Handling of Memory Page Faults during Virtual-Address RDMA

Antonis Psistakis

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 GH Katevenis*

Thesis Defense: Heraklion, July 2019
Thesis Publication: Heraklion, October 2019

This work has been performed at and supported by the Computer Architecture and VLSI Systems (CARV) Laboratory, Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH), within the ExaNeSt project, funded by the European Commission under the Horizon 2020 Framework Programme (Grant Agreement 671553).

University of Crete
Computer Science Department

Handling of Memory Page Faults during Virtual-Address RDMA

Thesis submitted by
Antonis Psistakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Antonis Psistakis

Committee approvals: _____
Manolis GH Katevenis
Professor, Thesis Supervisor

_____
Angelos Bilas
Professor, Committee Member

_____
Evangelos Markatos
Professor, Committee Member

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

Heraklion, July 2019

Handling of Memory Page Faults during Virtual-Address RDMA

Abstract

Nowadays, avoiding system calls during cluster communication (e.g. in Data Centers, High Performance Computing etc) in modern high-speed interconnection networks comes as a necessity, due to the high overhead of the multiple copies (kernel-to-user and user-to-kernel). User-level zero-copy Remote Direct Memory Access (RDMA) technologies overcome this problem and, as a result, increase the performance and reduce the energy consumption of the system. Common RDMA Engines like these cannot tolerate page faults caused by them and choose different ways to circumvent them.

The state-of-the-art RDMA techniques usually include pinning address spaces or multiple pages per application. This approach has some disadvantages in the long run, as a consequence of the complexity induced in the programming model (pinning/unpinning buffers), the limit of bytes that an application is allowed to pin and the overall memory utilization. Furthermore, pinning does not guarantee that someone will not experience any page-faults, due to internal optimization mechanisms, such as Transparent Huge Pages (THP), which is enabled by default in modern Linux operating systems.

This thesis implements a page fault handling mechanism in association with the DMA Engine of the ExaNeSt project. First, the fault is detected by the fault handler of the ARM System Memory Management Unit (SMMU). Then, our hardware-software solution resolves the fault. Finally, a retransmission is requested by the mechanism, if needed. In our system, this mechanism required modifications to the Linux driver of the SMMU, a new library in software, alterations to the hardware of the DMA engine and adjustments to the scheduler of the DMA transfers. Our tests were run on the Quad-FPGA Daughter Board (QFDB) of ExaNeSt, which contains Xilinx Zynq UltraScale+ MPSoCs.

We evaluate our mechanism and we compare against alternatives such as pinning or "pre-faulting" pages, and we discuss the merits of our approach.

Χειρισμός των Σφαλμάτων Σελίδας Μνήμης στη Διάρκεια Άμεσων Απομακρυσμένων Προσβάσεων Μνήμης με Εικονικές Διευθύνσεις

Περίληψη

Στις ημέρες μας, η αποφυγή των κλήσεων συστήματος κατά την διάρκεια επικοινωνίας συστάδων υπολογιστών (π.χ. Κέντρα Δεδομένων, Υπολογισμοί Υψηλής Επίδοσης κ.ά.) στα μοντέρνα δίκτυα διασύνδεσης υψηλής-ταχήτητας προκύπτει ως ανάγκη, λόγω του υψηλού κόστους των πολλαπλών αντιγράφων (πυρήνα-σε-χρήστη και χρήστη-σε-πυρήνα). Οι τεχνολογίες Άμεσων Απομακρυσμένων Προσβάσεων Μνήμης (Remote Direct Memory Accesses – RDMA), οι οποίες είναι επιπέδου-χρήστη και μηδενικών-αντιγράφων, ξεπερνούν αυτό το πρόβλημα και ως αποτέλεσμα αυξάνουν την επίδοση και μειώνουν την κατανάλωση ενέργειας του συστήματος. Κοινές μηχανές RDMA όπως αυτές, δεν ανέχονται τα σφάλματα σελίδας μνήμης που προκύπτουν από εκείνες.

Οι τελευταίας τεχνολογίας τεχνικές συνήθως περιλαμβάνουν «καρφίτσωμα» (pinning) των χώρων διευθύνσεων ή πολλαπλών σελίδων μνήμης για κάθε εφαρμογή. Αυτή η προσέγγιση έχει κάποια μειονεκτήματα μακροπρόθεσμα, ως συνέπεια της πολυπλοκότητας, η οποία προκαλείται στο προγραμματιστικό μοντέλο («καρφίτσωμα»/«ξεκαρφίτσωμα» ενταμιευτών), του ορίου από bytes τα οποία μία εφαρμογή επιτρέπεται να κάνει «pin» και της συνολικής χρήσης της μνήμης. Επιπλέον, το «καρφίτσωμα» σελίδων μνήμης δεν εξασφαλίζει ότι κάποιος δεν θα αντιμετωπίσει κανένα απολύτως σφάλμα σελίδας, εξαιτίας των εσωτερικών μηχανισμών βελτιστοποίησης, όπως ο Transparent Huge Pages (THP), ο οποίος είναι ενεργοποιημένος ως προεπιλογή στα μοντέρνα λειτουργικά συστήματα Linux.

Αυτή η εργασία υλοποιεί έναν μηχανισμό διαχείρισης των σφαλμάτων σελίδας μνήμης σε συνεργασία με την μηχανή DMA του έργου ExaNeSt. Πρώτα, ανιχνεύεται το λάθος από τον διαχειριστή σφαλμάτων του προγράμματος οδήγησης (driver) της Μονάδας Διαχείρισης Μνήμης Εισόδων/Εξόδων (IOMMU) της ARM, η οποία ονομάζεται SMMU. Έπειτα, η λύση υλικού-λογισμικού μας επιλύει το σφάλμα. Τέλος, στέλνεται ένα αίτημα ώστε να γίνει επαν-αποστολή, όταν χρειάζεται. Στο σύστημα μας, ο μηχανισμός αυτός χρειάστηκε τροποποιήσεις στο πρόγραμμα οδήγησης (driver) Linux για την SMMU, την υλοποίηση μίας νέας βιβλιοθήκης λογισμικού, αλλαγές στο υλικό της μηχανής Άμεσων Απομακρυσμένων Προσβάσεων Μνήμης και τροποποιήσεις στον χρονοπρογραμματιστή των μεταφορών Απομακρυσμένων Προσβάσεων Μνήμης. Οι δοκιμές μας έγιναν επάνω στο Quad-FPGA Daughter Board (QFDB) του ExaNeSt, το οποίο εμπεριέχει τα Xilinx Zynq UltraScale+ MPSoCs.

Εκτιμούμε το κόστος του μηχανισμού μας και κάνουμε σύγκριση με τις εναλλακτικές επιλογές όπως το «καρφίτσωμα» ή την πρώιμη πρόκληση σφαλμάτων σελίδας μνήμης, και συζητάμε τα οφέλη της δικής μας προσέγγισης.

Acknowledgements

The work for this thesis was performed at the CARV Laboratory of ICS-FORTH, from June 2017 to June 2019 and was supported by the ExaNeSt project, which was funded by the European Commission under the Horizon 2020 Framework Programme (Grant Agreement 671553).

There are many people I would like to thank and give credit to because of their generous help in many ways throughout my M.Sc. thesis.

First of all, I would like to thank my supervisor Prof. Manolis GH Katevenis, who supported, guided and trusted me during my thesis. We knew from the beginning that the topic of my M.Sc. thesis is challenging, yet we did not hesitate to choose and work on it.

Secondly, I would like to take this opportunity to thank Prof. Angelos Bilas and Prof. Evangelos Markatos, for being members of my M.Sc. Committee and their feedback.

I would like to express my gratitude to Dr. Fabien Chaix. He was always supportive from the beginning of my M.Sc. thesis. I am grateful for his guidance and interest in my work and progress. Dr. Chaix provided to me constructive feedback during the design, implementation and verification of the FIFO that was implemented, as part of this thesis.

Also, I would like to thank Dr. Nikolaos Chrysos. We had brief discussions in the beginning of my M.Sc. thesis that became more regular and extended in the latter part of my work, when we spent more time both on the translation-fault path of the FORTH PLDMA as well as the mechanism implemented as part of this thesis. I am thankful for Dr. Chrysos' patience, guidance and support.

I would like to give credits to Marios Asiminakis and Vasilis Flouris. Their advice, support and mindset helped me overcome many obstacles in the implementation, the evaluation and even the way of presenting the results.

Also, I would like to thank Dr. Vassilis Papaefstathiou for his feedback in the design of the mechanism, especially during the beginning of the thesis.

I would like to thank Michalis Giannioudis, Pantelis Xirouchakis, Leandros Tzanakis, and Dr. Nikolaos Chrysos, who have worked on and implemented the FORTH PLDMA. I collaborated with Giannioudis, Xirouchakis, and Dr. Chrysos on the debugging of the translation-fault path of the PLDMA. I mostly worked with Giannioudis, since he is the designer of the receive path of the FORTH PLDMA, where the FIFO part of this thesis resides.

I would also like to thank Dr. Manolis Marazakis, Nikolaos Dimou, Nikolaos Kossifidis and Panagiotis Peristerakis, for their support, whenever it was needed.

I greatly appreciate the feedback that Dr. Fabien Chaix, Sotiris Totomis, Vasilis Flouris, Nikolaos Dimou, Pantelis Xirouchakis gave me for the document of my thesis before I submitted it.

Last but not least and while I am hoping I did not forget to thank personally anyone that I should have, I would like to thank everyone in CARV, for all the support throughout my thesis. I could not be more grateful.

To my father George, mother Rania and sister Marianna

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter we will describe the motivation behind this thesis, the contributions and the background information, that was needed for the purposes of this thesis, hardware- and software-wise.

1.1 Motivation

As the years go by, when it comes to Computer Science and the industry, it seems there is a direction in creating more complex sets of computers that will be able to solve difficult problems. An example would be many computing nodes, or "Coherent Islands", as they are described in *Unimem* [1], trying to communicate with each other (e.g. transfer data). Figure 1.1 provides a very simplistic view of that system.

Coherence among all cores (CPUs) in one node is somehow granted. The problem occurs when we have many nodes embedding many coherent cores and we want to achieve coherence among them – the more nodes the bigger the problem.

The ExaNeSt project [1] tries to address this issue by allowing remote memory accesses through a Global Virtual Address Space (GVAS).

In order for such system to be coherent, each node that needs to access information, that belongs to the memory of another node, should request the information from the other node, without keeping a copy to its memory. In order to achieve this goal, Unimem proposed a virtualized system that distinguishes all nodes by utilizing a virtual global address space. This has as a result the necessity of an "arbiter" in each node, that will handle all incoming transactions in order to access safely their memory. This arbiter is called I/O Memory Management Unit (IOMMU) and is necessary in each node.

We prototype this approach by using one Xilinx Zynq UltraScale+ [2] per node. This chip consists of many components including four (4) ARM 64-bit A53 cores and one IOMMU (or in other words, ARM SMMU - System Memory Management Unit).

Kernel-level transactions, as the name reveals, are initiated by the kernel (e.g.

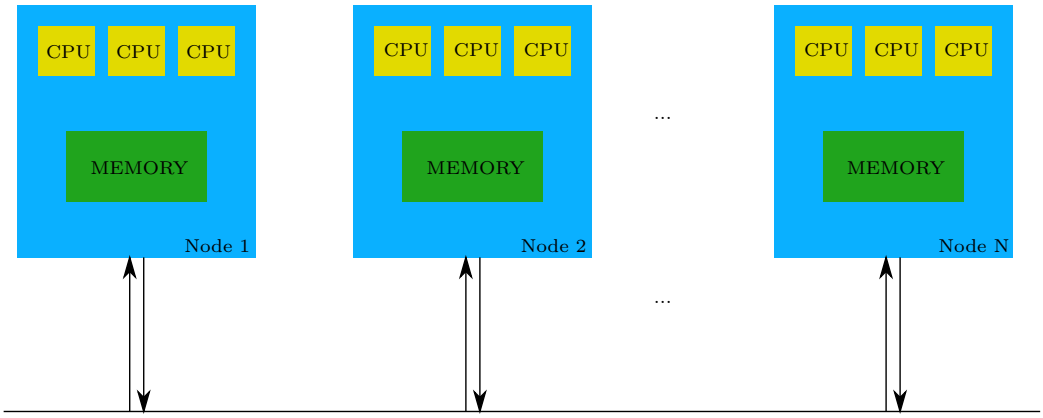

Figure 1.1 – A system with many nodes (N in total) that are connected

Linux) of a node. An alternative is the user-level initiation of remote DMA (RDMA) transactions, which eliminate the involvement of kernel (e.g. system calls) during the transfers, by reducing unwanted overheads such as the initiation latency [3]. User-level zero-copy RDMA transfers allow page migration, simplify multi-programming and also improve security, just as virtualization did in the past and well-known single-node systems [4].

Common user-level RDMA approaches cannot tolerate page faults during the RDMA transfer, thus avoid them by pinning multiple pages or whole address spaces of the processes. Pinning can hinder the memory utilization [5] and might not suffice due to optimizations of the Operating System (e.g. Transparent Huge Pages) or lack of privileges for a user/application. We also argue that pinning complicates the programming model because a kind of synchronization might be needed before/after the DMA and some extra actions from the application/user side are expected (e.g. someone has to unpin the memory after the DMA). A non-pinned memory-pages design results to page faults that require handling, which is the main work of this thesis.

1.2 Contributions

In cluster-communication systems we prefer to eliminate the kernel involvement and the multiple copies (kernel-to-user and user-to-kernel) when communicating. User-level zero-copy Remote DMA transfers answer this problem. This thesis addresses the need of supporting page faults that might occur during these RDMA transfers. This work was supported by the ExaNeSt project.

Initially, the author of this thesis implemented and tested the main components of the page fault handling mechanism using the Low-Power-Domain (LPD) DMA Engine embedded in the Processing System of the Zynq UltraScale+. When the prototype of ExaNest became more mature, this author worked on handling the page faults occurred during RDMA transfers through the custom DMA engine

residing in the Programmable Logic (FORTH PLDMA). The PLDMA Engine – FORTH PLDMA (designed by other members of the Lab) could provide useful and necessary information for the page fault handling mechanism, that required a different approach to be used than the approach considered when using the LPD DMA Engine in the Processing System. For the rest of this thesis, the author describes only the mechanism developed when using the FORTH PLDMA.

In order to achieve the goals of this thesis, the work consisted of three different parts.

First, this author worked on the background/theory of a page fault – what are the main reasons that it can be caused and the extensive research needed in kernel code in order to handle the fault properly. This includes all the information and knowledge that was collected and used in order to proceed with the implementation of the necessary modifications in the Linux kernel driver of ARM's IOMMU (SMMU) and the library for any application/user. Chapter 1 and Chapter 2 provide a detailed description of all this information.

Second, this author worked on the implementation: all modifications in the Linux kernel driver of ARM's IOMMU and the user library necessary for the mechanism, that will be activated when a page fault is caused during a RDMA from a user application. Also, as part of the implementation, the author modified the receiver hardware block of the FORTH PLDMA Engine and added extra functionality on the scheduler of the RDMA transfers (firmware), which is the R5 processor. Chapter 3 has a detailed description for the implementation.

Third, this author worked on evaluating the mechanism. This was essential in order to see the good and bad aspects of the current mechanism and propose possible future optimizations. Detailed information about the measurements and how they were conducted, can be found in Chapter 5.

1.3 Environment

1.3.1 Hardware

1.3.1.1 Trenz board

The hardware that was initially used was a TET0808 Trenz board with the FPGA: XCZU9EG-FFVC900-1 [6]. It is an MPSoC module integrating a Xilinx Zynq UltraScale+, that includes a 2 Giga Byte DDR4 SDRAM with 64-Bit width, 64 MByte Flash memory for configuration and operation, 20 Gigabit transceivers, and switch-mode power supplies for all on-board voltages.

1.3.1.2 Quad-FPGA Daughter Board

The hardware that was later and finally used in order to achieve our goals was based on Quad-FPGA Daughter Boards (QFDB), each one of them embedding four (4) Xilinx Zynq UltraScale+ MPSoCs: XCZU9EG-FFVC900 [2], with 64 Giga-Byte of

DDR4 SDRAM (16GB/FPGA at 160Gb/s), 512 Giga-Byte SSD/NVMe (4x PCIe v2 (8 GBytes/s)) and 10 High Speed Serial (HSS) links (10 Gb/s per link).

According to ExaNeSt project [1], the QFDB FPGAs are specialized for different tasks:

- two of them are pure computing nodes
- the "Storage FPGA", among other things, manages an SSD interface
- the "Network FPGA" is the QFDB network peer

The first experiments were conducted on QFDBs on mini feeders, and the final target was QFDBs embedded on a Mezzanine board. FPGAs on one QFDB (one hop) are connected (intra-QFDB) and QFDBs on a Mezzanine are also connected, consisting a network of nodes. The routing between nodes occurs using coordinates that differentiate them. In Figure 1.2 from [1], we can see a schematic of the QFDB module.

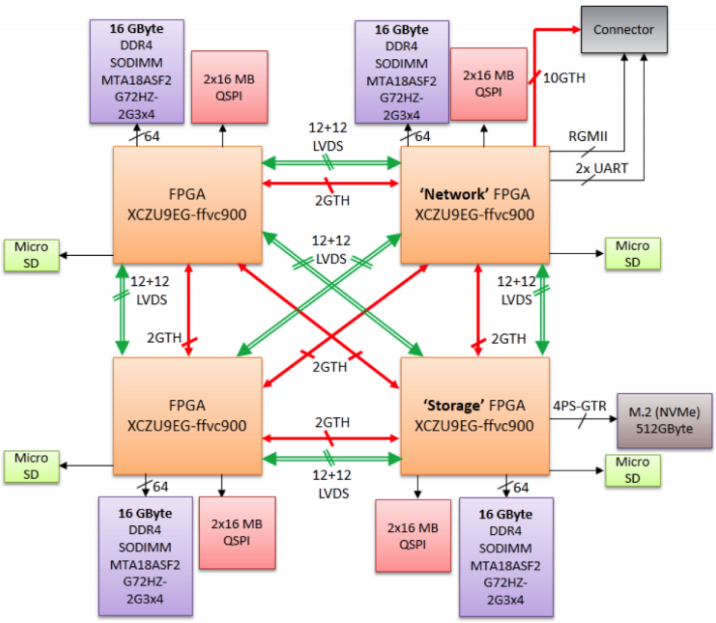

Figure 1.2 – Schematic of the QFDB module
Source: ExaNeSt [1]

1.3.1.3 Zynq UltraScale+

Zynq UltraScale+ MPSoC is the Xilinx second-generation Zynq platform, combining a Processing System (PS) and user-Programmable Logic (PL) into the same device. The Zynq UltraScale+ MPSoC has four (4) different power domains.

