fDSM: An FPGA-Accelerated Distributed Shared Memory for Heterogeneous Instruction-Set-Architecture Hardware

Naarayanan Rao VSathish

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Binoy Ravindran, Chair

Edson L. Horta

Cameron D. Patterson

February 10, 2022

Blacksburg, Virginia

Keywords: Distributed Shared Memory, Sequential Consistency, Popcorn Linux, FPGA, Instruction-Set-Architecture

Copyright 2022, Naarayanan Rao VSathish

fDSM: An FPGA-Accelerated Distributed Shared Memory for Heterogeneous Instruction-Set-Architecture Hardware

Naarayanan Rao VSathish

(ABSTRACT)

Due to the diminishing relevance of Moore's Law, traditional multi-core systems are increasingly struggling to meet the computational demands of many emerging workloads. Heterogeneous computing, which involves exploiting higher degrees of parallelism (e.g., GPUs) and application-specific specialization (e.g., FPGAs), is increasingly used to meet this demand. An important architectural trend in this space involves using instruction-set-architecture (ISA) heterogeneity. An exemplar case is emerging I/O devices that include CPU cores with ISAs (e.g., ARM, RISC-V) that differ from that of host CPUs (e.g., x86) and have physically discrete memory. Shared-memory programming of such systems requires the Distributed Shared Memory (DSM) abstraction. Software DSM incurs significant OS overhead for maintaining memory coherency. Despite outperforming software predecessors, hardware DSM and cache-coherent interfaces require custom chips and lack the flexibility to experiment with different DSM consistency protocols. This thesis presents **fDSM**, an FPGA-accelerated DSM framework for ISA-heterogeneous hardware. **fDSM** implements a high-speed messaging layer to enable inter-node communication across ISA-different CPU cores and a DSM protocol processor that maintains virtual memory coherency using a multiple-reader single-writer DSM algorithm. Experimental studies reveal that **fDSM** outperforms prior art, including Popcorn Linux's software DSM abstraction, that uses TCP-IP and state-of-the-art

Infiniband RDMA messaging layers by **2.8X** and **7%**, respectively. **fDSM** also provides reconfigurability and thereby allows implementation and experimentation of different memory consistency models.

fDSM: An FPGA-Accelerated Distributed Shared Memory for Heterogeneous Instruction-Set-Architecture Hardware

Naarayanan Rao VSathish

(GENERAL AUDIENCE ABSTRACT)

Moore's Law predicts that the number of transistors in a chip will double approximately every two years. Chip vendors are increasingly observing that this law is nearing its limit when transistor sizes are shrunk to 5nm and 3nm due to power consumption and heat dissipation issues. As a result, innovation in new computing architectures has increasingly focused on heterogeneity, i.e., the use of hardware performance accelerators like graphic processors and reconfigurable logic used in confluence with a computer's CPU (host). To improve the programmability of these architectures, which usually have physically separate memory, the shared-memory programming model is usually used to provide coherent virtual memory. The shared memory model, when applied to such distributed systems, called distributed shared memory (or DSM), has been previously developed in software as well as in hardware. The former usually suffer from high latency overheads, while the latter often requires custom chips and lack programmability for implementing new memory consistency protocols. This thesis presents **fDSM**, a reconfigurable distributed shared memory framework that provides coherent shared memory between a host and a smart I/O device such as a SmartNIC. **fDSM** is implemented in FPGAs, which are increasingly available in hosts and Smart I/O devices at the commodity scale. Our prototype implementation uses ISA-heterogeneous hosts to emulate such an environment. Our experimental evaluation using applications from High-Performance Computing benchmark suites reveal that **fDSM** yields performance benefits over a state-of-the-art software DSM.

Dedication

I dedicate this thesis to my brother, Abishek

Acknowledgments

First, I would like to thank my advisor, Dr. Binoy Ravindran, for supporting me throughout my graduate career. Without his encouraging words and advice, this thesis wouldn't have been a reality.

Furthermore, I would like to thank Dr. Edson Horta for being an excellent mentor and for his valuable feedback on this thesis. His FPGA expertise helped me get through tedious hardware debugging.

Also, I would like to thank Dr. Cameron Patterson, for being on my committee and providing highly insightful feedback.

A special mention to SSRG friends Ashwin Krishnakumar and Ho-Ren Chuang for their timely advice and help in making this thesis possible. I would be remiss if I didn't thank my close friends, Sidhaarth and Abhijit, for all the fun, advice, support, and encouragement.

Finally, a shoutout to my parents, Sathish and Bhavani, and my brother Abishek, for their constant love and support. Also, I would like to thank all my relatives for always being there for me.

This work is supported by the US Office of Naval Research under grants N00014-18-1-2022, N00014-19-1-2493, and N00014-21-1-2523.

Contents

List of Figures **xi**

List of Tables **xiv**

1 Introduction **1**

1.1 Motivation 2

1.2 Thesis Contributions 5

1.3 Thesis Organization 6

2 Background **8**

2.1 Distributed Shared Memory 8

2.2 Field Programmable Gate Arrays and SmartNICs 10

2.3 Popcorn Linux 11

2.3.1 Popcorn Compiler 12

2.3.2 Messaging Layer 13

2.3.3 Popcorn Thread Migration 15

2.4 Software-DSM in Popcorn 16

2.5 On-Demand Migration in Popcorn 18

3 Related Work **23**

3.1 Interfaces for software DSM systems 23

3.2 Hardware-Accelerated DSM systems 25

3.3 Cache Coherent Interconnects 28

3.4 Baselines for Evaluating fDSM 29

4 FPGA Messaging Layer **31**

4.1 Overview 31

4.2 Hardware Logic 33

4.2.1 PCIe-DMA Express™ 34

4.2.2 Ultrascale 100G Ethernet Subsystem™ 34

4.2.3 FIFO Logic™ 35

4.2.4 Descriptor Controller Module 35

4.2.5 DSM Protocol Processor 36

4.3 FPGA Driver 36

4.3.1 PCIe Helper Block 37

4.3.2 Polling Thread Handler 39

4.3.3 Send and Receive Buffers 40

4.3.4 IRQ Handler 41

5 fDSM Design **43**

5.1 DSM Processor 43
5.1.1 Memory Mapped Interface 44
5.1.2 Register Space 44
5.1.3 AXI-Stream Input/Output Modules 45
5.1.4 FSM Core 47
5.2 Popcorn Kernel Support 50

6 Experimental Evaluation **54**
6.1 Overview 54
6.2 Experimental Results 55
6.2.1 Speedup Analysis 58
6.2.2 Timing Breakdown 63
6.2.3 Homogeneous Setup Evaluation 65
6.3 Discussion 66
6.3.1 Ideal Application Behavior 68
6.3.2 FPGA resource utilization 70
6.4 Evaluation Summary 71

