UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Memory Access Efficiency in Deeply Heterogeneous Systems

MASTER DEGREE FINAL WORK

Master Degree in Computer Engineering

Author: Vincenzo Scotti

Tutor: José Flich Cardo

Course 2017-2018

Resumen

Este trabajo se centra en la implementación de mecanismos de transferencia de memoria eficientes en un Sistema Heterogneo. La solución propuesta en este trabajo se desarrolla en el contexto del proyecto Europeo MANGO, que provee de una plataforma tanto Software como Hardware para el desarrollo de sistemas heterogéneos multi-acelerador, para así hacer frente a las demandas de prestaciones requeridas actualmente en sistemas HPC. Los logros aquí presentados mejoran las transferencias de memoria tanto en las aplicaciones que se ejecutan en el Servidor, como las transferencias que tienen lugar en el sistema Hardware entre los diferentes aceleradores. Los resultados obtenidos en las diferentes pruebas realizadas muestran valores cercanos a los máximos posibles ofrecidos por el Hardware utilizado. Tambieén se ha desarrollado un sistema de reserva de ancho de banda para las diferentes transferencias en curso mediante el uso de pesos, posibilitando la futura implementación de políticas de Calidad de Servicio en las transferencias de memória.

Este trabajo se ha llevado a término durante una estancia de cinco meses en la Universitat Politécnica de Valéncia. Esta estancia está vinculada a un acuerdo entre la Universitat Politécnica de Valéncia y la Universitá degli Studi di Napoli Federico II.

Palabras clave: Sistema de cómputo heterogéneos; PCI express; Calidad de servicio

Abstract

This work focuses on the implementation of efficient memory transfers in a highly heterogeneous system. The proposed solution is developed in the context of the MANGO European project, which provides a software and hardware framework to support the deployment of custom multi-accelerator systems, as demanded by the modern HPC performance requirements. The achievements here presented benefit both the memory transfer efficiency as seen by applications running on a host computer and by hardware accelerators. The measured performances show results very close to the ideal ones. Support for weighted memory transfers is also developed, allowing the future implementation of Quality of Service policies regarding memory access bandwidth.

This work has been performed during an internship of 5 months. The internship is linked to an agreement between UPV and UniNa (Universita' degli Studi di Napoli Federico II).

Key words: Heterogeneous computing system; PCI express; Quality of Service

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface.

ASIC Application Specific Integrated Circuit.

CPI Clock cycles Per Instruction.

DFG Data Flow Graph.

FF Flip-Flop.

FLOPS Floating-Point Operations Per Second.

FPGA Field Programmable Gate Array.

GPGPU General Purpose computing on GPUs.

HPC High Performance Computing.

ILP Instruction Level Parallelism.

IPC Instructions Per Clock cycle.

LUT Look-Up Table.

MIG Memory Interface Generator.

QoS Quality of Service.

CHAPTER 1
Introduction

The field of computer architecture as a whole has faced several paradigm shifts in the last decades, lead by the constant goal of achieving higher performance on specific workloads, while still meeting any possible power constraint.

For many years exploiting the Instruction Level Parallelism (ILP) has been the main area of research, with the goal of improving processors' performances. ILP focuses on trying to identify and reschedule the instructions that could be executed in parallel, without breaking the sequential semantic upon which the programmer relies. The goal is to maximize the Instructions Per Clock cycle (IPC) performance metric, or to minimize its inverse, the Clock cycles Per Instruction (CPI).

On the other hand, parallel execution of instructions cannot happen if there are dependencies between them. In particular, the process of translation of a task into a set of sequential operations can give birth to additional dependencies. These dependencies reflect the limitations of the underlying hardware (the number of processor registers, etc...) and are not correlated to the task that is being processed. An optimal execution should be constrained only by the dependencies that are intrinsic in the task to execute, and that cannot be overcome. A common way to identify these dependencies is using a Data Flow Graph (DFG), which is an abstract representation of the transformations to be applied to the input data. If the DFG's transformations are annotated with execution times, the longest path on the graph is a lower bound on the effective execution time, obtained after the mapping on a specific hardware platform.

As ILP started to show its limitations, new paradigms emerged, which try to exploit parallelism at different levels: Thread Level Parallelism and data parallelism. Thread Level Parallelism aims to keep a processor's utilization level as high as possible, by switching between multiple flows of execution, called *threads*, allowing to overcome the limitations shown by ILP on a single flow of execution. This allows to push further the IPC performances if we consider instructions coming from different threads. On the other hand, there are workloads where the same execution flow should be applied to different input data: in this case the parallelism should be exploited at the data level and not on the control flow level. Data parallelism can be easily found in some domain-specific workloads like machine learning, physics, finance, etc...

In terms of hardware support, off-the-shelf components nowadays support a mix of these approaches. Modern CPUs are made of multiple *cores*, each one being independent from the other and executing a different instruction stream. In each core, we can find multiple execution threads, which can again execute different instruction streams, and compete for the core resources. Each execution flow is then analyzed and rescheduled to exploit ILP. Data parallelism is usually exploited through the use of the so-called *vector extensions*, which provide a vector register file (opposed to a classic scalar register file) and a set of vector instructions that operate on multiple registers at once.

GPUs gained a lot of relevance in the computer architecture field. At first they were meant as mere graphical accelerators, providing specialized hardware blocks used to solve efficiently rendering problems. However, as rendering workloads expose a great amount of parallelism, the same hardware can also be used to solve efficiently many other computational problems. So the General Purpose computing on GPUs (GPGPU) paradigm started to emerge, where GPUs are used as general purpose hardware accelerators. This imposed both a software and hardware architecture shift. Beside the graphical rendering Application Programming Interface (API), modern GPU vendors provide a generic programming model, along with the required tools (compilers, loaders, etc...), to allow a programmer to launch a compute kernel on the GPU cores. The most notable examples are the CUDA library provided as a proprietary solution by NVidia, and the OpenCL API developed as an open standard by the Khronos Group. The internal architecture of GPU cores evolved more in the direction of a general purpose multi-core processor, still offering hardware support for data parallel workloads accompanied by a light control logic. This hardware support results in improved performances both under the system throughput and power consumption points of view.

On the other side of this spectrum we find the High Performance Computing (HPC) subfield. HPC applications distinguish themselves for the huge amount of computing power they need. Relevant examples are weather forecasting, molecular modeling, cryptoanalysis, etc... They are usually run on *supercomputers*, which are computers targeting the highest levels of performance. Due to the nature of their workloads, performance is measured in Floating-Point Operations Per Second (FLOPS). Supercomputers gained importance for their scientific and political importance, as a sign of the technological progress of a country. The most powerful computers are listed on the *TOP500* [1] ranking, which is published twice a year and sorts supercomputers by performance. However, building such a huge computing infrastructure can require an unsustainable amount of energy: it is of main importance to evaluate performance related to power consumption. One common metric is called *FLOPS per Watt*, and takes into account the computing performance compared to the power cost. TOP500 supercomputers are also ranked by FLOPS per Watt on the *Green500* [2] list.

To account for this aggressive performance requirements, special care must be given to provide a hardware architecture that is designed around the specific application's needs. For this reason, Field Programmable Gate Arrays (FPGAs) are getting more and more relevance as reconfigurable platforms in the HPC field. They allow to implement a fully custom hardware architecture that can be tailored around any domain-specific application, without going through the long

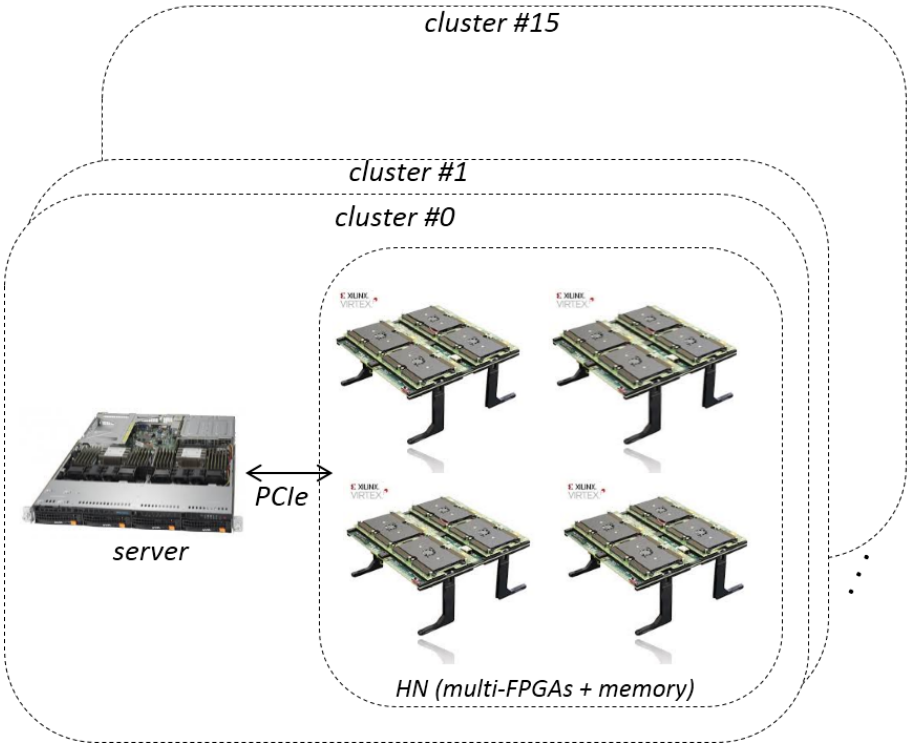


Figure 1.1: The complete MANGO cluster

design process and the costs needed for custom Application Specific Integrated Circuit (ASIC) design. As they offer great flexibility while allowing to target high throughput and power performance, they are currently being deployed in data centers and HPC scenarios as custom hardware accelerators. Most notably, the Project Catapult [4] from Microsoft is already using FPGA technology to enhanced data center computing power.

However, FPGA programming still requires a significant engineering effort. Moreover, the high level of flexibility offered doesn't allow for proper interoperability measures. The MANGO project [3] aims to tackle this issue, by providing a comprehensive software and hardware framework for the deployment of highly heterogeneous accelerators on reconfigurable hardware, while still exposing a standard interface to the hardware designer and to the application programmer.

This thesis is developed in the framework of the MANGO project.

MANGO Project

The MANGO (*exploring **M**anycore **A**rchitectures for **N**ext-**G**enerati**O**n HPC systems*) project aims to design and build a cluster of boards, each equipped with several FPGAs and memory modules, onto which generic accelerators can be implemented. User applications will then be allowed to launch a compute-intensive kernel on custom hardware, optimized for the problem at hand. This allows to target the main HPC challenge: achieving maximum performances while keeping the power consumption under control. The cluster configuration is pictured in Figure 1.1.

Figure 1.2: Physical deployment of a MANGO cluster

On the left side we have an HPC server computer. The server's responsibility is to manage the resources offered by the cluster, and to offer a clear interface to the applications that want to access those resources. In particular, a resource manager should decide how to allocate resources such as accelerators, network bandwidth and memory bandwidth, taking into account any possible Quality of Service (QoS) requirement. The physical communication with the cluster is empowered by an industry-standard high speed PCIe connection.

On the right side we can see the hardware components, by the means of a set of boards connected with each other, equipped with memory modules offering different interfaces (DDR3, DDR4, ...) and with a set of reconfigurable hardware devices, onto which accelerators can be synthesized.

As one of the goals of the project is to explore maximum customizability, each cluster configuration is identified by an *architecture ID*. This determines the type and number of accelerators used, how they are physically deployed in the system, the type of FPGAs and memory modules used, etc... A resource manager detects the configuration at runtime and acts accordingly.

The accelerators developed thus far are now briefly introduced, while a physical deployment of the cluster is reported in Figure 1.2.

PEAK

***P**artition-**E**nabled **A**rchitecture for **K**ilocores* is a RISC multi-core processor, that can be used as a general purpose computing unit inside the MANGO system. It implements a large subset of the MIPS R32 instruction set, with support for exceptions and interrupts.

The accelerator features a configurable number of cores, deployed in a mesh, where each node is called a *tile*. Each tile has also a private L1 cache bank and a shared L2 cache, which is part of a bigger distributed L2 cache [5]. A MESI-based coherence protocol is in place. Also a bank of registers is deployed in each tile, to allow for runtime configuration and statistics collection. A Network on Chip is used to implement the communication between the tiles.

Programming is supported by a complete compiler toolchain, which is based on the LLVM compiler.

nu+

nu+ is a GPU-like accelerator, featuring a multi-core processor based on a mesh topology, with support for the SIMT paradigm enhanced by vector instructions. The focus again is on the high level of configurability of the accelerator.

This translate to a mesh composed of a parameterized number of tiles, into which each core is deployed. Each core then supports multiple threads of execution, which gets internally scheduled in a round-robin fashion. A vector register file is in place, with the relative vector manipulation instructions. A Network on Chip enables the communication between the tiles and with the memory.

Programming is once again supported by a toolchain based on the LLVM compiler.

TETRaPOD

***T**ime Multipl**E**xed Fully Pipelined **Th**roughput Oriented **Pr**ogrammable **D**ata-path FPGA Overlay* is an hardware overlay designed to exploit the DSP blocks of the underlying FPGA.

It features a matrix of interconnected functional units, that form an arbitrarly deep data path for efficient data manipulation. The functional units are fully programmable at runtime, so the accelerator reads both its configuration and the input data from memory at startup.

The configuration data could be generated by a compiler, which after looking at the kernel to execute should extract a DFG and map it on the underlying hardware.

Goals and Motivations of this Work

The communication infrastructure plays a vital role in the system architecture. As depicted in Figure 1.3, the accelerators are distributed over a 2D mesh that can be spread over multiple FPGAs. The infrastructure thus includes all the components needed to make modules communicate inside the same tile, among different tiles and even different FPGAs. Moreover, it includes the connection to the HPC server and to the physical memory modules.

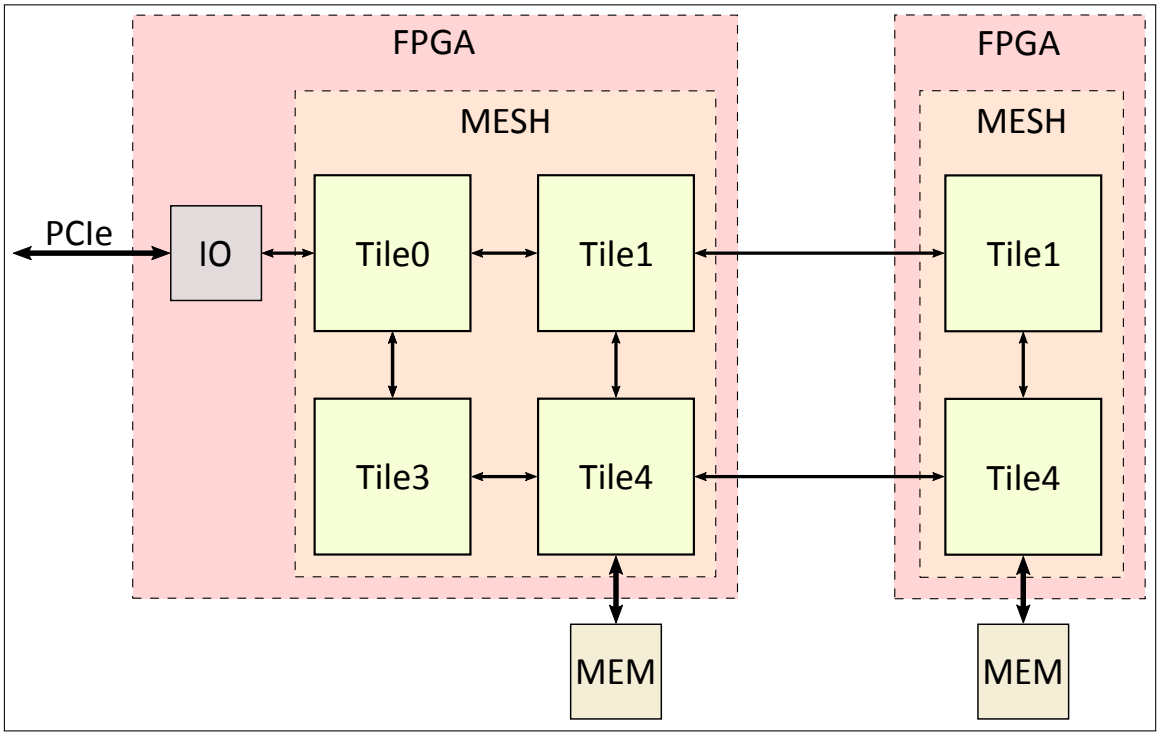


Figure 1.3: Logical view of a MANGO cluster

The performance of the communication infrastructure determines the system performance. PEAK and nu+ will fetch from memory the kernels to execute and the data to work upon. TETRaPOD will fetch a stream of data where configuration settings and data to manipulate are interleaved. Efficiency in these transfers is critical.

At the same time, user applications have to move big chunks of memory to and from the system, to quickly move execution informations (instructions in case of processors, configurations in case of hardware overlays) and data.

Moreover, some critical applications could require a minimum level of performance, that is, a quality of service requirement. The infrastructure should be able to manage the available bandwidth to the cluster and to the memory modules, to stay in line with QoS constraints.

The goal of this work is to implement the required modifications, both in the software and in the hardware layers, to allow efficient memory transfers by exploiting the full performance of the system, as seen by user applications and hardware accelerators. Moreover, the foundations are laid to effectively implement QoS policies, adding support for weighted memory accesses, allowing to split the total memory bandwidth among different transfers.

Structure of the Document

In chapter 2 the existing MANGO infrastructure is described, with a particular focus on memory access oriented functionalities.

In chapter 3 an overview of the proposed solution is introduced at the system level. Then this solution is analyzed in depth, starting from the software changes in sections Shared Memory API and Data Burst Transfers Management, and continuing with the hardware architecture changes in sections Hardware Data Channels and Memory Wrapper Redesign, highlighting all the required modifications.

In chapter 4 performance considerations are reported. The implementation is then validated by launching a set of weighted memory transfers and analyzing the resulting measurements. The resources needed to implement the design are also evaluated.

In chapter 5 the final conclusions are presented, along with some suggestions on how the results could be further improved.

CHAPTER 2
MANGO Architecture

Hardware Architecture

MANGO hardware components are organized in a *mesh* topology (Figure 1.3), where *tiles* are placed in a bidimensional structure and identified by a set of coordinates. Each tile is equipped with an accelerator, called from now on *unit*, along with other peripherals that implement the configuration and communication infrastructure.

Tiles can be spread among multiple FPGAs and motherboards according to the architecture configuration, while still offering a clear abstraction to the software and to the units.

Some tiles are connected to the memory modules, thus offering memory access to local and remote units. At least one tile should also be connected to an IO device, to allow access from the server side.

Tile Structure

The internal structure of a tile is depicted in Figure 2.1. All the components but the unit are here briefly described, grouped by the functionality they provide.

Communication Interface

The communication interface allows modules inside a tile to exchange data among themselves and with other tiles. The first thing that should be noted is that two networks are actually implemented in MANGO: the item network is used to route items through the system, and so it's just used for configuration and debugging purposes; the data network is a complex network in charge of routing all the traffic related to memory accesses from the unit.

Each network is made of two main components: a **router** and a **network interface**. The router has the responsibility of sending and receiving messages to and from other tiles. The network interface abstracts away the interconnection details from the modules, converting modules' requests to the correct network format if they should reach another tile, or forwarding the requests to local modules when it is needed. In a similar way the network interface converts network messages

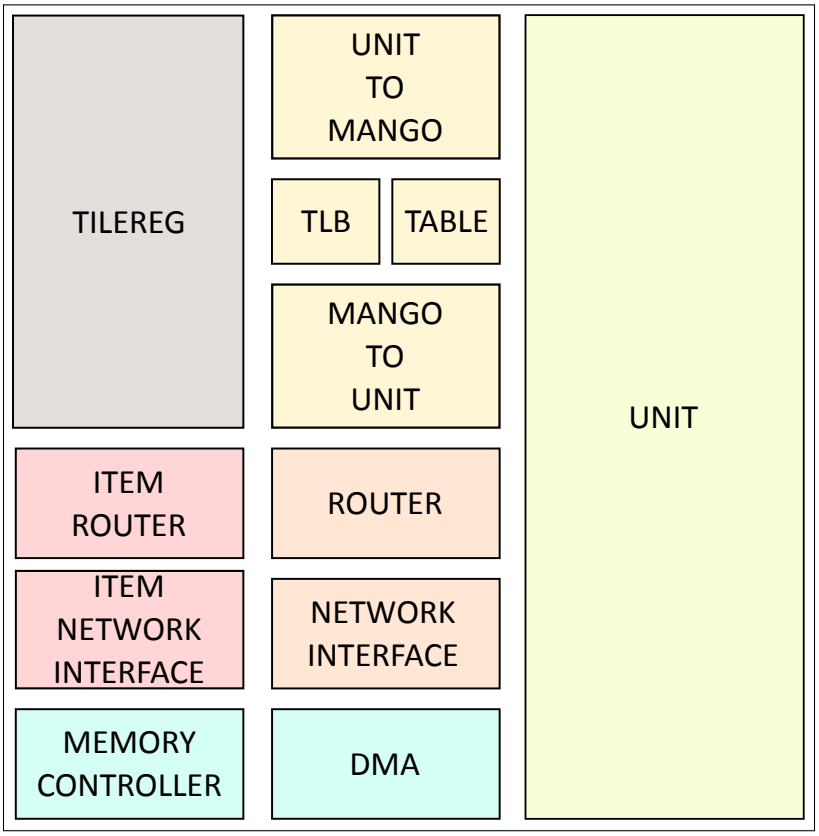


Figure 2.1: MANGO tile

in a format that can be understood by the local destination module. An in-depth description of the two networks is presented later in this chapter.

