Hardware Support for Quality of Service in an RDMA Engine

Bartzis Sokratis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Manolis G.H. Katevenis*

Thesis Co-Advisor: Dr. *Nikos Chrysos*

This work has been performed at and supported by the Computer Architecture and VLSI Systems (CARV) Laboratory, Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH).

University of Crete
Computer Science Department

Hardware Support for Quality of Service in an RDMA Engine

Thesis submitted by
Bartzis Sokratis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Bartzis Sokratis

Committee approvals: _____
Prof. Manolis G.H. Katevenis
Professor, Thesis Supervisor

_____
Prof. Polyvios Pratikakis
Associate Professor, Committee Member

_____
Prof. Vassilis Papaefstathiou
Assistant Professor, Committee Member

Departmental approval: _____
Prof. Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, June 2022

Hardware Support for Quality of Service in an RDMA Engine

Abstract

In recent decades, both research and industry have turned to High Performance Computing (HPC) for their ever-increasing computational needs. In an attempt to provide a high-performance communication framework for European supercomputers, under the EU-funded ExaNeSt and RED-SEA projects, we design a novel Remote Direct Memory Access (RDMA) engine, capable of low latency (less than 0.5μs) and high throughput communication (100 Gb/s).

In this thesis, we design the Quality of Service (QoS) hardware of our RDMA engine. Transfers are segmented into blocks, so as to enable selective re-transmissions, multi-path routing and to avoid per packet acknowledgment overheads. Small-sized transfers can bypass the RDMA-DRAM path, to further minimize latency. We schedule transfers at block level, based on a user-defined priority, we support end-to-end flow control, and we enable network multi-pathing and congestion management options. We also implement a completion notification engine in hardware. We expose 2048 virtual channels to users supporting multiple outstanding data transfer requests. Finally, we introduce a novel way of collectively polling the status of multiple channels.

Our register-transfer-level (RTL) hardware implementation is pipelined in order to achieve higher clock and message rates (1 operation/clock cycle, or 150 MOP/s in our FPGA implementation), while maintaining a low latency of 4 clock cycles for single block transfers. To further reduce latency, we implement multiple (32) scheduling queues in shared space, that support one (1) enqueue and one (1) dequeue operation per clock cycle, as well as back-to-back dequeue operations.

We synthesized our design for the Zynq Ultrascale+ MPSoC. The RDMA's QoS part leverages 13.3K Look-Up Tables (LUTs), 5.1K register and 23 BRAM blocks (848 kbits). The maximum frequency achieved in this FPGA was 150 MHz, but this can be further improved, especially in a VLSI implementation.

Extensive functional verification tests were performed using the Vivado Design Suite. The QoS engine developed in this thesis completed in simulation 100K outstanding transfers of varying size, up to 1 MB. Additionally, we integrated our QoS implementation with the RDMA send unit in another simulated testbench, issuing 5K transfers of maximum 256 KB (256 packets), which the design also completed successfully. In these tests, we examined every possible transfer type, including congestion managed and fast-path flows, as well as completion notifications.

The design was implemented on the Zynq's FPGA and performance measurements were taken from user-level programs on the Zynq's A53 ARM core. Completion time for small transfers of up to 512 Bytes was measured at 360 ns, when transferring intra-node, BRAM to BRAM (excluding network and DRAM latencies),

ten times lower than the latency of the ExaNeSt RDMA, a previous implementation on the same MPSoC, using the ARM Cortex-R5 co-processor for QoS support. Moreover, we significantly improved the transfer rate that can be achieved, reaching the theoretical maximum (line) throughput as early as with 16KB transfers, whereas using the previous implementation the corresponding transfer size was 4MB. Finally, although the RDMA engine is optimized for and tested using AXI processor interconnects, it can also be connected to PCI or CHI host-processor interconnects.

Υποστήριξη μέσω Υλικού της Ποιότητας Υπηρεσίας μιας Μηχανής για Απομακρυσμένες Άμεσες Προσπελάσεις Μνήμης

Περίληψη

Τις τελευταίες δεκαετίες, τόσο ο κλάδος της έρευνας, όσο και αυτός της βιομηχανίας έχουν στραφεί προς την Υπολογιστική Υψηλών Αποδόσεων για να καλύψουν τις αυξανόμενες ανάγκες τους για υπολογιστική ισχύ. Σε μία προσπάθεια να υλοποιήσουμε ένα πλαίσιο επικοινωνίας υψηλής απόδοσης για ευρωπαϊκούς υπερυπολογιστές, στα πλαίσια των ευρωπαϊκών προγραμμάτων ExaNeSt και RED-SEA, σχεδιάζουμε μια νέα διεπαφή δικτύου χαμηλής καθυστέρησης (λιγότερο από 0,5 μs) και υψηλής παροχής (100 Gb/s), ικανή για απομακρυσμένες άμεσες προσπελάσεις μνήμης.

Σε αυτήν την εργασία σχεδιάζουμε μια μηχανή υλικού για την βελτίωση της παροχής υπηρεσιών (Quality of Service, QoS) μιας μηχανής Απομακρυσμένων Άμεσων Προσπελάσεων Μνήμης (Remote Direct Memory Access, RDMA). Οι μεγάλες μεταφορές δεδομένων χωρίζονται σε μικρότερα τμήματα, έτσι ώστε να επιτραπεί η επιλεκτική αναμετάδοση δεδομένων, η χρήση πολλαπλών διαδρομών μέσα στο δίκτυο, καθώς και να αποφευχθεί ο επιπλέον φόρτος που προκύπτει από επιβεβαιώσεις λήψεων σε επίπεδο πακέτων. Οι μεταφορές μικρού μεγέθους μπορούν να παρακάμψουν την διαδρομή RDMA-DRAM, ελαχιστοποιώντας περαιτέρω τον χρόνο ολοκλήρωσής τους. Προγραμματίζουμε τις μεταφορές σε επίπεδο τμημάτων, βασιζόμενοι σε σειρά προτεραιότητας που καθορίζεται από τον χρήστη, και υποστηρίζουμε διαχείριση συμφόρησης του δικτύου. Επιπροσθέτως, παρέχουμε 2048 εικονικά κανάλια στον χρήστη για την έκδοση πολλαπλών εκκρεμών αιτημάτων μεταφοράς δεδομένων, υλοποιούμε μια μηχανή ειδοποίησης ολοκλήρωσης σε υλικό και εισάγουμε έναν νέο τρόπο μαζικής, διαδοχικής διερεύνησης της κατάστασης πολλαπλών καναλιών.

Η υλοποίησή μας σε επίπεδο μεταφοράς καταχωρητών χρησιμοποιεί ομοχειρία για να επιτύχει υψηλή συχνότητα ρολογιού και υψηλό ρυθμό αποστολής μηνυμάτων (1 πράξη/κύκλο ρολογιού ή 150 MOP/s για υλοποίηση στην συστοιχία επιτόπια προγραμματιζόμενων πυλών (Field Programmable Gate Array, FPGA) που χρησιμοποιήσαμε, ενώ παράλληλα διατηρεί χαμηλούς χρόνους καθυστέρησης, 4 κύκλους ρολογιού για μεταφορές του ενός (1) τμήματος. Για να μειώσουμε περαιτέρω τον χρόνο καθυστέρησης, υλοποιήσαμε πολλαπλές ουρές (32) προγραμματισμού μεταφορών, σε κοινόχρηστο χώρο, οι οποίες υποστηρίζουν μια (1) πράξη εξαγωγής και μία (1) εισαγωγής κόμβου από/στις ουρές ανά κύκλο ρολογιού, καθώς και πράξεις εξαγωγής σε διαδοχικούς κύκλους ρολογιού.

Υλοποιήσαμε την εργασία στην FPGA του Zynq Ultrascale+ MPSoC της Xilinx. Για την μηχανή βελτίωσης Ποιότητας Υπηρεσίας χρησιμοποιήθηκαν 13,3K Προγραμματιζόμενες Πύλες (LUTs), 5,1K καταχωρητές και 23 μνήμες τυχαίας προσπέλασης (848 kbits). Η μέγιστη συχνότητα που επετεύχθη ήταν 150 MHz, μπορεί, ωστόσο, να βελτιωθεί περαιτέρω, ιδιαίτερα σε μία υλοποίηση πολύ μεγάλης κλίμακας ολοκλήρωσης (Very Large Scale Integration, VLSI).

Εκτενείς δοκιμές για την επαλήθευση της λειτουργικότητας της μηχανής πραγματοποιήθηκαν χρησιμοποιώντας το Vivado Design Suite. Η μηχανή QoS που αναπτύχθηκε σε αυτή την διατριβή ολοκλήρωσε σε προσομοίωση 100K μεταφορές δεδομένων, μεταβλητού μεγέθους, έως 1 MB. Επιπρόσθετα, ενσωματώσαμε την μηχανή QoS με την μονάδα αποστολής σε έναν προσομοιωμένο πάγκο δοκιμών, εκδίδοντας 5K εκκρεμείς μεταφορές, μεγίστου μεγέθους 256 KB (256 πακέτων), οι οποίες ολοκληρώθηκαν και αυτές με επιτυχία. Σε αυτές τις δοκιμές εξετάσαμε κάθε είδους μεταφορά, συμπεριλαμβανομένων των ροών υπό διαχείριση συμφορήσεως και των ροών γρήγορης διαδρομής, και επαληθεύσαμε τον μηχανισμό ειδοποίησης ολοκλήρωσης.

Η μηχανή RDMA υλοποιήθηκε στην FPGA του Zynq και ελήφθησαν μετρήσεις απόδοσης από προγράμματα σε επίπεδο χρήστη, εκτελεσμένα στον επεξεργαστή ARM A53 του Zynq. Ο χρόνος ολοκλήρωσης για μικρές μεταφορές έως 512 Byte ανέρχεται στα 360 ns, κατά τη μεταφορά εντός κόμβου, από BRAM σε BRAM (εξαιρουμένων των καθυστερήσεων δικτύου και DRAM), δέκα φορές χαμηλότερο από τον αντίστοιχο χρόνο της μηχανής ExaNeSt RDMA, μιας προηγούμενης υλοποίησης λογισμικού-υλισμικού στο ίδιο MPSoC, χρησιμοποιώντας τον συνεπεξεργαστής ARM Cortex-R5 για να υποστηρίξει QoS. Επιπλέον, βελτιώσαμε δραματικά τον ρυθμό μεταφοράς δεδομένων, επιτυγχάνοντας την μέγιστη θεωρητική παροχή με μεταφορές των 16 KB, ενώ στην προηγούμενη υλοποίηση απαιτούνταν μεταφορές των 4 MB. Τέλος, παρ΄ότι η μηχανή RDMA έχει δοκιμαστεί και βελτιστοποιηθεί για διασυνδέσεις κεντρικού επεξεργαστή τύπου AXI, μπορεί επίσης να συνδεθεί και με διασυνδέσεις τύπου PCI και CHI.

Acknowledgments

I would like to express my deepest gratitude towards my advisor, Dr. Nikolaos Chrysos, for his guidance throughout my research and studies. This work would have never been realized without his support. I would also like to extend my appreciation to my supervisor, Professor Manolis G.H. Katevenis, for introducing me to the wonders of Computer Science and to the hardware design fundamentals, and for the interesting discussions that I had the chance to be a part of during my studies. I express my sincere thanks to Professor Vassilis Papaefstathiou for feeding my passion for Computer Architecture and for taking part in the examination committee of my thesis evaluation. Finally, I would like to thank Professor Polyvios Pratikakis for also being a part of the examination committee.

Last but far from least, I would like to express my deepest gratitude to Pantelis Xirouchakis for aiding me achieve my goal of becoming a computer scientist, from showing me the basics of FPGA design and discussing this thesis' design obstacles, to the moral support he provided throughout my research years. I would also like to thank all the members of the CARV Laboratory team for their collaboration and all my fellow students and friends that made the tough times bearable. Finally, I would like to thank my family from the bottom of my heart for believing in me and supporting me in every aspect of my life.

I thank the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS), for funding this work. This work has been partially funded by the RED-SEA project, which has received funding under grant agreement No 955776 from the European High-Performance Computing Joint Undertaking (JU) and from France, Greece, Germany, Spain, Italy, and Switzerland. The JU receives support from the European Union's Horizon 2020 research and innovation program.

στους γονείς μου

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the era of large dataset analysis, the need for parallel processing is ever more present. Modern applications like quantum physics simulations and medical history analysis require a large amount of computational power that powerful personal computers cannot provide. High-Performance Computing (HPC) deals with exactly this problem. It offers parallelization that is only limited by the number of resources available in the system. This is achieved by connecting thousands of cores into clusters and assigning parts of an application's computation to each of these cores.

Most applications are not completely parallelizable, hence the speedup they can achieve is dictated by Amdahl's law. But even for well parallelizable applications, a significant portion of the execution time is spent in inter-processor communication, since the different parts of the applications do not run in a shared address space, but instead communicate using message passing. As the number of cores increases, so does the number of messages being exchanged between cores. Thus, an efficient means of communication must be established. [5][1].

An inefficient approach would be to use store commands to write the data to the network interface and load commands to acquire the arrived data at the receiving end. While this may be efficient for small sized messages, larger messages would take up valuable computational time.

A low CPU overhead approach is using a Direct Memory Access (DMA) engine to handle the exchange of data. A DMA module is assigned the role of copying the data to and from the network interface buffers, while the processor continues its computation. To further minimize the communication latency, the data are transferred directly from the sender's memory to the receiving node's memory, bypassing any intermediate buffers (zero-copy Remote Direct Memory Access, RDMA) [4]. The zero-copy RDMA initiation is depicted in Figure 1.1. Finally, in RDMA, the user specifies the source and destination memory locations using virtual addresses, which must be translated to physical by the network interface [7][8].

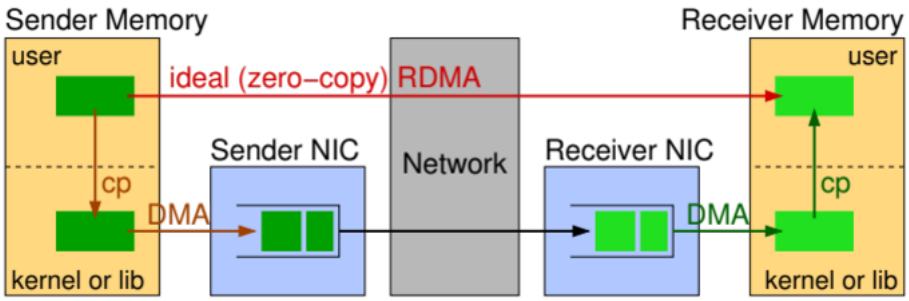

Figure 1.1: Zero-copy user-level initiated RDMA transfer.

In this thesis, we design and implement in hardware a fast and lean RDMA engine. The design consists of two parts: an existing, hardware implementation of a send unit and a receiver unit, developed by Pantelis Xirouchakis [10] and Michalis Gianioudis under the EU-funded ExaNeSt project [6], as well as the Quality of Service (QoS) engine of the RDMA, which is developed under the EU-funded RED-SEA project and is the contribution of this thesis. All parts are designed in hardware and are implemented and tested in the Programmable Logic (PL) of the Xilinx Zynq Ultrascale+ MPSoC.

In the QoS engine implementation described in this thesis, a user may issue an RDMA transfer by writing a 32 or 64 Byte descriptor to one of the QoS engine's 2048 virtual channels. In order to enable selective re-transmissions and multi-pathing, as well as to avoid per packet acknowledgment overheads, the QoS engine segments transfers into 64 KB blocks and issues block descriptors to the RDMA send unit. Acknowledgments are created on a per block basis. Moreover, the QoS engine further divides a transfer into flows (i.e. groups of 4 blocks in this thesis), so as to enable congestion management [3] and multi-pathing in the network. Blocks of the same flow follow the same path in the network and are subject to the same rate limiting. Both the block and the flow sizes are completely parameterized. Finally, to completely bypass the memory at the source node for small-sized transfers, the descriptor may include the actual data to be transferred in the transfer descriptor (inline payload).

Our register-transfer-level (RTL) hardware implementation is pipelined in order to achieve higher clock and message rates (1 operation/clock cycle, or 150 MOP/s in our FPGA implementation), while maintaining a low latency of 4 clock cycles for single block transfers. To further reduce latency, we implement multiple (32) scheduling queues in shared space, that support one (1) enqueue and one (1) dequeue operation per clock cycle, as well as back-to-back dequeue operations.

We synthesized our design for the Zynq Ultrascale+ MPSoC. The RDMA's

QoS part leverages 13.3K Look-Up Tables (LUTs), 5.1K register and 23 BRAM blocks (848 kbits). The maximum frequency achieved in this FPGA was 150 MHz, but this can be further improved, especially in a VLSI implementation.

The design was implemented on the Zynq's FPGA and performance measurements were taken from user-level programs on the Zynq's A53 ARM core. Completion time for small transfers of up to 512 Bytes was measured at 360 ns, when transferring intra-node, BRAM to BRAM (excluding network and DRAM latencies), ten times lower than the latency of the ExaNeSt RDMA, a previous implementation on the same MPSoC, using the ARM Cortex-R5 co-processor for QoS support. Moreover, we significantly improved the transfer rate that can be achieved, reaching the theoretical maximum (line) throughput as early as with 16KB transfers, whereas using the previous implementation the corresponding transfer size was 4MB. Finally, although the RDMA engine is optimized for and tested using AXI processor interconnects, it can also be connected to PCI or CHI host-processor interconnects.

