the schedule tables of the slave processors. The $T_{\text{st}}$ of $C_{\text{new}}$ depends on the currently executing schedule $C_{\text{curr}}$. The worst-case analysis of software depends on the processor that executes the software. Therefore, we do not include the software overhead of setting up DMA transfers in the $T_{\text{st}}$. We assume that $C_{\text{new}}$ is loaded in the processor local SPM of the reconfiguration master.

A NoC schedule is different in each NI and with the compact schedule representation that we evaluated in subsection 6.3, the schedules for each NI might be of different sizes. The $T_{\text{st}}$ of transferring $C_{\text{new}}$ to the slave NIs is the maximum of the individual worst-case latencies for each slave NI. We calculate the $T_{\text{st}}$ as:

$$T_{\text{st}} = \max_{i \in N} \left( L^i_{\text{curr}} + \left\lceil \frac{S^i_{\text{new}} - P^i_{\text{curr}}}{B^i_{\text{curr}}} \right\rceil \cdot L_{\text{curr}} + L^i_{\text{chan}} \right) \tag{2}$$

Here $i$ is the slave NI from the set $N$ of nodes in the platform, $L^i_{\text{curr}}$ is the worst-case latency of waiting for a time slot to slave $i$, $S^i_{\text{new}}$ is the number of words of $C_{\text{new}}$ to be sent to slave $i$, $P^i_{\text{curr}}$ is the number of words that can be sent in one packet towards slave $i$ in $C_{\text{curr}}$, $B^i_{\text{curr}}$ is the bandwidth of $C_{\text{curr}}$ towards slave $i$, $L_{\text{curr}}$ is the TDM period of $C_{\text{curr}}$, and $L^i_{\text{chan}}$ is the NoC latency in clock cycles to slave $i$.

We apply (2) to calculate the worst-case schedule transmission time between the schedules of the MCSL benchmark and an All2all schedule, shown in Table 6.

| $C_{\text{curr}} \backslash C_{\text{new}}$ | FFT-1024 | Fpppp | RS-dec | RS-enc | H264-720p | Robot | Sparse | All2all |
|---|---|---|---|---|---|---|---|---|
| FFT-1024 | – | 2010 | 1418 | 1270 | 1341 | 1560 | 1122 | 2229 |
| Fpppp | 2577 | – | 1814 | 1624 | 1716 | 2004 | 579 | 2862 |
| RS-dec | 2091 | 2088 | – | 1318 | 1398 | 1626 | 1164 | 2316 |
| RS-enc | 1983 | 1980 | 1396 | – | 1326 | 1548 | 1104 | 2196 |
| H264-720p | 2115 | 2112 | 1494 | 1338 | – | 1644 | 1176 | 2355 |
| Robot | 3435 | 3438 | 2422 | 2174 | 2292 | – | 1914 | 3822 |
| Sparse | 822 | 549 | 579 | 519 | 546 | 639 | – | 912 |
| All2all | 2034 | 2037 | 1431 | 1281 | 1365 | 1587 | 1137 | – |

**Table 6:** Worst-case schedule transmission time of a new schedule $C_{\text{new}}$ expressed in clock cycles. Since this depends on both the current schedule $C_{\text{curr}}$ and the new one $C_{\text{new}}$, we show a matrix of the combination of current and new schedules.

We see that the $T_{\text{st}}$ in Table 6 is between 519 and 3822 clock cycles. The Sparse benchmark, as $C_{curr}$, results in the lowest $T_{\text{st}}$, as Sparse has the shortest TDM period, and thus the highest bandwidth to the slaves.

In the rare case that a schedule needs to be transmitted to the slave NIs, our approach is still comparable. The maximum schedule transmission time in Table 6 is 3822 clock cycles. For Argo 2.0, this transmission represents the transmission of 255 VCs. In this time interval, Æthereal and dAElite can only set-up 16 and 64 VCs, respectively, this does not include tearing-down VCs.

## *6.6. Hardware Results*

This subsection presents the evaluation of the Argo 2.0 FPGA implementation presented here with respect to hardware size and maximum operating frequency. All the results presented in this section were produced using Xilinx ISE Design Suite (version 14.7) and targeting the Xilinx Virtex-6 FPGA (model XC6VLX240T-1FFG1156). All the synthesis properties were set to their defaults, except for the synthesis optimization goal which were set to area or

|  | Optimized for area |  |  | Optimized for speed |  |  |  |
|---|---|---|---|---|---|---|---|
|  | aelite | dAElite | Argo 2.0 | aelite | dAElite | Argo 2.0 | IDAMC |
| Slots | 8 | 8 | 8 | 8 | 8 | 8 | N/A |
| Conn | 1 | 1 | 2 | 1 | 1 | 2 | 8 |
| LUTs | 1916 | 2506 | 1185 | 2351 | 3117 | 1342 | 9160 |
| FFs | 3861 | 3081 | 1021 | 3960 | 3243 | 1047 | 5462 |
| BRAM | 0 | 0 | 0 | 0 | 0 | 0 | 7 |
| $f_{max}$ (MHz) | 119 | 122 | 125 | 200 | 201 | 204 | N/A |

**Table 7:** Hardware resources and maximum operating frequency of the Argo 2.0 architecture presented here and three similar designs for one 3-port router and one NI.

speed. The results are expressed in terms of numbers of flip-flops (FFs), 6-input look-up tables (LUTs), and block RAMs (BRAMs).