Configuration Interface

The **TILEREG** module provides an extensible interface for the configuration of the peripherals inside a tile. The details of the interface are presented in Figure 2.2.

It offers the abstraction of a register bank, where each register implements a specific functionality. The main functionalities provided are the configuration of local modules, the implementation of synchronization primitives and the collection of statistics from the local modules for performance analysis.

The registers can be accessed by the server through the item network, or they can be mapped in a unit's address space and be accessed through the data network.

Unit Interface

Every unit should adapt to a well-defined interface to interact with the MANGO system. The details are reported in Figure 2.3.

The interface can be split in two logical parts: the data interface, which allows the unit to access the memory modules in the system or the memory-mapped

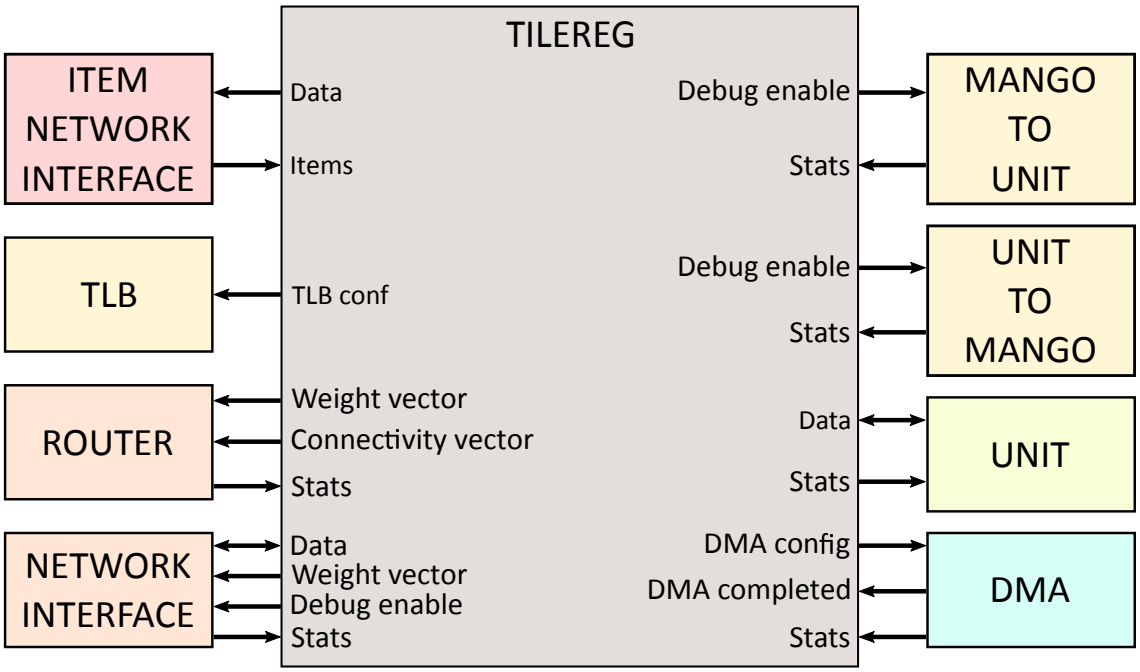


Figure 2.2: MANGO TILEREG

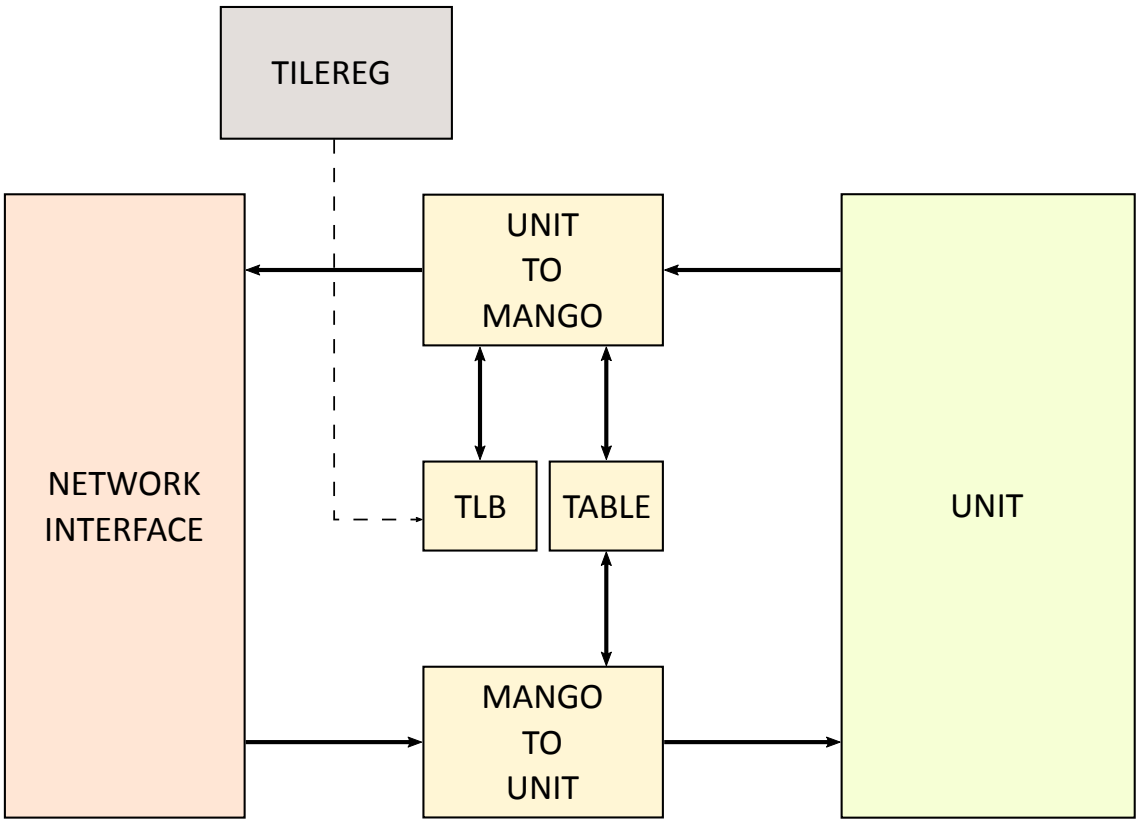


Figure 2.3: Unit interface

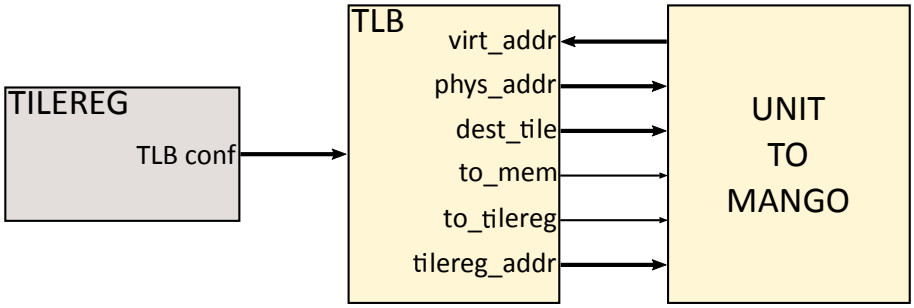


Figure 2.4: TLB module

TILEREG registers, and the control interface, which allows the bidirectional communication between the unit and the server for control related traffic.

The control interface is completely generic and every unit can implement its own commands. All the control traffic flows on the link between the unit and the TILEREG: the server can thus send commands to the unit by writing on a specific TILEREG register.

The data interface offers to the unit a clear abstraction of the memory resources. In particular, the unit should have no knowledge of the physical memory modules plugged into the system: every unit has a 4 GB address space, onto which physical memory modules and TILEREG registers can be arbitrarily mapped. For this reason, memory operations flow through the **UNIT TO MANGO** and **MANGO TO UNIT** modules (from now on *U2M* and *M2U*) before reaching the network interface. Those two modules implement a translation layer: every memory access is associated with the corresponding physical resource, and the corresponding request is generated. To do so, the U2M and M2U modules rely on two additional modules: a **TLB** and a **TABLE** module.

The TLB, whose interface is detailed in Figure 2.4, stores the translation informations needed to map memory addresses to physical resources. Its configuration can be modified at runtime by writing a specific TILEREG register. In this way the server (in particular the resource manager) is offered maximum flexibility in the mapping of physical memory modules to the unit's address space. Moreover, registers from a TILEREG inside the system can be mapped. This is especially useful for synchronization purposes, as multiple accelerators or even user applications can share access to the same set of synchronization registers.

The TABLE module, described in Figure 2.5, is used for bookkeeping purposes: when the U2M translates a memory read operation from the virtual address space to the physical one, it requests the TABLE module to store the original informations, also generating a unique ID for the transaction. The ID is sent to the memory controller and then back, so upon reception of a memory transaction's response the M2U module can recover the informations from the table.

Memory Interface

A tile can interact with a memory module through the **Memory Controller** (from now on *MC*). Its presence is not mandatory: it is implemented only in the tiles connected to a memory. Along each MC comes a **DMA** module, which allows

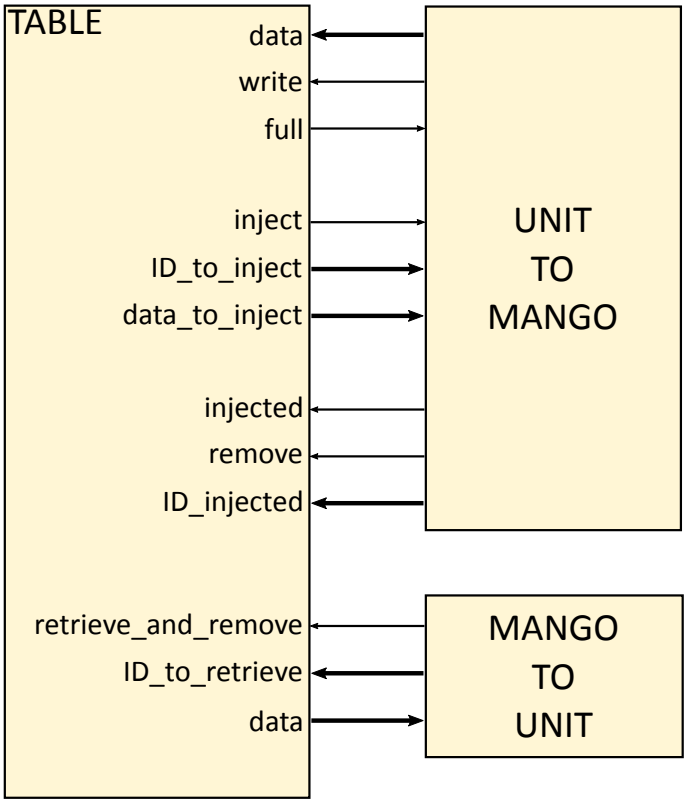


Figure 2.5: TABLE module

to move efficiently blocks of data from the local memory to a unit or to another memory in the system. The modules involved are depicted in figure Figure 2.6.

Memory controller The structure of the MC is reported in Figure 2.7. It can receive requests from two sources: the local network interface or the DMA controller. The interface with the network interface allows to request a read or write on a block, word, halfword or byte granularity. Of great importance for the read requests is the transaction ID, which will be used by the sender tile to recover transaction's informations. The DMA interface is simpler: beside the block address to read, the destination field specifies if the data should be kept in the local tile (and passed to the unit) or to a remote tile (in particular, a remote unit or MC).

On the other hand, the MC uses a well-defined interface to communicate with the physical memory wrapper, whose details are given in the Memory Wrapper section. These interface abstracts away the physical memory protocol from the MC. The only assumption made by the MC, and that should be enforced inside the memory wrapper, is that the requests must be served in-order. On the interface boundary some decoupling FIFOs are placed: this allows to run the memory wrapper at an arbitrary clock frequency.

A round robin scheme is used to arbitrate between the network interface and the DMA. The winning request then flows to the decoupling FIFOs.

In case of a read operation, some transaction informations are stored in a FIFO, while the read command is sent to the memory wrapper. When responses start to flow from the memory wrapper, the needed return informations are dequeued: for this reason it is of main importance that read requests are served in order.

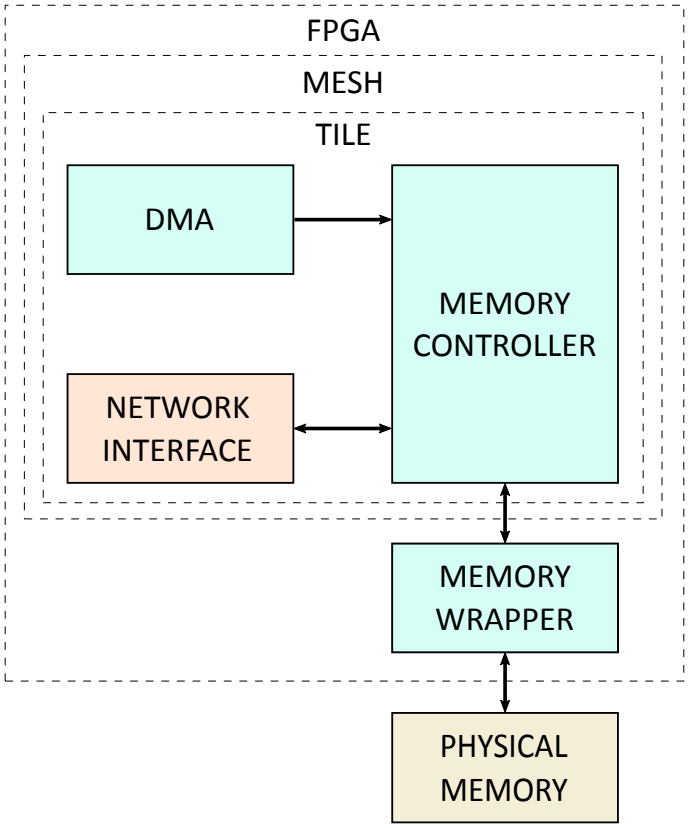


Figure 2.6: Memory interface modules

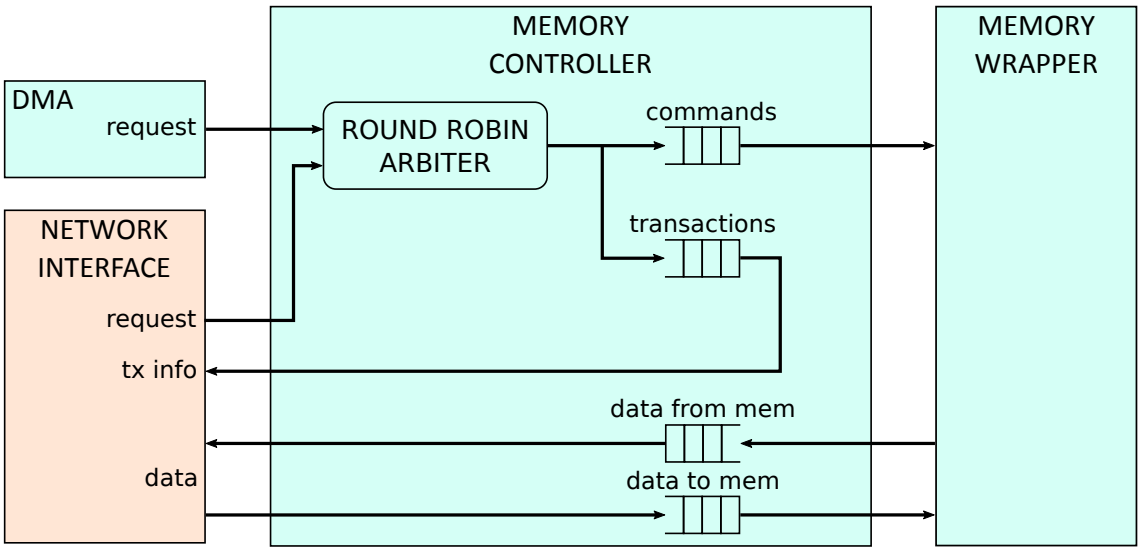


Figure 2.7: Memory controller

	WORD0	*WORD1*	*WORD2*			*WORD3*
Bit range	31 - 0	31 - 0	31 - 16	15 - 4	3 - 0	31 - 0
Width	32	32	16	12	4	32
Name	Read address	Write address	-	Destination	Function	Blocks

Table 2.1: DMA configuration words

In case of a write operation, no return informations need to be stored. The write command is sent to the memory wrapper along with the memory block to be written. In case of sub-block size writes, the necessary padding is added and alignment is performed.

DMA controller The DMA controller offers a programming interface to the TILEREG, from which it accepts a sequence of configuration words. The configuration register can be used from the register side, or be mapped into a unit's address space. The format of the configuration words is reported in Table 2.1.

The *read address* field determines the start address for the read requests generation. The *write address* field is used when the destination of the transfer is another memory controller: it determines the address from where the data is written in the destination memory controller. The *destination* field determines the destination tile and component. The *function* allows to set the DMA working mode: at the moment just the memory read is supported. The *blocks* field specifies the number of memory blocks to be moved.

On the other hand, it offers a completion signal which is used to generate an *interrupt item* that gets routed to the server: this notification system allows applications to know when a transfer gets effectively completed.

As soon as a transfer is programmed, the controller starts to generate requests to the MC, updating each time the request address until the programmed block count is reached, raising the completion signal.

Data Network

The MANGO data network is the backbone of the system. It is designed with flexibility and configurability in mind, offering the required performance while meeting QoS constraints.

The network can be split in two main parts: the Router, which forwards messages between the tiles, and the Network Interface, which abstracts away the network details and adapts to the local tile's modules interfaces.

A packet, before being handled by the network, has to be split in *flits*. The flit is the smallest chunk of data that the network routes atomically. In MANGO, the flit size is set to 64 bits, although the system is fully parameterized and can support arbitrary flit sizes. The flow control is applied on a flit-basis: this allows to exploit the full capacity of the network input buffers [6]. Every flit has an associated *flit type* field, that is used to identify the first and last flit of a packet.

The flit size doesn't necessarily match the network bus size: for this reason, the concept of *phit* is introduced, which corresponds to the smallest amount of data that can be transferred between two routers: in the most general case, a flit can be composed of an arbitrary number of phits. In the MANGO network a flit is split into phits, which are 8 bit in size, just on the boundary between two FPGAs. The reason is the limited amount of IO resources that can be found on an FPGA: splitting a flit into smaller chunks allows to use less resources, although special care must be taken to deal with the obvious loss of bandwidth. A common solution is called *pin multiplexing* and consists in using a faster clock signal on the interface between the FPGAs, to balance out the bus width shrinking.

The system has full support for *Virtual Networks* (or VN), which allow to implement multiple logically separated networks sharing the same physical links. In particular, in Network on Chip design it is of main importance to guarantee that no deadlocks can occur at the routing and application protocol level [6] (more details later). Virtual networks serve the purpose of logically splitting different packet streams, to guarantee that no deadlock can occur between any two modules using different virtual networks. For this reason, each virtual network has its own buffers, with the corresponding flow control signals.

The virtual network in the system are allocated in the following way:

- VN 0: messages that should reach the memory controller are routed through this VN. They can be read and write operations triggered by units, or memory transfer operations triggered by the DMA module.
- VN 1: it's allocated to messages that should reach the units. They are the reply to a memory or TILEREG read operation.
- VN 2: used for messages that should reach the TILEREG. These are the read and write operations triggered by units.

As it emerges from the virtual networks allocation scheme, replies are always sent on a different virtual network than the requests: this, together with the resource isolation guaranteed by the virtual network implementation, ensures that no deadlock can happen between the modules involved in the communication.

