Stream communication across RISC-V Coherence Islands, with Read-Invalidate and Write-through-Combine Cache Policies

Orestis Mousouros

Thesis submitted in partial fulfillment of the requirements for the

Master 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 Katevenis*

Thesis Co-Advisor: Dr. *Nikolaos Chrysos*

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

University of Crete
Computer Science Department

Stream communication across RISC-V Coherence Islands, with Read-Invalidate and Write-through-Combine Cache Policies

Thesis submitted by
Orestis Mousouros
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________
Orestis Mousouros

Committee approvals: ______________________
Prof. Manolis Katevenis
Professor, Thesis Supervisor

____
Prof. Polyvios Pratikakis
Associate Professor, Committee Member

____
Prof. Vassilis Papaefstathiou
Assistant Professor, Committee Member

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

Heraklion, April 2022

Stream communication across RISC-V Coherence Islands, with Read-Invalidate and Write-through-Combine Cache Policies

Abstract

In the last decades, technology has reached a point of slow scaling, mainly due to limitations caused by the increasing amounts of power consumption. To gain performance speedup, hardware architects have turned to energy efficient processors, including some that are based on open-source RISC-V Instruction Set Architecture (ISA), which promises energy efficiency and high performance on multi-core chips.

This thesis contributes the design and implementation of a new approach for interprocessor stream communication across RISC-V Coherence Islands. Traditionally, the coherence islands use memory-to-memory communication over TCP/IP or Remote Direct Memory Access (RDMA) interconnections. Writing and reading data to and from memory at the endpoints heightens latency and depletes processor cycles. Instead, in our work, the communication confines itself between a core and another (remote) node that can either be a core or a memory. In particular, we propose a new Streaming Cache that resides next to Level 1 Cache (L1 Cache) and uses the same fast interface for communication with the core. We split the Streaming Cache into two logical parts: a) the producer, an outgoing streaming cache that handles streaming data departing from the node; b) the consumer, an incoming streaming cache that handles streaming data arriving to the node. Effectively, in the proposed streaming framework, instead of moving data across the main memory of the end-points, data of both the producer and the consumer can be accessed with same latency as the L1 Cache. To improve performance, we use the read-once/store-once cache policies in the Streaming Cache, which immediately recycle the space of already accessed streaming data. Furthermore, a Prefetcher fetches data from the (remote) node before they are needed, thus reducing the cost of read accesses, while the write accesses take advantage of a Write-Combiner, which combines neighboring data and sends them to the (remote) node. In our work, accesses to streaming data are recognized using virtual addresses without the need of extending ISA.

We implemented the proposed system in SystemVerilog, as an extension of the CVA6 (former ARIANE) single-core RISC-V CPU. We built the Incoming and Outgoing schemes of Streaming Cache, each with four (4) contexts (hardware streams), in order to support virtualization, and we tightly-coupled them with the Load/Store Unit (LSU) of the ARIANE. We also built a communication logic at the edges that sends/receives data over an AXI-4 interconnect.

We synthesized our design for Xilinx Zynq UltraScale+ MPSoC Field Programmable Gate Array (FPGA). The Incoming logic of our design utilizes 16839 Look-Up Tables (LUTs), 7506 Registers and 8 Block Random Access Memories

(BRAMs), and operates at 275 MHz, while the Outgoing logic utilizes 23606 LUTs, 8615 Registers and 8 BRAMs, and operates at 210 MHz.

We performed behavioral simulations to our RTL design in order 1) to verify the streaming functionality when coupled with the RISC-V cores and 2) to evaluate its performance. In our preliminary evaluations, we stream data from/to main memory of the ARIANE core, first using the traditional memory hierarchy and second using our optimized streaming cache. The promising results underline the performance gains due to the stream-optimized cache policies of our design, by managing to almost completely eliminate the latency of network's interconnection in our indicative hand-made bare metal benchmarking programs.

Επικοινωνία Ροών μεταξύ Νησιών Συνοχής RISC-V, με Πολιτικές Κρυφής Μνήμης Ανάγνωσης-Ακύρωσης και Εγγραφής-δια μέσου-Συνδυασμού

Περίληψη

Τις τελευταίες δεκαετίες, η τεχνολογία έχει φτάσει ένα σημείο αργής κλιμάκωσης, κυρίως λόγω περιορισμών που οφείλονται στις αυξημένες ανάγκες κατανάλωσης ενέργειας, με επιπτώσεις όπως τη δυσκολία αύξησης της ταχύτητας ενός πυρήνα ή προσθήκης περισσότερων πυρήνων σε πολυπύρηνους επεξεργαστές. Επειδή υπάρχει ακόμα ανάγκη για αύξηση της απόδοσης, οι αρχιτέκτονες υπολογιστών έχουν στραφεί σε ενεργειακά αποδοτικούς επεξεργαστές, συμπεριλαμβανομένων ορισμένων που βασίζονται στην ανοιχτού κώδικα Αρχιτεκτονική Συνόλου Εντολών (Instruction Set Architecture - ISA) RISC-V, η οποία υπόσχεται ενεργειακή απόδοση, χαμηλό κόστος υλοποίησης και βελτιωμένη απόδοση σε πολυπύρηνους επεξεργαστές.

Η παρούσα εργασία συμβάλλει στη σχεδίαση και υλοποίηση μιας νέας προσέγγισης επικοινωνίας ροών μεταξύ επεξεργαστών που βρίσκονται σε διαφορετικά Νησιά Συνοχής (Coherence Islands) RISV-V. Παραδοσιακά, τα νησιά συνοχής επικοινωνούν μέσω δικτύων σε επίπεδο συστήματος, τα οποία βασίζονται σε διασυνδέσεις που χρησιμοποιούν είτε TCP/IP ή Απομακρυσμένες Άμεσες Προσπελάσεις Μνήμης (Remote Direct Memory Access - RDMA). Σε αυτές τις δομές, οι κόμβοι επικοινωνίας ανταλλάσσουν δεδομένα που βρίσκονται αποκλειστικά στις μνήμες τους, κάτι που αυξάνει τις χρονικές καθυστερήσεις και εξαντλεί κύκλους επεξεργασίας. Το RDMA βελτιώνει την επικοινωνία μεταξύ μνημών, προσφέροντας μεταφορές δεδομένων οι οποίες εκκινούνται σε επίπεδο χρήστη, με μηδενικές αντιγραφές και μηδενικές επεξεργαστικές επιβαρύνσεις.

Σκοπός αυτής της εργασίας είναι να προσφέρει επικοινωνία μεταξύ ενός πυρήνα κι ενός άλλου (απομακρυσμένου) κόμβου, ο οποίος μπορεί να είναι ένας πυρήνας ή μια μνήμη. Συγκεκριμένα, προτείνουμε μια καινούρια Κρυφή Μνήμη αποκλειστικά για την υποστήριξη επικοινωνίας ροών, η οποία βρίσκεται δίπλα από την Κρυφή Μνήμη Επιπέδου 1 (L1 Cache) του πυρήνα και χρησιμοποιεί την ίδια γρήγορη διεπαφή για επικοινωνία με αυτόν. Χωρίσαμε την Κρυφή Μνήμη Ροών σε δυο μέρη λογικής: α) του παραγωγού, όπου το εξερχόμενο μέρος διαχειρίζεται δεδομένα που αναχωρούν από τον κόμβο, και β) του καταναλωτή, όπου το εισερχόμενο μέρος διαχειρίζεται δεδομένα που καταφθάνουν στον κόμβο. Ουσιαστικά, στην προτεινόμενη δομή διαχείρισης ροών, αντί τα δεδομένα να μετακινούνται μεταξύ των κυρίων μνημών των κόμβων, τα δεδομένα τόσο του παραγωγού, όσο και του καταναλωτή, μπορούν να προσπελαστούν με καθυστέρηση όπως αυτής της L1 Cache. Για να βελτιώσουμε την απόδοση, επιλέξαμε οι πολιτικές της Κρυφής Μνήμης Ροών να βασίζονται στην αρχή μοναδικής-ανάγνωσης/μοναδικής-εγγραφής, ώστε να γίνεται άμεση ανακύκλωση του χώρου δεδομένων ροών στα οποία έχει υπάρξει ήδη πρόσβαση. Επιπλέον, ένας Προανακτητής (Prefetcher) ανακτά δεδομένα από τον (απομακρυσμένο) κόμβο πριν χρειαστούν, με αποτέλεσμα τη μείωση του κόστους στις προσβάσεις ανάγνωσης, ενώ οι προσβάσεις εγγραφής επωφελούνται από έναν Συνδυαστή Εγγραφών (Write-Combiner), ο οποίος συνδυάζει γειτονικά δεδομένα και τα στέλνει στον (απομακρυσμένο) κόμβο. Στην εργασία μας, οι προσβάσεις σε δεδομένα ροών αναγνωρίζονται από τις εικονικές διευθύνσεις των εντολών, χωρίς την ανάγκη επέκτασης του ISA.

Υλοποιήσαμε αυτό το σύστημα, με τη γλώσσα περιγραφής υλικού SystemVerilog, και το προσθέσαμε ως επέκταση στον μονοπύρηνο RISC-V επεξεργαστή CVA6 (πρώην ARIANE). Σχεδιάσαμε τα Εισερχόμενα και Εξερχόμενα μέρη λογικής της Κρυφής Μνήμης Ροών να χρησιμοποιούν το καθένα (4) πλαίσια εργασίας σε πραγματικό υλικό προκειμένου να υποστηρίξουμε εικονικοποίηση, και είναι άμεσα συνδεδεμένα με τη Μονάδα Αναγνώσεων/Εγγραφών (Load/Store Unit - LSU) του ARIANE. Επίσης, στα άκρα έχει υλοποιηθεί λογική επικοινωνίας, η οποία ζητά και στέλνει δεδομένα μέσω μιας διασύνδεσης AXI-4.

Η εργασία μας έχει υλοποιηθεί για τη Συστοιχία Επιτόπια Προγραμματιζόμενων Πυλών (Field Programmable Gate Array - FPGA) Zynq UltraScale+ MPSoC της Xilinx. Για το Εισερχόμενο μέρος λογικής, από πλευράς χώρου χρησιμοποιήθηκαν 16839 Προγραμματιζόμενες Πύλες (LUTs), 7506 Καταχωρητές και 8 Μνήμες Τυχαίας Προσπέλασης (BRAMs), λειτουργώντας στα 275 MHz, ενώ για το Εξερχόμενο μέρος λογικής, χρησιμοποιήθηκαν 23606 LUTs, 8615 Καταχωρητές και 8 BRAMs, λειτουργώντας στα 210 MHz.

Προσομοιώσαμε την υλοποίησή μας προκειμένου 1) να επαληθεύσουμε τη λειτουργικότητα των ροών σε συνδυασμό με πυρήνες RISC-V και 2) να αξιολογήσουμε την απόδοσή της. Στις αξιολογήσεις μας, μεταφέρουμε δεδομένα ροών από και προς την κυρίως μνήμη του πυρήνα ARIANE, χρησιμοποιώντας πρώτα την παραδοσιακή ιεραρχία μνήμης και ύστερα την βελτιστοποιημένη Κρυφή Μνήμη Ροών. Τα αποτελέσματα παρουσιάζουν κέρδη απόδοσης χάρη στις πολιτικές βελτιστοποίησης ροών της υλοποίησή μας, αφού επιτυγχάνεται η σχεδόν πλήρης εξάλειψη των χρονικών καθυστερήσεων της διασύνδεσης του δικτύου στα ενδεικτικά προγράμματα συγκριτικής αξιολόγησης, χωρίς την υποστήριξη λειτουργικού συστήματος.

Acknowledgments

I would like to express my gratitude to Advisor Dr. Nikolaos Chrysos for his guidance and support during my studies and this work. Additionally, I would like to thank my Supervisor, Prof. Manolis GH Katevenis for his overall assistance and interesting discussions all these years. Furthermore, I would like to show appreciation to Prof. Polyvios Pratikakis and Prof. Vassilis Papaefstathiou for taking part in the examination committee for the evaluation of this work. Special thanks to Evangelos Mageiropoulos that helped with the writing of my master thesis.

Moreover, I would like to express my appreciation to Pantelis Xirouchakis for training and introducing me to the world of hardware design. Also, I would like to thank all the members of the CARV Laboratory team for their support and collaboration.

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

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the increasing need to analyze more data and the slow scaling of technology, both industry and academia try to find ways to push the boundaries and improve performance on processor chips. Initially, architects were trying to improve performance by continuously increasing the clock frequency, which caused enormous power consumption and heat dissipation. As the years passed and the chip manufacturing technology improved, multi-core era arrived, where architects could fit many cores in a single chip, by enabling work parallelization. Even though work parallelization restored energy efficiency, lowered silicon costs and improved the overall system performance, there was still need for performance speedup.

Last years, computer architects have turned their interest to energy efficient designs, such as Reduced Instruction Set Computer (RISC) architectures. Based on RISC principles, the arrival of open-source RISC-V Instruction Set Architecture (ISA) [1] has opened the way, both in academia and industry, to push even further the boundaries of designs, from micro-controllers to server-class processors. That helped computer architects to turn their interest to the utilization of accelerating designs, where exposing rich semantics and designing specialized hardware further increases performance.

However, even with the wide range of available RISC-V implementations and accelerating designs, there is a lack or poor support of features, which can be found already implemented in different ISAs. This makes it difficult to decide which design to choose and when to start from the ground up.

1.1 Motivation

In the wide variety of specialized designs on memory management, only a very small part of the research community has focused on accelerating the memory access patterns of data streaming and its communication between (remote) nodes. Stream communication can be found in many academic and industry applications, from High Performance Computing (HPC), to Internet of Things (IoT). In stream communication, applications typically operate on sequential and possibly remote

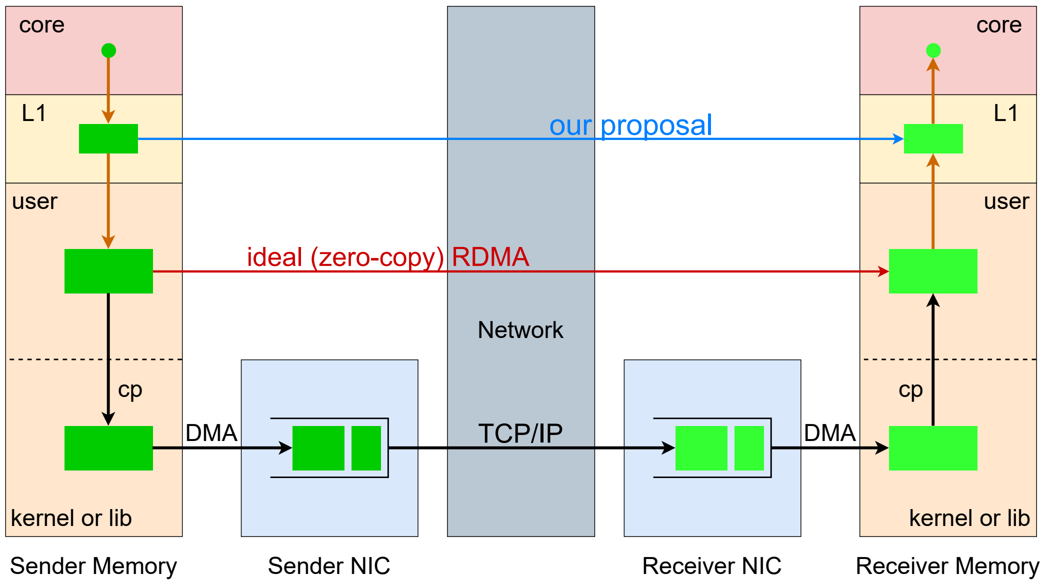


Figure 1.1: Diagram showing the different types of network communication. TCP/IP interconnect moves data from the user space of first node's memory to the user space of second node's memory through their kernel space, while RDMA bypasses the Operating System. In our approach, we will try to bypass memory accesses, by using them only if we want to fetch data from or backup data to the memory of the (remote) node.

chunks of data, like the ones that can be found in big data analytics' frameworks or sensory networks. To enable further higher performance speedup, systems must provide low latency and high throughput, and tightly couple stream communication with the processing core engines. In current RISC-V implementations, processors communicate either through shared memory (e.g. Open-Piton [2]) inside the same Coherence Island (CI), or using RDMA for memory to memory communication across CIs.

Traditionally, coherence islands communicate through system-level communication networks that are based on either TCP/IP or Remote Direct Memory Access (RDMA) interconnects, as shown in Figure 1.1. In these frameworks, the communication peers exchange data located in the main memory. RDMA advances memory-to-memory communication by offering user-level initiation of zero-copy and zero CPU overhead transfers.

Our goal is to design and implement a new hardware primitive for interprocessor communication across RISC-V CIs. Our approach leverages on the sequential access pattern in stream communication to offer read and write latencies for streaming data on the order of those of first-level caches. In our proof-of-concept implementation based on the ARIANE RISC-V hardware, an application running in a core can read and write remote data with virtual constant latency of 2 processor cycles, similar to L1 cache hit access time, for any size of data in the stream. Note that this is much faster than the local DRAM accesses, even more so than state-of-the-art RDMA technology, which places remote data in the DRAM, possibly invalidating or updating cached data in the destination coherence island (node).

In order to achieve this performance, we design a tightly-coupled cache for streaming data that sits next to the L1 cache of the ARIANE, with small timing overheads, but relatively big area (40445 LUTs and 16121 Registers) in our early-stage prototype FPGA implementation. The block uses the existing system interconnection to issue and prefetch remote data when needed, and adopts novel *read-once* and *write-once&combine* cache policies. In order to decouple the producer and consumer of the communication channel between them, called **stream**, we allow the consumer (producer) to fetch (issue) data within a window's distance from the head, called **sub-stream** or **sliding-window**, where the head is the oldest data available in the buffers of the stream.

1.2 Contributions

Throughout the duration of this thesis, the author dealt with a plethora of subjects, ranging from designing the idea of stream communication between (remote) nodes in cache-like manner, to implementing the Stream Cache for Incoming and Outgoing communication to and from the processor, with minimal changes to the Load/Store Units (LSUs) and interface of the processor, and evaluating it through test-benching. In more detail, the author's contributions to this thesis are the following:

- He was given the design of a specific RV64GC core called CVA6 (former ARIANE [2]) that the team in his laboratory is currently using, and for the purpose of understanding **a.** how it works, **b.** how to make minimal changes to it, so there will be no extra overheads, and **c.** how to test-bench the design later, he **1.** emulated the logic of the Load/Store Unit of the core and used the exact same interface as the one used for the communication between the core and the L1 Cache, **2.** emulated the logic of the L1 Cache, so he can make sure that his changes on the design will fit perfectly to the original core, and **3.** emulated the logic of the network adapter, for test-benching later in his design.

- He designed his implementation based also on the knowledge gained from the previous steps. For the network Incoming part, he designed a prefetcher to fetch the data as soon as possible. For the network Outgoing part, he designed a write-combiner for combining neighboring data and releasing them to the network as soon as all neighboring data are available. For the processor communication, because the original core was designed to communicate with a Virtually Indexed, Physically Tagged L1 Cache, he made changes to the emulated design of the core, to support his Virtually Indexed, Virtually Tagged Stream Cache, for the purpose that as long as the data are on the Stream Cache, there is no need to allocate space on the local address space.

- He created extra channels for the communication between the core and the Stream Cache, for the purpose that the original L1 Cache should not be enabled for processor requests that are intended only for Stream Cache, and vice-versa.

- He split the logic for the incoming and outgoing part, because they were not interfering in any way, and designed the Incoming and Outgoing Stream Caches. Each Stream Cache consists of multiple buffers and their individual control logic, where each one communicates with a specific (remote) node. Each pair of buffer and its control logic for communication purpose, is identified in the remainder of the paper as *incoming or outgoing stream.*

- He implemented the hardware block for the **Incoming Stream Cache**, which is responsible for transferring the incoming stream data from the network to the processor, when requested. It consists of **1.** multiple incoming stream buffers, where each one has its own controller to handle the requests and responses from either the network or the processor, **2.** the network handler, which is responsible for sending requests to the network in a round-robin manner, based on the active streams, and can receive the responses out-of-order, and **3.** the load instruction handler, which is responsible for forwarding the processor's load instruction requests to the appropriate incoming stream, for the intended incoming stream to respond back to the processor with the requested data.

- He test-benched the Incoming Stream Cache with the emulated Load Unit and Network, to test both the in-order and out-of-order requests of the Load Unit and responses of the Network, by setting an individual or all available hardware streams active.

- He implemented the hardware block for the **Outgoing Stream Cache**, which is responsible for transferring the outgoing stream data from the processor to the network, when there are enough available to be combined. It consists of **1.** multiple outgoing stream buffers, where each one has its own controller to handle the requests and responses from either the processor or the network, **2.** the store instruction handler, which is responsible for receiving the processor's store instruction requests and forwarding them to the appropriate outgoing stream, for the outgoing stream to combine and release them to the network, and **3.** the network handler, which is responsible for sending the combined stream data to the network in an out-of-order and round-robin manner, based on the active streams that have available combined data.