- Low-power domain (LPD)
- Full-power domain (FPD)

It also has the PL power domain (PLPD) and the Battery power domain (BPD), that are outside of the scope of this thesis. The Zynq UltraScale+ MPSoC PS block has three major processing units:

- Cortex-A53 application processing unit (APU)—ARM v8 architecture-based 64-bit quad-core multiprocessing CPU
- Cortex-R5 real-time processing unit (RPU)—ARM v7 architecture-based 32-bit dual real-time processing unit with dedicated tightly coupled memory (TCM)

The third processing unit is the Mali-400 graphics processing unit (GPU) with a pixel and geometry processor and a 64KB L2 cache. For the purposes of this thesis we will not use it.

As we can see in Figure 1.3, in the top-level block diagram of the Zynq UltraScale+ MPSoC, part of the Processing System (PS) is the APU that consists of four (4) Cortex-A53 processors that will run the system and more specifically they will run Linux that we will use in order to achieve our goals. Also, part of the PS is the RPU that consists of two (2) Cortex-R5 Real-Time processors, which is utilized by FORTH as the scheduler of ExaNeT, a network developed fully by FORTH as part of ExaNeSt.

As we can see, the SMMU and the CCI (Cache Coherent Interconnect) are in the same block in the top-level block diagram – that of course does not mean that they indeed are part of the same block, but that they collaborate in order to have coherent accesses to the memory. Last but not least, part of the top-level diagram is the Programmable Logic (PL). A user, among other things, can program the FPGA and add blocks in PL that can access the PS. This is where the custom PLDMA developed at FORTH (FORTH PLDMA) resides in, and we will embed part of our mechanism there.

1.3.1.4 Memory Management Unit

Serving the purposes of this thesis, we utilized one of Zynq UltraScale+ MPSoC's major components, which is the System Memory Management Unit (SMMU). Before we continue with the remaining of the thesis, we should explain the basic idea of the Memory Management Unit (MMU).

The Memory Management Unit (MMU)

The Memory Management Unit (MMU), in general, is a computer hardware unit having all memory references passed through itself, primarily performing the

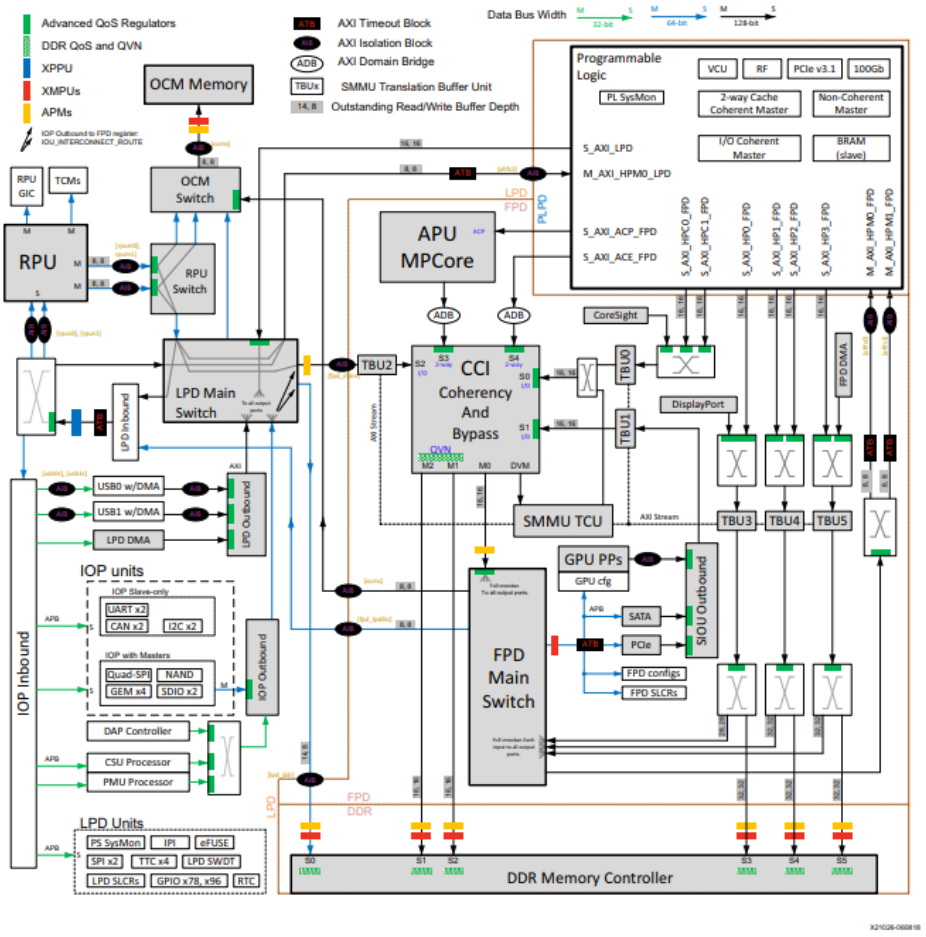

Figure 1.3 – Zynq UltraScale+ MPSoC Top-Level Block Diagram
Source: Xilinx [2]

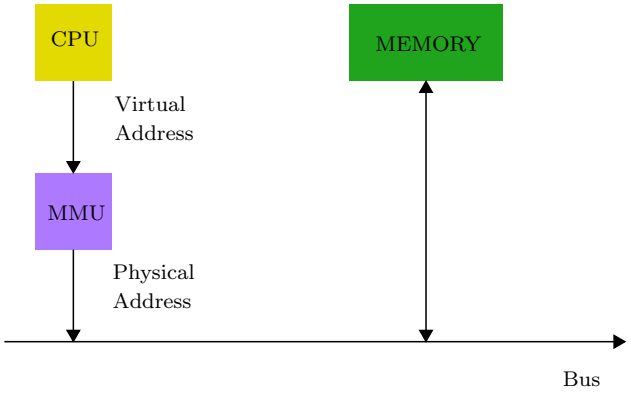

Figure 1.4 – A simple use of the MMU

translation of virtual memory addresses to physical addresses. It is usually implemented as part of the CPU, but it also can be in the form of a separate integrated circuit. We can see a simplified use of the MMU in Figure 1.4.

An MMU can effectively perform virtual memory management, handling at the same time memory protection, cache control, bus arbitration and, in simpler computer architectures (especially 8-bit systems), bank switching.

The I/O Memory Management Unit (IOMMU)

I/O Memory Management Unit (IOMMU) is an MMU that connects a Direct Memory Access-capable (DMA-capable) I/O bus to the main memory. IOMMU is basically responsible to map device-visible virtual addresses (also called device addresses or I/O addresses in this context) to physical addresses. In Figure 1.5 we can see a comparison of the IOMMU to the MMU.

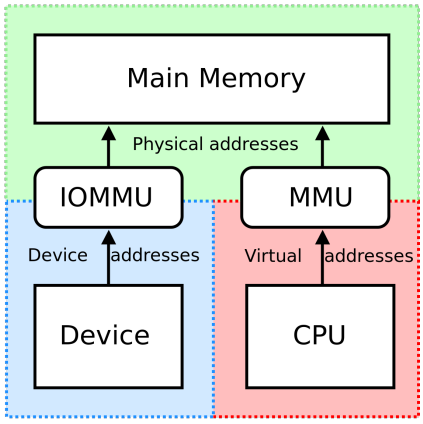

Figure 1.5 – Comparison of the IOMMU to the MMU
Source: Wikipedia

The System Memory Management Unit (SMMU)

In order to address some issues (e.g.: memory fragmentation, multiple DMA-capable masters, system integrity guarantee) and limitations, the ARM Architecture Virtualization Extensions introduced the System Memory Management Unit (SMMU) concept to the ARM Architecture. In other words, as we mentioned before, ARM's IOMMU is called SMMU.

The SMMU performs address translation of an incoming AXI (Advanced eXtensible Interface) [7] address (virtual address - VA) and AXI ID (mapped to a context) to an outgoing address (physical address - PA), based on address mapping and memory attribute information held in translation tables [2].

In Figure 1.6, we can see examples of where a SMMU block could be located in a system – coherent interconnects ensure cache coherence between masters.

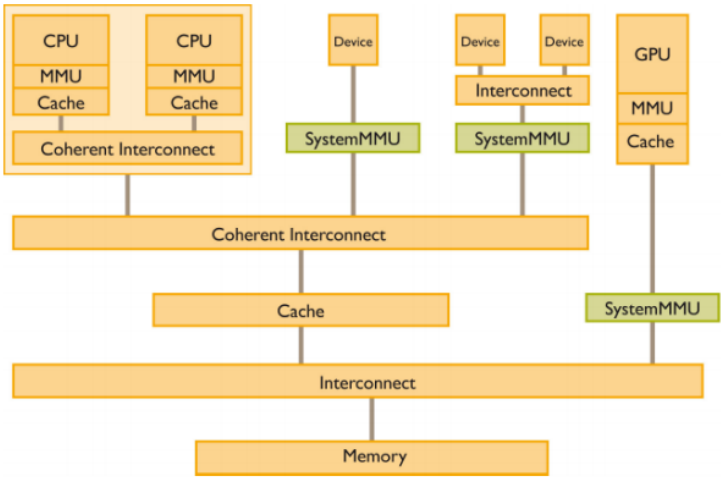

Figure 1.6 – Examples of where a SMMU could be located in a system.
Source: ARM

Translation Context

The translation context (as known as the context bank) provides information and resources required by the SMMU to process a transaction. SMMU can process multiple transaction streams from different threads of execution and supports multiple live translation contexts.

A translation context bank includes:

- state for configuring the translation process
- capturing fault status, and operations for maintaining cached translations

In implementations that support stage 1 followed by stage 2 translations:

- one translation context bank is specified for single-stage translation
- two translation context banks are specified for two-stage translation

A translation context bank is arranged as a table in the SMMU configuration address map. Each entry in the table occupies a 4 KB or 64 KB address space.

In theory, SMMU architecture provides space for up to 128 translation context banks. In our Zynq UltraScale+ MPSoC, we have only 16 translation context banks.

Each context bank of the SMMU can be considered as one page table – to be more accurate, each context bank has a field that points to a unique page table for this context. Each context bank has the **SMMU_CB**n**_TTBR**m, where m can be 0 or 1. TTBR0, as known as the Translation Table Base Register 0 holds

the base address of translation table 0. Respectively, the TTBR1 holds the base address of translation table 0.

It is recommended by ARM that TTBR0 should be used to store the offset of the page tables used by user processes and TTBR1 should be used to store the offset of the page tables used by the kernel. It seems that most Linux implementations for ARM have decided to eliminate the use of TTBR1 and stick to using TTBR0 for everything – this is also happening with the ARM SMMU driver (arm-smmu.c), we worked with.

For each context bank there is also the Translation Control Register, called **SMMU_CBn_TCR**, that determines translation properties, including which one of the Translation Table Base Registers, SMMU_CBn_TTBRm, defines the base address for the translation table walk required when an input address is not found in the TLB. An extension of the SMMU_CBn_TCR exists with the name **SMMU_CBn_TCR2**, that basically extends the SMMU_CBn_TCR by adding control information about the translation granule size and the size of the intermediate physical address.

1.3.2 Firmware

Page Fault Handling mechanism handles page faults caused during Remote Direct Memory Accesses (RDMAs). The mechanism, as we mentioned before, initially was built to support page faults caused when using the Direct Memory Access Engine, called ZDMA, embedded in the Processing System of the Zynq UltraScale+ MPSoC, which is a fundamental component of the QFDB. ZDMA was the first Engine that we used in order to support Virtual-Address Remote DMA, as part of the ExaNeSt. ZDMA has some fundamental limitations that would constrain the ultimate goal of ExaNeSt. Some of them include limited number of channels (8 for the low-power domain DMA engine), limited use of the address space (with ZDMA we could have access to a window of ∼500GB/s remotely – which is equal to the physical address space from PS to PL, meaning we could only have 500 GB memory space per node), one acknowledgement per transfer (not the best option for a resilient environment), "end-to-end" retransmission, which is not possible with ZDMA (page faults or packet transmission errors), not clear if multi-pathing is actually supported by ZDMA, fast notifications (ZDMA would incur one (1) extra round-trip-time (RTT) latency). Also, low performance was detectable with ZDMA, probably because of the small packet size of 64 Bytes and the small number of outstanding transactions, which was six (6). These are some of the reasons that led ExaNeSt to design and implement a new custom-made programmable logic Direct Memory Access (PLDMA) Engine. This is a work designed, implemented and supported mainly by our colleagues at FORTH [8, 9]. The new PLDMA among other things supports low latency transfers, resilience (fast retransmissions) and multi-pathing.

When ExaNeSt's own custom PLDMA was more mature, we moved our testing environment and implementation efforts for the Page Fault Handling mechanism

to it. From this point on, we will focus only on the work implemented and used in order to perform the transfers that will be completed (successful) after deliberate failures due to one or more page faults.

Although the design and implementation of the FORTH PLDMA Engine was not part of this thesis, it was a necessary component during the testing process of our Page Fault Handling mechanism, which is the reason why we will briefly describe its fundamental parts and the basic idea of it.

In Section 1.3.1.3 we mentioned the Real Time co-processor R5 embedded in the Processing System (PS) of the Zynq UltraScale+ MPSoC. Co-processor's main task is to segment, prioritize, initiate and monitor the DMA transfers. Each process, which runs under a Protection Domain can use 64 virtualized channels of the PLDMA. Our system supports up to 16 Protection Domains (since our SMMU has 16 context banks), thus the PLDMA can support up to $64x16 = 1024$ outstanding transfers.

Each transaction can be up to 16 KB. Real-time processor (R5) segments transfers into 16 KB blocks, and makes sure that they are 16 KB aligned (as a result some blocks might be $<$ 16 KB). Each transfer can have a parameterized number of outstanding transactions (currently the number is two (2)). The hardware segments 16 KB transactions into 256 Byte blocks. The main reason the transfers are splitted into packets with Maximum Transfer Unit (MTU) of 256 Bytes is that this size has shown to be more efficient at doing network congestion work. Furthermore small MTU can guarantee network buffers to be small, which saves utilization space and therefore cost, as mentioned in [9]. It is really important that R5 can monitor the state of each block, by using acknowledgments and if necessary fast retransmissions.

1.3.2.1 Remote Write — Initiator/Sender part

First, ARM's A53 cores and the user-space library provided to support RDMA, initialize both the TCM (scratchpad) of R5 and the mailbox dedicated to an R5 core. When the R5 core detects new information in its dedicated mailbox, it initiates an RDMA transfer, by also initializing accordingly the necessary components of the hardware. The custom hardware of PLDMA will be advised by the System Memory Management Unit (SMMU) to translate the remote virtual-address based on the protection domain of the process that triggered the DMA [10, 11]. Using the SMMU we can achieve transfers to/from pages that reside in DRAMs of different computing nodes. If this translation is successful we have a remote DMA write transfer to another (or the same) computing node. If the translation at the other node using, once again, the SMMU is successful, then we probably (not certainly, because it might trigger a permission fault) have the memory access, that eventually generates a positive acknowledgment that is sent to another mailbox dedicated to the R5 of the initiator node. This is how the node that initiated the transfer is becoming aware of the completion of the transfer.

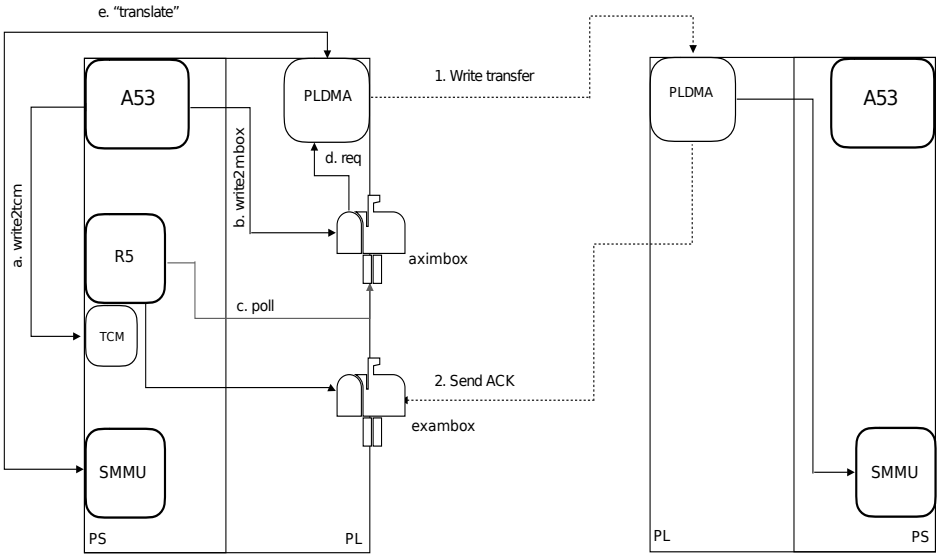

Figure 1.7 – Remote write flow path between two nodes

1.3.2.2 Remote Read — Target/Receiver part

For a remote read transaction, the process, running on an ARM A53 core, triggers the transfers using a user-space library that sets a packetizer responsible to forward the read request to the corresponding (probably remote) node. Then, a mailbox dedicated to the R5 of the target node receives the request and transforms, in a way, this request to a write transfer. This way we make sure that we have the same mechanism as a write request, which makes easier the programming model. In simple words, this allows us to reuse same blocks of hardware that can perfectly fit our purposes. The target node will write to the memory of the initiator node, which effectively acts like a read request.

1.3.3 Software

Most of the effort for this thesis was related to Software aspects. From the environment we used to test our prototype-mechanism to the user-space library of the mechanism and the generation of the needed image files (kernel, real-time, etc). Part of the software that was used for our mechanism includes the Netlink sockets, which we will briefly describe later in this sub-section.

1.3.3.1 Linux

We chose to use the Linux OS as the Operating System for the purposes of this thesis - the Linux version was 4.9.0.

At first, while working on Trenz boards (Section 1.3.1.1), we were using SD cards loaded with the necessary image files, such as the image file of the kernel

(Image), the device tree, the FSBL etc. Before migrating to our brand new prototype, the QFDB, a tool called "yat" was implemented at FORTH to automatically generate all necessary image files according to the requested and given as input details of the platform. After setting it up, by giving the Xilinx Software Development Kit (SDK) version and the name of the profile (trenz, QFDB, etc..), we could generate our images and finally our BOOT.bin.

Since most part of this work was mainly kernel development, when it comes to Linux we had to generate a new Linux kernel image after every new modification to the ARM SMMU Linux driver. After the build was done, we could "kexec" the new kernel Image on top of the Linux environment on QFDBs, that was already boot-ed. The process of booting our own kernel image can be found in Appendix B.

1.3.3.2 Xilinx Environment

We used Vivado, a software suite produced by Xilinx for synthesis and analysis of HDL designs, and SDK (Software Development Kit), that is the Integrated Design Environment for creating embedded applications on any of Xilinx's microprocessors like the Zynq UltraScale+ MPSoC.