As traffic from multiple virtual networks compete for the same physical connection, a proper arbitration policy should be put in place. This arbitration stage also affects the network bandwidth seen by the modules in the system. To be able to target different QoS requirements, the MANGO network employs a *weighted arbitration scheme*. That is, every time the physical link must be allocated to a virtual network, maximum priority is given to a specific one, or a round-robin fallback policy is used. This allows to allocate a certain amount of network bandwidth on the virtual network granularity. The weights are specified by a *weight vector*, whose default value can be specified in the architecture file and gets mapped on a specific TILEREG register: the weights propagate from the TILEREG to the required modules. This also means that the weight vector can be modified at runtime by writing on the corresponding register.

Virtual channels are also supported by the network, although not currently used. Virtual channels are a common technique employed in data networks to

avoid *head-of-line blocking*, which occurs when the first flit in the input buffers cannot proceed, thus blocking the next ones that could potentially be able to leave the router through another output link.

Network Interface

As stated above, the network interface has the goal of abstracting away the interconnection details from the local tile's modules. This means that it forwards the requests to the local modules, or inject them through the data network after converting them in the appropriate format.

In Figure 2.8 the structure of the module is presented. The INJECT module has the responsibility of properly injecting flits into the network. It supports a parameterized number of input ports, each injecting on an arbitrary virtual network. Internally it implements an arbitration stage, which follows the weighted arbitration described above.

The EJECT module receives flits from the network and rebuilds the original packet. Depending on the virtual network and on the message format, the ejector forwards the packet to the correct receiving module.

Both the TILEREG and the MC modules interface with a *from net* and a *to net* module. The *TR from net* and *MC from net* modules receive requests from the ejector in network format, or from the local unit modules. They have the responsibility of interfacing with the corresponding module, abstracting away the source of the request. The *TR to net* and *MC to net* modules send requests coming from the corresponding module to the local unit, or to the injector in case the destination unit is placed in another tile. Once again, the corresponding module has no knowledge of the interconnection details.

The U2M module forwards messages coming from the unit, and it interfaces directly with all the possible destinations: the TILEREG (through the *TR from net*), the MC (through the *MC from net*) and the injector. The M2U module works in the dual way, receiving messages from the TILEREG, the MC or the ejector and forwarding them to the unit.

Router

The router is the core of the MANGO network, and its interface is described in Figure 2.9. It features 5 bidirectional communication ports: one for each cardinal direction, to communicate with other routers in the mesh, and a local one, to communicate with the local tile's modules.

The relevant signals for each port are: the flit to transfer, the flit type, the VN number, the Go signals (one per VN) for flow control implementation and the Valid signal for handshaking purposes.

It is designed in a pipelined fashion to achieve high performances, with the pipe stages reported in Figure 2.10 and now described in detail.

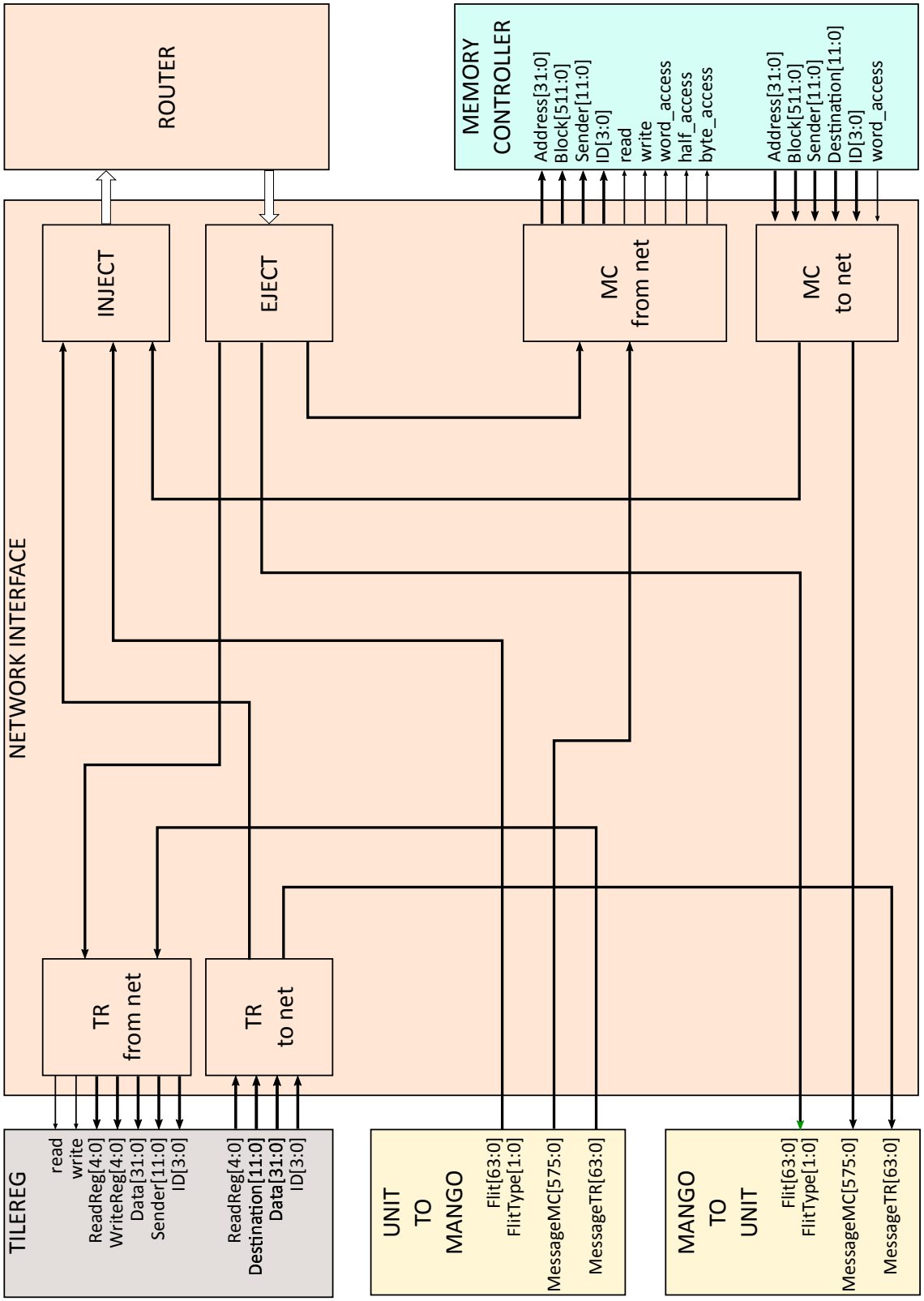


Figure 2.8: MANGO network interface

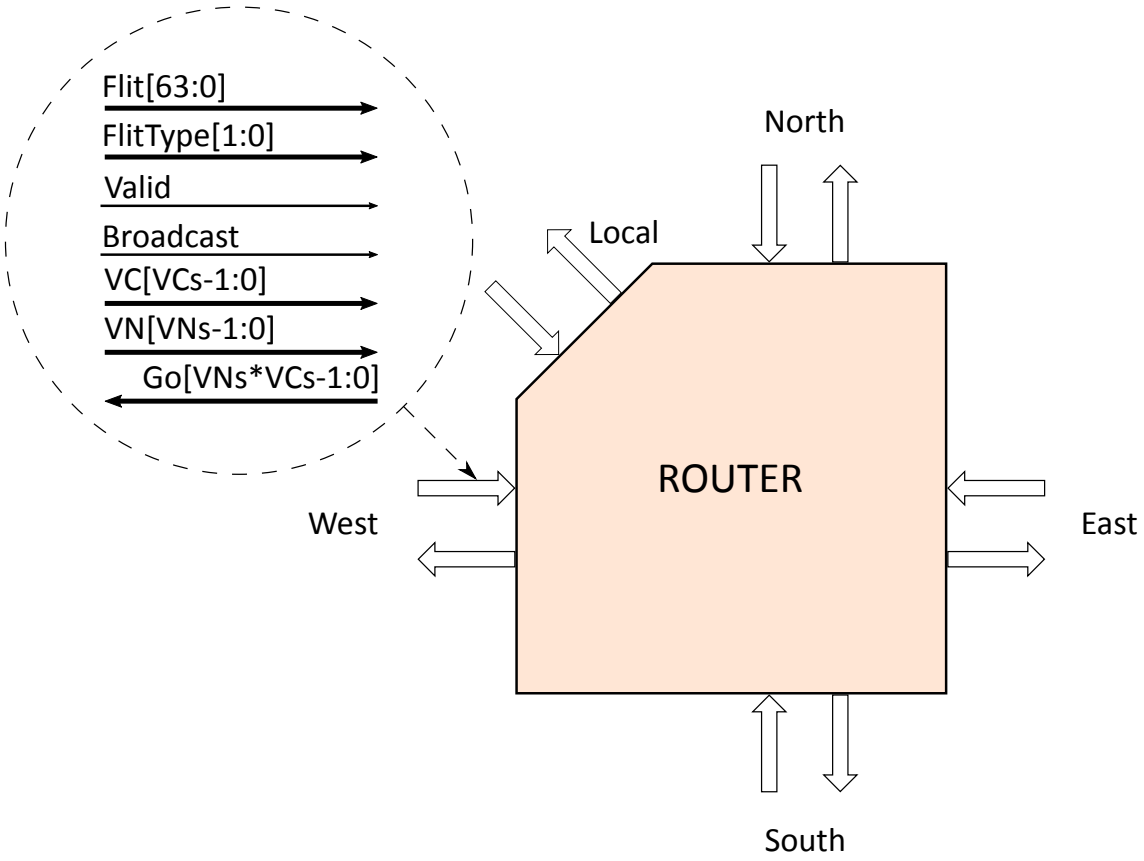


Figure 2.9: MANGO router

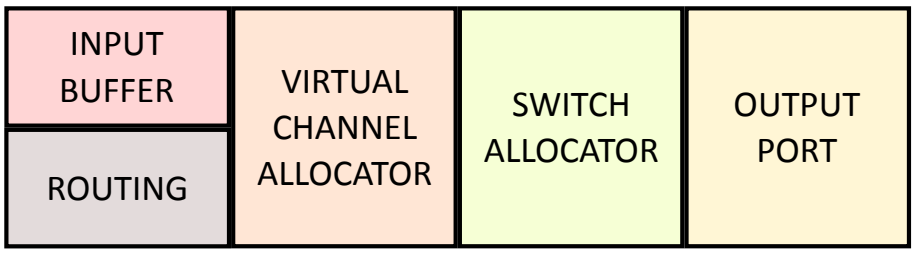


Figure 2.10: Router pipeline

Input Buffer and Routing The input buffering stage is used to store the flits to be processed. Every input port has its own set of buffers. To guarantee resource isolation between the virtual networks, a different set of buffers is used for each one.

The buffer utilization level can thus be used to generate the flow control signals for the injecting module.

Special care must be taken if the input port size doesn't match the flit size: in this case the input buffer receives the phits and rebuilds the corresponding flit, then it can enqueue it.

The routing phase consists on analyzing the destination tile and, given the current tile, selecting the correct output port.

Virtual channel allocator The virtual channel allocator stage assigns a virtual channel to each packet. It is important to stress that virtual channel allocation is done on a packet basis and not on a flit basis: this is to guarantee that flits belonging to the same packet get routed on the same virtual channel. For this reason the VA stage works only on the reception of a header flit.

After this stage, each input virtual channel has a corresponding output virtual channel assigned.

Currently the MANGO network does not use virtual channels, which means that every virtual network has only one virtual channel: for this reason, the arbitration always succeeds if the next router is available, assigning the only virtual channel available to the requestor.

Switch allocator The switch allocator analyzes the virtual channel allocations and resolves any conflict regarding flits that should leave from the same output port. In this stage the weighted arbitration takes place: for each output port, priority is given to the flits coming from a specific VN, as described previously. The result of this stage is a mapping of input virtual channels onto router's output ports: the flit is now ready to reach the corresponding output port.

Output port The output port interfaces with the next router's input port or with the ejector, managing the physical connection between the two devices.

In its most basic form, it is just composed of an output buffer. However, the physical connection size could not match the flit size: it is its responsibility to generate the corresponding phits from the current flit. In the MANGO network this happens when traffic reaches the FPGA boundary.

Message Format

In this section the format of the messages routed through the network is described in detail.

Depending on the message type, the packet is composed of a variable number of flits. For example, a memory read request is composed of just one flit, while

the subsequent answer is composed of a header flit plus all the additional flits needed to carry the memory content requested.

	Fields		
Bit range	63 - 52	51 - 40	39 - 0
Width	12 (9 + 3)	12 (9 + 3)	40
Name	Source (tile + component)	Destination (tile + component)	-

Table 2.2: MANGO message common fields

Common fields All MANGO messages include the source and destination fields, as described in Table 2.2. In particular, these two fields are composed of two parts: the tile ID, which is 9 bits wide, and the component ID, which identifies a module inside a tile and is 3 bits wide.

It should be noted that the minimum amount of informations needed to correctly route a message is just the destination field: in this way the router can correctly allocate the internal resources and forward the message on the right path.

	Fields				
Bit range	63 - 40	39 - 38	37	36 - 32	31 - 0
Width	24	2	1	5	32
Name	Common	10_b	-	Register	Data

Table 2.3: TILEREG write request format

TILEREG write requests The TILEREG write request (Table 2.3) includes the register address to write, which is 5 bits wide, and the new register contents which is 32 bits wide.

	Fields					
Bit range	63 - 40	39 - 38	37 - 36	35 - 32	31 - 5	4 - 0
Width	24	2	2	4	27	5
Name	Common	01_b	-	ID	-	Register

Table 2.4: TILEREG read request format

TILEREG read messages The TILEREG read request (Table 2.4) includes the request ID, which is the unique ID assigned to the transaction by the sender tile's TABLE (see Unit Interface), and the register address, which is 5 bits wide and so allows to access 32 registers.

The response (Table 2.5) includes the request ID, which is the same value sent inside the read request, and the register content, which is 32 bits wide.

	Fields				
Bit range	63 - 40	39 - 38	37 - 36	35 - 32	31 - 0
Width	24	2	2	4	32
Name	Common	11_b	-	ID	Data

Table 2.5: TILEREG read response format

	Fields					
Bit range	63 - 40	39 - 36	35 - 34	33 - 32	31 - 26	25 - 0
Width	24	4	2	2	6	26
Name	Common	Word offset	-	Byte offset	Command	Address

Table 2.6: Memory write request format

Memory write requests The destination module of this type of message (Table 2.6) is always a memory controller. The Command field can be one of the following:

- **WB_DATA_TO_MC**, if a whole block sent by a unit has to be written to memory. As a memory block is 512 bits wide, this header is followed by 8 flits with the content to be written to memory.
- **WORD_WRITE_TO_MC**, **HALF_WRITE_TO_MC**, **BYTE_WRITE_TO_MC**, if respectively a word (32 bits), halfword (16 bits) or byte (8 bits) has to be written to memory. As a flit is 64 bits wide, this header is followed in all the cases by 1 flit with the content to be written to memory.
- **DMA_MEM_TO_MEM**, if a whole block has to be written to memory, and the request was sent by a DMA attached to another memory in the system. This header is followed by 8 flits with the content to be written to memory.

	Fields				
Bit range	63 - 40	39 - 36	35 - 32	31 - 26	25 - 0
Width	24	4	4	6	26
Name	Common	Word offset	ID	Command	Address

Table 2.7: Memory read response format

Memory read messages This format (Table 2.7) is used to read data from memory, so the destination module of such a message is always a unit. The Command field can be one of the following:

- **DATA_FROM_MC**, if the message is a block read request or response. In case of the response, the header is followed by 8 flits with the memory block contents. The word offset field is ignored.
- **WORD_READ_FROM_MC**, if the message is a word read request. The word offset is used to identify the word to be accessed, and the response will come as a *DATA_FROM_MC* message.

Item Network

The item network has a simpler structure than the data network. It is used to route items through the system, which are control and debug related informations. As debug informations can very easily congest the system, a clock gating mechanism is in place: when the item network gets congested, the data network and unit clocks are gated. Then different performance counters are used for each of the two clock domains; this allows to make accurate performance measures even when extracting a great amount of debug information.

The format of an item is reported in Table 2.8.

	Fields			
Bit range	31 - 26	25 - 14	13 - 8	7 - 0
Width	6	12	6	8
Name	-	Destination	Command	Payload

Table 2.8: Item format

An item is represented on a 32 bits word, although just 26 are used. The destination field includes the tile and the component ID: the former is used to route the item through the network, while the latter is used to identify the destination module. The command and data fields contain the real request which gets analyzed by the corresponding module.

The item network is once again divided in two parts: the network interface used by local modules to interact with the network, and the router used to forward items to the adjacent tiles.

The network interface is responsible for the communication with the local modules. The interface is synchronous with the unit clock, so internally the network interface brings the items into the item network's clock domain, then injecting them through the network. To keep the structure as simple as possible, during the ejection phase the items are broadcasted to the local modules, which check the destination field and decide whether to ignore or process the request.

The router has the same implementation as the one used in the data network, thanks to its high level of configurability. In the item network the flits are 32 bits wide, so each item corresponds to a flit, and only one virtual network is implemented.

Memory Wrapper

The memory wrapper abstracts the physical memory architecture from the rest of the system. In principle multiple memory types could be plugged into the system, as long as there is a memory wrapper capable to handle its interface. At the moment the MANGO system supports DDR3 and DDR4 memory modules. Support for an FPGA block RAM-based memory implementation is also provided, mostly for simulation and development purposes.

Regardless of the memory type, the only requirement posed on the memory wrapper is that it should serve the read requests in-order. The memory controller,

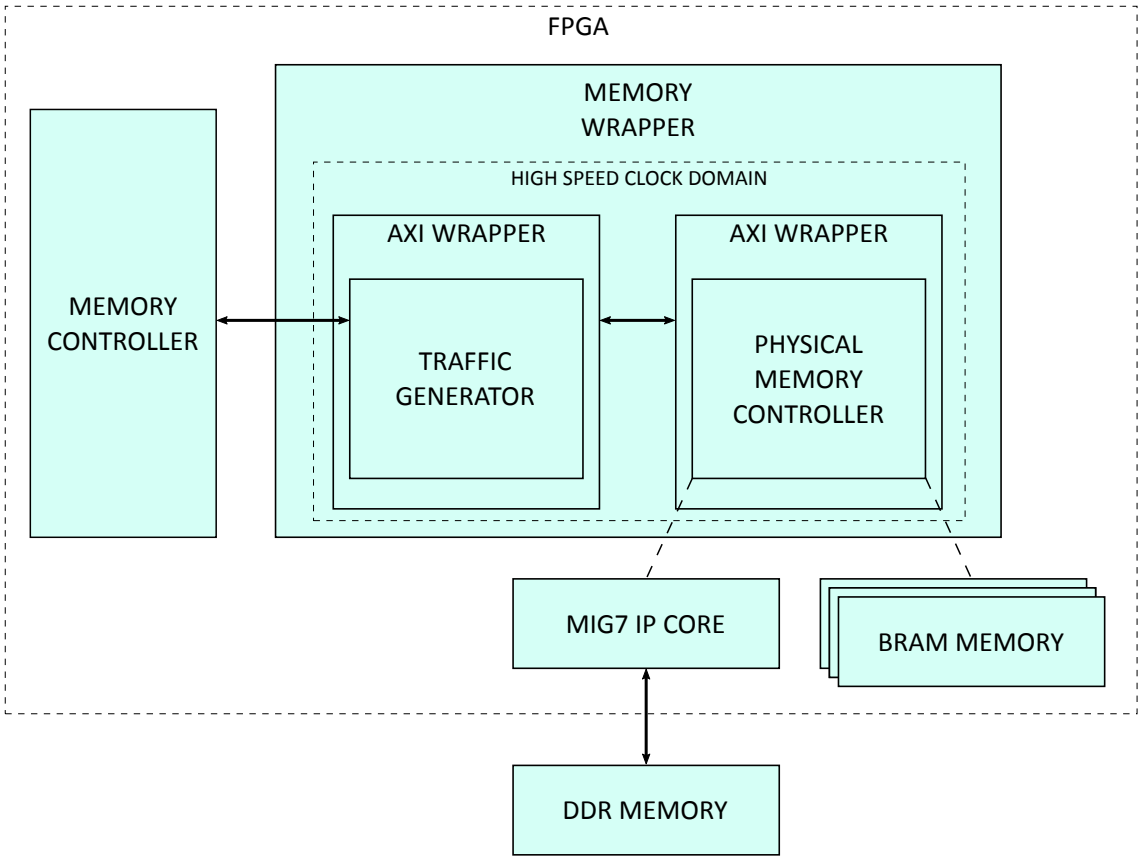


Figure 2.11: Memory wrapper architecture

which interfaces with the memory wrapper, relies on this assumption for proper read requests handling.