- He test-benched the Outgoing Stream Cache with the emulated Store Unit and Network, to test both the in-order and out-of-order requests of the Store Unit and responses of the Network, by setting an individual or all available hardware streams active.

- The original design was supporting communication to the network through the Cache Subsystem, i.e. Bypass channel and L1 Cache, by setting specific address ranges for non-cacheable and cacheable data requests. He made minor extensions to the Load/Store Unit of the core, for the purpose of supporting the extra channels created to communicate with the Streaming Caches. For the core to enable the appropriate channel each time to send data requests, a specific address range was set for the stream data inside the cacheable address region. That way, it was managed fr the design to have latency same to that of a L1 Cache, by making no ISA extensions.

- In order to evaluate the performance of the design, he test-benched the Stream Caches after integrating them with the actual ARIANE core. The test showed a latency of **2 clock cycles** for individual load/store instructions and **1 clock cycle** for back-to-back load/store instructions, same as the hit access of the Main Data Cache. Additionally, by taking advantage of the prefetcher (and write-combiner), the design yielded on our hand-made microbenchmark for back-to-back word loads an average latency of **9 clock cycles** for each load instruction request, with packet size equal to 64 Bytes. This shows an improvement compared to the average latency of **41 clock cycles** of using the Main Data Cache, which fetches one cacheline of 16 Bytes per cache miss, by having the core running at 1.5GHz with 150 cycles memory access latency.

- He synthesized the project in Vivado 2020.1, with a Xilinx Zynq UltraScale+ MPSoC (xczu9eg-ffvc900-2-e) as a target FPGA. The design achieved clock frequency of **275MHz** for the Incoming Stream Cache and **210MHz** for the Outgoing Stream Cache. Furthermore, the design with both Streaming Caches consumed an area of **40445 LUTs**, **16121 Registers**, and **16 BRAMs**.

1.3 Remainder of this thesis

The remainder of the paper is organized as follows: Chapter 2 presents our design, as we have tightly coupled it with ARIANE RISC-V core in hardware, Chapters 3 and 4 showcase the implementation details for a stream channel in the Streaming Caches, Chapter 5 presents the extensions that were implemented on the Load and Store Unit of the core to support our Streaming Caches, Chapter 6 demonstrates the hardware requirements of the module for an FPGA implementation and shows a rudimentary evaluation of its performance, Chapter 7 discusses related work items, and finally, Chapter 8 concludes the paper with notes on future work items.

Chapter 2

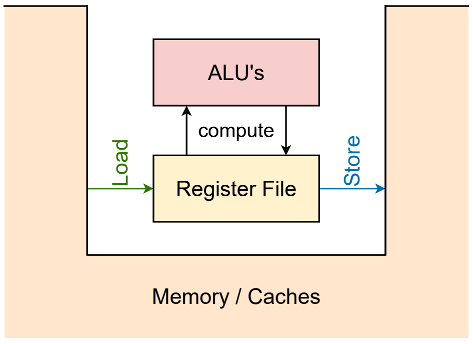
Streaming Cache

The main goal of our design is to achieve interprocessor stream communication across RISC-V coherence islands with very low latency and high throughput. In RISC architectures, the core accesses the memory by using load/store instructions, and makes computations only through registers, as shown in Figure 2.1a.

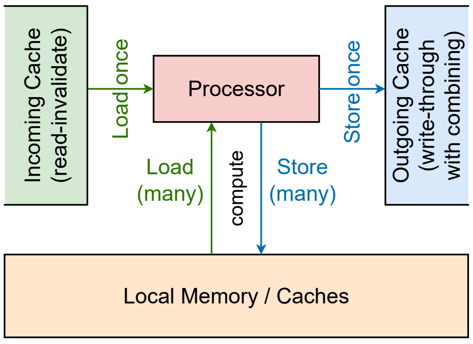
In our proposal, we use the same paradigm of load/store instructions in order to achieve stream communication, as shown in Figure 2.1b. The load/store instructions that trigger stream communication actions target a dedicated virtual address range (segment of a global address space).

Our work is based on the observation that accesses to stream data happen almost-sequentially and are rarely read from or written to the memory more than once. For that reason, our proposal supports read-once and write-once semantics for that data. To elaborate more, the idea behind our work is that if a programmer wants to load stream data, they have to be moved from the stream address space to the local address space. From the moment that the stream data are in local address space, they no longer allocate space in the incoming part of the streaming cache, but only physical space in the local address space. That way, if the programmer wants to reuse those data, it has to either store them on local address space, or to create a new stream flow for local usage, i.e. loopback. When the computations on stream data complete, the programmer has to move the data from local address space to stream address space, by storing them in the outgoing part of the streaming cache, because stream data are ready to be sent elsewhere and there are no more computations for them in the local address space.

To reduce the latency of reading and writing stream data, we have to use some type of accelerating logic. In the case of reading data, we need to have them available as soon as possible, before they get requested by the core. For that reason, we utilize a prefetcher in the incoming part of the streaming cache to hide the latency of the network. On the other hand, when we are writing stream data to be sent through the network, from one hand we want to avoid having the buffers filled and from the other we want to reduce the congestion in the network, by avoiding sending many small write requests. The solution that we followed is



(a) Basic RISC architecture



(b) Extended RISC architecture

Figure 2.1: A comparison between the old and the new RISC architecture, where the latter showcases the new semantics for handling the stream data. The proposed architecture reads once the stream data from the incoming part of the streaming cache, by loading them in local variables, and after the core finishes with computation, it writes the data once in the outgoing part of the streaming cache.

to combine a bunch of neighboring data, which go to the same (remote) node, and release them together as soon as possible. For that reason, we utilize a write combining buffer in the outgoing part of the streaming cache, for the purpose of reducing the congestion of the network, by sending bigger packets of data. Both prefetcher and write-combiner work independently to the core's read and write requests of stream data.

2.1 Cache Policies

In our work, we followed a Read-once / Read-Invalidate and Write-once & Combine / Write-through-Combine cache policy for stream data, which helps to automatically flush old data from the streaming caches.

- **Read-Invalidate** means that after a word has been read by the core, it becomes invalid in the incoming part of the Streaming Cache. In this way, stream data occupy space of the incoming part of the Streaming Cache only until they are loaded into registers. Of course, if stream data are read and need to be used again later, they will need to be copied in local memory, because of our read-once semantics.

- **Write-through-Combine** means that the outgoing part of the Streaming Cache combines the data (core store instructions) into lines as they arrive, and after the lines/packets have been sent successfully to memory or to remote node, their entries in the outgoing part of the Streaming Cache become invalid and the corresponding space becomes available for new data.

2.2 Tightly-coupling with ARIANE RISC-V core

In our first implementation, we use the RV64GC core called CVA6 (former ARIANE [2]). To adapt ARIANE to our protocol, we modified the Load/Store Unit (LSU) of the ARIANE's Execution Stage, as well as the interface between the core and the L1 Cache.

To reduce the latency of accessing stream data, we dedicate a new *Streaming Cache* that is tightly coupled with the core, similar to the L1 Cache of ARIANE, as outlined in Figure 2.2. The Streaming Cache is virtualized, offering *multiple channels* (totally eight in our current implementation) that can be allocated for example to different processes. Each channel for stream data has a separate physical buffer dedicated to stream communication.

In our first implementation, the physical buffer has a size of 4 KBytes (parametric), however it supports data streams with an infinite number of words. The reason that we choose a 4 KBytes buffer, is that we want enough space for the producer to fill the buffers (i.e. the remote node that feeds the incoming part of the streaming cache or the core that feeds the outgoing part of the streaming cache). With a very small sized stream buffer, the core will stall trying to consume or produce stream data, because buffers in streaming cache will not have enough space to fetch or send stream data, while with a very big sized buffer, we will be unnecessarily allocating a lot of resources, because the core will not be so fast to consume or produce stream data.

The Streaming Cache communicates with the core via load/store instructions in the global address space, same as the L1 Cache. To simplify the design of the Streaming Cache, we split it into an incoming and an outgoing segment. Specifically:

- The *Incoming Stream Cache* prefetches data, to anticipate core's read requests.
- The *Outgoing Stream Cache* combines neighboring write-data into packets and evicts them to the network as soon as possible to feed the (remote) consumer.

As shown in Figure 2.2, the LSU consists of separate load and store sub-units, where the first can issue a load and the latter a store (from the store buffer) request to the data cache. Furthermore, the L1 cache of ARIANE is *Virtually Indexed, Physically Tagged*. For that reason, together with the address request to the cache for tag comparison, the LSU sends another request for address translation to the Memory Management Unit (MMU), and expects the response in the same or the following cycles. Therefore, in best case scenario, it takes two cycles to complete the cache request, without necessarily stalling the core, i.e. the cache subsystem can handle back-to-back cache hits.

In more detail, when there is a valid load instruction, the Load Unit issues a load request to the Cache Subsystem, and an address translation request to the

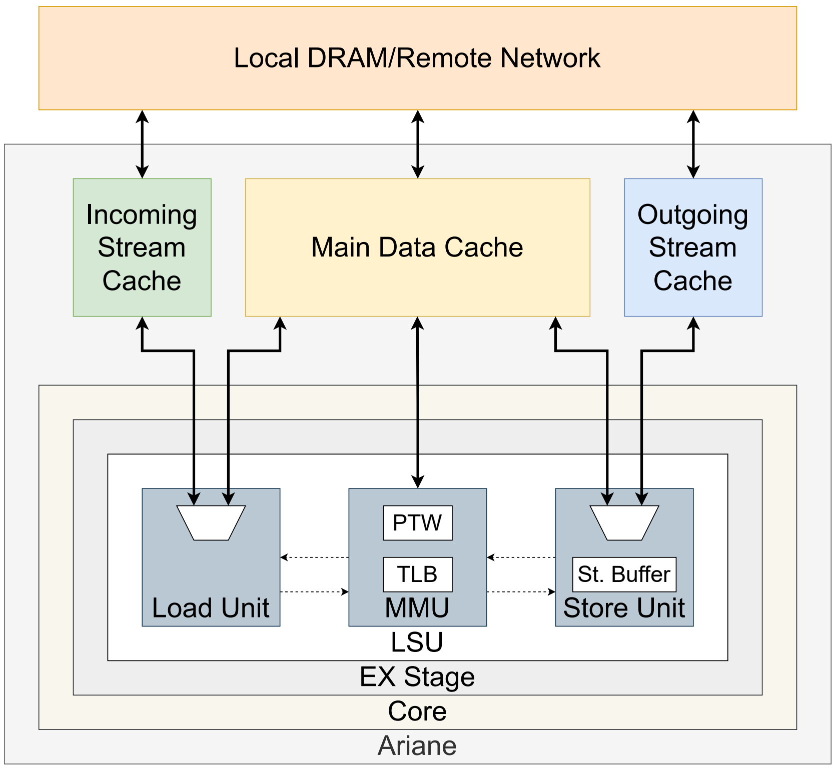


Figure 2.2: Overview of ARIANE with the Streaming Caches. Either of the Load or Store Units can send data requests to either Main Data Cache or the corresponding Streaming Cache, based on the virtual address of the instruction.

MMU. If there is no Translation Lookaside Buffer (TLB) hit on the same cycle, the Load Unit aborts the load request to the Cache Subsystem, and issues a new load request to the Cache Subsystem when there is a TLB hit. After, the Load Unit waits a request grant from the Cache Subsystem. When Load Unit receives the request grant, it forwards the translated address to the Cache Subsystem for tag matching and loads the data, when available.

On the other hand, when there is a valid store instruction, the Store Unit issues a store request to the Store Buffer, if it has available space, and an address translation request to the MMU. If on the same cycle there is no TLB hit or no available space on Store Buffer, the Store Unit aborts the store request to the Store Buffer, and issues a new store request to the Store Buffer when there is a TLB hit or available space on Store Buffer. When the Store Buffer has valid store requests for the Cache Subsystem, it issues the first in order store request to the Cache Subsystem with the translated address. When Store Unit receives the request grant, it can issue the next in order valid store request to the Cache Subsystem.

With the addition of our design, the Incoming and Outgoing Stream Caches

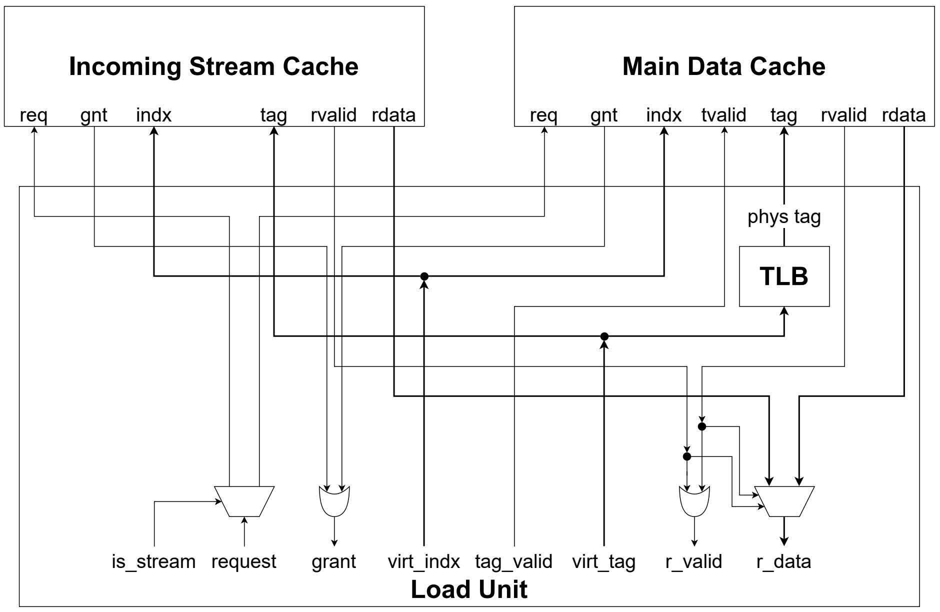


Figure 2.3: Incoming Stream Cache interface with Load Unit. Request signal is enabled when there is a valid load request, `is_stream` signal is enabled when the request is intended for the Incoming Stream Cache, grant signal is enabled when the Cache has accepted the load request, tag valid signal is enabled when the physical tag is valid, and `read_valid` signal is enabled when the requested data are valid.

are located next to the L1 Cache. The Incoming Stream Cache receives stream requests from the Load Unit of the LSU and the Outgoing Stream Cache from the Store Unit, respectively. The channels of these requests are almost identical with the ones that go to the main data cache.

It is important to note that our Streaming Caches use *untranslated virtual addresses*. The stream data belong to the virtual address of the process, as will be discussed in more detail in the next section, because they do not correspond to allocated physical space, other than the buffers of the Streaming Caches. So, the Streaming Caches are *Virtually Indexed, Virtually Tagged.*

In Figure 2.3, we depict in more detail the format of this interface for the case of the Load segment of the LSU. The requests either index the Incoming Stream Cache or the Main Data Cache, as identified by the request signals selected by the `is_stream` control. `is_stream` is active when the address of the Load instruction is inside the stream data address region. As happens with the main data cache, the incoming stream cache receives in the first cycle the index of the virtual address and uses it to read all stream channels (stream buffers coupled with control logic). In difference though to what happens in the main data cache, the incoming stream cache receives also in the first cycle the tag of the untranslated virtual address, in order to locate the channel of the stream request. A data miss will occur if the

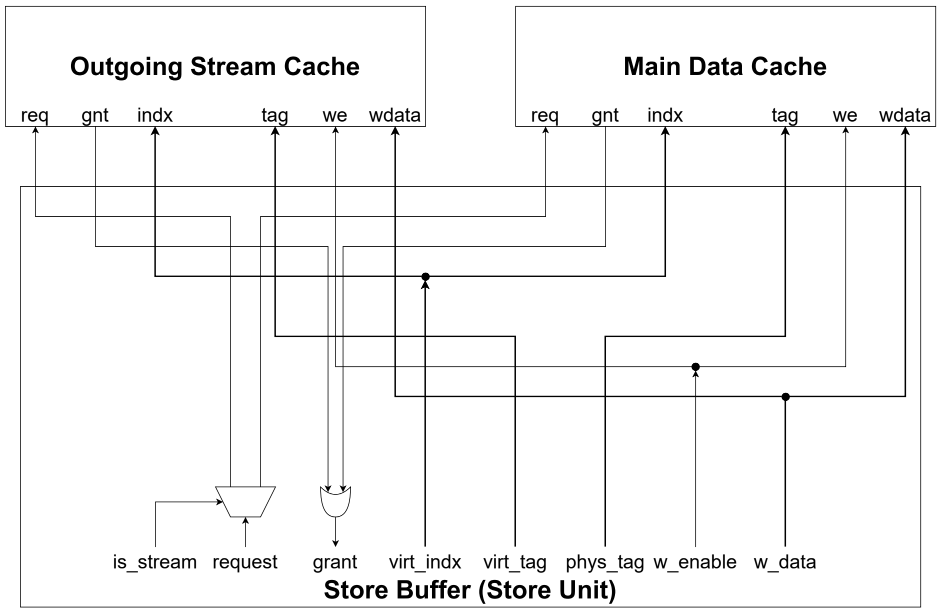


Figure 2.4: Outgoing Stream Cache interface with store Unit. Request signal is enabled when there is a valid store instruction, is_stream signal is enabled when the request is intended for the Outgoing Stream Cache, grant signal is enabled when the Cache has accepted the store request, and write enable signal is enabled when write data are valid.

requested stream channel does not yet have valid (fetched) data on the location of the index.

In a similar manner, in Figure 2.4, we show the interface of the Store segment of the LSU. When the data are ready to be stored, either Main Data Cache or Outgoing Stream Cache will be requested to accept the data in the same cycle, because the issue and the commit of a store instruction happen on different cycles, and the translation, if needed, happens on the same cycle or one cycle (or more) after the store instruction is issued. When the ARIANE core issues a store request, it uses the store buffer to store the request on the speculative queue, if the speculative queue has available space; otherwise the core stalls. After that, if the commit queue has available space and the commit signal is enabled, Store Buffer moves the request to the commit queue, for the purpose of the request to be handled later by the Cache Subsystem.

More details about the extensions on the LSU will be given in Chapter 5.

2.2.1 Identifying Stream Accesses

In our design, cores fetch (or push) stream data through various stream channels via load (or store) instructions. To split common from stream data, without extending the ISA, we reserve a region of the virtual address space for stream data,

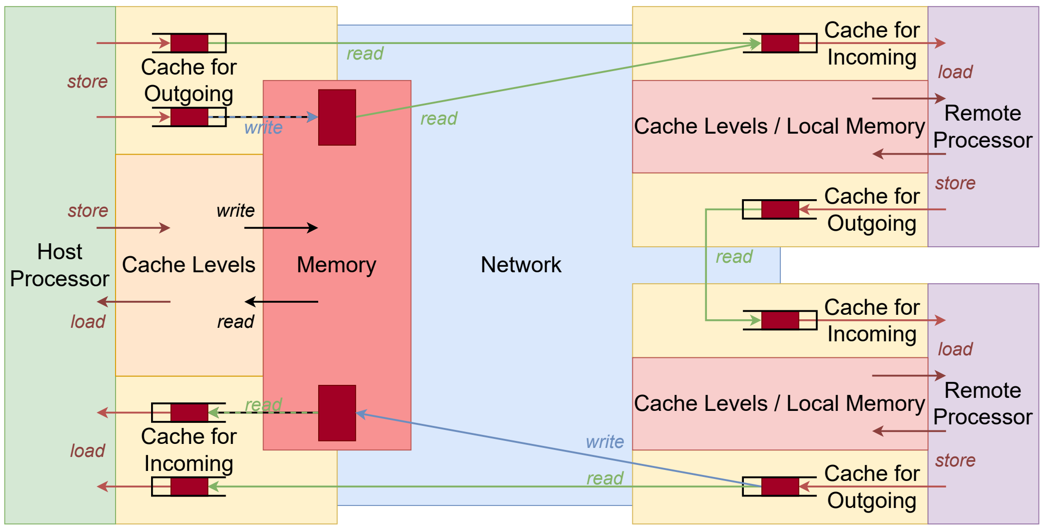


Figure 2.5: Different types of communications supported by our Streaming Caches. In the Incoming Stream Cache, each stream is responsible to request data, either from an Outgoing Stream Cache or a memory, while in Outgoing Stream Cache, a stream can either send the data to a memory when it has them ready, or should wait till they are ready and requested from an Incoming Stream Cache.

e.g. a few TBytes, and split that space for each active stream, e.g. 4 GBytes. In this way, each stream can be recognised uniquely by the virtual address of the request and has a unique Base Address. The software should issue requests only 4 GBytes past that Base Address to access the stream. In our current implementation, in case that the stream needs to be greater than the maximum allowed allocated virtual space, like on infinite sized streams, instead of creating new streams, the programmer can create programs, where the address can just wrap-around to access the data after the allocated virtual space of the stream. Note that there can be several streams established, however only a specific amount of them can be concurrently active in one node, as many as the channels offered by the streaming cache, e.g. 4 in our current implementation for each streaming cache.