The developed design was tested both in simulation, using Xilinx's Vivado Design Suite, and in hardware using the MPSoC mentioned before. As a standalone design, the QoS engine passed a simulation test of 100.000 randomly generated transfers, with maximum size of 1 MB, in which the block descriptor fields were compared to expected fields, calculated in simulation. The validity of inline payload transfers' data as well as the support for congestion management (i.e. flow mechanism) were also verified in these tests. A second major testing, including the RDMA send/receiver units, was also deployed, issuing 5.000 randomly generated transfers, maximum size of 256 KB, which also completed successfully. Finally, several real applications testing the new RDMA features ran on the ARM A53 core of the Xilinx Zynq Ultrascale+ MPSoC, with the RDMA design implemented in the PL. Data verification was also a part of this final, successful test.

1.1 Motivation

In an attempt to shift towards less power-hungry systems while also sustaining the performance gains of adding more resources, the HPC world now relies on coupling low-power processors with tailor-made accelerators. In this path, the RED-SEA EU-funded project aims to create a low-power system consisting of ARM or RISC-V based processors, tightly coupled with lean network interfaces implemented in FPGAs. However, in the scale of millions of processing cores, the inter-processor communication overhead becomes a bottleneck for exascale performance. Moreover, the reduced frequency of low-power cores makes computational clock cycles even more precious. The solution to low latency and high throughput communication with minimal CPU overhead comes in the form of tailor-made RDMA engines that offer ultra low-latency and high message rates.

A software/hardware hybrid RDMA implementation had already been developed under the ExaNeSt project. The QoS part of the RDMA ran on the Xilinx

Zynq Ultrascale+ MPSoC's real time ARMv7 Cortex-R5 co-processor [9][6], while the RDMA send unit was implemented in hardware in the PL. Despite supporting both the aforementioned transfer segmentation and resiliency features, like selective re-transmissions, the RDMA suffered from increased latency and low throughput on small and medium sized transfers, mainly due to 1) the software running in the R5 co-processor being serial, 2) the co-processor's caches being small and 3) the high PS-PL round trip latency (approximately 150ns) that was paid when issuing a block to the RDMA send unit, or when an acknowledgement arrived from the network. In this thesis, we attempt to lower this latency and increase the throughput of the RDMA even further, implementing the QoS part in hardware. Our final design implemented for the Zynq MPSoC's FPGA runs at the targeted frequency of 150 MHz.

1.2 Contributions

The author of this thesis has contributed to the creation of a new Remote Direct Memory Access (RDMA) engine, implemented completely in hardware, increasing the RDMA's throughput and reducing the individual transfer latency, compared to a previous hybrid (software/hardware) implementation. After studying the software implemented Quality of Service (QoS) part of the ExaNeSt RDMA, the author implemented the corresponding functions in hardware. In particular, the author made the following contributions:

- I have implemented a hardware module that receives RDMA transfer descriptors issued by local processors, using the AXI protocol for the processor-RDMA communication. Compared to issuing RDMA transfers to the R5 processor, issuing the RDMA transfer descriptor directly to the Programmable Logic (PL) completely eliminates any polling overheads (no unnecessary PS-PL round-trips) and makes accesses to the transfer descriptors faster, since they are stored in BRAM instead of the co-processor's memory. This was one of the determinant factors of reducing the latency of small and medium sized transfers.

- I have implemented RDMA transfer scheduling in hardware for outstanding RDMA operations. For this purpose, I have designed multiple FIFO queues, implemented in a dynamically shared space, that support 2 operations per clock cycle, 1 enqueue and 1 dequeue. In addition, I have implemented a transfer handler that segments transfers into 64KB blocks, and issues them, subject to end-to-end flow control, to the RDMA send unit. Transfers up to 32 Bytes can bypass the source node's memory. The shared space ensures no under-utilization of the scheduling queues, while the 2 operations per clock cycle guarantees that the segmenter is not halted when a new transfer is introduced into the system; finally, supporting back-to-back dequeues enables pipelining the segmenter. The transfer segmenter is configured to serve small

sized transfers first, in order to achieve low latency, and is implemented as a 3-stage pipeline in order to sustain high throughput even with small/medium transfers.

- I have implemented a Network Message Handler, i.e. a module that receives acknowledgments from the network and, in the future, will handle the remote read requests. The acknowledgments mechanism may restart the block issuing process (i.e. a transfer may have up to a fixed number of outstanding blocks) or signals the completion of a transfer. The hardware version of this handler re-schedules a transfer back to its scheduling queue or updates the transfer's status from *ONGOING* to *DONE* within 2-3 clock cycles of receiving an acknowledgment. Being able to both receive acknowledgments and issue new blocks to the RDMA, also contributes towards lowering a transfer's latency and increasing the RDMA's throughput.

- I have implemented a mechanism allowing the local processors to poll the status of multiple RDMA transfers with a single load operation. A transfer status table is implemented in PL, that holds the status of every transfer issued by the local processors (write channels only) and a processor's read on this table has the added option of reading the status of 32 ongoing transfers at once. This can be applied to broadcast operations by issuing multiple outstanding transfers and then collectively polling their status, increasing the RDMA's throughput for small sized transfers. User-level programs run on the Zynq's ARM A53 processor showed that issuing outstanding transfers and collectively polling their status can achieve rates of more than 20 Gbps, for 128-Bytes transfers.

The author of this thesis has also contributed to the verification of the QoS design and its integration with the RDMA send unit. A layered verification approach was followed: each module was initially tested individually in simulation using the Xilinx's Vivado Design Suite. In the case of the scheduling FIFO queues, this module proved to be a challenge as it produced numerous logical loops during its design and was ultimately further tested alongside a software implementation of a multi-priority FIFO queue, as well as another hardware implementation of multiple instances of single FIFO queues. When the individual modules reached their expected functionality, they were integrated together (only the QoS engine of the RDMA) and further tested in simulation.

In the final step of the simulation process, I created a simulated CPU that issued either hand-crafted or randomly generated transfer descriptors, written in System Verilog, and was used to test the QoS engine alongside the RDMA send unit. This simulated CPU proved extremely valuable as, when the verification moved to the actual hardware, the different test cases could be replicated in simulation, speeding up the debugging process. Finally, the author aided in writing the user library that makes use of the new RDMA features, mainly by providing

information about the new transfer descriptors format and the ways the software can poll the transfer's status.

As a final note, the target frequency of 150 MHz was ultimately achieved for the QoS engine of the RDMA. Towards the end of this work's timeline, the RDMA send unit evolved, and is now running at 200 MHz in order to achieve 100Gb/s (with a 512-bit datapath). This motivated the author to further attempt to increase the frequency of the QoS engine. The author pinpointed the critical path of the design using the Vivado tools and tried to implement a different "transfer's total blocks" calculation, which made up half of the paths latency. Although the new calculation lowered the number of logic levels in the path, the overall path latency increased due to higher gate/cable fan-out. The calculation can be divided in different pipeline stages but that requires careful planning since it will induce major design changes in the Transfer Segmenter's pipeline and is left for future work.

1.3 Thesis Outline

The remainder of this thesis is structured as follows:

- **Chapter 2 -** shows the abstraction layers of an RDMA transfer, lists the QoS engine features and presents the overall functionality of the QoS engine and its interaction with the Send Unit.
- **Chapter 3 -** provides a description of each individual hardware block of the QoS engine along with the information stored in the engine's tables.
- **Chapter 4 -** presents the performance evaluation results, the FPGA resource utilization and the functional verification strategy.
- **Chapter 5 -** concludes this thesis and provides future work suggestions.

Chapter 2

Design Overview

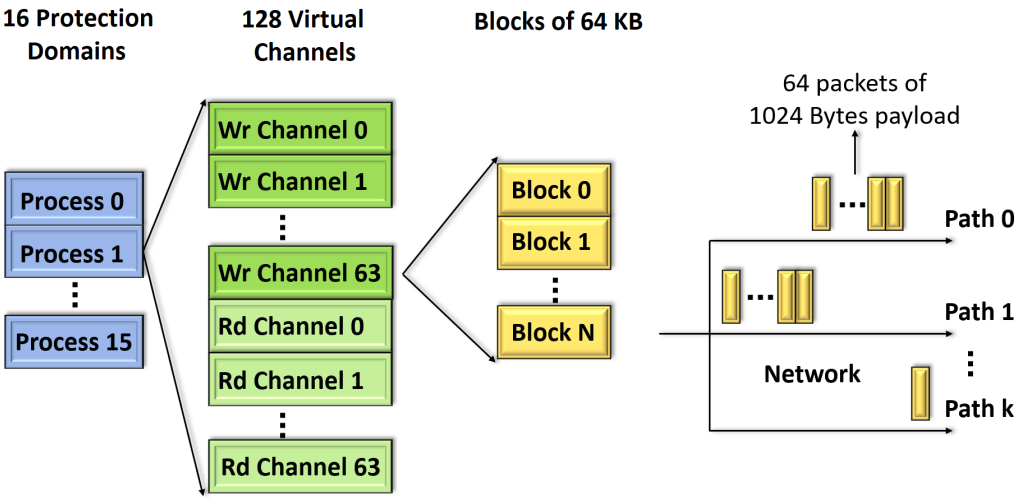

Figure 2.1: RDMA transfer abstraction layers

The engine described in this work provides the following functions to improve the Quality of Service of the new RDMA design:

- **Transfer segmentation:** The QoS engine segments transfers into blocks of 64 KB and issues block descriptors to the RDMA send unit. This offers the ability to re-transmit data selectively and to avoid acknowledgment creation per packet received, which would decrease the throughput of the RDMA.
- **Flow scheduling:** In order to provide low latency data transmissions and to avoid Head of Line (HoL) blocking by elephant flows [11], transfers are scheduled at block level, giving highest priority to smaller transfer sizes or to user-defined high-priority transfers.

- **Reliability:** Successful data transmission and fault tolerance must be ensured inside the system, thus a QoS engine should support re-transmissions for timed-out blocks (packet loss in the network) and for negative acknowledgments. In this thesis, we have implemented the basic mechanisms and data structures needed in order to support selective retransmissions upon time-out or negative acknowledgement in the future.

- **Completion notification:** The QoS engine creates a control packet that, along with the arrival of the final block of data, signals the receiver to create a completion notification.

The functionalities provided by the Quality of Service engine are summarized in Figure 2.1. The RDMA engine supports 16 different protection domains, or pages, with 128 channels per page. The pages can be allocated to one or more processes that issue RDMA requests using the available channels. Out of a total of 128 virtual channels channels per protection domain (or page), 64 channels are reserved for RDMA write requests and 64 channels for read requests. Each of these channels hosts a single transfer with a maximum size of 4 GB. RDMA transfers are further divided into blocks of 64 KB, which in turn are composed of 64 packets of 1 KB payload each. Finally, blocks can follow different paths inside the network (multi-pathing) and rate limiting is applied to multi-block flows (following the same path), so as to apply congestion management.

The complete design of the RDMA's QoS engine, and its interface with the send unit, are depicted in Figure 2.2. A local processor may issue a transfer to the RDMA by writing a 256-bit (32 Bytes) or 512-bit (64 Bytes) descriptor to one of the write channels of the Transfer Table. This descriptor is written to the table using the AXI protocol and contains information about the source and destination addresses of the data to be sent, as well as the size and priority of the transfer. However, other protocols can be used to implement the processor-RDMA communication. The QoS engine writes segment/block descriptors to the Transaction Table. The send unit is responsible to further segment the block into network packets to issue memory read requests to read their payload from memory and to forward network packets to the network [10].

The different descriptor formats are shown in Figure 2.3. The size of the descriptor depends on the type of transfer that is being issued. We support two types of transfers: inline payload and regular memory transfers. If the overall transfer size is less than or equal to 32 Bytes, the actual payload of the transfer can fit into its descriptor. Such transfers are described by a 256-bit or 512-bit Payload Descriptor, descriptor size depending on the payload size. Larger transfers, of size greater than 32 Bytes, are issued using the top descriptor format in Figure 2.3

Based on its user-defined priority, the transfer is then enqueued to a scheduling queue along with all the ongoing transfers. Inline payload transfers are considered latency sensitive and are thus automatically scheduled in the highest priority queue. Regular memory transfers are scheduled in different queues based on

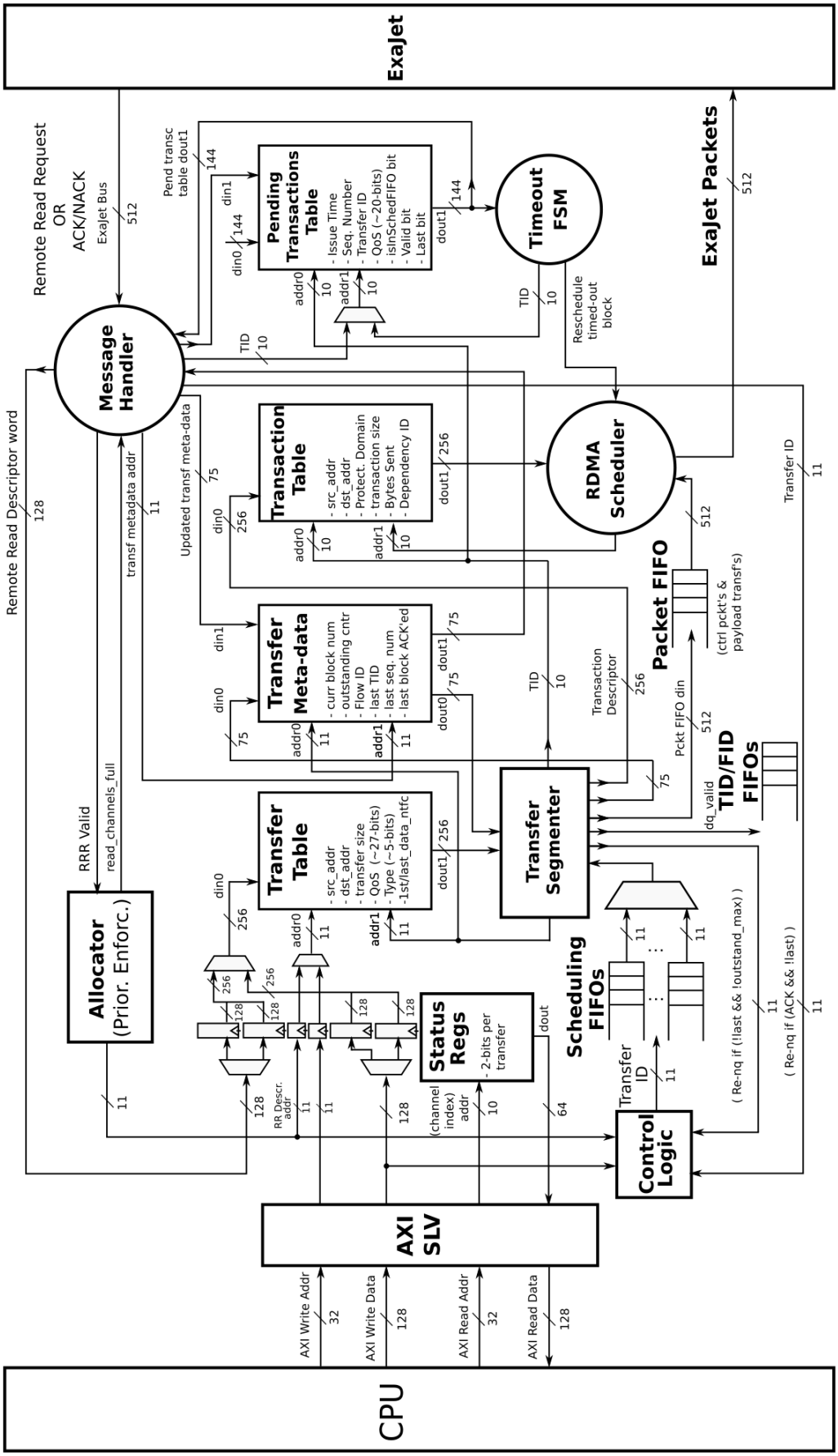

Figure 2.2: Representation of the RED-SEA RDMA's QoS engine.