7 Conclusions and Future Work **72**
7.1 Limitations 74
7.2 Future Work 75

Bibliography 76

List of Figures

2.1 Popcorn Linux Single System Image 12

2.2 Transport Layer and Device Drivers 14

2.3 Popcorn Kernel Infrastructure 16

2.4 Popcorn Sequential Consistency FSM 17

2.5 Popcorn Linux Details - Remote Fault 19

2.6 Popcorn Linux Details - Origin Fault 21

4.1 Hardware Setup 32

4.2 Overview of FPGA Messaging Layer 33

4.3 FPGA Hardware Logic 33

4.4 DMA Descriptor 35

4.5 FPGA Driver 37

4.6 (a) Comparison of IOMMU and MMU (b) Kernel Memory Layout [24] . . . 38

4.7 Send and Receive Buffers memory allocation 41

5.1 DSM Processor 44

5.2 AXI-Stream Input Output modules 46

5.3 Page Key 47

5.4 FSM Core 48

5.5 RPR Handling Flowchart 51

5.6 Local fault Handling Flowchart 52

6.1 Speedup of fDSM over Popcorn with TCP-IP and Popcorn with RDMA in A class of NPB and PARSEC Benchmarks 57

6.2 Speedup of fDSM over Popcorn with TCP-IP and Popcorn with RDMA in B class of NPB and PARSEC Benchmarks 57

6.3 Speedup of fDSM over Popcorn with TCP-IP and Popcorn with RDMA in C class of NPB and PARSEC Benchmarks 58

6.4 Compute Intensity and Memory Intensity and number of page faults raised in the benchmarks 60

6.5 Page fault periods with respect to messaging layers 61

6.6 Ratio of number of page faults to execution time of benchmark 62

6.7 Timing breakdown of the resolution of a page fault using Popcorn with RDMA 64

6.8 Timing breakdown of the resolution of a page fault using fDSM 64

6.9 Speedup of fDSM over Popcorn with TCP-IP and Popcorn with RDMA in B class of NPB and PARSEC Benchmarks - Homogeneous Setup 66

6.10 Speedup of fDSM over Popcorn with TCP-IP and Popcorn with RDMA in B class of NPB and PARSEC Benchmarks - Homogeneous Setup 67

6.11 Speedup of fDSM over Popcorn with TCP-IP and Popcorn with RDMA in B class of NPB and PARSEC Benchmarks - Homogeneous Setup 67

6.12 Reason for decreasing speedup over bigger classes of benchmarks 69

6.13 Execution time observations in both heterogeneous and homogeneous setups 69

List of Tables

5.1 Sample Register Space 45

6.1 FPGA Resource Utilization 70

List of Abbreviations

AXI Advanced eXtensible Interface

DSM Distributed Shared Memory

FIFO First In First Out

FPGA Field Programmable Gate Array

FSM Finite State Machine

GPF General Page Fault

HPC High Performance Computing

IOMMU Input-Output Memory Management Unit

IP Intellectual Property

ISA Instruction Set Architecture

MMU Memory Management Unit

MSI Modified, Shared, Invalidate

OS Operating System

PCIe Peripheral Component Interconnect Express

PID Process Identifier

PTE Page Table Entry

RDMA Remote Direct Memory Access

RPR Remote Page Request

SC Sequential Consistency

TX/RX Transmit/Receive

VMA Virtual Memory Area

Chapter 1

Introduction

Shared memory centralized computing systems are increasingly incapable of handling large volumes of computations in many application domains such as high-performance computing, transactional systems, and networked services, among others. Distributed computing, a computing paradigm that allows running multiple programs across several computing nodes connected via a network, overcomes these performance and scalability challenges. A distributed system provides scalability with the effortless addition of more machines and reliability, meaning if one node fails, others aren't affected. The advent of distributed systems dates back to the late 1960s with the invention of local area networks like ARPANET and the Internet and remains to be the foundational model of today's technologies like Cloud and Edge computing.

Programming a distributed system requires a model that is fundamentally different from that of shared memory systems since there is no globally coherent shared memory in a distributed system. The most widely used model for programming distributed systems is message passing. In the message-passing model, processes operating on distributed nodes communicate using primitives such as `send()` and `receive()`. Since the message passing model is, in general, only applicable to loosely-coupled systems, migrating existing shared memory programs (e.g., legacy) to such systems is usually challenging. In addition, shared memory programmers often have a non-trivial entry bar for loosely coupled systems, especially for obtaining high performance.

The shared memory paradigm is the most efficient way of passing data between processes in a tightly-coupled system like a multiprocessor computer [31, 32, 37]. In the shared-memory model, all co-operating processes have access to the same global address space. In the late 1980s, researchers studied the applicability of the shared memory model for distributed systems, with each node having its own local memory [52, 53, 93, 94]. These efforts resulted in a model known as Distributed Shared Memory, commonly known as DSM, which provides a shared memory abstraction over physically distributed systems, thereby giving the illusion of a globally "shared" memory [57, 75, 89]. It combines the benefits of a shared-memory model (flexibility and programmability) and the distributed system model (low cost and scalability) [76]. The DSM model's ability to provide a transparent interface and convenient programming environment for parallel applications has drawn significant interest over the years [8, 15, 45, 49, 51, 63, 64, 76, 87].

1.1 Motivation

With the approaching limits of single-threaded CPU performance, chip vendors are quickly transitioning toward greater specialization and accelerated computing architectures. Standalone multi-core systems can no longer provide the computational power required by today's applications. As a result, there have been advancements toward integrating CPUs with other specialized processing platforms, commonly known as accelerators, like GPUs, DSPs, ASICs, and FPGAs. These host and accelerator combinations are commonplace now, especially the CPU+GPU combination, and they efficiently process applications using the advantages of different compute units and achieve higher performance, increased parallelism, and lower power.

An important emerging architectural trend in this space is the emergence of "Smart" I/O devices such as SmartNICs [67, 68, 69] and Smart NVMe storage devices (SmartSSD) [84]. Along with an accelerator like GPU or FPGA, these smart I/O devices also house a multi-core CPU (usually of the ARM ISA class). This enables such devices to perform tasks like low-level packet processing, traffic routing, and load balancing. For instance, a SmartNIC has a CPU+FPGA combination to offload high-performance applications from the host server and high-speed network interfaces like Infiniband or Ethernet to connect to different servers or remote SmartNICs [68]. With many high-performance servers using the x86 ISA, these connected smart accelerators enable heterogeneous computing.

With such heterogeneous-ISA computing environments becoming increasingly mainstream, it can be advantageous to integrate them into a coherent distributed system to share resources among them at high speeds while maintaining memory consistency. Toward this end, Heterogeneous Distributed Shared Memory (HDSM) [11, 54] has been studied to enable parallel applications to exploit resources available on ISA-different computing cores.

There exists a large body of work on DSM. Early works include TreadMarks [45] and its many derivatives [6, 8, 21, 28, 60, 64, 71]. More recent efforts include Popcorn Linux's DeX [46], K2 [54], and [12, 17, 19, 27, 62, 85, 96], which provide a software-based DSM but they incur the overhead of maintaining consistency in software. In contrast, efforts such as FLASH [49], DASH [51], Concordia [98], Tempest [77], and [1, 5, 25, 73, 78, 79, 92], are hardware-supported DSM built using custom chips and accelerators like GPUs and network switches. Hardware-based DSM uses hardware support to eliminate the software overhead of maintaining memory consistency. For example, FLASH [49] is a scalable multiprocessor that uses a MAGIC (Memory And General Interconnect Controller) chip that controls the cache-coherent shared memory protocol and high-performance message passing. DASH [51] is a multiprocessor that provides hardware-accelerated cache-coherent shared memory between

multiple processing nodes. Other examples include MIT Alewife [1], MEMNET [25], and CAPNET [92] systems. Although these works show favorable performance benefits compared to software DSM systems, they do not allow reconfigurability to experiment with other consistency models. In addition, they cannot be scaled up to a distributed system to maintain memory coherency between multiple server-grade nodes.