2.2.2 Supported communication flows

As shown in Figure 2.5, our design supports communication: *(a)* from memory to incoming Stream Cache, *(b)* from Outgoing Stream Cache to Incoming Stream Cache, and *(c)* from Outgoing Stream Cache to memory.

Note that our design supports chaining, which means that two or more cores (or accelerators) can communicate with each other in a producer-consumer way and without the need of involving any memory in-between (i.e. Outgoing Stream Cache to Incoming Stream Cache). In chaining, we decided that only the prefetcher of an incoming stream channel can request data from an outgoing stream channel (i.e.

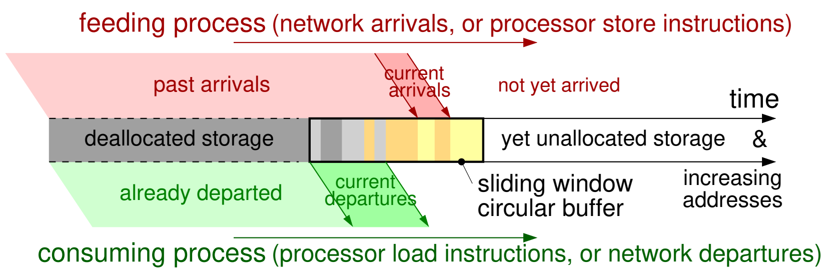

feeding process (network arrivals, or processor store instructions)
past arrivals
current arrivals
not yet arrived
time
deallocated storage
yet unallocated storage &
increasing addresses
already departed
current departures
sliding window circular buffer
consuming process (processor load instructions, or network departures)

Figure 2.6: Timing diagram showing the active sliding window in an Incoming or Outgoing Stream Cache. Both arrivals and departures can happen Out-of-Order, but the sliding window moves steadily in a step-wise way, when the departure of its head data finish successfully.

Read Channel), for the purpose of minimizing congestion and avoiding wasting packets in the network, by having an outgoing stream channel sending data to an incoming stream channel that either doesn't need at the moment or doesn't have available space for.

In the scenario of a stream channel communicating with a memory, the network channel needs to pass from an I/O MMU, for the purpose of the remote virtual address to get translated to a physical address. Our design does not support any type of address translation, other than pairing the stream base virtual address and the remote base virtual address for each stream channel, which are responsible for the core and network communication, respectively. Those base virtual addresses are given by the core to the stream channel during its initialization, together with the virtual space that each address can utilize, when the core wants to enable the stream channel.

In the hardware, every active stream channel has a dedicated memory that we implement using a circular buffer, as shown in Figure 2.6. The processing units are expected to access stream data in almost-sequential manner, i.e. our design tolerates out-of-order accesses (e.g. due to out-of-order production/consumption or due to out-of-order network delivery) up to a certain window size. Effectively, the buffer of the incoming or outgoing stream channel stores the currently active portion of the stream, what we call *sliding window*.

For the needs of most software libraries and applications for parallel and distributed systems, the current version of our design can support *workload sharing*, such as **splitting** and **reduction** operations for stream flows. For splitting, the core has to allocate various outgoing stream channels to store the stream data that should be sent to the (remote) nodes that will share the processing of the produced workload, while for reduction, the core has to allocate various incoming stream channels to load the stream data for consumption that will arrive from the (remote) nodes.

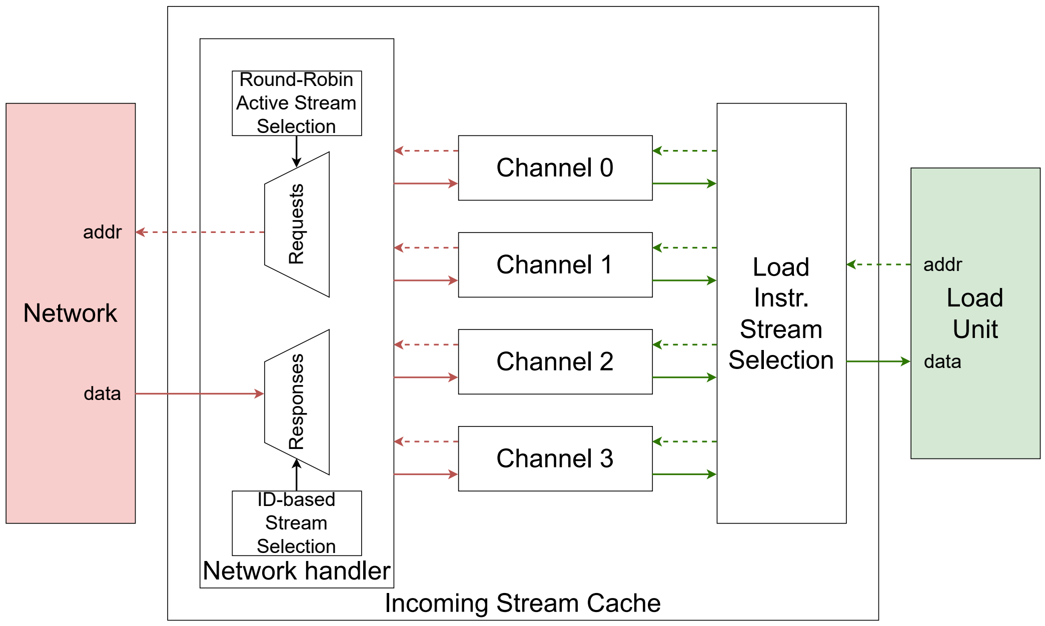


Figure 2.7: General design of the Incoming Stream Cache, with four hardware stream channels (circular buffers). As with the main data cache and its ways, a load instruction is processed by all stream channels in parallel, but only one will handle the request and respond back. Each stream channel can be assigned for only one type of communication, either with a memory or an outgoing stream cache.

2.3 Overall diagram of Streaming Caches

Figure 2.7 shows the overall block design of the incoming stream cache. We use one main controller for each stream channel and two request/response handlers for all stream channels.

One part of the controller is responsible of handling the requests and responses of the Load Unit of the core. The other part of the controller is responsible of handling the network requests and responses that can either come from a memory on an outgoing stream cache. Both cooperate to update correctly the stream data and the per-word validity bits per stream channel.

About the request/response handlers, one resides between the core and the stream channels, while the other one resides between the stream channels and the network. The first handler is responsible for the stream channel selection and simply multiplexes and de-multiplexes the signals, because there can be only one load request per cycle, intended for one stream channel only. The other handler has a round-robin mechanism to choose fairly a request from each active stream channel, while it simply multiplexes and de-multiplexes the response signals to the correct stream channel.

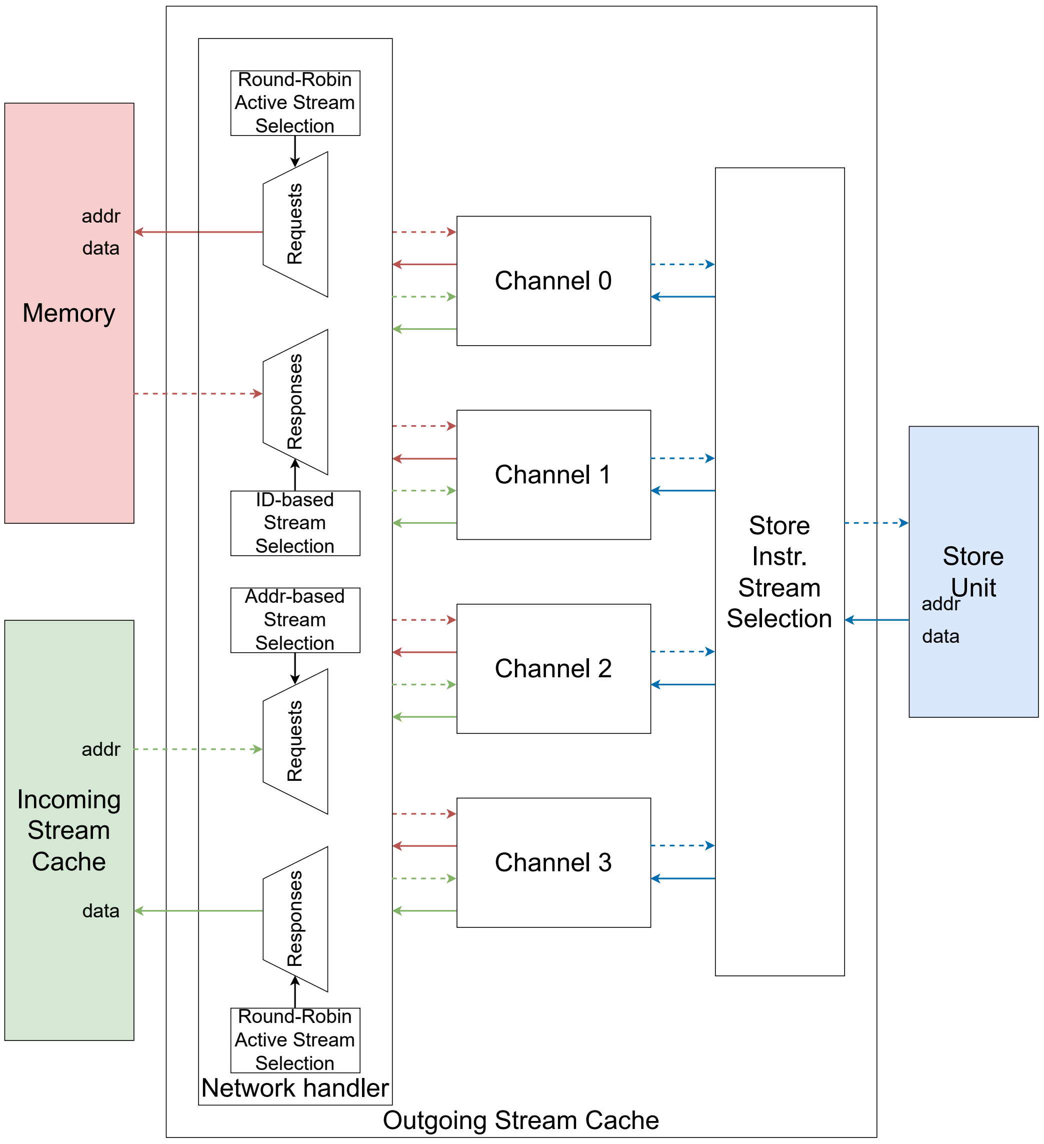


Figure 2.8: General design of the Outgoing Stream Cache, with four hardware stream channels (circular buffers). As with the main data cache and its ways, a store instruction is processed by all stream channels in parallel, but only one will handle the request and respond back. Each stream channel can be assigned for only one type of communication, either with a memory or an incoming stream cache.

In contrast to the incoming stream cache, outgoing stream cache has two modes for each stream channel. When the first mode is enabled, it means that the stream channel is set to communicate with a memory, while when the second mode is enabled, it means that the stream channel is set to communicate with an incoming stream cache, as shown in Figure 2.8. On the first case, the stream channel controller is responsible to send the data when available and combined, while, on the second case, the data are sent when requested and combined. For that reason, in the outgoing stream cache, we use one main controller for each stream channel and three request/response handlers.

One part of the controller is responsible of handling the requests and responses of the Store Unit of the core. The other part of the controller is responsible of handling the network requests and responses, but for each stream channel only one mode can be enabled. Both cooperate to update correctly the stream data and the per-word validity bits per stream channel.

Regarding the request/response handlers, one resides between the core and the stream channels, while the other two reside between the stream channels and the network. The first handler is responsible for the stream channel selection and simply multiplexes and de-multiplexes the signals, because there can be only one store request per cycle, intended for one stream channel only. The second handler, i.e. for communication with a memory, has a round-robin mechanism to choose fairly a request from each active stream channel and multiplexes and de-multiplexes the response signals to the correct stream. The third handler, i.e. for communication with an incoming stream cache, simply multiplexes and de-multiplexes the signals for the correct active stream channel to update its state, while it has a round-robin mechanism to choose fairly a response from each active stream channel.

For both Streaming Caches, all network handlers work simultaneously to transform the requests or responses to a format that either the controller of the stream channels or the network can understand. The network is shared among the different stream channels on a per-packet basis. Currently, for remote requests in our design, we use the AXI4 interface [3]. For the incoming stream cache, the network handler uses an AXI4 Read Master Adapter. For the outgoing stream cache, the first network handler uses an AXI4 Write Master Adapter, while the second network handler uses AXI Read Slave Adapter.

2.4 Internal organization of Stream Caches

Figure 2.9 depicts the pointers deployed in each stream channel of the Incoming Stream Cache and the states of stream entries. Each position in the stream channel can be in one of the following states:

- **Unallocated**: Data currently outside the sliding window that will get requested and read by core in the future.

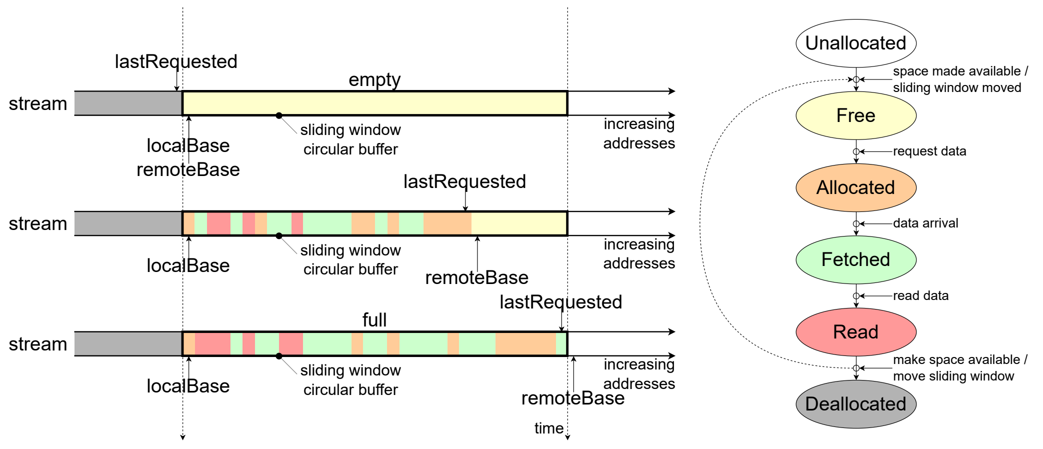


Figure 2.9: Incoming stream circular buffer for communication with either a memory or an outgoing stream cache, showing three different timestamps and the states of the data for a relatively small buffer.

- **Free**: Data inside the sliding window, but not requested (i.e. prefetched) yet.
- **Allocated**: Data requested but not yet present in the cache.
- **Fetched**: Data present in the incoming cache but not yet read from the core.
- **Read**: Data present in the cache and read from the core.
- **Deallocated**: Data outside the sliding window that have been read from the core in the past.

The *localBase* points to the oldest word inside the sliding window that has not yet been processed (i.e. read from the core). The *lastRequested* points to the last data that have been requested for the incoming stream. The *remoteBase* pointer stores the (possibly remote) address of the word that needs to be prefetched next, and in a way points to the data after where the lastRequested points. In the policy that we have implemented, the lastRequested and remoteBase pointers move incrementally as long as there is free space in the incoming stream circular buffer, by requesting data to fill the circular buffer.

Figure 2.10 depicts the corresponding states and pointers for the outgoing stream channels that send data to a memory. Each position in the stream channel can be in one of the following states:

- **Unallocated**: Data outside the sliding window that will get stored and sent to memory in the future.
- **Allocated**: Space available in the cache for data to get stored.

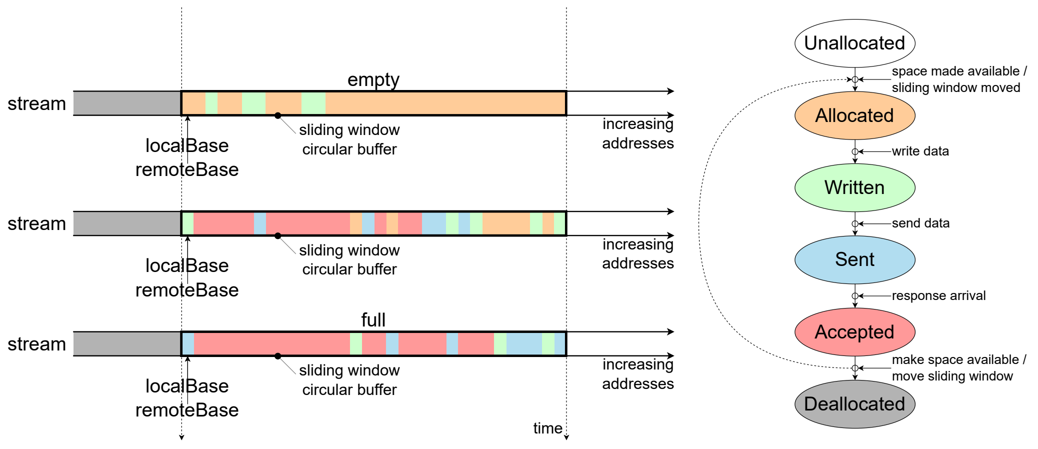


Figure 2.10: Outgoing stream circular buffer for communication with a memory, showing three different timestamps and the states of the data for a relatively small buffer.

- **Written**: Data present in the outgoing cache but not sent yet to memory.
- **Sent**: Data sent to network.
- **Accepted**: Data sent and accepted by memory.
- **Deallocated**: Data outside the sliding window that were written from the core and sent to the network in the past.

Figure 2.11 depicts the corresponding states and pointers for the outgoing stream channels that send data to an incoming stream cache. Each position in the stream channel can be in one of the following states:

- **Unallocated**: Data out of the sliding window, i.e. future data to get stored and sent to an incoming stream cache.
- **Allocated**: Space available in the cache for data to get stored.
- **Requested**: Data requested by the prefetcher of an incoming stream cache, but not written yet by the core.
- **Written**: Data written by the core, but not requested yet by the prefetcher of an incoming stream cache.
- **Ready**: Data requested and ready to be sent in incoming stream cache.
- **Sent**: Data sent to network.
- **Deallocated**: Data out of the sliding window, i.e. written from the core and sent to the network in the past.

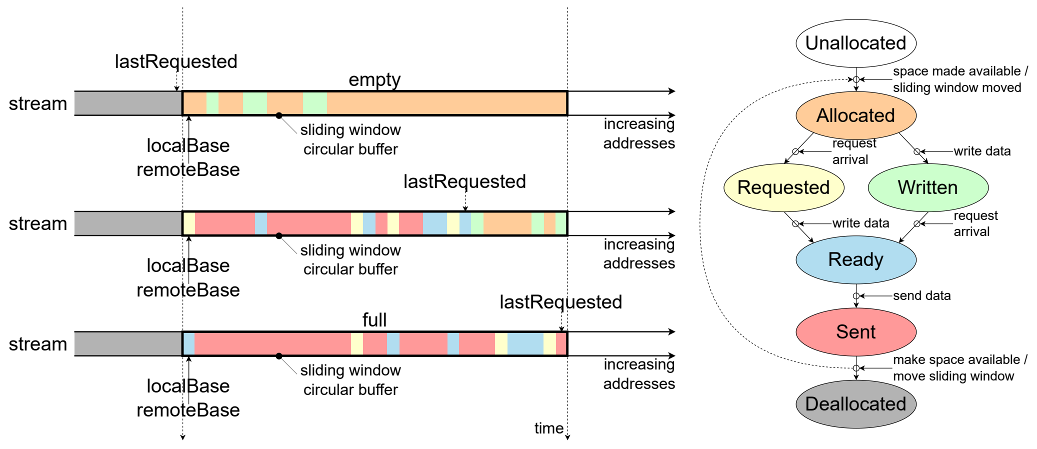


Figure 2.11: Outgoing stream circular buffer for communication with an incoming stream cache, showing three different timestamps and the states of the data for a relatively small buffer.

The *localBase* points to the oldest word inside the sliding window that has not yet been sent to network (or not even written by core). The *remoteBase* pointer stores the (possibly remote) address of the first word in the sliding window that can be sent to the network. In the policy that we have implemented, the remoteBase pointer moves together with the localBase pointer, because we can easily index the remaining data that are stored in the sliding window of outgoing stream cache.

In all stream channels, localBase pointer is used to flush old data from the stream channels, which enables that pointer to (incrementally) move ahead, therefore the sliding window too. Furthermore, there are two more pointers that can issue read/write requests or responses on the stream channels. One is used to handle the core's requests, while the other is used to handle the network arrivals or departures of data. Both of them can point anywhere at any time inside the sliding window, or more specifically between the area of the localBase and lastRequested pointers, due to the out-of-order production/consumption or out-of-order network delivery. For that reason, per cycle, each stream channel controller is responsible to handle simultaneously (with the help of the aforementioned pointers) the following four scenarios: **1.** requests/responses of the core, **2.** flushing of old data, **3.** network requests, and **4.** network responses.

Another important thing to mention is that each one of these pointers is used in only one of the four scenarios/cases. The control logic of these pointers expects the area that each one is pointing in the circular buffer to reach a specific state, if not already there, and when the operation on that area finishes, the state of that area changes and the corresponding pointer either moves ahead incrementally, or lets the stream channel's control logic (based on the requests/responses of the core or the remote node) to point it to a new area.

