Quality of Service Framework for Low Power RDMA Operations over Cortex R5 Real Time Microcontroller

Leandros Tzanakis Arnaoutakis

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. Nikolaos Chrysos*

This work has been performed at and was supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS), Computer Architecture and VLSI Systems (CARV) Laboratory, within the ExaNeSt project, funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 671553.

March 2019 (Published in March 2020)

University of Crete

Computer Science Department

Quality of Service Framework for Low Power RDMA Operations over Cortex R5 Real Time Microcontroller

Thesis submitted by
Leandros Tzanakis Arnaoutakis
in partial fulfillment of the requirements
for the Masters' of Science degree in
Computer Science

THESIS APPROVAL

Author: ______________________
Leandros Tzanakis Arnaoutakis

Committee approvals: ______________________
Manolis Katevenis
Professor, Thesis Supervisor

____
Angelos Bilas
Professor, Committee Member

____
Polyvios Pratikakis
Assistant Professor, Committee Member

Departmental approval: ______________________
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, March 2019 (Published in March 2020)

Quality of Service Framework for Low Power RDMA Operations over Cortex R5 Real Time Microcontroller

Abstract

The High Performance Computing (HPC) contributes to the progress of science and the competitiveness of global industry. Nowadays, scaling the performance of supercomputers is limited by strict power consumption constraints. Low-power servers tightly coupled with high-speed FPGA accelerators can offer a feasible solution to deal with this challenge. Along this direction, the ExaNeSt EU-funded project develops and prototypes a system composed of power-efficient ARM-based processors, tightly coupled with FPGAs. In our system, we leverage the FPGAs in order to implement a custom low-latency interconnect that will allow computing nodes to communicate with each other as well as with fast, non-volatile, in-node storage devices. This creates the need for a sophisticated network interface to bridge the processes that run on the ARM cores with the interconnection hardware. For bulk memory-to-memory transfers, we have developed a custom low-latency multi-channel Remote Direct Memory Access (RDMA) engine, which allows processes to bypass the kernel in order to avoid the overheads of system calls and of traditional TCP/IP protocol processing. In this thesis, we have implemented in software, using a special Real Time co-processor, several stages of the RDMA protocol, including a novel transfer segmentation into blocks, per-block timeouts and retransmissions, quality-of-service (QoS), as well as end-to-end flow control and a novel protocol for fast completion notifications. The new RDMA supports per-block multi-pathing and selective (block-level) retransmissions, which advance InfiniBand state-of-the-art RDMA. The new RDMA, including the co-processor part, which is the outcome of this thesis, and the hardware part implemented at FORTH, is now fully functional, and has been used to run real HPC applications on the ExaNeSt prototype, which consists of tens of interconnected Ultrascale+ MPSoCs. By implementing several block and transfer level functions using the co-processor, we have reduced the complexity and development time of the RDMA, without affecting its processing rate. In this thesis, we report the features that we have implemented in the co-processor, the rationale behind our design choice, and system-level performance evaluation results.

Ανάπτυξη Λογισμικού Βελτίωσης της Ποιότητας Υπηρεσιών σε Μικροελεγκτή Πραγματικού Χρόνου για Λειτουργίες Άμεσης Προσπέλασης Μνήμης Χαμηλής Ενεργειακής Κατανάλωσης

Περίληψη

Η υπολογιστική υψηλής απόδοσης (HPC) συμβάλλει καταλυτικά στην πρόοδο της επιστήμης καθώς και στην αύξηση του ανταγωνισμού στην βιομηχανία σε παγκόσμιο επίπεδο. Στις μέρες μας, η απόδοση των υπερυπολογιστών περιορίζεται από αυστηρές προϋποθέσεις ενεργειακής κατανάλωσης. Ωστόσο η χρήση διακομιστών χαμηλής κατανάλωσης σε συνδυασμό με επιταχυντές FPGA υψηλής ταχύτητας μπορούν να προσφέρουν μια εφικτή λύση για την αντιμετώπιση αυτής της πρόκλησης. Σε αυτή την κατεύθυνση, το έργο ExaNeSt το οποίο χρηματοδοτείται από την Ευρωπαική Ένωση αναπτύσσει και παράγει ένα σύστημα που αποτελείται από επεξεργαστές αποδοτικής κατανάλωσης τεχνολογίας ARM, σε συνδυασμό με FPGAs. Στο σύστημά μας, χρησιμοποιούμε FPGAs για να υλοποιήσουμε μιας υψηλής ταχύτητας προσαρμοσμένη διασύνδεση η οποία επιτρέπει στους υπολογιστικούς κόμβους καθώς και στις γρήγορες μνήμες τους να επικοινωνούν μεταξύ τους. Αυτό δημιουργεί την ανάγκη για μια εξελιγμένη διεπαφή δικτύου η οποία γεφυρώνει τις εφαρμογές που τρέχουν στους πυρήνες ARM με το υλικό διασύνδεσης. Για μεγάλες μεταφορές δεδομένων μεταξύ μνημών, έχουμε υλοποιήσει μια προσαρμοσμένη πολυκάναλη μηχανή RDMA (Remote Direct Memory Access) με χαμηλή καθυστέρηση, η οποία επιτρέπει στις εφαρμογές να παρακάμπτουν τον πυρήνα του λειτουργικού ώστε να αποφεύγεται το κόστος κλήσεων συστήματος καθώς και η παραδοσιακή επεξεργασία πρωτοκόλλων τύπου TCP/IP. Σε αυτήν την εργασία, αναπτύξαμε λογισμικό, χρησιμοποιώντας έναν ειδικό επεξεργαστή πραγματικού χρόνου, διάφορα στάδια του πρωτοκόλλου RDMA, συμπεριλαμβανομένης μιας καινοτόμας τμηματοποίησης του μεγέθους μεταφοράς RDMA σε μπλοκ, εφαρμογή χρονικών ορίων και αναμετάδοσης ανά μπλοκ, ποιότητα υπηρεσιών QoS, καθώς και έλεγχο ροής από άκρο σε άκρο και ένα νέο πρωτόκολλο για ειδοποιήσεις γρήγορης ολοκλήρωσης. Η νέα μηχανή RDMA υποστηρίζει επιλεκτικές και πολλαπλών διαδρομών αναμεταδόσεις ανά μπλοκ, οι οποίες προωθούν την υπερσύγχρονη τεχνολογία RDMA του InfiniBand πρωτοκόλλου. Η νέα RDMA επίσης, συμπεριλαμβανομένου του τμήματος συν-επεξεργαστή, το οποίο είναι το αποτέλεσμα αυτής της εργασίας, και το τμήμα υλικού που υλοποιήθηκε στο ΙΤΕ, είναι πλέον πλήρως λειτουργική και χρησιμοποιήθηκε για την εκτέλεση πραγματικών εφαρμογών HPC στο πρωτότυπο ExaNeSt, το οποίο αποτελείται από δεκάδεσς διασυνδεδεμένων Ultrascale + MPSoCs. Με την εφαρμογή πολλών λειτουργιών επιπέδου μπλοκ και συναρτήσεων μεταφοράς χρησιμοποιώντας τον συν-επεξεργαστή, έχουμε μειώσει την πολυπλοκότητα και τον χρόνο ανάπτυξης πρωτοκόλλου RDMA, χωρίς αυτό να επηρεάσει το ρυθμό επεξεργασίας του. Σε αυτή τη διατριβή, αναφέρουμε τα χαρακτηριστικά που έχουμε εφαρμόσει στον συν-επεξεργαστή, το σκεπτικό πίσω από την επιλογή σχεδιασμού μας και τα αποτελέσματα της αξιολόγησης απόδοσης σε επίπεδο συστήματος.

Leandros Tzanakis Arnaoutakis ICS-FORTH,UOC

Ευχαριστίες

Πρώτα απ όλα θα ήθελα να ευχαριστήσω το ITE-ΙΠ-CARV για την χορήγηση υποτροφίας που μου προσέφερε στα πλαίσια του Ευρωπαικού χρηματοδοτούμενου προγράμματος Exanest H2020 Programme, Grant Agreement 671553. Έπειτα θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα μου Δρ. Νικόλαο Χρυσό για την διαρκή καθοδήγηση και αμέριστη συμπαράσταση του σε εμένα καθ' όλη τη διάρκεια της μεταπτυχιακής εργασίας μου. Επίσης θέλω να ευχαριστήσω τον σύμβουλο Καθηγητή μου Μανόλη Κατεβαίνη για την υψηλή εποπτεία του όλο αυτό το διάστημα καθώς και για την ευκαρία που μου έδωσε να εκπονήσω την μεταπτυχιακή μου εργασία στο CARV. Εκτός αυτών θα ήθελα να ευχαριστήσω τους παραπάνω, καθώς και τον Δρ. Βασίλη Παπαευσταθίου, για τον πλήρη σχεδιασμό πρωτοκόλλων επικοινωνίας που πραγματοποίησαν τα οποία υλοποιούνται στην παρούσα εργασία. Τέλος θέλω να εκφράσω τις ευχαριστίες μου στα μέλη του CARV Παντελή Ξηρουχάκη και Μιχάλη Γιαννιούδη, με τα οποία συνεργάστηκα και με βοήθησαν για την επιτυχή ολοκλήρωση αυτής της εργασίας καθώς και τους Κυριάκο Παρασκευά, Αντώνη Ψιστάκη και τα υπολοιπα μέλη του εργαστηρίου για την επί μέρους βοήθεια τους σε εμένα.

Στους γονείς μου

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern computing clusters consist of many heterogeneous computing units that work collectively in order to serve high computing tasks. Low latency communication between the remote processes that run on these servers is a critical factor for achieving high performance. In this effort, there is a demand of a sophisticated network interface, which is quite flexible to handle in an efficient way multiple concurrent tasks between remote nodes and memories that communicate each other, is considered as a necessity. However, this could succeed in case we manage to achieve remote RDMA transfers initiated by the user level processes instead of kernel API. This avoidance of kernel intervention gains better performance saving us from many time-effective CPU cycles overhead as a result of inevitable system calls. Subsequently, in terms of bypassing the kernel stack technique, a full custom, intellectual and flexible framework was developed which offers advanced Quality of Service (QoS) and resiliency features for RDMA transfers initiated by user level space. Particularly, this specialized software is implemented in a real time co –processor acting as a RDMA controller that interfaces the processes that run on the ARM cores with the interconnection hardware.

1.1 Motivation

In the road to Exascale, supercomputers and warehouse-scale machines adopting low-power, slim processors in order to reduce the power consumption of the fat processors used today. Although, power efficiency is very challenging attempt, simple, low-power processors tightly coupled with accelerators offer a feasible pathway. However, as the number of computing components increases, the inter-process communication bottlenecks become a crucial factor for exascale performance. This creates the need for a low latency and reliable interconnect that can support the communication among millions of computing cores, accelerators with memories and fast. At the same time, as the cycles of general purpose processors become all and more valuable, the industry is shifting to RDMA-capable networks that offload segments of the traditional kernel network stack to special hardware inside the network and the processor-network adapters. RDMA networks achieve lower-latency (due to zero copy copy transfers) and higher line rate, due to lower software intervention as the kernel is completely bypassed -- see Figure 1.1. As a result, data is transferred directly between application-processes to remote memory spaces without any extra packet copy overhead.

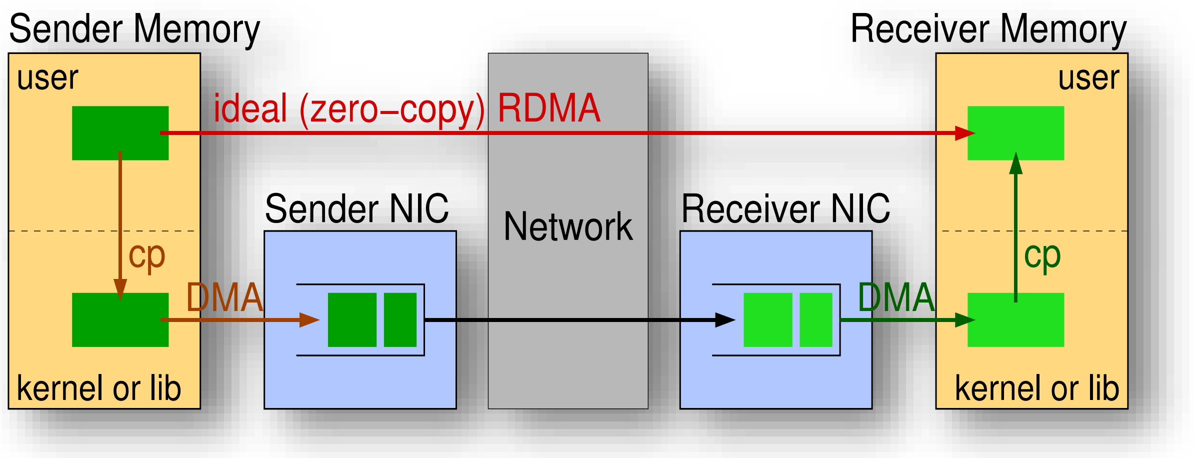


Figure 1.1: Zero copy User Level Initiated RDMA Transfer

In the framework of this thesis, we design a sophisticated network interface that bridges the user level processes with a special purpose interconnection network and initiates low-cost, fast and reliable RDMA transfers.

1.2 Contributions

This thesis has contributed to the design of a new Remote Direct Memory Access (RDMA) engine, suitable for user-level initiated memory-to-memory transfers in systems consisting of ARM+FPGA nodes working in a global virtual address space (GVAS). The RDMA engine leverages the System Memory Management Unit (SMMU) in order to translate process virtual addresses to physical memory locations, and provides multiple channels that can be allocated to user processes, in order to bypass the kernel overhead of traditional TCP/IP processing, and support InfiniBand-like capabilities in clusters of (ARM+FPGA) nodes. The RDMA supports advanced quality-of-service (QoS) and resiliency features, which we report in this thesis, together with our preliminary performance evaluation results.

The design of the new RDMA engine is split into a *software-programmable part*, which is the core of this thesis, and a *hardware part*, which is implemented inside Zynq Ultrascale+ Xilinx MPSoCs at FORTH. In this thesis, we implement a network co-processor using the Real-time ARMv7 R5 microcontroller that services a number of infrequent, (for hardware-speed), operations that are nevertheless critical for RDMA networks. These operations include transfer segmentation, end-to-end flow control, transfer scheduling, fast notifications, and reliable services. The complete RDMA design is now functional, running on the ExaNeSt-project prototype, which consists of Quad-FPGA-Daughter-Boards (QFDBs), interconnected in a Hybrid-Torus interconnect.

The RDMA design offers low-latency/high-bandwidth user-level read/write transfers, on par with InfiniBand RDMA, and other members of the group currently port MPI applications on top of it.

The author of this thesis implemented the following R5 Software modules that are responsible for the following processing steps of the RDMA:

- Provides *multiple virtual channels in a scratchpad* memory that are allocated to user processes using a newly-developed driver. We currently expose 1024 virtual channels located in 16 different memory pages of R5 scratchpad memory, which can be allocated to 16 processes running on A53 ARM cores. Each page (4KB) in the scratchpad accommodates 32 virtual channels for RDMA write operations and 32 for RDMA read operations, for a total of 64 channels per page.
- *Segments transfers into blocks* (or transactions), which are aligned to destination memory addresses, *collects the per-block hardware acknowledgements*, and implements a sliding window protocol and a *custom end-to-end protocol for fast completion notifications at the receiver,* with ½ RTT latency for small transfers, and 1 RTT latency for large ones.
- *Offers selective retransmissions* of failed or timed-out blocks, warranting *end-to-end reliable transfer* delivery. These offloaded resiliency features, which incorporate both software and hardware components, further obviate the need to use TCP at the communication end-nodes.
- *Implements a fast-path* that provide low-latency transmission, *speculatively* bypassing unnecessary (at low-load) processing stages. We also optimized the RT code, in order to minimize the overhead of the RT.
- *Prioritizes urgent transfers* over low priority ones, based on transfer size or user-hints. In addition, additional scheduling policies can flexibly be implemented in our software controller.
- *Provides an RDMA read protocol*, where the initiator issues special read request to the target node. The RT in that node, reads the request, allocates an available read virtual channel of its corresponding process, and initiates write transaction.
- In context of the *high-level overview of the RDMA* operation, author implemented separate mbox queues in order to avoid protocol-induced deadlocks.

Note that in this section 1.2 the above implemented communication protocols are completely designed by Dr.Nikolaos Chrysos, Dr. Vassilis Papaefstathiou, Prof. Manolis Katevenis and not by the author of this thesis.
In this thesis, author also :

1. coded several user applications that run on ARM cores and use the RDMA,

2. developed an API library and several kernel modules,

3. applied various low-level memory settings, in order to accomplish RDMA transfers in bare-metal as well as Linux environments in different prototype platforms.

An especially time-consuming task proved to be the continuous integration with concurrent hardware development of the RDMA engine, the network interface, and the interconnect. This step required dense software-hardware co-debugging actions, especially in cases where we were not confident for the source of the bug (platform, FPGA firmware, and RT controller). Another source of difficulty was the RT environment. Developing code for the RT controller required deep knowledge of the RT co-processor capabilities, and limitations, and the Zynq MPSoC, including coherency issues between the main ARM processors, L1 and L2 caches, the scratchpad of the RT and of on-chip memories.

1.3 Outline

The remainder of this thesis is organized as follows. The ExaNeSt project, its goals as well as its prototyping platforms are detailed in Chapter 2. In addition, in this chapter we report the software that was developed in order to run our experiments by user level processes and describe the application programming interface (API) of the software part of the RDMA network.The main design and the core of the RDMA-unit that we implemented are explained in chapter 3. In chapter 4, we present the structure of the resilience mechanisms and the quality-of-service (QoS) features that we support in our implementation. The verification and performance evaluation experiments that we conducted in the prototype platform are described in Chapter 5. Chapter 6 concludes this thesis, providing also a discussion of potential future work items that could be developed as further work. Following Chapters include bibliography and appendices with some procedural content of integration tools.

Chapter 2

Context

2.1 Exanest Project

The ExaNeSt is European Union funded project which develops, evaluates, and prototypes the physical platform and architectural solution for a unified communication and storage interconnect and the physical rack and environmental structures required to deliver European Exascale Systems. Building on years of advanced R&D knowledge, ExaNeSt is going to deliver the solution that can support exascale deployment in the follow-up industrial commercialization phases. Using direction from the ETP4HPC roadmap and high density and efficiency computations, Exanest project will model, simulate, and validate through prototype, a system with the following features.

First of all, high throughput, low latency connectivity between computing nodes with their (volatile) memories, (non-volatile) storage and their input/output (I/O) devices, in a way that all of them can cooperate tightly and effectively in solving huge problems with congestion mitigation, Quality of Service (QoS) and resilience guarantees. Furthermore, ExaNest offers support for task-to-data software locality models to ensure minimum data communication energy overheads and property maintenance in databases. The platform also ensures management scheme for big-data I/O resilient, unified distributed storage compute architecture while reducing energy, complexity, and costs. At the end, the project is going to demonstrate the applicability of the platform for the complete spectrum of Big Data applications, e.g. from HPC simulations to Business Intelligence support.

However, such large supercomputers connectivity already consumes a huge order of magnitude of energy. For this reason, the ExaNeSt EU funded project prototypes a system composed of power efficient ARM based processors tightly coupled with FPGAs contrary to power-hungry processors used by other HPC systems. In order to satisfy this need of power efficiency and low-latency in this thesis, UNIMEM architecture of Global Address Space from the EuroServer project is used allowing user level zero copy Remote Direct Memory Access (RDMA) operations for communication between nodes. Last but not least, RDMA leverages the System Memory Management Unit (SMMU) of the FPGA in order to translate process virtual addresses to physical memory adresses.

2.2 Hardware Resources

2.2.1 Prototype Platforms

This section presents the development platforms in which we implemented and tested the new software and hardware IP blocks. In Exanest project, we have built a prototype based on both commercial Trenzs boards and Quad-FPGA Daughter Boards (QFDBs) which feature the same Zynq Ultrascale+ FPGA's.

2.2.1.1 Trenz board

This board includes a Processing System (PS), which mainly for this purpose of this thesis embeds a cluster of 4x A53 cores and 2x R5 Real time cores. The commercial product featuring a Zynq Ultrascale+ is manufactured by Trenz. It is composed of an FPGA board codenamed TE0808, as shown on Figure 2.1, containing only strictly minimal peripherals, and a carrier board (codenamed TEBF0808). The FPGA board features 2Gbytes of DDR4-RAM and 64Mbytes of QSPI non-volatile memory. The carrier board hosts 2 SFP+ cages and a RJ45 connector. This hardware is funded by the EuroServer and ExaNode projects, however it is also used for developing the firmware and system software for the ExaNeSt prototype.

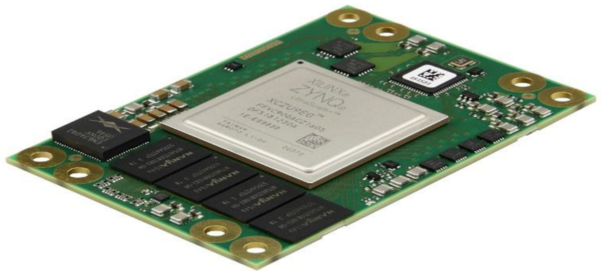


Figure 2.1: The Trenz TE0808 system-on-module (SoM), featuring a Zynq Ultrascale+ FPGA

2.2.1.2 Quad-FPGA Daughterboard (QFDB)

The Node is the Quad-FPGA Daughterboard (QFDB) containing four (4) Ultrascale+ FPGAs connecting each other hardwired with 2x High Speed Serial Links (HSS) in one all-to-all mesh topology as shown in Figure 2.2. Each FPGA features 2x 16MB QSPI and 16GB DRAM so that one (1) QFDB aggregates 64 GB of DRAM as well as 512GB SSD storage. Moreover, each QFDB provides a connector with ten (10) bidirectional HSS links (10 x 16Gbit/s = 160Gbit/s = 20GB/s) for high-throughput communication with other devices. Four (4) of those links are used to connect neighboring QFDBs hosted on the Blade. The remaining six (6) HSS links are attached to the external link cages (SFP+), mainly for connection with other blades within the same Chassis.