Table 7 shows the comparison of the Argo 2.0 implementation to the TDM-based NoCs aelite and dAElite [9], and to the IDAMC [17] NoC that uses a classic router designed with virtual channel buffers and flow control. The table shows the results of the four designs for one router and one NI and the number of supported TDM slots and connections per node. The published numbers we compare against are available for comparison for a 2-by-2 platform with mesh topologies. From these results, we derived the hardware consumption of one 3-ported router and one NI.

The results in Table 7 show that overall the Argo 2.0 NoC implementation is smaller than the other NoCs. The results also show that the numbers for the IDAMC are much higher that the aelite, dAElite and Argo 2.0. This is due to its use of virtual channels buffers and the flow control mechanisms.

The results in Table 7 shows that the maximum frequency $f_{max}$ of the Argo 2.0 implementation is comparable to the ones of aelite and dAElite for a 3-port router. The $f_{max}$ results for the IDAMC NoC are not available for comparison.

|  | Opt. for area |  | Opt. for speed |  |
|---|---|---|---|---|
|  | Argo | Argo 2.0 | Argo | Argo 2.0 |
| LUTs | 926 | 1071 | 1155 | 1358 |
| FFs | 897 | 908 | 923 | 931 |
| BRAM | 4 | 4 | 4 | 4 |
| $f_{max}$ (MHz) | 155 | 166 | 167 | 179 |

**Table 8:** Hardware resources and maximum operating frequency of the Argo 2.0 architecture and the original Argo NoC, for one 5-port router and one NI. The implemented design has 256 slots and 64 connections.

As mentioned, the Argo 2.0 NoC is designed to be used in a domain-specific platform. Therefore, Table 8 presents numbers for a network node comprising one 5-ported router and one NI with 256 TDM slots and 64 connections, which we consider reasonable numbers for a larger platform. Moreover, it compares the Argo 2.0 NoC against the original Argo NoC to show that our extensions only add a small amount of hardware resources. We re-synthesized the original Argo implementation for the results in the table. For the results in Table 8, we used BRAM to implement the tables in the NI.

In terms of $f_{max}$, the Argo 2.0 5-port router implementation optimized for area is around 33% faster than the 3-port one, since it uses BRAM instead of distributed memory (implemented using FFs), and around 7% faster than the original Argo.

## *6.7. Scalability*

The results in the previous subsections are based on a 16-core platform. As the number of cores in the platform increases, we consider the hardware size and the TDM period to evaluate the scalability of the new reconfiguration capability of Argo 2.0. We consider the hardware size of the NoC per core and the extension of the TDM period due to statically allocating VCs for reconfiguration. As the number of cores increase, the hardware size of one NI and one router increases due to the number of required entries in the schedule table and in the DMA table.

The number of entries that are required in the schedule table is the accumulated number of entries per core. Our reconfiguration approach requires the TDM schedule to include VCs for configuration from the master to all slave cores. The lower bound on the TDM period increases linearly with the number of slave cores, due to the increasing number of VCs for configuration. The lower bound is two clock cycles per slave core, which can be represented by one schedule table entry. The aelite NoC has the same property, as it also allocates VCs for reconfiguration. The dAElite NoC does not allocate VCs for configuration in the TDM schedule, but dAElite uses a separate single-master configuration tree network that causes a minor increase in the hardware size. The single-master configuration tree network results in a fixed, i.e., less flexible, allocation of bandwidth between the configuration and data communication.

As mentioned previously, the number of active entries in the DMA table is the number of outgoing VCs from the processor node. The number of DMA table entries in the Argo 2.0 NI grows in the same way as the VC buffers with credit-based flow control grows in the aelite and dAElite NIs. The hardware size of a DMA table entry is considerably smaller than the size of a VC buffer with credit-based flow control.

## 7. Conclusion

This paper presented an area-efficient time-division multiplexing network-on-chip that supports reconfiguration for mode changes. The NoC addresses hard real-time systems and provides guaranteed-service VCs between processors. The NI provides reconfiguration capabilities of end-to-end VCs to support mode changes at the application level. For the set of benchmarks used for evaluation, we showed that the TDM period overhead of statically allocating VCs for configuration was on average 10%. Furthermore, we showed that our compact schedule representation reduces the memory requirements by more than 50% on average.