Similar to hardware DSM, hardware and software vendors like ARM, AMD, Xilinx, Qualcomm, Microsoft, IBM, and NVIDIA have developed several cache-coherent interconnect standards like CCIX [18], CXL [23], and OpenCAPI [88] to facilitate cache coherency between a host machine and off-chip accelerators. These interconnects move the application data autonomously between the processor and accelerator caches without the intervention of a software driver using memory addressed with virtual addresses. The downside of these interconnects is that despite some of them being implemented on FPGAs, they do not allow reconfigurability and are locked to a particular accelerator card. For instance, an OpenCAPI-enabled FPGA card, like the Alpha Data 9v3 accelerator card [4], employs the CAPI interface standard to maintain cache coherency and does not allow any other consistency protocol, and avoids DMA of data between the host and the accelerator. Although this could be advantageous in a host/accelerator setup, the developer is constrained by this cache coherency model and cannot utilize these interconnects to provide a virtual memory page-based DSM.

Based on these observations, this thesis proposes an FPGA-accelerated DSM framework, namely **fDSM**. fDSM provides a memory consistency model between heterogeneous ISA nodes using a virtual memory page-level, multiple-reader single-writer consistency algorithm. Our intended use case of fDSM is to provide page-based VMA coherency between a host CPU and a smart I/O device such as a SmartNIC. fDSM is customizable and can implement different DSM algorithms to support different consistency models. It is also expandable to a phys-

ically distributed hardware configuration consisting of ISA-similar (i.e., ISA-homogeneous) and ISA-different (i.e., ISA-heterogeneous) computing nodes. fDSM provides reprogrammability using Field Programmable Gate arrays (FPGAs), a matrix of configurable logic blocks connected via programmable interconnects. Our work assumes that FPGAs are available in both hosts and Smart I/O devices, as is the case in today's commodity hosts and I/O devices at the high end of the computing spectrum.

We design a scaled-up prototype to emulate a host and a SmartNIC heterogeneous setup using two server-grade heterogeneous-ISA machines that are interconnected through Xilinx FPGAs and 100 GB QSFP cables. Since we employ an atypical approach of providing virtual memory coherency at a 4KB page granularity, there aren't any direct comparisons to our work. However, we compare our prototype to Popcorn Linux's software DSM [9]. We build fDSM's OS support with task migration, page modification, and message passing constructs that are leveraged from Popcorn, evaluate fDSM using the NPB and PARSEC HPC benchmark suites, and demonstrate speedups.

1.2 Thesis Contributions

This thesis makes the following contributions:

- We design a high-bandwidth, low latency FPGA-based messaging layer that acts as an inter-OS kernel communication interface between two connected nodes in a distributed system. We employ the Ethernet protocol and operate at a link speed of 100GBps. We develop a plug-in driver to install the messaging layer, and a programmer can reroute the messages to the FPGA logic with minimal OS changes.

- We introduce **fDSM**, an FPGA-accelerated DSM for heterogeneous-ISA hardware.

fDSM includes a DSM protocol processor implemented in the FPGA logic, which is responsible for managing shared memory consistency between ISA-heterogeneous nodes at page granularity. fDSM is built on top of the FPGA messaging layer, and it provides Sequential consistency (SC) [50] using a single-writer/multi-reader algorithm. The algorithm keeps track of page owners and their read/write permissions, and invalidates current owners to provide new owners with exclusive write permissions. Read-only pages are replicated for concurrent read access. The DSM processor is equipped with pipelined-registers and packet forwarding modules, enabling it to transmit, receive, and process remote page requests.

- We evaluate the performance of fDSM using 9 NPB applications, and 1 PARSEC application. Our results show that fDSM outperforms Popcorn Linux's software DSM that uses state-of-the-art Infiniband RDMA interconnect by 7%.
- We also evaluate with the homogenenous-ISA hardware (x86 + x86), which could also be used with an ARM + ARM setup, where an ARM accelerator can be connected to a ARM host CPU. Our results show fDSM achieves performance gains of 6.2% over state-of-the-art Infiniband RDMA interconnect.

1.3 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we provide a background on DSM and FPGAs along with a comprehensive overview of Popcorn Linux, the software DSM protocol of Popcorn, and its task migration and on-demand paging mechanisms. We discuss past and related works in the research domains fDSM target and contrast them against fDSM in Chapter 3. We describe the design and implementation details of the FPGA-based messaging layer and fDSM in Chapters 4 and 5, respectively. In Chapter 6, we present the

evaluation of fDSM's performance. Finally, we conclude our work, and discuss its limitations, and outline future work in Chapter 7.

Chapter 2

Background

In this chapter, we explain the Distributed Shared Memory model in Section 2.1, and Field Programmable Gate Arrays (FPGAs) and Xilinx™ services in Section 2.2. In Sections 2.3 and 2.4, we explain the internals of Popcorn Linux [9], and its Sequential consistency protocol. In Section 2.5, we illustrate task and page migration in Popcorn Linux.

2.1 Distributed Shared Memory

To meet the growing demand for high-performance computing and overcome the bottleneck of memory contention in tightly-coupled systems, researchers came up with the concept of Distributed Shared Memory (DSM) that incorporates the advantages of Shared Memory and Message Passing in Distributed Systems. DSM is an abstraction of memory over several physically distributed systems, commonly known as nodes, that gives processes running on these systems an illusion of a ”shared” virtual address space [76]. One of the essential benefits of DSM is that no code modifications are required to run parallel programs developed for shared memory systems. DSM allows processes executing on interconnected processors to share memory by making the address space transparent to the entire system. The general structure of a DSM system involves several nodes interconnected by a communication interface (typically network interconnects). Each node can be a uni-processor or a multi-processor system with local memory and caches. The sharing granularity, or the size of the

memory transferred, is closely related to the structure. In modern DSM systems, memory is shared in 4kB or 16kB pages, depending on the granule size of the system.

DSM classifies into two broad categories: Implementation level and DSM algorithm. We could further categorize Implementation into Hardware and Software implementation, which impacts the programming model and overall system performance. Hardware DSM provides transparency to the programmer and can achieve lower latencies, whereas software DSM can exploit the application behavior and is more flexible for experimenting with new algorithms. The second criterion, the DSM algorithm, is based on the ownership and existence of multiple copies of shared memory granules. Maintaining consistency and coherency of memory between the nodes is one of the pivotal aspects of DSM [75]. There are three types of algorithms used in conjunction with the coherency model:

- Single Reader/Single Writer
- Multiple Reader/Single Writer
- Multiple Reader/Multiple Writer

The convenient programming model and the transparency provided by DSM have made it the focus of several research efforts in recent years. One of the main objectives of this research was to design high-performance and low-latency coherence and consistency models [44] [110] [86] [108]. Some examples of coherency algorithms include strict, sequential, weak, release, eager release, and lazy release consistency protocols. These protocols, along with consistency models like write-invalidate, writer-shared, and type-specific ensure that the requesting node receives the most up-to-date copy of the data [12]. In addition to this, the responsibility for DSM management can be centralized or distributed, meaning a leader node would be taking the coherency-related actions in a centralized system. However, in a distributed DSM

management, consistency is maintained by all the nodes in the system. In Section 2.4, we describe the coherency and consistency model we used in our implementation.

2.2 Field Programmable Gate Arrays and SmartNICs

Field Programmable Gate Arrays or commonly known as FPGAs, provide the designer with a matrix of Programmable Logic Blocks that can be reconfigured based on the design. These logic blocks include flip-flops, logic gates, multiplexers, and static Block Random Access Memory connected with the help of programmable interconnects. The first FPGAs were introduced in the market by Xilinx in 1984, and they have gone through several stages of technological advancements [95].