Although it was not part of this work, Xilinx Vivado was used to export the bitstream (.bit) and the hardware design/description file (.hdf), and the Xilinx SDK to create the device tree (.dtb) and the first-stage-boot-loader (.fsbl) from the exported hardware design/description file (.hdf). As we mentioned before, prior to our migration to the QFDB prototype, we had in our hands a tool, called "yat", that could generate all necessary files for us, such as the First Stage Boot Loader (FSBL), the Power Management Unit Firmware (PMUFW), the Flatened Device Tree image for the board (DTB), the EL3 Secure Monitor (BL31), the Second Stage Boot Loader (U-Boot), the Linux Kernel Image (Kernel), the Ramfs image (Initramfs) and finally the BOOT.bin, that included all the necessary image files.

Part of this thesis was to build a modified, by the author, version of Real-Time R5 co-processor, in order to produce a firmware that could support our mechanism. In order to achieve this we used Xilinx SDK tool.

The version of SDK mainly used during this work was: 2017.2 at first, and then 2017.4.

1.3.3.3 Modules/Drivers

At this point, it is good to define "modules" and "drivers", that were a big part of this thesis.

A device driver (commonly referred to simply as a driver) is a computer program that operates or controls a particular type of device that is attached to a computer. A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used.

A module is a piece of code that can be loaded and unloaded into the kernel upon demand. Modules extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want to add a new functionality.

In order to test and evaluate the mechanism of this thesis, the author developed and modified some drivers and modules, which will be described in Chapter 3.

In Figure 1.8, we can see how a system can use drivers and modules.

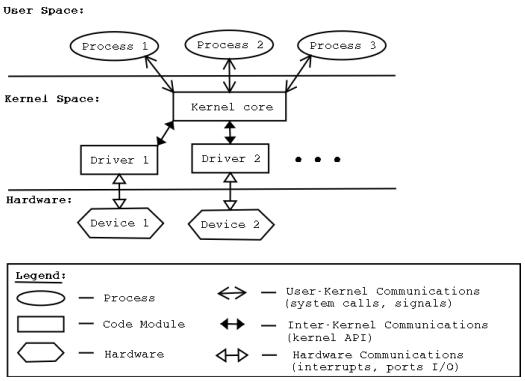

Figure 1.8 – The Kernel, The Processes And The Hardware. Source: haifux.org

1.3.3.4 Netlink Sockets

Netlink sockets first appeared in Linux kernel 2.2 mainly as a flexible alternative to the IOCTL communication method, that can be used between userspace processes and the kernel. One of the main disadvantages of IOCTL method, apart from the complexity to use, is that the IOCTL handlers cannot send asynchronous messages to the userspace process from the kernel. When using the IOCTL method it is required from the programmer to define IOCTL numbers, that of course by itself increases the complexity considerably. The Linux header file */usr/include/asm/iocth.h* defines macros that must be used to create the IOCTL command number for each command used through IOCTL. This number should be unique for the whole system, and picking it arbitrarily is a bad idea, that could lead to bad situations, including damage to hardware [12]. The userspace processes that use the Netlink library "open" and then "register" a Netlink socket, in order to handle bidirectional communication (send and receive messages) from the kernel.

It should be mentioned that in theory Netlink sockets can also be used for the

communication between two (2) userspace processes, but this is not very common and certainly not the original goal of Netlink sockets.

Some advantages using Netlink sockets over other ways of communication:

- No need for polling
- Kernel can be the initiator of sending asynchronous messages to userspace. For alternatives such as IOCTL or sysfs entry, an action from userspace is required.
- Supports multicast transmission

According to the manual, Netlink protocol is not considered reliable and may drop messages when an out-of-memory condition or other errors occur. When a message from kernel to userspace cannot be sent, the message will be dropped, which means the application and the kernel will no longer be able to have the same view of the kernel state. It is the responsibility of the application to detect it when this happens.

1.3.3.5 Tasklets

Several tasks among those executed by the kernel are not critical, which means that they can be delayed for a long period of time, if it is necessary.

In general, interrupt service routines are not preempt-able, until the corresponding interrupt handler has terminated. Tasklets overcome this critical restriction, by being preempt-able, which means they can execute while having all interrupts enabled. By having this functionality, we can keep the kernel response time relatively small: something very important for time-critical applications, whose interrupt requests should be serviced in a few milliseconds. Another name for tasklets is deferrable functions. A given tasklet will run on only one CPU, the CPU on which the tasklet was scheduled. The same tasklet will never run on more than one CPU of a given processor at the same time. Different tasklets can run on different CPUs simultaneously.

In order to use a tasklet in a driver someone has to declare it first e.g DECLARE_TASKLET(<name of tasklet>, <name of function/handler>, <input>), where name of <function/handler> points to the code to be executed and the <input> is the data (input parameter) passed to the tasklet. When using this call, the tasklet is enabled by default, which means it will be executed as soon as possible after we schedule it. The second part is to schedule the tasklet, calling the tasklet_schedule(<name of tasklet>) method. The scheduling of a tasklet can happen anytime as part of the main interrupt handler, but the tasklet will run when it is safe to run. When the driver is about to exit, we can remove the tasklet by calling tasklet_kill(<name of tasklet>). This function ensures that the tasklet will not run again.

Chapter 2

Related Work

2.1 Page Fault Support in NICs

While working on our topic, that is based on Page Fault support necessary when using RDMA technology, we found a similar topic-wise work, that was presented in ASPLOS 2017. The title of this work is "Page Fault Support for Network Controllers", which was conducted by I. Lesokhin et. al. [13]. This seems to be the most relevant work to ours, which is why we are going to cover it in this Chapter.

Isolation of the address spaces between different applications (or virtual machines) is one key benefit that comes with virtual memory. Also there is a simplicity that comes with it for the programmers; they do not bother to properly manage the memory or the storage. Last but not least, virtual memory allows optimizations that are quite important performance- and memory-utilization-wise, such as demand paging. According to the authors, programmers who write software that initiates DMAs do not enjoy all the benefits coming with virtual memory, because DMAs cannot tolerate page faults.

Table 2.1 – Direct network I/O terminology

Term	Description
IOchannel	hardware-provided virtual NIC instance
IOuser	untrusted process or VM assigned with IOchannel
IOprovider	trusted operating system (OS) or hypervisor

The authors of the paper use the terminology that can be found in Table 2.1. NICs allow IOusers to bypass the IOprovider, which is something that led to the sudden increase of NICs usage. According to the authors, a lot of research focused on benefits and improvements that can happen in such ecosystem, though mainly ignoring one thing: losing virtual memory benefits due to lack of DMA page fault support. There are currently two ways to avoid DMA page faults: static and dynamic pinning. Static pinning of the whole address space of an application enjoys

simple programming model, but loses on canonical optimizations, such as demand-paging. Alternatively, buffers can be pinned and unpinned before and after they are DMAed. Although this latter approach enjoys the canonical optimizations of virtual memory, it has two drawbacks: it complicates the programming model and when frequently used it hampers the performance.

2.1.1 InfiniBand Page Fault Support

The mechanism of InfiniBand page fault support consists of two different flows.

- Network Page Fault (NPF)

    1. When a new request is received, NIC consults IOMMU page tables. The NIC finds and marks the page involved that is not present.
    2. The modified firmware detects the fault and raises NPF interrupt.
    3. The driver catches the interrupt.
    4. The driver's NPF interrupt handler queries the OS regarding the physical address of the faulting IOVA (I/O virtual address). If necessary, the OS allocates the pages, possibly retrieving their content from secondary storage.
    5. The driver updates the IOMMU page table with the physical address and informs the firmware that the NPF has been resolved.

- Invalidation

    1. The operating system requests from the driver to remove the old IOVA and stop the device from using it.
    2. The driver updates the IOMMU page tables accordingly and issues the invalidation.
    3. The NIC acknowledges the invalidation.
    4. And then, the driver notifies the operating system that the relevant pages can safely be reused.

InfiniBand supports Reliable Connection (RC), which means that the mechanism can let the sender know to stop sending upon dealing with a page fault and retransmit when ready – this works because the data is local when a sender encounters a page fault. Receiving can be more tricky, but still doable using RNR (receiver-not-ready) messages to suspend sender upon NPFs. In addition to send/receive, RC supports RDMA operations, but in some cases RC does not permit RNR negative acknowledgements. For instance, there is no way to stop the sender during remote read operations, which means packets will be dropped. The only way for the sender to retransmit is to rewind the transfer after the page fault is resolved.

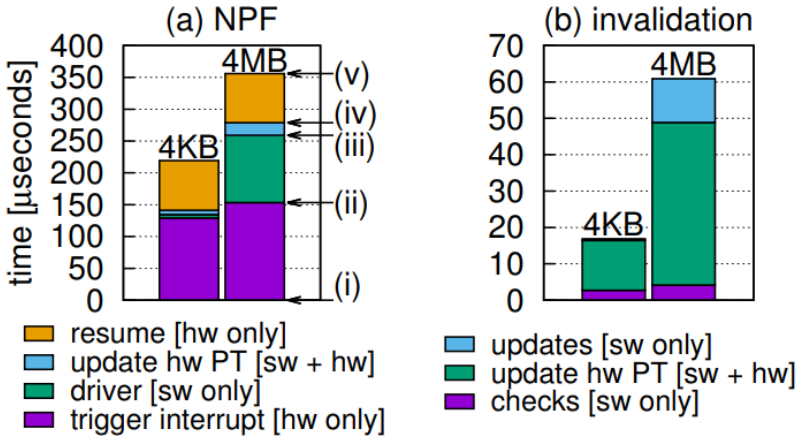

Figure 2.1 – Execution breakdown of NPF and invalidation
Source: [13]

In Figure 2.1 (a), we can see the average overhead breakdown of minor NPFs, which means no disk access, when sending 4KB and 4MB messages. We see a breakdown of the following events:

(i) an NPF that occurs is observed by the IOMMU and an interrupt is triggered

(ii) invocation of the driver's NPF handler

(iii) the mapping from the OS (the physical address that corresponds to the IOVA) is "recovered" and later sent to the driver

(iv) the driver finishes updating the IOMMU page table accordingly (due to coherence issues the driver needs to communicate with the IOMMU that is on the NIC – page tables normally reside in DRAM)

(v) the NIC identifies the update and resumes the transmission

As noted in the paper and can also be seen on Figure 2.1 (a), a minor NPF takes 220 µsec for a 4KB message, 90% of which is due to hardware (firmware). As it is explained later, this is a typical duration for Mellanox NIC firmware activity (not only for NPFs), "as the goal of the NIC circuitry that runs the firmware is usually to handle error paths", which is why it is considered acceptable to be slow. When the message is 4MB, the duration increases to 350 µsec, due to software overheads such as more translations and allocations by the OS.

In Figure 2.1 (b) we can see the overhead caused by the invalidation flow. First, the driver identifies the memory region that needs to be invalidated and checks if the mapping exists in the IOMMU on the NIC. If the mapping does not exist, no

additional overhead is incurred. Secondly, the driver needs to update the IOMMU page tables and its own internal state.

2.1.2 Ethernet Page Fault Support

Ethernet is considered more mainstream as a scenario with NICs. IOuser (application or process) utilizes a direct network channel through a regular Ethernet NIC. An IOuser probably uses TCP/IP protocol to drive its direct channel. This approach does not have the benefits of InfiniBand (Section 2.1.1) technology, because this is a very different ecosystem. There was hope that they could be benefited from TCP reliable communication (which includes retransmissions when a packet is lost), but dropping of packets does not seem a viable solution.

Dropping packets upon rNPFs seems to be an important problem due to the cold ring problem. At start, no buffers are pinned and as a result page faults are triggered one after the other. Meanwhile, packets get dropped, triggering TCP retransmissions and congestion avoidance that nearly deadlock the communication, or as the authors claim, completely halt it in the worst case. Cold ring problem does not occur only on startup situations, other examples would be when a virtual machine is resumed, brought back from swap memory, due to NUMA migration, copy-on-write (COW) semantics etc. Their proposed solution for rNPFs includes a Backup ring.

As we can see in the high level of the design of the backup ring (Figure 2.2), there is a communication between the NIC, the IOprovider and the IOuser in order for the mechaninsm to work. The backup ring is denoted as "buffer" in the schematic.

Below we can see a brief description of all steps followed in accordance with Figure 2.2.

1. Traffic is received from the network.

2. For each incoming packet, NIC inspects the target receive buffer of IOuser. If the buffer is available, data is written directly into it.

3. For each incoming packet, NIC inspects the target receive buffer of IOuser. If a page fault is encountered, packet is written to a small pinned backup ring, owned by the IOprovider.

4. After the IOprovider resolves the rNPFs, it copies (or merges) the packet into the original receive buffer of the IOuser.

In order to maintain the ordering, the NIC does not report reception of new packets to the IOuser until all previous page faults have been handled.

As the authors reveal, in practice it was not possible to implement the backup ring in firmware. Instead, they prototyped the driver within the IOprovider. All incoming packets were duplicated by the NIC into two receive rings: a primary

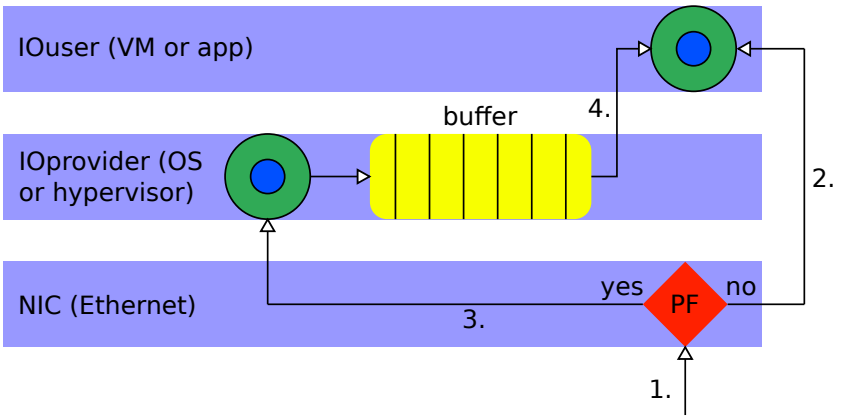

Figure 2.2 – High level design of the backup ring

p, where page faults could occur, and a secondary s, which was populated with pinned buffers. If no rNPFs on p, the duplicate packets in s were discarded. If rNPF on p, the driver would utilize s as the backup ring, copying faulty packets from s to an intermediate queue q. After resolving the fault, the driver would copy the packets from q to p. Unfortunately, the lack of true hardware support leads to halved throughput, due to the duplication of packets.

Their evaluation was splitted in three different categories: memory utilization, performance overheads and code complexity. In terms of memory utilization, their experiments show that NPF dynamic working set is more advantageous than static pinning. As for the performance overheads, they showed that there is an advantage of RDMA (zero-copying) over copying, especially for larger messages ($\geq 32KBytes$) and that the NPF configuration has similar performance as pin-down cache (a technique that unpins after exceeding an upper limit). In some HPC workloads, the NPF support can provide benefits without the need to pin. As for the code complexity, although as they say it is difficult to quantitatively measure the benefit of NPFs in reducing code complexity, they propose considering lines of code (LOC) involved. The authors state that very few lines (a few tens) of code needed to be modified or added to the design in order to work, compared to implementation requiring pinning (a few thousands LOC).

2.2 Pinning-Based Networks

For the approach of pinning-based techniques, one of the fundamental works we read was “New DMA Registration Strategy for Pinning-Based High Performance Networks” [5].

In this work, the authors propose a new memory registration strategy for supporting RDMA operations over pinning-based networks. One of the motivations was that existing approaches at that time were not efficient when implementing GAS (Global Address Space) languages. In fact, existing approaches, although

they could often maximize the bandwidth, required a level of synchronization that discouraged one-side communication and caused significant latency costs for small messages.

Their proposed memory registration strategy is described by an algorithm. The "Firehose" algorithm exposes one-sided zero-copy communication in the common case and at the same time minimizes the number of synchronization messages that is required in order to support remote memory operations.

Algorithm description:

- Determine largest amount of application memory that can be registered (upper bound on total number of physical page frames that can be pinned simultaneously – this is limited to some reasonable fraction of existing physical memory). If this amount is in total M bytes, using P byte pages, then only a total of M/P pages can be pinned at any time during execution. This mechanism supports many nodes, which is why available space is equally divided and $F = \frac{M}{P*(nodes-1)}$ physical pages can assigned to each remote node.

- Conceptually, a firehose is a handle to a remote page. Each node owns F of these firehoses. The authors mention that a functionality of freeing firehoses to establish new mappings to remote pages in order to pin and then serve the pending remote memory operations is supported.

- A round-trip synchronization message is required in order for a node to situate one of its firehoses, by mapping it to a region in the remote virtual memory. This way it is guaranteed by the remote node that the virtual page will be pinned for the duration of the mapping.

In general, the Firehose algorithm includes some tunable parameters, such as the maximum amount of the physical memory used for remote firehoses (M), the maximum size of bucket victim FIFO queue (MAXVICTIM), which will be used to unpin only when necessary. Also, the bucket size, which is the basic unit of physical and virtual memory of the Firehose algorithm, is configurable (equal to page size by default).

The main benefit of this approach is that the handshake per transfer is avoided, which in general case is necessary in Rendezvous approaches, because it is the only way to advertise and make sure that the related virtual pages are resident (pinned) in memory. In the Firehose approach, pinning happens only once at the beginning in the common case and the handshake is required for the non common case, which comes with a cost that is negligible. This was the main focus of this work; to reduce the frequency of the registration operations.

Existing pinning-based strategies:

- "Pin Everything": This method is not on-demand-based. A single segment of memory is pinned at startup and kept pinned until the program terminates. This method seems rational when the total memory requirements are known and constrained to a relatively reasonable size within the physical

memory limits of the host. In this case, it is preferable to pin the entire remotely-accessible region of memory at startup. It does not require additional synchronization and can complete as one-sided.

- "Bounce Buffer": This method, uses temporary buffers residing in pinned memory to hold data for outgoing and incoming DMA operations. In case of a "put", the DMA operation completes, the target processor is informed of its delivery and must copy the data to the final remote destination. When a "get" request is received, targeted code copies the data into a bounce buffer and executes a "put" operation to the requesting node. The main advantage is that the cost of registration is being paid only at startup and no more pinning is required. It has some disadvantages as well. It is two-sided in a strict way (latency for remote transfer operations is likely to increase), copying costs may be significant (even for small messages, because of the interrupts and the different kind of CPU and TLB invalidations) and and complexity and handshake overheads may appear, arising questions about the scalability of the mechanism.

- "Rendezvous": For large transfers, cost of pinning on-demand can be amortized over more data and provide performance improvements over the use of many bounce buffers. Rendezvous includes two (2) main steps:

  1. Send a message to the remote node indicating the region to be pinned
     - for "puts", remote node processes the message and pins the relevant memory region and then sends a reply, indicating that the DMA transfer can be initiated.
     - for "gets", similar approach with "puts": as an optimization, reply may coalesce acknowledgement and payload.
  2. Optionally, there may be some final handshaking to unpin the relevant regions once the DMA transfer is complete.