We evaluated an implementation of the proposed architecture in terms of hardware cost and worst-case reconfiguration time. The results show that the proposed architecture is less than half the size of NoCs with similar functionality and that the worst-case reconfiguration time is comparable to those NoCs. If the new schedule is already loaded in the schedule table, the worst-case reconfiguration time is significantly shorter.

*Acknowledgment*

The work presented in this paper was funded by the Danish Council for Independent Research | Technology and Production Sciences under the project RTEMP,[2] contract no. 12-127600.

*Source Access*

The presented work is open source and can be downloaded from GitHub and built under Ubuntu as described in the Patmos reference handbook [41, Chap. 6].

[1] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, N. Wehn, A 477mW NoC-based digital baseband for MIMO 4G SDR, in: Proc. IEEE Intl. Solid-State Circuits Conference (ISSCC), 2010, pp. 278–279.

[2] L. Benini, E. Flamand, D. Fuin, D. Melpignano, P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator, in: Proc. Design, Automation and Test in Europe (DATE), 2012, pp. 983–987.

[3] G. Chen, M. A. Anders, H. Kaul, S. K. Satpathy, S. K. Mathew, S. K. Hsu, A. Agarwal, R. K. Krishnamurthy, V. De, S. Borkar, A 340 mV-to-0.9

[2]`http://rtemp.compute.dtu.dk`

V 20.2 Tb/s Source-Synchronous Hybrid Packet/Circuit-Switched 16 x 16 Network-on-Chip in 22 nm Tri-Gate CMOS, IEEE Journal of Solid-State Circuits 50 (1) (2015) 59–67.

[4] B. Dupont de Dinechin, D. van Amstel, M. Poulhiès, G. Lager, Time-critical computing on a single-chip massively parallel processor, in: Proc. Design, Automation and Test in Europe (DATE), 2014, pp. 97:1–97:6.

[5] S. Murali, G. D. Micheli, A. Jalabert, L. Benini, XpipesCompiler: A tool for instantiating application specific networks-on-chip, in: Proc. Design, Automation and Test in Europe (DATE), 2004, pp. 884–889.

[6] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, G. D. Micheli, NoC synthesis flow for customized domain specific multiprocessor systems-on-chip, IEEE Trans. Parallel and Distributed Systems 16 (2) (2006) 113–129.

[7] D. Berozzi, Network interface architecture and design issues, in: G. DeMicheli, L. Benini (Eds.), Networks on Chips, Morgan Kaufmann Publishers, 2006, Ch. 6, pp. 203–284.

[8] A. Radulescu, J. Dielissen, S. Pestana, O. Gangwal, E. Rijpkema, P. Wielage, K. Goossens, An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 24 (1) (2005) 4–17.

[9] R. A. Stefan, A. Molnos, K. Goossens, dAElite: A TDM NoC Supporting QoS, Multicast, and Fast Connection Set-Up, IEEE Transactions on Computers 63 (3) (2014) 583–594.

[10] K. Goossens, A. Hansson, The aethereal network on chip after ten years: Goals, evolution, lessons, and future, in: Proc. ACM/IEEE Design Automation Conference (DAC), 2010, pp. 306 –311.

[11] J. Sparsø, E. Kasapaki, M. Schoeberl, An area-efficient network interface for a TDM-based network-on-chip, in: Proc. Design, Automation and Test in Europe (DATE), 2013, pp. 1044–1047.

[12] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. T. Müller, K. Goossens, J. Sparsø, Argo: A Real-Time Network-on-Chip Architecture with an Efficient GALS Implementation, IEEE Transactions on VLSI Systems 24 (2) (2015) 479–492.

[13] R. B. Sørensen, L. Pezzarossa, J. Sparsø, An area-efficient TDM NoC supporting reconfiguration for mode changes, in: Proc. of ACM/IEEE International Symposium on Networks-on-Chip (NOCS), 2016, pp. 1–4.

[14] T. Bjerregaard, J. Sparsø, A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip, in: Proc. Design, Automation and Test in Europe (DATE), 2005, pp. 1226–1231.

[15] B. Dupont de Dinechin, Y. Durand, D. van Amstel, A. Ghiti, Guaranteed services of the NoC of a manycore processor, in: Proc. Intl. Workshop on Network on Chip Architectures (NoCArc), ACM, New York, NY, USA, 2014, pp. 11–16.

[16] R. L. Cruz, A calculus for network delay. I. Network elements in isolation, IEEE Transactions on Information Theory 37 (1) (1991) 114–131.

[17] B. Motruk, J. Diemer, R. Buchty, R. Ernst, M. Berekovic, IDAMC: A many-core platform with run-time monitoring for mixed-criticality, in: Proc. IEEE Intl. Symposium on High-Assurance Systems Engineering (HASE), 2012, pp. 24–31.