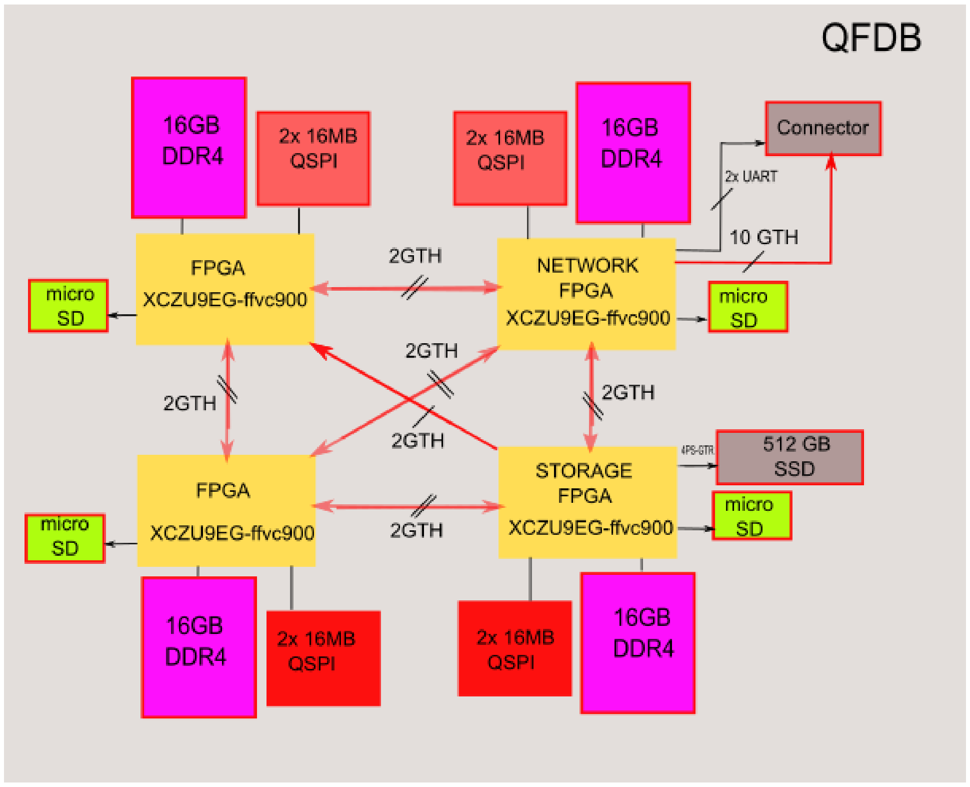


Figure 2.2: Quad-FPGA Daughterboard overview

2.2.2 Zynq Ultrascale+

In ExaNeSt we use the XCZU9EG-ffvc900 model of Xilinx Zynq UltraScale+ FPGA. A block diagram of this FPGA is shown in Figure 2.3. The Zynq Ultrascale+, includes the Processing System (PS) and the Programmable Logic (PL).

The Processing System (PS) features a Quad-core ARMv8 Cortex-A53 MPCore, clocked to 1.2 GHz which incorporates 32 KB of Instruction / Data Cache per core and 1MB of shared L2 Cache. It also includes a Real Time Dual-core ARMv7 Cortex- R5 MPCore clocked to 600 MHz with 32 KB of Instruction / Data Cache, 128KB of total Tightly Coupled Memory (TCM) as scratchpad and 256 KB of aggregate on Chip Memory (OCM). In addition, it supports DRAM controller for high-throughput external 16 GB of DDR4 SDRAM main memory with 32/64-bit width. System (I/O) MMU provides two stage translation (appropriate for virtualization), up to 48-bit physical address and also allows external devices to use virtual addresses, thus enabling user-level initiation of UNIMEM communication. The Programmable Logic is the area which is intended for hardware IP blocks development. This FPGA offers six (6) low latency AXI ports from PL to PS and two (2) AXI ports vice versa, and one (1) ACE port which offers cache-coherent accesses from the Programmable Logic (PL), as required by the UNIMEM architecture.

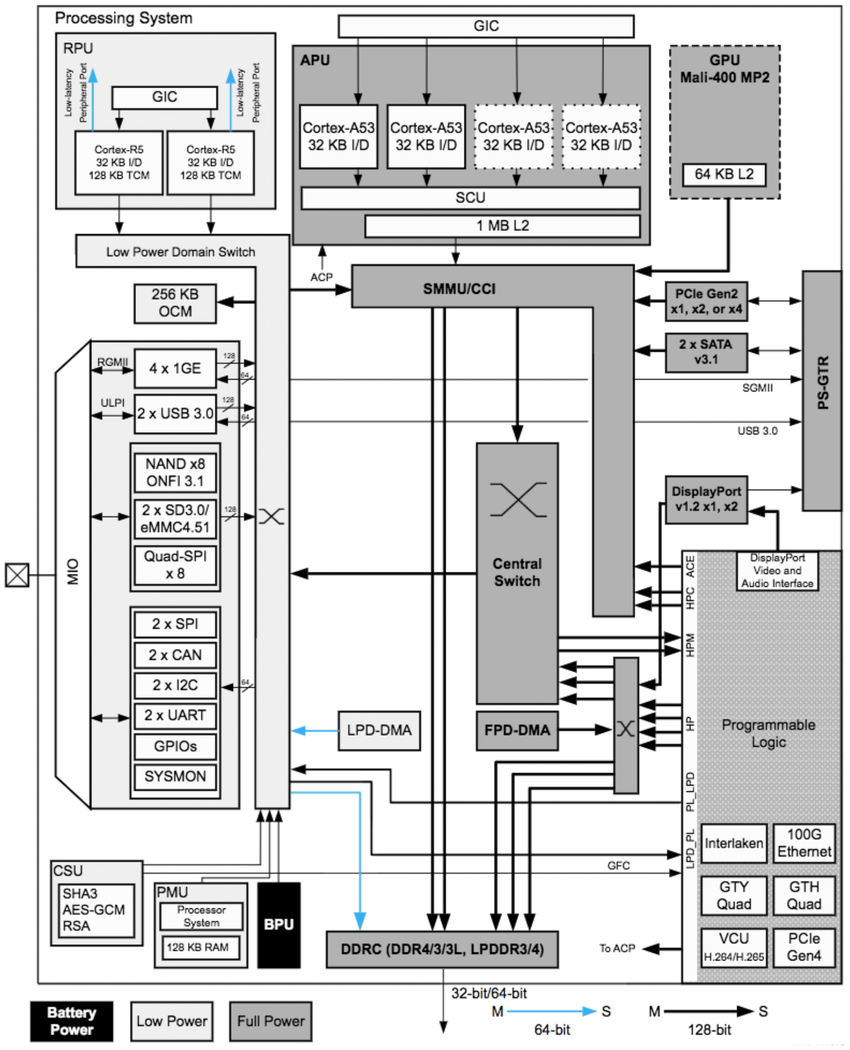


Figure 2.3: Zynq Ultrascale+ Top Level Block Diagram (source Xilinx)

The Zynq UltraScale+ MPSoC contains four (4) main power domains:

- Low-power domain (LPD)
- Full-power domain (FPD)
- PL power domain (PLPD)
- Battery power domain (BPD) Battery power domain (BPD) and 3rd processing unit Mali-400 graphics processing unit (GPU) with pixel and geometry processor and 64KB L2 cache, are not used for the purposes of this thesis

2.2.3 Real-time Processing Unit

The Zynq UltraScale+ MPSoC includes a pair of Cortex-R5 processors for real-time processing which are also based on the Cortex-R5F MP processor core from ARM. The Cortex-R5 processor implements the ARM v7-R architecture. The Cortex-R5F includes a floating-point unit that implements the ARM VFPv3 instruction set. In the Cortex-R5 processor, interrupt latency is kept low by interrupting and restarting load-store multiple instructions. This is achieved by having a dedicated peripheral port that provides low latency access to the interrupt controller and by having tightly coupled memory (TCM) ports for low latency and deterministic accesses to local RAM.

Despite of the fact that the Cortex-R5 processor is used mainly for safety-critical applications, it was used in this thesis in order to develop a QoS framework with resilience features playing the role of RDMA controller as part of the network interface. The most important features of Real Time Processing unit (RPU) can be considered the following:

1 Integer unit implementing the ARM v7-R instruction set
2 ARM v7-R architecture memory protection unit (MPU)
3 Single and double precision FPU with VFPv3 instructions
4 64-bit master AXI3 interface for accessing memory and shared peripherals
5 64-bit slave AXI3 interface for DMA access to the TCMs
6 Separate 128KB TCM memory banks with ECC protection for each TCM
7 32KB instruction and data L1 caches with ECC protection
8 32-bit master advanced eXtensible interface (AXI) peripheral interface on each processor for direct low-latency device memory type access to the interrupt controller
9 Performance monitoring unit

The RPU has two (2) Cortex-R5 processors that can operate independently-split or in lock-step together.

- Split mode operates as a twin-CPU configuration. Also known as performance mode.
- Locked mode operates as a redundant CPU configuration. Also known as safety mode.

When the Cortex-R5 processors are configured to operate in the lock step configuration, only one (1) set of CPU interfaces are used. Because the Cortex-R5 processor only supports the static split/lock configuration, switching between these modes is only permitted right after the processor group is brought out of reset. During the lock-step operation, the TCMs that are associated with the redundant processor become available to the lock-step processor. The size of each ATCM and BTCM becomes 128 KB with BTCM having interleaved accesses from the processor and AXI slave interface. In our case, we use RPU configuration in split mode.

2.2.4 RPU Tightly Coupled Memory (TCM)

Tightly-coupled memories (TCMs) are low-latency memories that provide predictable instruction execution and predictable load/store data timing. Each Cortex-R5 processor contains two 64-bit wide 64 KB memory banks on the ATCM and BTCM ports, for a total of 128 KB of memory. The division of the RAMs into two banks, and placing them on ports A and B, allows concurrent accesses to both banks by the load-store instructions, prefetch instruction, or AXI slave ports. The BTCM memory bank is divided into two 32 KB ranks that are connected to the BTCM-0 and BTCM-1 ports of the Cortex-R5 processors. There are two TCM interfaces that permit connection to configurable memory blocks of tightly-coupled memory (ATCM and BTCM).

- An ATCM typically holds code section of the program and interrupts or exceptions code that must be accessed at high speed, without any potential delay resulting from a cache miss
- A BTCM typically holds a block of data for intensive processing

The entire 256 KB of TCM can be accessed by R5_0 only in lock-step mode. In our implementation, we use R5_0 in split normal operation mode. Thus, the 2-bank 128 KB TCM support for each Cortex-R5 processor in the split mode includes the following:

- Each TCM is 64 KB
- One BTCM is composed of two ranks allowing interleaved accesses
- 32-bit ECC support is available in both normal and lock-step mode
- TCMs can be combined for a total of 256 KB (128 KB each of ATCM and BTCM) for use by R5_0 in lock-step mode
- External TCM access from AXI slave interfaces

2.2.5 On-chip Memory (OCM)

On-chip Memory is a 256 KB memory which resides in Low Power Domain (LPD) and provides low latency memory accesses from the RPU Core. In this thesis, OCM is used to store 3 class priority scheduler for pending DMA requests, essential data structures which are needed for bookkeeping 1024 pending transactions/blocks and some useful queues mainly for arbitration operations. In our current implementation, we use only first 64 of 256 KB of OCM memory, however these data could not be stored in scratchpad TCM due to space limitations.

2.3 Platform Software

In the context of this thesis, the most time of effort was spent in software design, development and testing of the framework. An important milestone was the effort that was conducted in order to overcome the limitations of R5 microcontroller, maintaining however high-performance requirements. Another time-consuming part of this work was the continuous testing and integration process with the related hardware so that the new RDMA engine would perform user level-initiated transfers in Linux environment both in Trenz boards and QFDBs. Profound understanding of the Xilinx Ulrascale+ MPSoC Technical Reference Manual (UG1085) was necessary for this implementation, as well as getting familiar with the software design tools that are provided by Xilinx.

In addition, both bare metal and user level programs were developed and configured respectively for testing and debugging needs. Some modifications and additions of generated boot-up files were also necessary so that the software integration to succeed. More detailed information about the process of integration in prototyping platforms is provided in appendices.

2.3.1 Design Tools

Xilinx is an American technology company, primarily a supplier of programmable logic devices. It is known for inventing the field-programmable gate array (FPGA) and as the semiconductor company that created the first fabless manufacturing model. Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs, superseding Xilinx ISE with additional features for system on a chip development and high-level synthesis. For the purposes of this work , Xilinx Software Development Kit (XSDK) was used which is the Integrated Design Environment for creating embedded applications on any of Xilinx's microprocessors Zynq UltraScale+ MPSoC.

Firstly, the version of the SDK was 2017.4 but later was downgraded to version 2017.2 for compatibility reasons with the hardware team.

2.3.2 Linux Booting Process

In context of running processes in user level space, Linux environment was needed to be used as Operating System. As first step, the main tool that was used in order to boot the OS into the quad A53 cores of Trenz boards and QFDBs platforms was Yet –Another-Tool (YAT). YAT is a useful tool developed by CARV-ICS and provides operations which builds the appropriate software of boot-up process. In detail, YAT tool supports the following features:

1. **Bootstrap**: Initializes environment and toolchains from scratch taking as argument the path to SDK directory. In our case, we used SDK version 2017.2

2. **Build_fsbl**: Builds a custom First Stage Boot Loader (FSBL) that boots Linux kernel image in A53 cores and loads R5 executable. This feature takes as parameters the board profile name (e.g trenz board, qfdb) and the path of the generated hardware definition file (HDF). The loading process of R5 executable differs in terms of Trenzs and QFDBs and it is described step by step in each board set up in appendices.

3. **Build_pmufw**: Builds the Power Management Unit Firmware

4. **Build_bl31**: Builds the EL3 Secure Monitor from ARM trusted firmware

5. **Build_dtb**: Builds the Flattened Device Tree image for the board (DTB) taking as parameters board profile and path of HDF as above description. However, DTB image creation needed to be modified in order to fit in the needs of SSMU translation process, otherwise transfers were blocked. Thus, a special set up was coming up both in Trenzs and primarily in QFDBs. The detailed process is also described in appendices.

6. **Build_uboot**: Builds U-Boot (second stage boot loader) taking as parameter only board profile

7. **Build_kernel**: Builds the Linux Kernel Image, in our case it was used pre-built Kernel version 4.4.9 in both platforms

8. **Build_initramfs**: Builds initial ramfs image

9. **Build_bootbin** : Builds BOOT.bin which was loaded into Trenz board A53 cores using SD Card Interface.

10. **Build_bootbin_all**: Builds BOOT.bin and all its above elements

11. **Build_package**: Builds the requested Boot Package –assuming above elements have been build.

12. **Build_package_all**: Builds the requested Boot Package and its above elements

Boot package method was used in order to boot linux with the appropriate hardware definition (HDF) and R5 executable in an already bootbin pre-flashed QFDB platform. More information about boot package loading process is given in appendices. HDF path, board profile name and boot package name (e.g plupdate) is also essential information provided by the user.

2.3.3 RDMA Software Interface

2.3.3.1 RDMA Semantics

Remote Direct Memory Access (RDMA) involves the capability of reading and writing operations in the memory of other nodes with significant CPU offload, while retaining the principles of memory protection.
Typically, the implementation demands the following minimal set of features:

a. **Remote addressing of other processors' memory**. This is the ability of the hardware to place data directly into the software application memory of remote processors.

b. **Asynchronous queues**. These queues are the common interface between the RDMA capable hardware and the low level user software. The Application Programming Interface (API) usually exposes a send-queue and an optional receive-queue or in other words a mailbox in a destination memory of the receiver.

c. **Kernel Bypassing**. User-level software initiates fast-path read/write operations interacting directly with the hardware without the intervention of the kernel-space. In this case, the arrival data should include the destination data as well as the destination memory address. This allows software to avoid the overhead caused by systems calls in low latency and high performance environments.

Next section presents RDMA Programming Interface API which was implemented in order to achieve user level zero copy initiated RDMA transfers in low latency environments.

2.3.3.2 RDMA Programming Interface

The software QoS framework that operates as the RDMA controller between the processes which are running on A53 cores and custom hardware in PL, developed in C language on Real Time Processor R5 in split mode, using the existing embedded hardware and memories as mentioned above. However, an application library was needed to be developed further which constitutes the Application Programming Interface (API) in order to initiate and test user level transfers.

For the purposes of this thesis, a system driver-module, which is called "scrachtpad_alloc", was also developed for the transfer initiation and fast path communication between R5 controller and the user process, which is running on Linux. This kernel module is responsible to allocate TCM 4KB pages of R5 and to assign them to corresponding processes. User should insert this module as well as an addition module called "exanest_virt" with destination board identifier as input parameter after booting up Linux process. The mentioned API offers the following set of function calls:

1. **alloc_comm_chan**: The application creates a handle for the process, with the appropriate addresses

2. **alloc_dmable_buf:** The application allocates source and destination buffers

3. **insert_descriptor_with_completion**: The application initializes all appropriate RDMA descriptors including source and destination buffers

4. **trigger_dma_transfer**: The application copies RDMA descriptors to corresponding virtual channel in TCM scratchpad and write the channel both in PL mailbox and fast path via TCM scratchpad in a special position in order to inform R5 software for the new initiated RDMA transfer.

5. **poll_dma_transfer**: The remote application polls a predefined destination memory address in order to check if completion data have been written there and afterwards reports the status of the transfer. The status of the transfer completion can vary; the following table 2.1 shows the different state cases.

Status	**Description**
DMA_FINISHED	Transfer completed successfully
DMA_ONGOING	Transfer is ongoing
DMA_PAUSED_NO_ERROR	Transfer has stopped unsuccessfully
DMA_FINISHED_WITH_ERROR	Transfer completed unsuccessfully

Table 2.1: Transfer completion state

Below we present the transfer structure which is used both by user level processes and by the R5 software framework. It is composed of the descriptors which are defined in user level process and initiate a RDMA transfer in a corresponding virtual channel at R5 TCM memory. Each field is 64-bit descriptor except for "done" and "transfer size" fields which are 32-bit. The aggregate space of the transfer structure is 64 bytes, which also corresponds to the size of each virtual channel register in TCM. This means every RDMA read/write request occupies 64 bytes in TCM scratchpad and abstains exactly the same distance from the previous and following channel. R5 controller reads the descriptors of the RDMA transfer structure which is copied in TCM and triggers the specified transfer. Each process can copy up to 64 successive transfer structures in its allocated 4KB page, that correspond to 64 virtual channels (64 bytes transfer structure * 64 virtual channels => 4KB = page/process).

```
Typedef struct dma_transfer {
        uint64_t src_address;
        uint64_t dst_address;
        uint64_t dst_adress_notification;
        uint64_t first_data_notification;
        uint64_t last_data_notification;
        uint32_t transfer_size;
        uint32_t done;
        uint64_t reserved_0;
        uint64_t reserved_1;
} dma_transfer;
```

A detailed explanation of each field follows:

- **src_address:** This field contains the source address of RDMA transfer in local memory. Substantially, this defines from where dma will read the data.

- **dst_address:** This filed contains the destination address of RDMA transfer in remote memory. Substantially, this defines where dma will write the data in remote memory.

- **dst_adress_notification:** This field contains the remote destination memory address where the remote process polls if the following data (first_data_notification and last_data_notification) have been written in order to check if the transfer has finished or not as it is shown in above table 2.1.

- **first_data_notification:** This is the first 64-bit data which have been written in previous memory address (dst_address_notification) when the transfer has finished.

- **last_data_notification:** This is the second 64-bit data which have been written in previous memory address (dst_address_notification), after the first_data_notification, when the transfer has finished.
- **transfer_size:** This descriptor contains the size of the write RDMA transfer in bytes which are going to be copied from the local to the remote memory or vice versa, in case of read RDMA. Each transfer cannot exceed 4 GB size. So, this specifies how much successive data should be copied starting from source address and how much far away from the remote destination memory the data will reach in bytes.
- **done:** This is field is used by the local user process to acknowledge the completion of a single RDMA transfer and also as status flag of the transfer. Firstly, the TCM memory is initialized with 0xffffffff value and when the "done" field takes the value 1 then the transfer has totally completed. The following table 2.2 shows the different values that "done" field can take during an end to end RDMA datapath and what is the meaning of the transfer status in each case.
- **reserved_0:** reserved 64-bit field for future use
- **reserved_1:** reserved 64-bit field for future use

Values	**Status**	**Description**
0	Initialized	Write transfer just initiated
1	Completed	Write transfer is totally finished
2	On Going	Write transfer has been initiated and is still working

Table 2.2: Write "done" field state

Chapter 3

Design and Implementation

3.1 Overview of the end to end RDMA Datapath

The software framework was developed in C using Xilinx SDK for R5 Real Time microcontroller. The project is created based on a Hardware Definition File (HDF) which is exported from Vivado design suite.
In a few words, RDMA engine implemented in three major blocks as shown on figure 3.1. The first is the software framework and is responsible for transfer segmentation and scheduling (QoS), as well as for the retransmissions of 16KB transactions or blocks of a transfer and developed in R5 microcontroller that resides in Processing System (PS). The second is implemented in Programmable Logic (PL) and is responsible for executing the transactions i.e. reading data from memory, correcting their alignment, and writing them to the destination in a custom protocol ExaNet packet format. Finally, the third block implemented also in PL and is responsible for bookkeeping the transactions at the receiver, monitoring their execution and delivering the negative/positive acknowledgments.