2.4.1 State differentiation

To differentiate the states for each of the three communication flows, we have to uniquely identify them. That way, we decided to use three identifiers for stream data, which are based on the requests/responses of the core or the (remote) node.

For an Incoming Stream Channel, as shown in Table 2.1, and based on the state description that was given earlier, we have:

- **valid not read**: a per-word validity identifier to describe if the core has read or not the specific word.
- **outstanding enabled**: a per-packet validity identifier to describe if a network request has been issued to the (remote) node for the specific data.
- **valid fetched**: a per-cacheline (i.e. data burst / network width) validity identifier to describe if the specific data have been fetched from the (remote) node.

State	**Validity Bits**		
	Fetched	***Not Read***	***Outstanding***
Free	Disabled	Disabled	Disabled
Allocated	Disabled	Disabled	Enabled
Fetched	Enabled	Enabled	Disabled
Read	Enabled	Disabled	Disabled

Table 2.1: Incoming stream states for communication with either a memory or an outgoing stream cache. Free state means that no data have been requested. Allocated state means that data have been requested, but not fetched yet. Fetched state means that the requested data have been arrived and are ready for the core to read. Read state means that the data have been read by the core and the space is ready to be recycled.

For an Outgoing Stream Channel, as shown in Tables 2.2 and 2.3, and based on the state descriptions that were given earlier, we have:

- **valid written**: a per-word validity identifier to describe if the core has written or not the specific word.
- **outstanding enabled**: a per-packet validity identifier to describe if a network request has been issued to or by the (remote) node for the specific data.
- **valid sent**: a per-cacheline (i.e. data burst / network width) validity identifier to describe if the specific data have been sent successfully to the (remote) node.

State	**Validity Bits**		
	Sent	***Written***	***Outstanding***
Allocated	Disabled	Disabled	Disabled
Written	Disabled	Enabled	Disabled
Sent	Disabled	Enabled	Enabled
Accepted	Enabled	Enabled	Disabled

Table 2.2: Outgoing stream states for communication with a memory. Allocated state means that no data have been written on the available space. Written state means that data have been written by the core on the available space. Sent state means that the written data have been sent to the network. Accepted state means that the data have been sent successfully and the space is ready to be recycled.

State	**Validity Bits**		
	Sent	***Written***	***Outstanding***
Allocated	Disabled	Disabled	Disabled
Requested	Disabled	Disabled	Enabled
Written	Disabled	Enabled	Disabled
Ready	Disabled	Enabled	Enabled
Sent	Enabled	Enabled	Disabled

Table 2.3: Outgoing stream states for communication with an incoming stream cache. Allocated state means that no data have been written on the available space. Requested state means that the data have been requested, but not written by the core yet. Written state means that data have been written by the core on the available space, but not requested yet. Ready state means that the written data are ready to be sent to the network. Sent state means that the written data have been sent successfully and the space is ready to be recycled.

It is important to note that the reason we decided to use three sub-identifiers for only four states inside the stream buffers has to do with the following point. When we started designing the main idea of this work, we were intending to support packets that may include invalid words. Adding this feature would not increase the design complexity by much, compared to what we have already presented. However, we saw that such implementation would most likely increase the packets in the network, which would result to high congestion if there is no proper congestion management. For that reason, we decided to go with a simpler design, where we only send packets when all of the words are valid.

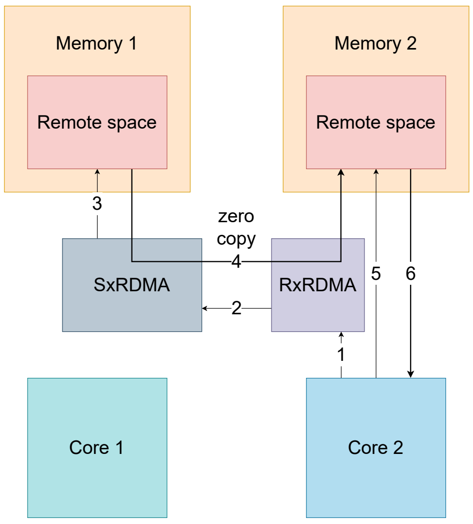
2.5 Comparison with Remote Direct Memory Access (RDMA)

RDMA is a widely known protocol in High Performance Computing (HPC) that is used in computer clusters to access directly the memory of one node from another. Its leverage is that there is no need to involve either node's Operating System (OS), by reducing the copies needed, and allowing high throughput and low latency networking. That concept is called zero-copy networking, which means that no data need to get copied between the application memory and the data buffers of either OS when the transfer happens between the memories. The main issue that this protocol has, is that the target node doesn't know when the request gets finished, because the communication is single-sided. On the next paragraphs, we will describe briefly the Read and Write operations of an RDMA engine and compare them to our design, by describing further the advantages and disadvantages of each design.

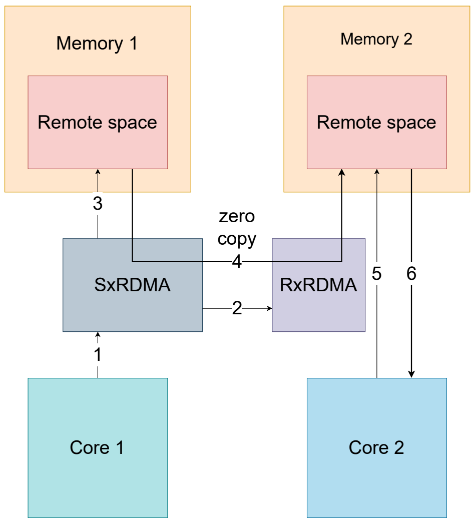
Figure 2.12a presents the read operation between two nodes, where the core of the target node wants to read data from the memory of the remote node. At the start, the target core initiates a Read RDMA request to the local RDMA engine. That request is received by the Receiver part of the local RDMA engine and forwarded to the Sender part of the remote RDMA engine. When the latter receives the read request, it triggers its DMA engine to read data from the local memory of the remote node, and sends the data to the target RDMA engine, so it can save them to the local memory of the target node. When the remote data are saved successfully on the target memory, the target core can request them from there, such as any other local data.

Figure 2.12b presents the write operation between two nodes, where the core of the target node wants to write data to the memory of the remote node. At the start, the target core initiates a Write RDMA request to the local RDMA engine. That request is received by the Sender part of the local RDMA engine and forwarded to the Receiver part of the remote RDMA engine. Afterwards, the Sender part of the local RDMA engine triggers its DMA engine to read data from the local memory, and sends the data to the target RDMA engine, so it can save them to the local memory of the target node.

As described briefly earlier, due to the communication being single-sided in RDMA, the target node cannot know when the transaction finishes. One way that is commonly used to solve that issue is for the target RDMA engine to set a register as ready when the transaction finishes (i.e. completion notification), for the target core to get notified (i.e. through polling) and be able to request that data from its local memory. The same issue is observed in our design when a Streaming Cache communicates with a memory. From the moment a stream channel is enabled, it starts immediately to request or send data to the (remote) memory. For that reason, the stream has to be initialized right after the (remote) memory is ready to accept read or write requests, for the core to enable the stream channel. For



(a) RDMA Read Operation.

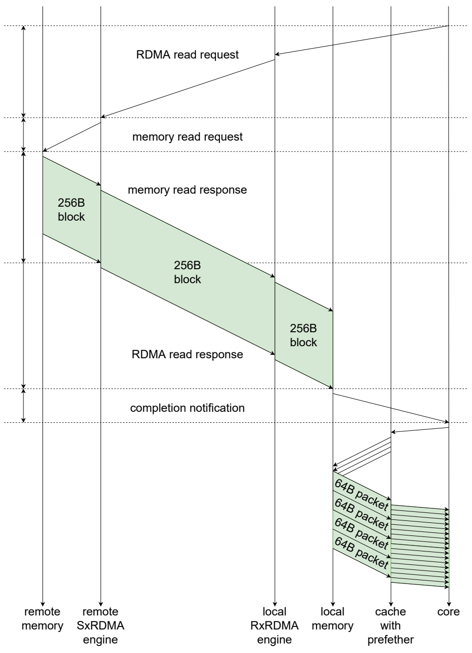


(b) RDMA Write Operation.

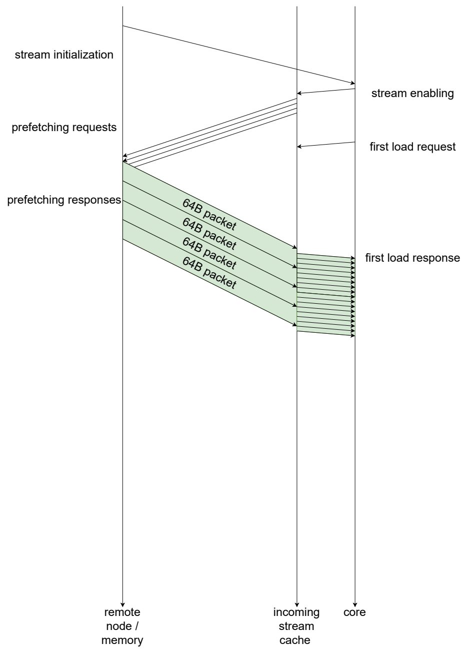
Figure 2.12: Basic idea for the Read and Write operations of RDMA engine. On Read operation, the core sends a read request on the Receiver of the local RDMA engine, where the latter sends a read request on the Sender of the remote RDMA engine. Conversely, on Write operation, the core sends a write request on the Sender of the local RDMA engine, where the latter sends a write request on the Receiver of the remote RDMA engine. Afterwards, in either operation, the DMA engine of the Sender requests data to be sent from one memory to the other, through the two RDMA engines, with zero copies in between.

streams bigger than a page, there is need for a mechanism near the memory of the (remote) node, which should be responsible to handle correctly the requests to the memory, for the purpose of avoiding the Stale Data problem of IO-coherence, like the one that can be found in RDMA. By achieving to always have valid data or available space in the requested area of the memory, the prefetcher of an incoming stream channel won't fetch again data that have already been read, and the write-combiner of an outgoing stream channel won't overwrite data that haven't been backed up or read yet. Generally, our protocol is designed mainly for core to core communication, where the communication between a core and a memory is used either for small transactions or for temporally storing versions of data (i.e. back up of latest stored data). Otherwise, we consider that the page that is located in a (remote) memory changes only when the sliding window moves, by requesting data from a new page that has already valid data.

Figure 2.13 compares the Read operation of an RDMA engine with the initialization and data prefetching of an Incoming Stream Channel, by considering that any type of acknowledgement messages are not causing extra time overheads. Additionally, on the timing diagram of the RDMA engine, we consider a core with a cache that supports a similar prefetcher to the one that we use for the Incoming



(a) Timing diagram of RDMA initiation and fetched data loading.



(b) Timing diagram of Incoming Stream initiation and fetched data loading.

Figure 2.13: Timing diagram comparison between RDMA Engine's Read operation and Incoming Stream's prefetching. On RDMA Engine approach, we observe a coarse-grain/block-sized arrival of remote data with per-block synchronization, while on Incoming Stream Cache approach we observe a fine-grain/packet-sized arrival of remote data, which allows per-word synchronization for lower latency.

Stream Cache, which can prefetch multiple cachelines in a single request.

In RDMA approach, the remote data arrive in a coarse-grain format (i.e. block-sized), and can start get processed after whole block arrives and the core gets notified (i.e. per-block synchronization). In difference, in an Incoming Stream Cache, after a stream gets initialized, the prefetcher can request data in a fine-grain format (i.e. packet-sized), where the core can read a word as soon as it arrives (i.e. per-word synchronization), if not already there, like what happens on the cache miss of the L1 Cache of the core.

A big issue on the RDMA engine of the target node is that there is need to flush the caches in every RDMA transaction, to avoid the Stale Data problem of IO-coherence. Comparing to our design, there is no need to invalidate any of the data saved on the caches of the memory hierarchy, because the stream channels will always have the correct version of the data. That is achieved by saving and handling stream data on a specialized space, the stream buffers of the Streaming Caches. So, even when the stream data are getting invalidated after they are read, they can still be accessed from the local address space, if the programmer saves them there, like what happens on the RDMA, after a transaction finishes.

Another advantage of our design is that the core simply stalls till the data arrive, like any load or store instruction of the common memory hierarchy, comparing to the polling that needs to happen on the RDMA transaction, till it finishes, when the target core receives the completion notification message. In addition, as described earlier, another benefit of our design is that two or more cores of different nodes can communicate with each other in a producer-consumer way and without the need of involving any memory in-between, which reduces further the latency.

The only disadvantage of the current implementation of our design, compared to RDMA, is that all the data have to be read, if requested, because of the stream semantics that we have included. For that reason, it can currently support only streams of infinite data (or finite, with few extensions on the design to correctly terminate and reset the logic on the stream channels).

Chapter 3

Incoming Stream Cache

In this chapter, we present the hardware implementation of the Incoming Stream Cache that is located next to the L1 Cache of ARIANE. The Incoming Stream Cache consists of multiple incoming stream channels, where each of them can fetch data from a (remote) node. It utilizes two interfaces for communication that all the incoming stream channels share, one being the LSU inteface for communication with the core, and the other being the AXI interface for network communication. First, we will present the design of a single incoming stream channel, and then we will describe how the incoming stream channels utilize the aforementioned interfaces.

3.1 General Description

The incoming stream cache consists of multiple circular buffers that support multiple concurrent stream channel channels. This is similar to the multiple ways in a typical data cache. One such circular buffer is depicted in Figure 3.1, where we present how an incoming stream channel handles the load instruction request from the processor and responds when it has the data available. Data words are organized into lines, each consisting of four (4) 32-bit words in our design. However, instead of having a tag per cacheline (CL), in the streaming cache, we have a tag per stream channel. Each circular buffer has space of 4KBytes (parametric) dedicated to a single stream channel, i.e. 256 lines. This corresponds to the size of the sliding window.

For each cache line, we have a Valid Fetched bit, to indicate whether the corresponding words are present in the cache (either in Fetched or in Read state). In addition, we maintain an additional state per word to indicate whether the words have been already read by the processor (per-word validity bits). Furthermore, there is an Outstanding Enabled bit per CL, to indicate whether a read request has been issued and we wait for a response for the specific CL. As soon as all words in a line have been read, the line can be recycled, i.e. can used for future data in the stream. In that scenario, the basePointer and the sliding window move by one

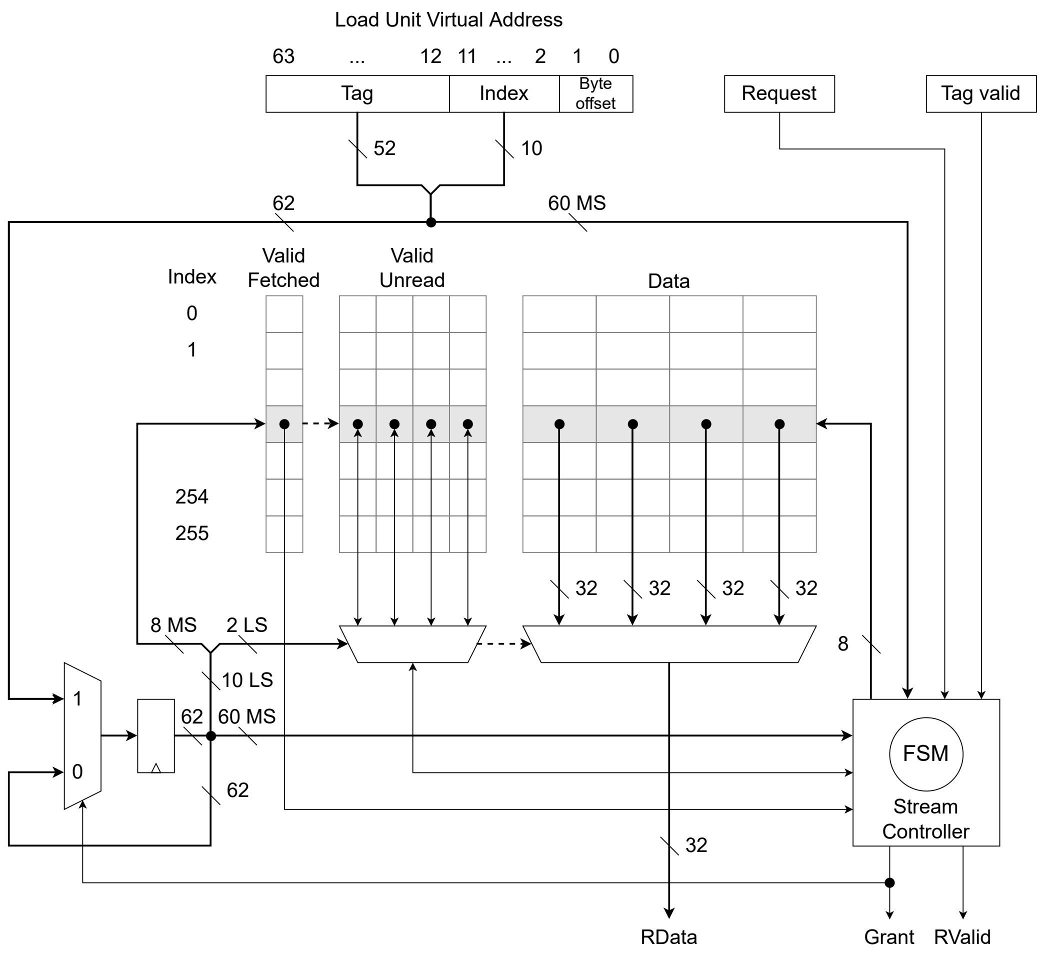


Figure 3.1: Load instruction diagram for Incoming Stream Channel. The Address is virtual (untranslated) and is issued by the LSU unit of the ARIANE. The tag in the address field is compared with the stream channel tag of the four stream channels. The signals at the bottom (Valid, Data and Grant signals) are awaited by the ARIANE Load-Store-Unit.

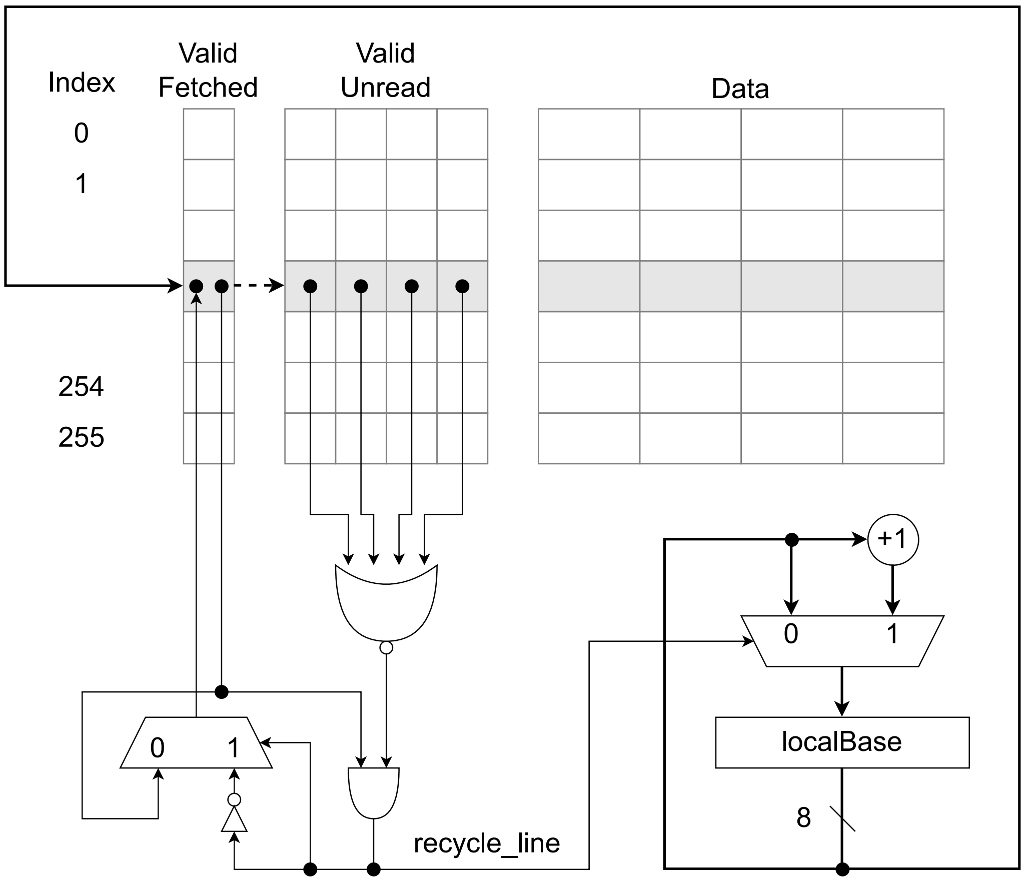


Figure 3.2: Recycling diagram for Incoming Stream Channel. localBase pointer moves when all the data in the cacheline that it points have been fetched and read, and become invalid.

line, like in Figure 3.2.