The general architecture of the memory wrappers implemented is reported in Figure 2.11. The interface is composed of a command bus, which encodes the request type together with the target address, and a data bus in both directions, which moves memory blocks from and to the memory.

The internal structure of the memory wrapper is derived from the example designs provided by Xilinx. The traffic generator adapts the interface with the MC to the AMBA AXI interface, which is supported by all the physical memory controllers.

The traffic generator submits requests in order, waiting for the current request's completion before submitting a new one. This implementation guarantees the ordering requirement on read requests, although it carries a big performance penalty, preventing any type of optimization that could be exploited by reordering the requests inside the physical memory controller.

The physical memory controller is the Memory Interface Generator (MIG) [7] provided by Xilinx, which can support DDR3 and DDR4 memories. In the case of the block RAM-based memory, the physical memory controller is a bank of 16 block RAM IP cores from Xilinx.

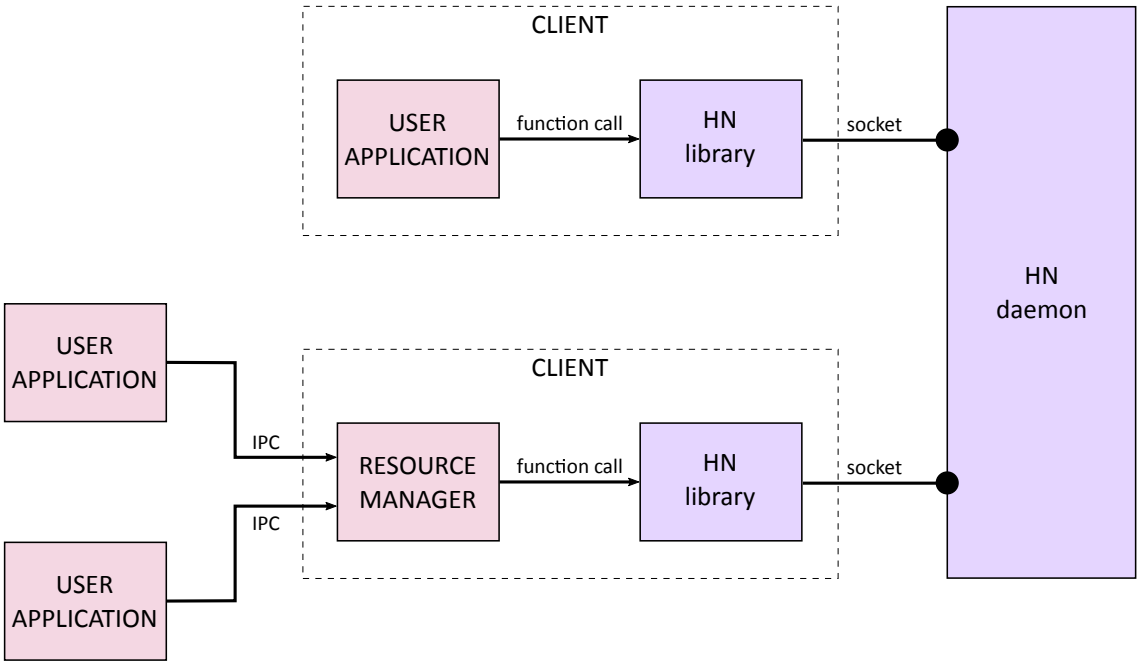


Figure 2.12: Software architecture overview

Software Architecture

The MANGO software architecture has the responsibility of offering a low-level API to interact with the cluster. This interface can be used directly by an application, in case it needs low-level access to the system, although a resource manager is being developed as an additional software layer. The resource manager would be in charge of satisfying the application's requests, by allocating accelerators and bandwidth to the corresponding application. The overall architecture is described in Figure 2.12.

HN Library

The *HN library* implements the low-level API offered to the applications or to the resource manager. It is written in the C language and compiled as a shared library, so applications can link against it and interact with the cluster via function calls.

The main responsibility of the HN library is to translate requests to a format that can be understood by the HN daemon. For this reason every function call to the API generates one or more requests on the daemon side.

The HN library communicates with the daemon using a UNIX socket, which is a common API used on UNIX-based systems for inter-process communication. For this reason a serialization step is required before sending the request, to convert it in a format that can be sent over such a communication channel. The corresponding deserialization step is performed on the daemon side when the request is received.

The main functionalities offered by the API are here briefly summarized.

- Reset and architecture identification functions

- Configuration registers and performance counters access functions
- Accelerator interface functions (querying for which accelerator is present in a given tile, sending commands to it, etc...)
- Low-performance memory access functions, for reading and writing memory at block, word, halfword or byte granularity

Moreover, every accelerator adds some functions in the API, which are dependent on the specific accelerator and so aren't analyzed here further.

HN Daemon

The *HN daemon* manages all the communications between the HN library and the cluster.

After starting up it runs in background, initializing the hardware architecture and opening a UNIX socket to receive requests from applications. At the same time it waits for the cluster to send data to the host. All the commands to and from the cluster get encoded into *items*, which are described in detail in Item Network.

The internal structure of the daemon is depicted in Figure 2.13. The duties of the daemon are distributed among a set of loosely coupled threads that communicate with each other through software FIFOs.

- The **master thread** is responsible of handling the server socket and listening to incoming connections, then generating a new slave thread for each client.
- The **slave thread** handles the communication between the daemon and a specific client. The incoming requests, which have been properly serialized by the HN library, are parsed and converted to items, then sent to the collector thread.
- The **collector thread** handles the requests coming from the clients and convertes them into items. If needed, it generates any additional item needed, and sends them to the cluster through the backend thread, or to the dispatcher thread in case a reply must be forwarded to the application.
- The **dispatcher thread** reads items coming from the cluster and from the collector, and handles them accordingly, generating any required notification and sending it to the client application through the slave thread.
- The two **backend threads** handle the low-level communication with the cluster. They send and receive data through FIFOs and they interface with the PCIe driver.

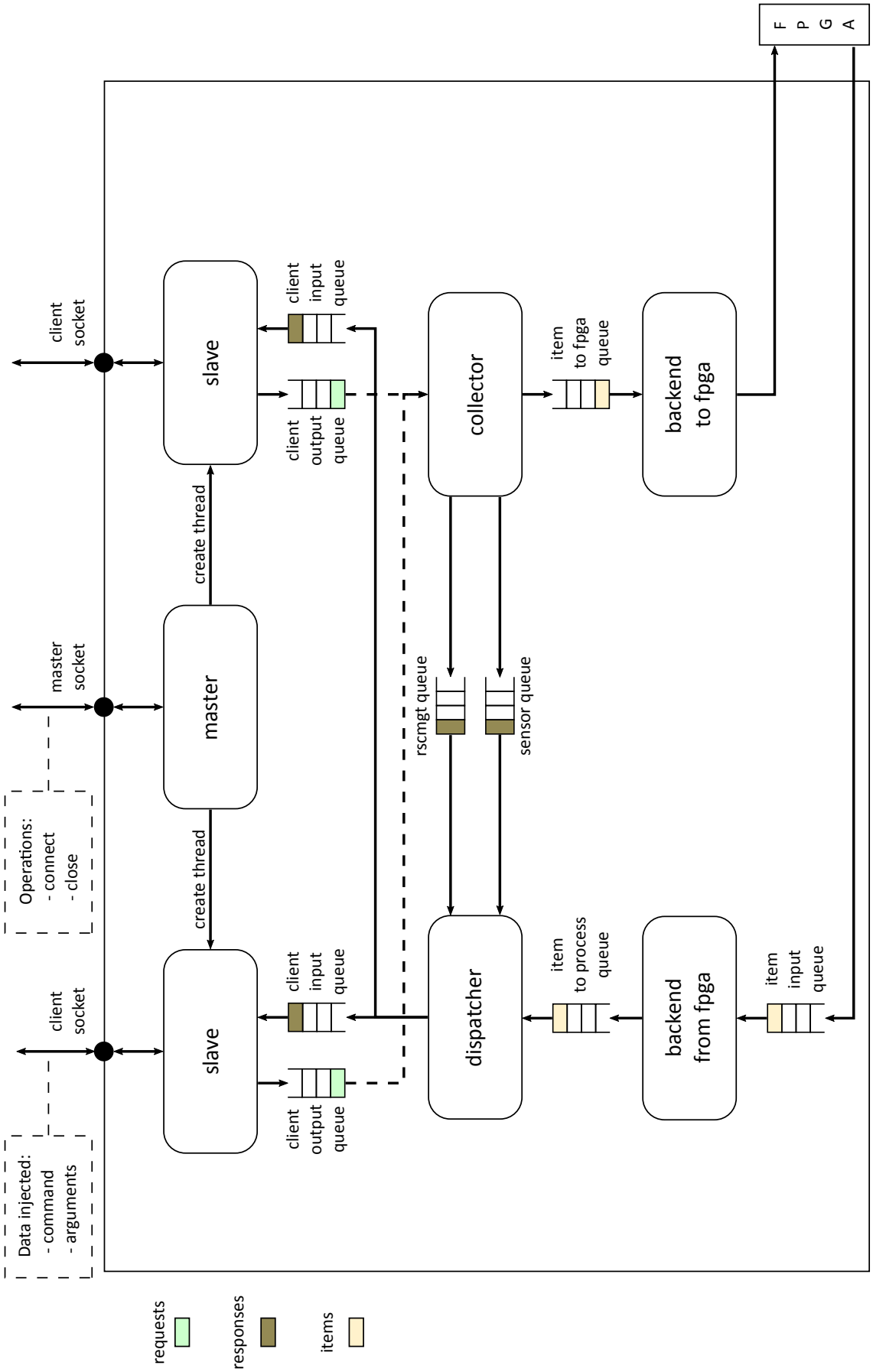


Figure 2.13: HN daemon structure

Conclusions

In this chapter we described the current architecture developed within MANGO. As shown, the system doesn't provision mechanism for proper memory transfer achievements nor any provision for effective bandwidth reservation policies.

It is the goal of this thesis to provide such support by analysing a strategy, deploying it and validating it. We describe it in the next chapter.

CHAPTER 3
Implementation of Efficient Memory Transfers

As the main objective in this thesis we need to implemented efficient memory transfers on top of the architecture previously described. As the architecture is highly complex, we need to break down the effort and modifications in several parts. One important aspect is to allow multiple applications concurrently transferring big amounts of data between the host and the memories located on the FPGA, and the fact that those transfers need to be weighted (each application may reserve different amounts of bandwidth).

Taking this into account, the following changes are identified:

Changes are required on every layer of the system:

- the application needs an API to send big chunk of memories and the manipulation of the QoS policies
- the HN library needs an efficient way to send the data to the daemon
- the HN daemon has to handle multiple transfers, potentially coming from multiple applications, guaranteeing the QoS requirements
- the hardware architecture has to apply the same QoS policies adapted by the daemon, offering efficient access to the memory modules
- the built-in network and the IO module need to be modified in order to support concurrent transfers of data

An overview of the proposed solution is depicted in Figure 3.1. The figure shows both, the software side running on the host, and the hardware system on the FPGA side. The following modifications are suggested:

- applications will define and use shared memory segments instead of pipe communication channels to send and receive data. This will remove a first bottleneck for efficient transmission between applications and the HN daemon;
- the HN daemon will be modified in order to support concurrent transmissions, multiplexing them into the PCIe connection and following the QoS policies set by applications;

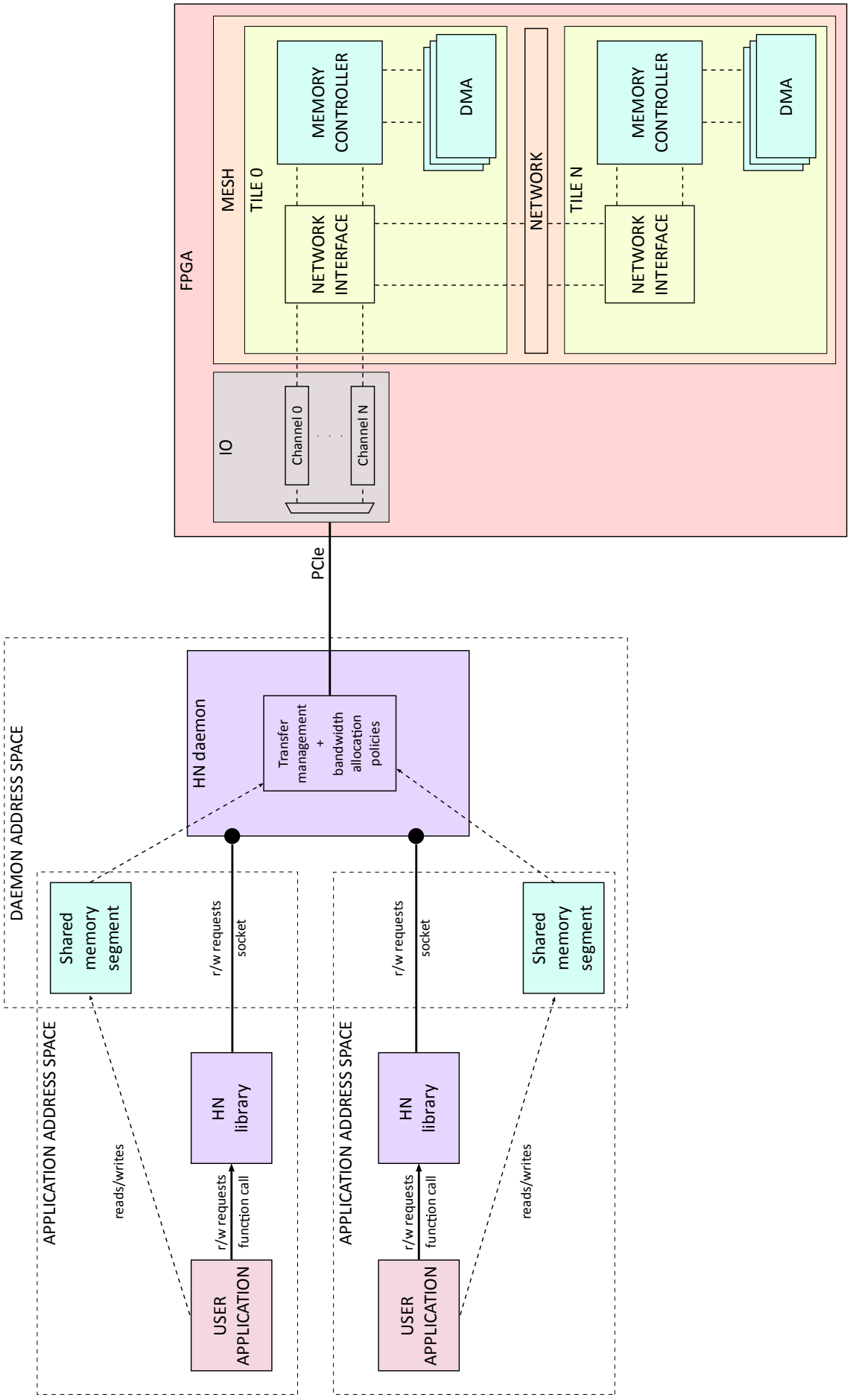


Figure 3.1: Hardware and software architecture of the proposed solution

- the IO module will be modified in order to support multiple concurrent transmissions (in both directions). This will be mainly provided by the instantiation of different queues, one for each transmission. We refer to this as data channels;
- the built-in network will be also adopted to support concurrent data channels by using more virtual networks. Indeed, one new virtual networks will be instantiated for each data channel;
- specific modules will be implemented to support data channels transmission in the network interface modules;
- the DMA device will be modified in order to support also concurrent data transfers. Currently, DMA supports only one data transfer;
- accesses to the memory controller need to be adapted as well in order to support weighted accesses;
- memory access will be also optimized in order to reach maximum bandwidth on DDR3 and DDR4 memories.

In the next sections, every part of the system is analyzed in depth.

Shared Memory API

The UNIX socket channel is not meant to move efficiently big chunks of data. Sending data over a socket requires the kernel to make at least one copy of the data, with a relevant additional overhead.

In the proposed solution, the data to transfer flows through the software layers in the form of a *shared memory*. A shared memory can be conceptually seen as a memory segment which is shared among two different applications for interprocess communication purposes. The kernel can easily implement this by mapping this memory segment in the address space of the involved processes. The setup only requires a modification in the *page table* of the processes, and then allows shared access to the data without incurring a data copy overhead. However, the operations to handle the shared memory segments are still sent through the UNIX socket, and is handled by the daemon like every other command.

The HN library implements the following functions to offer shared memory creation and destruction.

```
1 uint32_t hn_shm_malloc(uint32_t size, void **ptr)
2 uint32_t hn_shm_free(void *ptr)
```

The **hn_shm_malloc** function allows an application to request the allocation of a shared memory segment of the given size. The second parameter is actually used to return the address of the shared memory segment in the application's address space. The underlying implementation relies on the POSIX shared memory objects API and its main steps are described below.

```
hn_shm_malloc(uint32_t size, void **ptr) {
  prepare allocation request of size bytes
  send request to daemon over socket

  read response from socket
  extract shared memory ID from response
  map shared memory in local address space using the given ID
  set ptr to the local shared memory address
  add shared memory informations to list of open shared memory
      segments
}
```

The memory isn't allocated in the HN library: it sends an allocation request to the daemon and uses the returned handle to map the memory area into the application address space. Then the address is returned to the application. A list of active memory segments is kept for bookkeeping purposes.

The steps taken by the daemon upon the reception of such a request are reported below.

```
on shared memory allocation request reception {
  extract the size of the shared memory from request
  allocate requested buffer
  assign a unique name to it through the shared memory POSIX API
  add buffer address to shared memory buffers list
  send the name back to the application
}
```

The daemon, after receiving the allocation request, allocates the memory buffer and maps it on a shared memory object, associating a unique name to it. This name is then transferred to the user application, that using the same API can access the shared object through its name and map it in its own address space.

The **hn_shm_free** function destroys a shared memory segment. On the application side, the buffer is unmapped from the address space and removed from the shared memory segments list. Then, a deallocation request is sent to the daemon, which unmaps the shared memory object through the POSIX API, removes it from the active memory segments list and frees the resources.

To send a memory chunk to the cluster, the following functions are provided.

```
uint32_t hn_shm_write_memory(void *ptr, uint32_t size, uint32_t
    address_dst, uint32_t tile_dst, uint32_t blocking_transfer)
uint32_t hn_shm_read_memory(void *ptr, uint32_t size, uint32_t
    address_src, uint32_t tile_src, uint32_t blocking_transfer)
```

The **hn_shm_write_memory** and **hn_shm_read_memory** functions have very similar prototypes. They require a pointer to the memory area to send, together with its size. Then the source/destination tile and address should be provided. The last flag allows the application to choose between a synchronous or an asynchronous transfer: the former requires the transfer to be completed before returning the control to the application, while the latter allows the application to continue its execution. The implementation of these functions relies on a set of functions provided by the daemon called *data burst API*, which is documented in

the section Data Burst Transfers Management. Here the main steps taken by these functions are reported.

```
hn_shm_write_memory(void *ptr, uint32_t size, uint32_t address,
    uint32_t tile, uint32_t blocking_transfer) /
hn_shm_read_memory (void *ptr, uint32_t size, uint32_t address,
    uint32_t tile, uint32_t blocking_transfer) {
  for each registered shared memory segments shm {
    if ptr inside shm address range {
      calculate offset = ptr - shm base address
      format data burst request
      send data burst request to daemon

      wait response from socket

      break
    }
  }
}
```

Both functions allow the application to send just a portion of the mapped memory. For this reason, the pointer could be placed anywhere inside a shared memory segment. The HN library uses the list of the active shared memory areas to check if the given pointer and buffer size are within the bounds of a mapped segment. It then converts the pointer to an offset relative to the shared memory base address: this offset is passed on the daemon side to identify the start of the chunk of data to be sent.

Weights API

A weight management function has been added to the HN library. It allows to change the weight vector at runtime, thus giving maximum flexibility to a resource manager in terms of network and memory bandwidth allocation.

The prototype is here reported.

```
uint32_t hn_set_vn_weights(uint32_t *weights, uint32_t
    num_weights)
```

The function takes as arguments the weight vector and the number of valid weights in the vector. The pseudocode of the implementation is then reported.

```
uint32_t hn_set_vn_weights(uint32_t *weights, uint32_t
    num_weights) {
  pack weights into a bit vector

  for each tile t in the system {
    generate tilereg write request on weight vector register at
        tile t
    send request to daemon
  }

  generate daemon backend weights update request
  send request to daemon
}
```

A TILEREG write request with the new vector is generated for every tile in the system and sent to the daemon. This allows to propagate the weights on the hardware side. Then, a special request is sent to the daemon, that instructs it to update the weights inside the backend. In this way, both the hardware and software arbiters are properly updated.