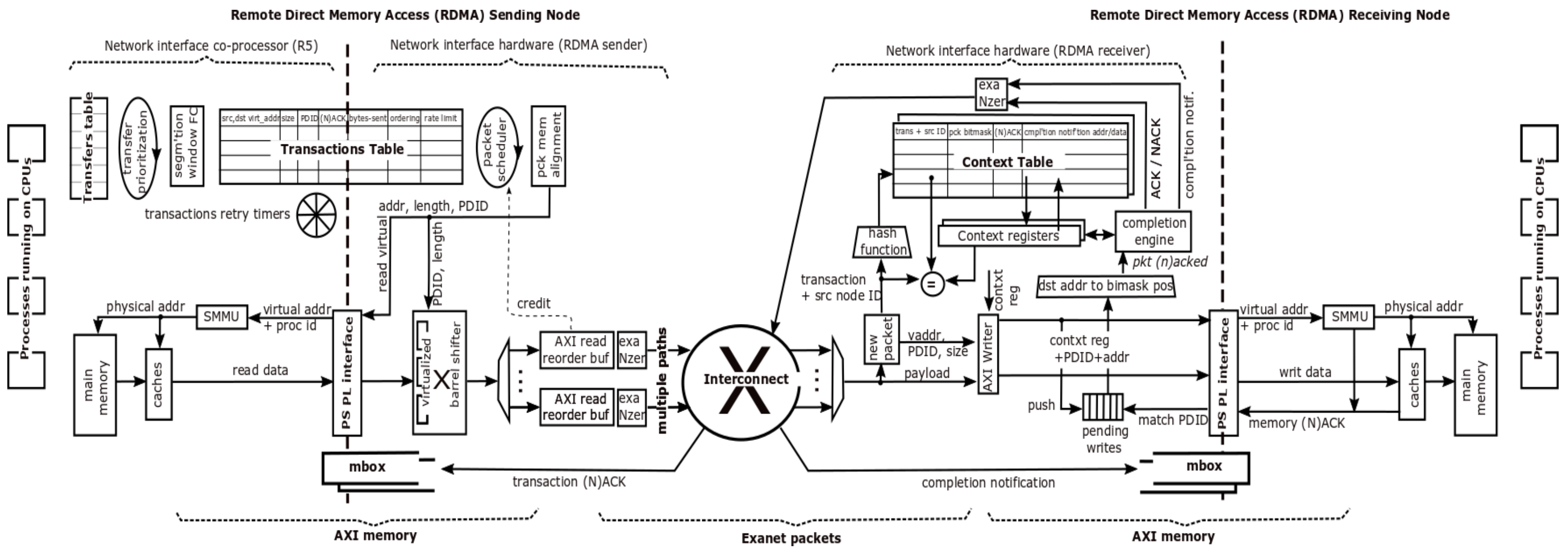


Figure 3.1: Overview of end to end RDMA Datapath

Following, it is described briefly an overview of the first block which contributes in this thesis. Initially, the user level processes that run on A53 cores initiate new transfers to 1024 available virtual channels allocated to R5 TCM memory. At this point, a mailbox at the PL is developed which receives messages from processes running on A53 cores and notifies R5 processor each time a new virtual RDMA transfer has been issued at R5 TCM memory. Subsequently, in terms of a sophisticated interface, a flexible scheduler issues the transfer, depending on its size, into a three (3) class priority queue. Latency sensitive transfers (size <=16KB) obtain the highest priority. Afterwards, the R5 processor takes this new transfer and divides it into segments of 16KB each, which we call transactions, according to 16KB aligned destination memory address.

Each time a transfer is selected by the scheduler, some of its transactions are initiated to DMA in PL based on a maximum threshold (number of transactions) and the transfer is rescheduled again. This is repeated until all transactions for this particular transfer have been issued. In addition, software keeps the transactions into a history list for future possible retransmissions. Specifically, when R5 receives a negative acknowledgement from receiver node for a particular transaction, it takes this transaction from history list and retransmits it.

Finally, R5 supports advanced resilience for timeout events of each RDMA transaction in a case of a packet loss or poisoning over the network. R5 compares the head transaction of the history list with a global time counter software in every interrupt tick and retransmits this transaction if it was expired. Extensive description of the framework implementation follows step by step in the next sections.

3.2 Write Operation

This section is a detailed explanation of the write RDMA operation and describes the steps that are needed to be followed, so that each transfer being initiated until being completed. The figure 3.2 below is a top-level diagram of a write RDMA transfer focused mainly in functionality of the framework that was implemented in R5 microcontroller.

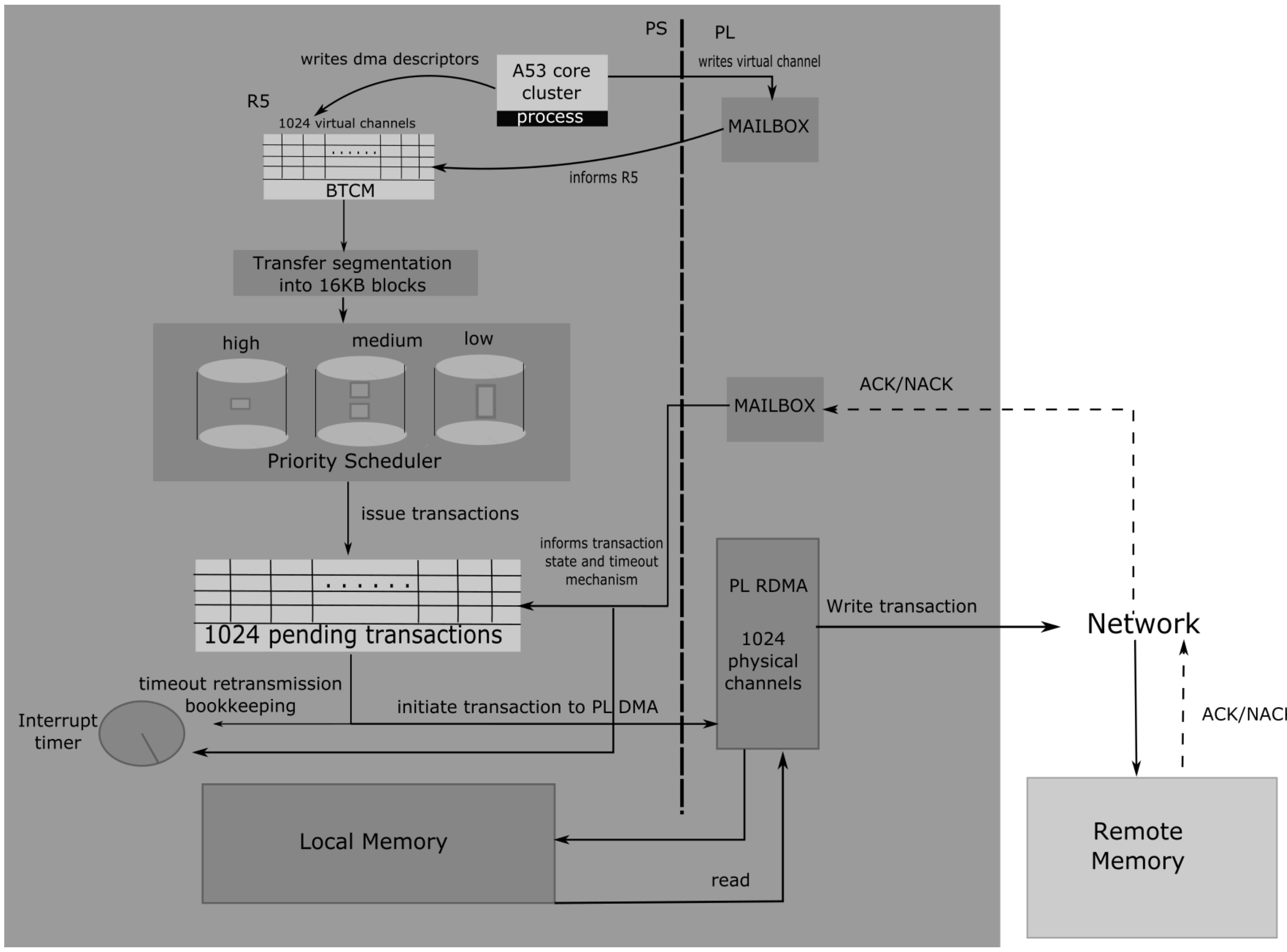


Figure 3.2: Write RDMA timeline

3.2.1 User Process Interaction

The user level process which runs over A53 core involves all appropriate steps that are needed and should be followed by a user level zero copy write RDMA transfer in order to be initiated and receiving completion notification in conjunction with R5 framework capabilities. The following steps were firstly conducted in bare metal without kernel calls on A53 core for immediate testing and debugging purposes.

First of all, each user process corresponds to only one (1) unique 4 KB page in BTCM in which there are 64 successive virtual channels that can accommodate 64 different RDMA transfers. Due to BTCM memory space limitations (64 KB total space), 16 unique processes can be supported in 16 successive 4KB pages of 64 virtual channels each.

That means 1024 virtual channels of 16 different processes can be supported simultaneously in BTCM memory of R5. Therefore, a user process should request and obtain from the operating system an available BTCM page as well as its corresponding protection domain id which is the serial number of the acquired page (0-15). In this implementation, 32 first virtual channels of a page are assigned for write RDMA transfers and the rest 32 channels for read RDMA requests. This is achieved by a Linux kernel module called "scrachtpad_alloc" which is responsible for allocating the BTCM pages to corresponding processes and managing this bookkeeping. However, if there are 16 running processes occupying 16 pages in BTCM (one page for each of them), then there are no other available pages for a new process apart from in case of a process is terminated or frees its committed page. Moreover, this kernel module is responsible for page allocation to a mailbox which is implemented in PL so that the user process can gain access to it.

If the above steps succeed, the user process asks to obtain source and destination virtual addresses regarding to their physical mapping in local and remote memory respectively. Afterwards, process begins to initialize the fields of the RDMA transfer structure initiation with the appropriate information. The RDMA transfer structure as was described in section 2.3.3.2 of Chapter 2, consists of "source" and "destination addresses", the demanding data "transfer size", the "done" status field as well as the "data notification" information and "destination address notification" of RDMA transfer and occupies space 64 bytes, as much as the exact size of a virtual channel. Now the process is ready to copy this RDMA structure to its corresponding BTCM page in one (1) of the first 32 available virtual channels of the page which are intended for write requests as shown Figure 3.3.

After process copies this transfer structure into the right virtual channel in its BTCM page, it should notify R5 processor that a new pending write transfer request is initiated in its scratchpad memory. This is achieved by writing into the mailbox base address, that was implemented in PL, the particular virtual channel number which is assigned to this transfer. R5 processor polls consistently this precise position of the mailbox in PL and checks if there is any new arrival in order to be informed from the process which virtual channel is initiated by the new transfer. After that, R5 takes the descriptors of this channel and begins the new transfer.

As it is mentioned above, the process has to calculate the exact position of the virtual channel in BTCM scratchpad according to the corresponding page in order to copy the RDMA descriptors which initiate the new transfer. This is achieved by using the following mathematical formula:

**BTCM base address + Protection domain id * PAGE_SIZE + virtual channel*transfer structure size/4 = Virtual BTCM address + virtual channel*64/4 =
Virtual BTCM address + virtual channel*16**

- BTCM base address is the memory address BTCM initially begins
- Protection domain id is the page number that corresponds to the process
- PAGE_SIZE is the size of a page = 4 KB
- Virtual channel is the number of the channel which accommodates the transfer
- Transfer structure size is the size of the RDMA transfer structure consisting of the descriptors and in our case is 64 bytes.
- Virtual BTCM address is the virtual 64-bit address that is returned by OS which points to the corresponding base address of the allocated page

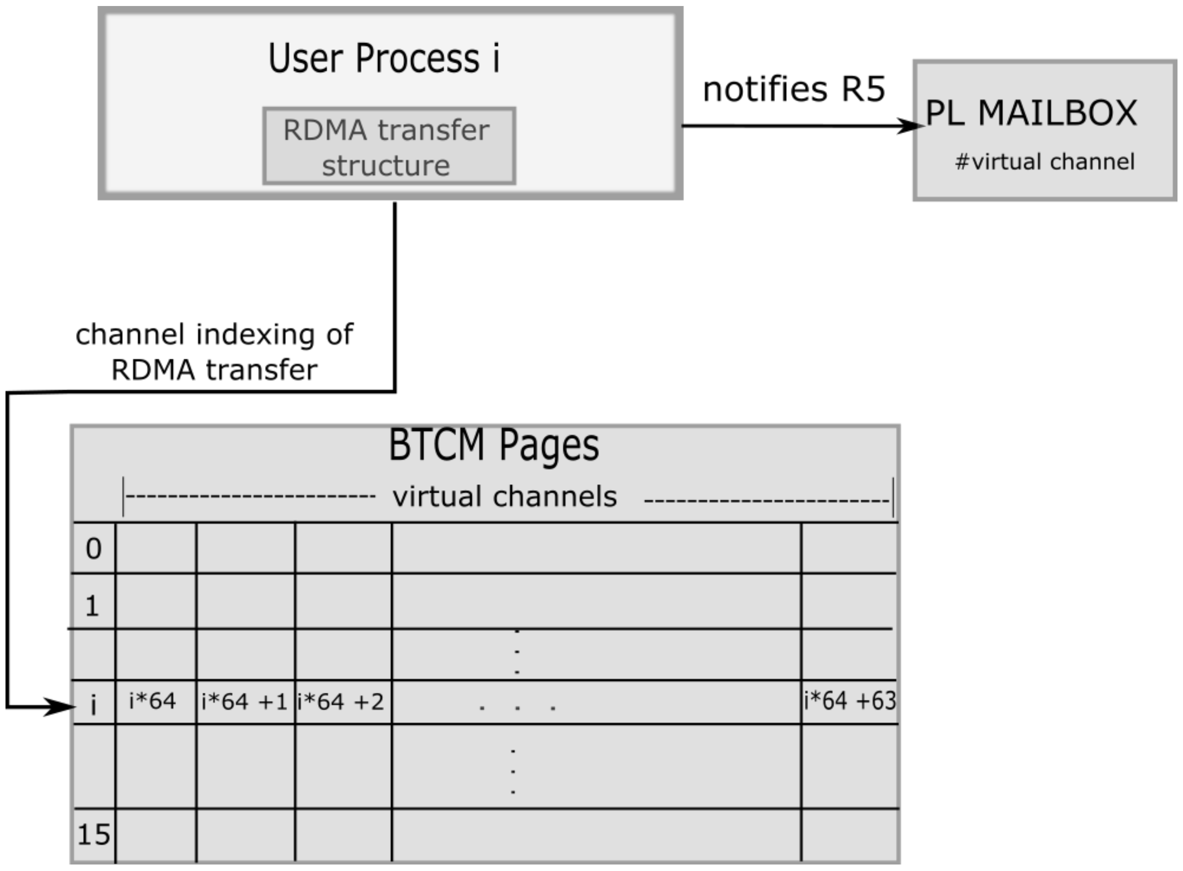


Figure 3.3: Indexing of transfer structure in BTCM

Initially the status of the "done" field as mentioned in section 2.3.3.2 in chapter 2, is "0" which means that the transfer is already initiated. When all pending acknowledgments of the transfer have been received through the network from the remote node in a new local PL also implemented mailbox, R5 software updates the value of the "done" field in "1", which means now the transfer is finished. So, the user process which polls this field can be informed for the transfer completion.

Furthermore, protocol supports a completion mechanism allowing hardware to inform faster the remote process when the transfer has been completed. This is achieved by writing a completion notification message, that is comprised of “first” and “last data notification” as well as “dst address notification” fields which are defined by the user, to remote node. When remote node receives this completion message generates the last acknowledgement and writes the “data notification” to “dst address notification”. In this way the remote process polls the “dst address notification” and is notified that the transfer is completed when the data notification has been written.

3.2.2 Transfer Segmentation

While FSBL loads the framework inside R5 core, BTCM and OCM memories are initialized with appropriate values. Soon after, the interrupt controller is initialized as well as the interrupt handling routines attached to the corresponding interrupt lines.

Afterwards, Real Time Processor R5 starts to poll the mailbox base address in order to be notified from A53 process if there is any new initiated transfer. As soon as, process writes the value of the virtual channel into mailbox, R5 reads this value and calculates using the following formula the exact position that indicates where the new transfer has been written into BTCM scratchpad.

DMA transfer index in BTCM :
btcm_base_addr + [(page_offset * protection_domain_id)/4) + (size of dma_transfer * (virtual_channel mod (VIRTUAL_CHANNELS_NUM_PER_PROCESS-1))/4]

- btcm_base_addr : BTCM base address
- page_offset: 4KB
- protection_domain_id: The number of the transfer page ranges {0-15}
- size of dma transfer: 64 bytes
- virtual channel: The value that process writes into mailbox and implies where the transfer is pointed in its corresponding BTCM page. Virtual channel values of write operation range: 64* protection_domain_id + {0-31}
- VIRTUAL_CHANNELS_NUM_PER_PROCESS: 64 defined by default

Therefore, R5 core finds the transfer based on above indexing and reads its size. In context of advanced QoS features, this framework supports a segmentation technique which divides the transfer size into segments 16KB each, which are called transactions or blocks, according to 16KB aligned destination memory address. In this way, this method specifies the first and last block size of each transfer as well as bookkeeps the total number of transactions which compose the transfer. Figure 3.4 below illustrates precisely the memory segments which arise from the above segmentation process in an unaligned 16 KB destination address.

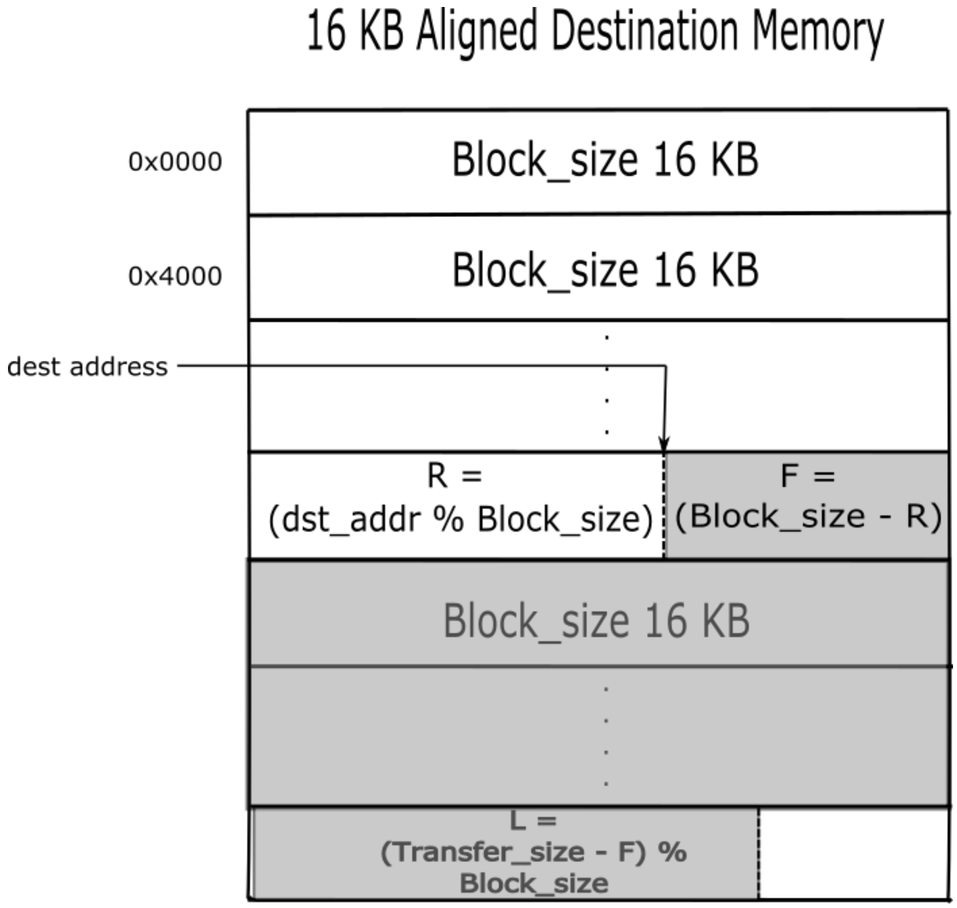

16 KB Aligned Destination Memory
0x0000
Block_size 16 KB
0x4000
Block_size 16 KB
dest address
R = (dst_addr % Block_size)
F = (Block_size - R)
Block_size 16 KB
L = (Transfer_size - F) % Block_size

Figure 3.4: Transfer size segmentation in 16KB aligned destination memory

As we mentioned each block or transaction size is 16 KB. However, in a case which the destination address is not aligned to 16 KB memory boundary, we need to calculate the size of the first and last block of the transfer. The first block size is the remaining bytes from destination address until the block boundary and it is estimated by the formula: **First block size = Block_size - (Destination Memory Address modulo Block_size)**. If the result of (destination memory address modulo block_ size) is equal to zero, then it means that destination memory address is aligned to 16 KB boundary and the size of the first block is the same with the block size. On the other hand, if the result of (transfer size – first block size) is less than zero then this means that transfer size is smaller than a block size 16 KB and it constitutes a single issue transaction.