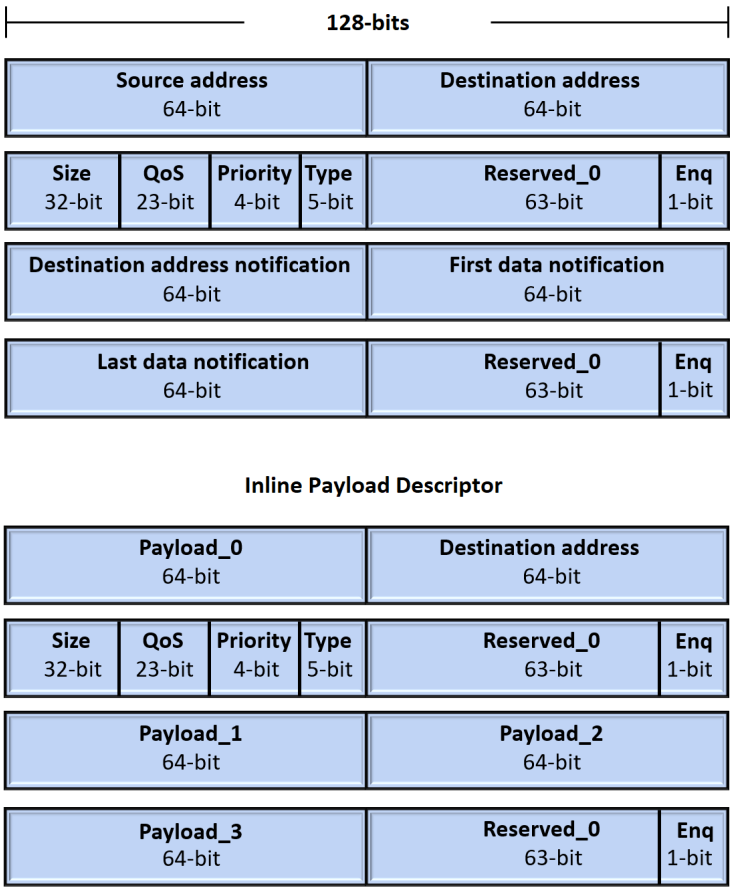

Figure 2.3: Transfer descriptor formats. Descriptor size for inline payload transfers is determined by the size of the payload.

their size in combination with their eligibility for multipathing in the network. We assume that larger transfers are more likely to use the multipathing feature, thus those transfers are scheduled in the low priority queue. The medium priority queue is reserved for transfers with size greater than an inline payload transfer and smaller than the low priority ones. Finally, we support a maximum of 16 user-defined intra-priorities for the medium and low priority queues. To summarize, the transfer priority hierarchy is shown in Figure 2.4

When a transfer is dequeued from a scheduling queue, its descriptor is read from the Transfer Table by the Transfer Segmenter 2.2, along with some meta information about the transfer, located in the Transfer Metadata Table. A transaction ID is assigned to the dequeued transfer from the Transaction ID FIFO queue. If the transfer contains the payload in the descriptor, the Transfer Segmenter creates an ExaJet packet of this transfer and enqueues it to the Packet queue. The RDMA

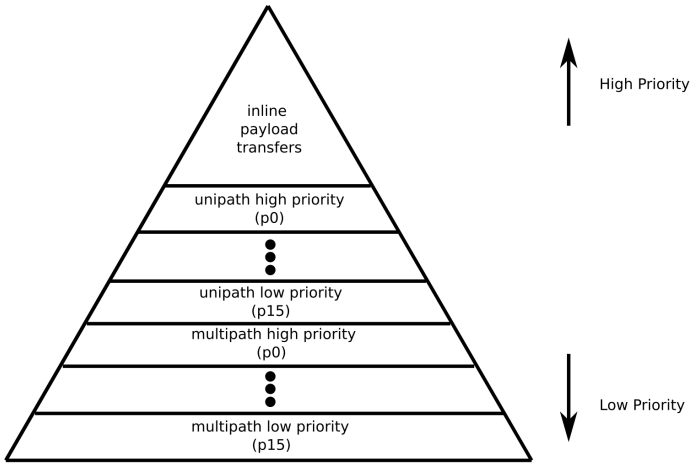

Figure 2.4: Transfer priority hierarchy

scheduler extracts the packet from this FIFO queue and sends it through the network. On the other hand, if the transfer is a regular memory type, the Transfer Segmenter issues 64KB block descriptors to the RDMA via the Transaction Table. A single block is issued to the RDMA before the transfer is re-enqueued back to the scheduling FIFO queues (on conditions). In order to keep track of how many blocks have been issued from each transfer, a Transfer Metadata entry is read and updated on each block issuing. This entry also contains information about the total ACKs received per transfer, which is used in completion notification creation by the Transfer Segmenter. Finally, on payload transfer/block issuing to the RDMA, an entry to the Pending Transactions Table is also created, containing resiliency fields (e.g. issue time of a block, used in timeouts) and fields used by the acknowledgments mechanism (e.g. sequence number, transfer ID) that can be used to later retransmit the block upon time-out or negative acknowledgement as further discussed below.

When an acknowledgment is received for a transfer's block, or for a payload transfer, the Message Handler 2.2 compares the arrived ACK'ed block's sequence number with the one stored in the corresponding Pending Transactions Table entry, and, either re-enqueues the transfer to its scheduling queue, or marks the transfer as DONE in the Status Registers. Furthermore, if N-1 ACKs have arrived in total, where N is the total number of blocks, and the last block of the transfer has been issued by the Transfer Segmenter, the Message Handler enqueues the transfer to a specific scheduling queue, reserved for completion notifications (control queue). The Transfer Segmenter is responsible for creating control packets, that form a completion notification, and storing them in a Packet queue that the RDMA send unit accesses. On the other hand, if no ACK is received within a certain timeframe,

the Timeout FSM is responsible for issuing a request to the RDMA for block re-transmission. When implemented, this FSM will check the *issue time* of each entry of the Pending Transactions table sequentially (1024 entries, 5μs with a 200 MHz clock) and if a certain period has passed (20μs) without receiving an acknowledgment, this module will send a re-transmission request to the RDMA send unit.

Chapter 3

Hardware Implementation

In this chapter, we present the information that the QoS engine needs to store for an RDMA operation and for transactions, we describe the individual blocks that compose the engine, the interactions between them and, finally, demonstrate their overall integration.

3.1 Transfer Table

The Transfer Table stores all transfer descriptors that are issued by the local processors and the remote read requests. It is a dual port memory with a height of 2048 lines and a width of 256 bits (total of 512 Kbits). Port 0 is reserved for descriptor writes from local and remote CPUs and Port 1 is reserved for descriptor reads by the Transfer Segmenter. The table is divided into 16 pages, each consisting of 128 total channels, 64 write channels and 64 read channels. A transfer described by a long descriptor (512-bits) occupies 2 channels (since it takes 2 transfer table lines), whereas a short inline payload transfer (8 Bytes) occupies only 1 channel. Figure 3.1 shows how the different types of transfer descriptors, as well as their the individual words, are stored in the table.

3.2 Transfer Metadata Table

The Transfer Metadata Table is an extension of the Transfer table that stores information about the issued number of blocks per transfer, the number of acknowledgments that have arrived as well as completion notification information. It has a size of 2048 x 75-bits, total 153.600 bits, and its entries contain the following fields:

- Flow ID (9-bits): used in congestion management and multipathing.
- Tid Bitmap (16-bits): used in assigning the next transaction ID of a congestion managed flow.

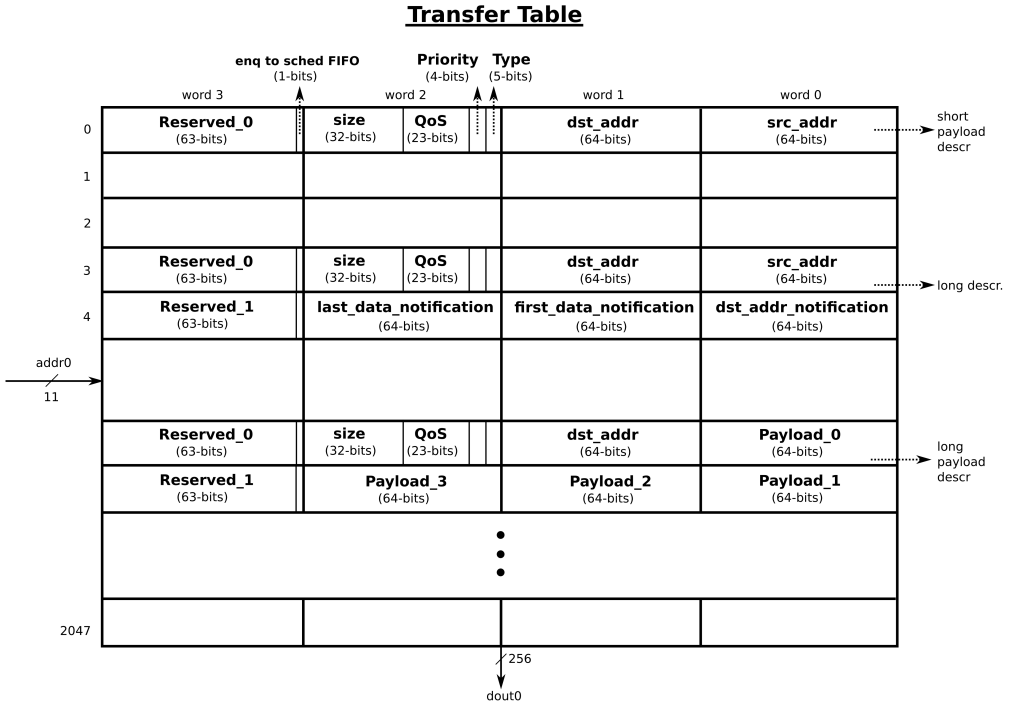

Figure 3.1: A depiction of how the individual descriptor fields are stored in the Transfer Table

- Next TID (4-bits): used in assigning the next transaction ID of a congestion managed flow.

- isInSchedulingFIFO (1-bit): used in rescheduling a transfer on ACK. The Transfer segmenter may reschedule a transfer back to its queue in order to issue a second, outstanding block. If the acknowledgment arrives while the transfer is already in the queue due to the outstanding mechanism, the Message Handler should not re-enqueue the transfer to its queue (each transfer is allowed to appear only once in the scheduling queues).

- block number (18-bits): indicates how many blocks of a particular transfer have been sent.

- outstanding blocks left (4-bits): indicates how many blocks can be issued to the RDMA without having received acknowledgments for the previous blocks (this mechanism exists in order to mask the round-trip latency of sending a block to a remote node and receiving its acknowledgment).

- last block issued (1-bit) used in determining the condition on which the Transfer Segmenter creates a control packet (completion notification)

- last block ACK'ed (1-bit) same as above.
- last transaction ID (10-bits): used in Control packet header (completion notifications)
- last sequence number (12-bits): used in Control packet header

The total number of ACKs is not explicitly stored as transfer metadata, but it is calculated as:

$$TotalACKs = (Block\ Number) - (Number\ of\ Outstanding\ Blocks\ Left)$$

Finally, this table is implemented using a dual-port memory and it is accessed by the Transfer Segmenter and the Message Handler for reading and writing, thus, port arbitration is required.

3.3 Transaction Table

The Transaction Table is used for issuing transfer block descriptors to the RDMA. Its size is 1024 x 256-bits = 256 Kbits, it is implemented as a dual-port memory (BRAM) and it contains the following fields:

- source address (64-bits): starting address from which the RDMA fetches the data to be sent. It is calculated for every block by the Transfer Segmenter
- destination address (64-bits): also calculated for every block by the Transfer Segmenter
- protection domain ID (16-bits)
- sequence number (14-bits)
- bytes sent (16-bits)
- congestion managed (1-bit): Set to 1 if transfer type = 1xFID/4xFID (see section 3.8)
- initialized (1-bit): The QoS engine should always set this to zero when issuing a block.
- not used 0 (9-bit)
- done (1-bit): The QoS engine should always set this to zero when issuing a block.
- notification enable (1-bit): The QoS engine sets this field only for the last block of a transfer with completion notifications

- chained (1-bit): set to 1 for all blocks of a congestion managed flow, except from its first block
- block size (16-bit): the total number of bytes of the current block
- not used 1 (12-bits)
- has next (1-bit): set to 1 for all blocks of a congestion managed flow, except from its last block
- not used 2 (11-bits)
- not used 3 (20-bits)

The Transfer Segmenter (section 3.9) writes to this table via a valid/ready handshake. The valid signal is raised from either the second or the third pipeline stage and is kept high until the Transaction Table raises the ready signal. While the valid signal is high and the ready signal is low, the segmenter's pipeline is stalled.

3.4 Status registers

The status registers hold the status of every write channel. A local CPU's status read request returns the status of multiple write channels in a given page. It is implemented using a 3072-bit register (1024 write channels x 3 status bits per channel, one-hot encoding). To avoid multiple load operation on this table for reading the status of transfers belonging to the same page, a single processor load operation may return the status of half the write channels (32) in a protection domain (32 channels x 2-bits status, after conversion from one-hot to binary = 64 bits = max processor load operation). Thus, reading the status of all channels in a protection domain would require 2 processor load operations. The ability to read multiple statuses with a single read enables the processor to issue multiple outstanding RDMA transfers (for example a broadcast operation) and monitor them in an efficient way, which ultimately leads to increasing the RDMA's throughput for small and medium transfer sizes. Finally, the state diagram of a transfer's status can be found in Figure 3.2. This diagram suggest that when the QoS engine receives a negative acknowledgment, it sets the state to *ERROR*, which will change when functionality for block re-transmissions is added.

The status table could have also been implemented as a dual-port memory of 1024 x 3 bits = 3072 bits instead of registers, but that would mean losing the ability to read multiple statuses with a single read operation. A total of 3 modules access the status table:

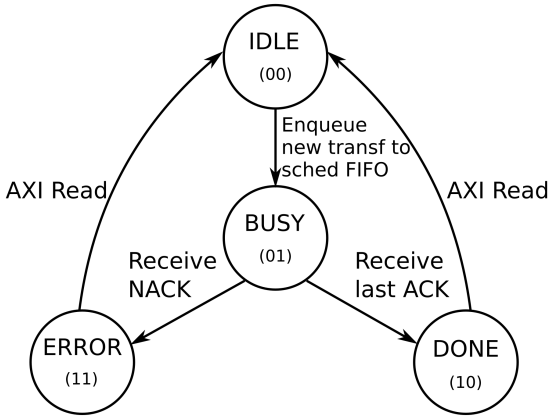

Figure 3.2: Transfer Status state diagram

- AXI reads: Processor load operations
  - Clock cycle 0: Read status of write channels
  - Clock cycle 1: Update status to IDLE if transfer has finished
- AXI writes: Issuing of a new transfer descriptor
  - Clock cycle 0: Read status of write channels
  - Clock cycle 1: Update channel status from IDLE to BUSY on transfer enqueue to the scheduling FIFOs.
- Message Handler: Receiving ACK/NACK
  - Clock cycle 0: Read status of write channels
  - Clock cycle 1: Update channel status from BUSY to DONE/ERROR

Updates to the status of a channel would require a read on the table's line and a write to the same line, updating a single channel, in the following clock cycle. Those updates are frequent for small sized transfers and, consequently, could become a threat of potential deadlocks due to arbitration. Thus, the register approach was chosen.

3.5 Pending Transactions Table

The Pending transactions table is an extension of the Transaction Table and contains information used by the acknowledgments and the re-transmissions mechanisms (the latter is not yet implemented). An entry to this table is created when a transfer block is issued to the RDMA, or when an inline payload transfer is enqueued to the Packet queue, and it is invalidated upon receiving the block's

acknowledgment. It has a size of 1024 x 144-bits = 147 kbits and its fields are as follows:

- valid (1-bit): Set to zero on valid ACK receipt.
- issue time (32-bits): to be used in re-transmissions
- sequence number (12-bits): if this value does not match with the sequence number on the ACK's header, the ACK is dropped
- transfer ID (11-bits): used in rescheduling the transfer to the scheduling FIFOs on ACK and is used as an index to the Metadata table (Message Handler increments total ACKs).
- transfer priority (4-bits): same as above
- QoS (22-bits)
- total blocks (18-bits): This field requires information located in the transfer descriptor for its calculation, so, in order to avoid introducing a forth access to the Transfer Table, the Transfer Segmenter calculates and stores this field.
- last (1-bit): Used in control packet creation
- has notification (1-bit) Used in control packet creation
- reserved (19-bits)

This table is implemented as a dual port memory and is accessed by the Transfer Segmenter for writing, by the Message Handler for reading and writing (when receiving acknowledgments) and by the Timeout FSM for reading and writing. Arbitration to this memory's ports will be required when the Timeout FSM is implemented.

3.6 AXI Slave

The AXI Slave is used for receiving transfer descriptors written by local processors, storing those descriptors to the Transfer Table, enqueuing newly arrived transfers to the scheduling queues and, finally, for reading the status of issued transfers. It is composed of 5 channels, 3 of which are used for AXI write operations (write address, write data and write response channels) and 2 channels (read address, read data channels) for AXI reads.