  The cost of registration is paid on every operation, which is prohibitive for small messages and debatable for larger messages.

For the evaluation, Firehose assumed M=400MB of pinnable memory, MAXVICTIM=50MB, and total pinned memory: M+MAXVICTIM=450MB. Tests are run long enough to reach a steady state. Bucket size is set to single-page buckets to provide an upper bound on the overhead in managing Firehose data.

The authors used two parallel applications implemented in Titanium (GAS language) as their benchmarks: Cannon's Matrix Multiplication and a Bitonic Sort.

Their results show that Firehose is twice as good as the Rendezvous average put latency, with no-unpin. However, the results of the synthetic microbenchmark (Figure 2.3), show that the latency of Firehose past the M+MAXVICTIM

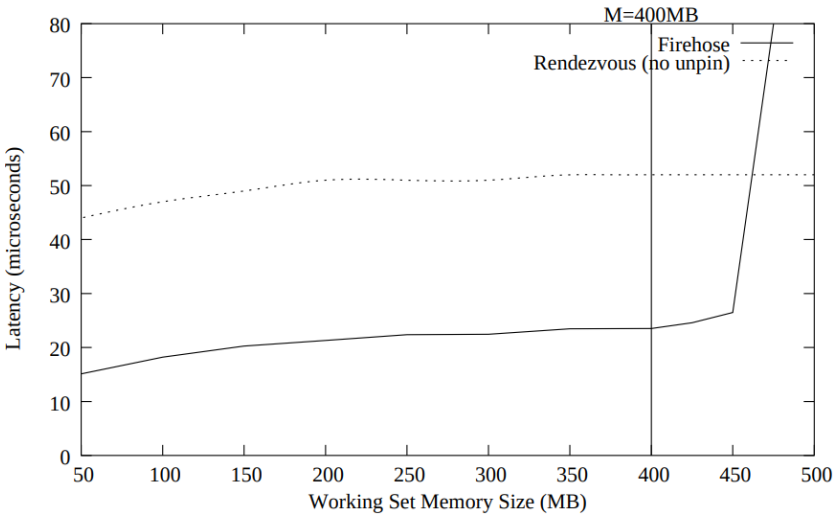

Figure 2.3 – 8-byte put latency over increasing working set memory size (M = 400MB)
Source: [5]

point (450MB) increases sharply, approaching the Rendezvous with no-unpin performance. Also, although not depicted in Figure 2.3, their results for the large-message bandwidth show that when the working set exceeds M, the Firehose hit rate decreases (more "handshaking" is required).

The above results show that there is a high overhead of pinning, which can become non-tolerable after the point of pinnable memory (M) – which by itself can become a limiting factor. Also, we argue that pinning the whole address-space of the process of each node at startup has to be possible size-wise and it cannot be the case all the times. Furthermore, it does not look like a viable solution, since a user application might need to allocate (request) more virtual pages, which either we will have to pin (costly) or we will need a mechanism to page them in when a page fault occurs. Besides that, Linux Operating System comes with some optimization techniques like Transparent Huge Pages –THP (see Section 3.1.2.3) enabled by default, that will eventually cause (minor) page faults that require handling. If we disable THP we will not take advantage of the possible performance benefits provided by it.

Chapter 3

Remote Page Fault Handling

3.1 Theory

3.1.1 Definition

According to a definition provided by Daniel P. Bovet and Marco Cesati [14] a page fault occurs when "the addressed page is not present in memory, the corresponding Page Table entry is null, or a violation of the paging protection mechanism has occurred". Before attempting to find a solution for page faults, it is important to try to break down this definition in a way to find what is the actual problem.

The first part of the definition states that the addressed page is not present in memory, which is probably the most obvious assumption that everyone has in mind when they hear about a page fault. For different reasons, some of them will be covered later (see Section 3.1.2), it is possible that a page does not reside in RAM (memory). This means, that when a legitimate user (process) tries to have access to a page (whether it is a read or write request), they will have to wait until kernel has provided the related physical page (frame) to them by bringing (or allocating) a page frame. While doing this work, kernel context-switches to a different process that can utilize the core, until the page fault has been resolved, which is when the core is ready to switch back to the process that initially triggered the page fault.

The second part says that there is no mapping (yet), meaning a mapping of the virtual address to the physical address of the page. This can happen for many reasons. One reason is that the virtual address is illegal, in a sense that it should not be allowed to be translated and eventually have access to the memory. Operating Systems such as Windows report invalid memory references in this way. Another cause is that the process has not shown interest in accessing it yet, so the kernel "lazily" decided not to have a corresponding page frame for it, since it might not use it (see Demand Paging, Section 3.1.2.1).

The third part describes a violation of the paging protection mechanism. Memory protection in paging is achieved by having protection bits for each page. These bits are associated with each Page Table Entry (also known as PTE) and specify

the protection on the corresponding page. A valid/invalid ("v") bit guards against a process trying to access a page that does not belong to its address space. Read ("r"), write ("w"), and execute ("x") bits are used to allow accesses of the corresponding type. Illegal attempts institute a memory-protection violation, that causes a hardware trap to the Operating System [15].

Page faults are mainly categorized in two types: minor and major page faults. Minor page faults are those which can be handled by just reclaiming (or allocating) a page frame. Major page faults are the faults, whose handling requires I/O, e.g. disk access. Having this in mind, it is safe to say that major page faults induce greater overheads than minor page faults, because more time is required in order to recover from them.

3.1.2 Page Fault Causes

In general, two of the main reasons page faults are caused in CPUs are due to some techniques in Operating Systems (e.g. Linux). In our system, we examine and handle page faults triggered in the System Memory Management Unit (SMMU). Another reason for a page fault to occur is due to the Transparent Huge Pages (THP) mechanism.

3.1.2.1 Demand Paging

Demand paging is a dynamic memory allocation technique that defers page frame allocation until the last possible moment. This moment is when a process attempts to address a page that does not reside in RAM (memory), thus causing a page fault.

The reason why this mechanism makes sense is that processes do not utilize all the pages that their addresses are mapped, as part of their address space, right from the beginning. In fact, some of these addresses may never be used by the process. The principle of program locality ensures that at each stage of execution only a small subset of addresses will be used [14].

This technique increases the average number of free page frames in the system and therefore allows better utilization of memory. Also, as mentioned in Section [14], it allows the system as a whole to get better throughput with the same amount of memory.

3.1.2.2 Copy On Write

The creation of processes from the first-generation Unix systems was in a way "graceless". During the fork() system call, kernel was responsible to duplicate the whole parent process address space and assign the copy to the child process. This included many steps such as: allocating page frames for the page tables of the child process, for the pages of the child process, initializing the page table of the child process and copying the pages of the parent into the corresponding pages of the child process. Because of this procedure, we had many memory accesses

and a great consumption in CPU cycles, which means that this activity was time consuming.

Modern Unix kernels, including Linux, utilize a new approach called Copy on Write (COW). Initially, the page frames are shared between the parent and the child process, instead of having a duplicate. Neither the parent nor the child process are allowed to modify any content on the page frames (read-only). If any of them want to modify any page frame, an exception occurs. This is the moment that the kernel duplicates the page into a new page frame and marks it as writable, while the original page frame remains read-only. When the other process tries to write into it, the kernel checks whether it is the only owner and in such case makes the page frame writable for the process.

3.1.2.3 Transparent Huge Pages

Most of the architectures supported by Linux are able to work with pages larger than 4KB, such as 2MB or even 1GB pages. These are considered "huge pages" compared to what is considered normal 4KB page size. Huge pages are in common case beneficial performance-wise, since they can mostly offload the Table Look-aside Buffers (TLBs), making at the same time TLB misses less expensive. According to [16], the mechanism of Transparent Huge Pages (THP) works quietly, substituting huge pages into a process' address space, when these physically contiguous pages are available and it appears that the process would benefit from this mechanism. This feature was first added in 2.6.38 kernel.

One important part of this mechanism is the *khugepaged* kernel thread, that occasionally attempts to substitute smaller pages being used currently with a hugepage allocation, thus maximizing transparent huge page usage. The reason it is called transparent, is because the user does not need to modify the applications in order to work with the new page size. This kernel thread will automatically start when transparent_hugepage/enabled option is set to "always" or "madvise", and it will be automatically shutdown if it is set to "never". This option exists in the path: /sys/kernel/mm/transparent_hugepage/enabled and can have two values:

- always - always use THP
- never - disable THP

It might be the case that the kernel thread khugepaged is not successful in converting "small"-sized pages (e.g. 4KB) to huge pages (e.g. 2MB). Even in that case, it may still be taking processor time to search for candidate pages [17].

This triggers random page faults during RDMA transfers even when the buffers are touched and there is no swap device for the kernel to move pages. The reason we could actually witness page faults due to this mechanism was because it seems that while kernel was trying to merge some pages, the previous mappings of these pages were invalid the moment a transfer would try to be translated from the local SMMU. Since someone might want to keep the optimization that the THP

mechanism offers in performance, it works as a great motivation for us to have a mechanism that supports page faults during an RDMA transfer.

3.2 Approach

In Section 1.3.2, we briefly described the DMA engine and the environment that we worked on in order to implement a hardware-software co-design that would support the page faults caused during virtual-address RDMAs.

During an RDMA that is based on virtual-addresses, a page fault might occur in:

- the source buffer (address)
- the destination buffer (address)
- both buffers (addresses)

We believe that the most common case for a page fault to occur during an RDMA is the destination buffer. We expect that when an application triggers an RDMA transfer, the source buffer will be "touched" prior to the transfer. We will still cover this scenario, which is possible to happen for many reasons, e.g internal optimizations in Linux, such as Transparent Huge Pages (as described in Section 3.1.2.3).

3.2.1 Common path

The main task of handling the page faults is to make sure that the pages are brought into the main memory, if they are valid, so that after the retransmission, the transfer will succeed. We designed two implementations to achieve this:

1. Netlink Sockets: A user-space thread is responsible to touch the pages that previously triggered a page fault. Kernel (driver) notifies the thread of the corresponding protection domain to touch the pages.
2. get_user_pages(): Kernel-space approach of bringing pages to main memory.

The get_user_pages() approach is very recent in our work, which means it might require more investigation in the future about its characteristics and features. It seems interesting and quite beneficial, because using it will allows us to handle page faults completely from kernel-space without any userspace involvement. Also, we will probably be more efficient, since kernel handling will result to less context-switch overheads compared to Netlink/touch pages from user-space approach. Last but not least, we also call this approach "Touch-Ahead", because it can page-in more than one pages. Netlink-sockets approach can touch one (1) page per invocation, thus it is called "Touch-A-Page".

The current version of the mechanism requires the utilization of Netlink sockets even for get_user_pages() approach: the packetizer implemented at ICS-FORTH

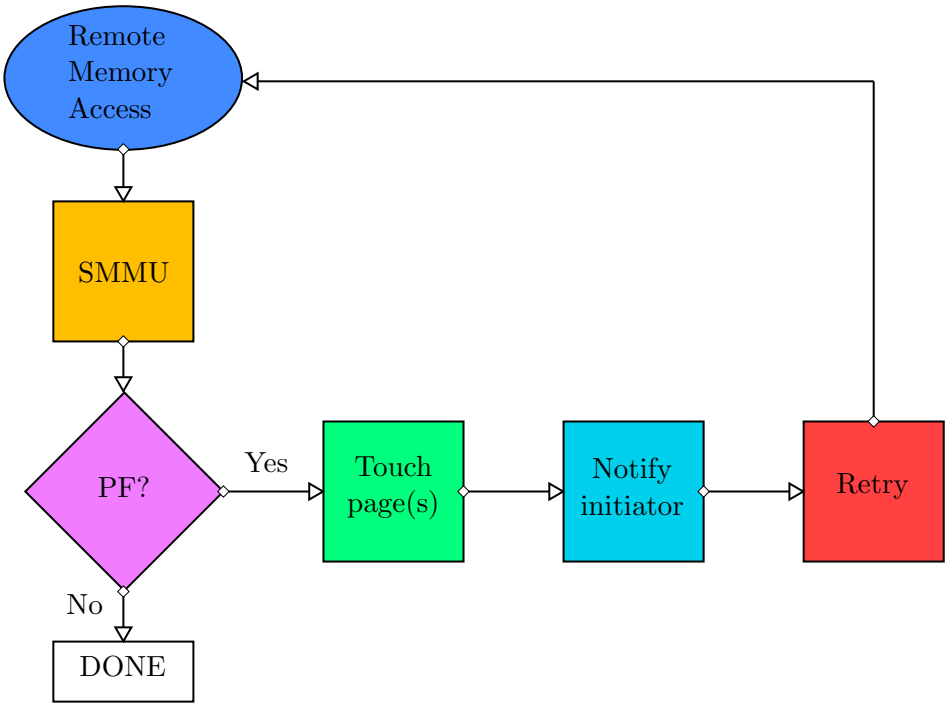

Figure 3.1 – General Page Fault flow

(from others at FORTH) works when it is configured and utilized by user-space applications. Until this very moment, we can only send messages from the provided packetizer to the mailbox that is polled by the R5 co-processor through user-space. Netlink sockets appear to be the best choice to communicate messages from kernel to userspace and thus to the mailbox, in order for the transactions to be retransmitted when the corresponding page fault has been resolved (this will be covered in detail later).

We will now describe the common parts of the page fault mechanism both from the sender- and the receiver-side and later we will describe the distinctive details of each case (read and write path). Common parts can be found both in user-space library and the driver of SMMU. Below we give a description of them.

Page Fault Library A page fault user-space library was necessary to fulfill the needs and purposes of our work.

The library includes the following methods:

void enable_pgfault_mechanism(): This method is responsible to create the thread that belongs to the process of a specific protection domain. This thread will be responsible to be woken up and catch the message coming from kernel through Netlink sockets, in order to handle the pafe fault that was triggered.

void sig_handler(int signo): This method is responsible to catch and handle a segmentation fault. This method was initially built for testing purposes, since in our environment it is expected that when a segmentation fault occurs (e.g. erroneous not-mapped virtual address), the application will crash, as it happens in most of the systems.

However, while working on our mechanism and more specifically the "Netlink sockets" solution (which touches one page, in contrast to our get_user_pages() approach), we experienced an interesting phenomenon that can be seen in Figure 3.2. In a micro-benchmark that consists of many iterations that each iteration experiences page faults, it is possible to be requested from the userspace library to touch (page in) a page that belongs to a previous iteration and thus no longer belongs to the address space of the process, causing a segmentation fault. With the help of our segmentation fault handler we overcome this obstacle. By using the get_user_pages() approach, we do not witness such problem at all, which is why it is the preferred solution.

int pckzer_to_mbox(uint64_t *pckzer_addr, uint64_t dst_coord, int trid, int seqnum, int pdid): This method takes as input arguments the virtual (mmap'ed) address of the packetizer and the useful information for the retransmission of the previously faulty transfer due to a page fault, such as: the coordinates of the computing node we are going to send to its mailbox, that is polled by its local Real-Time R5 co-processor, the transaction identifier (id) that will be retransmitted, the sequence number of this transfer (for debugging purposes mostly), and the protection domain identifier (id), that this transfer belongs to (for safety check, since packetizer will either way have the protection domain wired to the packet – cannot be changed by user for security reasons). The least significant bits (LSB) of the message to-be-sent consist of the opcode for the message and are given by the driver. The opcode is used by the mailbox to distinguish different types of messages destined to the mailbox, opcode = 2 is for RAPF (Retransmit After Page Fault handled) and other opcodes are used for acknowledgement messages or read requests.

void* handle_pgfault(void* x_void_ptr): This method describes the main mechanism for page fault handling. The first thing to do is to "open" the device (module) that is already generated before, which allows us to allocate and utilize a packetizer by returning its virtual address. This device generation required a module to be implemented and used, utilizing calls such as mmap, which takes as vma_off the protection domain identifier (PDID). Later, we initialize the source and destination information that is necessary for the header of the message to be sent through Netlink sockets. In our case, since we are now describing the userspace process point of view, the source is the current process (so the src_addr.nl_pid takes the pid of the process, using the getpid() method) and the destination is the kernel, which by default has

zero (0) as the value for pid, thus dest_addr.nl_pid is equal to zero. In our current mechanism, we do not want to send any information to the kernel (only the kernel/driver sends occasionally information to the userspace), but the functionality exists mostly for debugging purposes. After this, we are entering a while loop of the thread that busy waits until it receives a new message from the kernel, which will include the necessary information for the faulty transaction to be handled.

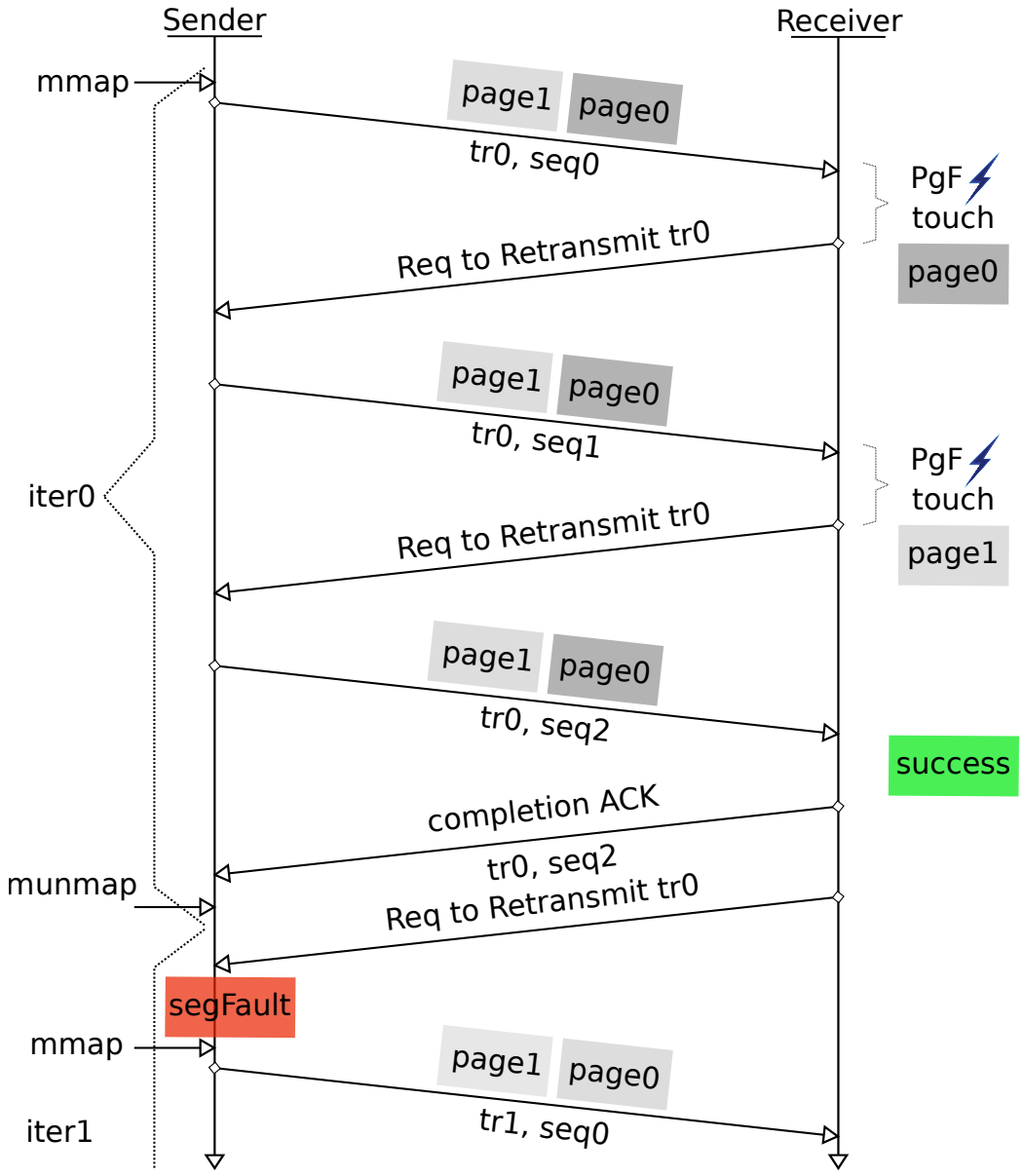

Figure 3.2 – Segmentation Fault scenario during Page Fault handling

When a message is received from kernel, we need to decode it, because originally it was sent as a string of combined information related to the transfer. This string is by design sent as a string of hexadecimal digits and by splitting the incoming message accordingly, we extract the useful information. In Table 3.1, we can see the format of the Netlink message as it is sent by kernel (driver) to the process of the related protection domain. Upon arrival of the message, the userspace process checks the least significant bit. If it is equal to 0, we had a page fault in the virtual address of the source (sender) buffer, otherwise (1) the page fault occurred in the destination (receiver) buffer.