Since their introduction, FPGAs have been thriving in the HPC (High Performing Computing), telecom, and Datacenter markets. Their power efficiency and high-performance communication capabilities can boost the compute capacity of servers and accelerate application performance in the Datacenter space [99]. Additionally, their parallelism, reconfigurability and lower operational costs give them a competitive edge over GPUs (Graphics Processing Units), ASICs (Application Specific Integrated Circuits), and hybrid CPU+GPU combinations. FPGAs continue to be a rich platform for researchers from several distinct areas like Computer Architecture, DSP (Digital Signal Processing), Image Processing, Machine Learning, Circuit Design, Distributed Systems, and Big Data Analytics [13] [34] [3].

The evolution of FPGAs over the past three decades has been exponential. Modern FPGAs have adopted a modular approach and have become easier to program, and they are more compatible with different OS and technologies. They support in-chip protocols like AXI and are compatible with cutting-edge PCIe x3 and x4 protocols. Top FPGA vendors like Xilinx and Intel are pushing towards high-performance Data Accelerator cards that provide

various benefits like smaller form factors, better networking capabilities, and higher data bandwidths. One such FPGA used by us is the Xilinx Alveo U50 Data Center accelerator card. It is equipped with 8GB of High Bandwidth Memory and is used to expedite memory and compute-intensive applications in the Datacenter space [100]. We use the IPs provided by the Xilinx Vivado Design Suite to design our system and to place and route the resources and program the FPGA [102].

Today, FPGAs have revolutionized the data acceleration industry to such an extent that Smart Network Interface Cards (SmartNICs) have started using FPGAs for offloading packet processing tasks. Thanks to their hardware reconfigurability, FPGA NICs provide an additional benefit of supporting diverse accelerators and a wider set of applications [16]. For instance, the Bittware 250 Accelerator card board features a Xilinx SoC (System on Chip) with a 64-bit ARM processor and reconfigurable PL (Programmable Logic). It also hosts two 100GBe network ports, onboard DDR4 memory, and a PCIe generation 3 interface. It is designed for Non Volatile Memory interface options and enables remote disaggregated memory storage [14].

2.3 Popcorn Linux

The primary objective of Popcorn Linux, commonly known among us as Popcorn, is to improve the programmability of heterogeneous ISA hardware by providing OS and run-time support to the Linux kernel. Figure 2.1 shows a replicated-kernel OS model for the Linux kernel, presented by Popcorn Linux, which provides the applications a single-image OS view over different nodes. In this model, multiple nodes transparently provide a consistent and coherent view of the memory, allowing threads to migrate freely across machine boundaries [11] [10]. Popcorn boots a Linux kernel on single-core/multi-core x86 or ARM ISA ma-

chines, and it allows multiple instances of the kernel to share hardware resources without virtualization.

A user can clone the Popcorn Linux repository comprising hardware-customized configuration files and kernel resources, and compile them to obtain the boot images [91]. Upon boot of the kernel instance, the infrastructure is ready to be used, and developers can switch to the Popcorn namespace and run applications on the replicated-kernel OS [9]. The Popcorn Linux project provides a Popcorn Kernel, a Linux kernel based OS support to facilitate thread and page migration, and a Popcorn compiler toolchain to generate heterogeneous ISA binaries of C/C++ applications.

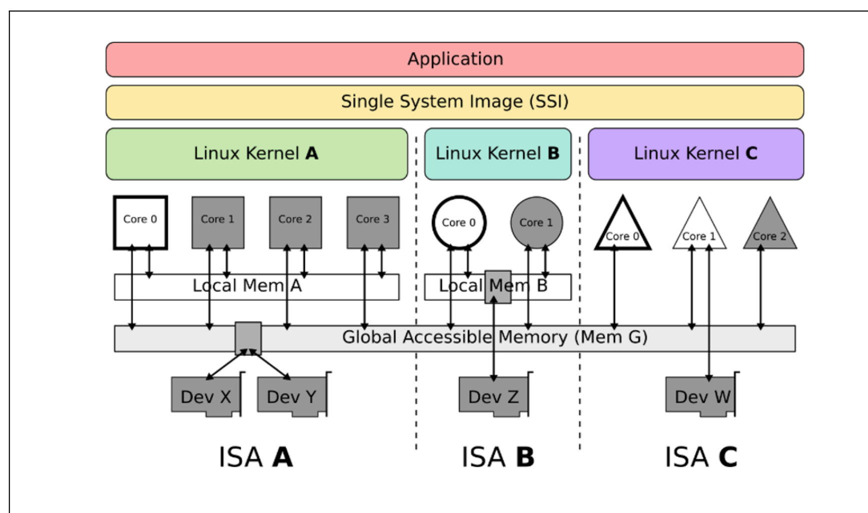


Figure 2.1: Popcorn Linux Single System Image

2.3.1 Popcorn Compiler

Popcorn Linux uses Popcorn compiler, an LLVM-based customized compiler that translates C/C++ applications into machine code, facilitating seamless execution and run-time migration across multiple ISAs. This toolchain adjusts the data and code layout of the applications and prepares them for migration by inserting migration checkpoints into the generated ma-

chine code. The compiler automatically inserts the migration points into the source code at function call sites, which use the system calls in the Popcorn kernel (Popcorn Linux OS support) to transfer the thread information and process execution contexts between the nodes [80] [90]. To prepare migratable applications, the compiler requires support to application interfaces and SMP semantics like the POSIX thread libraries.

2.3.2 Messaging Layer

Note - In this work, we often use the term heterogeneous ISA/setup, meaning the interconnected Popcorn nodes have distinct ISAs (x86 and ARM, or x86 and RISC-V). Similarly, a homogeneous setup means the nodes have a similar ISA (x86 and x86, or ARM and ARM).

The heterogeneous/homogeneous nodes communicate with each other through a high-performance low-latency messaging layer. They migrate the execution contexts and the VMA of a distributed process between the nodes and the handlers in the destination nodes receive the incoming data and process them. The messaging layer consists of the Messaging Interface, Transport Layer, Device Drivers and the Physical interconnects. The messaging layer is the critical aspect of Popcorn and governs the performance of the overall system.

Popcorn enables inter-kernel communication using something called a Popcorn Kernel Message (`pcn_kmsg`), a data structure with message headers to indicate the type of the message, and a memory region to package the message data. The message data can be a thread's execution context, process construction/zapping information, remote page requests, invalidation messages, node acknowledgments, or invalidation/remote page responses.

Inter-node communication happens in four steps. First, the kernel acquires/allocates a memory region and writes the message data. Second, it calls the Messaging Interface, an intermediate layer between Popcorn Linux and the Device layer. The Messaging module checks

the message for errors and prepares it for transmission by encapsulating the type of the message and the data. Third, based on the nature of the operation (send/write), the messaging interface calls the corresponding transport layer functions, which invoke the handlers in the device driver. The device driver then reads and transfers the message through the registered interconnect.