In order to calculate the last block size, we need to find how much bytes are occupied of the last block size of the transfer. This is achieved by calculating the formula: **Last block size = (Transfer size – First block size) modulo Block_size**. If (transfer size – first block size) is multiple of block size, then the last block size is equal to zero. The aggregate number of blocks of a transfer is the sum of the first and last block in addition with the remaining number of 16 KB uncut blocks. This above information of first and last block sizes as well as the total number of blocks of a transfer is stored in an array for bookkeeping reasons for each virtual channel which points to the initiated transfer.

3.2.3 Transfer Scheduling

Soon after the segmentation of the transfer size, a three (3) class priority scheduler that is implemented in OCM is responsible to issue the transfer, depending on its size, into a circular queue. In this implementation, we have decided to provide highest priority to single issue, latency sensitive transfers (<=16 KB) in order to optimize the flow completion time. If there is no single transfer issue in the high queue, scheduler decides to serve a transfer from a medium queue. Scheduler is programmed to place a transfer in the medium queue in a case of its size is less than equal 512 KB. In any other case if high and medium queues are empty, the selected transfer is one of the biggest into the low queue.

Each time a transfer is selected based on its priority by the scheduler, some of its transactions of 16 KB are initiated to hardware DMA in PL based on a maximum outstanding threshold and the transfer is rescheduled again as it is shown by below figure 3.5 in a circular way until there are no other pending transactions of this transfer that have not been issued to hardware yet. Hence, the scheduling policy algorithm in steps is as follows.

1. R5 Controller checks the transfer size of the incoming DMA transfer

2. Enqueues that transfer to the corresponding queue according to its size

3. Chooses to issue the transfer in an order from highest to lowest priority. First of all, checks the elements of the high queue and if this is empty then checks the medium and if this is also empty checks the low queue.

4. If selected transfer is a single issue transfer (<= 16KB) from high queue, then controller issues this one to hardware. Otherwise, if the selected transfer is from medium or low queue, scheduler issues each time some of its pending transactions (<=16KB) based on a maximum configurable threshold which is usually two outstanding transactions.

5. If selected transfer has still pending transactions for issuing, this is re-scheduled again to its circular priority queue.

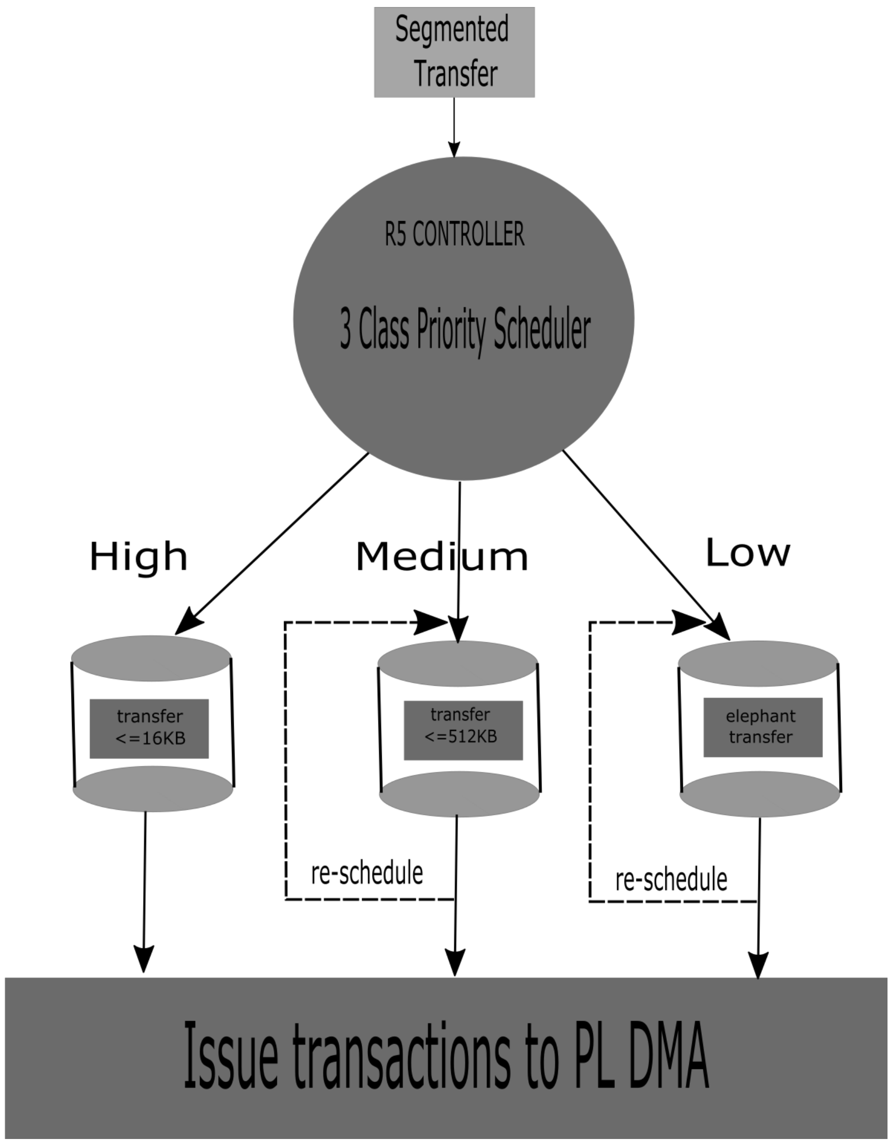


Figure 3.5: Three (3) class priority transfer scheduler

3.2.4 Transaction Bookkeeping

Since software chooses to initiate a new transaction of a transfer to PL RDMA, it needs firstly to write down some useful information. For this purpose, an array of structures called pending transactions table implemented in OCM in order to bookkeep the state of 1024 pending initiated transactions as shown the below figure 3.6. This bookkeeping of 1024 transactions is also mirrored to PL DMA in hardware. The state of each initiated transaction is composed of the following appropriate fields as follows:

```
typedef struct block_transfer {

        uint32_t transaction_size;
        uint16_t block_number;
        uint8_t state;
        uint16_t tick_counter;
        uint16_t transfer_id;
        uint8_t completion_notification;

} block_transfer;
```

1. Transaction_size: this field is the size of bytes of each initiated transaction

2. Block_number: this field is the sequence number of transaction that belongs to a transfer. For instance, if a transaction is the first to be transfered, it will take the number zero (0), the second the number one (1) etc.

3. State: this field declares the status of the transaction during its transferring and the taken values are listed in following table 3.1.

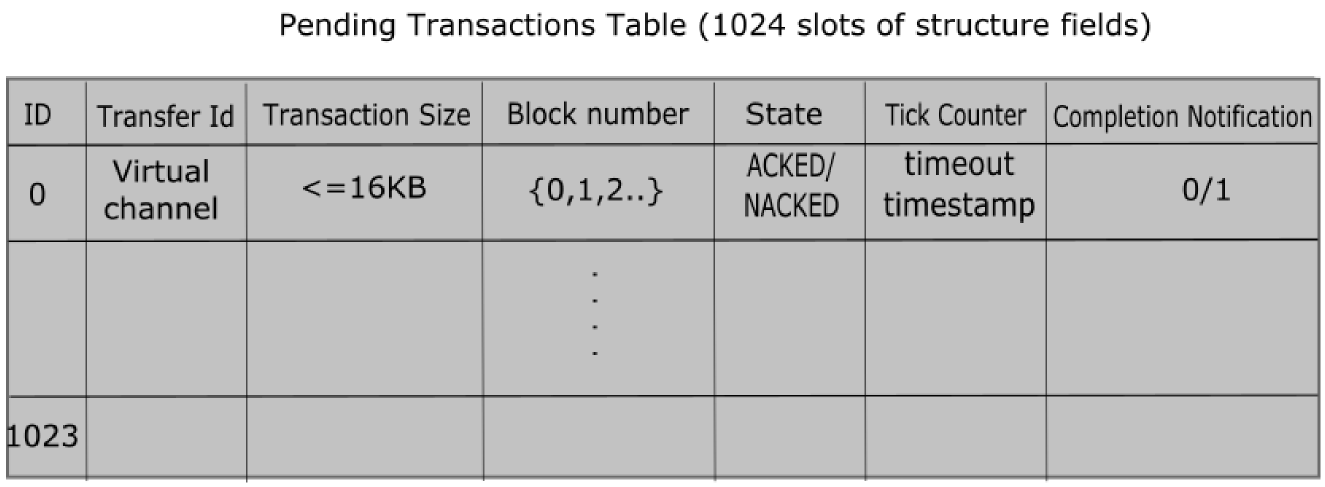
4. Tick_counter: this field represents either the number of interrupt ticks that are needed in comparison with a global tick counter or the number of processor cycles that have to be reached so that this transaction is considered as expired. This depends on timeout retransmission mechanism that is used.
   In first case, tick counter is assigned as the interrupt ticks since current time plus the timeout time is expressed in ticks. Interrupt tick period is configurable by PS clock. The equation is:
   tick_counter = current_tick_counts + timeout_ticks

   In second case, tick counter is assigned as the current real time processor cycles plus the extra cycles in absolute time that are needed in order for this transaction to be considered as expired. The equation is:
   tick_counter = r5_current_cycles + timeout_cycles

   Further details about timeout retransmission mechanisms in Chapter 5.

5. Transfer_id: this is the value that indicates the corresponding virtual channel of each initiated transaction in BTCM and is equal with the value of the virtual channel

6. Completion_notification: this field is a flag that notifies hardware that this is the last transaction of a transfer in order to generate completion notification message. This must be set to one (1) when software initiates last transaction of a transfer and be cleared with value zero (0) when software receives the last positive acknowledgement of this transfer.



Pending Transactions Table (1024 slots of structure fields)

ID	Transfer Id	Transaction Size	Block number	State	Tick Counter	Completion Notification
0	Virtual channel	<=16KB	{0,1,2..}	ACKED/ NACKED	timeout timestamp	0/1
			. . . .			
1023						

Figure 3.6: Pending transactions table

State	Values	Description
TRANSACTION_FREE	0	Transaction is not issued yet
TRANSACTION_ISSUED	1	Transaction is initiated to hardware
TRANSACTION_ACKED	2	Transaction has received positive acknowledgement
TRANSACTION_NACKED	3	Transaction has received negative acknowledgment
TRANSACTION_RETRANSMITTED	4	Transaction has received either NACK or timeout

Table 3.1: Transaction state

3.2.5 Transfer Initiation to PL RDMA

This section delineates the procedure which software follows in order to initiate a whole transfer to hardware implemented in RDMA in PL. As we mentioned in previous section, the above transfer scheduler checks each queue according to its level priority policy and if there is any existing transfer to issue, it conducts the above bookkeeping for each candidate transaction issued before initiates it to PL DMA.

First of all, if the selected transfer was dequeued from medium or low priority queue, means that this transfer is consisting of more than one transactions. In this case, R5 DMA controller follows the below steps:

1. Checks if this dequeued transfer has remaining transactions that are not issued yet and also if the outstanding transactions of this transfer do not exceed a maximum threshold. By placing limits on the number of transactions that can be transmitted at any given time, this operation acts as a sliding window protocol which allows two or more number of transactions to be communicated using sequential ids. If the above condition is true, then gets an available transaction id from a ticket based circular queue and assigns this id to the candidate issued transaction

2. Checks if this transaction is the first or the last one and sets its size. Otherwise transaction size is the default 16KB

3. Checks if this transaction is the first and last one of the same transfer simultaneously, meaning it is a single issue transfer, or if it is just the last transaction of a regular transfer which has received all the positive acknowledgements of its previous transactions. In both cases, software should assert a flag of completion notification message

4. Bookkeeps the fields that constitute the state of this transaction as it was described in previous section, before initiating this to PL DMA

5. Initiates the transaction to PLDMA transmitter as explained below.

6. Sends completion notification message to DMA transmitter if completion notification flag is asserted

7. Updates some temporary bookkeeping like new outstanding blocks and remaining unissued transactions of this transfer

8. Inserts this transaction id into the rear of a timeout list so that software is capable to retransmit this transaction if does not receive its acknowledgment during a timeout period

If first condition is still valid, repeats the same procedure. Otherwise, if there are remaining transactions that are not issued yet, reschedules this transfer to its priority queue again.

The first job that software needs to do in order to initiate a transaction/block to the PL implemented RDMA in step 6, is to determine transaction's source and destination addresses. This could be achieved by adding an offset in source and destination base addresses of the transfer in which this transaction belongs to. The offset can be calculated according to block number field of this issued transaction and is defined by the following formula.

offset = first transaction size + ((block_number -1) * 16384)

where block_number possible values ={1,2,3…}

Except for the case in which the issued transaction/block is the first of the transfer, the offset is zero (0). Afterwards, R5 controller makes three (3) 64-bit writes to a parameterized-based on transaction id PL DMA destination memory. These writes are comprised of transaction's source and destination addresses as well as the transaction size, protection domain id that corresponds to this transaction and completion notification flag that indicates if this is the last block of the transfer.

By sending completion notification message to DMA transmitter in step 7, R5 controller needs to make three (3) separate 64-bit writes to a parameterized-based on transaction id PL destination memory. The first one is the destination address notification, the second the first data notification and the third the last data notification of the transfer.

On the other hand, if the selected transfer was dequeued from high priority queue, means that is a single issue transfer (<=16KB). Due to the fact that in worst case scenario, when transfer is targeted on a 16 KB misaligned destination memory, it is separated in exact two (2) transactions with size <16KB, software doesn't need to check if the maximum outstanding transaction threshold is violated. In a case of a 16KB aligned destination memory the transfer issues just a single transaction with size <=16KB. All the above steps 2-10 of previous implementation for a medium priority transfer remains the same for a single issue transfer too.

3.2.6 Receiving Acknowledgements

This technique includes the steps should be followed by software in order to decode correctly a positive or negative acknowledgment for a transaction reaching through the network. Moreover, R5 controller takes over to notify a transfer that has completed, when it has received all its transactions acknowledgments. In context of completion notification mechanism, software in this section is committed to send completion message to DMA transmitter when it receives the penultimate transaction positive acknowledgment of a transfer whose the last transaction is already initiated.
The following routine explains step by step how R5 software handles the receiving positive acknowledgments from remote nodes through the network.

1. R5 controller polls a special for this purpose PL implemented mailbox in order to receive any kind of transaction acknowledgement which is sent by receiver node via network.

2. When routine receives a valid value from mailbox, checks if this value constitutes a positive or a negative acknowledgment. If the received value is a positive acknowledgment, extracts its transaction id in order to make the matching between the transaction and the mentioned acknowledgment.

3. Afterwards, framework drops out this positive acked transaction from timeout list so that this transaction does not take place in further retransmission.

4. R5 makes some appropriate bookkeeping updates such as marks the state of this transaction as ACKED, increases the acknowledgments and decreases the outstanding transactions which belong to the corresponding transfer located in BTCM

5. If receiving transaction acknowledgment is the penultimate of its transfer and r5 DMA controller has already initiated the last transaction of this transfer, software should send the completion message to transmitter which is obligated to transmit it to receiver node, otherwise software is informed to send completion message to receiver after initiating the last transaction.

6. In case of receiving transaction is the last one of a transfer, software should notify its transfer which located in BTCM scratchpad that is done and make some bookkeeping initializations of this transfer.

7. Last but not least, transaction id should be free so that it can be reused again by other initiated transaction

On the other hand, if the receiving mailbox value is a negative transaction acknowledgement, then r5 controller should make the following actions.

1. Extracts the transaction id from value

2. Updates the state of this transaction as NACKED

3. Drops out this negative (nacked) transaction acknowledgment from timeout list so that this transaction does not be retransmitted by timeout
4. Retransmits this transaction and sends completion notification message again if this is the last block of the transfer.

5. After retransmission of this transaction, R5 controller updates its tick counter field and puts it again into the timeout list.

The following table shows the different cases that receiver could return a negative acknowledgment for the corresponding transaction.

NACK ID (bits)	ERROR CODE
3’b001	AXI slave error
3’b010	AXI decoding error
3’b011	No context available
3’b100	Packet corrupted
3’b101	Protection domain mismatch
3’b110	Reserved
3’b111	Destination bufer full

Table 3.2: Negative acknowledgments cases

3.3 Performance Optimizations

In order to achieve a better performance in our design, a deep breakdown analysis required in our framework, for revealing where and why most of the time is consumed. Thus, the network interface was divided in three (3) main parts, and so we managed to infer valuable time results using R5 processor performance counters as a measurement tool. The first part was from the moment that a single issue transfer initiated by user level space until R5 is notified, the second one was from the moment that R5 informed for a new transfer before initiates it to PL DMA and the third was the time needed between transfer initiation to hardware until taking the acknowledgment from mailbox.

In the following paragraphs, we examine some feasible performance optimizations and alternative techniques that took place in R5 framework as shown below figure 3.7 compared to figure 3.2, as well as further proposals for achieving lower latency RDMA transfers as future work.

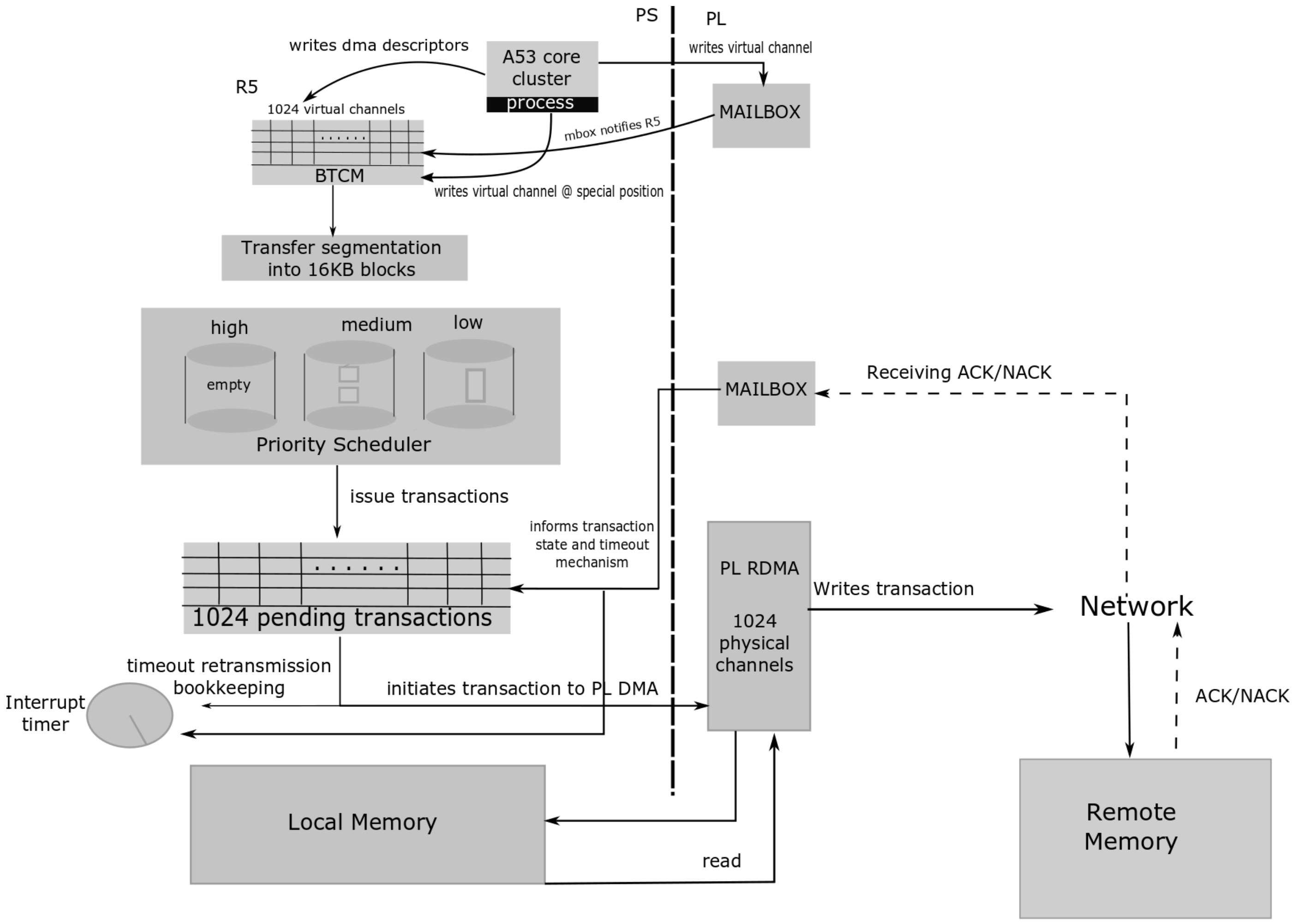


Figure 3.7: Optimized write RDMA timeline

R5 is notified firstly in scratchpad instead in AXI mailbox (PL) - Fastpath

First of all, a hybrid technique called fast-path which combines cache level R5 scratchpad memory in conjunction with PL mailbox allows R5 being notified faster by a user level process. In this technique, all processes notify DMA controller for their new initiated transfers both in a special position in scratchpad and in PL mailbox. However, one process will manage to access successfully this position in scratchpad and the rest of them will notify R5 via mailbox. R5 controller firstly reads this special position and soon after the mailbox, dropping existing duplicates of already initiated transfers.

In this way, software begins a new transfer earlier avoiding wasted latency effective reads in PL mailbox. Regarding to real time measurements that conducted using real time performance counters, this optimization was managed to reduce latency around 200 nanoseconds.