It should be noted that applications are not meant to use this function directly, even if they could. In fact the application, that can only access memory transfers through the shared memory API, has no way to ask the daemon for a transfer to be assigned to a specific data channel.

Bandwidth management is resource manager's responsibility. Some future work might consist in adding an API to allow a resource manager to handle the channel assignment among the applications. With the current implementation, channels are assigned in a round-robin fashion among the incoming requests. More details are provided in the next section.

Data Burst Transfers Management

The HN daemon needs to be restructured, to account for the shared memory and burst transfers API offered to the HN library, and for the data channels implemented on hardware. Moreover, the arbitration scheme between the data channels should be implemented to account for QoS requirements. In Figure 3.2 the daemon's structure is presented again, with all the required changes clearly highlighted.

The collector thread reads clients' requests from the corresponding output FIFO. It is thus the first thread to receive shared memory of data burst transfer requests. It has the goal of decoding the request, and forwarding every additional request to the other threads.

The backend threads share a common data structure named *shm_running_transfers_table*, used for bookkeeping purposes, and summarized in Figure 3.3. The table tracks, for each channel, the list of active downstream and upstream data transfers. By querying and updating the table, the two threads advance the execution of the transfers up to their completion. In case of downstream transfers, the dispatcher thread is in charge of receiving the completion notification from the FPGA. When the transfer is completed, a notification is generated for the client, if the transfer was blocking. In any case the table slot associated with the transfer gets marked as free.

Item Collector

The *item collector* thread must read and process requests coming from clients.

For the scope of the proposed modifications, two kind of requests are relevant: shared memory and data burst transfer requests. As the former have been already analyzed previously, only the latter are considered here. The collector thread, upon the reception of a data burst transfer request, after validating the request

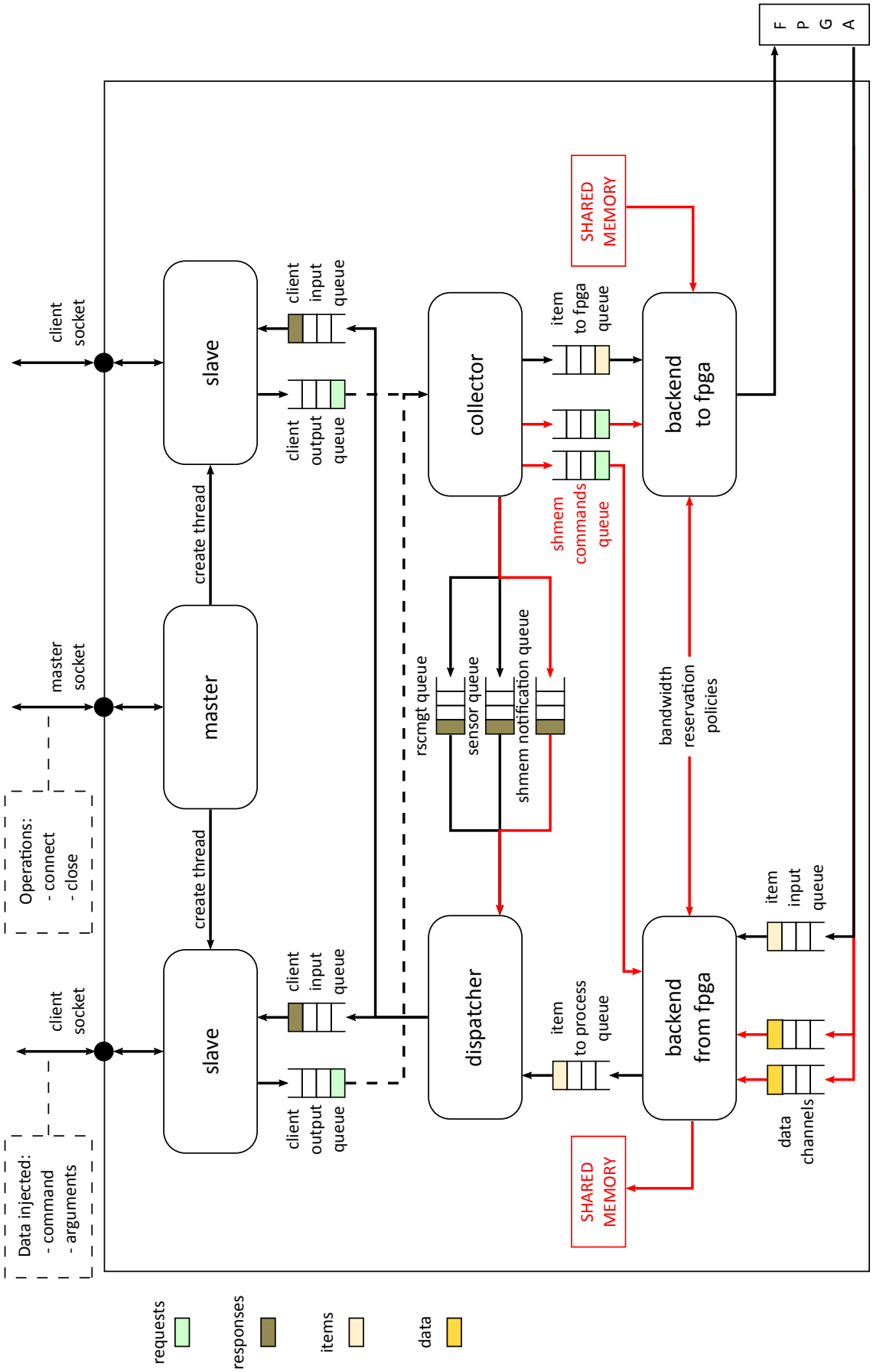


Figure 3.2: New HN daemon structure

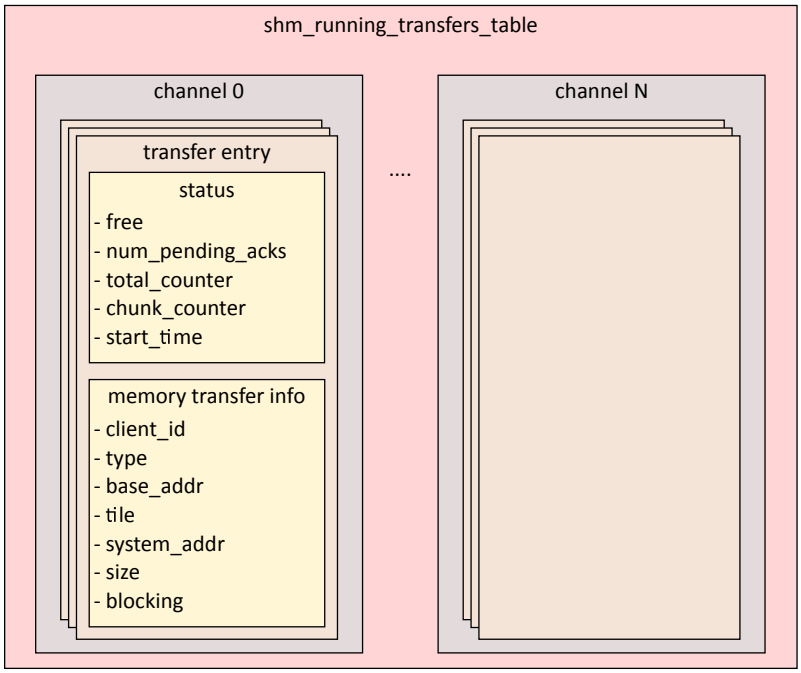


Figure 3.3: shm_running_transfers_table data structure

parameters, sends it to the appropriate backend thread, depending on whether the operation is a read or a write. The steps of the algorithm are here reported.

```
on data burst transfer request reception {
  extract shared memory ID from request

  for each registered shared memory segments shm {
    if shm matches ID {
      extract shm base address
      extract start offset from request
      calculate transfer start address as shm base address +
          offset

      if transfer is downstream {
        send transfer info to downstream commands queue
      } else if transfer is upstream {
        send transfer info to upstream commands queue
      }

      break
    }
  }
}
```

Backend to FPGA Thread

The *backend to FPGA* thread has the role of sending the data to a memory module in the system. At a given time, multiple transfers could be active on different data channels. This thread interleaves those transfers on the single physical PCIe channel, implementing a weighted arbitration scheme that allows to meet the desired QoS levels. For this reason, a write operation that is seen as a whole on the

application layer gets split into many smaller data burst transfers on the backend layer. All the necessary bookkeeping is done through the shared memory transfers table, and abstracted away from the upper software layers. The smaller transfers size can be fine-tuned by modifying a macro inside the daemon source code. The number of smaller transfers also determines the number of completion notifications that are received by the cluster: the dispatcher thread is aware of the expected number of notifications and detect when the transfer as a whole can be considered completed. More details are provided in the Item Dispatcher Thread section. The main steps of the algorithm are here reported.

```
if item FIFO to send not empty {
  dequeue and send the items to the FPGA
}

for each channel taken in weighted order {
  if pending transfers on this channel {
    send sub-transfer header and data to the FPGA
    update transfer info
    break
  }
}

while shared memory command FIFO not empty {
  dequeue command
  pick a channel in a round-robin fashion
  add the transfer to the shared memory transfers table on the
     selected channel
}
```

Backend from FPGA Thread

The *backend from FPGA* thread receives data chunks and items from the cluster.

Every hardware data channel is equipped with a FIFO, so in case of multiple running transfers, the thread must pick data in a way that preserves the bandwidth allocation requirements. For this reason, a weighted arbitration is used to pick among the upstream FIFOs. The main steps of the algorithm are here reported.

```
if item FIFO to receive not empty {
  read items and send them to dispatcher thread
}

for each channel taken in weighted order {
  if channel hardware FIFO not empty {
    read chunk of data from FIFO
    update transfer info

    if we got all expected data {
      mark transfer as completed

      if tranfer is blocking {
        notify completion to client application
      }
    }
```

```
17
18      break
19    }
20  }
21
22  while shared memory command FIFO not empty {
23    dequeue command
24    pick a channel in a round-robin fashion
25    add the transfer to the shared memory transfers table on the
          selected channel
26    launch the hardware DMA transfer on the selected channel
27  }
```

Item Dispatcher Thread

The *item dispatcher* thread processes the items coming from the cluster. It plays an important role in the downstream transfers management, as it is in charge of receiving and handling the transfer completion notifications coming from the cluster. In fact, the only way for the daemon to be sure that all the data has been sent, is to wait for all the notifications to be received by the dispatcher. Every notification carries the channel ID: the dispatcher thus uses it to access the shared memory transfers table and update the corresponding transfer status. In case all the notifications are received, the transfer is marked as completed, and a notification is forwarded to the client if the transfer was marked as blocking.

The relevant algorithm steps are here reported.

```
1  on interrupt reception {
2    if interrupt is downstream data burst completion {
3      extract channel ID from the notification
4      update shared memory transfers table
5
6      if we got all expected notifications {
7        mark transfer as completed
8
9        if transfer was blocking {
10         notify completion to client application
11       }
12     }
13   }
14 }
```

Hardware Data Channels

Data channels are logic lanes onto which data transfers flow. They are independent from one another, meaning that they share no physical resources. It should be noted that to communicate with other tiles various data channels may need access to the router. To guarantee resource isolation, each data channel is mapped on a different virtual network (the first data channel to VN3, and so on...).

As data reaches the cluster through a single PCIe physical connection, the IO device is in charge of demultiplexing the traffic toward the correct data channel. It

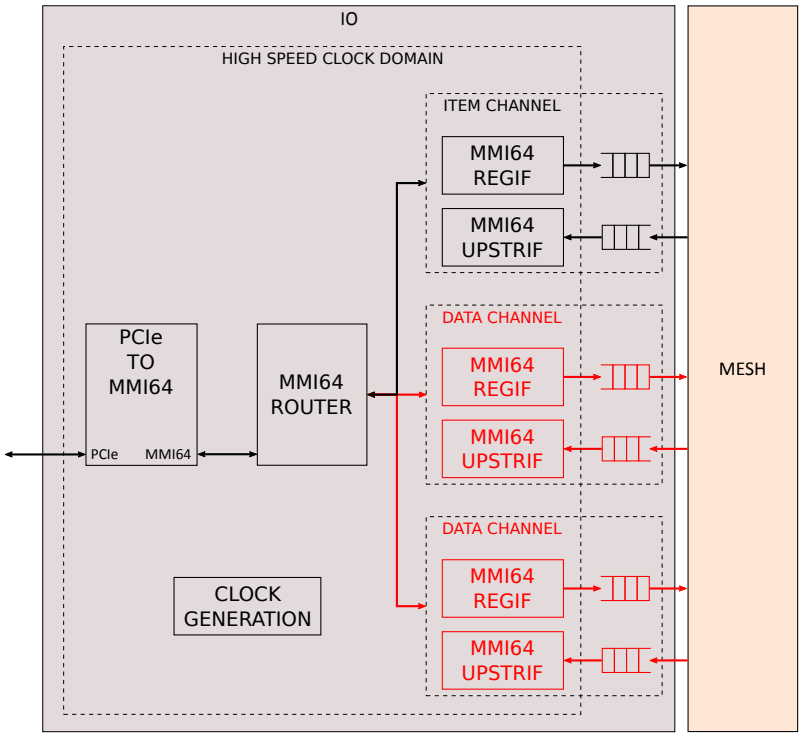


Figure 3.4: IO module

thus abstracts away from the mesh the details about how channels are generated and fed with data.

The channels enter the system from the tile where the IO device is plugged: from there they reach every tile in the system. A data channel can thus target every memory module inside the cluster.

On the other hand, multiple data channels could be targeting the same memory controller. For this reason, an arbitration stage is put in place. For downstream transfers, the arbitration is done as late as possible, inside the *MC from net* module, just before reaching the memory controller. For upstream transfers data is generated using the DMA channels: for this reason an arbitration stage is put just inside the memory controller, before the DMA read requests get sent to the physical memory module. When data from the memory module is ready, the *MC to net* receives and demultiplexes it among the various channels.

IO Module

The IO module is in charge of communicating with the server on one side, and with the mesh of tiles on the other. In particular, it receives the physical PCIe signals from the corresponding connector, and it extracts from there the communication data to inject into the cluster. The structure of the module is reported in Figure 3.4.

The low level communication is handled by a set of IP cores provided by ProDesign. They are clocked at a higher clock frequency to meet the required performance level.

First we have the physical PCIe module. It converts the PCIe interface into an *MMI64* interface, which is a communication protocol once again developed by ProDesign. Then the data gets to an MMI64 router, which has a set of peripherals connected to it. The server, through a specific API, can query the router inside the cluster to report the amount and the type of modules connected to it. Each MMI64 module instance has an associated identifier and an address. Through this addressing feature, the server can send data messages to a specific peripheral. A light MMI64 header is added to the sent payload, that allows the router to forward the message to the correct peripheral, working as a demultiplexer among the various channels.

The MMI64 router acts as a serialization point: from now on downstream traffic flows through parallel channels up to the *MC from net* modules, and the upstream traffic coming from the parallel channels gets serialized and sent to the server. In a similar fashion items flow through the item network interface. Even if they are supported by the IO module, the mesh instantiates just one item channel, as item traffic is usually composed of low priority debug informations, so more resources are left available for data channels.

Each channel is composed of a pair of peripherals: a ***REG**ister **I**nter**F**ace*, used for downstream traffic, and an ***UPSTR**eam **I**nter**F**ace*, used for upstream traffic.

Both modules interact with an asynchronous FIFO: this allows to bring the data from the MMI64 clock domain to the system clock domain, which is used by the system modules to read from the FIFOs.

Two types of channels are supported: item channels and data channels. They are differentiated on the hardware level because of the different interface with the mesh: items are 32 bits wide, while data words are 64 bits wide. For this reason, the FIFOs at the system boundary have different bus width. The clock signal used to read from the FIFOs is also different, as the item network and the data network are run by two different clock domains. The IDs used by the peripherals belonging to the two channel types are in different ranges: in this way the HN daemon can differentiate between them, and know how many channels of each type are present in the system.

Network Interface

The data coming from the IO device gets read by the mesh. The IO device can be seen logically connected to a specific tile into the mesh: this is the point where data enters the system and gets moved to the other tiles, if needed.

Inside the tile, the data is read by the module responsible for the communication: the network interface. The updated structure of the network interface is reported in Figure 3.5. The new modules and wirings are marked in bordeaux.

Additional ports are placed in the network interface to account for the data channels, each interacting with a pair of modules called *EXT to net* and *EXT from net*.

In case of a downstream transfer, data enters from one of the *iodev_data_i* ports, thus reaching the corresponding *EXT to net* module. The *EXT to net* module forwards the traffic from the IO module to the *MC from net* or to the injector, de-

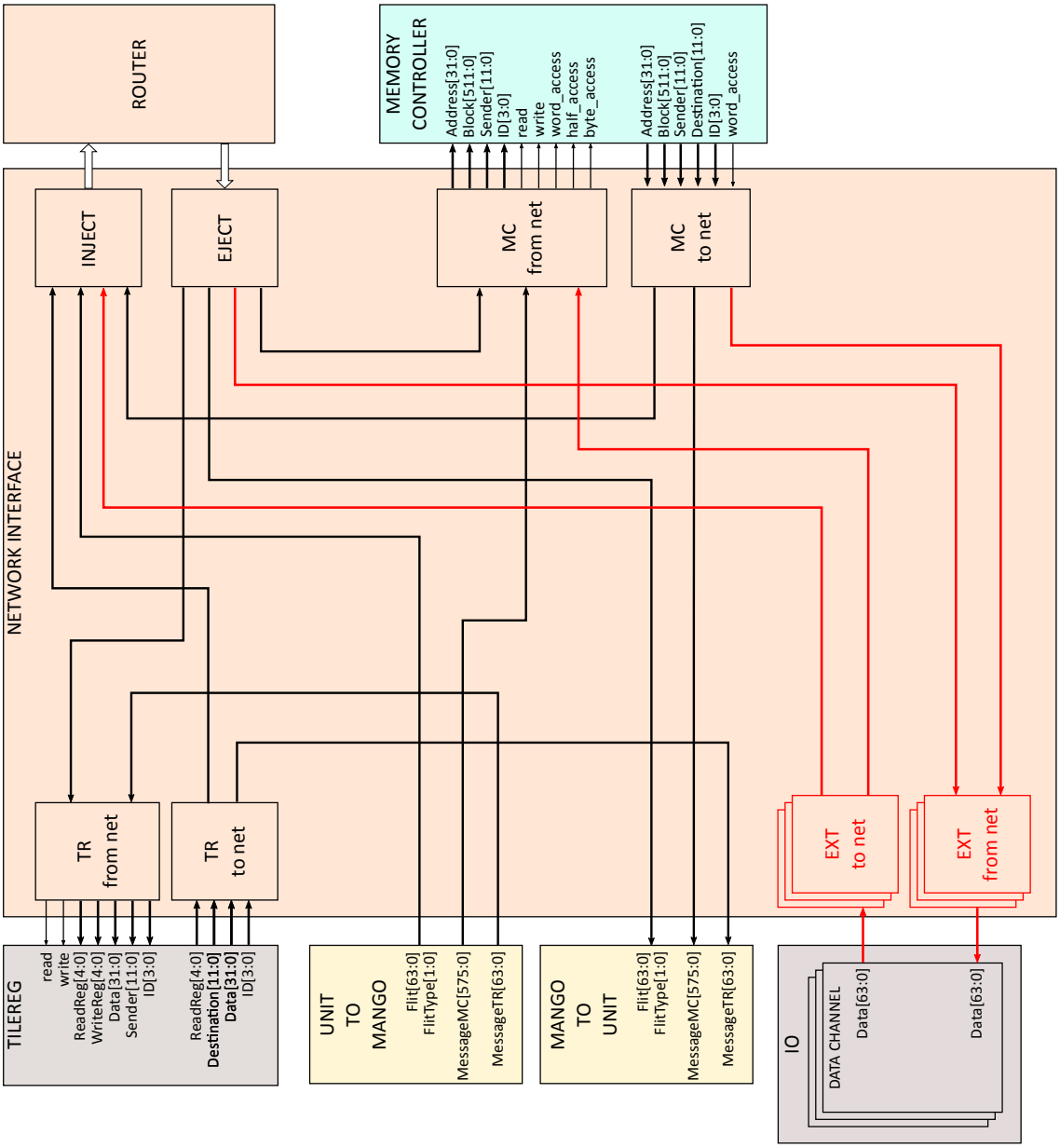


Figure 3.5: Network interface changes

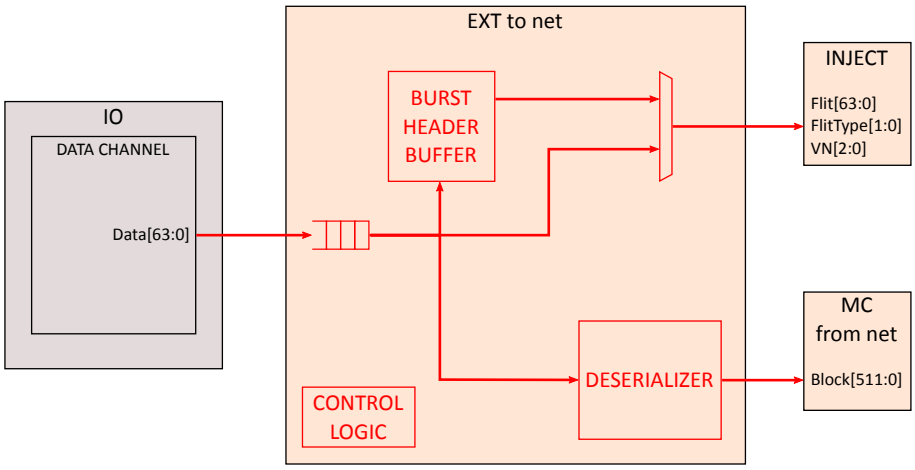


Figure 3.6: EXT TO NET module

pending on whether the destination memory controller is a local or remote one. For remote transfers, data is injected on the virtual network corresponding to the data channel. In the destination tile, the ejector forwards the data to the *MC from net* module, which once again provides a port for each data channel. This module arbitrates between the various data channels, generating a request sequence to the memory controller that takes into account the weights specified by the application.