At this point, the userspace process touches the faulty virtual address (IOVA), thus triggering an internal page fault that will be handled by the MMU of the CPU, resolving the fault transparently by doing all the necessary actions (mappings, page frames etc). After that, if the fault was triggered by the receiver (destination buffer), based on the type or R/W field (1 bit), we utilize the information we received from the kernel to acknowledge that the fault was resolved so the initiator node can retransmit it. In order to do this, we use the packetizer to send a message to the mailbox that the Real-Time (R5) co-processor polls. R5 uses the information of the transaction id (14 bits), the sequence number and the protection domain identifier (PDID) in order to safely trigger the now-resolved previously-faulty transfer. If the fault was caused by the sender (source buffer), we do not do anything else, because we are restricted by the design, but we expect that after the timeout period (initially it was 200 ms, since the parameter *TIMEOUT_PERIOD* was 200000000 in *r5_defines.h* file, but we changed it –currently the minimum we have used is 1 ms) for the transfer to be-re-transmitted.

Table 3.1 – Netlink Message format

Src_ID	**Tr_ID**	**Seq_Num**	**Faulty IOVA**	**PDID**	**R/W**
6 hex	4 hex	4 hex	8 hex	4 hex	1 hex
22 bits	14 bits	14 bits	32 bits	16 bits	1 bit
MSB					LSB

SMMU driver The mainline kernel driver of the ARM SMMU (version 2) includes a context fault handler, that will be triggered on occurrence of new translation (page) faults or permission faults, that might be triggered during a translation targeting an active and valid context bank of the SMMU (we have discussed about context banks in Section 1.3.1.4). This handler is called *arm_smmu_context_fault*. It is important to mention here that the SMMU reports in a different way the translation fault (no valid page frame for the given virtual-address) and the permission fault. While writing this thesis, we aim to support the translation fault, but we believe that permission fault can be handled by the same mechanism – in the future we expect to extend our tests further to tackle permission faults as well,

in order to safely say that they can be supported by our mechanism.

Of course, someone might be wondering what happens if multiple faults occur in one specific context back of SMMU "simultaneously", which includes transfers that request translation from a specific context back SMMU, while this context bank is still handling a previous context fault. To answer this question, we first need to describe what are the registers that can provide information about a context fault.

If a fault is encountered when all fields of SMMU_CBn_FSR, where n is the index number of a corresponding SMMU context bank ({0...15} in our system), are zero, the following registers provide full details of the fault:

- SMMU_CBn_FSR
- SMMU_CBn_FAR
- SMMU_CBn_FSYNRm

If a fault is encountered when the value of SMMU_CBn_FSR is non-zero, SMMU_CBn_FSR.MULTI is set to 1 and no details of the fault are recorded. In other words, we can keep information only on the first fault before it is cleared. SMMU_CBn_FSR.MULTI indicates that multiple outstanding faults occurred. As we read in Appendix A, MMU-500 (the model of our SMMU) supports either 8 or 16 parallel page table walks for a TBU. More details about this topic can be found there.

The Fault Status Register (FSR) of the context bank reveals the type of the fault. More specifically, the translation fault is revealed by the "TF" bit, that is the second bit (or bit 1, counting from zero) in FSR register (e.g. for context bank n, the name of the register is: SMMU_CBn_FSR) [18].

Both the "Fault Address Register" (SMMU_CBn_FAR) register and the "Fault Address Register - high significant bits" (SMMU_CBn_FAR_HIGH) register, hold the input address (i.e. virtual address) bits of the memory access that caused the translation fault. To be more precise, FAR holds the lower input address bits [31:0] and FAR_HIGH holds the upper input address bits [63:32], that triggered the page fault. In our case, FAR_HIGH holds only 16 bits, which in total with FAR register can give us: $32 + 16 = 40$ bits.

An SMMU handles a context fault, including a translation fault, by either stalling or terminating the transaction that caused the fault:

- Terminate the fault: SMMU does not perform the final access. Depending on the value of SMMU_CBn_SCTLR.CFIE, SMMU reports the fault to the initiator of the transaction that triggered a translation fault.
- Stall the fault: Software can either terminate or retry the faulty transaction, by writing to the register SMMU_CBn_RESUME. It is implementation defined whether SMMU supports the stall mode operation.

As the authors of ARM SMMU TRM [19] mention, it is not possible to guarantee that a stalled transaction in a context bank will not affect a transaction of another context bank. This is the reason this mode should be used wisely.

In our system we believe that the Stall fault model can be supported, because the corresponding field of all related registers (of all context banks) are read/write, meaning we can have this option enabled to any of them. Another hint is that the SMMU_SCR0.STALLD is zero (0), which allows SMMU to permit per-context stalling on context faults. However, it is not tested as much as we wanted due to lack of time. Hence, our tests followed the terminate fault mode. But we certainly expect to work and do experiments with Stall fault model in the future.

- If Stall mode:
  - If HUPCF==0 and a fault occurs: No more transactions are processed (=any subsequent transaction stalls) until the fault is resolved
  - If HUPCF==1 and a fault occurs: More transactions can be processed until the fault is resolved for this particular context bank

  According to the manual [19], the number of transactions processed after the original faulty transaction and the number of subsequent transactions that can raise a fault before no more transactions are processed until the fault in this particular context bank is cleared, is implementation defined.

  Another interesting thing with this mode is that if the SMMU is configured to raise an interrupt (which it is, as we mentioned above), one of the following can happen from the supervisory software:
  - Fix the fault and resume (retry)
  - Terminate the fault (no data are returned when read, and no data is affected when write)
- If Terminate mode:
  - If HUPCF==0 and a fault occurs: if a fault is active for that context, each subsequent transaction (whether it was faulty or not) terminates. FSR records only the original (active) fault.
  - If HUPCF==1 and a fault occurs: if a fault is active for that context, it terminates the new fault and records multiple faults in FSR.

The next thing we need to distinguish is whether the fault was due to a read or a write transaction, since our mechanism supports these cases in a different way due to technical limitations of the current system. In order to do this, we use the "Write Not Read" (WNR) bit of the "Fault Syndrome Register" (FSYNR) register of the context back, that experiences the translation fault. This register holds the fault syndrome information about the memory access that caused the fault. The WNR bit (bit 4) indicates whether the fault was part of a write or a read access,

which allows us to distinguish whether a fault happened in the source address or the destination address of the RDMA.

After this point, we pass the information of the protection domain to the input variable of the tasklet responsible for handling the page fault. In fact, we have implemented two (2) different tasklets, one to handle translation faults in source address and another one to handle the faults in destination address. As it is already mentioned, we have a different handling for each case (source and destination), thus we use two different tasklets. The protection domain is necessary, because it is the only way we can associate the faulty transfer of the context bank with the domain that will handle it. At this point, it is important to remind the readers that each context bank (or page table) is associated with one protection domain. More precisely, there is a one-on-one mapping of a process and a protection domain. Currently the team at FORTH works on having many processes that belong to the same protection domain. Although this is ongoing work, our implementation for the page fault handling takes this into account, which means that in the future it would be easy to adapt to a new system, that supports many processes per rank (or protection domain). As mentioned in Section 1.3.3.5, each tasklet will be scheduled and run when it is convenient (time-wise) for the system to run it, which will be after we have exited the interrupt handler.

Another thing we should mention about the driver is the configuration of the context bank. Basically, when we initialize a context bank of the SMMU in order to point to the page table of a process, we also set some settings, including how the faults will be handled for this specific context bank. In the *arm_smmu_init_context_bank* function in the ARM SMMU driver, we can set the Secure Control register (e.g. SMMU_CBn_SCTLR), that provides the top level control of the translation system for the related context bank. The default value of this register (Linux 4.9 version) had the following flags set (in most cases equal to 1) –the rest of the bits for this register were initialized to zero (0).:

- SCTLR_CFIE: Context Fault Interrupt Enable, if set, meaning that when a context fault occurs, an interrupt will be raised.
- SCTLR_CFRE: Context Fault Report Enable, allows the context bank to return an abort when a context fault occurs.
- SCTLR_AFE: Access Flag Enable, means that in translation table descriptors the AP[0] bit is an access flag.
- SCTLR_TRE: This bit indicates that the TEX Remap Enable is enabled, remapping the TEX[2:1] bits for use as two translation table bits, that can be managed by the operating system. As enabled by default and not having to do with what we were working on, we did not modify it.
- SCTLR_M: MMU (of CPU) behavior for this translation context bank is enabled. Basically, it means that the translation stage (1 or 2) that the context bank belongs to is enabled.

- CB_SCTLR_SHCFG_OUTER: Theses two (2) bits indicate the shareable attribute of a transaction where the translation context bank is disabled, which is when SCTLR_M=1. This is not true in our case –the default value is 2, which means the attribute is Outer Shareable.

In order for our mechanism to work when handling page faults, we had to check two (2) of the settings of this register, not enabled by default. The first was the SCTLR_HUPCF, which means Hit Under Previous Fault context fault. The second was about the SCTLR_CFCFG, which is the Context Fault Configuration and can have two values, based on the two modes discussed mainly above: 0 (zero) for Terminate mode and 1 (one) for Stall mode – Terminate mode is the default.

HUPCF setting allows us to process all subsequent transactions independently of any outstanding context fault. This is an interesting setting, since it was one of the reasons we would witness translation faults even in buffers (pages) that were resident in memory. For example, when we had a Remote Write transfer locally to the same node with the source buffer (pages) not being resident in memory, but with the destination pages residing in memory, since they were touched prior the RDMA transfer, we could detect page faults even in the destination pages, because first in time the source pages were "under a fault". In other words, translations of destination pages were subsequent translations of source pages, that were experiencing an existing fault, which is the reason why they were terminated in the end. By enabling this mechanism in our example, we could only see page faults that were indeed occurring in the source buffer.

3.2.2 Page Fault at Source address

There are two scenarios we can describe for the source buffer:

1. Remote write: In this case, it is expected that the source buffer will be initialized (or written) relatively soon prior to the RDMA transfer, which means the buffer (or pages) usually resides in memory. In other words, we do not expect to have any page faults in this buffer in the common case, except if in the meantime one or more pages have been swapped out to an external drive (major page fault) or the mapping was invalidated for any reason (such as THP optimization).

2. Remote read: In this case, it is possible that the source buffer is not populated prior to RDMA. At the same time, it will come as no surprise for a source buffer to-be written with an initial value first, just to make sure whether the data has been sent (read) correctly or not (e.g. when polling for a specific value).

Initially, translation faults in source buffer of the custom DMA were not supported. This means, that a when a page fault occurred, the transfer would be completed with no information (feedback) indicating that a fault was triggered and "garbage" data would be sent.

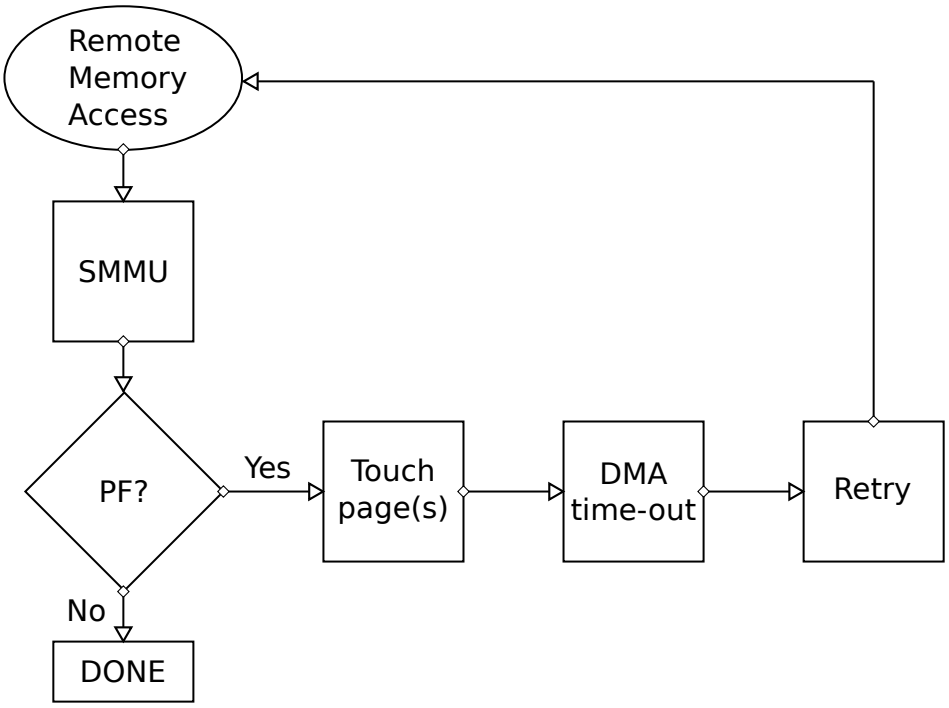

Figure 3.3 – Page Fault flow at source address

During the last one or two months of this thesis, modifications were made by the original designers of the FORTH PLDMA, in order to at least not complete the transfer in a case of a fault in the source buffer. This would give our page fault handling mechanism a time window to solve the page fault by bringing the pages to memory and then a time-out would trigger a re-transmission. Time-out retransmissions is a resiliency feature of the FORTH PLDMA implemented by others at FORTH. By the time a time-out occurs, we expect the pages that previously experienced a page fault to reside in memory –after that, the transfer will be completed.

The main modifications we had to do in order to handle this translation fault case, were in the driver of the SMMU (*arm-smmu.c*) and more specifically in the context fault handler, as described above.

3.2.2.1 Driver

The tasklet, called *pf_send_handler*, will only need to make sure that the pages are brought to main memory so after the time-out of R5 co-processor, the transfer will be completed.

The only thing necessary for this to work is to pass both the protection domain, the process index and the faulty virtual address to the tasklet, that as we know will run independently. We had to configure iommu_domain struct (iommu.h)

to hold information for both the protection domain and the process index in order to be able to associate a context bank with a protection domain and a process index. Reading both SMMU_CBn_FAR (Fault Address Register) and the SMMU_CBn_FAR_HIGH (Fault Address Register - high significant bits) while inside the handler and before calling the tasklet was sufficient. Then, we utilized some bits of the 64-bit unsigned integer input, that is given to the tasklet of the sender, in order for it to know what is the protection domain and process that would handle the page fault. Using the Netlink sockets approach, we send a Netlink message to the corresponding user-space process to request a touch of a specific 4KB page. Using the get_user_pages() approach, as an optimization, we request up to four (4) pages (the one that was faulty and the next three after it) and the kernel call will return the number of pages that it was allowed to bring to memory – it is possible that a next page does not belong to the user application, or in other words it was not mapped, and in that case we are not allowed to bring it to memory.

Currently, after scheduling the *pf_send_handler* tasklet, we also schedule the tasklet responsible for page faults in the destination buffer, just in case there is a new fault to-be-handled from that side, as well. This works as an optimization, thus it does not seem necessary.

3.2.3 Page Fault at Destination address

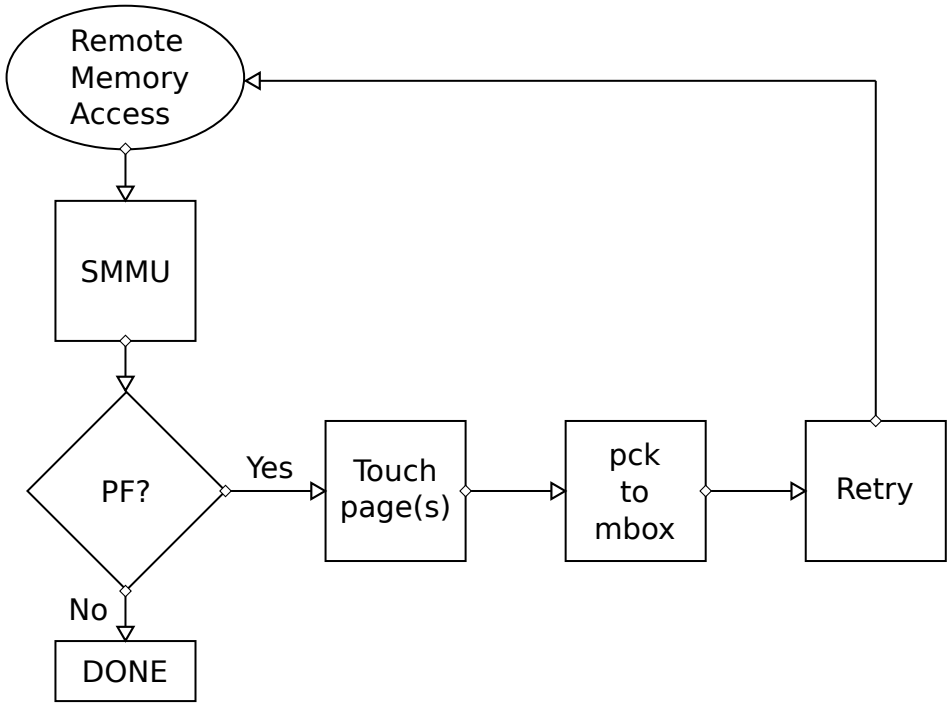

Figure 3.4 – Page Fault flow at destination address

The main mechanism includes modifications to the mainline kernel driver for the SMMU (ARM's IOMMU), a user-space library and additions to the hardware design and the firmware of the R5 processor. The common parts of the mechanism were described previously, so here we will describe the different characteristics of the receive (destination) path of the page fault handling mechanism.