Each circular buffer is implemented as a two-port BRAM, while we use a 256×4-bit array of registers to store the per-word validity bits and two 256×1-bit array of registers to store the Valid Fetched bit and the Outstanding Enabled bit of a whole line.

3.2 Control Logic for communication with the core

Each incoming stream channel has a controller that utilizes a FSM with two states, as shown in Figure 3.3. On the IDLE state, the controller waits for a read request. When it happens, the controller checks if the request is intended for that incoming stream channel. In case of an incoming stream channel hit, the controller enables grant signal, indexes to the requested CL, stores the virtual address, and changes to ACTIVE REQ state. Otherwise, miss signal is enabled and the state remains on IDLE. If there is no incoming stream channel match for any of the active stream channels, a miss-error signal is enabled. On ACTIVE REQ state, the controller checks if the virtual address of the read request is inside the sliding window, and the word is fetched and not read. In case of no error, the controller responds with that word to the processor and sets the validity bit of the word as read. Otherwise, the controller waits till the word gets prefetched. When the word is sent to the processor, if there is another request, i.e. back to back reads, the controller stays at the same state and does the same actions as on IDLE state, otherwise it returns to IDLE state. In case that we are on ACTIVE REQ state and the word is either positioned outside the sliding window, or fetched and read, the controller returns to IDLE state and a word-error signal is enabled. Currently, the design does not support error handling for any of the cases mentioned previously.

3.3 Prefetching

The prefetcher of an incoming stream channel brings data from the remote peer (end-point) of the stream whenever there is space available in the circular buffer. We consider as available space the state of a CL when its words are not fetched, and there is no active outstanding request for that CL (Free state). The prefetcher essentially issues remote read operations. As shown in Figure 3.4 and Figure 3.5, in our current implementation, the prefetcher issues AXI read requests for the local DRAM or an outgoing stream cache. In future implementations, these requests could be transported over another network.

On Address Read request channel, the prefetcher issues a read request when the next CL pointed after the lastRequested pointer is empty. The identifier (ID) of the read request is the unique ID of the incoming stream channel, for the network handler to forward correctly the response, and the index of the next CL pointed after the lastRequested pointer. The address of the read request is the one saved at remoteBase pointer. When the request gets acknowledged by the remote peer, the

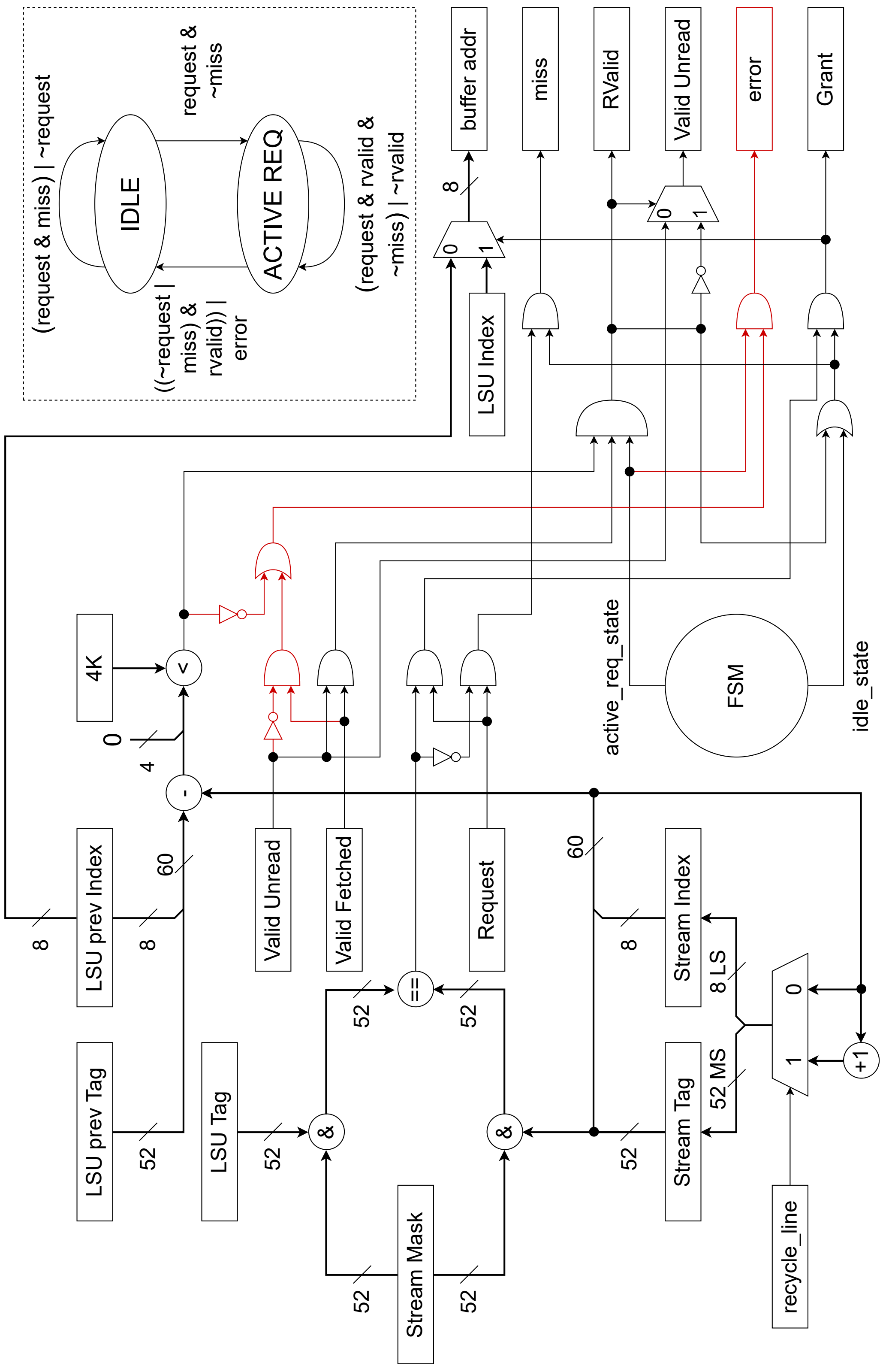


Figure 3.3: Incoming Stream Channel Controller.

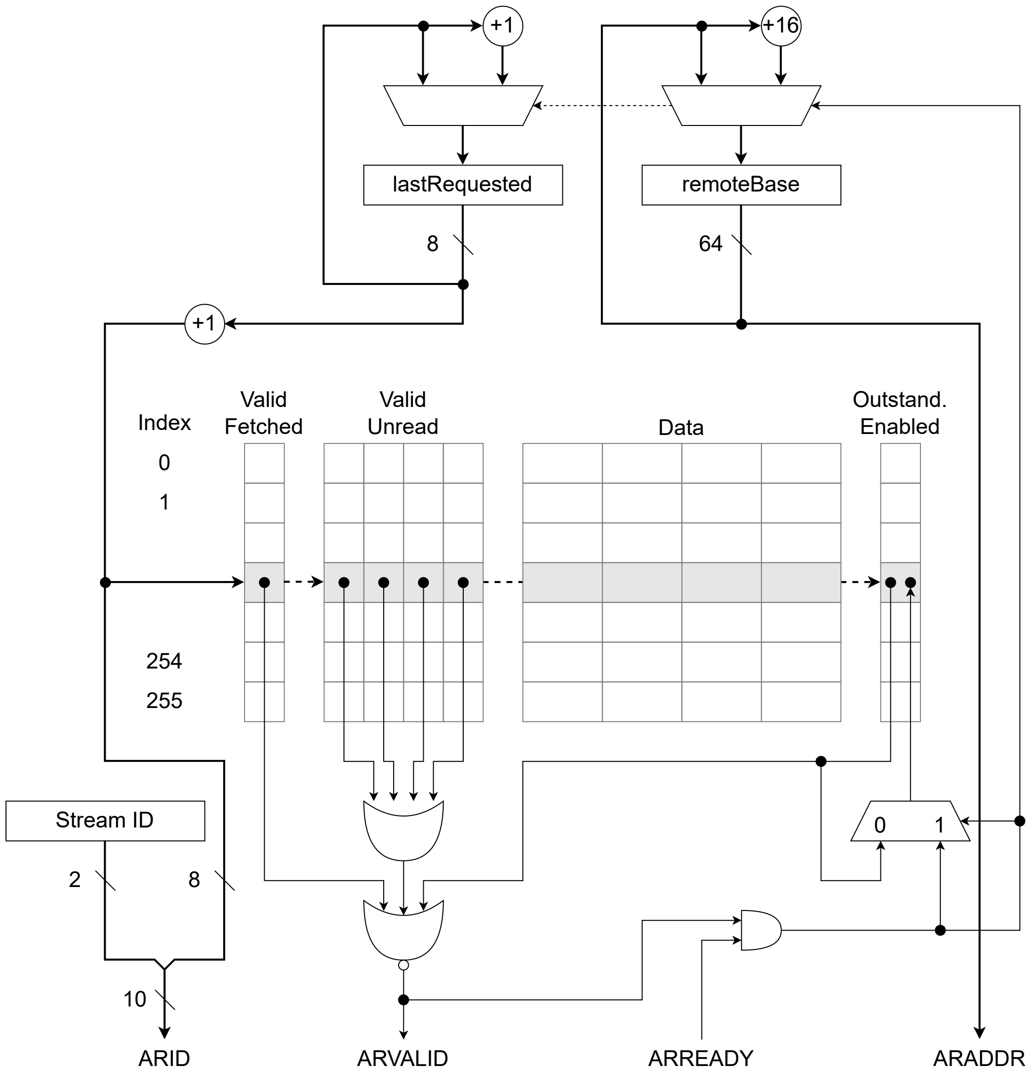


Figure 3.4: Data request diagram of Incoming Stream Channel. At the bottom of the figure we depict the AXI4 Master Read request channel signals that are issued to the memory subsystem of the ARIANE, by checking if the line after lastRequested has no unread words and no outstanding request. ARID stores both incoming stream channel ID and the index of the cacheline.

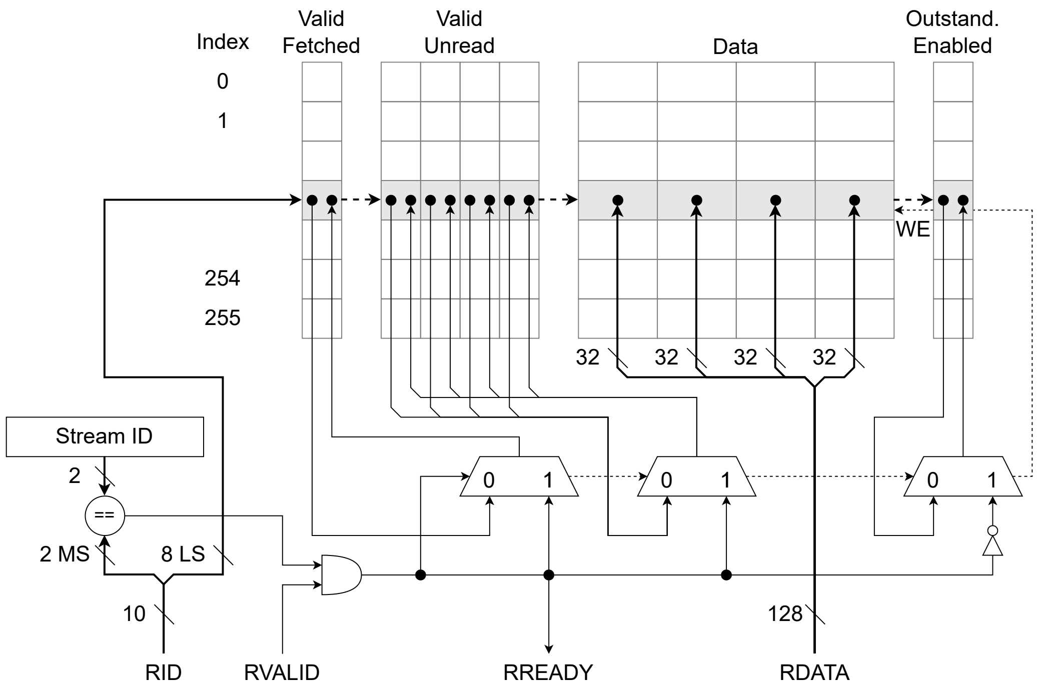


Figure 3.5: Data response diagram of Incoming Stream Channel. At the bottom of the figure we depict the AXI4 Master Read response channel signals coming from the memory subsystem of ARIANE. The controller therefore sets all validity bits to 1, closes the outstanding request and stores the data, when stream channel hit.

Outstanding Enabled bit gets enabled, and both lastRequested and remoteBase pointers increase.

On Read response channel, the prefetcher waits for a read response from the remote peer. When a response arrives and the response is intended for that incoming stream channel (i.e. the 2 MS-bits of the response ID are same as the unique ID of the incoming stream channel), the incoming stream channel **a)** indexes the correct CL based on the 8 LS-bits of the response ID, **b)** enables all the validity bits of that CL, **c)** stores the incoming data, **d)** deactivates the Outstanding Enabled bit, and **e)** sends an acknowledgement to the remote peer that it has received the read response. Also, it forwards the index and the incoming data for one cycle, in case the processor requests them on next cycle, where the validity bits would have been correctly updated.

3.4 Bigger packets (AXI Burst)

For the purpose of supporting bigger network packets than the size of a CL (where the size of a CL is equal to the network's data width), our design also supports multiple outstanding AXI data burst requests. Each data burst request is for 64Bytes (parametric), i.e. packet size of four cache lines. To support that feature, the controller has to handle quadruplet consecutive CLs' validity bits, update the pointers with quadruplet step, and has to include a 2-bit Outstanding Counter every 4 CLs, together with the Outstanding Enabled bit that those lines share, to know exactly for which CL the upcoming responses are intended to, because the base address of the requested data packet points to the first CL only. On AXI burst, the data responses arrive in order, but not necessary back to back. Other than that, the control logic remains almost the same. There is another difference on the response channel, where the new index is the 6 MS-bits of the 8 LS-bits of the response ID, together with the value of the Outstanding Counter that the new index points to. Furthermore, when a valid response arrives for a specific stream channel, in addition to what described earlier, the stream channel has also to increase the indexed Outstanding Counter of the intended CLs, while the Outstanding Enabled bit for those CLs gets disabled only when the last data burst response arrives.

3.5 Operations on Addresses

On address operations, like on additions for remoteBase pointer and stream channel address, some of the MS-bits have to remain static to uniquely identify the stream channel, e.g. if the virtual space for a stream channel is 4GBytes (parametric), the 32 MS-bits bits of the address have to remain static and only the 32 LS-bits change, to support wrap-around every 4GBytes.

3.6 LSU and network interfaces

The LSU of the core and the network communicate with the stream channels of a Stream Cache (either Incoming or Outgoing) through a single interface each. When a data request arrives from either interface, all the stream channels receive it, but only one handles it and responds back.

A unique case for only the Incoming Stream Cache is that the Load Unit of ARIANE can issue outstanding read requests. That can happen because the Load Unit can request new data before it gets the previous data response. So, with the addition of an Incoming Stream Cache with four incoming streams, a maximum of six read requests can be issued, one to the main data cache, four to the incoming stream cache, and one pending. For that reason, other than the Load Unit, the Incoming Stream Cache has to include logic to respond back to Load Unit with the order that the read requests arrived. To support that, a small queue is included, which has size equal to the number of the incoming stream channels that the Incoming Stream Cache is supporting, and only the IDs of the incoming stream channels need to be pushed there. When a data request arrives and an incoming stream channel grants the request, we push in the queue the ID of the stream channel. That way, we accomplish to allow only the incoming stream channel that is pointed by the head of the queue to respond back to the core. When that incoming stream channel responds with the data to the core, we pop its ID from the queue and the next incoming stream channel can respond back to the core, in case there were not back-to-back data requests from that same stream channel, otherwise its ID gets pushed again.

Chapter 4

Outgoing Stream Cache

In this chapter, we present the hardware implementation of the Outgoing Stream Cache that is located next to the L1 Cache of ARIANE. The Outgoing Stream Cache consists of multiple outgoing stream channels, where each of them can send data to a (remote) node. It utilizes three interfaces for communication that all the outgoing stream channels share, one being the LSU inteface for communication with the core, and the other two being the AXI interfaces for network communication, one Write Master for sending write requests to a (remote) memory and one Read Slave for seding read responses to an Incoming Stream Cache. Below, we will present the design of a single outgoing stream channel. As with Incoming Stream Cache, Outgoing Stream Cache supports also AXI Burst, and it works exactly as we described in Section 3.4. Furthermore, the operations on addresses are exactly the same as described in Section 3.5, while the three interfaces of the Outgoing Stream Cache share the same utilization as the interfaces of the Incoming Stream Cache that is described in Section 3.6.

4.1 General Description

Similar to incoming stream cache, the outgoing stream cache consists of multiple circular buffers that support multiple concurrent stream channels. In Figure 4.1, we present how an outgoing stream channel handles the store instruction request from the processor and responds when all the necessary signals are valid to store the data.

For each cache line, we have a Valid Sent bit, to indicate whether the corresponding words have been sent successfully to the network (either in Sent or in Accepted state). In addition, we maintain additional state per word to indicate whether the words have been already written by the processor (per-word validity bits). Furthermore, there is an Outstanding Enabled bit per CL, to indicate whether a write request has been issued and we wait for a response for the specific CL. As soon as all words in a line have been written, the stream channel is able to send that line to the network. When the packet has been sent successfully, the line

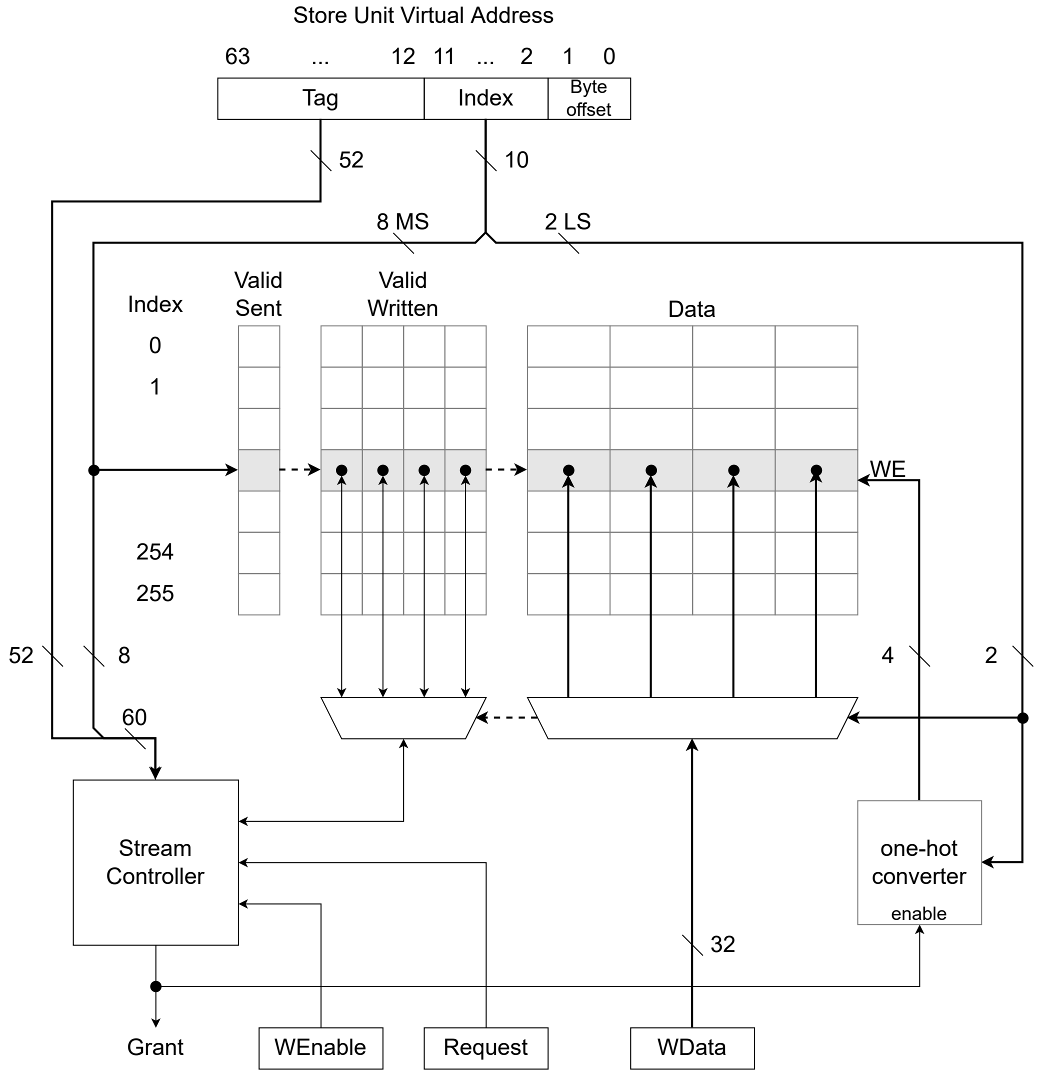


Figure 4.1: Store instruction diagram for Outgoing Stream Channel. The Address is virtual (untranslated) and is issued by the LSU unit of the ARIANE. The tag in the address field is compared with the stream channel tag of the four stream channels. The Grant signal at the bottom is awaited by the ARIANE Load-Store-Unit.