[18] J. Diemer, R. Ernst, Back suction: Service guarantees for latency-sensitive on-chip networks, in: ACM/IEEE International Symposium on Networks-on-Chip (NOCS), 2010, pp. 155–162.

[19] K. Goossens, J. Dielissen, A. Radulescu, AEthereal network on chip: concepts, architectures, and implementations, IEEE Design Test of Computers 22 (5) (2005) 414–421.

[20] A. Hansson, M. Coenen, K. Goossens, Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip, in: Proc. Design, Automation and Test in Europe (DATE), 2009, pp. 954–959.

[21] S. Murali, M. Coenen, A. Radulescu, K. Goossens, G. D. Micheli, A methodology for mapping multiple use-cases onto networks on chips, in: Proc. Design, Automation and Test in Europe (DATE), 2006, pp. 118–123.

[22] Y. Xue, P. Bogdan, Improving NoC performance under spatio-temporal variability by runtime reconfiguration: a general mathematical framework, in: Proceedings of the 10th International Symposium on Networks-on-Chip (NOCS), 2016, pp. 1–8.

[23] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, A. Tocchi, T-CREST: Time-predictable multi-core architecture for embedded systems, Journal of Systems Architecture 61 (9) (2015) 449–471.

[24] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, T. 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, pp. 11–20.

[25] J. Garside, N. C. Audsley, Investigating shared memory tree prefetching within multimedia noc architectures, in: Memory Architecture and Organisation Workshop, 2013, pp. 1–7.

[26] M. D. Gomony, B. Akesson, K. Goossens, Architecture and optimal configuration of a real-time multi-channel memory controller, in: Proc. Design, Automation Test in Europe Conference Exhibition (DATE), 2013, pp. 1307–1312.

[27] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, G. Gebhard, The T-CREST approach of compiler and WCET-analysis integration, in: Proc. Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS), 2013, pp. 33–40.

[28] R. Heckmann, C. Ferdinand, Worst-case execution time prediction by static program analysis, Tech. rep., AbsInt Angewandte Informatik GmbH, [Online, last accessed November 2013].
URL http://www.absint.de/aiT_WCET.pdf

[29] A. Rocha, C. Silva, R. B. Sørensen, J. Sparsø, M. Schoeberl, Avionics applications on a time-predictable chip-multiprocessor, in: Proc. 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), IEEE Computer Society, 2016, pp. 777–785.

[30] M. Kistler, M. Perrone, F. Petrini, Cell multiprocessor communication network: Built for speed, IEEE Micro 26 (2006) 10–25.

[31] A. Hansson, K. Goossens, M. Bekooij, J. Huisken, CoMPSoC: A template for composable and predictable multi-processor system on chips, ACM Transactions on Design Automation of Electronic Systems (TODAES) 14 (1) (2009) 2:1–2:24.

[32] A. Olofsson, T. Nordström, Z. ul Abdin, Kickstarting high-performance energy-efficient manycore architectures with Epiphany, in: Proc. Asilomar Conference on Signals, Systems and Computers, IEEE, 2014, pp. 1719–1726.

[33] R. B. Sørensen, J. Sparsø, M. R. Pedersen, J. Højgaard, A metaheuristic scheduler for time division multiplexed networks-on-chip, in: Proc. IEEE/I-

FIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS), 2014, pp. 309–316.

[34] R. Stefan, K. Goossens, A TDM slot allocation flow based on multipath routing in NoCs, Microprocessors and Microsystems 35 (2) (2011) 130–138.

[35] R. B. Sørensen, W. Puffitsch, M. Schoeberl, J. Sparsø, Message passing on a time-predictable multicore processor, in: Proc. IEEE Symposium on Real-time Distributed Computing (ISORC), 2015, pp. 51–59.

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

[37] E. Kasapaki, J. Sparsø, The Argo NOC: Combining TDM and GALS, in: European conference on circuit theory and design (ECCTD), 2015, pp. 1–4.

[38] E. Kasapaki, J. Sparsø, R. B. Sørensen, K. Goossens, Router designs for an asynchronous time-division-multiplexed network-on-chip, in: Proc. Euromicro Conference on Digital System Design (DSD), IEEE, 2013, pp. 319–326.

[39] A. Burns, A. Wellings, Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX, Addison-Wesley, 2001.

[40] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast, Z. Wang, A NoC traffic suite based on real applications, in: Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2011, pp. 66–71.

[41] M. Schoeberl, F. Brandner, S. Hepp, W. Puffitsch, D. Prokesch, Patmos reference handbook, Tech. rep., Technical University of Denmark (2014). URL `http://patmos.compute.dtu.dk/patmos_handbook.pdf`