3.2.3.1 Hardware

In order to serve all page faults caused when the receiver part of the custom DMA, described in Section 1.3.2, tries to access the memory, we implemented a FIFO, responsible to log all necessary information of the faulty transfers. We will use this information to solve the page fault and to also request from the sender to retransmit the previous faulty transfer –which mainly works as an optimization, because the system, as already mentioned before, has the functionality of a time-out, that would eventually trigger a re-transmission.

The FIFO originally was created to be read through an interface (AXI-lite) that when working in our Trenz experimental prototype allowed read requests of 32-bits. Later, when moving to new designs we could have read requests of 64-bits. So this is what our design has in mind, although the current interface in-use allows us to read up to 128-bits. The 64-bit mechanism includes a Finite State Machine (FSM), that makes sure that when a user has consumed/read the related data, only then it pops this information (FIFO entry). But since we need two (2) read requests to pop the entry from the FIFO, the FSM ensures that this happens in a safe order and way (e.g. if someone tries to read the second 64-bits of data, the entry will not be popped out from the FIFO). The depth of our FIFO is currently 512 and the width is 128 bits.

Our FIFO basically logs the information of all incoming packets that experienced a slave error (AXI NACK), which by the way could happen in different scenarios and not only in page faults. That is why we expect our mechanism to be used in the future by embedding solutions for other problems/type of faults as well, including invalid addresses. Currently, as an optimization, when a new negative acknowledgement (slave error) arrives in the receiver part of the FORTH PLDMA, we check it and if it has the same source node identification, transaction identification, sequence number and virtual page (excluding the page offset) with the entry we added (pushed) last time in our FIFO, we do not add it again. In the future, using get_user_pages(), it would be okay to check only the source node id, the transaction id and the sequence number, provided that pages arrive in order and when we request to get the user pages of a virtual address, the first time we add it to our FIFO, it will belong to the first page, in order for all pages to be touched.

The author of the design of this thesis had two (2) things in mind: first was to be able to serve 32-bit read requests if necessary (currently we do two (2) 64-bit read requests) and also to be convenient for the user and the developer to check these entries, while debugging the mechanism. In the future, we expect this FIFO

to be optimized space-wise, which was in our plans but due to lack of time it did not happen before submitting this thesis. In the Table 3.2 below we can see the format of each entry in our FIFO.

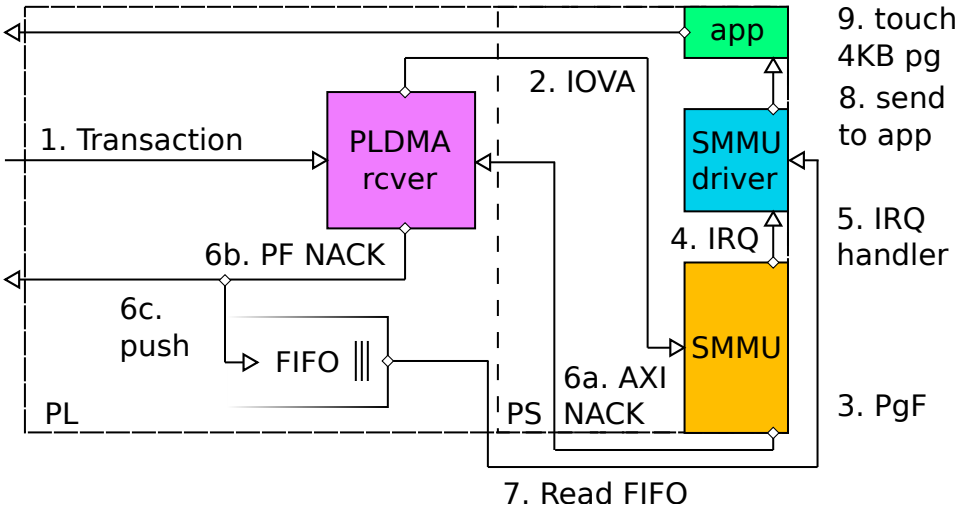

Figure 3.5 – The detailed flow of handling a Page Fault at the destination buffer using Netlink sockets

Table 3.2 – FIFO entry in Receiver side of custom PLDMA

00	**src_ID**	**00**	**tr_ID MSB**	**000**	**Valid**
2 bits	22 bits	2 bits	2 bits	3 bits	1 bit
6 hex		1 hex		1 hex	
tr_ID LSB	**00**	**seq_num**		**000**	**Valid**
12 bits	2 bits	14 bits		3 bits	1 bit
3 hex	4 hex			1 hex	
PDID		**Faulty IOVA MSB**	**0**	**EXA_ACK**	**Valid**
16 bits		12 bits	1 bit	2 bits	1 bit
4 hex		3 hex	1 hex		
Faulty IOVA LSB			**0000_0000_000**		**Valid**
20 bits			11 bits		1 bit
5 hex			3 hex		

3.2.3.2 Driver

When the interrupt handler, *arm_smmu_context_fault*, catches the related exception and distinguishes that it was due to a translation fault of a local write request to the memory, it schedules a *tasklet*, called *pf_rcv_tasklet*. This tasklet is responsible to do a couple of things, that will be described below:

- Read from the 128-bit FIFO added to the hardware part of the receiver all the related information to the transaction (block) that failed, such as:

  1. the source node ID (src_ID) - 22 bits: corresponds to the initiator computing node of the transfer
  2. transaction ID (tr_ID) - 14 bits
  3. sequence number (seq_num) - 14 bits
  4. protection domain ID (PDID) - 16 bits
  5. faulty virtual address (iova) - 32 bits: 4 most significant bits correspond to the process index in a rank/protection domain and the remaining 28 bits are the most significant bits of a 39-bit virtual address, which is currently the width of the address our system supports. The 12 less significant bits of the virtual address correspond to the page offset which is not needed in our mechanism, since we handle requests per page. 39-12 = 27 bits. The 28th bit is wired to zero.
  6. other kind of bits (such as Read/Write, Valid bits etc)

  This information is crucial and necessary in order to be able to successfully initiate a re-transmission of the faulty transfer – when the time comes and we know for sure that the page is already resident in memory. This seems rational when we do not rely on time-out re-transmissions, which cannot be effective when we cannot be really sure when exactly the page is paged in.

- Now that we know the information of the transaction that failed due to a page fault, we can initiate the handling mechanism - a mechanism responsible to page in the page that is not resident yet. As it is already mentioned, we mainly used Netlink sockets in order to transfer the information of the faulty transaction to the corresponding process. This process belongs to the same protection domain, that the faulty virtual address belongs to. Our initial approach causes an internal page fault, that will be handled by the Memory Management Unit (MMU) of the CPU. In simple words, we inform the corresponding process to touch the faulty page through the Netlink sockets, triggering an internal fault. The other approach, is using get_user_pages(), which allows us to be benefited by some optimizations as described in the page fault in the source buffer approach.

- After this step, we need to let the initiator node know that the page fault is handled, so the initiator node should re-transmit it. For the purposes of this sub-task, we utilize a packetizer that sends the necessary information of the transfer to the mailbox that the R5 polls, in order to initiate the re-transmission. We have mentioned this in Section 3.2.1, because currently this kind of re-transmission is only supported from user-space. In the future, we expect to provide such service from kernel-space, that will probably lead to less context-switches when we do all the work completely in the kernel (using get_user_pages()).

As already mentioned in Section 3.2.1, we also developed a module that, when inserted, it is detected as a device. When a user calls mmap() with the file descriptor of the new device, the call-back of mmap() allocates channel 0 out of 0..3 (4 channels in total) of the first available packetizer. Also, in the end, when the user no longer needs this mapping (e.g. exiting the program), the related call-back frees the previously allocated packetizer. We mention this module in this part of the document, since modules and drivers are somehow connected as terms (see Section 1.3.3.3).

3.2.3.3 Real-Time co-processor modifications

In order for this part (receive path) of the mechanism to work, we modified the firmware of Real-Time co-processor R5 of Zynq UltraScale+.

At this point, it is important to clarify that the development of the R5 framework was part of the work conducted by other colleagues of ours at FORTH and mostly as part of [8].

The modifications for this work were made in the r5_dma_controller.c source file that when built with other files results to the firmware file (.elf).

When receiving a Page Fault NACK and more specifically, any AXI Slave Error response (2'b10), instead of triggering an instant retransmission (if the transfer was valid), R5 pauses (for this transaction). In that case, there are only two (2) ways for this transfer to be re-transmitted:

1. Either by the expiration of the time-out period of the corresponding transaction

2. Or by receiving a specific message in its mailbox, that explicitly requests a retransmission for a transaction identification (trID), with a specific sequence number that might be valid and the protection domain

Explicit or on-demand requests to retransmit using the packetizer to send to the mailbox polled either way by the R5 co-processor works probably, as already mentioned before, as an optimization. Initially we did not know how much time it takes for a faulty, due to page faults, transfer to be resolved, meaning we could not decide what is the optimum time-out period for retransmission – something still uncertain, because it is not easy to generalize a solution based on our current environment, that behaves differently when a fault occurs in the sending or the receiving part of the PLDMA.

Below we see the part of the code that was added when R5 decodes the Page Fault Negative acknowledgement that is sent from the receiver part of the FORTH PLDMA. The previous (default) behavior was to instantly retransmit. Now we have modified this behavior and we expect a retransmission to be triggered either by a time-out or by a corresponding message that will be received in the mailbox polled by R5.

```
// [...]
// PF Negative ACK: Message for R5 to NOT retransmit, which was the default behavior
else if(((mbox_value >> 16) & 0x07) == 1){
  transaction_id = ((mbox_value >> 2) & 0x03FFF);
    #if PF_DEBUG
      xil_printf("PAGE FAULT NACK for trans_id %d with errorcode %d @ set\n\r",transaction_id,(mbox_value >> 16));
    #endif
  return;
}
else{
// [...]
}
// [...]
```

The code below, decodes the message that was received in R5's mailbox. If this message has as opcode the value 2, it means that is a "Page Fault: Ready to retransmit" message.

It is important to acknowledge here that the first (least significant) twelve (12) bits of this message after the two (2) bits of the opcode are wired by the packetizer (kernel-space) and not by the user who sent the message. This way we can avoid malicious users who may pretend they legitimately request an access to the local memory of a protection domain that does not exist. We need to elaborate further on this. One approach would be to check the wired protection domain and the one sent/given by the user to see if they match.

We expect that when moving all the mechanism to kernel space, we will not have to deal with such issues.

```
// [...]
// Request-to-retransmit sent from packetizer as part of page fault mechanism (receiver side)
case 2:
    #if PF_DEBUG
        xil_printf("NEW PF msg received! \n\r");
    #endif

    word0 = mbox_value; // consume first 32 of 64 bits from msg received
    wired_opcode = word0 & 0x03; // wired opcode (2 bits -- if 2, means PF msg)
    wired_pdid = ((word0 >> 2) & 0x0FFFF); // wired protection domain ID (4x4 = 16 bits)
    transaction_id = ((word0 >> (2+16)) & 0x3FFF); // received transaction id (2+3x4 = 14 bits)

    word1 = *(ack_mbox_b); // consume second 32 of 64 bits from msg received
    seq_num = (word1 & 0XFFF); // received sequence number (3x4 = 12 bits)
    rcved_pdid = ((word1 >> 12) & 0x0FFFF); // received protection domain ID (4x4 = 16 bits)
```

```
16
17      #if PF_DEBUG
18          xil_printf("TID is: %d, SEQ_NUM is: %d\n\r", transaction_id, seq_num);
19          xil_printf("Wired PDID is: %d, received PDID is: %d\n\r", wired_pdid,
            rcved_pdid);
20      #endif
21
22      // Message with out-dated seq. number was received -- it will be ignored
23      if(seq_num != pending_transactions[transaction_id].seq_num){
24      #if PF_DEBUG
25        xil_printf("PF msg: expected seq was: %d -----BUT------  seq received was:
          %d and tid %d
          \n\r",pending_transactions[transaction_id].seq_num,seq_num,transaction_id);
26      #endif
27        return;
28      }
29
30      // call timeout_drop_transaction function in order to drop out the transaction
        that received NACKED
31      timeout_drop_transaction(&timeout,transaction_id);
32      // find the virtual channel of transfer
33      int virtual_channel_num = pending_transactions[ transaction_id ].transfer_id;
34      // retransmit this transaction and send completion notification again if this
        is the last block
35      transaction_retransmission(virtual_channel_num, transaction_id);
36
37      return;
38  }
39  // [...]
```

Chapter 4

Evaluation

In order to evaluate our work, we conducted some experiments to measure mostly the latency of RDMA transfers when using our mechanism.

Ideally, we would like to have results from measurements on real applications but due to lack of time, we can only evaluate our mechanism based on custom-made micro-benchmarks. In other words, we are able to only run an application that triggers RDMA transfers using the FORTH PLDMA, that experience minor page faults. We expect our mechanism to also work with major page faults.

When performing latency measurements, it is common to exclude the overhead of pinning or touching the pages before they are DMA'ed. As a result, we do not expect our approach to have better measurements compared to these cases, because our mechanism causes pages to be paged in on-demand, which adds an extra overhead during the transfer.

On the other hand, we propose that page faults are not common and this type of overhead will not be paid often; page faults are considered a rare phenomenon.

Another measurement we would like to conduct, that was not possible due to lack of time, was the memory utilization. Our mechanism ensures better utilization of the physical memory, because in the common case we will have pages that belong to a process (application) reside in memory only when they have to be there (i.e. they are actually used).

During the time we had on our hands to evaluate our work, we were interested in measuring:

1. The latency / execution time of an RDMA transfer (Remote Write) with:
   - our mechanism enabled
   - pinned buffers prior to the RDMA transfer
   - touched buffers prior to the RDMA transfer
2. The cost (time) of pinning pages
3. The cost (time) of unpinning pages

4. The cost (time) of touch
5. The latency / execution time of the page fault handler (driver)

In order to perform measurements in user-space, we made use of the method named "clock_gettime()", defined in the Linux header file *time.h*. The resolution of the result from this method is in nanoseconds. Although this method is not a system call (it is a VDSO, which in theory is used to minimize the overheads of system calls), it might add a delay - as we will see later in our results. In kernel-space, we made use of the call/method named "ktime_to_ns", which also produces a result in nanoseconds resolution.

We have conducted our measurements in one FPGA of a QFDB (intra-QFDB), but we do not expect significant differences when moving to more hops (FPGAs) or even other computing nodes (QFDBs) – the hop-to-hop latency is about 100 nanoseconds.

In our experiments the transfer sizes are: 16 Bytes, 64 Bytes, 256 Bytes, 1024 Bytes, 4K Bytes, 16K Bytes, 32K Bytes and 64K Bytes. We consider 64K Bytes as the maximum transfer size in our experiments, because in Linux kernel, as mentioned in [13], the upper limit of Bytes that a user can lock in memory (using the "mlock" method), is 64K Bytes.

Because of this upper limit in our PLDMA transfers, it would not make any difference whether we have Transparent Huge Pages (THP) mechanism (see Section 3.1.2.3) enabled or disabled – since we do not work with Huge Pages in this set of experiments, we do not expect any THP to occur. Moreover, we hope that our mechanism will be used by the main ExaNeSt prototype soon, which is when we expect the system to be able to recover from translation faults caused during THP activity.

Table 4.1 – Overhead (time in µsec) of mmap, munmap, pin, unpin and touch

	16 B	**64 B**	**256 B**	**1 KB**	**4 KB**	**16 KB**	**32 KB**	**64 KB**
mmap	2	2	2	2	2	2	2	2
munmap	6	6	6	6	7	10	12	19
pin	6	6	6	6	6	15	27	49
unpin	2	2	2	2	2	5	8	14
touch	3	3	3	3	3	10	19	40

Table 4.1 describes the overhead of each different method for one buffer (e.g. the source address of a DMA transfer) –in our measurements someone would have to consider each overhead twice (two buffers: source and destination). The way we measured the overhead is described in Listing 2, where instead of the call appeared as "PLDMA_transfer" we had the call of the corresponding method (e.g. "mmap"). This means that an overhead for "get_time()" is probably included in our measurements. Initially, our RDMA measurements for page faults included the overheads of these methods (Table 4.1), which is the reason we evaluated them.

In the case of the "pin", we also expect a call of "unpin" to follow for two reasons:

- This is the only way for the "pin" system call to be effective in each next iteration of the loop, when using the same address (page). Also, the approach of pinning/unpinning of the buffers before/after they are DMAed is used by many [13].
- It is expected that when someone pins a page, they will eventually have to unpin it (e.g. at the end of the PLDMA transfer), in order to make more space available for the current or other users (memory utilization).

While running experiments that were triggering page faults, we faced many challenges because the fault path of the custom FORTH PLDMA was not stressed enough until that moment. This is the main reason that all measurements related to page faults were a result of only 500 iterations for each transfer size.

Listing 1 Pseudo-code for "ideal" measurements

```
start = get_time();
for(i=0; i<iterations; i++){
    PLDMA_transfer();
}
end = get_time();
time = (end-start)/iterations;
```

Listing 2 Pseudo-code for "real" measurements

```
for(i=0; i<iterations; i++){
    start = get_time();
    PLDMA_transfer();
    end = get_time();
    time += (end-start);
}
avg_time = time/iterations;
```

The round-trip latency of a remote DMA write transfer that experiences zero (0) page faults during the RDMA is low, i.e. 4 µsec for 16 Bytes, when using the "Ideal" (Listing 1) type of measurements.

In the "Ideal" type of measurements we do all the memory mapping/unmapping work before/after the RDMA transfer. This measurement includes only the method, that triggers the transfer and the method that is polling the status register of the PLDMA, that signals the completion of a transfer. For this experiment, we conducted 10000 iterations for each transfer size and we calculated the average.

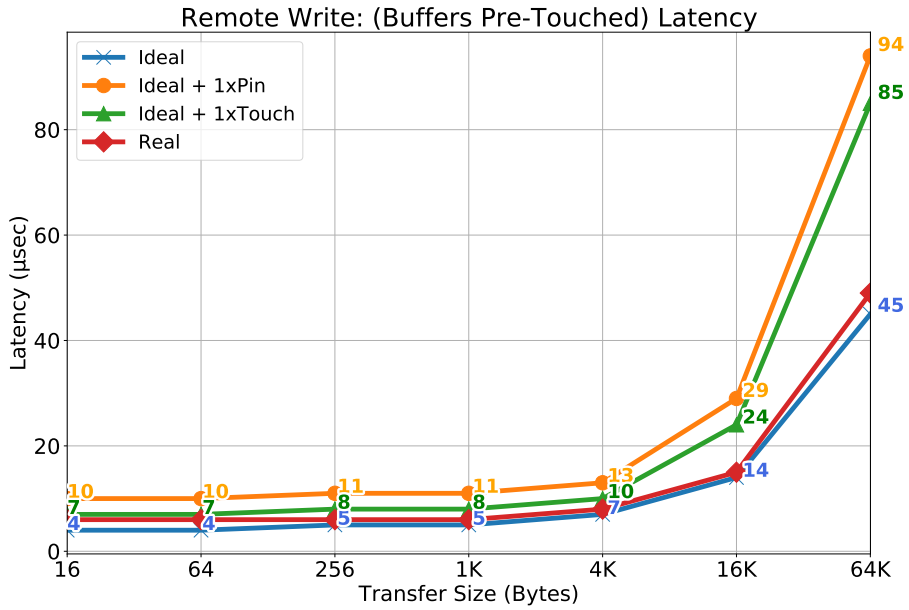

Figure 4.1 – Remote Write Transfer (One FPGA), All buffers pre-touched, Transfer-Only Latency

We use the name "Ideal", because we expect no overhead from any kind of system call during the measurement.