Moreover, one alternative optimization based on communication between R5 controller and user process could be the following proposal. Last page of BTCM scratchpad can be separated in 15 successive 4-byte process channels that correspond to 15 different processes. Each user process writes the virtual channel in which is going to initiate the new transfer into its own corresponding process channel. So, R5 which polls in a round robin way these process channels, is notified for a new transfer from each different process without the need of reading each time PL mailbox. In this way, R5 processor avoids to stall valuable working time. From the other side, user process in A53 core polls "done" field of each corresponding process channel in scratchpad and if transfer has completed then issues a new transfer for this process.

One more possible optimization could be accomplished by device driver which is responsible to allocate the pages of scratchpad. Each time user process orders to allocate its corresponding page in BTCM in order to initiate a new transfer there, a character device driver called "scratchpad_alloc" takes over to make this job. So, this driver could write directly the virtual channel in which process is going to initiate the new transfer in scratchpad. On condition that each time process will initiate transfers in successive virtual channels of its page, R5 can poll only the new ordered position of virtual channel in scratchpad avoiding time consuming overhead. This solution demands co-design between user space, device driver and R5 software.

Last but not least, in context of reducing communication latency between A53 and R5, we could implement a distributed queue in which user process using atomic operations (e.g. ADD/FETCH) will be able to put the number of the virtual channel that is going to be issued at scratchpad. On the other side, R5 processor could obtain and fetch this notification message about virtual channel which is initiated at scratchpad by distributed queue, without conducting time consuming reads in PL.

Scheduler Bypassing

R5 controller checks priority scheduler before initiates a new single issue transfer (<=16KB) and if there is no other pending transfer in high priority level, then bypasses the queue and issues this transfer directly into pending transaction table. This optimization contributes to further latency reduction around 300 nanoseconds.

Ticket-based circular transaction queue

Every time software issues a transaction to DMA, takes a transaction id based on a ticket-based circular OCM implemented queue in complexity O(1) instead of searching available ids in 1024 slots of pending transaction table. This saves us according to measurements around 200 nanoseconds. When this transaction takes a positive acknowledgment of this id, software inserts again this id into the rear of the queue.

Peripheral optimized counters

In order to avoid wasted time consuming non computational PS-PL RTT, we added some optimization counters that tell us which peripheral buffer, such as scheduling queues or timeout calendar queue or pending read request buffer implemented in OCM, is empty so that R5 controller does not need check that at all.

Retransmission bit

We write one distinct bit (57) in the third of 64-bit word of each transaction initiation to PL with the value zero (0). Thus, each time this block is needed to be retransmitted because of a timeout, software checks if this bit remains zero or has been changed by hardware with the value one (1). While this bit is still remaining zero, R5 controller doesn't need to initiate this block again to hardware because previous transmission hasn't finished yet.

Memory settings and Compiler optimizations

The code section of R5 frameworks as well as the interrupt line set up was stored in the first 64B part of scratchpad in ATCM. For the purpose of R5 controller to begin instantly DMA transfers, we assigned all the virtual channels in which each process initiates a new transfer into second 64B part of R5 cache level scratchpad BTCM. This achieved by configuring the linker script settings in XSDK project. Also, in order to gain further time improvement, we optimized compilation flag -O3 configuring build setting optimizations in the same project. However, sometimes compiler should be defined optimized based in program size so that R5 runs well. Last but not least, it is worth to pay attention on some stack and heap sizes limitations that should be also fitted well inside ATCM memory, making debugging process of R5 controller even more harder.

Dynamic transfer scheduling

In this part, we optimized the flexibility of the transfer scheduler for medium and low priority DMA transfers. When R5 controller completes the initiation of some transactions of a transfer instead of rescheduling the transfer again in its initial priority queue, now decides to put this transfer on run time into a new priority queue according to its new size without any explicit given input from user. In this way, R5 scheduler features a kind of dynamic priority policy.
This optimization does not affect the latency for single issue transfers because these are not subject to rescheduling policy, however contributes to the flow completion time reduce for bigger transfer sizes.

Due to TCM scratchpad limitations, we decided to put scheduler, timeout and transaction ids queues into the closest to R5's memory which is OCM.

3.4 R5 Memory Attributes

ARMv7 architecture provides a variety of memory page attribute settings in order to satisfy different memory operation scenarios. Cortex R5 is based on ARMv7 architecture and contains a Memory Protection Unit, (or MPU) that configures the attributes of a predefined number of memory regions.

3.4.1 Memory Protection Unit (MPU)

The Memory Protection Unit (MPU) works with the L1 memory system to control accesses to and from L1 and external memory. For a full architectural description of the MPU, see the ARM Architecture Reference Manual. The MPU enables you to partition memory into regions and set individual protection attributes for each region. The MPU supports zero, 12, or 16 memory regions. Attributes are only determined from the default memory map when zero regions are implemented. Each region is programmed with a base address and size, and the regions can be overlapped to enable efficient programming of the memory map. To support overlapping, the regions are assigned priorities, with region 0 having the lowest priority and region 15 having the highest. The MPU returns access permissions and attributes for the highest priority enabled region where the address hits.

3.4.2 Memory regions

Depending on the implementation, the MPU has a maximum of 12 or 16 regions. By modifying the file "mpu.c" which can be found inside the board support package (BSP) of the Xilinx SDK project, user can specify the following for each memory region:

- **Region base address:**

  The base address defines the start of the memory region. User must align this to a region-sized boundary. For example, if a region size of 8KB is programmed for a given region, the base address must be a multiple of 8KB. Note If the region is not aligned correctly, this results in Unpredictable behavior.

- **Region size:**

  The region size is specified as a 5-bit value, encoding a range of values from 32 bytes, a cache-line length, to 4GB. Table 4-34 on page 4-55 shows the encoding.

- **Sub-regions:**

  In each region can be split into eight equal sized non-overlapping sub-regions. An access to a memory address in a disabled sub-region does not use the attributes and permissions defined for that region. Instead, it uses the attributes and permissions of a lower priority region or generates a background fault if no other regions overlap at that address. This enables increased protection and memory attribute granularity. All region sizes between 256 bytes and 4GB support eight sub-regions. Region sizes below 256 bytes do not support sub-regions.

- **Region attributes:**

  Each region has a number of attributes associated with it. These control how a memory access is performed when the processor accesses an address that falls within a given region. The attributes are:

- Memory type, one of:
  - Strongly Ordered
  - Device
  - Normal
- Shared or Non-shared
- Non-cacheable
- Write-through Cacheable
- Write-back Cacheable
- Read allocation
- Write allocation

- **Region access permissions:**

  Each region can be given no access, read-only access, or read/write access permissions for Privileged or all modes. In addition, each region can be marked as eXecute Never (XN) to prevent instructions being fetched from that region. For example, if a User mode application attempts to access a Privileged mode access only region a permission fault occurs. Instructions cannot be executed from regions with Device or Strongly-Ordered memory type attributes.

3.4.3 Memory types

The ARMv7 architecture defines a set of memory types with characteristics that are suited to particular devices. There are three mutually exclusive memory type attributes:

- Strongly Ordered
- Device
- Normal.

MPU memory regions can be assigned a memory type attribute. Table 3.3 shows a summary of the memory types.

Note: The processor's L1 cache does not cache shared normal regions.

Memory type attribute	Shared or Non-shared	Description
Strongly Ordered	-	All memory accesses to Strongly Ordered memory occur in program order. All Strongly Ordered accesses are assumed to be shared.
Device	Shared	For memory-mapped peripherals that several processors share.
	Non-shared	For memory-mapped peripherals that only a single processor uses.
Normal	Shared	For normal memory that is shared between several processors.
	Non-shared	For normal memory that only a single processor uses.

Table 3.3: Memory attributes summary

Difference between Device and normal Memory type

The main difference between Device and normal Memory types is that normal Non-Cacheable memory is not looked-up in any cache and the requests are sent directly to memory. Read requests might over-read in memory, for example, reading 64 bytes of memory for a 4-byte access, and might satisfy multiple memory requests with a single external memory access. Write requests might be merged with other write requests to the same bytes or nearby bytes. Strongly-ordered and Device memory types are used for communicating with input and output devices and memory-mapped peripherals. They are not looked-up in any cache.

All of the processor interfaces to the external memory system have associated store buffers that help to improve the throughput of accesses to Normal type memory. Because of the ordering rules that they must follow, accesses to other types of memory typically have a lower throughput or higher latency than accesses to Normal memory.

In particular:

- Reads from Device memory must first drain the relevant store buffer of all writes to Device memory and wait for all Device writes to the relevant interface that have been posted onto the bus to complete
- All accesses to Strongly Ordered memory must first drain the store buffer completely and wait for all writes that have been posted onto the buses to complete.
- Read requests in normal memory might over-read in memory (read ahead) and write requests may be merged with other write requests in order to satisfy multiple write requests with a single memory access.

Memory attributes

Apart from the Memory types, there are other attributes, each with different functionality:

- Shared or Non-shared
- Non-cacheable
- Write-through Cacheable
- Write-back Cacheable
- Read allocation
- Write allocation

Shared data or Non-Shared:

Data marked as non-shared are non-cacheable. This bit only applies to **Normal** memory type.

Non-Cacheable:

Normal non-Cacheable memory is not looked-up in any cache. The requests are sent directly to memory. Read requests might over-read in memory, for example, reading 64 bytes of memory for a 4-byte access, and might satisfy multiple memory requests with a single external memory access. Write requests might be merged with other write requests to the same bytes or nearby bytes.

Write-Back Read-Write-Allocate:

This is expected to be the most common and highest performance memory type. Any read or write to this memory type searches the cache to determine if the line is resident. If it is, the line is read or updated. A store that hits a Write-Back cache line does not update main memory. If the required cache line is not in the cache the line is obtained from the local memory and stored into cache.

3.5 R5 Processor Drawbacks

This section highlights the handicaps of real time R5 processor which add extra overhead and limit the performance of the new RDMA engine. For this reason, we are going to describe the changes in settings that needed to be applied in order to overcome these difficulties making R5 framework as much as possible more compatible to low latency project requirements.

Coherency issues

Due to the fact that there is no cache coherence between A53 core cluster and real time R5 core cluster, some serious issues are coming up which related to inconsistency between their shared memories. For this reason, we needed to modify some crucial memory regions in MPU as we described above. First of all, we changed the 64KB memory region attribute of BTCM scratchpad as normal shared non-cacheable because this memory region should be shared between A53 cores, which expose new dma transfers in virtual channels there, and R5 which issues these transfers from there to hardware. On the other hand, BTCM should be non-cacheable due to the fact that there is no coherence between A53 and R5 clusters that causes inconsistency. In addition, we specified precisely ATCM memory region as normal non-shared write back write allocate which means cacheable because this memory region is used only by R5, so there is no consistency issue and we could cache its data. At the end, we defined the 256KB OCM memory region attribute as normal non-shared write back write allocate that means regular cacheable but non shared with A53 core cluster.

Double precision read/write operations from/to PL

The biggest handicap of real time coprocessor cortex r5 armv7-r edition that slows down dramatically the performance is its weakness writing/reading or storing/loading 64-bit or coupled 128-bit words to/from PL registers. Despite of the fact that Zynq Ultrascale+ technical reference manual refers that these instructions are supported by Real Time Processing Unit (RPU), specifically double precision FPU with VFPv3 instructions, we did not manage to apply this feature in order to write or read double precision words

to/from PL memory. Moreover, we tried to execute double precision load and store operations from/to address locations with vector multiple load/store inline assembly instructions such as VLDMIA/VSTMIA and LDM/STM as shows the following example including appropriate compilation flags -mcpu=cortex-r5 -mfpu=vfpv3 -mfloat-abi=softfp, however it did not work too.
e.g
asm volatile ("VLDMIA.64 %0, {D1,D2}\n" : : "r" (data) // data is address); // load 2 double precision words
asm volatile ("VSTMIA.64 %0, {D1,D2}\n" // Store 2 double precision words : :"r" (dst) //destination is address);

After systematic investigation we managed to succeed writing 64 bit words to PL using store buffer losing however some extra cost effective cycles in contrast of writing in order 32 bit words to PL which would added a pernicious overhead. The cache controller includes a store buffer to hold data before it is written to the cache RAMs or passed to the AXI master interface. The store buffer has three entries. Each entry can contain up to 64 bits of data and a 32-bit address. All write requests from the data-side that are not to a TCM or peripheral interface are stored in the store buffer. The store buffer has merging capabilities. If a previous write access has updated an entry, other write accesses on the same line can merge into this entry. Merging is only possible for stores to Normal memory. Merging is possible between several entries that can be linked together if the data inside the different entries belong to the same cache line. No merging occurs for writes to Strongly Ordered or Device memory.

The processor automatically drains the store buffer as necessary before performing Strongly Ordered accesses or Device reads. For this reason, we modified PL memory region attribute from MPU as normal memory non-cacheable and we took care to write cache line aligned words to store buffer. Therefore, we succeed to write multiple 64 bit instead of 32 bit words from PS to PL saving some cycles. Nevertheless, we observed that after writing the complete three (3) slots of store buffer, it takes about fifty (50) cycles for the next store of a 64-bit word to PL memory. So every time we write three 64 bit words for a PL-DMA initiation using store buffer in order to reduce this costly overhead.

Memory settings limitations

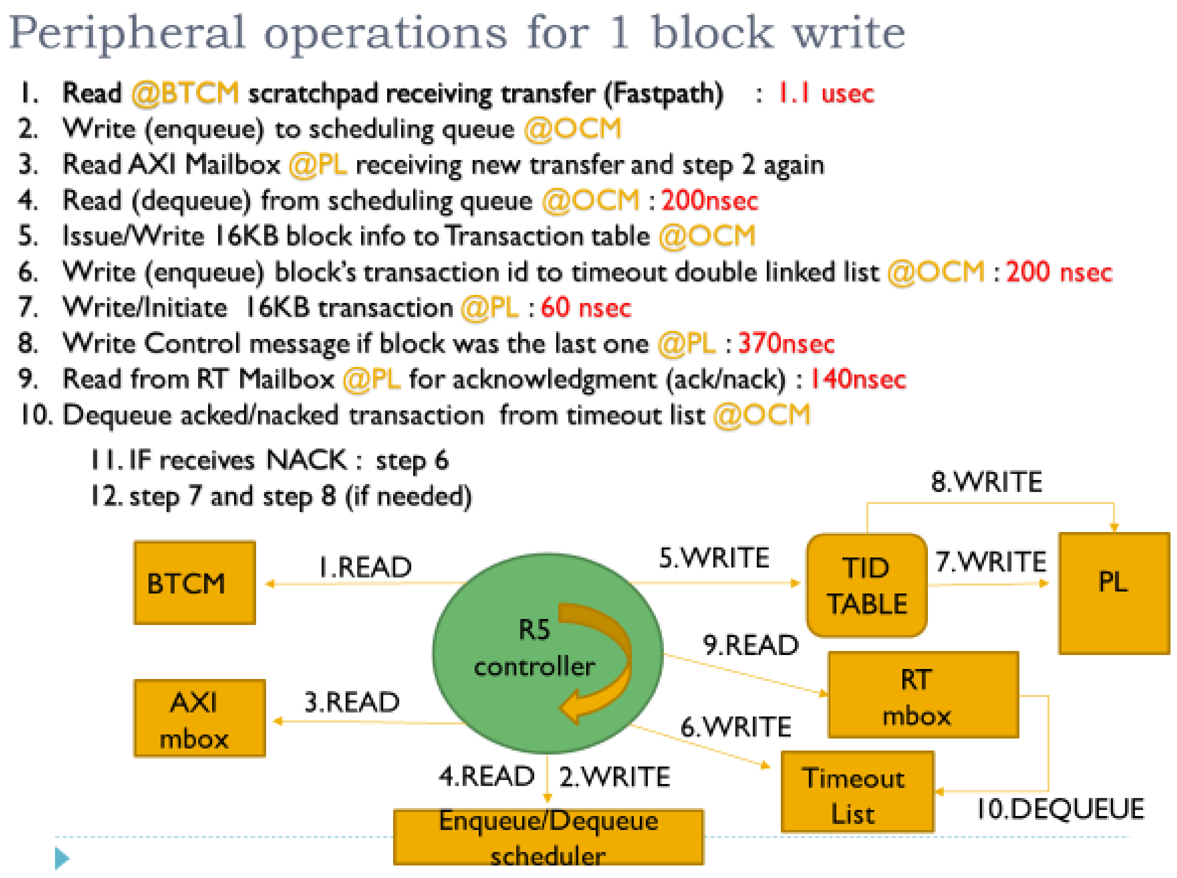
Another drawback of R5 processor was the need to define precisely the memory regions of code, interrupts, heap and stack sections and the size that should not exceed these memory boundaries. This made the testing process such a difficult work because the space of ATCM memory is limited (64KB) and we had no free space neither for printing commands. This fact makes R5 hard debugging. Also, when stack overflow happens due to no free space, compiler does not throw any error to console and the user does not know from where the memory violation is coming up.

Subsequently, we can break down the whole controller procedure into logical steps and highlight the methods that R5 controller calls in order to investigate which of them are more time consuming.

Main loop structure

```
While ( TRUE) {                                   // non blocking polling
Poll_btcm_and axi_mbox() {
   if  (BTCM_special_address  ==  new_vchannel){            // fastpath
          read_and schedule(transfer);}
   if(mbox_value != 0xdeadbeef){ dequeuer_and_schedule(transfer);}}
Select_transfer_to_issue_transactions() {          //avoid OCM reads
  if (high_priority_count >0){ dequeue_and_initiate_to_PL(); }
  elseif(medium_priority_count >0){ dequeue_and_initiate_to_PL(); }
  else(low_priority_count >0){ dequeue_and_initiate_to_PL(); }
  if(remaining_transactions){reschedule(transfer);} }
 Transaction_received_ack() {
 if (mbox_value != 0xdeadbeef)
          switch(mbox_value){
                    case "ACK/NACK": consume or retransmit transaction
                    case "read_request" : issue it to BTCM} } }
 Check_timeout_list(&timeout) :       // timeout retransmission handler
 if (expecting acks>0){read head(tid) of timeout list();              //avoid OCM read
                    if (head is expired) retransmit(head);}
 Check_pending_read_channels(pending_read_channel_table){
       if (pending_read_request>0) {                        //avoid OCM read
          assign_vchannel(read_request);
          issue_read request();} }   }
```

Using R5 performance counters and specifying some timestamps in several breakpoints of the above pseudocode, we could manage to decompose all the framework into segments and measure the time each of this segment need to be completed. The main observation was that the most time-consuming tasks that contribute extra overhead were the read / write operations from/to R5 controller to its peripheral memories. Therefore, in following figure 3.8 we show one by one the demanded memory operations that should be conducted by R5 controller in order to execute one single issue transfer which means for a block <= 16KB size.



Peripheral operations for 1 block write

1. Read @BTCM scratchpad receiving transfer (Fastpath) : 1.1 usec
2. Write (enqueue) to scheduling queue @OCM
3. Read AXI Mailbox @PL receiving new transfer and step 2 again
4. Read (dequeue) from scheduling queue @OCM : 200nsec
5. Issue/Write 16KB block info to Transaction table @OCM
6. Write (enqueue) block's transaction id to timeout double linked list @OCM : 200 nsec
7. Write/Initiate 16KB transaction @PL : 60 nsec
8. Write Control message if block was the last one @PL : 370nsec
9. Read from RT Mailbox @PL for acknowledgment (ack/nack) : 140nsec
10. Dequeue acked/nacked transaction from timeout list @OCM
    11. IF receives NACK : step 6
    12. step 7 and step 8 (if needed)

Figure 3.8: Breakdown Analysis

As the above figure 3.8 demonstrates, R5 is notified by the process in BTCM scratchpad after 1.1 useconds. Also, each read/write operation to OCM memory costs 200 nanoseconds. Moreover, the time needed to write three 64 bit descriptors from PS to PL with merge buffering is only 60 nanoseconds. However, right after that merge buffering the write descriptors for completion message cost 370 nanoseconds which is translated at least 50 PL cycles overhead. This gives us an obvious conclusion that the major R5 controller bottleneck is the read/write operations from/to PL.

PS-PL time cost communication

R5 real time processor is located in PS side of the FPGA but AXI mailboxes, that are used to communicate with A53 cores and receiver node, are implemented in PL side that is far way. This iterative RRT PS-PL communication costs at least 100-150 nanoseconds each time which makes R5 incapable to execute further computations while waiting to finish this operation.

3.6 Read Operation

This section is a detailed explanation of the read RDMA operation and describes the steps are needed to be followed, so that each remote read request being initiated until being completed. The figure 3.9 below is a top level diagram of a read RDMA transfer focused mainly on the functionality of the framework that was implemented in R5 microcontroller and is installed in each target node. As you can see, the biggest part of read RDMA implementation consists of the write RDMA design. In this chapter, we present the additional functionality that was needed in order to accomplish remote reads between multiple nodes as well as some protocol deadlock and limited resources issues that we managed to overcome.