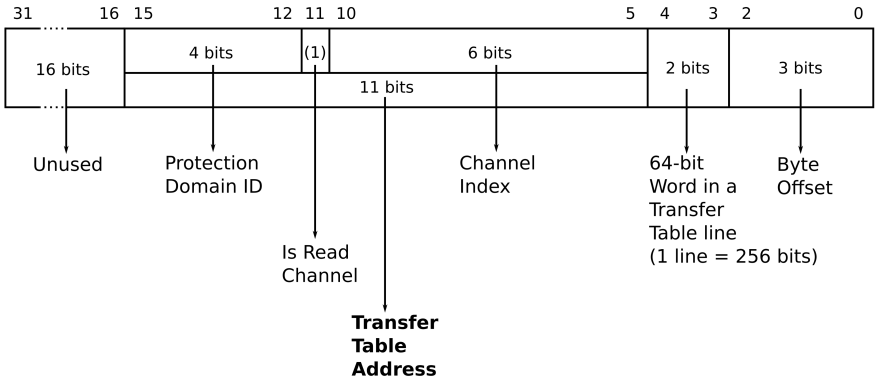

Figure 3.3: AXI Write Address breakdown

3.6.1 AXI Writes

The AXI write channels are used for receiving transfer descriptors issued by a set number of cores. Currently, we support 16 different protection domains and, subsequently, 16 different cores can issue transfers to the RDMA. We also support both 64-bit and 128-bit AXI writes on a 128-bit AXI write datapath, which is less than the Transfer Table line size (256-bits). For this reason, a set of accumulating registers is needed in order to perform a transfer descriptor write to the Transfer Table. The total size of those registers should be equal to the Transfer Table line size, 256-bit at the moment, since conflicts may arise when writing to the Transfer Table (e.g. a Remote Read Request from the network is also trying to write to the transfer table) or when enqueuing a transfer to the scheduling FIFO queues (e.g. when an ACK arrives and the Response Handler then tries to enqueue an ongoing transfer back to the scheduling FIFOs). Those scenarios are perceived as stalls in the descriptor writing process, thus the entirety of the Transfer Table line should be able to fit in those registers. Moreover, due to potential interleaving of incoming descriptor writes from different protection domains, a single set of accumulating register does not suffice. The number of register sets should be equal to the number of different protection domains, although, even with this configuration, interleaving might still arise due to context switching between threads (of the same protection domain). That being said, we do not implement multiple sets of accumulators, all testing has been performed using a single set and when interleaving arises, the interrupting descriptor write receives a negative acknowledgment (AXI BRESP = ERROR).

i. AXI write address channel

   A 32-bit address bus is used to index the channel that the new transfer is going to be written to. A breakdown of this address can be seen in Figure 3.3. Due to the interconnect that precedes the AXI Slave, it is possible for

the address of an AXI write to arrive before the write data. To combat this, we use a FIFO queue that enqueues AXI write addresses on write address channel handshake. This queue can hold up to 8 outstanding addresses. Since the write data are supposed to arrive in the order of the preceding addresses, the address of the arrived data will always be the head of the FIFO queue. In terms of latency, this FIFO queue induces a one-cycle startup cost, meaning that 100 back-to-back AXI write transactions would need 101 clock cycles to complete.

ii. AXI write data channel

The current datapath is 128-bit wide and, as mentioned above, both 64-bit and 128-bit write transactions are supported. Each individual 64-bit or 128-bit word is stored in a register and upon the arrival of the final descriptor word, the descriptor line is written to the transfer table. A transfer is considered ready to be enqueued to the scheduling FIFO only when the last descriptor line has been successfully written to the Transfer Table. For long descriptors, both inline payload and regular memory descriptors 2.3, 2 table lines need to be written. Thus, the enqueue to the scheduling FIFOs happens on the second descriptor line write. Due to interleaving of different descriptor writes from different protection domains, there is no way to distinguish when the 2^{nd} line of a descriptor arrives, other than placing a bit in both descriptor lines that suggests that the current line is the last one or not (see Figure 3.1, enq to sched FIFO field).

It is worth noting that the individual words in a descriptor line are allowed to arrive out of order, but the second descriptor line, if any, should never arrive before the first. This is a temporary solution to solving interleaving in transfer descriptor writes and will not be needed when multiple accumulating register sets are implemented. To clarify, suppose a long inline payload descriptor that is composed of eight (8) 64-bit words, word0, word1, ... , word7, and suppose that the descriptor is written by the CPU using 64-bit stores. Since this descriptor occupies 2 transfer table lines, the first line (256-bits) contains word0 to word3 and the second line contains word4 to word7. In Figure 3.4, three different orders of those individual words arriving are shown.

Second and third scenarios of Figure 3.4 are considered illegal because the 2^{nd} descriptor line write to the transfer table also triggers the enqueuing of the transfer to the scheduling FIFOs, essentially saying that the transfer is alive before its descriptor is written in its entirety.

iii. AXI write response channel

In order to complete an AXI transaction, the slave must send a completion response to the master. The AXI Slave replies with ERROR to an AXI transaction on bad transaction signals (e.g. 3 LS bits of the write address

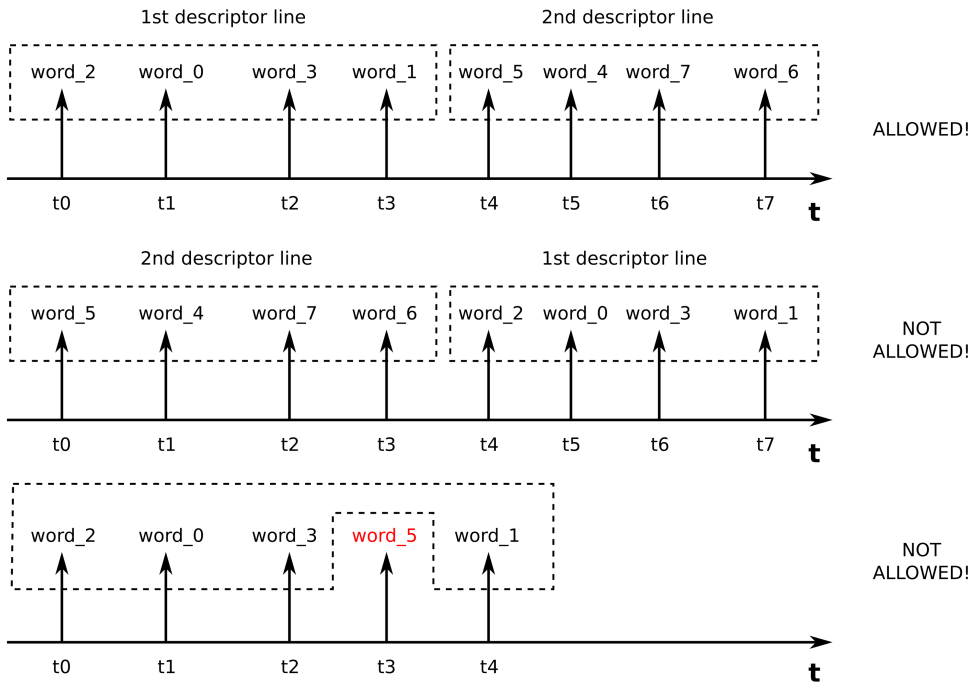

Figure 3.4: Transfer descriptor 64-bit words arriving at the AXI Slave in different orders. The individual words are allowed to arrive out of order only inside their Transfer Table line.

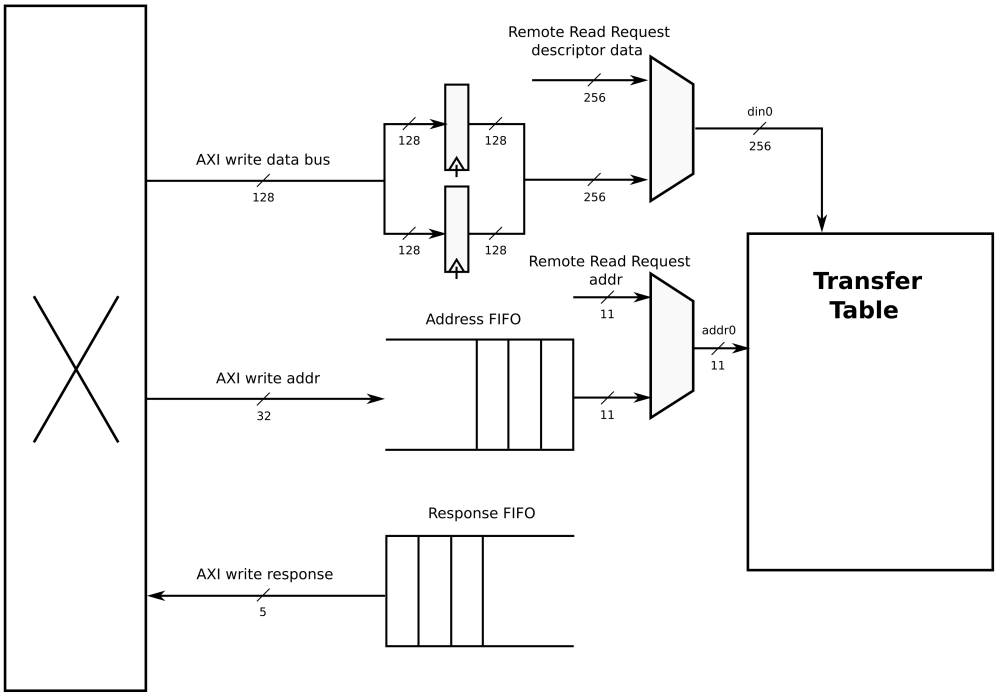

Figure 3.5: AXI Write Channels

are not zero, not using the data bus strobes correctly etc), or, as mentioned before, if an ongoing descriptor write is interrupted by another one, originating from a different protection domain. The latter will be resolved with the addition of multiple accumulating registers.

Since the request of the AXI Slave to the preceding interconnect (to send a response to the master) may not be granted in the same clock cycle as its issuing, the responses are also placed in a FIFO queue and are dequeued on response channel-interconnect handshake. This queue can hold up to 8 outstanding responses and does not induce any stalling to the descriptor writing process.

A simplified view of the AXI Slave's functionality can be found in Figure 3.5.

3.6.2 AXI Reads

The AXI read channels are used for reading the status of ongoing transfers. A local CPU may request to read the status of a transfer that it has issued, meaning that only the status of the write channels are saved.

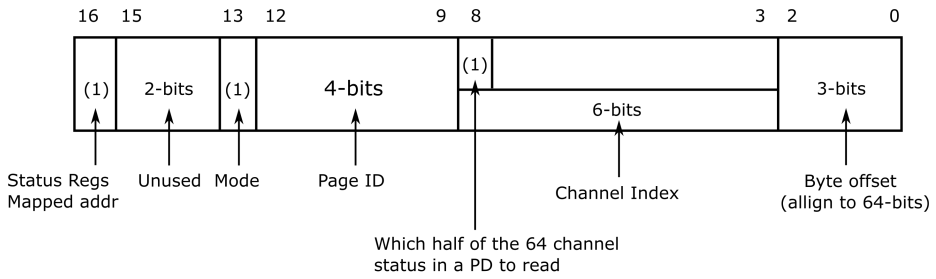

Figure 3.6: AXI Read address. The "mode" bit indicates reading a single channel's status and updating its state from DONE/ERROR to IDLE on read, or reading 32 channels statuses and updating every channel found on those states.

i. AXI read address channel

   The read address format can be found in Figure 3.6. Two extra bits are needed, one to indicate that the mapped peripheral is in fact the status table and another one to indicate the mode of the read request. Two modes are currently supported, one that returns the status of a single channel and resets its status if it is found to be in DONE/ERROR state, and a second that reads the status of 32 channels and resets every channel found in these states. The second reading mode finds application in broadcast operations: issue up to 64 outstanding transfers (broadcast to 64 nodes) and then poll the status of every transfer using a multi-channel read. This, however should go hand in hand with specialized software in order to handle the updates to every DONE/ERROR channel on a single read.

ii. AXI read data channel

   Through the AXI read data channel, a local CPU can read the status of 32 (out of the total 64) write channels in a protection domain on a datapath of 128 bits.

3.7 Scheduling FIFO Queues

A Scheduling FIFO queue entry holds a pointer to an ongoing transfer, the address of the first line of a transfers descriptor in the Transfer table. The need for multiple scheduling FIFO queues stems from the fact that there are multiple user-defined transfer priorities [2]. A total of 2048 queue entries are needed since the total number of active transfers, including both write and read channels, is 2048 and

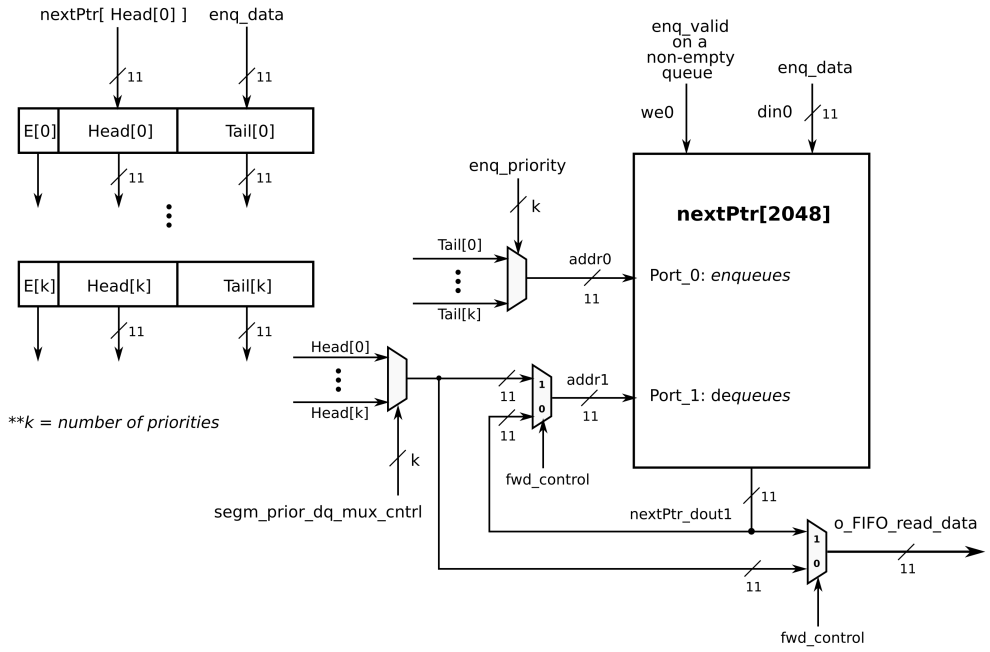

Figure 3.7: Scheduling FIFO queues implemented in a dynamic shared space. Each queue is represented by a set of Head-Tail registers and an Empty-bit register.

every transfer is allowed to appear only once in the queues. In order to avoid under-utilization, we implement these FIFO queues in a dynamic shared space, as shown in Figure 3.7.

Each FIFO queue is represented by a set of 3 registers: empty, head pointer, tail pointer. The nodes of a queue are connected to their successor via a next pointer. The next pointers are stored in a dual port memory 2048 x 11 bits, total 22.528 bits. The high level concept of the queue is presented in Figure 3.8. Unlike the queue in Figure 3.8, we don't need a separate memory for the data, since the data are considered to be the next pointer itself, the transfer table index. The enqueue and dequeue operations are described in Algorithm algorithm 1 and algorithm 2 respectively.

The number of different enqueue requests that can be issued to the scheduling FIFOs in a single clock cycle amount to 4:

i. AXI Slave enqueues newly arrived transfer's address

ii. Message Handler receives a remote read request and enqueues its address.

iii. Transfer Segmenter issues a transfer's block and re-schedules the transfer back to its queue (outstanding blocks mechanism)

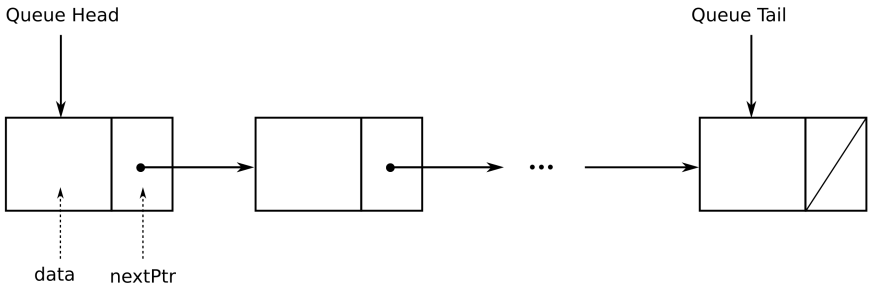

Figure 3.8: High level representation of a scheduling FIFO queue

Algorithm 1 Enqueue operation to a FIFO queue of a given priority in a multi-priority queue in dynamic shared space implementation

```
1: procedure ENQUEUE(data = x, priority = k)
2:     if queue[k] = empty then
3:         head[k] = x
4:         tail[k] = x
5:     else
6:         nextPointer[ tail[k] ] = x
7:         tail[k] = x
```

Algorithm 2 Dequeue operation from a FIFO queue of a given priority in a multi-priority queue in dynamic shared space implementation

```
1: procedure DEQUEUE(priority = k)
2:     prevHead = head[k]
3:     if head[k] = tail[k] then
4:         empty[k] = true
5:     else
6:         head[k] = nextPointer[ head[k] ]
       return prevHead
```

iv. Message Handler receives an ACK and reschedules the transfer if the ACK'ed block was not transfer's last.