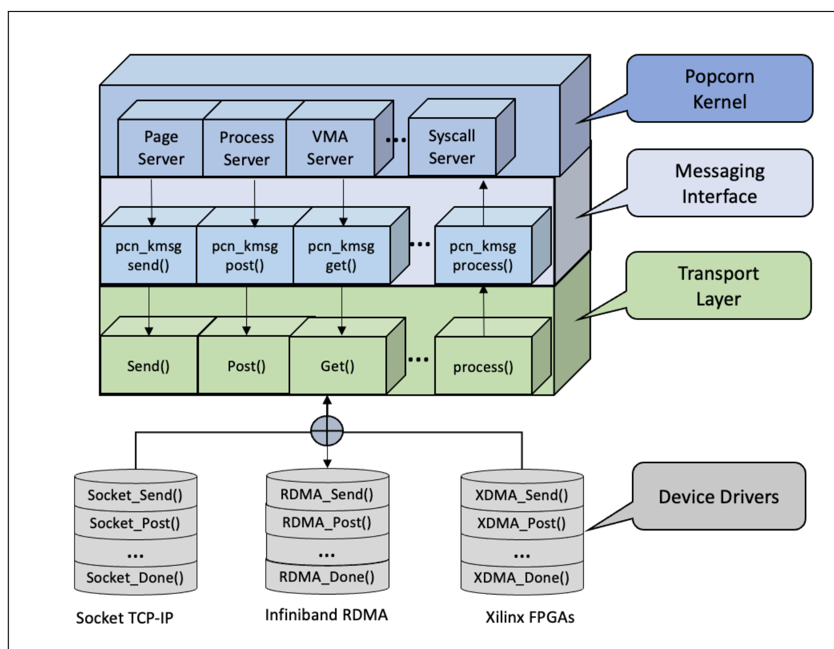


Figure 2.2: Transport Layer and Device Drivers

The kernel customizes the message according to the registered interconnect. For instance, the kernel packages the DMA address and remote DMA address into the message if the registered interconnect is an Infiniband RDMA adapter. Developers can choose the interconnect of their choice by loading the respective driver. Once the device driver corresponding to a particular interconnect is loaded, it is responsible for updating the transport layer and communicating this to the kernel. The transport layer then routes all the customized incoming messages through the interconnect. As shown in Figure 2.2, the device driver comprises different versions of the messaging interface functionalities tailored to the interconnect the driver represents. For example, the RDMA driver has the `rdma_send` functionality tailored to send

messages over the Infiniband-RDMA adapter [91].

Inversely, upon the reception of a message, the device driver calls the `pcn_kmsg_process` function in the Messaging Interface and points the received message to it. This function is responsible for processing the message and invoking the appropriate callback functions registered for the type of message. For instance, a remote page request callback function (located in the page server) is registered in the Messaging interface when the kernel is booted. This helps the `pcn_kmsg_process` to invoke this recorded callback function once a remote page request is received. Similarly, every type of Popcorn kernel message has a callback function associated with it.

2.3.3 Popcorn Thread Migration

Figure 2.3 depicts the Popcorn Kernel infrastructure. Each thread constructs its VMA through page faults and uses the system call interface to communicate with the Popcorn kernel and migrate its information to the destination node. The Popcorn kernel consists of various modules, each playing a particular role in the migration of the process [46]. For instance, the process server migrates the user-space threads and their execution contexts, terminates worker threads, and does other process handling functions.

In Popcorn, only user-space threads are migrated and kernel threads aren't. Instead, the kernel provides the service (process server) that guarantees continued execution of the migrated thread on the destination nodes. In order to do this, the process server fetches and transfers the process's stack pointer registers (`struct pt_regs`), program counter values, and frame pointers [41] [58] [70].

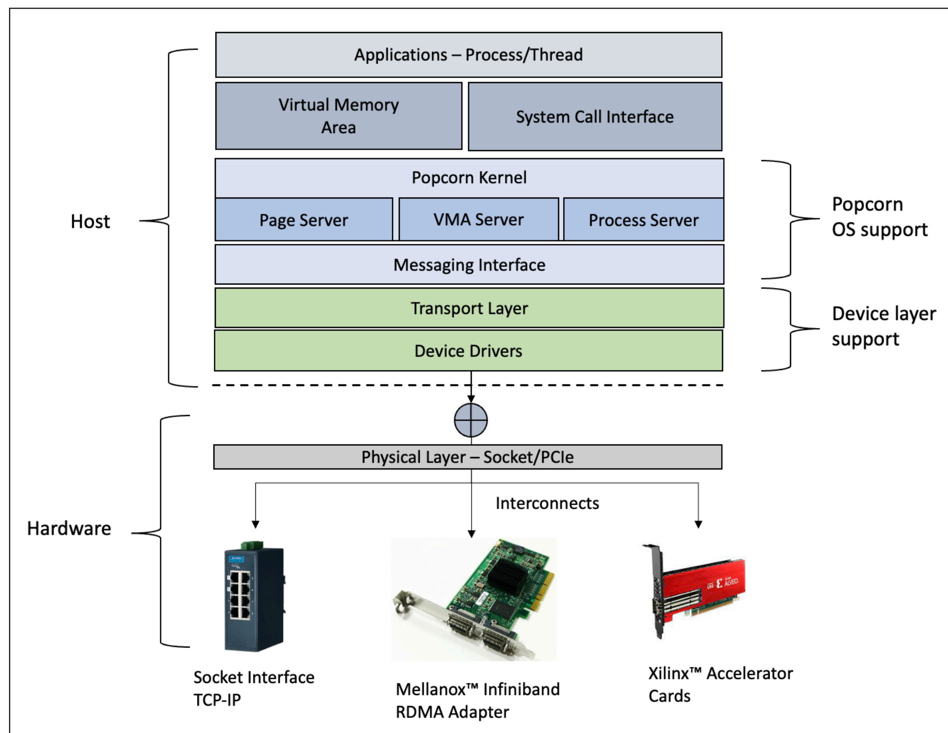


Figure 2.3: Popcorn Kernel Infrastructure

2.4 Software-DSM in Popcorn

Popcorn employs a Sequential Consistency model coupled with a Single-Writer-Multiple-Reader (SWMR) coherency protocol, similar to the MSI cache coherence protocol [72]. Popcorn implements this DSM protocol in the software, namely Software DSM, to guarantee a strict memory view coherence between the executing threads. The protocol is a centralized, writer-invalidate protocol and manages consistency at a page-level granularity. Ownership of these pages is stored and tracked in a per-process radix tree and on a per-page and per-node basis at the origin node [83]. Figure 2.4 depicts the Finite State Machine of Popcorn's DSM protocol from a node's perspective.

In a two-node setup, a page is in the Modified state when the origin/local node creates a VMA for the threads. A read fault on the remote side for the page changes the state to