Initially our case study involved measurements that used a "for" loop, which included some system calls such as memory mapping/unmapping of buffers, mainly for two (2) reasons:

1. In order for a page fault to be triggered in each iteration, a call that unmaps the buffer from the address space was necessary

2. The measurement of the latency (time) was outside of the loop (as in Listing 1)

Another reason behind this experiment choice is that we wanted to evaluate transfers that cause page faults and compare the results to the measurements of transfers that do not cause page faults during the RDMA, because they were either pinned or touched prior to the DMA transfer. Since touching pages is part of our page fault handling mechanism, it would only make sense to compare it with a mechanism that basically pre-faults the pages prior to the RDMA transfer.

Although this approach seemed rational we thought that the results would probably cause confusion to the readers due to the cost/overhead of the other methods and system calls involved, so we decided to follow the methodology that is depicted in Listing 2 for the measurements that involve page faults. In other words,

in measurements where we evaluate the page fault mechanism we compare against the no-page fault during RDMA evaluated using the "Real" type of measurements.

In Figure 4.1, we can see the measurements conducted using the average of the execution time of the transfer, that experiences no page faults and thus no handling from our mechanism is involved. In order to achieve this, we touch one (1) Byte for each page (4K Bytes) prior to the DMA transfer.

In the same Figure (Fig. 4.1) we can see that the "Ideal" latency increases when we add the overhead of one "pin" or "touch" operation of a buffer. We also see a line that describes the results of the "Real" type of measurements, as described above. Due to lack of space we could not show the value of "Real", but it can be found in the next Figures, whenever we show results of no involvement of our mechanism (no page faults).

Since we did not want our process under-test to be preempted by other tasks in a random way that could affect our measurements, we made sure that all interrupts of the system were moved to the CPU 0 and we conducted our experiments on CPU 2. In order to do this, a script was written that would overwrite the smp_affinity value of each IRQ in /proc/irq/. Of course this cannot guarantee that all IRQs will be moved to CPU 0, but it seemed sufficient to move the majority of them. For example, the interrupt with the name "arch_timer" could not be moved, but it appears to be a timer necessary for the system and each CPU, which is the reason it runs periodically on each one of them. At the same time, we were calling "taskset 0x4", before running the executable of our application, in order for it to run on CPU 2 –0x4, or 0100 in binary, is a mask that indicates the CPUs that an executable will run. This would allow us to get as much reliable as possible measurements for our mechanism. Although using this approach allowed us to eliminate the outliers, we noticed that it did not affect significantly the average values of our measurements since page faults are costly in any case. We also noticed that sometimes the process would fall asleep and never wake up, which means it requires further investigation before actually using the move of IRQs.

In the next Figures we evaluate the latency of a Remote DMA Write inside one MPSoC (FPGA) of one QFDB (node). We have three (3) different types of bars. The first bar (blue color), indicates the latency of a Remote DMA Write where no page fault occurs thus no invocation of our page fault handling mechanism, since the buffers were pre-touched. The second bar (orange color) indicates the latency of a Remote DMA Write, which experiences a page fault in the source/destination address and the paging-in mechanism used is touch-ing one page ("Touch-A-Page") per invocation of the handler (tasklet), by sending the corresponding information to the user-space and the related process through Netlink sockets. The third bar (green color) indicates the latency of a Remote DMA Write, which experiences a page fault in the source/destination address and the paging-in mechanism used is touch-ing "ahead" ("Touch-Ahead"), which means paging-in up to four (4) pages per invocation of the handler (tasklet), using the get_user_pages method. As already mentioned before, the reason we chose to "touch" up to four (4) pages is that each transaction (or block) is at most four (4) 4KB pages and we expect that

if a page fault is caused on the first page, it is very likely that the same will happen to the next three (3) pages (locality).

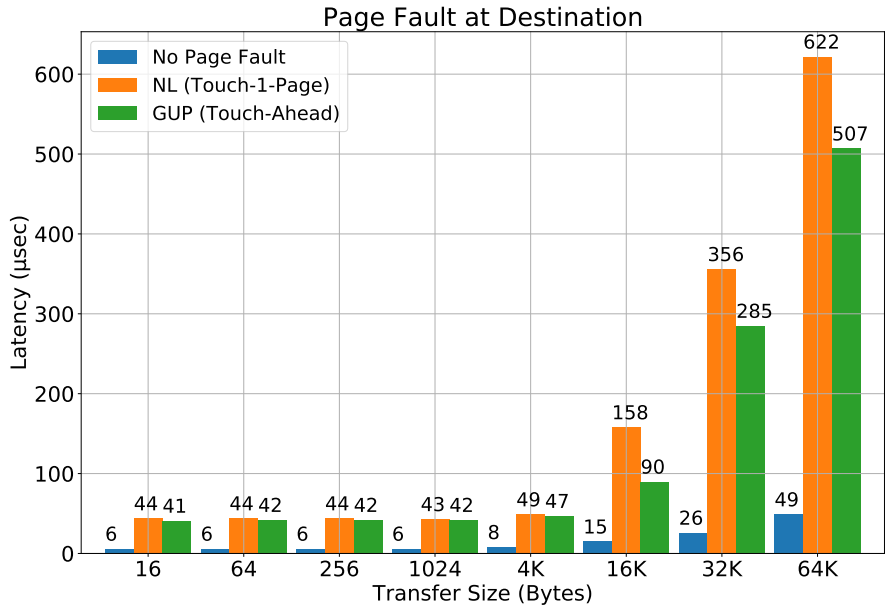

Figure 4.2 – Remote Write Transfers (One FPGA) with Page Fault at Destination - Latency

In Figure 4.2 we see the latency of a Remote DMA Write in one MPSoC (FPGA), where a (minor) page fault occurs in the destination address during the Remote DMA Write transfer. As expected, the results seem similar up to 4KB, which is the size of one page – the handling up to the size of one page is the same cost-wise. For the sizes of 16 KB, 32 KB and 64 KB we see a benefit of 1.7x, 1.2x and 1.2x when using "Touch-Ahead" instead of "Touch-A-Page". We believe that the decrease of benefit in 32 KB and 64 KB is due to the interleaving effect in FIFO: we noticed that we might have duplicate packets enqueued when the transfer size is more than 32 KB (two transactions or in other words, blocks). Because of this interleaving effect, it takes more time to find a new page / set of pages to page-in during the handling. As an optimization, we keep track of the last two (2) entries (transactions) that triggered a page fault, on the driver, and we do not repeat the handling for a page we have already handled. By design, only two (2) different blocks might come interleaved, so checking only the last two (2) entries of one source node, seems sufficient. In the future we expect to add this check on hardware, in order to eliminate the duplicates that can appear in our current design of the FIFO.

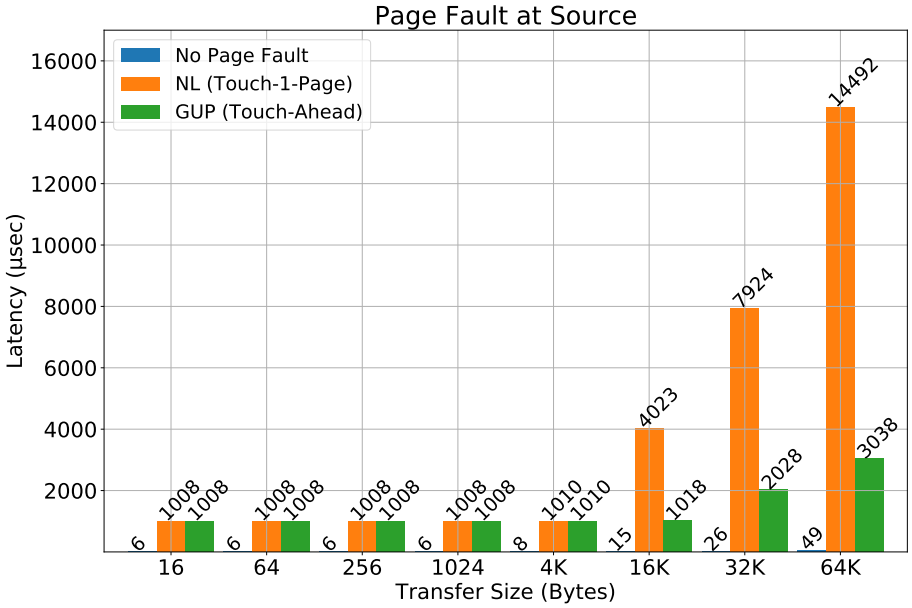

Figure 4.3 – Remote Write Transfers (One FPGA) with Page Fault at Source - Latency

We tried three (3) different time-out periods: 25 ms, 2.5 ms and 1 ms. As expected our best scenario was with time-out period set to 1 ms, which is reasonable due to the fact that we can complete the transfers that encountered a page fault sooner. At this point, it is important to remind to the readers that our mechanism expects time-out retransmissions when a page fault is encountered in the source addresses. This means that in all cases we expect the corresponding latency to be dominated by the time-out delay, which currently is 1 ms.

In Figure 4.3, we see a similar behavior for transfer sizes up to 4 KB, that we saw in Figure 4.2. For transfer sizes of 16 KB, 32 KB and 64 KB we see a benefit of 3.9x, 3.9x and 4.7x when using "Touch-Ahead" instead of "Touch-A-Page". 3.9x seems a rational benefit since "Touch-Ahead" pre-touches up to four (4) pages per transaction (block). It is uncertain why the 64 KB transfer size results to a better benefit (4.7x), but we believe it might be an effect due to the interleaving and the FIFO effect, already mentioned before. More investigation on this is expected in the future.

In Figure 4.4, we see a similar behavior for transfer sizes up to 4 KB, that we saw in Figure 4.2. It is obvious that transfers with smaller transfer sizes are dominated by the time-out in the source buffer. For the sizes of 32 KB and 64 KB we see a benefit of 1.5x and 1.4x when using "Touch-Ahead" instead of "Touch-A-Page". For the 16 KB transfer size the benefit when using "Touch-Ahead"

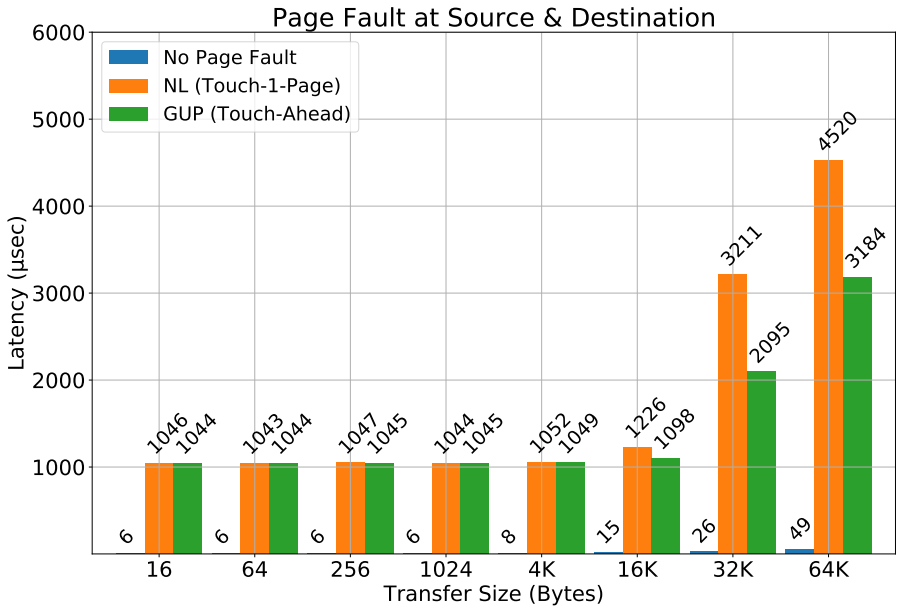

Figure 4.4 – Remote Write Transfers (One FPGA) with Page Fault at Source and Destination - Latency

is way smaller and we believe that is due to the fact that even when we utilize "Touch-A-Page", the mechanism manages to touch all pages (4) before the time-out retransmission. This can explain why "Touch-Ahead" and "Touch-A-Page" have similar results.

In Figure 4.5 it is noticeable that when a page fault occurs both in the source and destination buffers, the overall latency is significantly less than when we handle a (minor) page fault at the source, when moving to bigger transfer sizes. In Figure 4.6, we argue that this can probably be explained due to the less time-outs due to page faults at source buffer, which are considered the most costly ones in our mechanism. While a page fault at the source address is not resolved yet, requests for the faulty destination address begin to arrive and our mechanism can request explicit re-transmissions sooner than the time-out. In other words, when we have page faults only from the side of the source address and while using no optimizations ("Touch-A-Page" and not "Touch-Ahead"), we expect as many time-outs as the number of pages, since the handling takes place per page.

Another type of measurement we conducted in order to evaluate our mechanism was from the side of the kernel -how much time the page fault handling spends in kernel space, since a part of our mechanism is based on the driver of ARM's IOMMU (SMMU).

First, we calculate the number of invocations of SMMU's context bank fault

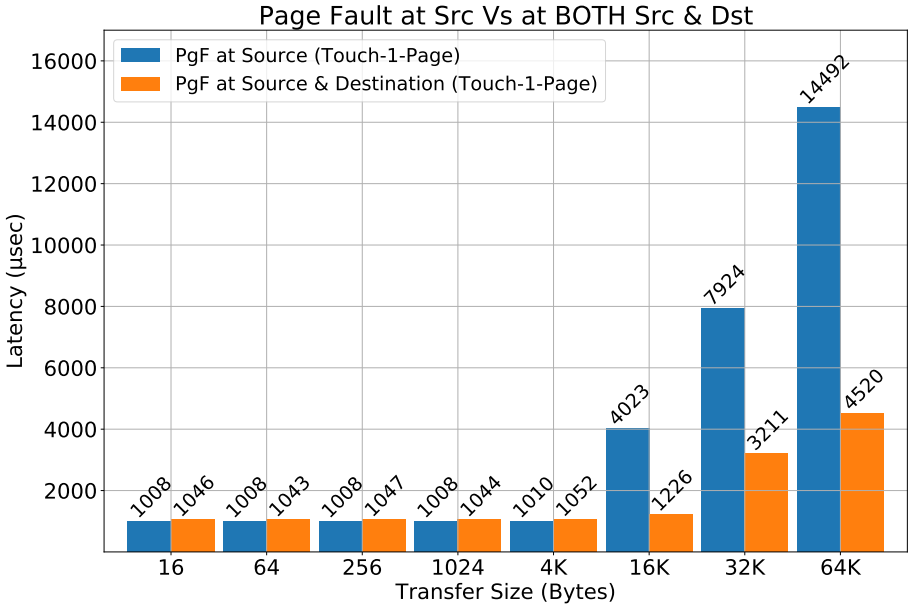

Figure 4.5 – Remote Write Transfers (One FPGA) with Page Fault at Source Vs Source and Destination - Latency

handler due to translation faults, for different transfer sizes. Then, as already mentioned before, judging from the type of fault (read/write), we schedule accordingly a tasklet, that of course will run when it is possible for the system, offloading this burden from the interrupt handler and allowing it to exit sooner. So, apart from measuring the time inside the interrupt handler, we also evaluate the time spent in the corresponding tasklet - excluding the time it took the tasklet to actually run. We are currently not sure if it is reasonable for someone to calculate it, since it will probably depend on the load a system has each time.

In the kernel driver measurements that we present in Figure 4.7, we show numbers only with 1 ms time-out, but having measured also scenarios with time-out as 25ms and 2.5ms, we know that even in this type of measurements, 1 ms time-out outperforms the other two scenarios.

In Figure 4.7 we see the driver latency of each approach. In order to measure accurately the latency, we conducted a "cold run" (e.g. cold cashes) prior to our "for" loop experiment, that we excluded from our results. To be more precise, the "cold run" was an 8-byte transfer that we knew in advance that will invoke the translation fault handler only once in case the page fault was caused in the source or the destination buffer. When a page fault occurs in both buffers, we expect two (2) invocations of the translation fault handler, so in these scenarios we excluded the first two (2) "cold run" measurements.