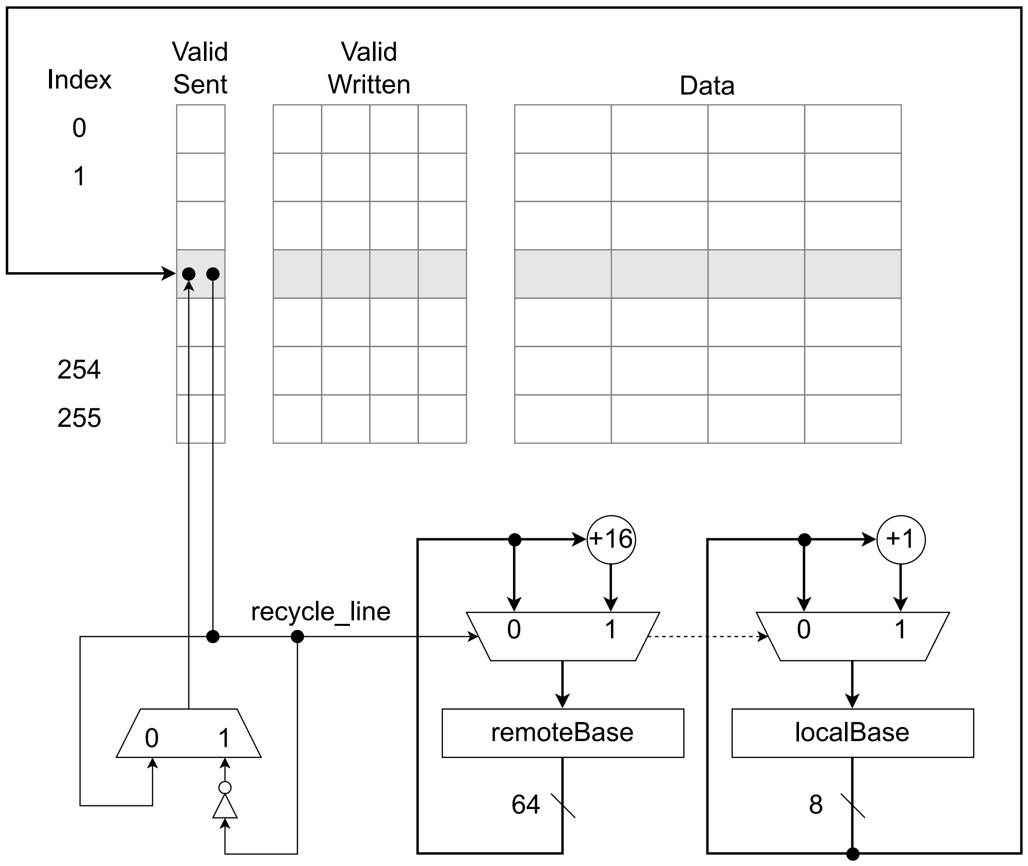


Figure 4.2: Recycling diagram for Outgoing Stream Channel. localBase pointer moves when all the data in the cacheline that it points have been written sent successfully. Also, the remoteBase pointer moves together with localBase pointer.

can be recycled, i.e. can be used for future data in the stream. In that scenario, the basePointer (with remoteBase pointer) and the sliding window move by one line, like in Figure 4.2.

4.2 Control Logic for communication with the core

Each outgoing stream channel has a controller, as shown in Figure 4.3. When the controller receives a write request, it checks if the request is for that stream channel based on the virtual address that came with the request. If the virtual address of the write request is inside the sliding window and there is no written word there, the controller enables grant signal to inform the processor that the store was successful and sets the validity bit of the word as written. Also, we save the index for one cycle, in case that the outgoing logic (i.e. Write-Combiner) needs to send the specific cache line to the remote peer if whole cacheline has valid written words. In case that the write request is intended for that stream channel and it is either outside the sliding window, or already written, the controller enables word-error signal. In case that the write request is intended for another stream channel, miss signal is enabled. If there is no outgoing stream channel match for any of the active stream channels, a miss-error signal is enabled. Currently, the design does not support error handling for any of the cases mentioned previously.

4.3 Write Combining

The write combiner of an outgoing stream channel sends data to the remote peer (end-point) of the stream whenever there are available words in the circular buffer. The write combiner essentially issues remote write operations. There are two modes for the controller of write combiner, based on the type of the remote peer. One is intended for the outgoing stream channel to communicate with a memory, as shown in Figure 4.4 and Figure 4.5, while the other is intended for the outgoing stream channel to communicate with an incoming stream cache, as shown in Figure 4.6 and Figure 4.7. In our current implementation, the write combiner issues AXI write requests for the local DRAM, while it issues AXI read responses for an incoming stream cache. In future implementations, these requests could be transported over another network.

4.3.1 Control Logic for communication with a Memory

On Write request channel, the write combiner issues a write request, based on the first CL that has all words ready to be sent. To find the index of that CL, we use a Queue that has the index of a CL pushed when a write request from the core has written the last word and made the CL ready to be sent. A CL is considered ready for its words to be sent when all words are Valid Written, and there is no pending or finished outstanding request for that CL, i.e. Outstanding Enabled

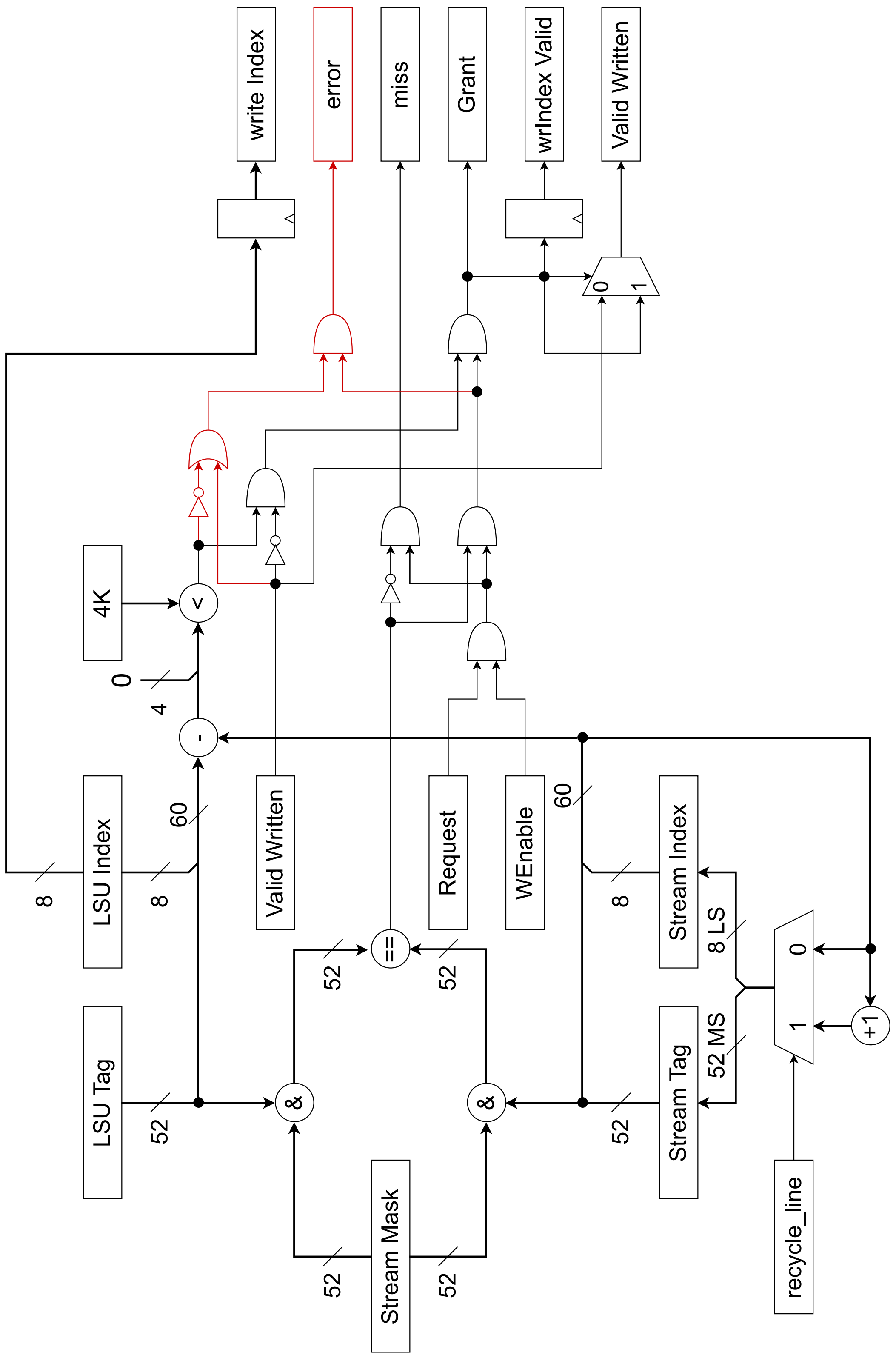


Figure 4.3: Outgoing Stream Channel Controller.

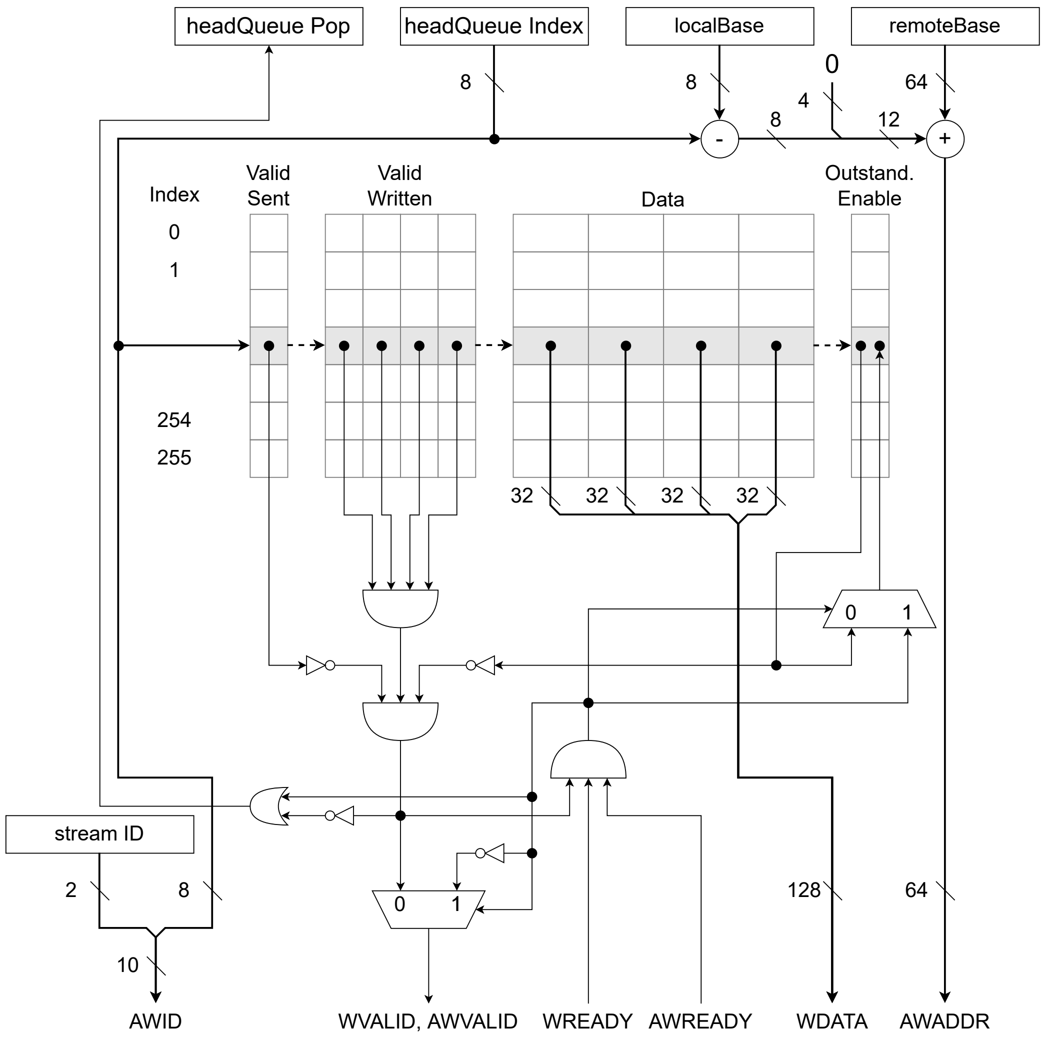


Figure 4.4: Data write request diagram of Outgoing Stream Channel communicating with a memory. At the bottom of the figure we depict the AXI4 Master Write request channel signals that are issued to the memory subsystem of the ARIANE. The controller uses a Queue to find the first available line that is ready to be sent to the network. When packet is sent successfully, opens the outstanding enable request. AWID stores both stream channel ID and the index of the cacheline.

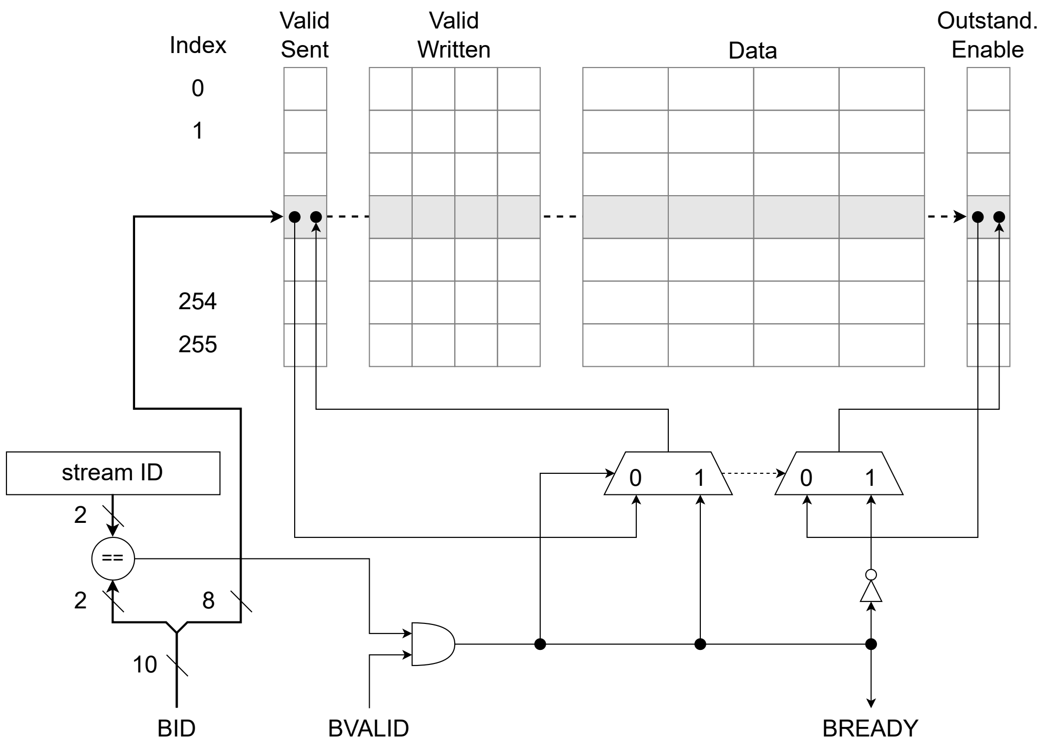


Figure 4.5: Data write response diagram of Outgoing Stream Channel communicating with a memory. At the bottom of the figure we depict the AXI4 Master Write response channel signals coming from the memory subsystem of ARIANE. When response arrives, closes the outstanding enable request.

and Valid Sent bits are not active. The ID of the write request is the unique ID of the outgoing stream channel, for the network handler to forward correctly the response, and the index of the CL. The address of the write request is the difference of localBase minus the index saved in the head of the Queue, shifted left by four, and added at remoteBase pointer. When the request gets validated by the remote peer, the Outstanding Enabled bit gets active and we pop the head of the Queue to fetch a new index for a new write request.

On Write response channel, the write combiner waits a write response from the remote peer. When a response arrives and the response is intended for that outgoing stream channel (i.e. 2 MS-bits of the response ID are same as the unique ID of the outgoing stream channel), the outgoing stream channel **a)** indexes the correct CL based on the 8 LS-bits of the response ID, **b)** enables the Valid Sent bit, **c)** deactivates the Outstanding Enabled bit, and **d)** sends an acknowledgement to the remote peer that it has received the write response.

4.3.2 Control Logic for communication with an Incoming Stream Cache

On Address Read request channel, the write combiner waits a write request from the remote peer. When the request arrives, the controller checks if that request is intended for that outgoing stream channel. It does that by checking if the difference of the requested address minus the remoteBase pointer is smaller than the sliding window. If the request is intended for that stream channel, it indexes the correct CL based on the 8 LS-bits of the request ID, i.e. the incoming and outgoing stream channels are aligned, activates the Outstanding Enabled bit, and saves the 2 MS-bits of the request ID as long as the outgoing stream channel is enabled, i.e. for the remote peer to identify later if the incoming response is intended for it. Also, we save the index for one cycle, in case that the outgoing logic (i.e. Write-Combiner) needs to send the specific cache line to the remote peer if whole cacheline has valid written words.

On Read response channel, the write combiner issues a read response, based on the first CL that has all words ready to be sent and is requested by the remote peer. To find the index of that CL, we use a Queue that has the index of a CL pushed when either a write request from the core has written the last word in an already requested CL and made the CL ready to be sent, or the CL has already all its words written and a read request for that CL arrived. A CL is considered ready for its words to be sent when all words are Valid Written, and there is pending outstanding response for that CL, i.e. Outstanding Enabled bit is active. The ID of the read response is the 2 MS bits that were saved on remote ID register and the 8 LS bits that the head of the Queue is currently indexing. When the response gets validated by the remote peer, **a)** the Outstanding Enabled bit gets deactivated, **b)** the Valid Sent bit gets enabled, and **c)** we pop the head of the Queue to fetch a new index for a new read response.

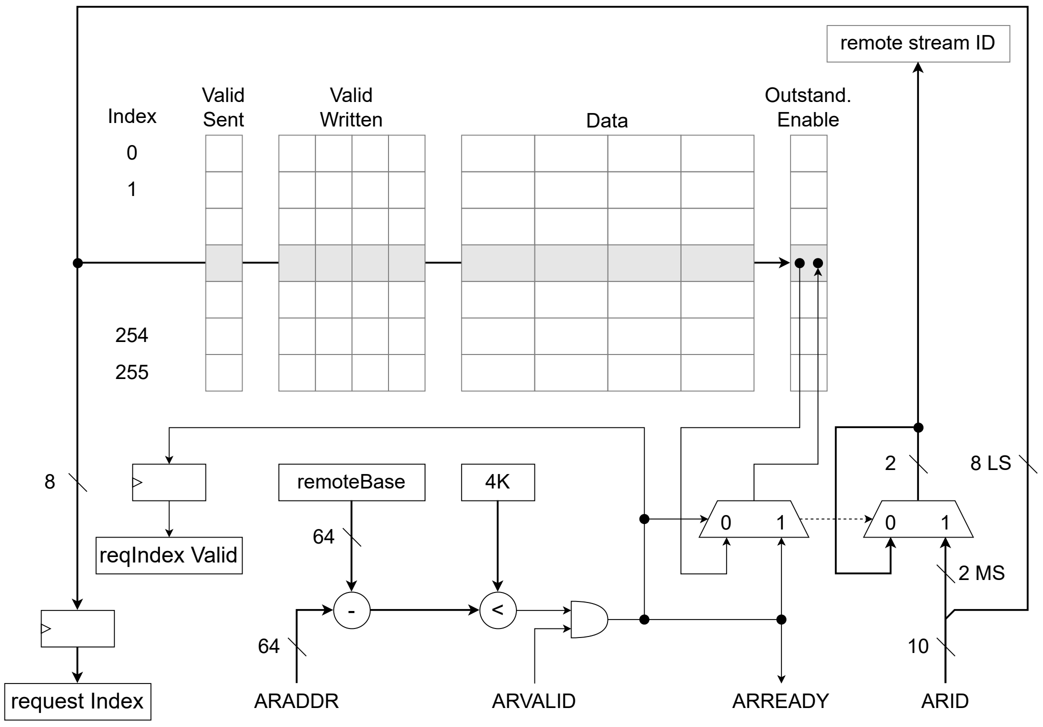


Figure 4.6: Data request diagram of Outgoing Stream Channel communicating with the prefetcher of an Incoming Stream Cache. At the bottom middle and right of the figure we depict the AXI4 Slave Read request channel signals that are issued to the memory subsystem of the ARIANE, by storing the request info, the remote stream channel ID as long as the outgoing stream channel is active, and the requested index for one cycle. Request has to be inside the sliding window of the stream, otherwise the request is intended for another outgoing stream channel (or an error).