Therefore, a total of 4 requests (3 enqueues and 1 dequeue) can potentially be issued to the scheduling FIFO queues in a single clock cycle. Looking at the pseudo-code in Algorithms algorithm 1 and algorithm 2, we deduce that the number of requests that can be served by the scheduling queues is determined by the number of ports of the next pointer memory: both the enqueue and the dequeue operations update a next pointer (except for enqueue on empty and dequeue on last item). For simplicity, we dedicate a nextPtr memory port for enqueues and the other one for dequeues. Being dual port, the memory allows only 1 enqueue and 1 dequeue to take place in the same clock cycle. Thus, a priority needs to be assigned to the different enqueue operations. We consider the following hierarchy, highest to lowest priority:

i. Message Handler (ACK enqueues)

ii. Transfer Segmenter

iii. AXI Slave

iv. Message Handler (Remote Read Request)

Highest priority is given to the acknowledgments mechanism, since it's the only mechanism that frees up resources (returns transaction IDs to their queues so as to re-enable the block/inline payload transfer issuing process). Next in the hierarchy we consider the Transfer Segmenter since delays in this module affect the latency of small transfers. Given that inline payload transfers are composed of a single block by definition, no enqueue requests to the scheduling queues are produced for such transfers. This, along with the fact that only 2 outstanding blocks are issued from a given transfer, allows the AXI Slave's (next in the hierarchy) enqueue requests to be granted with minimal delays. Although, the AXI Slave essentially serves the processor descriptor writes and, thus, should not be delayed, we consider that an infrequent, worst case scenario of a 2-clock-cycle delay is acceptable. Finally, since the remote read requests arrive from the network, we consider the latency to this enqueue requests to be negligible compared to the travel time of the request inside the network. That being said, delaying a remote read request's enqueue also induces a potential delay on arriving acknowledgements, since both of these events are handled by the same module, the Message Handler.

Finally, an enqueue operation takes one clock cycle to complete: updating the tail pointer (register) and the next pointer (in BRAM) can be performed in the same clock cycle. On the other hand, a dequeue operation needs 2 clock cycles to complete: read queue head's next pointer from memory (1^{st} cc) and then update the head register with the value read from memory (2^{nd} cc). Due to dequeue requests, potentially, arriving back to back (explained in the Transfer Segmenter section), a forward signal is needed when dequeuing from the same

queue in succession. Instead of returning the head value as dequeue data, the value read from the next pointer memory is returned during forwarding (see Figure 3.7, signal *fwd control*).

3.8 Transaction ID and Flow ID FIFO queues

As mention before, a transfer is segmented into 64KB blocks and a descriptor of each block is written to the Transaction Table to be read by the RDMA Scheduler. Each issued block is assigned a transaction ID, corresponding to an address to the Transaction Table (see Figure 2.2). This assignment of IDs differs between congestion managed and non-congestion managed transfers.

Distributing transaction IDs for non-congestion managed transfers is straight forward: all available IDs are stored in a FIFO queue (TID FIFO, Figure 2.2), the Transfer Segmenter dequeues a transaction ID on block issuing of a non-congestion managed transfer or an inline payload transfer and the IDs are returned to this TID FIFO queue by the Message Handler on ACK arrival. The total number of these independent transaction IDs is set to half the size of the Transaction Table, 1024/2 = 512 IDs. In total, this FIFO queue has a size of 512 x 10 = 5.120 bits.

On the other hand, the transaction IDs assigned to congestion managed flows are dependent on the Flow ID of the transfer. Congestion managed transfers are divided into flows. A flow is defined as an entity bigger than a block of 64KB (a transaction) and smaller than the transfer itself. In this implementation, a flow is set to 4 blocks (parameterized). Blocks of the same flow (same FID) follow the same path inside the network, thus, in order to support multipathing, transfers are assigned more than one FIDs. A congestion managed, unipath, transfer is assigned a single FID, at first block issuing, whereas a multipath one is assigned four (4) FIDs at first block issuing. This/These FID(s) will not be returned to its/their FID FIFO queue until the transfer is complete. The TIDs inside a flow are statically allocated, meaning that if the transfer is assigned FID 128, the available TIDs that its blocks will be issued with will be TID 512, TID 513, TID 514, TID 515 and in this order, circling back to the first TID if the total blocks of this transfer exceed the number of blocks in a flow. The range of TIDs in an FID is defined as:

$$TID \; \epsilon \; [FID \ll 2, (FID \ll 2) + 3],$$

where:

$$FID \; \epsilon \; [128, 252],$$

A total of 512/4 = 128 FIDs are available (half the Transaction Table size over the number of blocks in a flow), 64 of which are reserved for unipath transfers and 64 for the multipath ones. The FIDs of unipath transfers are stored in a different FID FIFO queue (1xFID FIFO queue) than those of the multipath ones (4xFID FIFO). The 1xFID FIFO has a size of 64 x 8 = 512 bits, whereas the 4xFID FIFO has a size of 16 x 8 = 128 bits.

While the transaction IDs of a unipath, congestion managed transfer should appear in a sequential order, the TIDs of a multipath flow appear in a frog-leap manner between flow IDs. For example, suppose a congestion managed transfer that uses the multipath feature and has a size of 16 blocks. If the transfer is assigned FID 192, it is automatically assigned FIDs 193, 194 and 195 as well (without the need of more than 1 dequeue on the 4xFID FIFO). The individual blocks of this transfer will be issued using the following TIDs:

- Block 0 : TID = $192 \ll 2 + 0 \Rightarrow$ TID: 768
- Block 1 : TID = $193 \ll 2 + 0 \Rightarrow$ TID: 772
- Block 2 : TID = $194 \ll 2 + 0 \Rightarrow$ TID: 776
- Block 3 : TID = $195 \ll 2 + 0 \Rightarrow$ TID: 780
- Block 4 : TID = $192 \ll 2 + 1 \Rightarrow$ TID: 769
- Block 5 : TID = $193 \ll 2 + 1 \Rightarrow$ TID: 773
  ...
- Block 15: TID = $195 \ll 2 + 3 \Rightarrow$ TID: 783

This way, we force a different flow ID between consecutive blocks, which are guaranteed to follow a different path in the network.

3.9 Transfer Segmenter

The Transfer Segmenter is a module responsible for:

- Creating ExaJet packets from inline payload transfers
- Dividing larger transfer into 64KB blocks (transactions) and issuing transaction descriptors to the RDMA.
- Creating control packets to signal the creation of a completion notification.
- Schedule transfers at block level

It is implemented as a three-stage pipeline. In Figure 3.10, transfer segmentation into blocks is shown, as well as the calculation of the individual block sizes.

3.9.1 Pipeline Stage 1

The first pipeline stage is dedicated to transfer scheduling at block level. The Transfer Segmenter checks if any of the Scheduling FIFO queues are non-empty, meaning that a transfer is eligible for block issuing. Which scheduling FIFO queue the Segmenter dequeues from is determined by the priority of the transfer and the

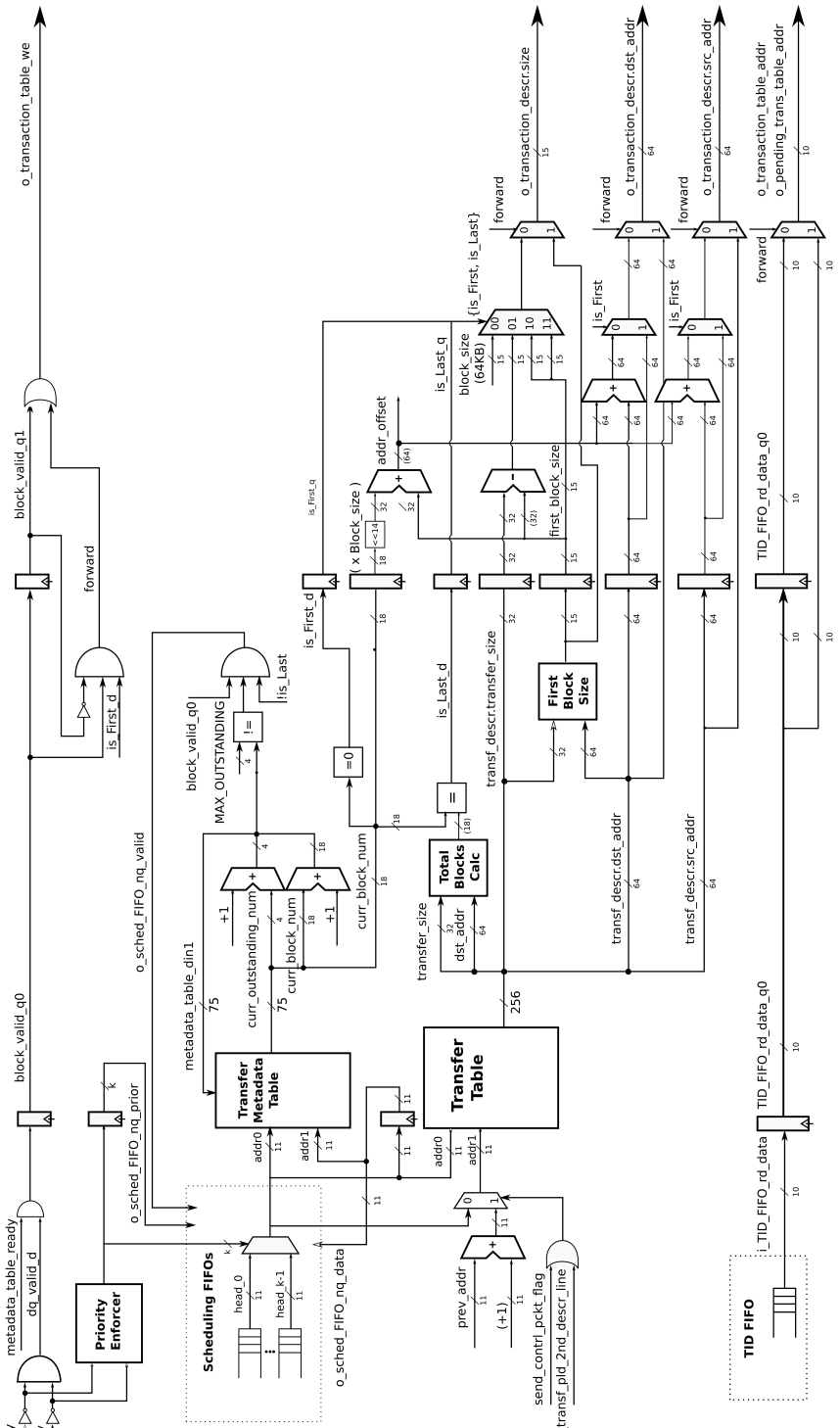

Figure 3.9: Transfer Segmenter's three-stage pipeline. In the first stage, it is decided which scheduling queue to serve a transfer from. Second stage is dedicated to re-enqueuing a transfer back to its scheduling queue and updating the Metadata table entry (the Segmenter also enqueues inline payload transfers' packets to the Packet Queue, not shown in the figure). Third stage is dedicated to a block's source and destination address calculations and writing a transaction descriptor to the send unit. Bypass of the third stage is also available when issuing the first block of a transfer.

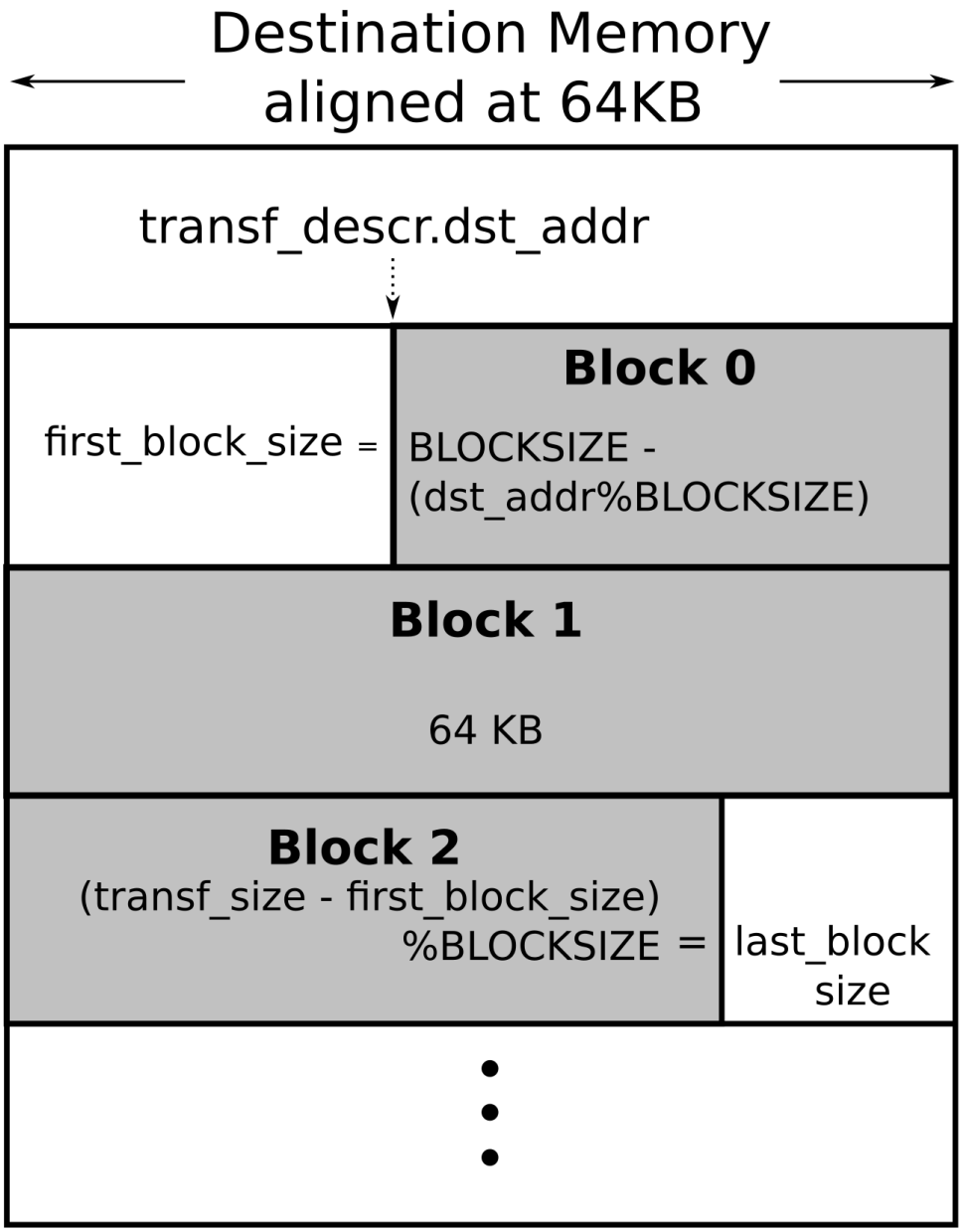

Figure 3.10: Transfer segmentation and block size calculations. *BLOCKSIZE* is a constant set to 64KB.

availability of transaction IDs (TIDs) and Flow IDs (FIDs). Thus, if a transaction ID is available, the transfer is dequeued from its FIFO queue and the address read from the queue is passed to the Transfer Table in order to read the transfer's descriptor, which will be available in the next clock cycle since this table is implemented in BRAM.

The total number of scheduling FIFO queues is determined by the formula :

$$Total\ queue\ count = 2 + (4 \times intra\ priority\ count)$$

where intra priority count is the number of different priorities of multipath and unipath types of transfers (see Figure 2.4). The functionality of each of these queues is listed bellow.

- Small transfers scheduling FIFO queue - TID only FIFO (1 queue):

  This queue is reserved for inline payload transfers and non-congestion managed transfers (transfers that are not assigned a flow ID).

- Congestion managed transfers, no multipathing, scheduling FIFO queues - 1 x FID FIFO (2 x intra priority queues):

  These queues are reserved for transfers of medium size that are assigned a single flow ID (single-path in the network). The queues are further divided into 2 categories: transfers that have already been assigned an FID and transfers with no FID assigned. This distinction is made to resolve Head of Line (HoL) blocking issues in the scheduling FIFO queues. Suppose the following example, in which the congestion managed, unipath, scheduling queues are not divided into the categories mentioned before, meaning that a single queue exists for both the transfers that already have an FID assigned and the ones that haven't. A transfer at index 2 of the Transfer Table is at the head of the queue and it hasn't been assigned an FID yet. Next in the queue resides a transfer at index 0 that has been assigned an FID. Now, suppose that the FID FIFO is empty, meaning that all flow IDs are being used by other transfers. Since the transfer at the head of this queue needs an FID in order to start issuing blocks, two options are available: either stall all transfers in this queue and wait until an FID becomes available when another congestion managed transfer finishes, or dequeue the transfer at the head of the queue and enqueue it at the back of the queue, hoping that the next transfer(s) will not be needing an FID, wasting clock cycles in the process. To avoid both of these scenarios, separating the queues into the proposed categories is essential.

- Congestion managed transfers, multipathing, scheduling FIFO queues - 4 x FID FIFO (2 x intra priority queues):

  These queues are reserved for transfers of large size that are assigned four (4) flow IDs (multi-path in the network). Although, the FIDs of this type of

transfers are located in a different FID FIFO than the ones used for single-path transfers, the same problems arise if the queues are not divided into have-FIDs and not-have-FIDs categories.

- Control packet scheduling FIFO queue (1 queue):

  When the (N-1)$^{\text{th}}$ ACK arrives at the sender, the Message Handler enqueues the ACK'ed block's transfer to this queue in order for the Transfer Segmenter to create the control packet that forms the completion notification. This queue has the highest priority between all scheduling FIFO queues, the reason being that sending the control packet frees up resources both at the sender and the receiver side.