For upstream transfers, the DMA generates the memory read requests required to generate the data flow coming from the memory controller. This traffic enters the network interface from the *MC to net* module. It analyzes the destination field of the message to determine if it should be delivered to a local or remote IO device. In case the destination is the local IO device, the memory block gets forwarded to the *EXT from net* module, otherwise it is injected on the virtual network associated with the data channel. In the destination tile, the ejector forwards the contents to the corresponding *EXT from net*.

The injector has thus to support an additional number of input ports that equals the number of data channels, to allow the various *EXT to net* modules to inject independently from each other. In the same way, the ejector has to support one output port per data channel, which are used to feed the *EXT from net* or *MC from net* modules.

EXT to net

The goal of this module is to identify the destination of a downstream transfer, thus forwarding the data to the local memory controller or to a remote one through the network. The structure of the module is depicted in Figure 3.6.

The incoming data gets enqueued in a FIFO and its output evolves the state machine inside the module. Although the data word size is 64 bits (8 bytes), data burst transfers have memory block granularity, which are 512 bits (64 bytes). On the other hand, network flits are 64 bits wide. For this reason, the two output ports of the module are asymmetric: if data has to reach the local memory con-

	Fields					
Bit range	97 - 66	65 - 62	61 - 50	49 - 18	17 - 6	5 - 0
Width	32	4	12	32	12	6
Name	Start address	-	Sender	Size	Destination	Command

Table 3.1: Downstream burst transfer header block

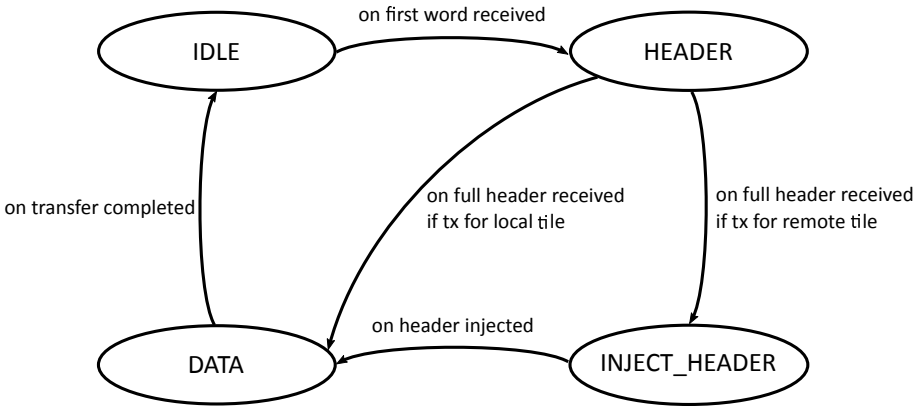


Figure 3.7: EXT TO NET control logic

troller, memory blocks are rebuilt with a deserializer; otherwise data flows as-is to the inject interface.

To determine the destination of a downstream transfer, the module looks at the first incoming block, from now on called *burst header* (not to be confused with the network header). This block contains informations about the incoming data burst and not data meant to be written to memory. It is used by the modules that are aware of the memory transfer, the *EXT to net* and *MC from net*, to handle it correctly. The format of the burst header is reported in Table 3.1.

The header and then the whole burst are processed by a control logic implemented as a finite state machine, whose details are provided in figure Figure 3.7. The first word that is received (that is, the lower part of the burst header) contains the destination tile. From this, the control logic knows where to forward the data.

For local transfers the burst header, and then the burst data, can proceed directly to the deserializer. It should be noted that the burst header is also passed along to the next module, which is the *MC from net*. The reasons are explained in the next section.

For remote transfers, the control logic has to first generate a network header flit for resource allocation inside the router. The format used is the one reported previously in Table 2.6, where the command field has the value *EXT_TO_MEM*. It should be noted that the header flit cannot be generated after the reception of the burst header's first word. In fact the destination address, that should be included in the network message, is placed in the burst header's second word. Thus the burst header is first rebuilt into an internal buffer, then the network header is generated and injected, followed by the buffered burst header. This introduces a small overhead (to be precise, an 8 clock cycles overhead needed to buffer the burst header), which becomes negligible when compared to the length

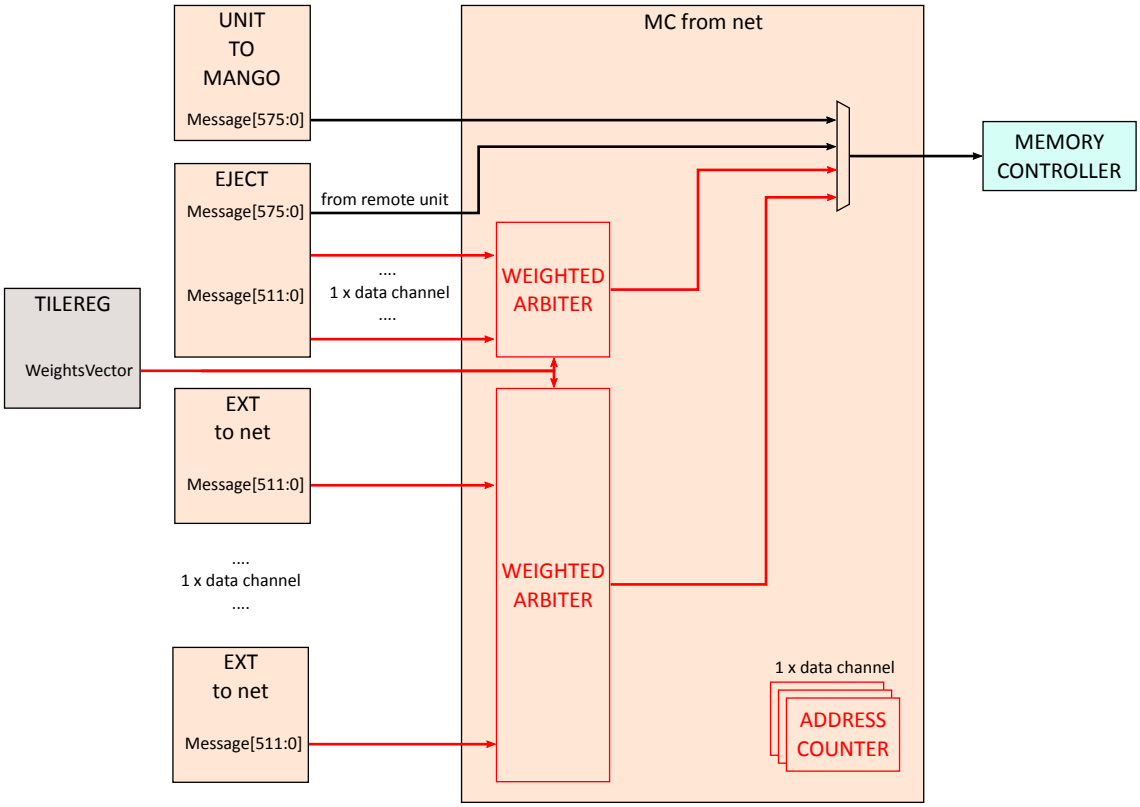


Figure 3.8: MC FROM NET module

of the burst transfer. The virtual network associated to the module is set by the network interface at design time.

The burst header is also used to extract the size of the burst. The *EXT to net* is thus able to detect the end of a transfer and generate the appropriate transfer completion signal. It is routed up to the item network interface, where an interrupt item is built and sent upstream. In case of remote transfers, the last flit must also be marked as a tail flit: this allows the routers along the path to release the resource allocated for this virtual network.

MC from net

The *MC from net* already provided memory access to the unit. The required modifications are depicted in Figure 3.8. New ports are added to account for burst transfers, which can come from the network or from the local tile.

The first block received on a data channel is always a burst header, with the format described previously. The header's field of interest is the destination address. The *MC from net* saves and updates it on every received block, thus allowing to save the bandwidth required to send the address for every block. It decrements the transfer size until it reaches 0, which means that the burst transfer is finished.

An arbitration stage decides which data channel is allowed to write to memory. The weight should be here enforced to guarantee the required write bandwidth to each data channel.

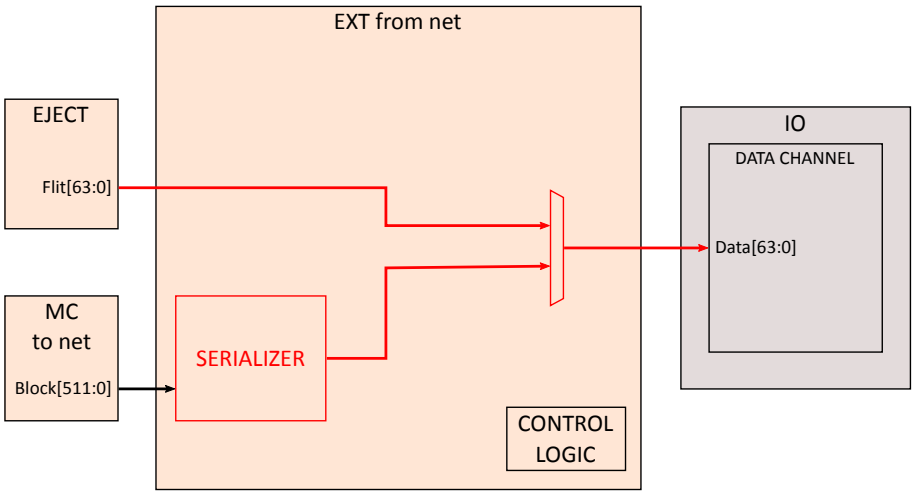


Figure 3.9: EXT FROM NET module

EXT from net

This module has the responsibility of sending to the IO device the data, which can come from a local or remote memory controller. The structure of the module is depicted in Figure 3.9.

The bus with the IO device is 64 bits wide, thus requiring a deserialization stage if data is coming from the local memory controller. In the other case, data can be directly forwarded to the IO device.

MC to net

The *MC to net* module delivers memory blocks coming from the local memory controller to the destination module, injecting it through the network if needed. The structure is reported in 3.10.

In the case of an upstream burst transfer, the DMA module submits the read requests to the memory. As a DMA channel is implemented for each data channel, if multiple upstream transfers are targeting the same memory the channels compete for memory access. Then the *MC to net* module receives the read results, and has to extract informations to understand if it is an upstream burst transfer, and from which DMA channel the request came.

The transaction information returned by the memory controller are used to detect burst transfers. In particular, the destination field of the transaction is used to detect if the memory block is meant to be delivered to an *EXT* module. The ID field of the transaction, which is used to identify memory reads when they are generated from a unit module, is used to save the DMA channel from which the request was made. The *MC to net* module thus analyzes those fields and acts accordingly.

If the data is meant for a local *EXT* module, the memory block is forwarded as-is. If the block has to be injected, it is first deserialized into flits. For each memory block a header flit is generated with the transaction information. The

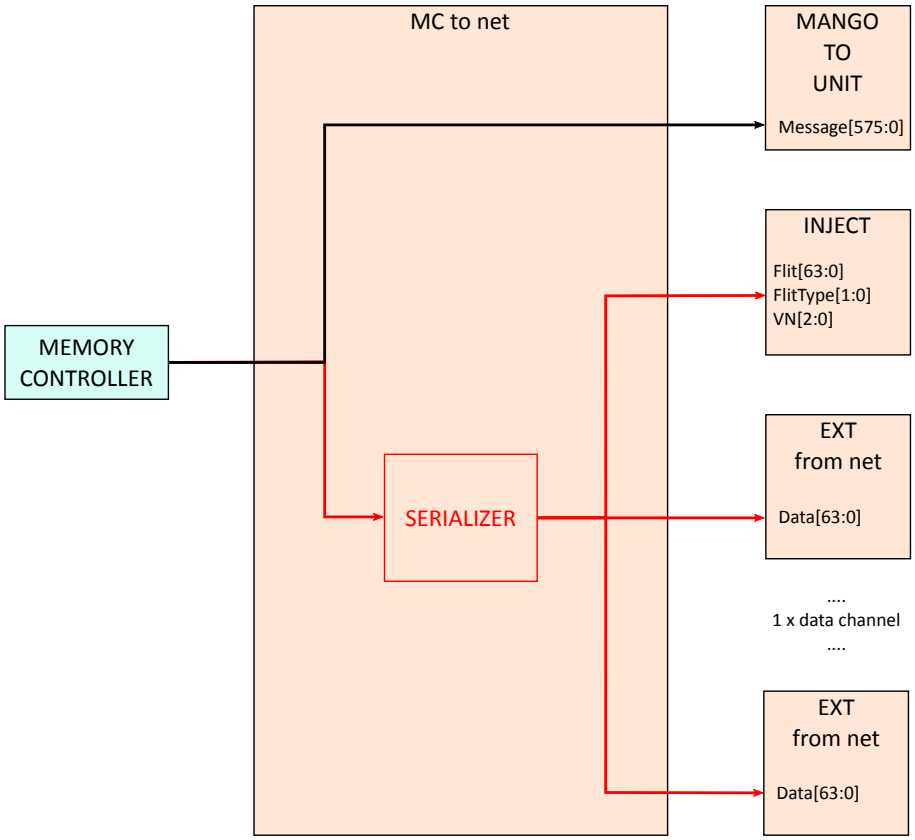


Figure 3.10: MC TO NET module

virtual network is chosen accordingly to the DMA channel that generated the request.

Injector

The injector supports a parametrised number of input sources, where each input source can inject on a different virtual network. The internal structure is depicted in Figure 3.11.

Both these functionalities are exploited for data burst transfers. For downstream transfers, a variable number of *EXT to net* modules are connected to the injector, each injecting on the virtual network associated to its own data channel. For upstream transfers, the *MC to net* module receives and injects data coming from the local memory controller, where each block can be the result of different DMA channels' requests, and thus has to be injected on a different virtual network.

To guarantee that the bandwidth reservation requirements are preserved, a weighted arbitration scheme is implemented inside the switch allocator.

Ejector

The ejector should forward the network messages to the corresponding local module. The structure of the module is reported in Figure 3.12. It is built of

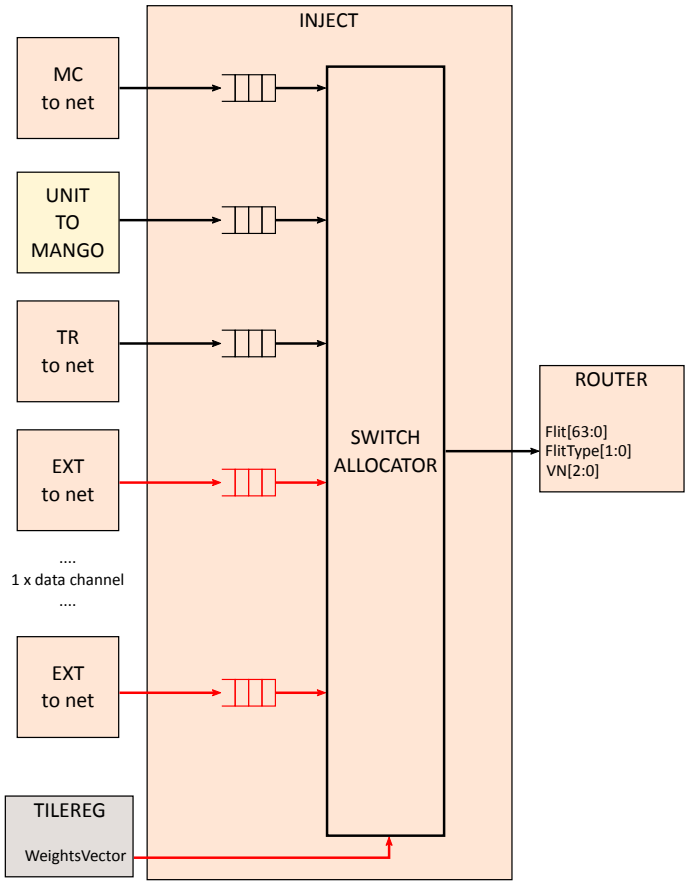


Figure 3.11: INJECT module

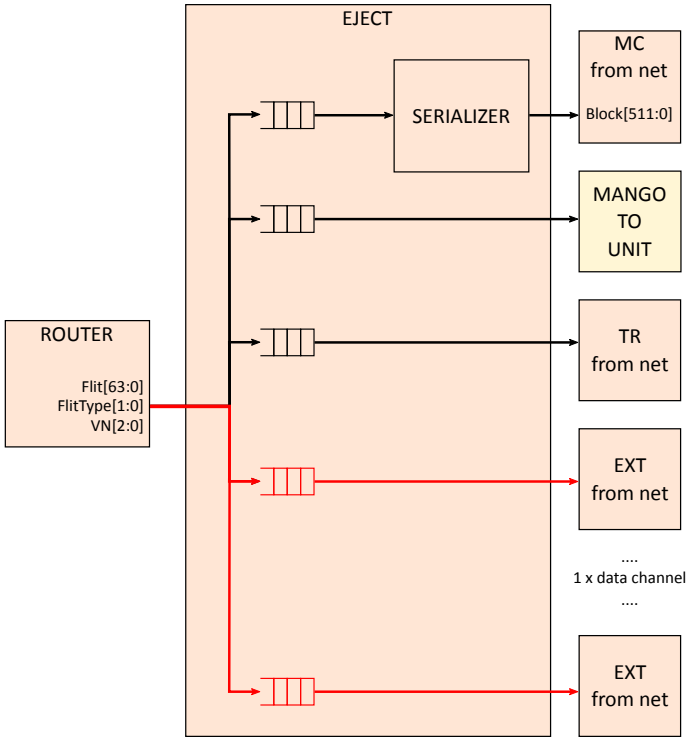


Figure 3.12: EJECT module

parallel lanes, one for each virtual network, and each lane forwards messages to one or more local modules.

The first lane, associated with virtual network 0, receives and rebuilds requests for the local memory controller. The deserialization process produces a 572 bits wide request, which is forwarded to the *MC from net* module.

The second lane, associated with virtual network 1, receives messages for the local unit. The messages meant for the unit are passed first through the *MANGO TO UNIT* module, which can receive the raw flits coming from the network. So the ejector just forwards the flits to the next module.

The third lane, associated with virtual network 2, receives messages for the local TILEREG. Raw flits are passed to the *TR from net* module.

Then there is a lane for each additional virtual network supported, which means one lane per data channel. After reception on one of these channels, the transfer type should be identified: downstream traffic has to be forwarded to the *MC from net*, while upstream traffic has to go to the *EXT from net* module. The demultiplexing is done analyzing the command field of the header flit. Then the channel is assigned to a specific output module until the tail flit is received and the assignment gets lost. A deserializer is also used on the downstream path to rebuild blocks meant for the local memory controller, while raw flits are forwarded through the upstream path.

DMA

The DMA module is modified to provide a convenient way to move memory segments, which is accessible both to the units and to the server applications. The target of a DMA transfer can be another memory in the system, a unit, or an IO device. To support several concurrent transfers, the DMA implements a number of DMA channels which are fully independent and that inject requests to the memory controller in parallel. It is the memory controller's responsibility to implement an arbitration policy. The structure of the module is reported in Figure 3.13.