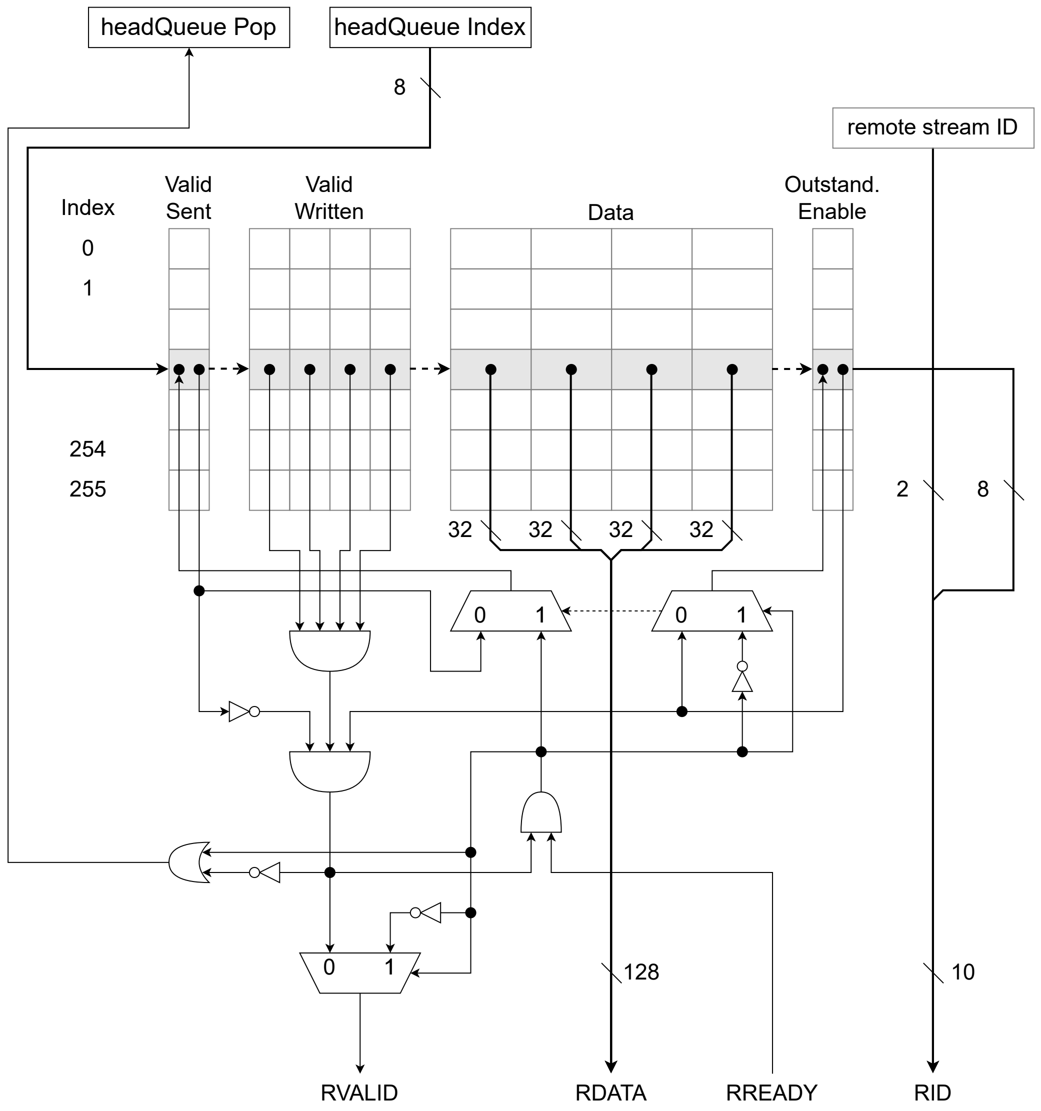


Figure 4.7: Data response diagram of Outgoing Stream Channel communicating with an Incoming Stream Cache. At the bottom of the figure we depict the AXI4 Slave Read response channel signals coming from the memory subsystem of ARIANE. The controller uses a Queue to find the first available line that is ready to be sent to the network. When packet is sent successfully, closes the outstanding enable request and sets the validity bit of the line as sent.

Chapter 5

Load Store Unit Extensions

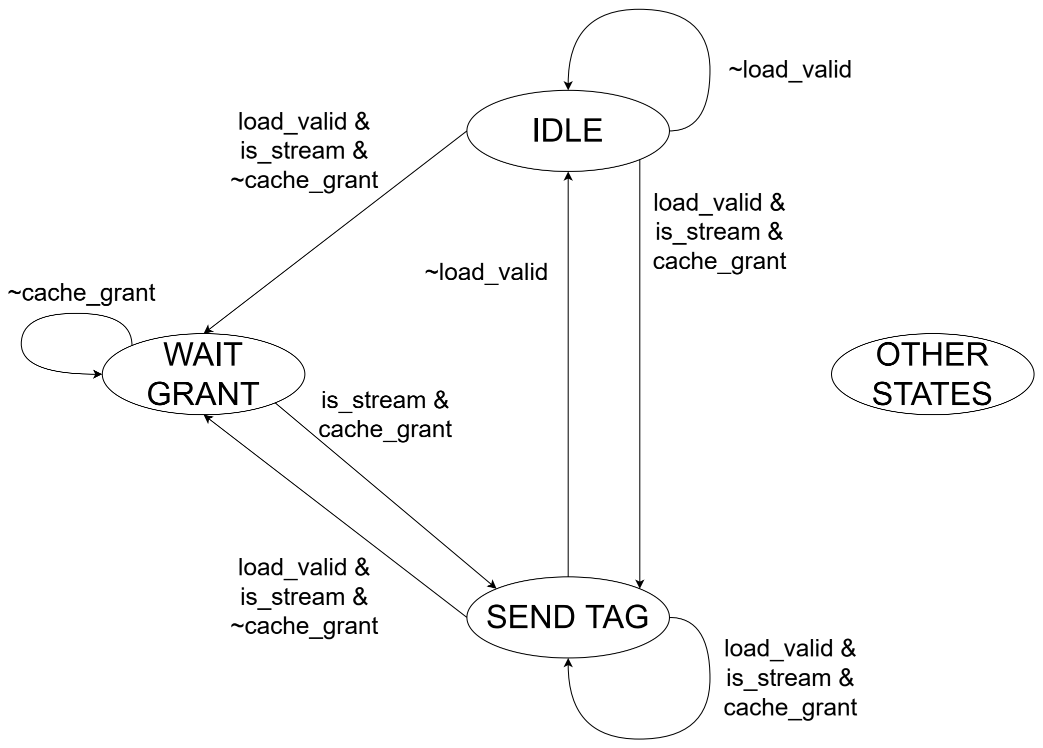
In this chapter, we present the extensions that we included on the Load/Store Unit of ARIANE, to support our protocol. Below, we will describe the changes that we made on each of the control logic of Load and Store Units of ARIANE, first on their FSMs and their state transitions, and second on the two new interfaces that we included for our Streaming Caches to communicate with the core and their signal handling.

5.1 Load Unit

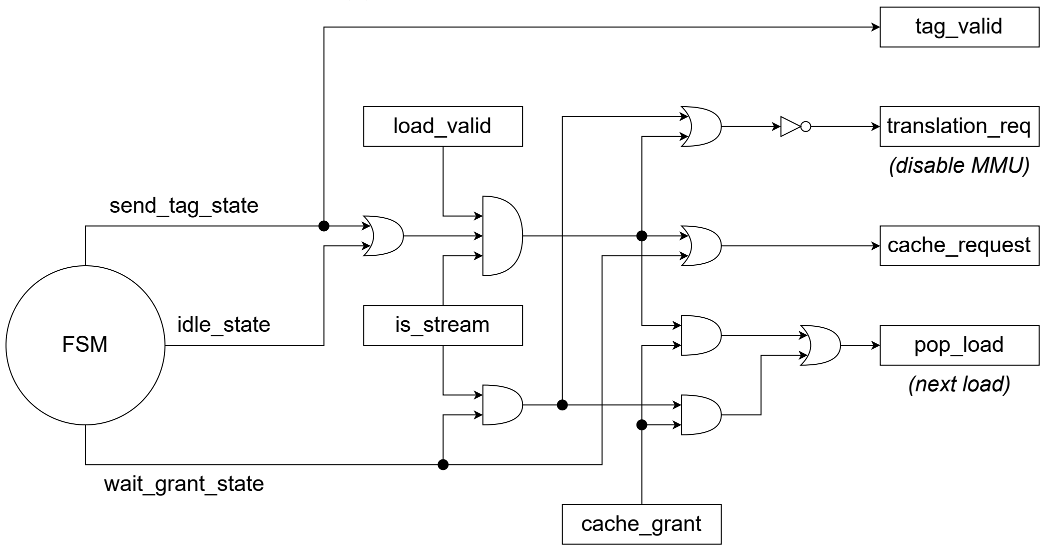
Load Unit controller on ARIANE core utilizes a FSM to control the load request/response flow. For the controller to support also our Incoming Stream Cache, we didn't include any new states, but only changed the flow logic on how to move from one state to another, as shown in Figure 5.1.

On IDLE state, if there is a valid load instruction, the controller recognizes if it is a load request for common or stream data, by checking if the virtual address is inside the stream address region or not. In case that the virtual address of the load request is not inside the stream address region, the flow continues as it was originally implemented. Otherwise, the controller disables the address translation request for the MMU and sends a cache request to incoming stream cache. In case that the incoming stream cache doesn't respond with grant signal enabled, the controller changes to WAIT GRANT state, i.e. previous load request is still active in the cache subsystem. Otherwise, the controller enables the load queue to fetch the next load instruction and changes to SEND TAG state.

On WAIT TAG state, the controller waits a grant from the cache subsystem. In case that the virtual address of the load request is not inside the stream region, the flow continues as it was originally implemented. Otherwise, the controller continues to disable the address translation request for the MMU and keeps enabled the cache request signal for the incoming stream cache. When the cache subsystem's grand signal arrives, the controller enables the load queue to fetch the next load instruction and changes to SEND TAG state.



(a) Load Unit FSM extensions.



(b) Load Unit datapath extensions.

Figure 5.1: Load Unit Controller extensions for supporting stream data. This diagram shows the extensions that we implemented on the existing design of the Load Unit Controller to support our Incoming Stream Cache. On IDLE we wait for a new load instruction, on WAIT GRANT we wait the last load instruction request to get granted, and on SEND TAG we can receive a new load instruction.

On SEND TAG state, the controller enables the tag valid signal and checks if there is another valid load instruction. In case that there isn't another valid load instruction, it changes to IDLE state. Otherwise, it does exactly the same things as on IDLE state.

In case that there are back to back loads intended for different Caches (either the Main Data Cache or the Incoming Stream Cache), the Load Unit waits the last data response to arrive before enabling the new data request. We implemented it that way, because the Load Unit can issue outstanding data requests. The issue with the multiple outstanding data requests is that as long as the data request for a common load is active, it can be killed at any time before the data grant is received, in case that the page offset matches with the one found in a store instruction that is currently active in the Store Buffer of the Store Unit. For that reason, each controller of either of the Caches can accept maximum one data request from the Load Unit.

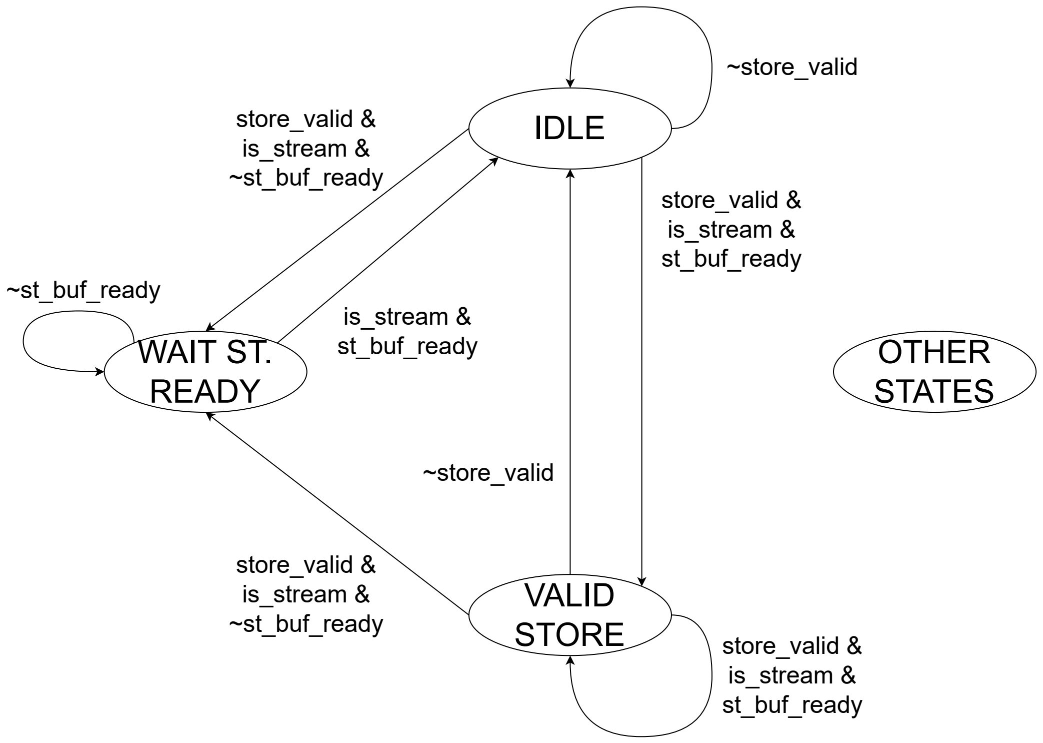
5.2 Store Unit

Store Unit controller on ARIANE core utilizes a FSM and a Store Buffer to control the store request/response flow. For the controller to support also our Outgoing Stream Cache, we didn't include any new states, but only changed the flow logic on how to move from one state to another, as shown in Figure 5.2. Furthermore, we included more elements on each line of the speculative and commit queue buffers, to save also the virtual address of the store instruction, as shown in Figure 5.3.

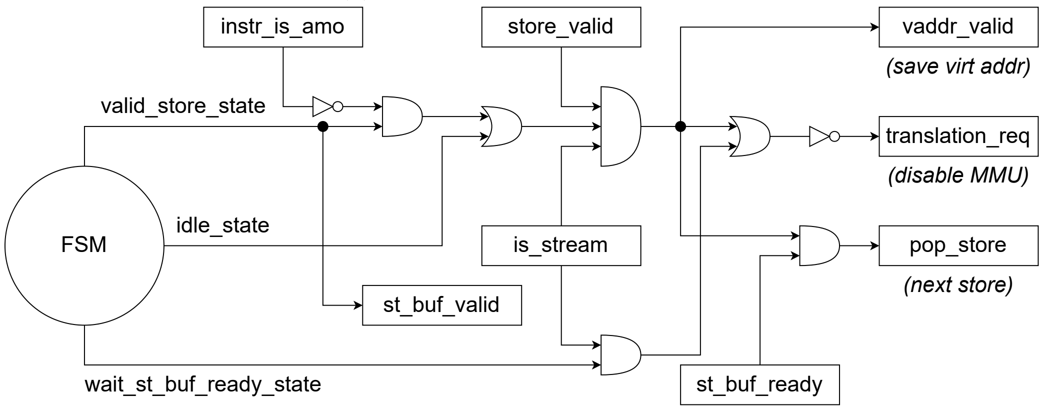
On IDLE state, if there is a valid store instruction, the controller recognizes if it is a store request for common or stream data, by checking if the virtual address is inside the stream address region or not. In case that the virtual address of the store request is not inside the stream address region, the flow continues as it was originally implemented. Otherwise, the controller disables the address translation request for the MMU and checks if the speculative queue has space. In case that it hasn't space, the controller changes to WAIT STORE BUFFER READY state, i.e. wait till store buffer ready signal is enabled. Otherwise, the controller enables the store queue to fetch the next store instruction and changes to VALID STORE state.

On WAIT STORE BUFFER READY state, the controller waits the speculative queue to have available space. In case that the virtual address of the store request is not inside the stream address region, the flow continues as it was originally implemented. Otherwise, the controller continues to disable the address translation request for the MMU. When the speculative queue has available space, the controller returns to IDLE state.

On VALID STORE state, the controller enables the store buffer valid signal, i.e. store on speculative queue the store instruction, and checks if there is another valid store instruction. In case that there isn't another valid store instruction or it is an Atomic Memory Operation, it changes to IDLE state. Otherwise, it does


~store_valid
IDLE
store_valid & is_stream & ~st_buf_ready
store_valid & is_stream & st_buf_ready
~st_buf_ready
WAIT ST. READY
is_stream & st_buf_ready
OTHER STATES
~store_valid
store_valid & is_stream & ~st_buf_ready
VALID STORE
store_valid & is_stream & st_buf_ready

(a) Store Unit FSM extensions.



(b) Store Unit datapath extensions.

Figure 5.2: Store Unit Controller extensions for supporting stream data. This diagram shows the extension that we implemented on the existing design of the Store Unit Controller to support our Outgoing Stream Cache. On IDLE we wait for a new store instruction, on WAIT STORE BUFFER READY we wait till the store buffer has available space, and on VALID STORE we can push the last store instruction request to the store buffer and receive a new store instruction.

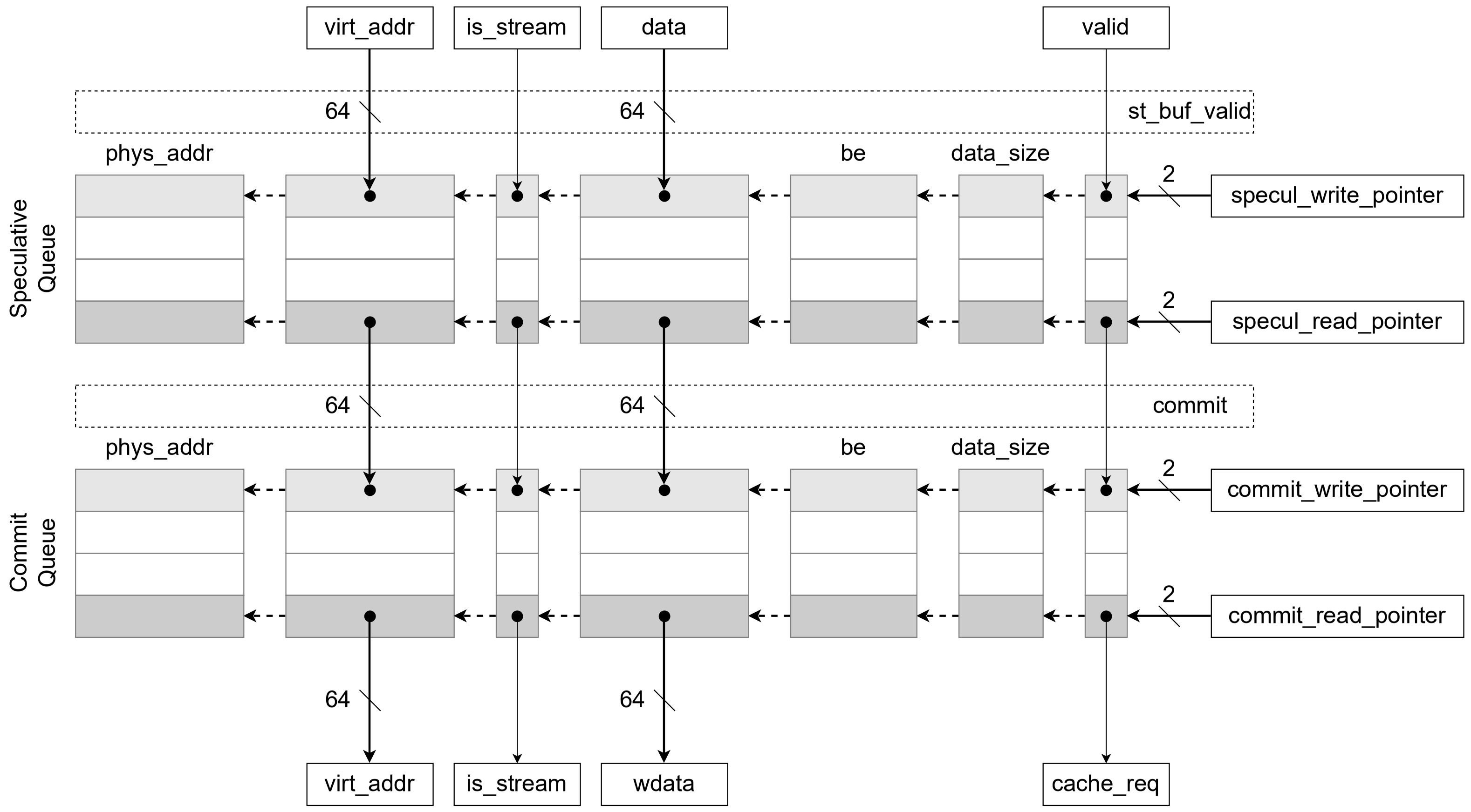


Figure 5.3: Store Buffer extensions for supporting stream data. In case of a store instruction intended for Outgoing Stream Cache, we have to also store the virtual address of the store instruction to send it together with the data.

exactly the same things as on IDLE state.