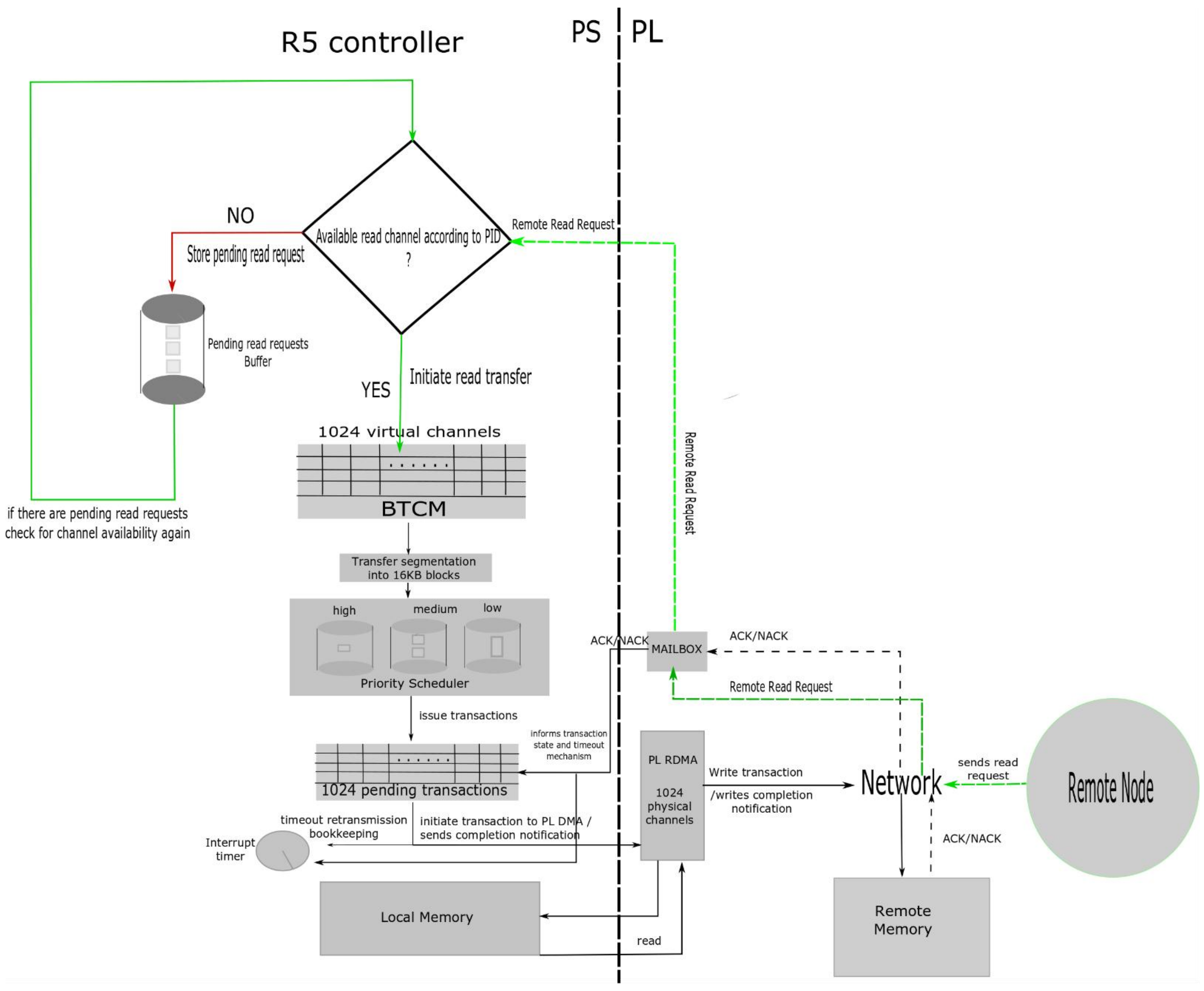

R5 controller
PS
PL
NO
Store pending read request
Available read channel according to PID ?
Remote Read Request
Pending read requests Buffer
YES
Initiate read transfer
1024 virtual channels
BTCM
if there are pending read requests check for channel availability again
Transfer segmentation into 16KB blocks
high
medium
low
Priority Scheduler
issue transactions
ACK/NACK
MAILBOX
ACK/NACK
Remote Read Request
Remote Read Request
informs transaction state and timeout mechanism
1024 pending transactions
PL RDMA
1024 physical channels
Write transaction /writes completion notification
Network
sends read request
Remote Node
ACK/NACK
timeout retransmission bookkeeping
initiate transaction to PL DMA / sends completion notification
Interrupt timer
Local Memory
read
Remote Memory

Figure 3.9 Read RDMA timeline

3.6.1 Read Implementation

In the implementation for read, a remote (Initiator) node which desires to read data from the memory of a local (Target) node, sends a read request through the network and asks explicitly the data from the local node. The local node receives this request and writes the demanded data using R5 framework included fast completion message mechanism as shown in above figure 3.9 back to remote node. In order to achieve sending the request, initiator node uses a custom hardware implemented packetizer which is responsible to write the appropriate descriptors of a read request in a predefined packet format and send this packet through the network to the target node.

Therefore, each time a remote initiator node sends a read request using packetizer through the network for asking the data, target node receives this remote request in a PL implemented mailbox, the same one which is used to receiving transaction acknowledgments too. So, R5 controller polls this PL mailbox and when receives a valid value from mailbox, checks if this value is a transaction acknowledgement or a read request from a remote node. If this is a remote read request, then software begins the decoding process of this request.

To begin with, software extracts this packet reading one by one the 32-bit fields of this packet request which are predefined in a standard format by packetizer and constitute the major descriptors of a read request initiation. In detail, these fields are the process domain id in which read request belongs to, the expected transfer size from the local memory in bytes, the source and destination addresses of the read transfer as well as first and last notification data and destination notification address that compose the completion message. Firstly, as it is mentioned in the following table 3.4 the "=done" field of each read channel at BTCM is initialized with the value two (2). After read request initiation, this transfer request considered as ongoing and its "done" field gets the value zero (0). When the read transfer finishes successfully its "done" field becomes one (1). In the next step, r5 controller tries to occupy a free read channel of 32 available in BTCM page according to request's process domain id in order to host this new initiated read request.

Therefore, framework checks the "done" field of each of these 32 available read channels which corresponds to the read request. The first channel that is free or completed, hence its "done" field is two (2) or one (1), hosts the new read request and software copies its descriptors in that area. Now the "done" field of the channel is zero (0) and the read request is considered as ongoing. Thus, r5 begins to serve this data transfer applying exact the same quality of service and resilience mechanism like write RDMA implementation in order to initiate this read request as write operation to hardware.

Values	Status	Description
2	Initialized	Read transfer just initiated or
1	Completed	Read transfer has totally finished
0	On Going	Read transfer has been initiated and is still working

Table 3.4: "Read done" field state

In the following sections, some issues which have to do with upcoming resources limitations as well as protocol deadlock cases related with this read RDMA implementation, will be discussed. Also, we recommend some alternatives architecture solutions which manage to overcome these problems as future work.

3.6.2 Protocol Deadlock Resolution

On the other hand, in a case software does not find any free available read channel for a long time in a process domain page at BTCM scratchpad, the receiving read requests in PL mailbox queue which correspond to this process domain are blocked and falling into starvation. Specifically, this scenario could happen in our implementation, if the 32 total positive acknowledgments which make the outstanding read channels completed for the corresponding process domain id in BTCM scratchpad, are located exact behind from read requests in the mailbox queue. Thus, read requests are going to wait until any outstanding transfer becomes completed in order to take its place, however the acknowledgements that satisfy the completion of the outstanding transfers reside behind these requests in the same queue. Obviously, this scenario may lead to a protocol deadlock between the read requests and the transfer completions in the mailbox queue, as shown in Figure 3.10 below, due to the existing dependence on each other.

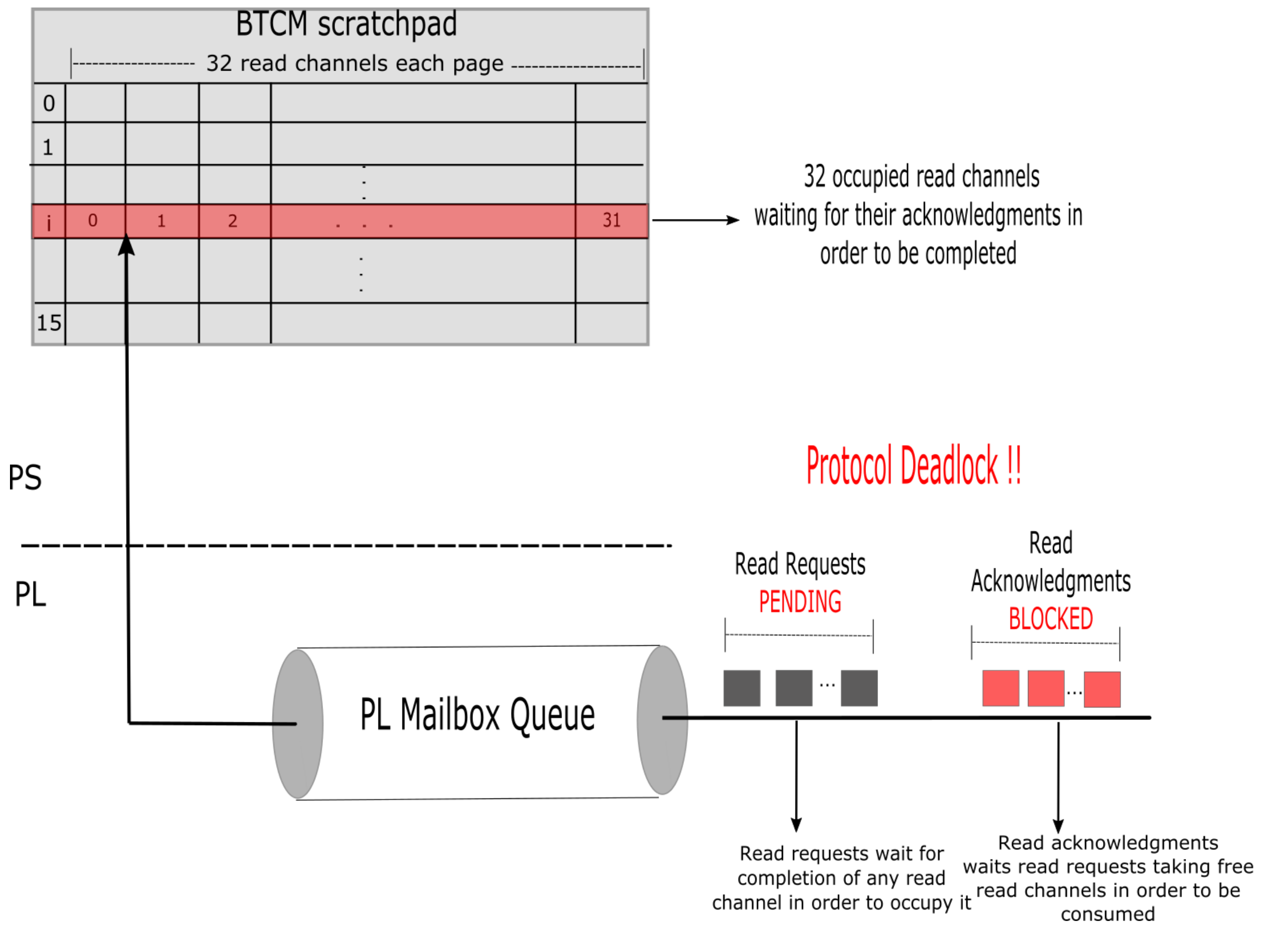

BTCM scratchpad
32 read channels each page
0
1
i
15
0
1
2
31
32 occupied read channels waiting for their acknowledgments in order to be completed
Protocol Deadlock !!
PS
PL
PL Mailbox Queue
Read Requests
PENDING
Read Acknowledgments
BLOCKED
Read requests wait for completion of any read channel in order to occupy it
Read acknowledgments waits read requests taking free read channels in order to be consumed

Figure 3.10 Protocol Deadlock Phenomenon

Therefore, in order to resolve this deadlock phenomenon, R5 controller takes over to store temporary the pending read requests in a special purpose FCFS buffer in OCM memory which are blocked by incomplete read channels as shows figure 3.11. As long as there are no released read channels of a process domain, each incoming read request which belongs to this specific domain, is stored by priority sequence in this buffer of pending requests.

Furthermore, R5 controller cares to serve these pending read requests of this buffer periodically. If this buffer is not empty, software gets a pending request and checks if there is available read channel for it, otherwise puts it again into the rear of the buffer. In this way, starvation of pending read requests is avoided.

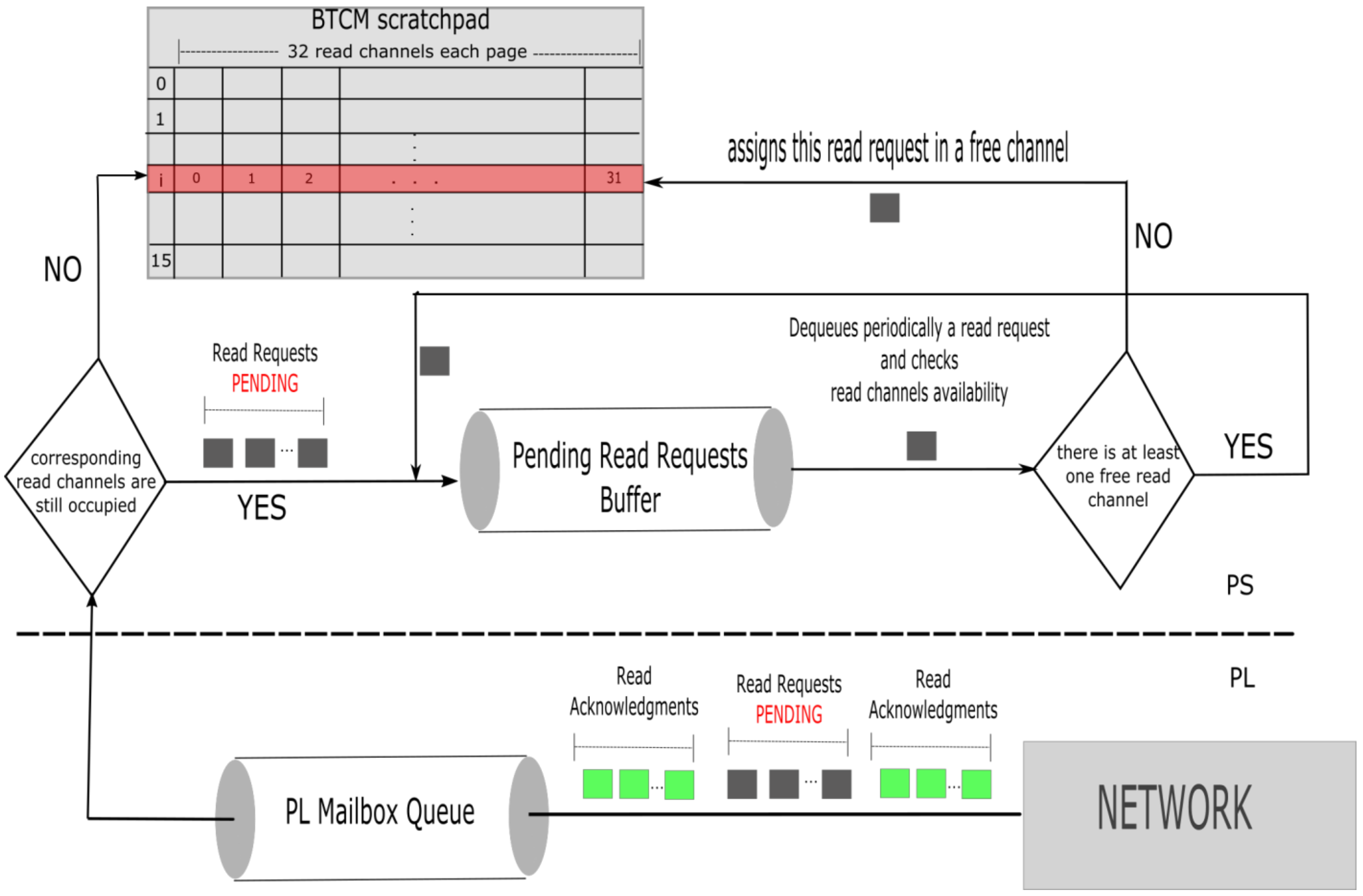

BTCM scratchpad
32 read channels each page
0
1
i
15
0 1 2 . . . 31
assigns this read request in a free channel
NO
NO
Read Requests
PENDING
corresponding read channels are still occupied
YES
Pending Read Requests Buffer
Dequeues periodically a read request and checks read channels availability
there is at least one free read channel
YES
PS
PL
Read Acknowledgments
Read Requests
PENDING
Read Acknowledgments
PL Mailbox Queue
NETWORK

Figure 3.11 Protocol Deadlock Resolution

At the end, there is also a case in which this special purpose buffer becomes full due to its limited capacity before any of the corresponding read channel is completed. In next section, we recommend a holistic method of read requests management in order to avoid the above mentioned buffer overflow as well as a better resolution approach of the protocol deadlock case.

3.6.3 Proposed Optimizations

In this section, we propose an alternative organization of receiving messages from PL mailbox queue. As we observe at previous implementation, different kind of messages such as read requests and transaction acknowledgements which are destined to the same queue with mutual dependence, could drive in a protocol deadlock. Moreover, either read requests or acknowledgments, that arrive at mailbox queue that are destined to different process domain pages at BTCM can be blocked between each other leading to a kind of head of line blocking or starvation.
In order to avoid these failures and low performance cases, we recommend the following implementation as shown figure 3.12 which resolve both deadlock and resources limitations scenarios and manage in a more efficient way receiving messages in PL.

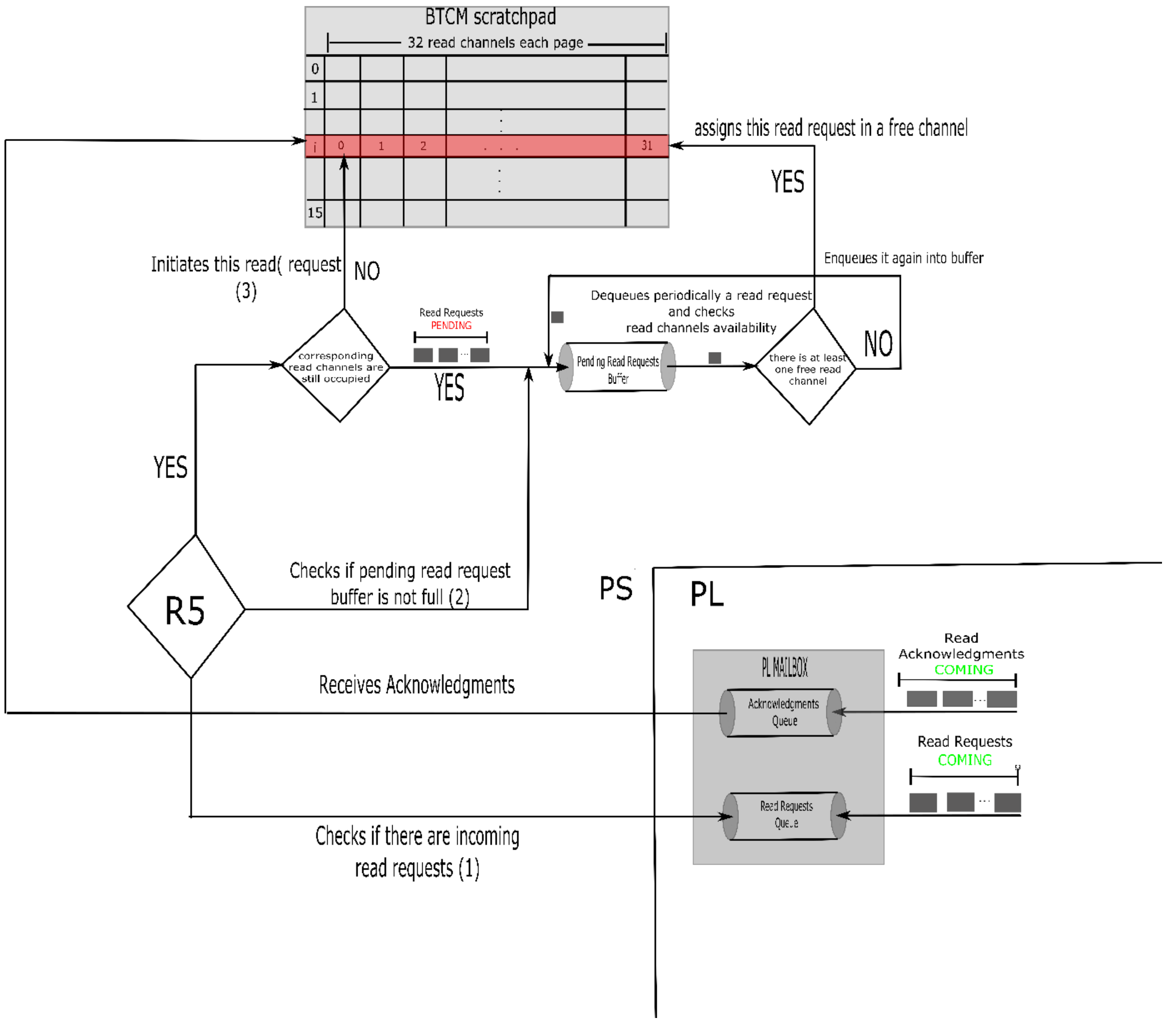

BTCM scratchpad
32 read channels each page
0
1
i
15
0
1
2
31
assigns this read request in a free channel
YES
Enqueues it again into buffer
Initiates this read(request
(3)
NO
Read Requests
PENDING
corresponding read channels are still occupied
YES
Dequeues periodically a read request and checks read channels availability
Pending Read Requests Buffer
there is at least one free read channel
NO
YES
R5
Checks if pending read request buffer is not full (2)
PS
PL
Receives Acknowledgments
PL MAILBOX
Acknowledgments Queue
Read Requests Queue
Read Acknowledgments
COMING
Read Requests
COMING
Checks if there are incoming read requests (1)

Figure 3.12 Proposed Read RDMA

First of all, R5 controller checks if there is any incoming read request from a separate queue at PL that is intended only for this purpose, while receiving, in an absolutely different queue, acknowledgments from completed transactions. Afterwards, R5 checks if there is quite space in pending read request buffer in case of receiving a read request that cannot be initiated and should be stored there at the moment. If both above situations are satisfied, then controller gets a read request and initiates that to a free read channel at BTCM according to its protection domain, otherwise stores that request temporarily in the pending read request buffer. In this way, we isolate messages like read requests and acknowledgments with possible mutual dependence in totally different incoming queues allowing controller treating them independently.