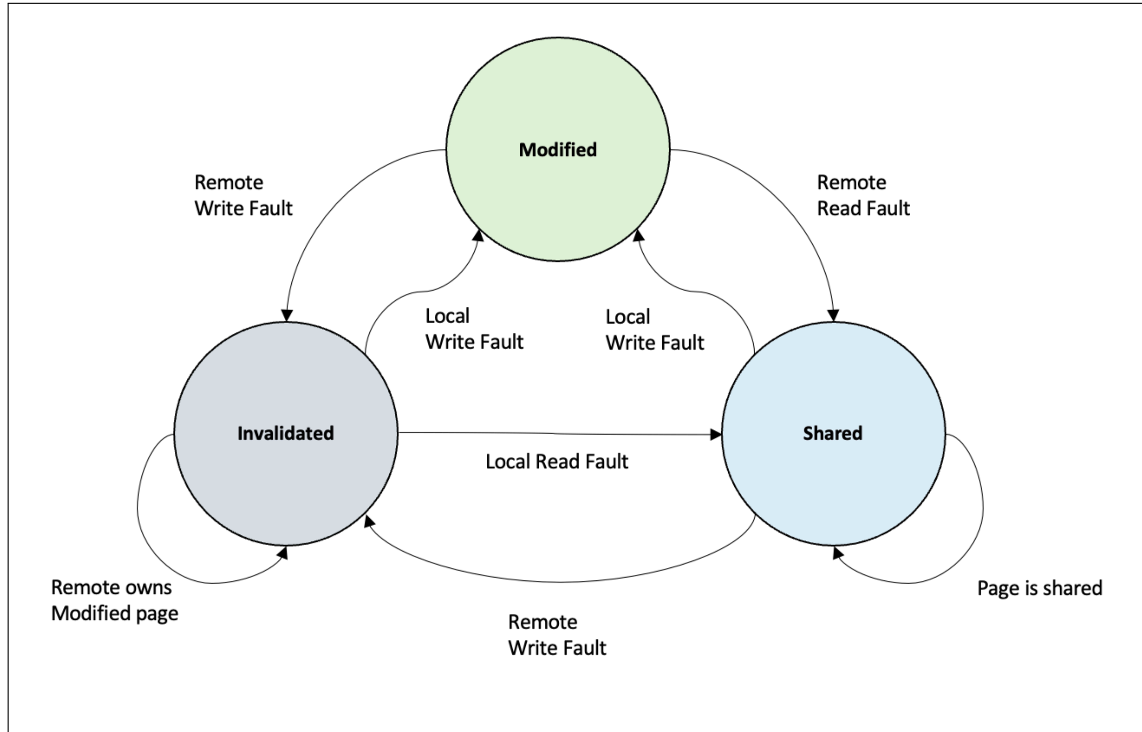


Figure 2.4: Popcorn Sequential Consistency FSM

Shared on both the nodes. The page owner shares a replicated copy of the page to the requesting node and protects it from further writes. On a write fault of a page, the state jumps from Modified/Shared to Invalidated on the owner side, and the state of the page changes to Modified on the requesting/receiving node. The owner invalidates the copy of the page it owns and cedes the ownership to the requesting node giving it exclusive access to modify the page it already has or receives. Upon a local read fault of an Invalidated page, the origin node sends out an RPR (Remote Page Request) in the form of a `pcn_kmsg` (Popcorn Kernel Message) and claims a shared copy of the page from the remote. Incidentally, the page state changes to Shared on both the nodes. If a local write fault occurs on an Invalidated page, the origin node sends out an RPR and claims back the page from the remote node, which invalidates the copy it owns and concedes the ownership back to the origin node. However, if a local write fault occurs on a Shared page, instead of an RPR, the origin sends

an Invalidation Request message to the remote and waits for an acknowledgment. Upon the reception of acknowledgment that the remote has invalidated its copy of the page, the origin node starts Modifying it. In this case, the state changes to Invalidated on the remote side and Modified on the origin side.

Note: In Popcorn and this work, a local fault isn't the same as the "origin node" fault. Similarly, a remote fault is different from a "remote node" fault. For instance, from the remote node's perspective, a local fault is a GPF occurring in its kernel, and a remote fault is a GPF occurring in the origin node's kernel. The same applies when we look at the faults from the origin node's perspective.

2.5 On-Demand Migration in Popcorn

In this section, we describe Popcorn's Task migration and On-demand Paging in detail. Figures 2.5 and 2.6 briefly outline the process flow of Popcorn's page faults. They also show software-implemented Popcorn's Sequential consistency protocol in action. To explain the process flow better, we choose a single-threaded C application that migrates a thread and ping-pongs a 4kB page between two nodes. In Popcorn, each node has an ID, known as Node ID (NID). The kernel uses these NIDs to determine the Popcorn Kernel message (`pcn_kmsg`) destination node.

Figure 2.5 illustrates the steps, (1) → (10), to resolve a local page fault on the remote node. The migration points in the application are Migrate() system calls that invoke Popcorn's kernel service to fetch the thread information and transfer it to the remote node. In the origin node (the node that initiates thread migration), the migrate system call invokes the task migration handler in the Popcorn kernel (process server). The handler organizes the task information, fetches the `task_struct` and the `mm_struct` (information related to the

process address space), encapsulates it in a Task Migration `pcn_kmsg`, and calls the Popcorn messaging interface to send this message to the destination node. The messaging interface invokes the device driver's Send handler to transfer this message through the registered interconnect, as shown in (1).

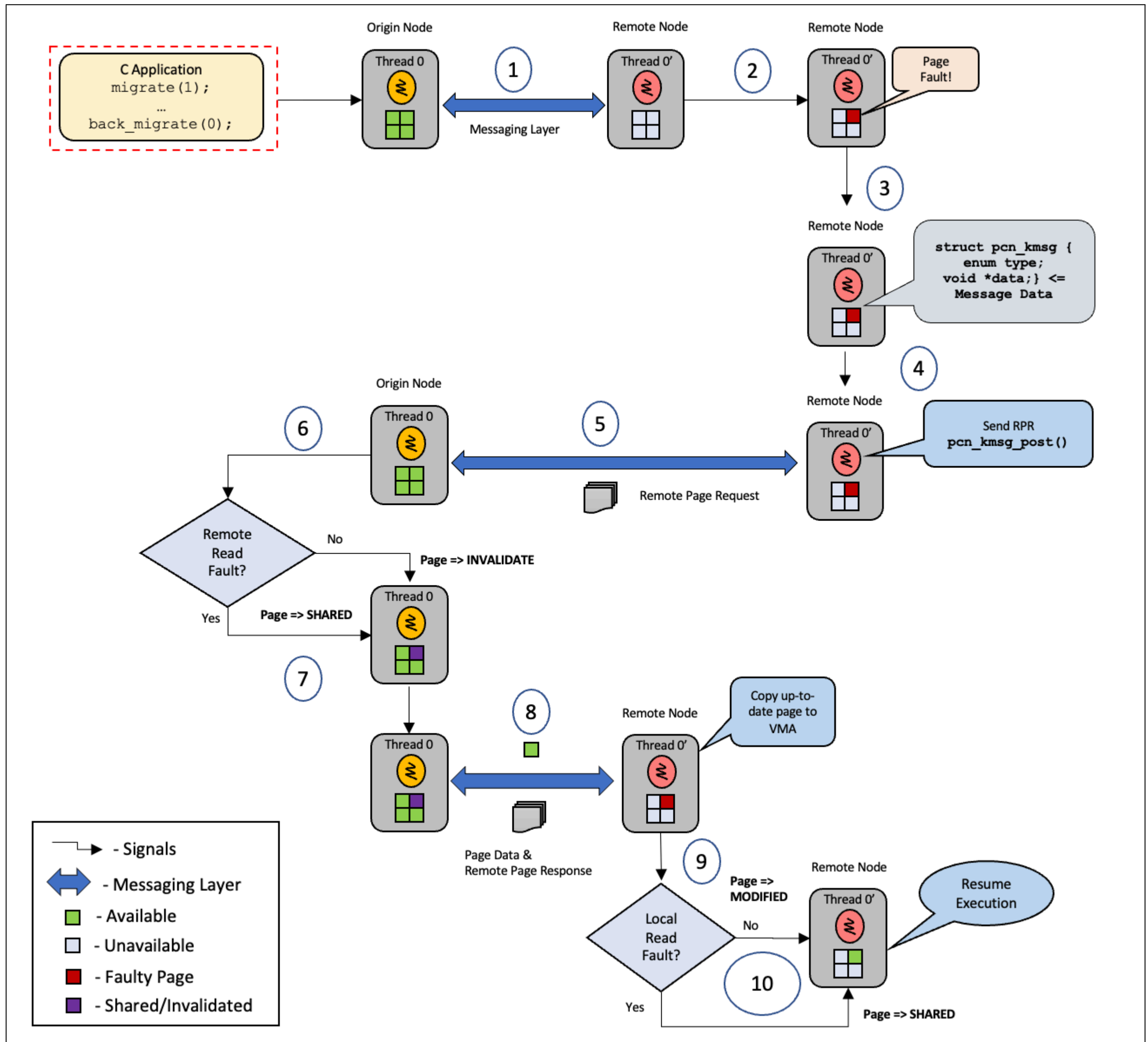


Figure 2.5: Popcorn Linux Details - Remote Fault

In step (2), the destination/remote node receives the migration message and processes it (by