Thus, in the first pipeline stage, the Transfer Segmenter decides which scheduling FIFO queue to dequeue from, based on a priority assigned to these queues. The queue hierarchy, from highest to lowest priority, is described below.

i. Control Packet FIFO queue (1 queue)

ii. TID only FIFO queue (1 queue)

iii. 1 x FID FIFO queues, have FID (2 x intra priority queues)

iv. 1 x FID FIFO queues, not have FID (2 x intra priority queues)

v. 4 x FID FIFO queues, have FID (2 x intra priority queues)

vi. 4 x FID FIFO queues, not have FID (2 x intra priority queues)

The reason for giving higher priority to the transfers that already have an FID is that those transfers have begun issuing blocks to the RDMA (an FID is assigned at first block issuing) and are possibly closer to completion.

Apart from the index to the Transfer Table, the scheduling FIFO queue head contains an extra bit that indicates whether the first block of the transfer has been issued. In general, this information can be found in the Transfer Metadata Table (see section 3.2), but in order for this information to be correct, the current block number needs to be set to zero by the AXI Slave when receiving a new transfer descriptor. However, this initialization introduces a third access (write) to the Transfer Metadata table (the other two being the Transfer Segmenter and the Message Handler for reading and writing in the next clock cycle), which means either arbitration to the tables ports or an implementation as a triple port memory. Instead, reading an extra bit from the head register of the scheduling FIFO queues gets rid of this initialization of the Transfer Metadata Table. However, a read request is issued to the transfer metadata arbiter nonetheless, in order to update in the next clock cycle the outstanding blocks counter, the last TID/sequence number values (if issuing the last block) as well as the *isInSchedFIFO* bit.

3.9.2 Pipeline Stage 2

In the second pipeline stage, an entry to the Pending Transactions Table is created for the block/inline payload transfer that the Segmenter is about to issue and depending on the type of transfer that was dequeued, the number of blocks that have been issued, as well as the usage of the third pipeline stage, transfers are served in the following ways:

- The Transfer Segmenter issues a request to the Packet Creator FSM in order to create an ExaJet packet from a inline payload transfer or create a control packet to signal the completion notification formation. The control packet creation request is issued either because the Transfer Segmenter dequeued a transfer from the control packet scheduling queue, or because the Transfer Segmenter is about to issue a transfer's last block and N-1 acknowledgments have already arrived for this transfer. The second scenario requires the calculation of the total blocks of the transfer, to compare this value with the current block counter (if those values match, the last block is being issued), and the calculation of the total ACKs of this transfer. The latter is straightforward:

$$total\ ACKs = transf\_metadata.curr\_block\_num - transf\_metadata.outstanding\_num\_counter$$

  The current block number is incremented on block issuing and the outstanding counter is incremented on block issuing and decremented on ACK receipt. The calculation of the total blocks of a transfer is more complicated and ends up being in the critical path of the design:

$$total\ blocks = (transf\_descr.dst\_addr + transf\_size)/BLOCKSIZE - (transf\_descr.dst\_addr)/BLOCKSIZE$$

  Furthermore, if the value (transf_descr.dst_addr + transf_size)%BLOCKSIZE is non-zero, we need to add one more block to the above calculation. This produces the circuit shown in Figure 3.11.

- the first block of a transfer is issued to the RDMA from the second pipeline stage, on condition that no block is issued from the third stage (stage 3 bypass). This type of forwarding is possible since, being the first block, the source and destination addresses of the block are identical to the transfer's corresponding fields (located in the transfer descriptor), and the only transaction table field that is calculated is the first block size (the first block's destination address is possibly not aligned to the 64KB block boundaries, see Figure 3.10), as shown in the following formula:

$$first\ block\ size = BLOCKSIZE - (dst\_addr\%BLOCKSIZE)$$

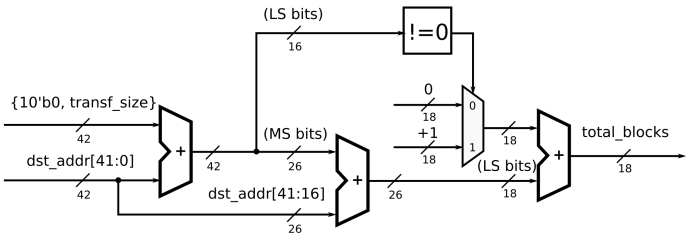

Figure 3.11: Transfer's total blocks calculation logical circuit.

where *BLOCKSIZE* is a constant set to 64 KB and *dst_addr* is located in the transfer descriptor. descriptor field

Finally, the transfer's metadata are updated in the second pipeline stage:

i. increment the current block counter

ii. increment the outstanding counter

iii. increment the next tid counter (+1 for single path transfers, +4/+5 for multipath, not used in TID only transfers)

iv. update isInSchedFIFO bit to 1 if the transfer is about to be rescheduled to its scheduling queue (outstanding block).

v. update the last TID/sequence number fields with the values used if the last block of a transfer is issued (needed for control packet creation).

If the outstanding counter hasn't reached a maximum threshold (currently set to 2 outstanding blocks), the transfer is rescheduled back its scheduling queue. This enqueue is performed with a valid/ready handshake, thus the pipeline may be stalled if the ready signal is not asserted (see section 3.7 for priority assignment to the scheduling queues accessing modules).

3.9.3 Pipeline Stage 3

In the third pipeline stage, the Transfer Segmenter calculates the source, destination address offset (same offset for both source and destination) of the block being issued, as well as the last block size, and writes a transaction descriptor to

the Transaction Table (section 3.3). Writing to this table is also performed with a valid/ready handshake. Due to timing issues, the Rate Limiter on the packet scheduling mechanism uses a slower clock than the RDMA's clock. The slower clock is derived from the RDMA clock, but still a handshaking mechanism needs to exists in order to transmit signals from slow to fast and vice versa. For this reason, when the valid signals are asserted, the ready signal from the rate limiter can either be asserted in the same clock cycle or some clock cycles later, depending on the relative values and phases of the clocks.

$$last\ block\ size = (transf\_size - first\_block\_size)\%BLOCKSIZE$$
$$address\ offset = first\_block\_size + (curr\_block\_num - 1) \times BLOCKSIZE$$

The original assumption was that the source and destination address calculation would be in the critical path of this design, since it requires 3 consecutive adders and some multiplexers, hence the need to split the calculation to stage 2 and stage 3 of the pipeline. However, the total blocks calculations proved more complicated and in the current Segmenter's design could not be split into 2 different stages without making major architectural changes. A different way of calculating the total blocks, with the use of less adders, was also tested, but due to higher fanout in the implemented design, the critical path (total blocks calculation included) became even slower.

3.9.4 Stalls

In this section the events that induce a stall to the Transfer segmenter's pipeline are listed.

i. Transaction table write handshake failure.

   Failing to write a transaction descriptor, either from stage 2 (forward) or from stage 3, to the Transaction Table stalls the pipeline until the ready signal is asserted.

ii. Packet FIFO queue full.

   A chronic congestion in the network's output buffers would not allow the RDMA to send any packets and, subsequently, the packets that the Transfer Segmenter enqueues to the Packet FIFO will accumulate until the queue is full. Thus, issuing new transactions is halted.

iii. Transfer Metadata table arbiter handshake failure.

   The Message Handler tries to access the Transfer Metadata table at the same time as the Transfer Segmenter and the arbiter temporarily gives priority to the handler. If the Transfer Segmenter cannot read from the Metadata table, no transfer is dequeued from Stage 1 of the pipeline.

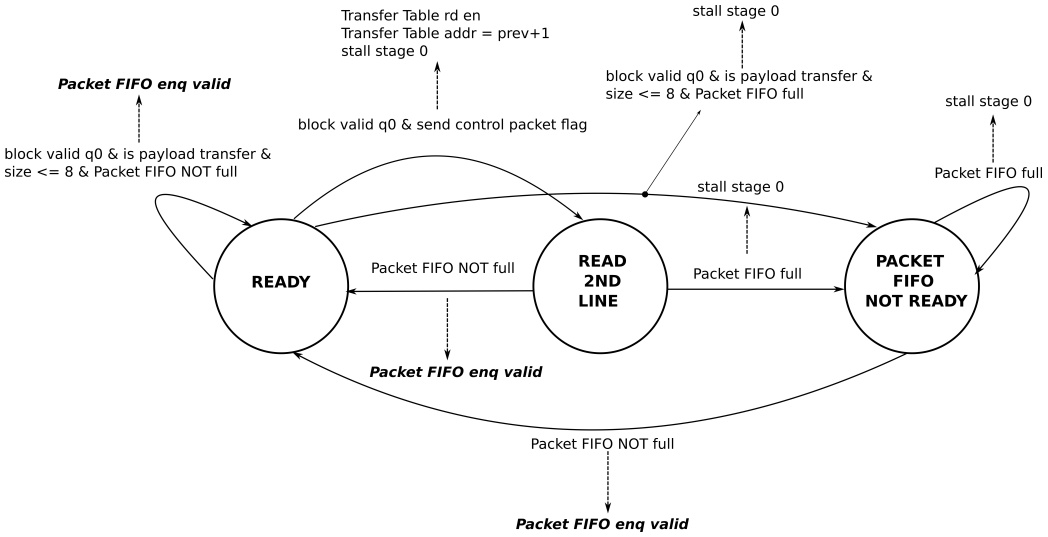

Figure 3.12: Segmenter's Packet Creator FSM state diagram.

iv. Scheduling FIFO enqueue handshake failure.

The Transfer Segmenter tries to enqueue a transfer back to its scheduling queue (outstanding block mechanism), but a higher priority enqueue (see section 3.7) happens in the same clock cycle. The pipeline is stalled until the ready signal is asserted by the scheduling queue's control logic (next clock cycle in the current implementation).

3.10 Packet Creator FSM

The Packet Creator FSM is a module that enqueues network packets to the packet FIFO queue and sends control signals to the Transfer Segmenter in order to stall the transfer issuing process when the 2^{nd} descriptor line is needed to extract the rest of a transfer's payload or the control packet information. As Figure 3.12 suggests, enqueuing a transfer's packet with payload of size less than or equal to 8 Bytes can be performed in a single clock cycle and does not induce any stalls to the Transfer Segmenter's pipeline. Inline payload transfers of size greater than 8 Bytes and control packets have extra information located in the second descriptor line (see Figure 2.3), thus a second read on the Transfer Table is needed to get this information. The first pipeline stage is stalled, since the Transfer Table is implemented as a dualport table and one of its ports is dedicated to the new descriptor writes from the AXI Slave and the remote read requests from the network. A possible improvement to this packet creation process would be to allow the Transfer Segmenter to read both lines of the transfer descriptor in the same clock cycle, when no new descriptors are being written to the table via Port 0. This would allow inline payload transfers of any size to be served in a single clock

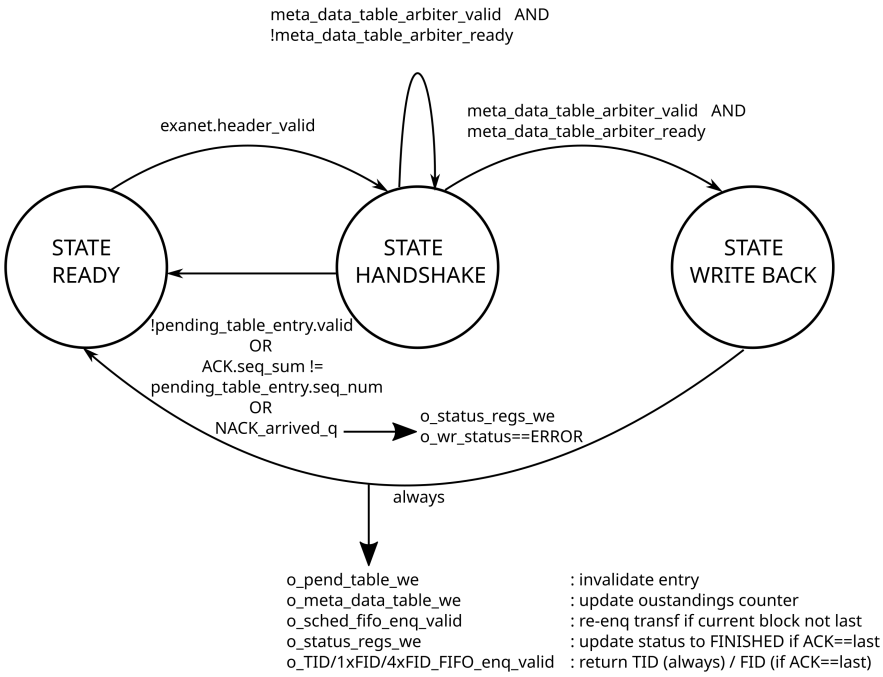

Figure 3.13: Finite State Machine (FSM) of the Message Handler. Functionality for handling remote read requests is not included in this transition diagram.

cycle. Another improvement would be to enqueue control packets in a single clock cycle when the transfer was dequeued from the control packet FIFO queue. In this scenario, the first descriptor line is only needed to extract the destination's coordinates (MS bits of the transfer's destination address), whereas all the control packet fields are located in the second one and the only extra information is located in the Transfer Metadata table. If the destination coordinate of the RDMA transfer was the same as the control packet's coordinate, the first read operation on the transfer table could be skipped altogether.

3.11 Message Handler

The Message Handler is a 3-state Finite State Machine (FSM) that is responsible for receiving acknowledgements from the network. Its functionality is also to be expanded to handling the remote read requests.

When in the first state, the header valid/ready handshake takes place, signaling the arrival of a packet from the network, in this case a response. The header ready signal is always raised in this state. The TID of the ACK'ed block is extracted from the response header and it's used as index to the Pending Transactions table. The sequence number is also stored in a register to be compared with the sequence

number of the table's entry. The response (ACK/NACK) is also extracted from the header and stored in a register.

In the second state, the Pending Transactions table entry is available. If the response was a NACK, the transfer ID is extracted from the table's entry (it isn't included in the response header, thus we need to read the table to get it) and the Message Handler updates the status register of the current NACK'ed transfer to ERROR, using the transfer ID as index to the status registers. On the other hand, if the response was an ACK and the table's entry is valid, the sequence number of the ACK'ed block is compared to the stored value in the table. If the sequence numbers do not much, the ACK is dropped, the transfer's status is updated to ERROR in the status registers and the handler's state returns to ready state. The channel's update to ERROR state is a temporary solution to not supporting re-transmissions and will become unnecessary when those resiliency features are added to the RDMA. Finally, if the sequence numbers do match, the Message Handler sends a request to read the transfer's metadata in the corresponding table. As mentioned before, this table is accessed by the Transfer Segmenter as well, which issues both read and write requests in the same clock cycle (being a pipeline). Thus, this read request is handled by the Transfer's Metadata Table arbiter, which grants read requests to the Transfer Segmenter and the Message Handler in a "last served" manner.

In the third state, the transfer's metadata are available and the Message Handler checks if all blocks of the transfer have been issued. If the current block number is not equal to the total number of the transfer's blocks, the handler checks the *isInSchedFIFO* field and decides whether to enqueue the transfer back to its scheduling FIFO queue. When the ACK'ed block was issued, the Transfer Segmenter might have re-enqueued the transfer back to its queue because of the outstanding blocks mechanism and since only a single instance of every transfer is allowed in the scheduling FIFO queues, the Message Handler must not re-enqueue the transfer a second time. On the other hand, if all blocks of the transfer have been issued, the total ACKs received are counted in order to mark the transfer as done (total ACKs == total blocks) or to enqueue the transfer the control packet scheduling FIFO queue (total ACKs == total blocks - 1 AND transfer has completion notifications).

Furthermore, the Message Handler updates the transfer's metadata, decrementing the outstanding blocks counter and setting the *isInSchedFIFO* bit to 1. It also invalidates the Pending Transaction's table entry and returns the TID/FID used by the block/transfer on the following conditions: the TID of a non-congestion managed flow is always returned to the TID FIFO queue on block ACK, whereas the FID of a congestion managed flow is only returned to its FID FIFO queue on transfer completion. However, the transaction IDs of congestion managed flows are given in an incremental fashion, thus the Message Handler updates the *tid_bitmap* field in the transfer's metadata table.

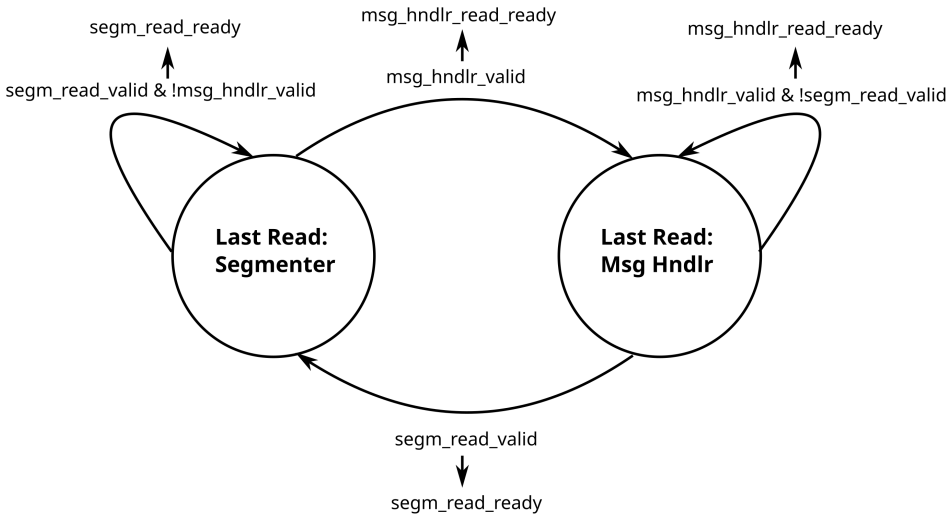

Figure 3.14: Transfer Metadata table Arbiter FSM. This arbiter gives priority to either the Transfer Segmenter's or the Message Handler's reads using adaptive control.