As a result, applying this design we avoid deadlocks cases such as read request that waits for acknowledgments which are exactly behind them in order to be initiated, as well as head of line blocking which is observed between these different kind of messages in a single queue. Last but not least, now we can ensure pending read request buffer overflow avoidance because of the fact that we check his capacity before gets a new incoming read request from PL queue.

Chapter 4

Resilience Mechanism

This chapter describes timeout retransmission timer implementation as part of resilience mechanism which is supported by our RDMA. The unit that expires after timeout event and should be retransmitted again to PL is a 16KB transaction/block which is composed of 64 packets of 256-byte each. As a result, we implement transaction but not packet level timeout retransmissions.

As you know, TCP protocol dynamically calculates timeout based on the round-trip time measured by itself. Currently, in our case we have defined retransmission timeout as a fixed minimum value of 200 milliseconds, but it is reconfigurable. However, with high-speed networks, such as 10 Gigabit Ethernet, the round-trip time (hence retransmission timeout) is expected to be much lower. We lose a lot of throughput for each millisecond we are not transmitting. A better method is needed to deal with high-speed and low-latency networks.

Following sections explain two different approaches of timeout retransmission mechanism that implemented in R5 microcontroller in order to reduce per-tick processing overhead of timer.

4.1 Timeout Double Linked List

Every transaction which is initiated by R5 controller to PL-RDMA transmitter constitutes a pending connection which expects taking acknowledgment through the network by receiver. Soon after software initiates a new transaction to PL-RDMA, stores connection's identity called as transaction id into a data structure that implemented for this special reason into OCM. This data structure is a time sorted double linked list consisting of ids from already initiated transactions. Each time a new transaction is being initiated, software takes over to insert its transaction id in the rear of this list. Thus, this timeout list is composed of pending transactions ids in a chronological order. The head of the list contains the id of the oldest initiated transaction and the rear the most recent respectively as shows the following figure 4.1.

When R5 controller receives a positive acknowledgment from real time mailbox at PL, drops immediately its corresponding transaction id from timeout list wherever it is into the list and reconnects the pointers. In case this acknowledgment corresponds to the head of the list, controller just make a simple dequeue operation. On the other hand, if the receiving response is a negative acknowledgment (nacked), then software drops its corresponding transaction id from the list, reconnects again the pointers in the list, retransmits this one transaction and inserts its id again in the rear of the list. In case of this failed transaction was the last one of the whole transfer, software is obligated to send again the control notification message too.

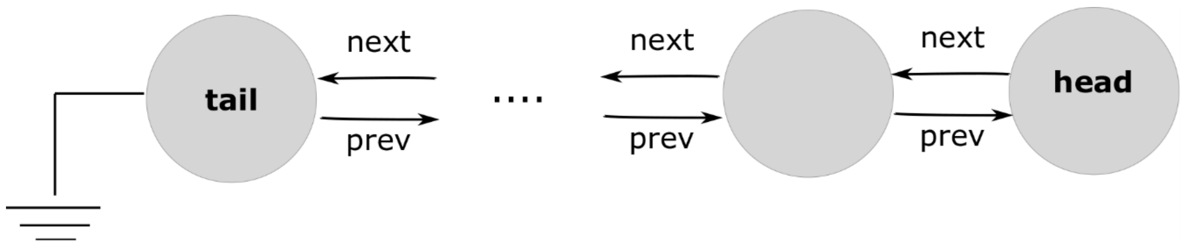


Figure 4.1 Timeout double linked list

Following sections present two different implemented techniques of timeout retransmission routines, influenced by TCP protocol, which use the mentioned timeout list in order to handle expired transactions.

4.2 Absolute Time Retransmissions

In every transaction (16KB) initiation to PL, before software puts its corresponding id into the timeout list, takes a timestamp of the cycles of R5 processor at this moment using R5 performance counters. Afterwards, R5 controller adds these cycles with the expected timeout period translated in cycles taking in this way the future expiration date of this transaction as following code lines describe briefly.

1. **TIMEOUT_PERIOD = 10000000** // period of expired transaction = 20 milisec
2. **proc_cycles = myperf_pmc_cntr_cycles();** // timestamp of processor cycles
3. **transaction_id.tick_counter = proc_cycles + TIMEOUT_PERIOD;**

In general, tick counter variable contains the absolute expiration time in cycles of each transaction. Consequently, R5 controller enqueues transaction id included its tick counter into the timeout list and initiates the transaction to PL RDMA. Every time softare inserts a new transaction id to timeout list increases a counter called "expecting_acks" so that taking knowledge how many acknowledgments expects from receiver. Therefore, when R5 controller checks the value of the "expecting_acks" and this is positive which means that timeout list is not empty, calls a routine in order to handle retransmissions. This routine is called check_timeout_list.

First of all, "check_timeout_list" routine handler reads the oldest transaction id which is the head of the timeout list that is sorted in a chronological order and takes a new timestamp called "time" translated in R5 cycles at this moment. After that, handler compares transaction id's timestamp "tick counter" with the value "time" and if "tick counter" is greater than the value "time", means that the corresponding transaction is expired and should be retransmitted. So if this transaction is expired, R5 controller calls another routine called transaction_retransmission and makes the following operations:

1. Checks if "retransmission bit" which is the 57th bit of the third word of transaction initiation is 1 then initiates again whole transaction to PL again
2. Checks also if this is the last transaction of the transfer and if it is true then sends control notification message to PL too
3. Calculates the new expiration date for this retransmitted transaction with the exact same way as we described in the beginning
4. Puts its transaction id again into the rear of the timeout list

After that, "check_timeout_list" handler checks the next transaction id of the timeout list which is the new head element if it has expired too and executes the same procedure, otherwise breaks and returns. In a worst case scenario where all transactions have been initiated at the same time and have expired, the handler will traverse and drop all the elements of the timeout list.

4.3 Interrupt Timeout Retransmissions

In every transaction (16KB) initiation to PL, before R5 controller puts its corresponding id into the timeout list, specifies a tick counter of this transaction with a configurable value. This tick counter implies the total number of clock ticks that are needed to happen timeout for this mentioned transaction by this moment in the future. Thus, if the timeout period for each transaction is 20 milliseconds and the clock tick happens every 1 millisecond, this tick counter is set with the value of 20. In order to achieve an accurate clock tick interrupt every millisecond, we used one the Triple Timer Counters (TTC) that is clocked by PS. More about use and setup of TTC interrupts reported at appendices. Now, there are two (2) different ways so that interrupt handler routine can be implemented in order to check for expired transactions. Following paragraph is one step by step description of these different algorithms.

The steps of the first implementation are:

1. R5 controller declares a global tick counter (GTC) as number of ticks.
2. also, before software initiates each transaction to PL, assigns the timeout period of transaction as the sum of the GTC and the expiration time calculated in ticks. For example, if interrupt event happens every 1 millisecond and the expiration time is 20 then the timeout period = 20 milliseconds, so **transaction_tick_counter = GTC + expiration time;**
3. R5 initiates transaction to PL and puts its id into the rear of the chronological sorted timeout list.
4. When TTC interrupt handler is called, increases GTC plus 1
5. and checks the head of the timeout list if GTC has reached the tick counter of the mentioned transaction e.g if **(GTC == head. transaction_tick_counter)**
6. If it is true, then software calls the transaction_retransmission routine which makes the appropriate operations that described exactly in previous section.

The steps of the alternative implementation could be the following:

1. Before software initiates each transaction to PL, assigns the timeout period of transaction as the expiration date calculated in ticks. For instance,
   expiration time = 20;
   transaction_tick_counter = expiration time;
2. R5 initiates transaction to PL and puts its id into the rear of the chronological sorted timeout list
3. When TTC interrupt handler is called, decreases the **transaction_tick_counter** of the head of the timeout list
4. If transaction_tick_counter is equal to zero (0), that means this transaction has expired and should be retransmitted
5. Software calls transaction_retransmission routine

4.4 Sequence Numbers and Livelocks

In the context of high reliability and quality of network resilience, we developed a technique putting sequence numbers in each transaction that is initiated to PL RDMA in order to deal with two different existing scenarios.

First scenario has to do with early timeout event. This happens, when a transaction is considered as expired and transmitter is obligated to retransmit it. However, at some time transmitter receives the positive acknowledgment by the previous transmission of the transaction, not by the retransmission. This could be happened due to the miscalculation of the timeout period of this transaction.

At the moment, we define a timeout period with a fixed value of 20 milliseconds, but in the future this could be estimated based on critical parameters such as transmission distance, network topology, routing algorithm and bandwidth etc.

We resolve this phenomenon setting at the first transmission of each transaction its sequence number as zero (0). Each time a timeout of a transaction occurs, we increase this sequence number plus one (1). If R5 controller receives a positive acknowledgment of this mentioned transaction and its sequence number is smaller than the current, then ignores this acknowledgment waiting for the last one.

Second scenario is related to the level of retransmissions which are conducted in this version. Because of the fact that now we retransmit an expired transaction/block consisting of 64 total packets, but receiver can send back negative acknowledgment for this transaction in packet level, there is possibility receiver sends back more than 1 until 64 negative acknowledgments for one (1) transaction accordingly to its corrupted packets. Thus, we apply exact the same methodology setting at the first transmission of each transaction its sequence number as zero (0). Each time, software receives a negative acknowledgment of a specific transaction, increases the sequence number plus one (1) of the transaction and retransmits that.

Unless the sequence number of the receiving negative acknowledgment is smaller than current of this mentioned transaction, that means negative acknowledgment was caused from a corrupted packet that belongs in a transaction of previous transmission, so R5 controller should ignore that.

Livelocks

Due to the fact that in this implementation we retransmit a block level only if one packet fails. Thus, assuming that:

$$\text{Probability of a failed packet transmission} = p,$$

and each packet transmission is an independent experiment with fail probability p, then, the expected number of packet fails in a block of n packets is:

$$\text{Expected fails in n packets} = n \cdot p.$$

The probability of seeing at least one fail is

$$\text{Probability of at least 1 fail in n packets} = p(n) = 1 - (1 - p)^{n}.$$

Assume that the packet size is 256 Bytes, that a transfer wants to convey 1GB of data, i.e 4M packets, and that the packet fail rate is p= 1 / 1000. Under these assumptions, the block fail probability is

p(64) = 1-(1-0.001)^64 = 0.062025.

On the other hand, the fail probability of an unsegmented 1Mbyte message is

$$p(4096) = 1-(1-0.001)^{4096} = 0.98339496.$$

However, in the segmentation case, for a message to complete, all the blocks that comprise it need to complete successfully. For instance, the 1MB transfer consists of 64 blocks. Thus, if we assume for simplicity that these are 64 independent experiments, each with success probability p(4096), then, we will see at least on block fail with probability:

$$q = 1 - (1-p(64))^{64} = 0.98339492$$

Although the expected fail rates are equal for the segmented and the unsegmented case if we measure it on the message level, in practice, in the segmented case we expect that by retransmitting only the failing blocks, we:

1. We reduce the overhead of retransmissions
2. We reduce the livelock probability and durations, as in every fail we have to retransmit a smaller number of bits, thus reducing the probability that a fail will happen again. We leave a more comprehensive analysis of this fact for future work.

Following figure 4.2 demonstrates livelock protocol scenario that could be happened based on above conditions.

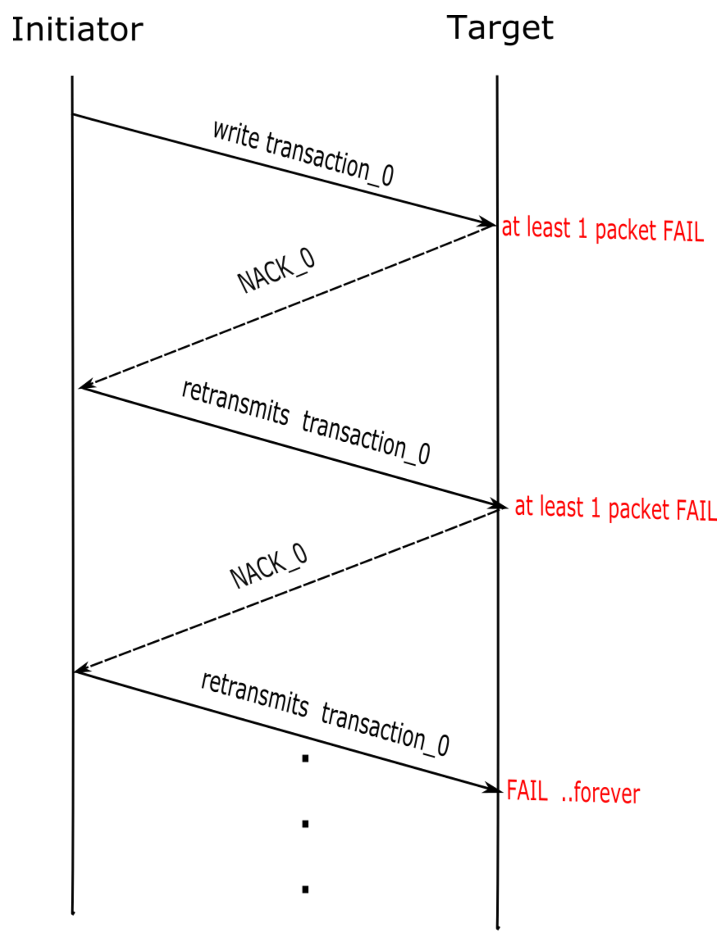


Figure 4.2 Livelock scenario

Chapter 5

Performance Evaluation

In this Chapter, we present the average flow completion time and the average throughput of our new advanced user level initiated read/write RDMA for various transfer sizes compared with user level initiated 8-channel Xilinx ZDMA and Kernel initiated RDMA, as well as some quality metrics which provide to us useful conclusions.

5.1 FORTH vs Xilinx ZDMA

This section covers the performance analysis as well as experiments that were performed between

- FORTH's PL RDMA implementation (FRDMA)
- and Xilinx PS RDMA,

in terms of message completion time and average throughput.

The following figure 5.1 demonstrates the average completion time in microseconds based on various transfer sizes in bytes for a node which conducts a write RDMA operation to itself. As we can observe for very small message sizes until 16 KB approximately ZDMA presents faster completion than FRDMA, however for bigger sizes, of one (1) 16 KB or more, FRDMA yields much lower latency than the ZDMA – note the y-axis is in logscale.

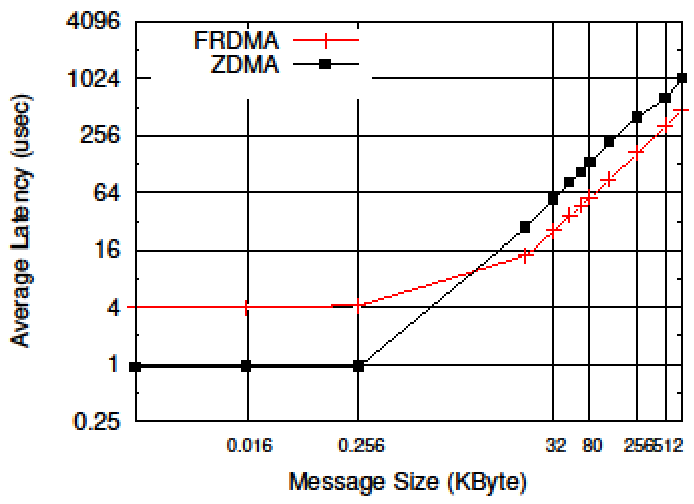


Figure 5.1 Average Completion Time Loopback write FRDMA vs ZDMA

Correspondingly, the figure 5.2 below compares the average throughput of FRDMA and of ZDMA. In this experiment, we perform loopback -- a node conducts a write RDMA operation to itself. It is obvious again that for message sizes bigger than 16KB, the FRDMA offers twice as much throughput when compared with ZDMA, reaching out the link capacity (10 Gb/s).

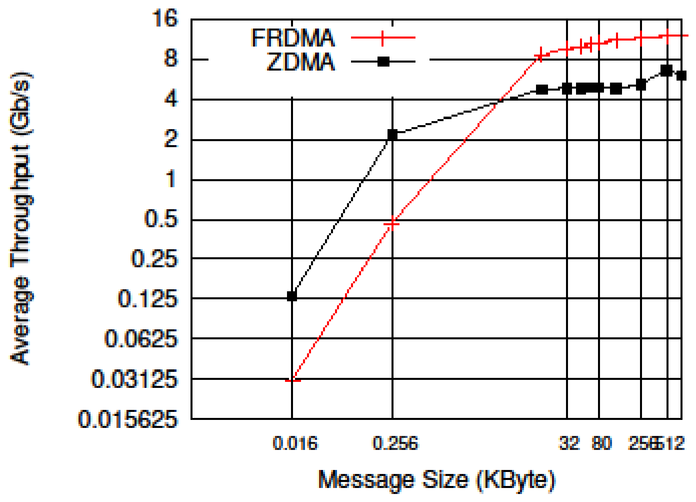


Figure 5.2 Average Throughput Loopback write FRDMA vs ZDMA

We conducted the same experiment but this time between 2 remote FPGA nodes. In this experiment we observed that 1 hop distance write RDMA operation performs much better performance using FORTH RDMA both in average latency measurements and throughput in comparison with Xilinx ZDMA.
The following figure 5.3 clearly shows that for transfer sizes bigger than 1 KB FORTH's RDMA presents one order of magnitude lower completion time than the ZDMA.

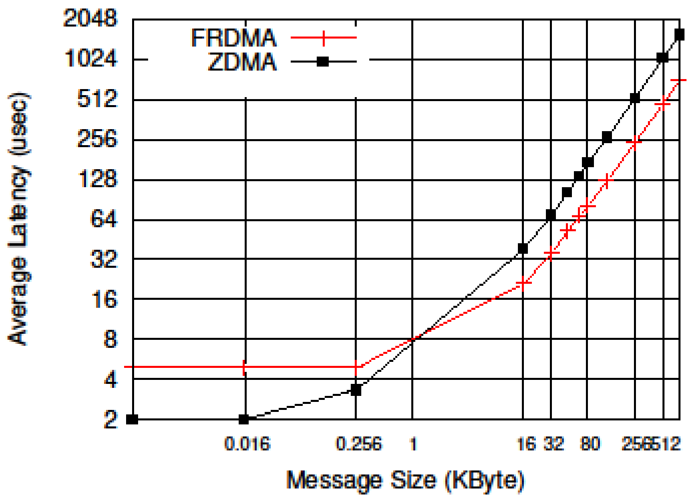


Figure 5.3 Average Completion Time 1-hop write FRDMA vs ZDMA

Exact the same behavior is reflected to 1-hop distance write RDMA operation measuring the average throughput between Forth RDMA and ZDMA as shown in figure 5.4.

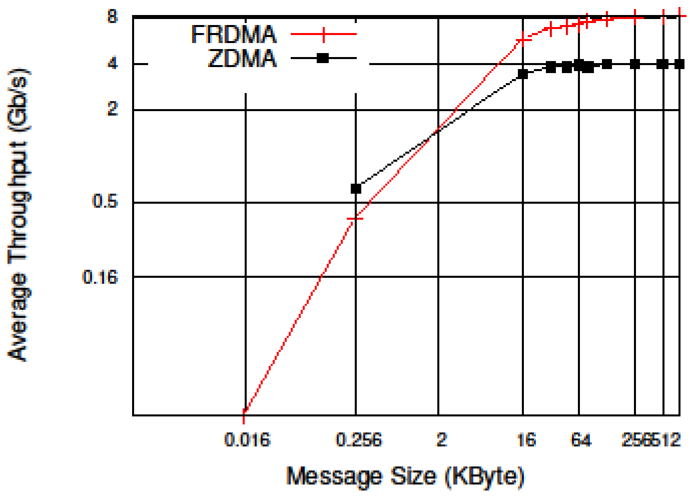


Figure 5.4 Average Throughput 1-hop write FRDMA vs ZDMA

Next, two (2) figures, 5.5 and 5.6, make a comparison between read and write 1 hop distance FRDMA operation. For latency sensitive transfer sizes such as 16B, 256B etc write operation FRDMA yields lower completion time than the corresponding read operation. However, over 16KB message size both operations present exact the same behavior.

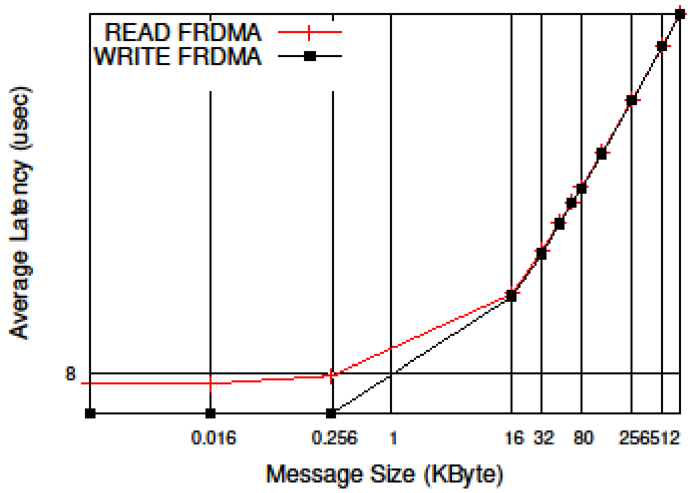


Figure 5.5 Average Completion Time 1-hop read vs write FRDMA

This behavior is reflected also in the throughput graph. The following figure 5.6 proves that write operation exhibits better performance for up to 16KB message sizes when compared to read FRDMA operation.