The DMA programming interface is meant to be used through the TILEREG module. It accepts a series of configuration words coming from a 32 bits wide port, which matches the size of a TILEREG register. The configuration words include information about the channel to be selected, the source address, the destination address if data is transferred to another memory module, the transfer destination module (unit, memory controller, IO device) and the number of blocks to move. Once the last configuration word is received, the channel starts to generate requests. In a similar way, an availability signal is generated for each channel, and they are mapped on another TILEREG register.

The server can thus use the DMA interface by sending a write command to the TILEREG through the item network, targeting the DMA programming register. In a similar way, this register can be mapped in a unit's address space, allowing the unit to access it like a memory mapped peripheral.

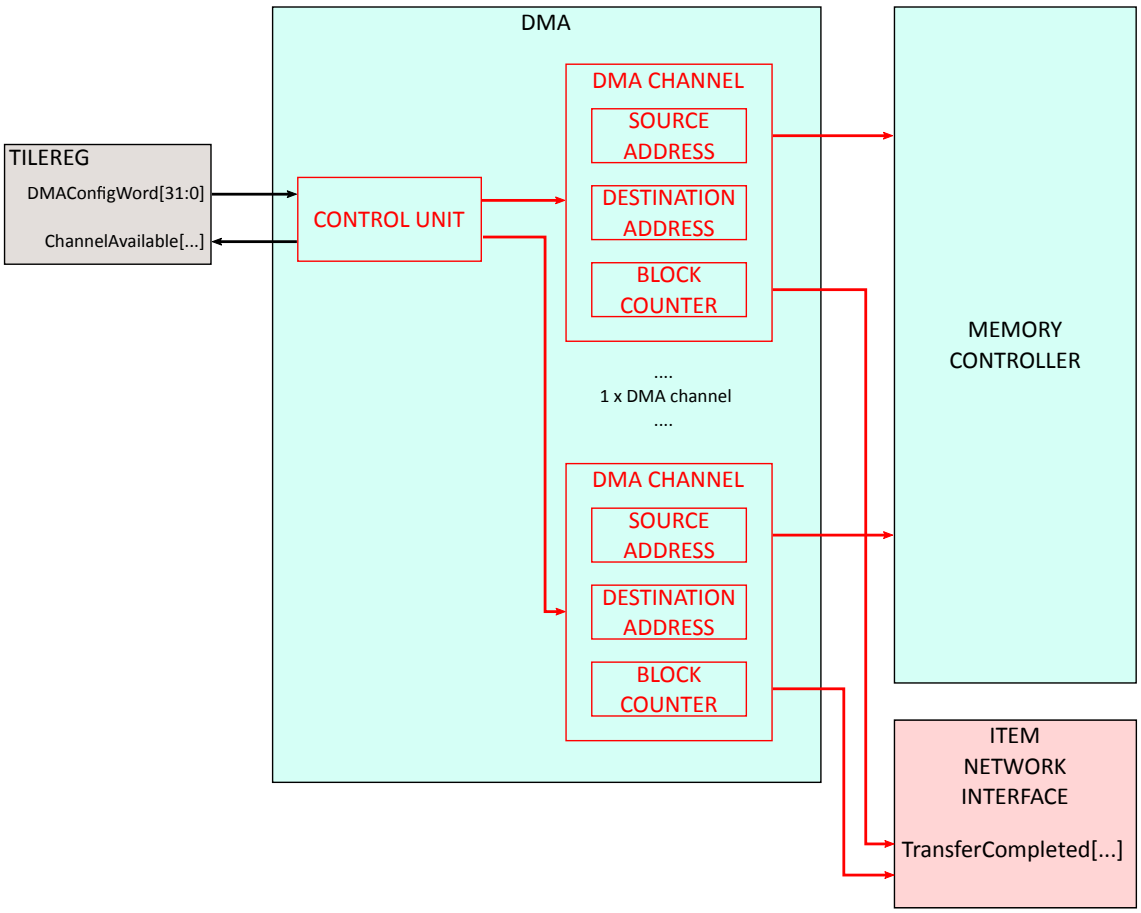


Figure 3.13: DMA module

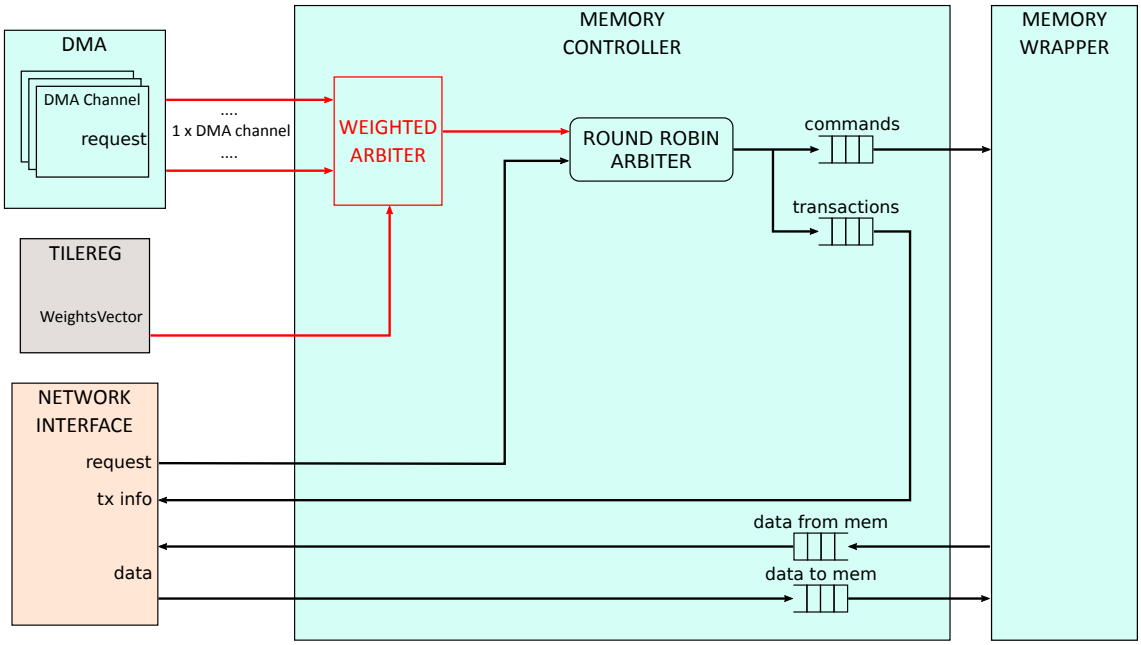


Figure 3.14: Proposed memory controller architecture

A transfer completion signal is provided on a per-channel basis. It is routed up to the item network interface, where an interrupt item is generated and sent to the server.

The memory controller annotates the channel number of each request, and forwards this information to the network interface. If the data should reach an IO device, that is, if the DMA requests are part of an upstream burst transfer, the DMA channel number determines the data channel onto which data is sent. For this reason a one to one mapping exists between DMA channels and data channels.

Memory Controller

The memory controller collects and serves memory access requests coming from two sources: the network interface and the DMA. Its interface has been already described previously; here only the required modifications are discussed. In Figure 3.14 the new module structure is reported.

Support is added for multiple DMA channels. A weighted arbitration stage grants access to one channel at a time, ensuring that the bandwidth allocation requirements are met. The winning channel then competes with the network interface for memory access: here a round-robin policy is applied. It should be noted that in case of parallel downstream and upstream transfers, the round robin stage distorts the applied weights, leading to a violation of the QoS constraints. A possible solution, although not implemented, would be to move the downstream transfers arbitration from the network interface to the memory controller: here downstream and upstream requests would be grouped by channel, and the arbitration stage could be properly applied on a channel basis, thus weighting the memory accesses among different channels and meeting the QoS constraints.

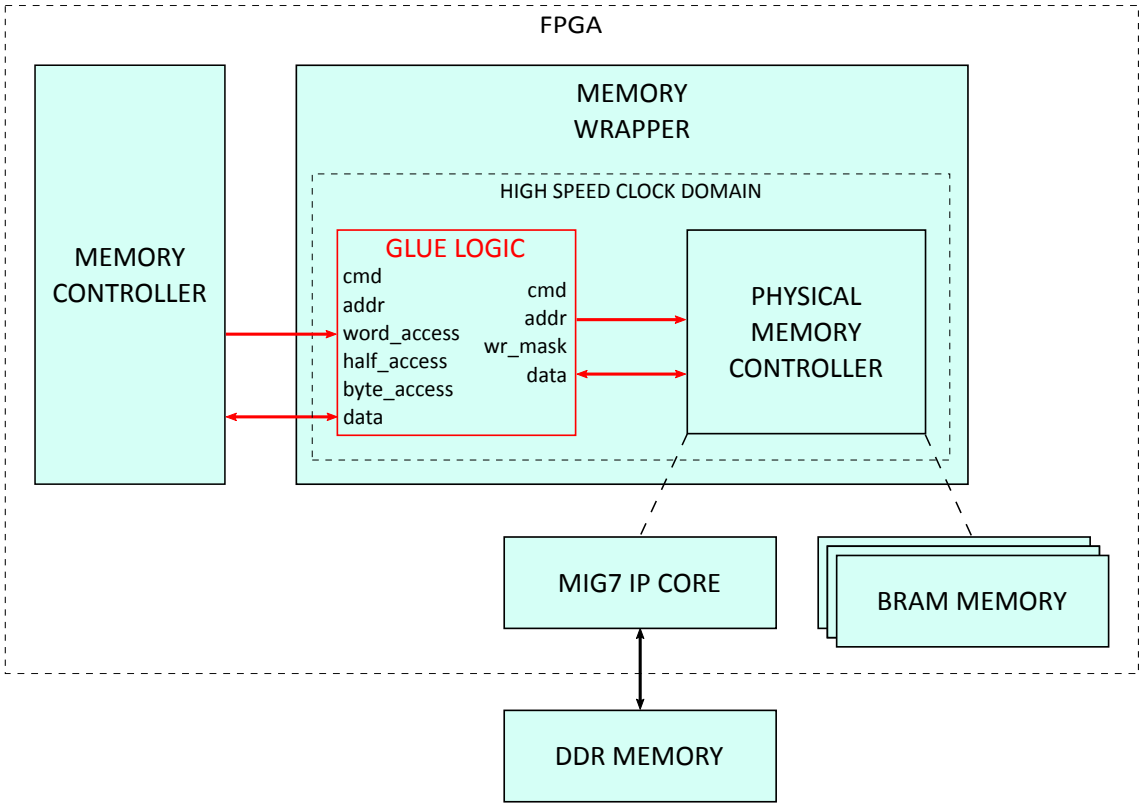
The winning DMA channel number is saved into the transaction information FIFO. When data comes from the memory wrapper, transaction details are dequeued from the FIFO and sent to the network interface. The channel number is sent in the transaction ID field, which is just used for unit memory access and would otherwise be not used in DMA memory transfers.

Memory Wrapper Redesign

The memory wrapper, already described previously has been redesigned to aim for performance improvements. The old implementation relied on a traffic generator module provided by Xilinx connected through an AXI bus to the MIG IP core, who handled the physical connection with the memory. The traffic generator waits for the current memory access completion before submitting the next one.

The MIG module provided by Xilinx can apply various optimizations to the memory access pattern to ensure better performance. In particular, memory access reordering can significantly improve DDR memory bandwidth. But the traffic generator behaviour prevents any optimization from being applied.

The memory wrapper is thus redesigned as depicted in figure Figure 3.15. The traffic generator together with the AXI wrapper around the MIG module have been removed. The MIG module thus offers what is called the *native* interface, which allows to exploit the maximum performance offered by the memory module. A light adapter logic is placed between the memory wrapper interface and the MIG interface, that doesn't introduce any clock cycle latency. The overall structure is thus simpler while allowing to achieve higher levels of performance, as analyzed in the next chapter.


FPGA
MEMORY CONTROLLER
MEMORY WRAPPER
HIGH SPEED CLOCK DOMAIN
GLUE LOGIC
cmd
addr
word_access
half_access
byte_access
data
cmd
addr
wr_mask
data
PHYSICAL MEMORY CONTROLLER
MIG7 IP CORE
BRAM MEMORY
DDR MEMORY

CHAPTER 4
Performance Analysis

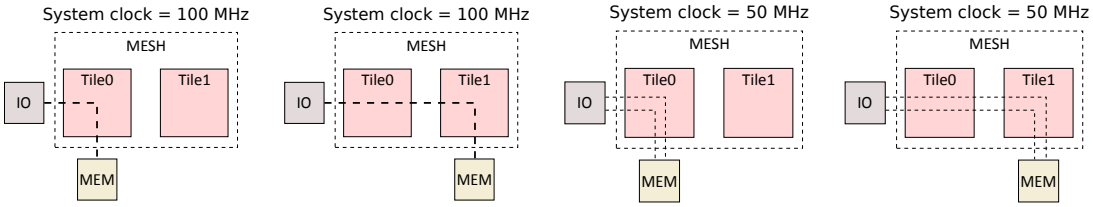
The lead design goal in the implementation of the documented changes has been to exploit the full system bandwidth of the MANGO platform. For this reason every module involved in memory transfer operations underwent an in-depth analysis to ensure that maximum throughput is achieved. As the system is completely parametrisable, the maximum reachable bandwidth depends on the details of the specific architecture being implemented.

The hardware and software tunable parameters that affect the transfer performance are now listed:

- **clock frequency**. The clock frequency used inside the mesh determines the throughput of the system. As the architecture is implemented on FPGAs, the clock speed mostly depends on the complexity of the hardware being implemented compared to the available physical resources;
- **relative position of the IO device and the memory controller**. If the data flowing through the IO device has to cross the network to reach another tile, the bandwidth could be reduced, also depending on the transfer type. More details are discussed further;
- **downstream transfer chunk size**. A downstream transfer gets split inside the HN daemon into smaller transfers, to allow them to be properly time multiplexed on the single PCIe link. As every PCIe transfer has an initial setup latency, the goal should be to minimize the amount of transfers that are launched. At the same time, setting a bigger chunk size could affect the performance of smaller transfers. An optimal trade-off can only be found by analyzing the specific application workload.

In the next sections performance is evaluated, both in case of a single running transfer and of multiple running transfers with bandwidth reservation requirements.

For the purposes of this tests, four hardware architectures have been synthesized, which differ in the placement of the memory module, the number of implemented data channels and the clock frequency. Besides that, all the architectures share the remaining configuration. They are composed of two tiles, with no unit module inside. The tiles are implemented on the same FPGA, which is a Xilinx



(a) L1 **(b)** R1 **(c)** L2 **(d)** R2

Figure 4.1: Hardware architectures for performance evaluation

Virtex-7 2000 with PCI express support and a 2 GB DDR3 memory attached. The weight vector, used for bandwidth allocation, is composed of ten elements. This allows to allocate the bandwidth with a 10% resolution. The various hardware configurations are depicted in Figure 4.1, where the dashed lines represent data channels. To account for the physical hardware limitations, when two data channels are implemented, the clock frequency must be lowered to meet the timing constraints of the design.

The data flows between the IO module and the mesh. The bandwidth on this interface, called from now on *IO bandwidth*, is an upper bound on the achievable bandwidth for a burst transfer. The interface is clocked with the system clock, and the bus width is 8 bytes (64 bits). We can deduce the following formula to calculate it.

$$IO\ Bandwidth = Clock\ Frequency * 8\ B/s$$

Then also the *network bandwidth* could lower the efficiency of a memory transfer that has to reach a remote tile. However, as the flit size is also 8 bytes and the network clock speed is the same, the IO bandwidth and the network bandwidth are the same. From now on they are referred to as *system bandwidth*. Depending on the data path structure, additional latencies can be introduced, which lower the effective bandwidth. In the next sections the downstream and upstream transfers data path are analyzed in-depth to identify the possible source of latency.

The *transfer bandwidth* is reported as perceived from the HN daemon, calculated as the size of a memory transfer divided by the time elapsed between the reception of the request and its completion. The completion of a downstream transfer is notified with an interrupt item, while an upstream transfer is considered complete when all the data has been received by the daemon.

Downstream Memory Bandwidth

Downstream transfer bandwidth is evaluated for local and remote transfers with just one data channel implemented. So architectures **L1** and **R1** are used. As the clock frequency is 100 MHz, the system bandwidth is equal to 800 MB/s.

The transfer bandwidth is affected by additional software and hardware overheads introduced between the time when the transfer is requested and the time when the hardware notifies the transfer completion. The list of relevant operations that are executed after receiving a downstream data burst request is here

reported. To minimize the overhead related to the PCIe transfer setup, the transfer is sent as a whole and not split into smaller chunks.

1. The first free channel is assigned in a round robin fashion (in this case the only one implemented);
2. the shared memory transfers table is updated;
3. the *start time* is registered;
4. the weighted arbiter selects the next transfer to process (in this case we have just one transfer);
5. the burst header and then the whole data are sent and the transfer is marked as finished;
6. when the dispatcher thread receives the completion notification, the *end time* is registered;
7. the bandwidth gets calculated as the size of the transfer divided by the difference between the two registered times.

The most noticable overheads are expected to be present on step 5 and step 6. In the first case, the PCIe transfer should be initialized. The PCIe driver could add a non negligible latency to the transfer setup. In the second case, the notification travels through the item network, then gets enqueued in the IO module until a PCIe transfer is started on the hardware side and the notification is sent to the daemon.

Moreover, for the reasons explained in section 3.3.2, an 8 clock cycles latency is introduced when a remote transfer is started. Also, for both remote and local transfer the burst header is sent as the first data block, incurring an additional 8 clock cycles penalty. However, this overhead is negligible compared to the amount of data blocks transferred.

In Figure 4.2 the measured bandwidth is reported. The dashed line represents the system bandwidth, that is, the upper bound on the achievable bandwidth. The measurements show that there is almost no dependency between the transfer size and the transfer bandwidth, which amounts to approximately the 97.5 % of the system bandwidth.

The downstream data path doesn't offer much room for improvements. A refactoring of the burst header format can allow to save the 8 clock cycles latency for remote transfers, but still the impact would be negligible. The PCIe transfer setup latency cannot be overcome in the existing architecture as the physical communication is handled by a set of proprietary IP cores. The IRQ transmission time represents a price that must be payed to be sure that data has reached the destination memory and is not still inside some buffers along the communication data path.

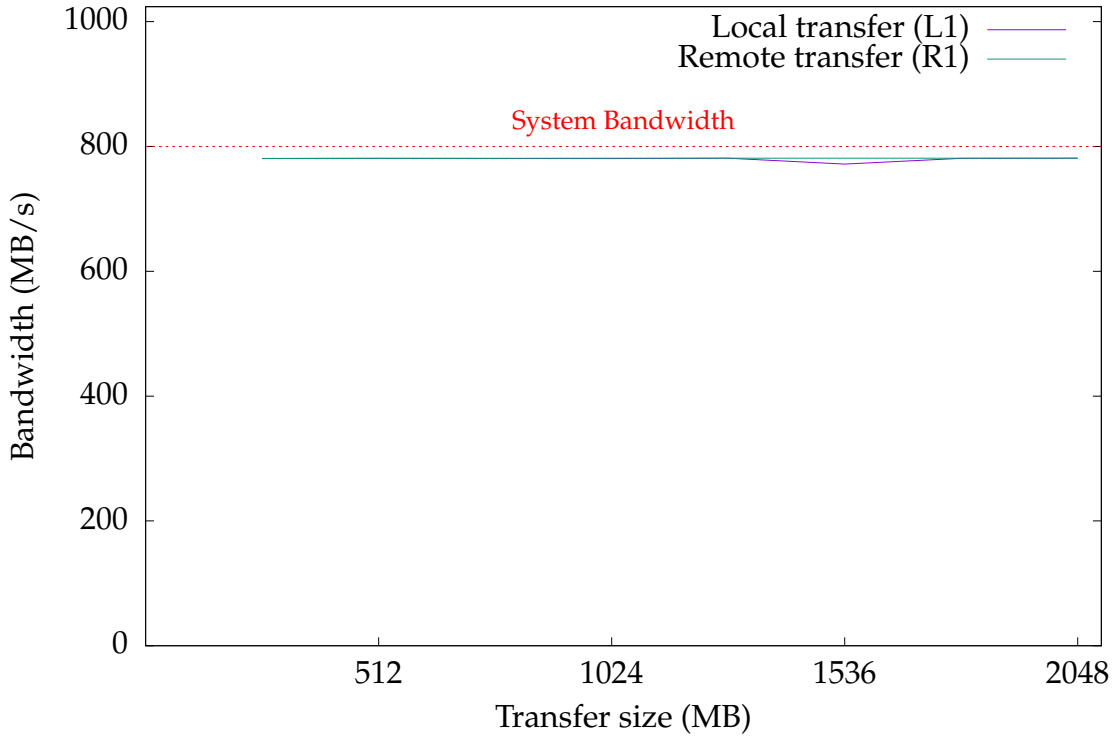


Figure 4.2: Downstream memory bandwidth per architecture

Upstream Memory Bandwidth

Upstream transfer bandwidth is also evaluated on architectures **L1** and **R1**, that offer a system bandwidth of 800 MB/s.