3.12 Transfer Metadata Table Arbiter

In section 3.9 and section 3.11 it is mentioned that the Transfer Metadata table (which contains information about the total blocks issued, total ACKs received and so on) is accessed by both the Transfer Segmenter and the Message Handler. The Transfer Segmenter issues a read to this table from its first pipeline stage and a write from its second pipeline stage, both during the same clock cycle, whereas the Message Handler issues a read and, in the following clock cycle, a write request. If we assign each port of the metadata table to either the reads or the writes, only the read requests require arbitration. The table's arbiter is implemented as an adaptive control FSM and its state diagram can be found in Figure 3.14.

3.13 Sequence Number Generator

A sequence number is assigned to each issued block and packets belonging to that block are sent using the same sequence number. When all packets of a block arrive at the receiver, an acknowledgment is created and sent to the block's sender. If a packet is lost during its transport in the network, the receiver will not send this

ACK and the transaction will be timed-out, resulting in the re-transmission of the whole block. Traditionally, the block's sequence number is incremented and the block is re-transmitted using the incremented number and the same TID as before. Packets arriving at the receiver with a lower sequence number than the one in the receiver's context are ignored. In other words, the sequence number was tied to a transaction ID (TID), each block that was assigned a TID incremented the TID's sequence number on block issuing or on block's re-transmission. To avoid the complexity of having the sequence number tied to a TID, a global sequence number counter is used. This counter is incremented each time a block is issued or re-transmitted. This guarantees that re-transmitted blocks will always have a higher sequence number than what they were previously assigned, but the overflowing of this counter on a to-be re-transmitted block should be taken into consideration.

3.14 Transaction-Flow ID FIFO queue initializer

On system reset, all TID and FID FIFO queues are inevitably emptied. However, the expected functionality is that these queues are full after reset, every ID is available because there are no transfers alive. This initializer starts sequentially enqueuing the available IDs to their corresponding queues on reset. This initialization procedure takes an amount of clock cycles equal to the total entries of the FIFO queues, which, according to section 3.8, is 512+64+16 = 588 clock cycles. During this period, it is advised that no transfers are issued to the RDMA, not only due to the fact that the FID FIFOs might be empty, but also because the ID queues do not support 2 enqueues per clock cycle. An acknowledgment arriving in the 512 clock cycle time-frame would produce an enqueue to the TID FIFO, which would collide with the initialization enqueue.

Chapter 4

Evaluation and Results

In this chapter we present the functional verification strategy, the performance evaluation of the design, as well as the resource utilization of the synthesized design. The tools used for both verifying the functionality of the QoS engine of the RDMA and synthesizing the design were Xilinx's Vivado Design Suite. All the test-bench simulations were created and run using this suite and another simulation of a particular module was performed using the C programming language. Apart from the simulations, the design was also evaluated in real hardware. Specifically, the whole RDMA design, including both the QoS part and the send unit, was implemented in the FPGA of Xilinx's Zynq Ultrascale+ MPSoC and was evaluated using various user-level programs, ran on the ARM A53 core of the chip.

4.1 Resource Utilization and Timing

In this section, the resource utilization of the synthesized design is reported, along with the achieved frequency. Using Vivado 2017.2, the RDMA hardware design was synthesized with Xilinx Zynq UltraScale+ MPSoC (xczu9eg-ffvc900-2-e) as a target FPGA. The synthesized design of the QoS part of the RDMA is comprised of a total of 13.313 LUTs, 5.113 CLB registers and it uses 22 RAMB36E2 and 2 RAMB18E2 primitives (total of 23 BRAM Tiles of 36kbit each). Detailed resource utilization of the individual modules can be found in table 4.1. Finally, a target frequency of 150 MHz was achieved and optimizations were deployed to further increase it. Despite reducing the levels of logic in the critical path, no improvements were realized in terms of frequency, due to higher cable fan-out in the new critical path.

As table 4.1 suggests, around 55% of the total Look-Up Tables (LUTs) and 60% of the total registers are used by the status registers. In general, the CLB Register primitives inside FPGAs are coupled with LUTs, in a single package, and one cannot use registers without leveraging the LUTs that come with it. This, along with the added complexity of updating each of the 1024 individual channels, explains the spike in LUT usage. The number of LUTs used would drastically drop

Module	Utilization		
	CLB LUTs	CLB Registers	Block RAM Tiles
AXI Slave	473	352	0
Transfer Segmenter	1600	383	0
Message Handler	616	26	0
Scheduling FIFOs	1953	403	1
Status Registers	7349	3072	0
Transfer Table	3	0	14.5
Metadata Table	94	152	4.5
Pending Transactions Table	0	0	4
Packet Creator	339	194	0
Metadata Table Arbiter	59	2	0
Sequence number generator	13	12	0
ID FIFO Initializer	26	12	0
TID FIFO	342	29	0
1x FID FIFO	35	16	0
4x FID FIFO	23	12	0
Statistics Registers	200	448	0
Total	13313	5113	23

Table 4.1: Resource utilization of the QoS part of the RDMA

if the status of the transfers was kept in BRAM instead. However, as mentioned in section 3.4, this would mean that every update to a transfer's status would require a read on this memory and a write operation in the following clock cycle, becoming a potential point of contention for 3 modules, 2 of which serve the processor's writes and reads to the RDMA. This contention would arise especially for small size transfers, but this is only true provided that the feature of reading multiple statuses with a single processor read is still supported. Dropping this feature completely avoids the read operation, but also undermines the potential performance of the RDMA (see section 4.3).

All in all, even with the spike of LUT usage of the status registers explained above, the QoS design remains lean in terms of resource utilization. Most block RAM tiles are leveraged by the Transfer Table, which is expected since this table supports 16 pages of 128 channels and each channel must contain a 256-bit descriptor. Every table is implemented as a dual-port memory, thus no extra block RAM tiles are wasted in triple-port alternatives.

4.2 Functional Verification

The functionality of the design was tested in simulation using Vivado 2020.1 in a layered approach:

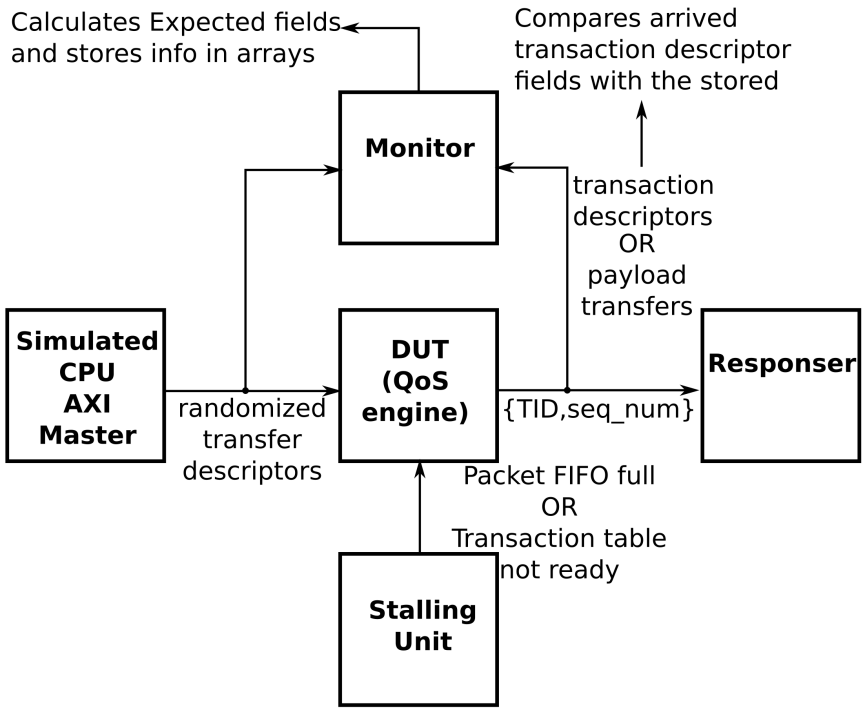

Figure 4.1: Functional verification of the QoS engine of the RDMA, high level depiction of the deployed test-bench.

i. Individual module test-benches

The first tests included test-benches written in System Verilog, created specifically for each of the modules under verification. In the case of the Scheduling FIFO queues, which required extra effort to support the 2 operations per clock cycle, as well as the back to back dequeue functionality, this module produced various logical loops during designing and thus additional testing was deployed. A simulated multi-priority FIFO queue written in C was created for this purpose. The two implementations were fed the same input operations, either a single enqueue, a single dequeue or both enqueue and dequeue operations in a clock cycle, and the outputs of both FIFO queues during dequeue operations were compared. The outputs were identical for the entirety of this test. Finally, the multi-priority scheduling FIFO queues in shared space implementation was further verified using traditional FIFO queues written in Verilog, with which the outputs were also in agreement during dequeue operations.

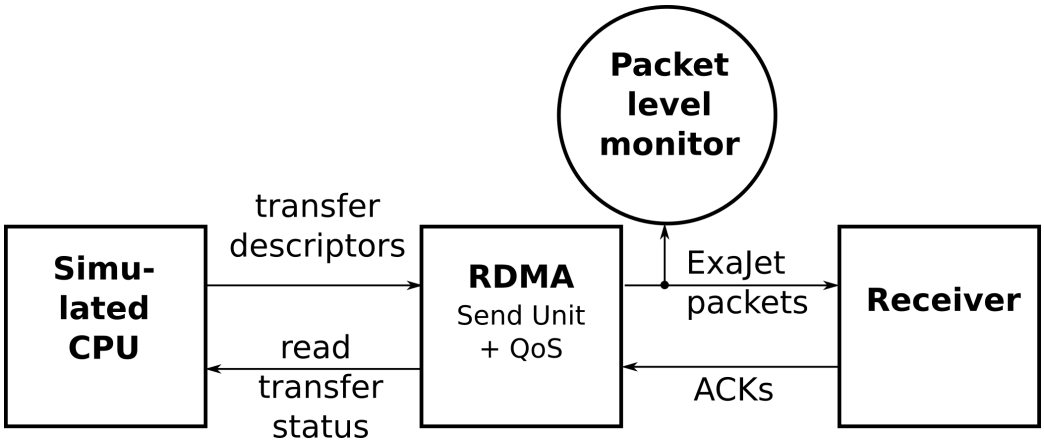

Figure 4.2: Integration of the RDMA send unit and QoS part test-bench.

ii. RDMA QoS design test-bench, send unit not included.

The second major functional verification test involved integrating the individual modules of the QoS engine and monitoring the outputs of the design. A high level depiction of this test-bench can be found in Figure 4.1. A simulated CPU issues randomized transfer descriptors to the Design Under Test (DUT), in this case the QoS, and at the same time, based on the newly created transfer descriptor, a monitoring module creates several entries of expected blocks along with their expected transaction descriptor fields (or in the case of inline payload transfers or control packets, the corresponding expected header/footer/payload fields of the created packet). When a block is issued or a packet is created by the DUT (remember that packets of payload transfers or control packets are created by the QoS part and enqueued to a Packet FIFO queue, from which the send unit dequeues them and sends them through the network), the Monitor compares the arrived fields with the expected ones. For every issued block, the Responser created an acknowledgment so as to further verify that the block issuing process does in fact proceed as expected and that transfers complete successfully. Support for acknowledgments arriving out of order was not verified in this test. Finally, a stalling unit was used to replicate the scenario of the output buffers being congested and, consequently the Packet FIFO queue being full, as well as potential stalls produced by the Transaction Table not being ready to accept a transaction descriptor write by the Segmenter.

iii. RDMA send unit and QoS integration test.

The final test involved integrating the existing RDMA send unit with the created QoS part and verifying the RDMA functionality, monitoring the output packets. The test-bench outlines can be seen in Figure 4.2. The simulated CPU's functionality was expanded to not only write transfer descriptors to

the RDMA, but to also issue read requests for transfer status polling. Acknowledgments were created by an actual receiver module instance and the scenario of out of order ACK arrival was ultimately verified in this test. This simulation proved to be extremely valuable since it offers the opportunity to accurately recreate most scenarios seen in hardware, giving the designer the ability to monitor every internal signal of the RDMA, an ability that a chipscope would struggle to offer.

4.2.1 Rate Results

To measure the message rate of the QoS engine, we examined the rate with which back-to-back, 8-Byte, inline payload transfers can be enqueued to the Packet FIFO queue, in simulation, using Vivado 2017.2. As Figure 4.3 suggests, the achieved rate is 1 transfer per 2 clock cycles. The Transfer Segmenter is able to serve both inline payload transfer and regular memory transfers with a rate of 1 transfer per clock cycle, but is ultimately limited by the 128-bit datapath of the AXI Slave (256-bit descriptors arrive in two 128-bit writes). However, support for Remote Read requests can increase the message rate to the Segmenter's capacity (1 transfer/clock cycle), since these requests can be served during the Segmenter's idle cycles. A more comprehensive breakdown of the test depicted in Figure 4.3 is the following:

- CC (0): AXI Slave write address channel handshake. The transfer descriptor's first 128-bit word address arrives, channel 0.
- CC (1): AXI Slave write data channel handshake. First 128-bit word of channel 0 arrives.
- CC (2): Second 128-bit word of channel 0 arrival. The transfer descriptor is written to the Transfer Table and the transfer is enqueued to its scheduling queue.
- CC (3): The Transfer Segmenter dequeues the transfer from the scheduling queue and indexes the transfer table with the address read from the head of the scheduling queue.
- CC (4): The transfer descriptor in channel 0 is now available to the Transfer Segmenter, which in turn creates an ExaJet RDMA packet and enqueues it to the Packet FIFO queue.
- CC (6-8): The RDMA send unit dequeued the packet from the queue and sends the packet's header, payload and footer in 3 consecutive clock cycles.

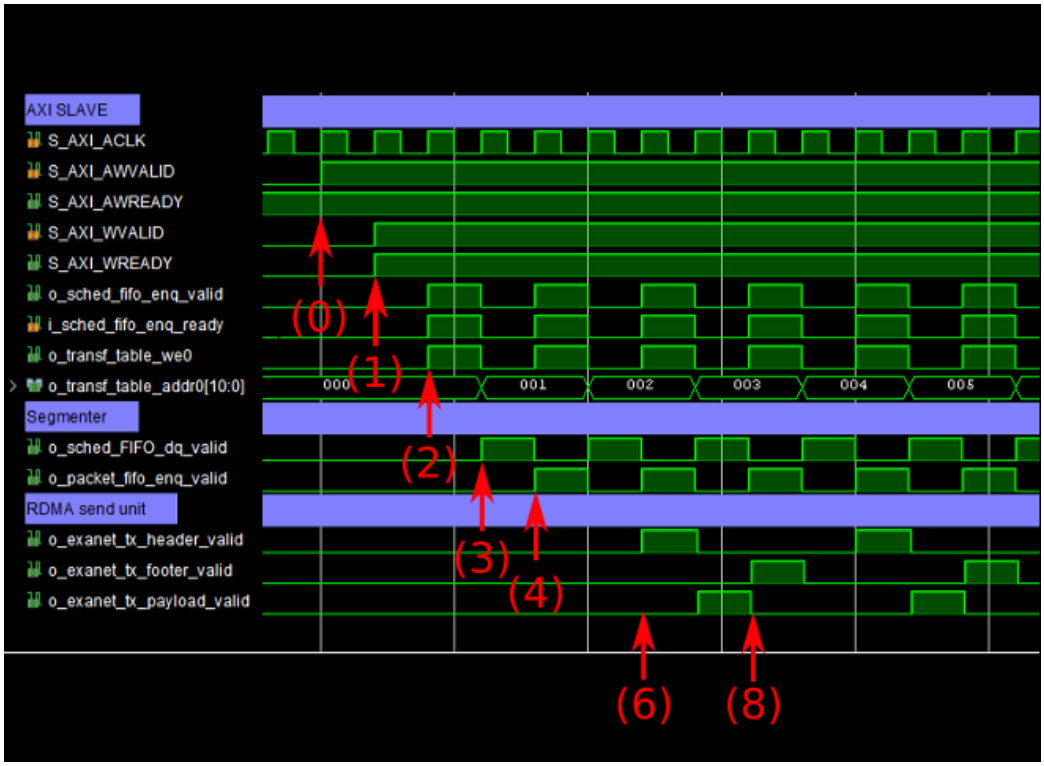

Figure 4.3: Simulation waveform of the RDMA engine in Vivado 2017.2. This test examines issuing back-to-back, 8-Byte, inline payload transfers in different write channels, in order to calculate the message rate. The QoS engine is able to enqueue the created packets with a rate of 1 packet per 2 clock cycles, limited by the 128-bit processor-RDMA datapath (256-bit descriptors). Serving Remote Read requests (when implemented) during the Segmenter's idle cycles will increase the message rate to 1 transfer/clock cycle. The RDMA's send unit needs 3 clock cycles to send the packet (header-payload-footer), but the total size of the packet is 256-bits header/footer + 64 bits payload = 320 bits, and the network datapath is 512 bits wide. It is therefore worth examining the possibility of it being sent in a single clock cycle.