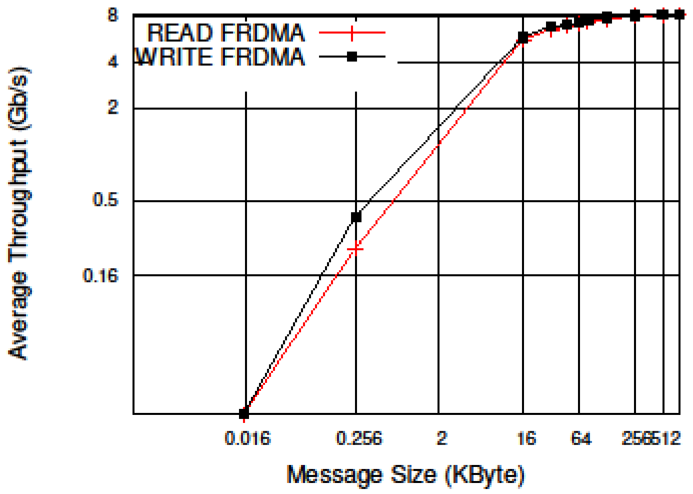


Figure 5.6 Average Throughput 1-hop read vs write FRDMA

The overhead of read operations is justified due to the fact that when the RT software receives a packet request, it needs to extracts the request from the real time mailbox, reading its payload in 32-bit words. This bottleneck is caused due to R5 handicap which cannot read double precision words as mentioned in section 3.5.

5.2 FORTH vs Kernel initiated RDMA

This section shows the performance benefits of user level initiated FORTH RDMA compared to to Kernel-initiated RDMA operations. The experiment that we conducted was 1-hop distance write RDMA operation between 2 remote nodes/FPGAs. As figure 5.7 indicates, the Kernel initiated write RDMA operation starts with a latency of about 11 microseconds, for small transfer sizes; on the other hand, FORTH's RDMA consistently presents a lower completion time behavior for all transfer sizes.

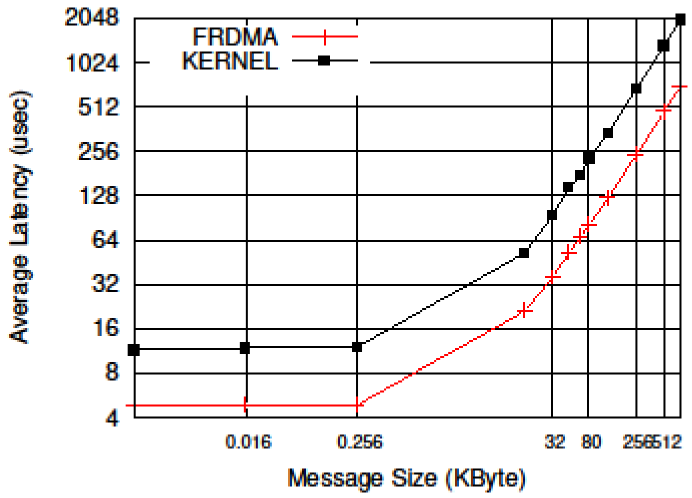


Figure 5.7 Average Completion Time 1-hop write FRDMA vs KERNEL

Next, in figure 5.8 we show the average throughput of user level initiated FRDMA when compared with Kernel initiated RDMA. In this experiment, we perform 1-hop distance remote write operation. As you can see the average throughput for Kernel initiated RDMA is limited to maximum 3 Gbps contrary to FORTH RDMA average throughput which saturates over 8 Gbps.

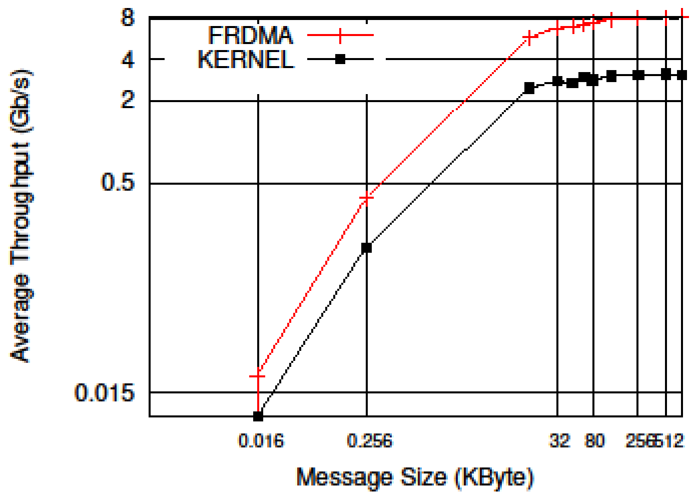


Figure 5.8 Average Throughput 1-hop write FRDMA vs KERNEL

5.3 Quality Measurements

This section introduces some qualitative measurements that were conducted in order to evaluate the behavior of Forth RDMA beyond the regular run cases.
Following figures 5.9 and 5.10 demonstrate the average message completion time and average throughput for the optimized and non-optimized write Forth RDMA operation. Non-optimized Forth RDMA is referred to RDMA with disabled all the optimizations that targeted latency reduction as described in chapter 3 and section 3.3. Both figures prove that optimized Forth RDMA has better performance for transfer sizes smaller than 16KB.

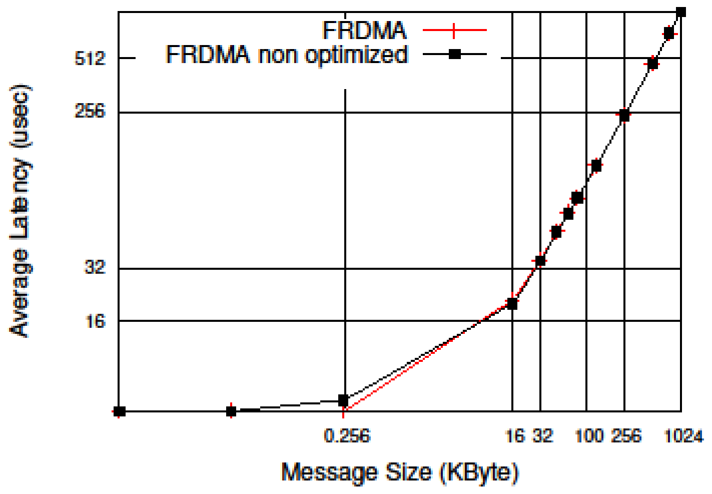


Figure 5.9 Average Completion Time optimized vs non-optimized FRDMA

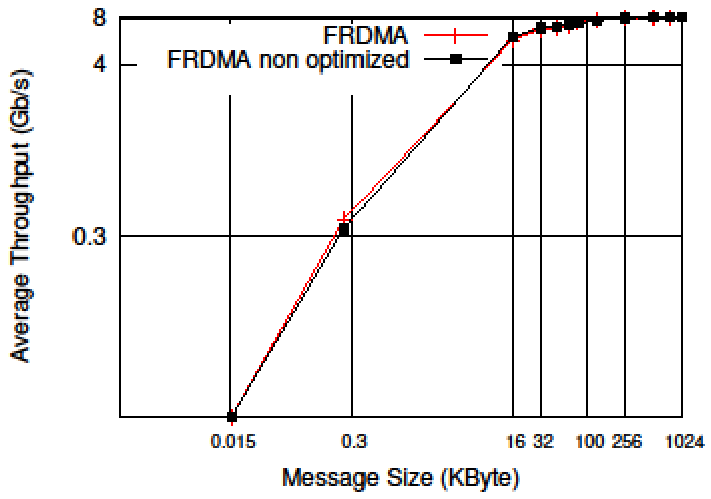


Figure 5.10 Average Throughput optimized vs non-optimized FRDMA

In another experiment that we conducted, we evaluated of performance behavior in the case of one (1) timeout retransmission for various transfer sizes. As we can observe in figure 5.11, the average completion time for write FRDMA operation presents at least 20 milliseconds overhead when one packet fails, and thus one block has to be retransmitted. This additional delay happens due to the fixed timeout period (about 20 milliseconds) which takes place until the lost block the source retransmits the failing block. Figure 5.11 proves obviously this phenomenon.

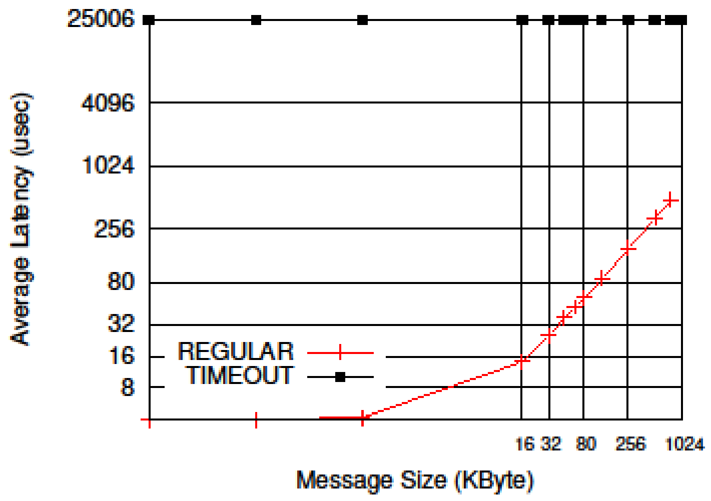


Figure 5.11 Average Completion Time Regular vs Timeout FRDMA

Figure 5.12 shows how much bandwidth could be lost in case of one (1) timeout retransmission event over the network for various message sizes. As we can see the damage of performance is so huge, even for 1KB transfer size the average throughput is much lower than 1 Gbps. This happens because of the timeout period during which the link is idle.

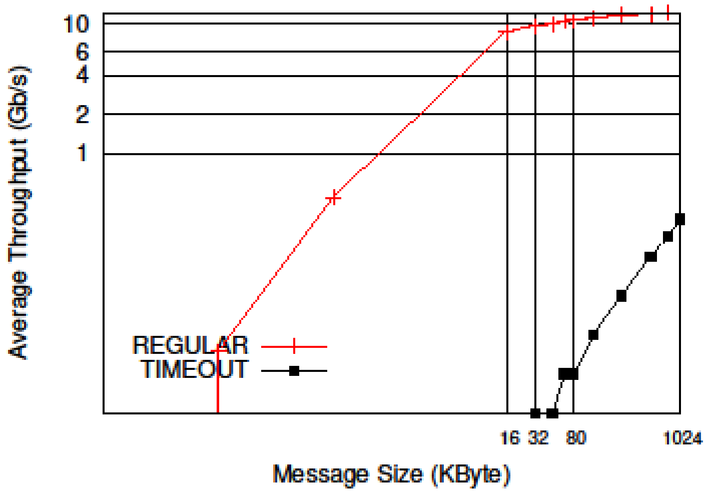


Figure 5.12 Average Throughput Regular vs Timeout FRDMA

Chapter 6

Conclusions – Future Work

Modern computing clusters consist of many heterogeneous computing units that work collectively in order to fulfill high computing tasks. Low latency communication between the remote processes that run on these servers is a critical factor for achieving high performance. In this work, we described the software programmable part of an advanced RDMA engine which is used in the ExaNeSt prototype. This firmware of the RDMA engine provides advanced Resiliency and Quality-of-Service (QoS) features and allows user-level transfer initiation. Performance evaluation demonstrates flow completion time reduce in various transfer sizes as well as a significant increase of throughput that approaches the upper bound of links.

The bottleneck that was observed, is the weakness of real time processor to make 64 bit read/writes from/to PL memory due to its limited 32-bit bus width. On the other hand, this issue could be solved either using an A53 core which is obviously capable of 64bits read/writes from/to PL instead of R5 microcontroller or making hardware implementation of the whole R5 framework into the PL.

One another solution for latency reduction could be the process write directly the dma descriptors for a small transfer (< =16 KB block) to PL registers in order to initiate instantly the RDMA operation and after that notifies the R5 for bookkeeping and receiving the corresponding acknowledgment.
Alternatively, we could replace R5 processor with a Microblaze Xilinx core however again some critical questions come up about feasibility due to Microblaze:

1. Supports 32-bit AXI master/slave interface
2. May not reach 500-600MHz clock
3. Need specified BRAMs > 128 KB

This idea could be feasible if we used a 64bit RISCVI included in PL otherwise supported with coherent IO accesses.
Also, it deserves further investigation if this framework can also corporate well applying a congestion control on each node implemented by a hardware rate limiting block based on the feedback of the network path.

Bibliography

[1 Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang. Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, Wei Zhao, Xunqiang Yin, Chaofeng Hou, Chenglong Zhang, Wei GeJian, Zhang Yangang, Wang Chunbo , Zhou Guangwen, Yang Email author (2016). The Sunway TaihuLight supercomputer: system and applications. *Science China Information Sciences*, *59*(7), 072001.

[2] R. Ammendola, e.a, "The next Generation of Exascale-class Systems: the ExaNeSt Project". Appears in: the Proceedings of the Euromicro Conference on Digital System Design (DSD 2017), Vienna, Austria, 30 August - 1 September, 2017.

[3] E. P. Markatos and M. G. H. Katevenis, "User-level DMA without operating system kernel modification," Proceedings Third International Symposium on High-Performance Computer Architecture, San Antonio, TX, 1997, pp. 322-331.

[4] Yuanwei Lu, Microsoft Research and University of Science and Technology of China; Guo Chen, Hunan University; Bojie Li, Microsoft Research and University of Science and Technology of China; Kun Tan, Huawei Technologies; Yongqiang Xiong, Peng Cheng, and Jiansong Zhang, Microsoft Research; Enhong Chen, University of Science and Technology of China; Thomas Moscibroda, Microsoft Azure, “Multi-Path Transport for RDMA in Datacenters”, Published 2018 in NSDI

[5] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, Scott Shenker , “Revisiting Network Support for RDMA”, SIGCOMM 2018 paper

[6] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitendra Padhye, Marina Lipshteyn Microsoft, “RDMA over Commodity Ethernet at Scale”, SIGCOMM '16, Florianopolis , Brazil 2016

[7] Diego Crupnicoff, Sujal Das, Eitan Zahavi, “Deploying Quality of Service and Congestion Control in InfiniBand-based Data Center Networks“ White paper

[8] Zynq Ultrascale+ MPSoC Technical Reference Manual

[9] Cortex-R5 Technical Reference Manual

Appendix A

R5 Memory Settings

We should specify and limit precisely the memory regions of R5 microcontroller in order to control application's memory map settings. This configuration can be succeed following the next steps:

1) Right click on the R5 project directory that you created in XSDK platform.

2) Choose "Generate Linker Script" selection

3) Click "Basic" button right on the top and define all place sections in first 64 KB part of R5 scratchpad memory (ATCM).

- Code Section: psu_r5_0_atcm_MEM_0
- Data Section: psu_r5_0_atcm_MEM_0

- Heap and Stack in: psu_r5_0_atcm_MEM_0

Also, we should specify the size of heap and stack memory about 1KB, no more than 2-3 KB because an overflow occurs and no less because this action leads to memory violation without any notice in compilation phase and the program throws segmentation fault on run time.

In order to choose compilation optimizations or any other settings, you should click right on the R5 project again and select "C/C++ Build Settings". For instance, if you prefer another optimization level click on "Optimization" selection of ARM R5 gcc compiler or if you want to add extra compilation flags click on "Miscellaneous" selection.

R5 Memory Attributes

As we mentioned in section 4.7.1 Memory Protection Unit (MPU) works with the R5 L1 memory system to control accesses to and from L1 and external memory. If you intend to change its memory regions in order to apply any memory protection from miss coherent access, to enable merge store buffer or any other modification then pay attention on following guideline.

1) You should make double click to open bsp directory that generated by R5 project

2) then cd psu_cortexr5_0/libsrc/standalone_v6_3/src/ and find mpu.c source file

3) In mpu.c file every memory region for R5 processor is defined and you are able to do necessary changes there, as Chapter 4 specifies.

- In our case, in order to enable merge store buffer, it was needed to make the attribute of PL memory region "normal shared non chacheable" with read/write user full rights as follows:

  Addr = 0x80000000; //EDITED// old = "STRONG_ORDERD_SHARED"

  RegSize = REGION_1G;

  Attrib = **NORM_SHARED_NCACHE** | PRIV_RW_USER_RW;

  Xil_SetAttribute(Addr,RegSize,RegNum, Attrib);

  RegNum++;

- If you desire to make OCM memory cacheable because only R5 processor conducts read/writes at the moment, you can define the following section:

/* 256K of OCM RAM from 0xFFFC0000 to 0xFFFFFFFF marked as normal memory */

  Addr = 0xFFFC0000U;

  RegSize = REGION_256K;

  Attrib = **NORM_NSHARED_WB_WA** | PRIV_RW_USER_RW;

  RegNum++;

And

- If you want to extend or split regions for example protecting **TCM** memory from miss coherency by A53 you can define the following regions:

//WE ADDED ANOTHER REGION, THE **ATCM**, TO BE OVERLAPPED AS NORMAL, NON SHARED WB-WA

Addr = 0xFFFE0000U;

RegSize = REGION_64K;

Attrib = **NORM_SHARED_NCACHE** | PRIV_RW_USER_RW;

Xil_SetAttribute(Addr,RegSize,RegNum, Attrib);

RegNum++;

//WE ADDED ANOTHER REGION, THE **BTCM**, TO BE OVERLAPPED AS NORMAL, NON -CACHEABLE

Addr = 0xFFFE2000U;

RegSize = REGION_64K;

Attrib = **NORM_SHARED_NCACHE** | PRIV_RW_USER_RW;

Xil_SetAttribute(Addr,RegSize,RegNum, Attrib);

A total of 10 MPU regions are allocated with another 6 being free for users.

Appendix B

Linux Booting Process in Prototypes

Trenz Prototype

In order to run Linux in Trenz prototype in cooperation with R5 firmware, you should build an appropriate BOOT.bin that has the R5 executable file (.elf) included.Thus, you must follow instructions below:

1) cd yat (YAT-yet another tool)

2) cd .../trenz/profiles/trenz-sd/bootbin directory and open the extras.sh script file.

3) look at **gen_bif()** routine in which you can format BIN the ROM image, as well as it gives you the opportunity to define precisely the destination of the R5 executable (.elf) file

4) More specifically, you should add highlighted entries such as:

```
function gen_bif () {

        printf "\t//arch = zynqmp; split = false; format = BIN \n \

        the_ROM_image: \n\ {  \n\

        [destination_cpu=a53-0, bootloader] fsbl.elf \n\

        [destination_cpu=pmu, destination_device=ps] pmufw.elf \n\

        [destination_cpu=a53-0, exception_level=el-3, trustzone] bl31.elf \n\

        [destination_cpu=r5-0] /home/leandros/r5_controller_v5.elf \n\

        [destination_device=pl] ${2} \n\

         [load=0x8000000,startup=0x8000000,destination_cpu=a53-0,
        exception_level=el-2] u-boot.elf \n\

        [load=0x800000, destination_cpu=a53-0] Kernel.bin \n\

        [load=0x000000, destination_cpu=a53-0] system.dtb \n\ }" > ${1} }
```

Afterwards, you can build a new BOOT.bin using the appropriate yat commands as were described adequately in section 2.3.2 and you can mount it on a SD card in order to boot Linux in Trenz preloaded with R5 firmware.

QFDBs Prototype

In order to run Linux in QFDBs prototype in conjunction with R5 firmware, you should build an appropriate "boot package" according to the desirable design description file (.hdf) using yat (YAT-yet another tool) toolchain commands as we described in section 2.3.2.

In the next step, you should connect to QFDB environment and if you can achieve to boot the design (.hdf) you need to follow the next bullets:

1. cd at directory where you will find a script that contains commands for loading packages such as "load_boot_package.sh"
2. edit "load_boot_package.sh" with the new command **/root/qfdb-stuff/boot_package /home/pxirouch/bps/2.0.<last_stable_version>/output.bp.<fpga_number>** and run this script.

So now, you have loaded the RDMA design in the FPGA in which you prefer to run processes. In order to load R5 firmware, you should follow the next steps:

1. Set two (2) FPGA reset registers with the corresponding values as:

   /root/qfdb-stuff/rwphys/write32 0x80070004 0x1

   /root/qfdb-stuff/rwphys/write32 0x80070000 0xffffffff

2. Install two (2) device drivers
   - Scratchpad_alloc: TCM memory allocation

     insmod ~/scratchpad_alloc.ko

   - Exanest_virt

     insmod ~ /exanest_virt.ko board_id=$"fpga_destination_id"
3. Copry R5 executable to /lib/firmware as follows:
   cp ~ /r5_controller_linux.elf /lib/firmware/

4. cd to directory in which exists r5 executable (.elf)
5. start R5 firmware running the commands:

   a. echo r5_controller_linux.elf > /sys/class/remoteproc/remoteproc0/firmware
   b. echo start > /sys/class/remoteproc/remoteproc0/state

6. If you want to stop and restart R5 firmware without conducting any board reset, then you should first of all run the command:

   echo stop > /sys/class/remoteproc/remoteproc0/state
   and after that you should **reload the boot package** and r5 firmware (above procedure) again.

Briefly, you can run “remoteproc.sh” and “removeproc.sh” scripts in order to load and stop respectively R5 firmware.