calling the relevant callback functions registered for the message type). It clones a remote worker thread with a new PID and reconstructs the thread's address space by raising VMA faults and fetching the up-to-date VMA from the origin node (on-demand VMA fetch). Upon the completion of VMA fetch, the process raises Read/Write GPFs (General Page Faults) when it tries to access the data in the reconstructed VMA. These page faults are trapped by the Page Server and the respective handlers packages the required page's information such as the virtual address of the faulty page, stack pointers, Remote Process ID, Node ID, Fault flags, and the wait station ID into another `pcn_kmsg`, as shown in step (3) in Figure 2.5.

The page server then calls the Messaging interface handler and sends out the RPR (Remote Page Request) message to fetch the page from the origin, in step (4). The function that sends out a message in the Messaging interface is the `pcn_kmsg_post()`. The thread creates a Wait Station (a Pthread barrier) and waits for a response from the origin. The RPR travels through the interconnect layer and reaches the origin node, as shown in (5). The origin invokes the callback function for an RPR in its local page server and processes the page request information.

The page server on the origin parses the fault flags received in the message and takes the appropriate DSM action on the requested page. For instance as shown in (6), if it is a remote Read fault, the origin retains the ownership and protects the page from further writing. The state of the page is changed to Shared. On the other hand, on a remote Write fault, the origin grants the ownership to the remote node and Invalidates its local copy of the page, thus changing the state to Invalidated. In step (7), the origin prepares the page for transmission and sends the page to the remote node, and follows it up with a remote page response.

In steps (8) and (9), the page and the response arrive at the remote node and are processed by the page server. Following that, the received page is copied to the faulty VMA of the worker thread. In step (10), the kernel changes the ownership and the status of the page

based on the type of page fault and resumes execution.

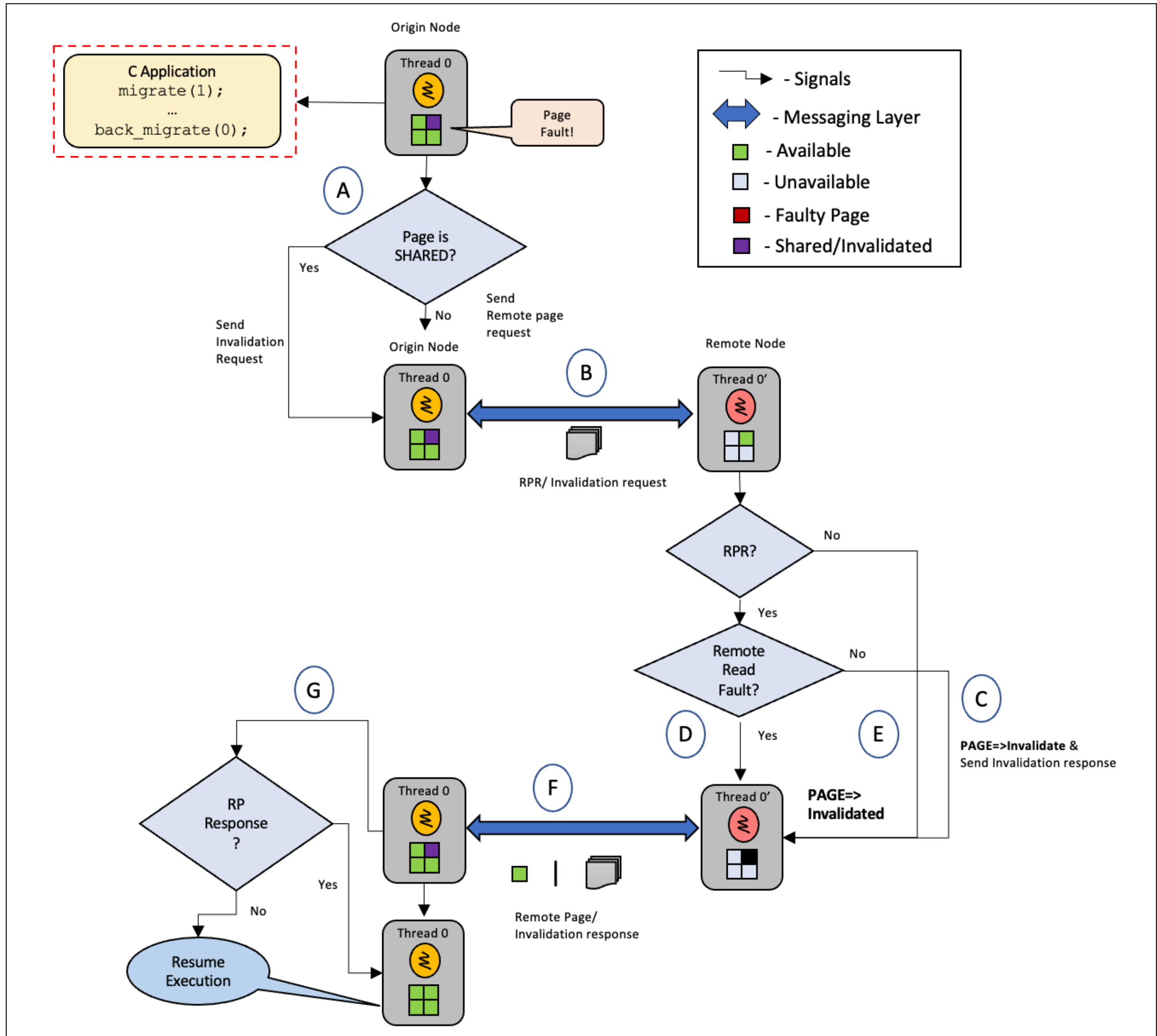


Figure 2.6: Popcorn Linux Details - Origin Fault

In Figure 2.6 we outline all the steps, (A) to (G), to resolve a local fault in the origin node. We use a different encoding to distinguish origin and remote faults. Upon the completion of a remote fault, the remote work thread exits and migrates the task back to the origin. Once the back migration of the worker thread is successful, the origin node raises page faults that

are trapped by the fault handlers in the page server, as shown in (A). In step (B), The origin checks the state of the page. If the Page is shared between the two nodes, an RPR would be redundant as the origin already owns the up-to-date copy. In this case, it sends out an invalidation message to the remote, requesting it to invalidate the page copy. Conversely, if the local copy of the page is Invalidated, i.e. Modified by the remote, the origin sends an RPR to the remote with all the page information.

The RPR/Invalidation request arrives at the remote and the relevant callback functions are invoked, as shown in (C). If it is an Invalidation message, the page server invalidates its copy and sends out an Invalidation response, acknowledging that it has voided its copy and the origin can resume modifying its local copy. Conversely, if it is an RPR, the remote page server checks for the type of fault in step (D). If it is a Read fault in the origin, the Page is sent with a response and it is Shared between the two nodes with the remote retaining the ownership. In the case of an origin Write fault as shown in (E), the Page is Invalidated and the up-to-date copy of the page is sent with a response and the remote concedes the ownership back to the origin.