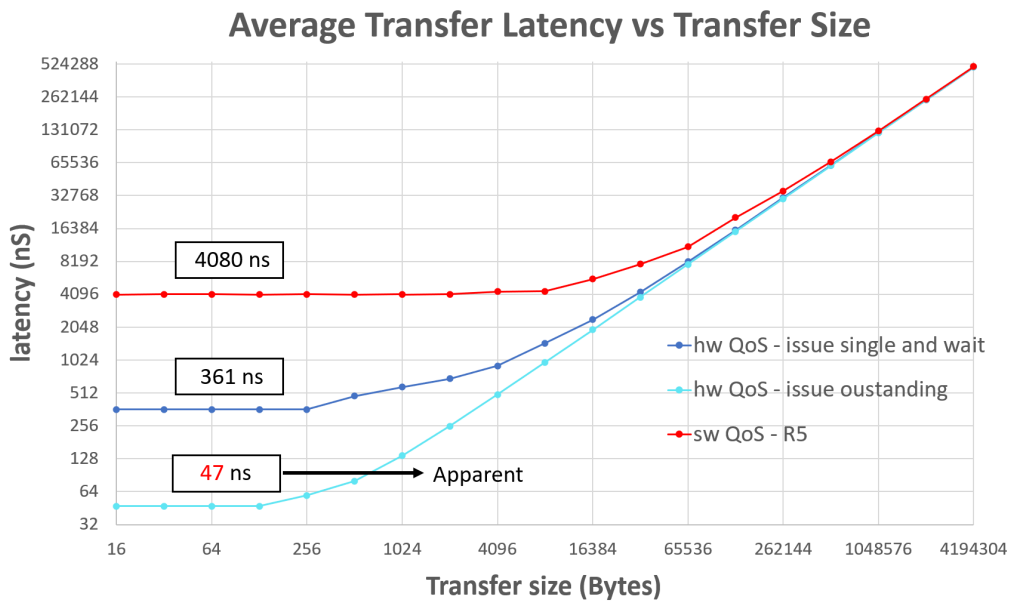

Figure 4.4: Average transfer latency for varying transfer sizes. Both axes are in logarithmic scale. The comparison is drawn between the existing hybrid RDMA, software implemented QoS that ran on the MPSoC's Cortex-R5 co-processor, and the new hardware implemented RDMA. Latency measurements on the hardware RDMA were taken using both modes of operation, by issuing a new transfer only when the predecessor is completed (blue curve), and by issuing multiple outstanding transfers and collectively polling their status (cyan curve).

4.3 Experimental Results

In this section, we present the average completion time of transfers, as well as the measured throughput of our RDMA implementation, both for various transfer sizes. All measurements were taken from user-level applications that ran on the ARM A53 core of the Zynq UltraScale+ MPSoC. The applications examined two possible modes of operation, one where all transfers are issued after the completion of their predecessor and one where transfers are issued in every available write channel of a particular page, and their status is collectively polled until every transfer is perceived as completed by the sender processor. Finally, a comparison is drawn between the new RDMA, both modes of operation, and the ExaNeSt RDMA on the same MPSoC, which used the co-processor R5 for QoS support.

All performance tests were conducted with the RDMA transferring data intra-node, and the data being written and read to and from BRAM. This allows us to observe the latency of the RDMA design itself, and not the added latency that characterizes the network and DRAM-RDMA path. The first test involved an ARM A53 core issuing a single transfer at write channel 0 of the Transfer Table

located in the PL of the Zynq MPSoC, the sender processor polling the transfer's status until the transfer is complete and then issuing the second transfer at write channel 1. A total of 64 transfers were issued in this manner, equal to the number of write channels in a Transfer Table page. This process was repeated for thousands of transfers to obtain an average completion time. The issued transfers did not include a completion notification, meaning that we measured the latency from the time of issuing the transfer until the point when all data have been transfered, the tranfer's status updated and read by the sender. This test produced the blue curves of Figures 4.4, 4.5 and 4.6.

The second test involved the same intra-node, BRAM to BRAM configuration but all 64 transfers were issued outstanding, without the predecessor transfer's completion blocking the issuing of a new one. This test produced the cyan curves of Figures 4.4, 4.5 and 4.6 and was mainly performed in order to demonstrate the the RDMA's throughput does benefit from issuing outstanding transfers, especially in the case of small and medium transfer sizes.

As Figure 4.4 suggests, the average latency for small size transfer in the new RDMA implementation (blue curve) is 10 times lower than the ExaNeSt RDMA implementation, at 360 ns. Considering that the PS-PL round trip time is measured at 120-150 ns, it is logical for a transfer's completion time to be 2-3 RTTs. The processor writes two 128-bit descriptor words to the PL and follows up with a status read operation on the target channel, immediately after the write operations. The first read will almost certainly return IDLE (or BUSY) status, as it arrives shortly after the descriptor writes and the transfer's block hasn't been issued yet. A second read operation is issued, since the returned status was not DONE (or ERROR), which will complete in another RTT. In the meantime, if an ACK has been received, the second read operation will return DONE status and the total completion time will be measured at 2 RTTs, else a third read will be issued.

The latency of small, outstanding transfers (cyan curve) appears to be lower than the actual PS-PL time distance, since we calculated it as the time for all 64 transfers to complete, over the total number of transfers issued. It should be taken as a demonstration of the average latency experienced by outstanding transfers. The actual latency of a transfer in our RDMA design is more accurately represented by the blue curve of Figure 4.4.

As for the throughput measurements, the RED-SEA RDMA is designed to operate on 200 MHz on the Zynq's FPGA, with a network datapath set to 512-bit, in order to achieve 100 Gbps throughput. However, the frequency of the RDMA design was set to 150 MHz (the QoS engine's maximum achieved) for this tests, which lowers the theoretical maximum to 76,8 Gbps. Furthermore, network packets are composed of 1 header, 1 footer and up to sixteen (16) 512-bit words of payload, meaning that out of the total 18 clock cycles the send unit needs to send the entirety of a packet, 2 clock cycles are header/footer overhead, which do not contribute to the overall throughput measurement. This further reduces the

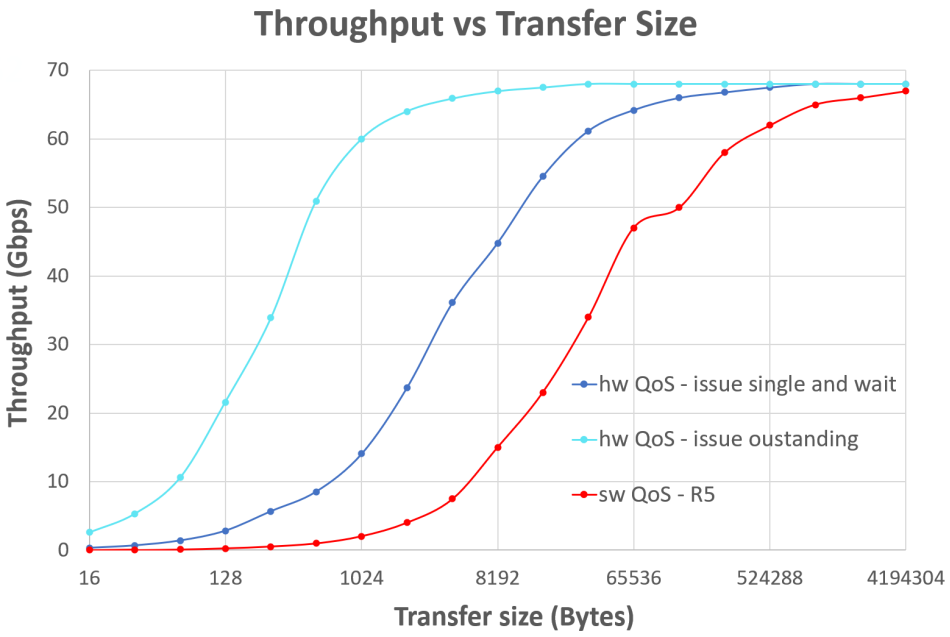

Figure 4.5: RDMA throughput vs transfer size. The x axis is in logarithmic scale. The maximum throughput that can be achieved is around 68 Gbps and both RDMA implementations eventually reach it. Issuing outstanding RDMA transfers greatly increases the throughput for small size transfers, thus the maximum throughput is reached way sooner for the cyan curve.

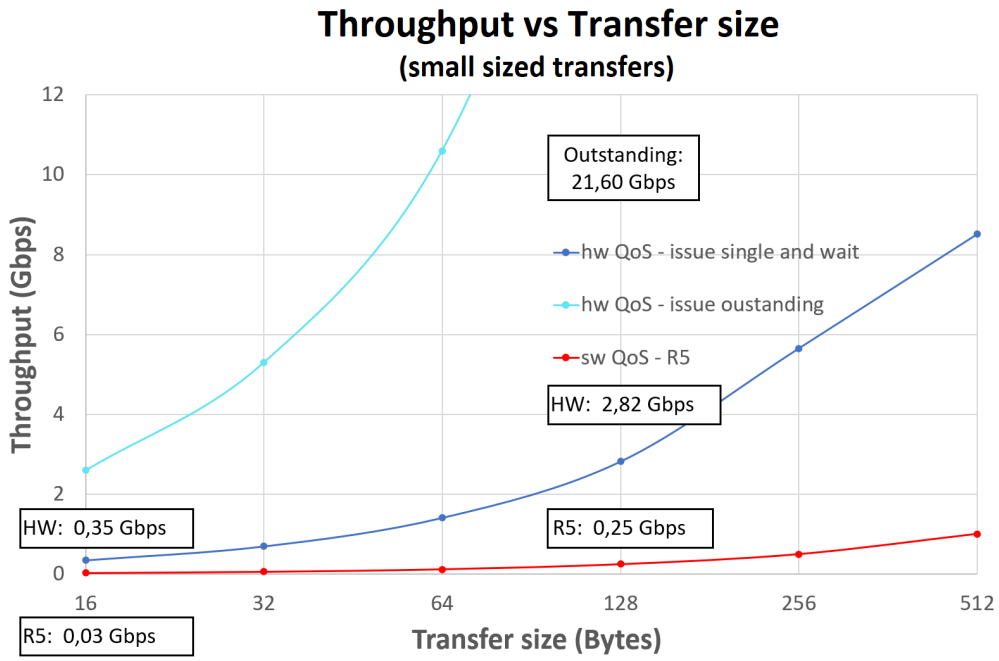

Figure 4.6: RDMA throughput comparison for small size transfers. The x axis is in logarithmic scale.

theoretical maximum throughput to 68,3 Gbps.

$$Max\ throughput = \frac{16}{18} \times 76,8 = 68,3\ Gbps$$

In Figure 4.5 we can see that both RDMA implementations eventually achieve the theoretical maximum throughput. The RED-SEA RDMA achieves it with transfers of 16 KB, when the CPU issues outstanding RDMA operations. At smaller transfer sizes (Figure 4.6), the new RDMA sustains high throughput, 21,6 Gbps for outstanding transfers of 128 Bytes and, even on the single issue mode, it achieves at least 10 times higher throughput than the ExaNeSt RDMA. Finally, we cannot reach maximum throughput with small sized transfers, since the payload size is comparable with header/footer size and because each transfer descriptor bears an overhead for its creation by the ARM A53 core.

Chapter 5

Conclusions and Future Work

The ever expanding number of computing units inside modern supercomputer clusters demands a solution to the corresponding increased communication overhead between their nodes, in order to achieve high performance. In this path, we presented a low-latency, high-throughput, hardware implementation of a new RDMA engine, developed under the EU-funded RED-SEA project. The work focused on providing an efficient Quality of Service hardware platform to the existing RDMA send unit, on which transfers are segmented into blocks, to enable selective re-transmission and multi-pathing, and small size transfers are prioritized to minimize latency, achieving a latency of 4 cycles for single block transfers, and a throughput of 150 MOP/s. We exposed to the user multiple channels that can host transfers of various types, including ones that completely avoid the RDMA-DRAM path.

Our evaluation demonstrated substantial improvement in terms of throughput and latency, the latter being 10 times lower than a previous hybrid RDMA implementation for small size transfers. In the functional verification process, we introduced a new framework, in which the designer/maintainer of the RDMA engine may replicate scenarios that arise in hardware and are hard to debug using conventional chip-scopes.

In order to obtain a scalable and resilient RDMA engine, the following functionalities must be implemented in future work:

- The addition of multiple sets of registers that accumulate transfer descriptors before they are written to the Transfer Table. A transfer descriptor is composed of multiple words that a processor cannot write with a single write operation. A single set of accumulating registers bears the drawback that interleaved descriptor writes to the RDMA, originating from different threads, cannot be supported, the interrupting descriptor write needs to be dropped. Adding a set per protection domain will solve this issue, however the case of interleaving in the same protection domain (context switch) must also be taken into consideration.

- Expanding the functionality of the Message Handler. In the current implementation, this module receives responses from the network, but has no way of handling negative acknowledgments. A re-transmission mechanism must be established. Furthermore, apart from responses, remote read requests also arrive from the network that this module should handle as well. A read channel allocator (priority enforcer) is also needed to facilitate these remote requests.

- Adding a time-out mechanism. A module that scans the Pending Transactions Table for timed-out transfers, issuing re-transmission requests to the RDMA send unit, is another essential expansion of the design.

- Dividing the critical path into more pipeline stage to achieve 200 MHz clock frequency. Although not essential for throughput (as the determinant factor for this metric is the send unit), reaching higher clock frequency in the QoS design would mean even lower latency. However, the number of clock cycles needed by the Segmenter to issue an inline payload transfer to the send unit is already 2 to 3, depending on payload size, which might change after improvements.

- Allowing the Transfer Segmenter to read from both ports of the transfer table when possible. This would allow for inline payload transfer of size between 8 and 32 Bytes (or 16 to 32 after transfer descriptor improvements) to be issued in a single clock cycle, by reading both descriptor lines in 1 clock cycle.

Terminology

control packet A network packet that the sender creates and sends to the recipient of the RDMA transfer. Along with the arrival of the last block of a transfer, this packets signals the receiver to create a completion notification.

ExaJet RDMA The new RDMA developed in the RED-SEA project, including the QoS engine designed in this thesis. This RDMA sends ExaJet network packets.

flow An entity larger than a transaction and smaller than or equal to an RDMA transfer. The current flow size is set to 4 blocks. A flow is subject to congestion management and packets of the same flow follow the same path inside the network.

packet A network packet, consisting of a header (128-bits), a footer (128-bits) and up to 1024 Bytes of payload. A transaction consists of up to 64 packets. In our current network, packets are not interleaved.

transaction A segment of an RDMA transfer, a block of 64 KB. The words *block* and *transaction* are used interchangeably in this thesis.

transfer The total amount of Bytes to be transferred from a sender node to a receiver node in an RDMA operation.

Bibliography

[1] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. An In-Depth Analysis of the Slingshot Interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2020.

[2] P. Vatsolaki M. Katevenis G. Kornaros, C. Kozyrakis. Pipelined Multi-Queue Management in a VLSI ATM Switch Chip with Credit-Based Flow Control. *IEEE Computer Soc. Press*, ISBN(0-8186-7913-1):127–144, 1997.

[3] Dimitris Giannopoulos, Nikos Chrysos, Evangelos Mageiropoulos, Giannis Vardas, Leandros Tzanakis, and Manolis Katevenis. Accurate Congestion Control for RDMA Transfers. In *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, 2018.

[4] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. Aquila: A unified, low-latency fabric for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1249–1266, Renton, WA, April 2022. USENIX Association.

[5] Ping-Jing Lu, Ming-Che Lai, and Jun-Sheng Chang. A Survey of High-Performance Interconnection Networks in High-Performance Computer Systems. *Electronics*, 11(9), 2022.

[6] Manolis Ploumidis, Nikolaos D. Kallimanis, Marios Asiminakis, Nikos Chrysos, Pantelis Xirouchakis, Michalis Gianoudis, Leandros Tzanakis, Nikolaos Dimou, Antonis Psistakis, Panagiotis Peristerakis, Giorgos Kalokairinos, Vassilis Papaefstathiou, and Manolis Katevenis. Software and Hardware Co-design for Low-Power HPC Platforms. *In the 5th International Workshop on Communication Architectures for HPC, Big Data, Deep Learning and Clouds at Extreme Scale (ExaComm'19) - in conjunction with the International Supercomputing Conference (ISC)*, 2019.

[7] A. Psistakis. Handling of Memory Page Faults during Virtual-Address RDMA. 2019.

[8] Antonis Psistakis, Nikos Chrysos, Fabien Chaix, Marios Asiminakis, Michalis Gianioudis, Pantelis Xirouchakis, Vassilis Papaefstathiou, and Manolis Katevenis. Optimized Page Fault Handling during RDMA. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2022.

[9] L. Tzanakis. Quality of Service Framework for Low Power RDMA Operations over Cortex R5 Real Time Microcontroller. March 2019.

[10] P. Xirouchakis. Design and Implementation of the Send Part of an Advanced RDMA Engine. March 2019.

[11] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. RDMA Performance Isolation With Justitia. 2019.