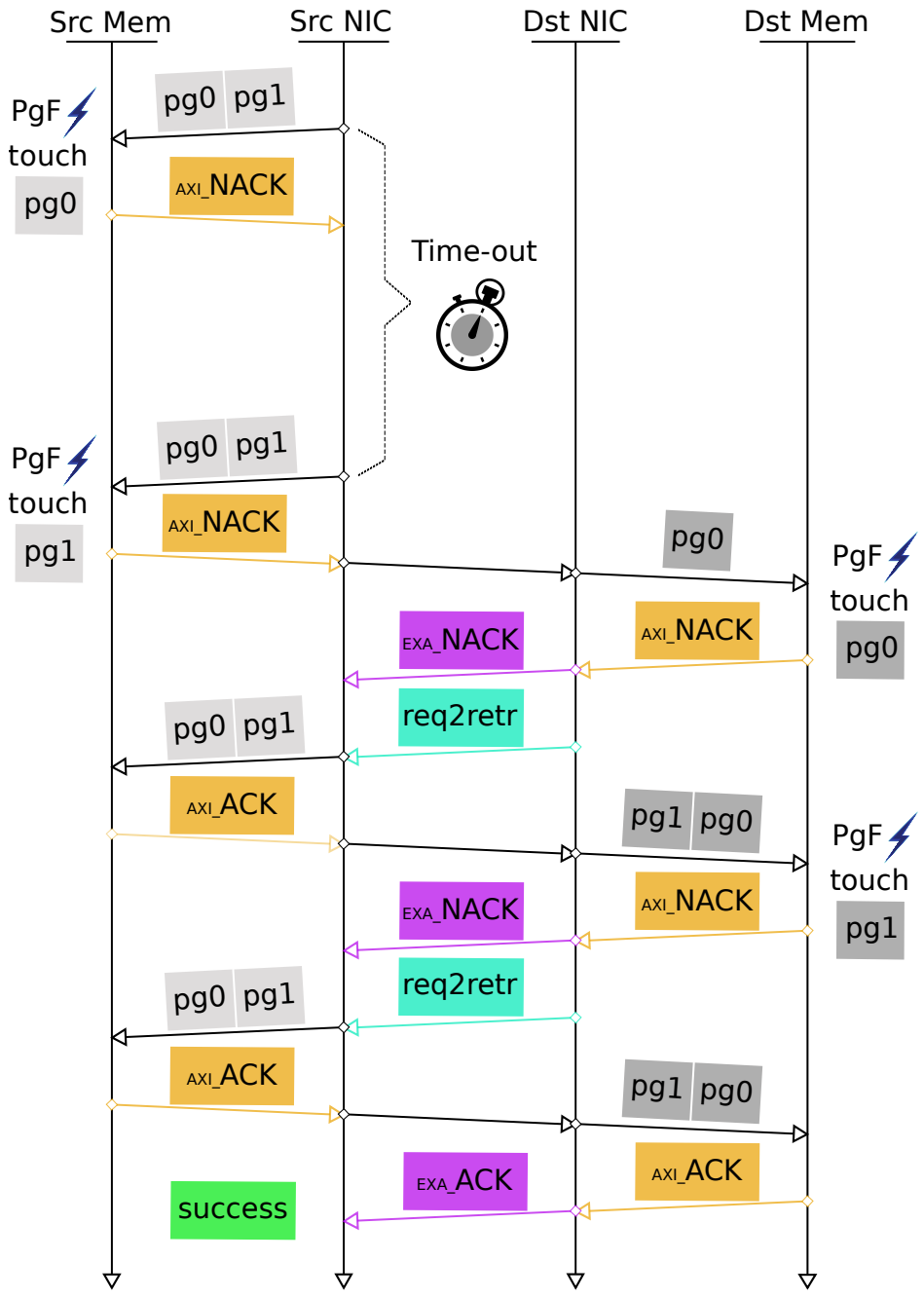

Figure 4.6 – Page Fault at source and destination addresses simultaneously leads to less time-outs than Page Fault only at source address. In this example, we have one (1) transaction (tr0), with page 0 and 1 (light grey) constituting the source address (buffer) and page 0 and 1 (grey) constituting the destination address (buffer).

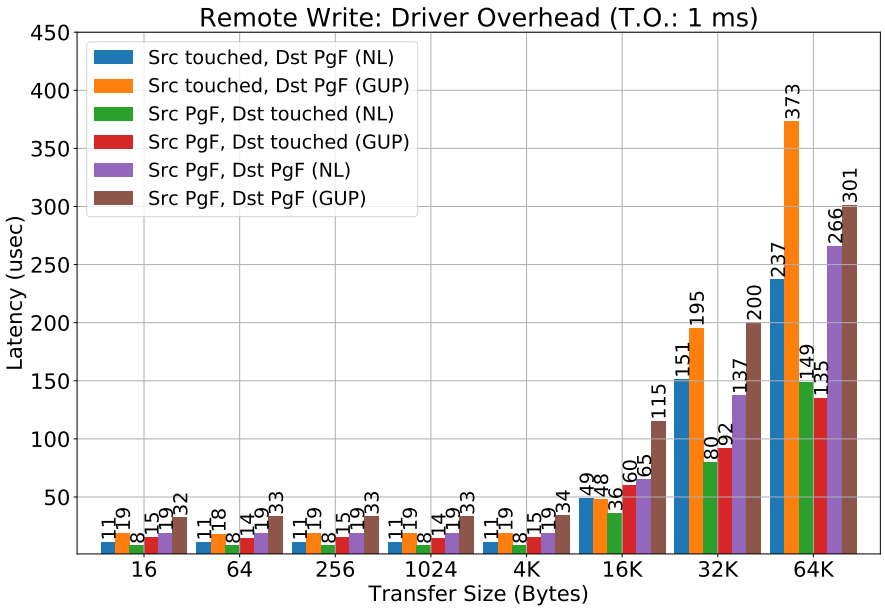

Figure 4.7 – Remote Write: Driver Latency

There are some insights we can extract from Figure 4.7. It is evident that "get_user_pages" (GUP) or "Touching-Ahead" costs more time in driver, because the burden of "touching" the pages is moved to it. In fact, our mechanism tries to touch up to four (4) pages each time when using GUP, as long as it is possible and permitted (e.g. it might be the case that a user application does not own more than one pages). This is a speculative approach, that comes with some costs that can be seen in Figure 4.7.

One thing we find interesting is that in 64 KB when a page fault occurs in the source buffer, the Touch1Page through Netlink seems to be more expensive than the Touch-Ahead through "get_user_pages" (GUP), which requires further investigation and possible re-run of our experiments in order to elaborate on the reasons behind this. Due to the prototype and the on-going work on it, there are many factors that could affect this measurement, including unexpected time-outs or timing problems.

Chapter 5

Conclusion

5.1 Summary

As part of the ExaNeSt project, we worked on a mechanism that provides support on page faults that might be triggered during remote direct memory accesses (RDMAs).

This mechanism supports translation faults in SMMU, that are expected to occur under many circumstances, including when having optimization mechanisms enabled like the Transparent Huge Pages (THP). Another reason to provide this kind of mechanism is to overcome the consequences of the alternatives such as pinning/unpinning or touch of the buffers before/after they are DMA'ed, which do not not seem good and viable solutions memory-utilization-wise in the long run.

In the end, we evaluated our work in terms of latency overhead and also measured the cost of the alternatives.

5.2 Future Work

There are many areas related to our work that, due to lack of time, we could not cover in this work. Some of the areas that we would like to investigate in the future are listed below:

1. Extend our mechanism to fully support get_user_pages() and packetizer-to-mailbox exchange of messages in kernel (full kernel implementation).

2. Finalize and evaluate the handling of page faults during Remote Read RDMA requests.

3. Build a smarter mechanism of logging information of faulty transfers (considering the interleaved packets), including a NACKer, that would offload the network from the multiple NACKs per faulty packet.

4. Add support for other types of faults such as permission faults (as mentioned before, more testing is required) and possible segmentation faults from invalid addresses.

5. Evaluate our mechanism on major page faults that require I/O (e.g. disk access).

6. Perform measurements on memory utilization, to see what is the fraction of the benefit we gain with our mechanism.

7. Elaborate more on the SMMU setting of stalling the faults and try to resume them from the driver after the handling.

Bibliography

[1] Manolis Katevenis, Nikolaos Chrysos, Manolis Marazakis, Iakovos Mavroidis, Fabien Chaix, Nikolaos Kallimanis, et al. The ExaNeSt Project: Interconnects, Storage, and Packaging for Exascale Systems. In *2016 Euromicro Conference on Digital System Design, DSD 2016, Limassol, Cyprus, August 31 - September 2, 2016*, pages 60–67, 2016.

[2] Xilinx. *Zynq UltraScale+ MPSoC Technical Reference Manual UG1085*, 1.3 edition, October 2016.

[3] Evangelos P. Markatos and Manolis Katevenis. User-level DMA without operating system kernel modification. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA '97), San Antonio, Texas, USA, February 1-5, 1997*, pages 322–331, 1997.

[4] Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization: Issues, security threats, and solutions, 2013.

[5] Christian Bell and Dan Bonachea. A new DMA registration strategy for pinning-based high performance networks. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, page 198, 2003.

[6] *TE0808 TRM.*

[7] *AXI Reference Guide*, 13.11 edition.

[8] Leandros Tzanakis-Arnaoutakis. Quality of Service Framework for Low Power RDMA Operations over Cortex R5 Real Time Microcontroller. M.Sc. Thesis, Computer Science Department, University of Crete (unpublished), 2019.

[9] Pantelis Xirouchakis. Design and Implementation of the Send Part of an Advanced RDMA Engine. M.Sc. Thesis, Computer Science Department, University of Crete (unpublished), 2019.

[10] Antonis Psistakis. IOMMU Support for Virtual-Address Remote DMA in an ARMv8 environment. B.Sc. Thesis, Computer Science Department, University of Crete (unpublished), 2017.

[11] Antonis Psistakis, Panagiotis Peristerakis, Pantelis Xirouchakis, Michalis Gianioudis, Giorgos Kalokairinos, Nikolaos Chrysos, Fabien Chaix, Vassilis Papaefstathiou, and Manolis Katevenis. User-level RDMA with IOMMU support on ARM platforms. Fourteenth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES) 8-14 July 2018, Fiuggi, Italy, 2018.

[12] Zhiyi Huang. Lab 6 - (COSC440) Advanced Operating Systems course, Department of Computer Science, University of Otago, 2019.

[13] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page fault support for network controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 449–466, 2017.

[14] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel - from I/O ports to process management: covers version 2.6 (3. ed.)*. O'Reilly, 2005.

[15] Pei Cao. Lecture 17 - Page Faults: putting it together, (CS537) Introduction to Operating Systems, University of Wisconsin, 1998.

[16] Jonathan Corbet. Transparent huge page reference counting, 2014.

[17] RedHat William Cohen. Examining Huge Pages or Transparent Huge Pages performance, 2014.

[18] Xilinx. Zynq UltraScale+ Devices Register Reference, 2019.

[19] ARM. ARM System Memory Management Unit Architecture Specification SMMU architecture version 2.0, 2019.

Appendix A

StreamID explained

System Memory Management Unit (SMMU) implements a TBU (Translation Buffer Unit) for sets of masters (devices) on the PS port. The Master ID for each AXI master and each port of the PS for the PL are listed in a Table titled "Master IDs List" in Chapter 16 on the Zynq MPSoC Technical Reference Manual [2].

Each PS port to the PL has a fixed master ID that is used together with the TBU number and an AXI ID to construct the StreamID. The AXI protocol includes two kinds of AXI transaction identifiers, AXI IDs and Master IDs. The Master ID is used to uniquely identify the master that initiated a transaction. AXI IDs are used to identify separate transactions that must be processed in order, for example when having multiple contexts or threads within a single master.

Table A.1 – StreamID format (15 bits)

TBU Number	Master ID	AXI ID
[14:10]	[9:6]	[5:0]
5 bits	4 bits	6 bits

Table A.2 – PS port connections to TBUs

TBU Number	PS port
0	S_AXI_HPC[0]_FPD
0	S_AXI_HPC[1]_FPD
1	
2	
3	S_AXI_HP[0]_FPD
4	S_AXI_HP[1]_FPD
5	S_AXI_HP[2]_FPD
6	S_AXI_HP[3]_FPD

In Table A.2 we can see where each port of the Processing System connects to

the TBU (essentially to SMMU). This information is extracted from the top-level diagram of the Zynq Ultrascale Plus MPSoC (Figure 1.3).

A.1 TBU (Translation Buffer Unit)

The Translation Buffer Unit (TBU) contains a Translation Look-Aside buffer (TLB) that caches page tables maintained by the Translation Control Unit (TCU). The SMMU implements a TBU for system masters. Each TBU has the following characteristics:

- designed to be local to the master
- 256 outstanding transactions for each TBU
- 16-deep TBU queue support
- 32-deep write buffer support (for each TBU) : 0, 4, 8 or 16 bursts
- Best-case hit latency
  - 2 clocks (when TBU addr. slave register slices are NOT implemented)
  - 3 clocks (when TBU add. slave register slices are implemented)
- Micro-TLB
  - caches PTW (page table walk) results returned by the TCU
  - fully associative
  - configurable depth (based on our requirements)

A.1.1 Outstanding transactions per TBU

Outstanding transactions are defined as transactions for which, the physical address access is generated and accepted by the slave, or write/read responses are stalled.

For every TBU, the MMU-500 supports 256 outstanding transactions each for write and read accesses. The MMU-500 generates a PTW when accesses from the master result in a TLB miss. However, based on the configuration, the MMU-500 supports either 8 or 16 such parallel PTWs for a TBU. If more than 8 or 16 PTWs are pending, a TLB miss on a channel indicates that the MMU-500 cannot accept additional transactions on the write or read channels.

A.2 TCU (Translation Control Unit)

This unit is responsible to control and manage the address translations (translation tables for the TBUs). It uses a private AXI stream interface to update the

translation tables in the TBUs. TCU core can run at half the clock speed of TCU external interfaces.

It consists of four (4) main components:

1. Macro-TLB: responsible to cache PTW results in the TCU
2. PTW cache: MMU-500 caches partial PTWs to reduce the number of PTWs on a TLB miss. It caches both stage 1 and stage 2 level 2 PTWs.
3. Prefetch buffer: MMU-500 fetches in advance 4KB and 64KB sized pages into the prefetch buffer, which reduces the latency for future PTWs. It has configurable width. It is a single four-way set associative cache, that can be enabled/disabled depending on the context. It also shares RAMs with the TLB cache.
4. IPA-to-PA cache: MMU-500 implements an IPA to PA single four-way set associative cache for stage 1 followed by stage 2 translations. It can be enabled/disabled depending on the context. It shares RAMs with the PTW cache.

Appendix B

Boot-up environment

In order to effectively work and run our experiments, we had to initialize our Linux environment. This included many settings, libraries, development and insertion of kernel modules, necessary for our work. In this Appendix, we will give a fully detailed description of all modules and libraries developed as part of this work and some other utilities developed by other members of the Lab at FORTH.

Assuming we are working on a QFDB, one had to run the following three calls before having an environment to work:

1. ./load_boot_package.sh <number of FPGA (0, .., 3)> (load boot package of a specific design)
2. ./kexec_script.sh (load kernel Image modified to serve our purposes - mostly for the arm-smmu.c driver modifications)
3. ./new_remoteproc.sh (initialize environment)

B.1 Boot package loading

```
#!/bin/bash
DESIGN_NAME="2.2.4";
FPGA_NUM=$1;
range_of_num='^[0-9]+$';
range_of_fpga='[0-3]'

if [ -z "$FPGA_NUM" ] ; then
        echo "ERROR: FPGA number is REQUIRED in order to load boot package." >&2;
        exit 1;
elif ! [[ "$FPGA_NUM" =~ $range_of_num ]] ; then
        echo "ERROR: Input parameter is NOT a number." >&2;
        exit 1;
elif ! [[ "$FPGA_NUM" =~ $range_of_fpga ]] ; then
        echo "ERROR: Input parameter is OUT of range." >&2;
        exit 1;
fi
```

```
17
18 echo "Loading boot package of $DESIGN_NAME design for F$((FPGA_NUM+1)).....";
19
20 if [ "$FPGA_NUM" = "0" ] ; then
21         /root/qfdb-stuff/boot_package /home/pxirouch/bps/$DESIGN_NAME/output.bp.F1
22 elif [ "$FPGA_NUM" = "1" ] ; then
23         /root/qfdb-stuff/boot_package /home/pxirouch/bps/$DESIGN_NAME/output.bp.F2
24 elif [ "$FPGA_NUM" = "2" ] ; then
25         /root/qfdb-stuff/boot_package /home/pxirouch/bps/$DESIGN_NAME/output.bp.F3
26 elif [ "$FPGA_NUM" = "3" ] ; then
27         /root/qfdb-stuff/boot_package /home/pxirouch/bps/$DESIGN_NAME/output.bp.F4
28 fi
```

B.2 Loading and booting a different kernel Image

```
1 #!/bin/bash
2 kexec -l /home/psistakis/Image --reuse-cmdline
3 kexec -e
```

The first command (line 2) is responsible to link the new kernel Image file. We used the Linux kernel 4.9 that was taken from Xilinx repository and is part of "xilinx-v2017.2" tag. After having a copy (clone) of this repository we only had to cross-compile it, having in mind the target architecture, which in our case is AArch64. The "–reuse-cmdline", basically says the new Linux Image to boot in the current console (command line).

The second command (line 3) reboots (executes) the system with the new Linux kernel Image linked before.

An example of the command to build the Linux kernel and generate the Linux kernel Image is:

```
1 make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j8
```

B.3 Environment initialization

We use the script below to initialize our environment, before we run any application. This script makes sure that all necessary modules are inserted along with the R5 firmware.

```
1 #!/bin/sh
2 FPGA_NUM=$1;
3 range_of_num='^[0-9]+$';
```

```
4  range_of_fpga='[0-3]'
5
6  if [ -z "$FPGA_NUM" ] ; then
7          echo "ERROR: FPGA number is REQUIRED in order to insert unimem_coord
           module." >&2;
8          exit 1;
9  elif ! [[ "$FPGA_NUM" =~ $range_of_num ]] ; then
10         echo "ERROR: Input parameter is NOT a number." >&2;
11         exit 1;
12 elif ! [[ "$FPGA_NUM" =~ $range_of_fpga ]] ; then
13         echo "ERROR: Input parameter is OUT of range." >&2;
14         exit 1;
15 fi
16
17 if [ "$FPGA_NUM" = "0" ] ; then
18         U_COORDS=0x0;
19 elif [ "$FPGA_NUM" = "1" ] ; then
20         U_COORDS=0x1000;
21 elif [ "$FPGA_NUM" = "2" ] ; then
22         U_COORDS=0x2000;
23 elif [ "$FPGA_NUM" = "3" ] ; then
24         U_COORDS=0x3000;
25 fi
26
27 echo "Loading all necessary modules (exanest etc) and R5 firmware in
   F$((FPGA_NUM+1)).....";
28
29 /root/qfdb-stuff/rwphys/write32 0x80070004 0x1
30 /root/qfdb-stuff/rwphys/write32 0x80070000 0xffffffff
31
32 # Path of drivers
33 DRIVERS_PATH=/home/psistakis/unimem-exanet-drivers
34
35 insmod -f $DRIVERS_PATH/remoteproc.ko  # A generic framework with which AMP remote
   processors can be controlled (powered up/down), provided by Linux
36 insmod -f $DRIVERS_PATH/zynqmp_r5_remoteproc.ko    # A Xilinx ZynqMP R5 remoteproc
   driver to enable Linux kernel to bringup R5, and enable communication between
   Linux kernel and R5 (provided by Xilinx)
37 insmod -f $DRIVERS_PATH/unimem_coord.ko unimem_coords="$U_COORDS"
38 insmod -f $DRIVERS_PATH/mbox_back.ko   # Mailbox Driver developed by others at
   FORTH
39 insmod -f $DRIVERS_PATH/pack_back.ko   # Packetizer Driver developed by others at
   FORTH
40 insmod -f $DRIVERS_PATH/pldma.ko # PLDMA Module developed by others at FORTH
41
42 # Scratchpad module developed by others at FORTH
43 insmod -f $DRIVERS_PATH/scratchpad/scratchpad_alloc.ko
44
45 # Load R5 firmware
46 cp /home/psistakis/pf_r5.elf /lib/firmware/
47 cd /home/psistakis/
48 echo pf_r5.elf > /sys/class/remoteproc/remoteproc0/firmware
49 echo start > /sys/class/remoteproc/remoteproc0/state
50
```

```
51 # Insert module developed as part of our page-fault mechanism for packetizer
52 insmod -f $DRIVERS_PATH/pf_pckzer/pf_pckzer.ko
53
54 # Disable Transparent Huge Pages (THP) (if/when needed)
55 echo never > /sys/kernel/mm/transparent_hugepage/enabled
```