The list of operations that are executed after receiving an upstream data burst request is here reported:

1. the first free channel is assigned in a round robin fashion (in this case the only one implemented);
2. the shared memory transfers table is updated;
3. the *start time* is registered;
4. the DMA transfer is configured on hardware;
5. the daemon start to receive the data from the cluster, writing it into the shared memory;
6. when the daemon receives all the requested data, the transfer is marked as finished and the *end time* is registered;
7. the bandwidth gets calculated as the size of the transfer divided by the difference between the two registered times.

The main source of latency is introduced in step 4. In fact, to program a DMA transfer some items need to be sent to the cluster over the low-performance item

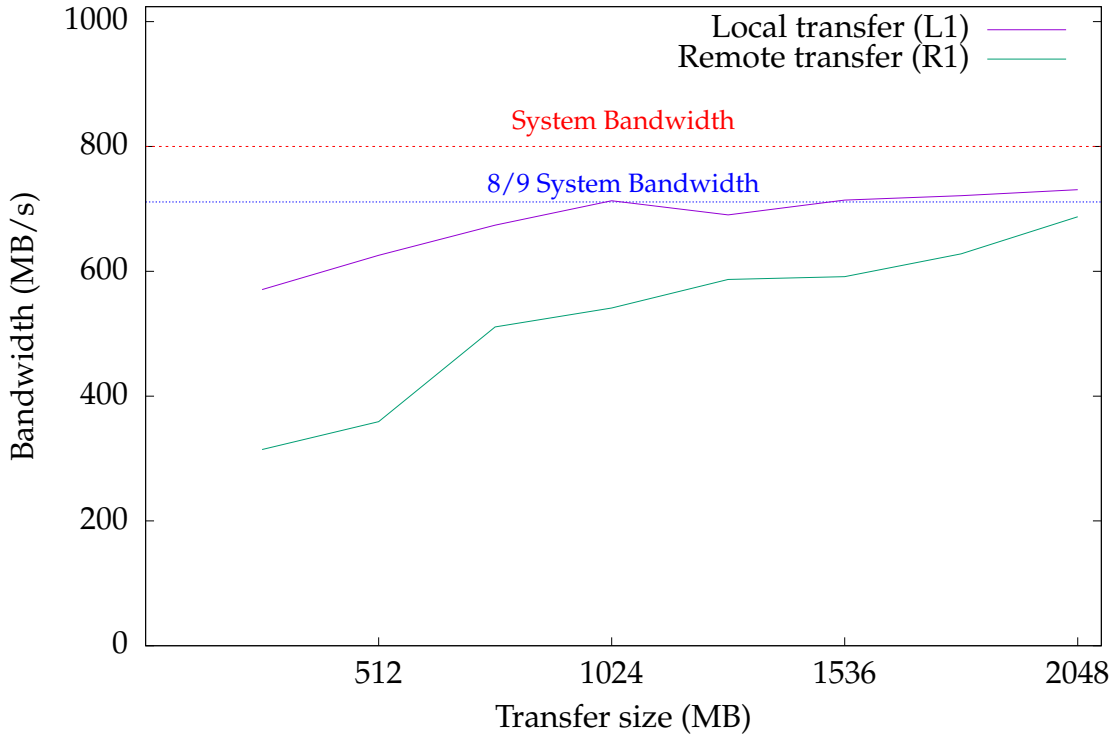


Figure 4.3: Upstream memory bandwidth per architecture

network. Once the items reach the destination tile, the DMA transfer is started and data start to flow from the memory controller. This is expected to impact significantly on the bandwidth perceived by the daemon. However, as this latency doesn't depend on the transfer size, the bandwidth should improve as the transfer size increases.

For remote transfers, another significant overhead is introduced. In fact, as described in section 3.3.2, a header flit is sent to the network for every memory block transferred, which means paying 1 flit every 8 flits of useful data. For this limitation, the maximum bandwidth that can be reached with a remote transfer is 88.8 % of the system bandwidth.

In Figure 4.3 the measured bandwidth is reported. The red line represents the system bandwidth, while the blue one identifies the 88.8 % threshold of the system bandwidth. As expected, a strong dependency is observed between the transfer bandwidth perceived by the daemon and the transfer size. The impact of the DMA transfer setup gets smoothed as the duration of the transfer increases. The remote transfer performances approach the maximum value, while the local transfer performances amount to 91.2 % of the system bandwidth.

Performances of the remote upstream data path could be further enhanced. The header flit could be sent just when the transfer is started, as it happens on the downstream data path. This would allow to remove the 88.8 % bandwidth limitation. The other main source of latency, the DMA transfer setup, is a price that cannot be avoided.

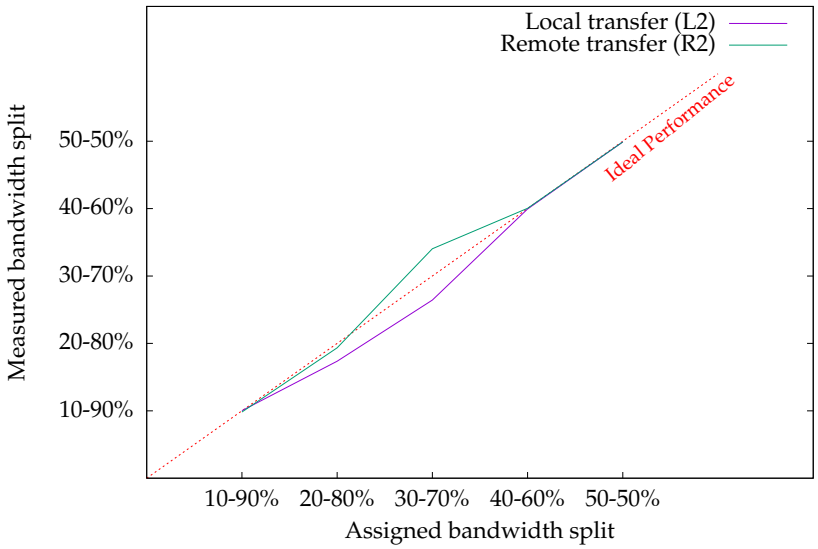


Figure 4.4: Downstream bandwidth reservation per architecture

Bandwidth Reservation

Bandwidth reservation is evaluated on architectures **L2** and **R2**, which implement two data channels thus allowing to run two concurrent transfers, in this case reaching the same memory module, whose position change between the two architectures.

The presence of two data channels forces to lower the clock frequency compared to the one channel implementation. Architectures **L2** and **R2** support a 50 MHz clock signal, thus offering a 400 MB/s system bandwidth.

The downstream chunk size has been set to 50 MB: the concurrent transfers are split into sub-transfers of 50 MB each and time multiplexed on the physical PCIe channel.

As the weight vector supports 10 elements, the bandwidth can be assigned to channels with a 10 % granularity. In the tests here reported, the 2 GB memory plugged into the system is fully written by two transfers. The sizes of the transfers are chosen in a way that, given a specific bandwidth reservation, should allow them to finish at the same time. This means that the size of each transfer is proportional to the corresponding channel's bandwidth, and the sum of the two is equal to 2 GB.

The result of the downstream tests are reported in Figure 4.4. The dashed line plots the expected performance. The measured performances are then plotted. Corresponding to the 30 - 70 % split, the maximum distance between the two curves is observed. In particular, the local transfer is 3.7 % far from the expected performances, while the value for the remote transfer differs of a 4 % from the ideal curve.

The result of the upstream tests are reported in Figure 4.5. Once again, the dashed line plots the expected performance. The measured performances are extremely close to the ideal ones.

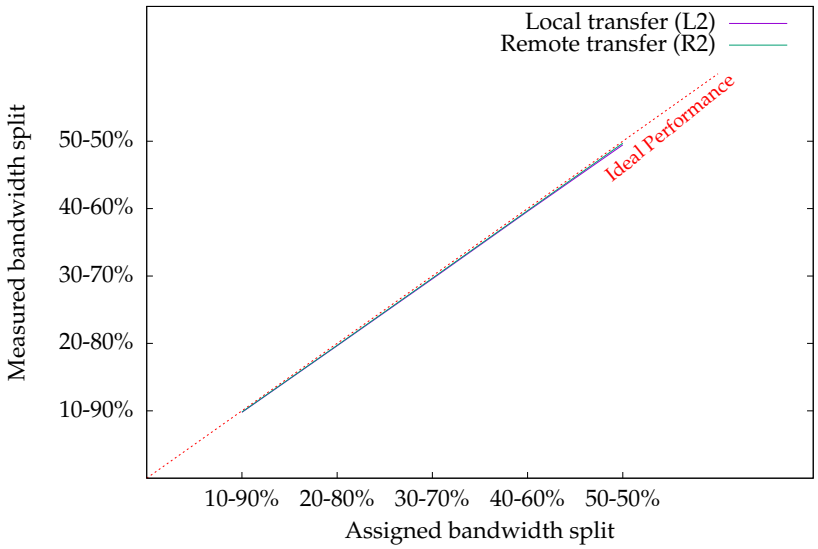


Figure 4.5: Upstream bandwidth reservation per architecture

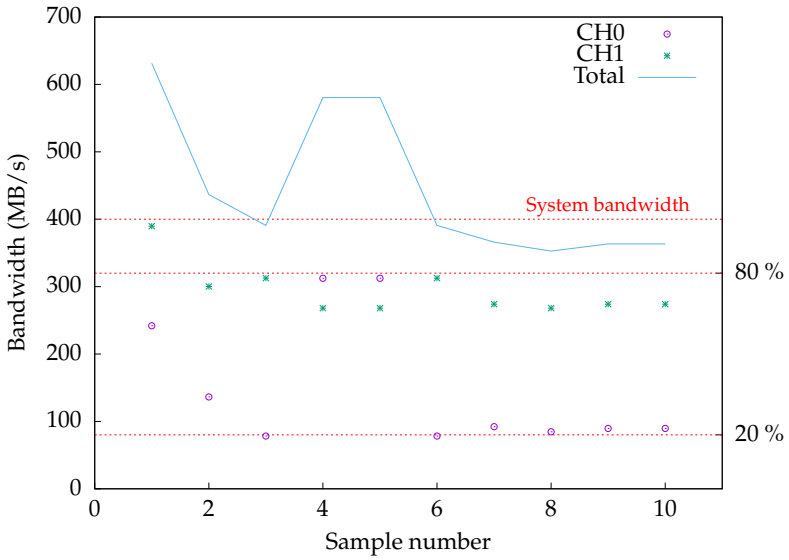


Figure 4.6: Measurement samples for a 20 - 80 % split

The bandwidth reservation measurements are affected by a great amount of dispersion. A few measurement samples, taken with a 20 - 80 % downstream split are reported in Figure 4.6. For some readings, the sum of the measured bandwidth for the two transfers is greater than the total bandwidth of the system. The cause of this can be traced back to the asynchrony between the transfers start time. As the transfers are launched by different applications (or different threads inside the same application), the difference between the transfer start times can be non-negligible and non-predictable.

An artificial example is reported in Figure 4.7. The example scenario assumes a total transfer bandwidth of 100 MB/s onto which a 20 - 80 % split should be applied. The two transfers are meant to run for a total of 10 seconds, so they are moving respectively 200 and 800 MB. When a latency is added between the start times of the two transfers, we can see that the bandwidth perceived by the

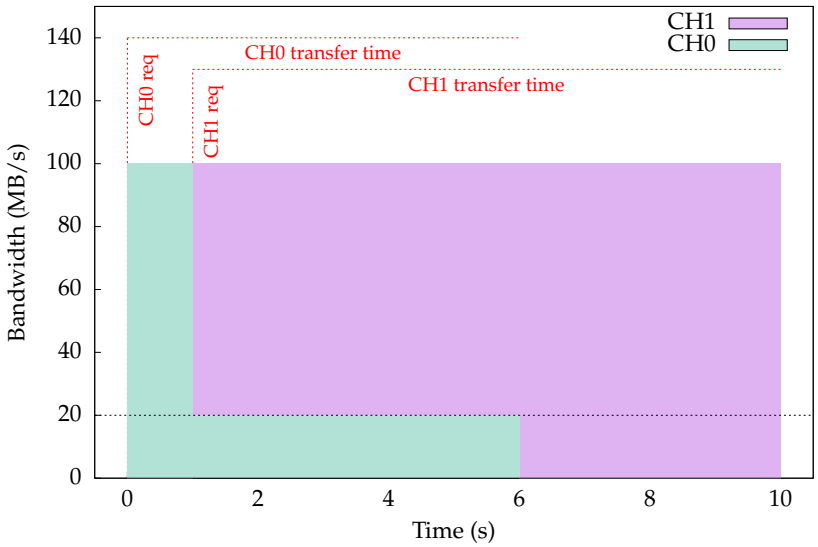


Figure 4.7: Effective downstream bandwidth for a 20 - 80 % split

Configuration	*Elapsed time (ns)*	*Bandwidth (MB/s)*
Old	1643420	297.11
New	44865	10883.34

Table 4.1: Memory wrapper performance on a 500 KB read transaction

application gets distorted. In particular, the application sending on Channel 0 is moving 200 MB in 6 seconds, with a corresponding bandwidth of 33.3 MB/s, or 33 % of the total bandwidth. In a similar way, the application allocated on Channel 1 perceives a bandwidth of 88.8 MB/s, or 88.8 % of the total bandwidth. The sum of the two bandwidths, which equals to 122.1 MB/s, also exceeds the total link bandwidth. This phenomenon does not pose any problem on the bandwidth reservation goal. In fact, both applications' QoS requirements are fulfilled.

Memory Wrapper Bandwidth

The performance of the proposed memory wrapper are here evaluated. The module is isolated from the rest of the MANGO system, which would otherwise limit the achievable bandwidth. The module is tested with an ad-hoc testbench, where a sequential memory access pattern is requested. The simulated memory model is a DDR3 memory with a 64 bit data bus clocked at 800 MHz, thus offering up to 11.92 GB/s of memory access bandwidth.

The simulation consists of a sequential memory read access pattern, moving in total 500 KB of data. The bandwidth is then measured as the difference between the time the first request is sent to the memory wrapper and the time when the last chunk of data is returned. The results of the simulations are summarized in Table 4.1. The new configuration offers approximately a X36 speedup on the bandwidth offered by the memory wrapper.

	IO		*NI*		*MC*		*DMA*		*MW*	
Data channels	*LUTs*	*FFs*	*LUTs*	*FFs*	*LUTs*	*FFs*	*LUTs*	*FFs*	*LUTs*	*FFs*
Old architecture	2894	2939	1811	4621	2756	1770	147	123	374	671
1	3186	3378	4933	8772	2699	1857	131	149	70	0
2	4608	4993	9622	1875	2734	1860	253	263	-	-
3	5753	6595	12723	4959	2752	1862	377	377	-	-
4	7035	8205	16896	8037	2799	1864	494	491	-	-

Table 4.2: Area utilization per module

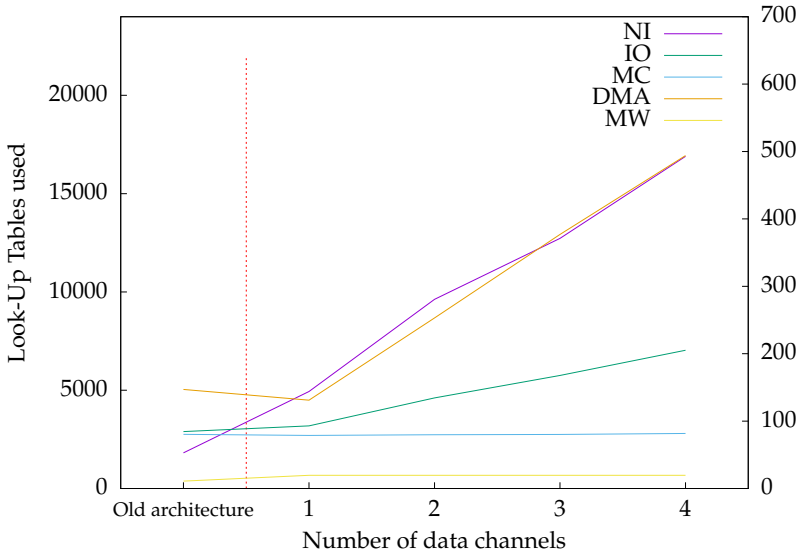


Figure 4.8: Look-Up Table utilization per module

Area Utilization

As the target hardware platform of the design is an FPGA, resource utilization is a factor that should be carefully evaluated, to allow the design to keep meeting the design and timing constraints. On FPGA digital components are synthesized using Look-Up Tables (LUTs) and Flip-Flops (FFs), grouped in logical cells called slices: they thus represent the main source of resource starvation. LUT and FFs utilization is reported in Table 4.2 for all the components that have been modified, while Figure 4.8 plots the number of LUT against the number of data channels for an easy visual analysis.

The internal structure of the modules is pretty regular: each data channel instantiation requires approximately the same amount of resources. For this reason the LUT utilization growth is linear. The network interface shows a growth of a factor of 2.7 in resource utilization between the old architecture and the single data channel configuration, and in the four data channels case the growth factor amounts to 9,3 compared to the old architecture. Indeed, the network interface is where most of the changes are placed.

The IO module shows a weaker dependency on the number of data channels. The most resource demanding parts of the IO module are the physical PCIe adapter and the MMI64 router, then for each data channel a register interface and

a upstream interface modules are instantiated, which give a smaller contribute to area utilization.

The DMA controller's resource utilization has a very smooth and clear dependency on the number of data channels. In fact, the number of data channels determines the number of implemented DMA channels. Then the control logic is barely unmodified in presence of more DMA channels: it just receives the control words from the TILEREG and forwards them to the selected channel.

On the other hand, the memory controller area utilization shows almost no dependency on the number of data channels. Indeed, the memory controller internal structure is mostly unchanged when multiple data channels are implemented. The only difference would be the number of ports offered to the DMA controller, and the complexity of the arbitration stage that grants access to a specific DMA channel.

At last the memory wrapper area occupancy is reported. The resources occupied by the MIG IP core have been removed. Thus the numbers shown take into account just the adapter logic between the mesh and the MIG module. It can be seen that the amount of resources has been reduced drastically: has explained before, the old design, which included a complex finite state machine to handle memory transactions and adapt them to the AXI standard, has been replaced by a simple combinatorial logic that adapts the mesh interface with the native MIG interface. It should also be noted that the structure of this module does not depend on the number of data channels, and so do the occupied resources.

CHAPTER 5
Conclusions

In this thesis efficient memory accesses in an heterogeneous system have been implemented. The proposed solution allows to get very close to the ideal maximum performance achievable, estimated after taking into account all the physical limitations. Related to this, better results could still be achieved. In the Performance Analysis chapter various future developments have been proposed, in particular regarding the upstream transfer efficiency.

Memory bandwidth reservation mechanisms have also been implemented and validated. This work has taken in consideration only bandwidth reservation among transfers of the same type, being downstream or upstream. In case of mixed transfers, further modifications are required to meet the desired performance. A possible solution has been briefly discussed in the Memory Controller section.

On top of this, an additional software layer acting as a resource manager could be developed. This would make the applications able to request the desired QoS level, in terms of bandwidth toward a specific memory controller. The resource manager would thus be in charge to assign a data channel to the application and fine tune the weight given to each channel to meet the QoS requirements, if possible.

Bibliography

[1] TOP500. https://www.top500.org/.

[2] GREEN500. https://www.top500.org/green500/.

[3] MANGO project. http://www.mango-project.eu/.

[4] D. Chiou. The microsoft catapult project. *2017 IEEE International Symposium on Workload Characterization (IISWC)*, Seattle, WA, 2017, pp. 124-124.

[5] Daniel J. Sorin, Mark D. Hill, David A. Wood. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, November 2011, Vol. 6, No. 3, pp. 1-212

[6] Natalie Enright Jerger, Tushar Krishna, Li-Shiuan Peh. On-Chip Networks, Second Edition. *Synthesis Lectures on Computer Architecture*, June 2017, Vol. 12, No. 3, pp. 1-210

[7] Xilinx Memory Interface. https://www.xilinx.com/products/intellectual-property/mig.html.