The store buffer works as following. When there is space in speculative queue and a store buffer valid signal arrives, the store buffer stores the store instruction in the line that is pointed by the speculative write pointer. When the ARIANE core issues a store commit, the store buffer moves the store instruction, which is pointed by the speculative read pointer, from the speculative queue to the line that is pointed by the commit write pointer on commit queue, and invalidates the store instruction on speculative queue. As long as the commit read pointer of the commit queue points to a valid store instruction, it issues a store data request to the cache subsystem. When the cache subsystem responds with the data grant signal enabled, the store buffer invalidates the store instruction on commit queue that is pointed by the commit read pointer.

Chapter 6

FPGA Implementation and Design Evaluation

In this chapter, we synthesize the RISC-V core with the extensions that we included to support our Streaming Caches. We present and evaluate the utilization and timing overheads of our implementation for a Zynq UltraScale+ MPSoC FPGA, and lastly we evaluate our design with some bare metal programs.

6.1 Time and Space Overheads

We have compiled our design, including the Incoming Stream Cache, with four (4) incoming stream channels (i.e. circular buffers), and the Outgoing Stream Cache, again for four (4) stream channels, using Vivado 2020.1. The target FPGA was the Xilinx Zynq UltraScale+ MPSoC (xczu9eg-ffvc900-2-e) [4].

As shown in Table 6.1, our design required two 16 two-ported BRAMs (two blocks per incoming or outgoing stream channel), 40445 LUTs, and 16121 CLB registers in total for the two Streaming Caches, while the changes on LSU were minimal. The reason for the big area utilization is that we use a lot of registers for the validity bits, where their control logic has created a very big Net List. In our future implementation, we will replace those registers with BRAMs. The issue that we will have to overcome is that our design can simultaneously read and write the validity bit arrays a maximum of four times for each operation per cycle, to support the control logic of the four registers that make changes in specific areas of each stream channel. One solution for that is to create four copies of the validity bits to support the four concurrent reads per cycle, but for the writes, we have to encode the validity bit buffers in a way that we can make more than one operation per cycle to all the copies. That needs careful redesigning of our logic to avoid deadlocks and unnecessary delays.

Furthermore, we accomplished to run on Synthesis the Incoming Stream Cache at 275MHz and the Outgoing Stream Cache at 210MHz.

Name	**Utilization**		
	CLB LUTs	***CLB Registers***	***Block RAM Tile***
xczu9eg-ffvc900-2-e	274080	548160	912
Original ARIANE	24350	17088	37
Load/Store Unit	3387	5120	0
Load Unit	162	22	0
Store Unit	786	1804	0
Store Buffer	629	1593	0
Cache Subsystem	5848	1184	37
L1 Cache	5010	1018	25
Extended ARIANE	66401	33445	53
Load/Store Unit	4119	5356	0
Load Unit	373	87	0
Store Unit	981	1975	0
Store Buffer	718	1700	0
Cache Subsystem	46380	17360	53
Incoming Stream Cache	16839	7506	8
Outgoing Stream Cache	23606	8615	8

Table 6.1: Space Utilization of Streaming Caches. The big area that our Streaming Caches take is because of the many registers and their control logic that we use for the validity bits.

6.2 Preliminary Evaluation

We have conducted a small number of micro-tests using the ARIANE core with our Streaming Caches in behavioral simulation. In our experiments, both Main Data Cache and Streaming Caches are connected to a local Memory through an AXI interconnect, as shown in Figure 6.1. The Main Data Cache communicates only with the Memory through its AXI Master Adapter that has 64-bit data width, by requesting one cacheline of 16 Bytes per cache miss (i.e. packet size equal to 2 AXI Bursts of 64-bit). The Incoming Stream Cache communicates with both the Memory and the Outgoing Stream Cache through its AXI Read Master Adapter that has 128-bit data width. The Outgoing Stream Cache communicates with the Memory through its AXI Write Master Adapter and with the Incoming Stream Cache through its AXI Read Slave Adapter, where both of them have 128-bit data width. All stream channels of Streaming Caches request or receive packets equal to 64 Bytes (i.e. 4 cachelines/AXI Bursts of 128-bit).

In our testbench, we consider that the core has a clock speed of 1.5GHz. Furthermore, each memory access bears a latency of 100ns (150 cycles), as the miss

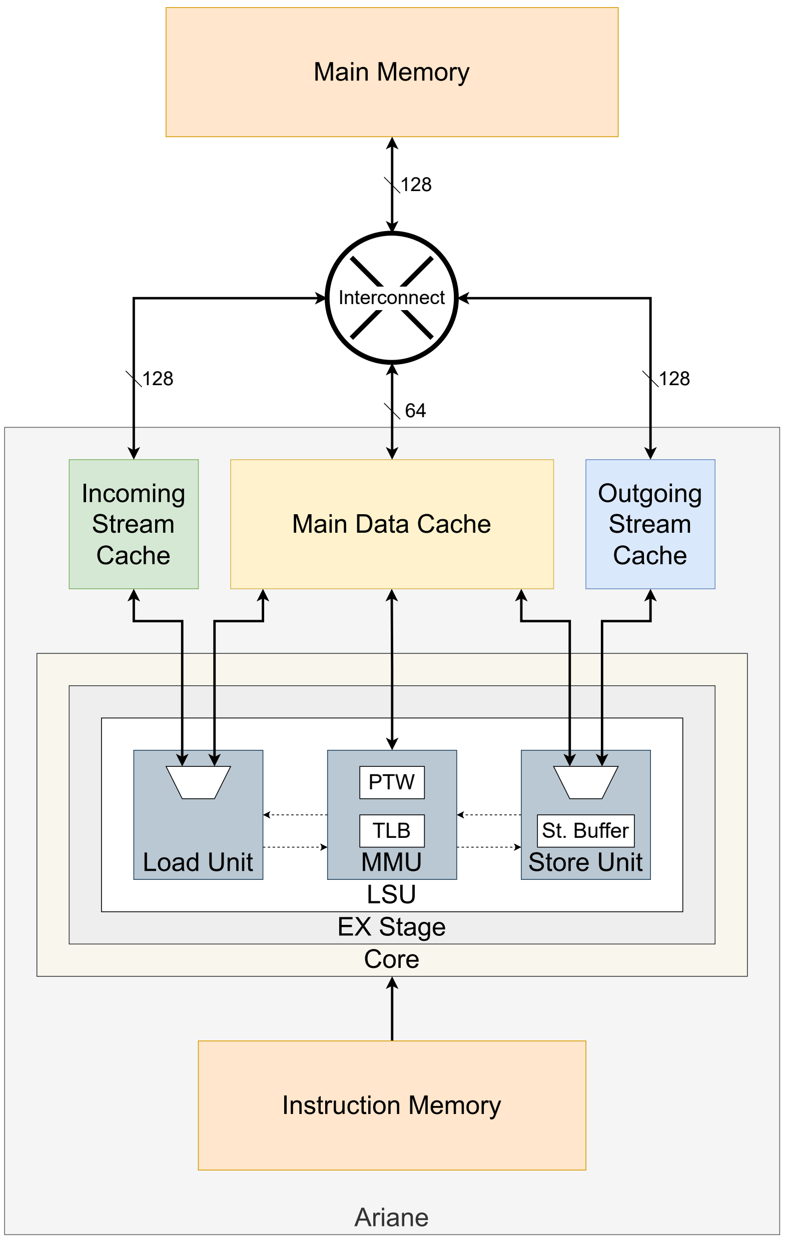
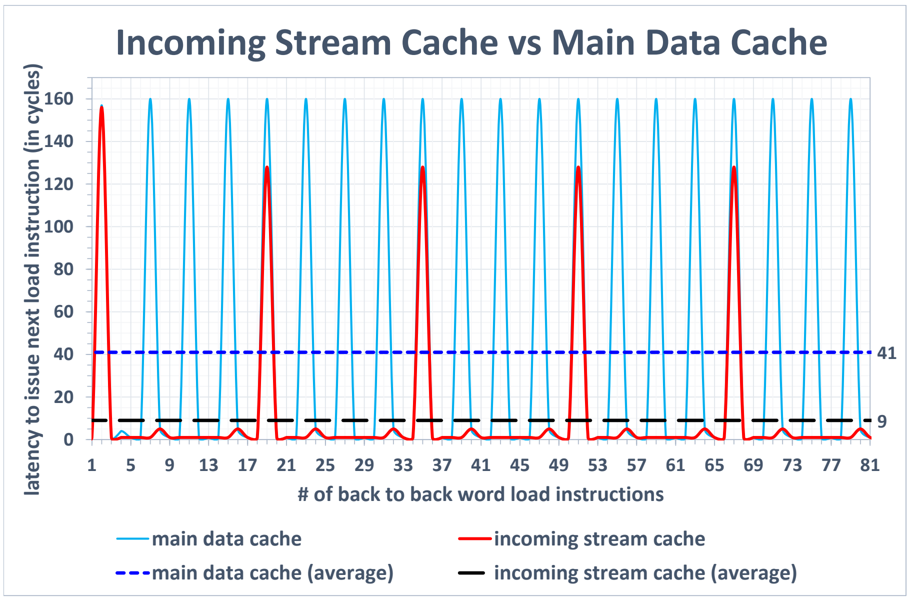
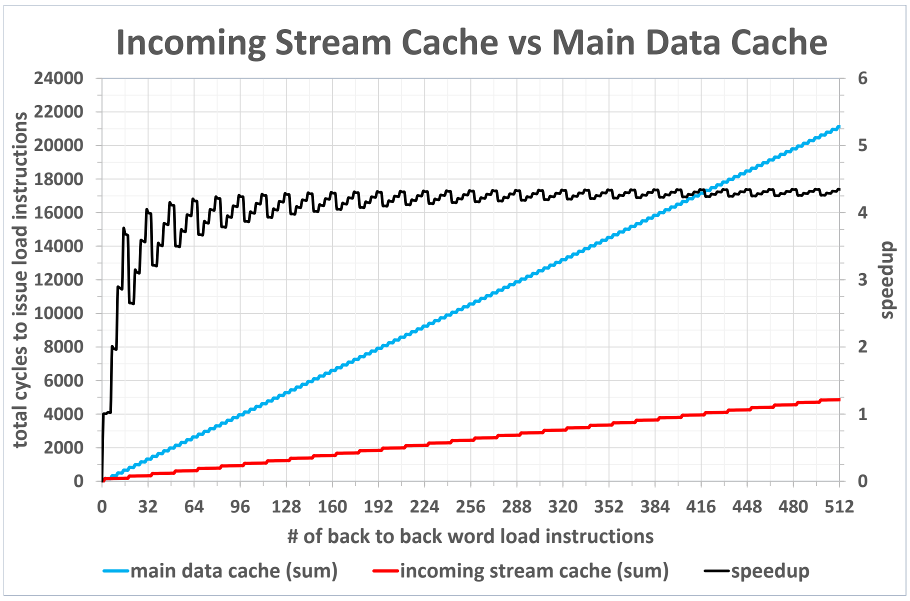


Figure 6.1: Testbench Overview of ARIANE with the Streaming Caches. Either of the Load or Store Units can send data requests to either Main Data Cache or the corresponding Streaming Cache, based on the virtual address of the instruction. All the Caches are connected to an AXI Interconnect that communicates with a Main Memory. The Main Data Cache has an AXI Master Adapter with 64-bit data width for communication the AXI Slave Adapter of the Main Memory, while Streaming Caches use an AXI Master and an AXI Slave Adapter with 128-bit data width for communication with each other or the Main Memory.



(a) Latency to issue next back-to-back word load instruction.



(b) Total latency for specific amount of back-to-back word load instructions.

Figure 6.2: Incoming Stream Channel comparison with Main Data Cache for back-to-back word load instructions. Main Data Cache accesses the memory every four words to fetch one cacheline, while Incoming Stream Channel fetches the words before they are requested by the core, except on the first load that the incoming stream was just initialized.

unit (of the main data cache), the data prefetcher (of the incoming stream cache) and the data write-combiner (of the outgoing stream cache) try to access the Main Memory.

On the *incoming communication*, we enable only one of the four incoming stream channels (where each one has 4 KBytes stream buffer size) to fetch 16 KBytes of data for the core to consume. As shown in Figure 6.2, most of the remaining bytes of the incoming stream channel are ready to be read by the core after just **1 core cycle** (i.e. back-to-back loads, otherwise it would take **2 core cycles**), as the prefetcher of the incoming stream cache manages to have them continuously present in the incoming stream buffer before the core issues a load instruction.

For comparison, if the data were loaded through the main memory hierarchy (i.e. the L1 cache connected to Main Memory), every cacheline access would incur a cache miss and thus a memory access latency, i.e. approximately 100 ns. For that reason, by using the incoming stream cache, the core manages to issue a 32-bit word load instruction every **9 core cycles** on average, while with the main data cache, the core manages to issue a 32-bit word load instruction every **41 core cycles** on average, because there is a cache miss every four 32-bit words (one cacheline).

Another observation in the results of using the incoming stream channel is that every 16 back-to-back word load instructions, there is a spike in latency, due to the network's delivering of words being slower than core's consumption. To hide such latency, we have to set the prefetcher to request bigger packets of data from the Main Memory than the 64 Bytes. So, as result, we can conclude that an incoming stream channel of 4 KBytes is too big for the parameters that we set in the testbench, especially for such demanding pattern of data consumption. From the other side, in more common patterns of consumption from the core, where there is at least one computational cycle for each word, a big sliding window could be more helpful, in case that the user program wants to read words that are way after the Base pointer (i.e. oldest data in the stream channel).

Lastly, it is important to mention that if the main data cache was supporting a prefetcher similar to the one that we utilize in Incoming Stream Cache, the results will be almost the same. But the advantage of our design is that it can communicate with a remote DRAM or another coherence island, similar to communicating with the local DRAM, without extra overheads, other than a small increase in latency due to the possible longer distance and the complexity in the interconnection.

On the *outgoing communication*, the results of the Outgoing Stream are almost the same with the ones of the Main Data Cache, where each individual store instruction takes **2 core cycles** to be written to either of the caches (otherwise **1 core cycle** for back-to-back stores). The advantage of the Outgoing Stream Cache, over the Main Data Cache, is that it utilizes a write-combiner, which releases neighboring words to the network as soon as they are written (with the exception of releasing them after they have been also requested, when communicating with an incoming stream cache). That way, in our micro-tests for back-to-back stores, we manage to avoid stalling the core comparing to the Main Data Cache, which has to write back to the Main Memory a cacheline, when there is no available space. For that reason, in an infinite stream of storing back-to-back data, when we use the Outgoing Stream Cache, the average latency to issue a store instruction is **1 core cycle**, while when we use the Main Data Cache, the average latency to issue a store instruction is equal to the memory access latency divided by four (i.e. the words that can fit in a cacheline), which is a bit less than **40 core cycles**.

Chapter 7

Related Work

7.1 Vivado High Level Synthesis Streams

Vivado High Level Synthesis (HLS) encourages hardware developers to create processes that follow the dataflow (producer-consumer) paradigm, i.e. one process only generates (produces) a set of data that another process reads (consumes). Furthermore, if the data are produced and consumed with the same access pattern, the communication channel can be declared as a stream and implemented as a FIFO. This method allows for the consumer to begin its operation without waiting for the producer to complete theirs. It also helps minimize the hardware requirements as no addressing logic is required and only a FIFO needs to be implemented, in contrast to large ping-pong buffers.

7.2 Data Streaming in inter-process CNN engines

Data between consecutive layers in Convolutional Neural Networks travel unidirectionally, following the producer-consumer paradigm and hence benefit from improved stream communication proposed in this paper. Hardware implementations focus on distributing CNN computational layers across multiple computing nodes to increase inference throughput by taking advantage of the larger hardware real estate, as shown in [5, 6, 7, 8]. Providing a reliable streaming interface for data transactions between computing nodes for such applications is important in order to avoid additional latency caused by intermittent large data transfers and minimize idle node time while waiting for data input.

7.3 Maxeler Data Flow Engine

Maxeler Technologies build upon the dataflow paradigm for Big Data applications, accelerating application data flows and loops up to 1000× compared to approaches that follow the control flow paradigm [9]. The Maxeler DataFlow Engine (DFE) contains chains of functional units that pass data to each other in a streaming manner; data also streams between the host and the DFEs for input/output [10]. Voss et al. implemented the VGG-16 convolutional neural network on a MAX-5 DFE. Their work used high-precision (18-27 bit) fixed-point data types and achieved an average throughput of 84.5 images per second and 2450 GOPS at 250 MHz clock frequency [11].

7.4 Stream-based Memory Access accelerator

This stream-specialized accelerator [12] proposes ISA extensions to take advantage on stream structure and semantics, by looking memory operations from a fine-grain view. Like our design, it focuses on the unnecessary latency of memory accesses and utilizes a prefetcher to take advantage of both latency and throughput in a localized area, from processor pipeline to cache subsystem.

Chapter 8

Conclusion – Future Work

In this thesis, we presented the design of a new approach for inter-processor stream communication across RISC-V Coherence Islands, with read-once/store-once cache policies to improve performance. We described that our design has many advantages in stream communication over other communication frameworks, like TCP/IP and RDMA interconnects, which support only memory-to-memory communication. Furthermore, we coupled our design with the ARIANE RISC-V core, next to the L1 cache, and achieved very low access time in stream communication (2 core cycles for individual load/store instructions and 1 clock cycle for back-to-back load/store instructions). Additionally, we extensively evaluated our Streaming Caches and compared them after with the Main Data Cache of ARIANE, by connecting all Caches to the Main Memory. By using hand-made microbenchmarks in behavioral simulation, our experiments showed that we accomplished great reduction in the average latency, when the core wants to access data from/to the main memory, because of the utilization of a data prefetcher in Incoming Stream Cache and a data write-combiner in Outgoing Stream Cache. Lastly, we synthesized our design on Vivado 2020.1 for a Xilinx Zynq UltraScale+ MPSoC as a target FPGA, and we achieved above 200MHz clock frequency for our Streaming Caches, but consumed a relatively big area (40445 LUTs and 16121 Registers).

This work can be continued in many different ways, so we list the most notable of them below:

1. One weakness of our Streaming Caches is the big area that they utilize on a FPGA. In a cycle, each stream channel needs multiple ports for reading and writing on validity bit arrays. Because of that, we currently implemented those arrays using registers to support the concurrent operations. One way to solve that issue is to replace the array of registers with multiple BRAMs (i.e. as many as the amount of concurrent operations) to support the concurrent reads, but another issue that arises is that we are limited to one write per cycle. For that reason, we need to either organise the priority of writes, to avoid unnecessary latency in our Streaming Caches, or split and encode the validity bit arrays in a way to support concurrent writes too.

2. Current version of the design does not support error handling, neither for programming errors (i.e. incorrect word accesses based on our policies), nor for network delays and errors, like network disconnections or noise. For that reason, timeouts need to be implemented for not stalling the network, like in cases that a response gets delayed.

3. Another thing that needs to be carefully designed and implemented is the termination of streaming channels. While the finite streams can be terminated easier by setting a counter for example, on infinite streams there is the issue of not knowing what to do with the remaining valid data on the stream buffers. Furthermore, timeouts will be needed here too, to terminate any unnecessary outstanding requests.

4. Last but not least, our design needs to be further evaluated and tested under real conditions, like on a FPGA, where ARIANE core runs with an Operating System, to fully evaluate our Streaming Caches and compare their performance with other networking frameworks, like the RDMA interconnection. Even if parametric, both the size of the stream circular buffers (i.e. sliding window) and the amount of stream channels need to be evaluated. The first based on the core's frequency and network's throughput and latency, and the latter based on the needs of distributed applications that will be executed on the nodes.

Bibliography

[1] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: user-level isa, version 2.0, eecs department. *University of California, Berkeley*, 10, 2014.

[2] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019.

[3] S Guruprasad and K Sudharshan. Design and analysis of master module for amba axi-4. *NCECS-2011) at Siliguri Institute for technology*, 2011.

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

[5] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, page 326–331, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Stylianos I. Venieris and Christos-Savvas Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47, 2016.

[7] E. Mageiropoulos, N. Chrysos, N. Dimou, and M. Katevenis. Using hls4ml to map convolutional neural networks on interconnected fpga devices. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 277–277, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[8] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S.

Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[9] Nemanja Trifunovic, Veljko Milutinovic, Jakob Salom, and Anton Kos. Paradigm shift in big data supercomputing: dataflow vs. controlflow. *Journal of Big Data*, 2(1):1–9, 2015.

[10] Exa2Pro Deliverable D4.4 – Initial report on implementation of StarPU on heterogeneous architectures p.7.

[11] Nils Voss, Marco Bacis, Oskar Mencer, Georgi Gaydadjiev, and Wayne Luk. Convolutional neural networks on dataflow engines. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 435–438, 2017.

[12] Zhengrong Wang and Tony Nowatzki. Stream-based memory access specialization for general purpose processors. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 736–749, New York, NY, USA, 2019. Association for Computing Machinery.