In steps (F) and (G), the Invalidation response/remote page response travels through the interconnect and reaches the origin, which processes it. If it's an invalidation response, the origin resumes execution and modifies the page. On the other hand if its a remote page response, the kernel copies the page into the faulty VMA and resumes execution.

Chapter 3

Related Work

In this chapter, we provide an overview of the related works in the research domains targeted by our work, fDSM. In Section 3.1, we discuss low-latency messaging interfaces implemented in software DSM systems. We explain how researchers have tried implementing DSM protocols in hardware, such as GPUs, FPGAs, and switches, in Section 3.2. In Section 3.3, we discuss modern interconnects that provide DSM coherency between a host and an accelerator. In section 3.4, we present the baselines for the comparison of fDSM.

3.1 Interfaces for software DSM systems

Due to the increasing demand for low-latency network communication in the data center space, there has been increasing interest among researchers and the industry to design high-performance scalable interconnects. Throughout the years, research on low-latency interconnects has resulted in technological breakthroughs. Network hardware has evolved from optical interconnects [47], 155 MBps Ethernet interconnects [59], 10 Gigabit Ethernet to 400 Gbps Infiniband RDMA [38] and RDMA over Converged Ethernet [81].

Several recent works have also explored RDMA over Infiniband interconnects for software-based DSM and memory disaggregated systems [2, 17, 20, 27, 30, 40, 42, 46, 56, 62, 85, 98]. Using Infiniband System Area Networks (SAN) enables applications to access hardware from a user level, reducing user-kernel crossing latency during data transfer. It also provides a

high-bandwidth PCIe DMA based communication utilizing kernel bypass.

DeX, the existing version of Popcorn software DSM, [46] uses RDMA over Infiniband to enable inter-node communication. It maps the I/O buffer regions to a DMA-capable address to allow the Infiniband host controller adapter (HCA) to DMA from/to the buffers. It sends small kernel messages using Infiniband VERB to avoid costly DMA transfers. The data transfer between the nodes happens at 40 Gbps, and it achieves 3X speedup over traditional TCP-IP socket-based implementations [11].

Likewise, Grappa [62], a software DSM system designed to support data-intensive applications on commodity clusters, uses Infiniband over RDMA adapters. However, Grappa's runtime is implemented in C++, and it is mainly designed for homogeneous systems, unlike fDSM and DeX, which provide heterogeneous support and a C runtime.

Another distributed shared memory exploiting Infiniband interconnects is implemented in MENPS [27], an RDMA-based runtime system. It proposes two methods, a floating home-based approach, and a hybrid invalidation method, to overcome performance issues of conventional coherence protocols when implemented with RDMA. But unlike the sequential consistency protocol provided in fDSM, it employs a release consistency protocol, which introduces acquire and release synchronization operations.

GAM [17], a distributed in-memory platform that provides a directory-based cache coherence protocol, also employs the RDMA adapters. The authors of this work take it further and exploit special features of RDMA, like one-sided DMA and pure header notification, to improve GAM's performance. LITE [96] is another work that provides a Local Indirection Tier in the kernel to virtualize RDMA for the data center space. Applications can use LITE to perform low-latency network communication and distributed computations. It addresses the issues posed by native RDMA like resource protection and performance scalability and

provides a higher level of abstraction for applications. Finally, many other works utilize RDMA over Infiniband interconnects to enhance their software DSM system's performance and reduce the overhead of inter-kernel transmission [26, 33, 55, 65, 66].

Similar to all these works, the messaging layer of fDSM is also a PCIe DMA-based inter-node communication interface. But, instead of Mellanox Infiniband adapters, we utilize Xilinx Alveo U50 boards. We build on top of DeX's implementation and use the concept of pre-mapped messages using send/receive buffers. As opposed to the 40 Gbps link speed in RDMA, we operate at 100 Gbps and employ the Ethernet protocol to implement our messaging layer and achieve speedups over the RDMA based Popcorn implementation. Additionally, the Infiniband over RDMA interconnect does not provide the reconfigurability to implement additional hardware logic. In fDSM, we implement the DSM logic in the FPGA and minimize host intervention to maintain coherency between the nodes.

3.2 Hardware-Accelerated DSM systems

To further amplify the performance of software DSM and reduce the overhead of shared operations, like access control and protocol processing, industry and academia in the past have explored hardware-accelerated DSM options [1, 49, 51, 74, 77, 78].

The DASH (Directory Architecture for Shared Memory) [51] prototype is one such work that implements hardware cache coherence for large-scale shared-memory multiprocessors. The hardware provides a snoopy-based bus protocol for intra-cluster (set of processor nodes) coherence and uses a directory-based protocol to provide inter-cluster coherence.

Likewise, the FLASH (Flexible Architecture for Shared Memory) [49] multiprocessor integrates support for cache-coherent shared memory and message passing into hardware to

reduce software overhead. Each node in the FLASH system has a MAGIC chip (Memory And General Interconnect Controller) to implement intra-node and inter-node high-speed data transfers. This chip has an embedded processor that controls the optimized data path to move the memory and governs the coherence protocol.

MIT also contributed to the hardware DSM domain with their Alewife machine [1], a multi-processor architecture that supports up to 512 nodes connected over a scalable mesh network. It provides a coherent shared memory, integrated message passing, support for small fine-grain computations, and latency tolerance mechanisms. Each Alewife node has an Electronic mesh routing chip (EMRC from Caltech), comprising a Communications and a Memory Management Unit (CMMU) that services the data requests and maintains coherency.

All these works have one thing in common; they provide cache-line coherency between processors in a multi-core system using complex custom chips that are expensive to manufacture and give less flexibility for other protocols. As opposed to this approach, fDSM maintains coherency between rack-scale servers at a 4kB page-level granularity. It also allows reconfigurability to implement different DSM protocols like Lazy Release, Aggresive Release, and other consistency models [22, 29, 43].

Researchers have also attempted implementing these protocols on other hardware like GPUs to maintain coherency in the shared memory [73]. For instance, [79] implements a Relativistic Cache Coherence (RCC) protocol, a modified version of sequential consistency to close the gap between SC and weak models. Similarly, [5] implements a heterogeneous Lazy Release consistency protocol for GPUs. It builds upon bulk synchronization flush and invalidation, which is consistent with the current approach of GPU synchronization. Similarly, NoCs (Network on chips) and reconfigurable switches have also been used to implement DSM protocols [73].

Concordia [98] is another high-performance rack-scale DSM that uses an in-network cache-coherence protocol (FlowCC), implemented with Barefoot Tofino switches, RoCE network layer, and commodity servers. Its deals with cache block consistency and uses the programmable interface support available in the Tofino switches to implement the FlowCC protocol. Although the numbers look promising, the implementation of the FlowCC protocol is complex. It employs an ownership-migration mechanism requiring additional logic both on the host and switch and contributes to migration overhead. Also, Concordia is a user-space DSM, and applications are linked to the DSM library and call the read/write interface. fDSM contrasts this work by maintaining VMA coherency in the process address space and completely delegates DSM to the hardware. We provide DSM support as an OS abstraction and currently implement it for a 2-node setup opposing rack-scale servers. Also, from a programmer's point of view, reconfiguring the FPGA to accommodate different consistency models is easier than utilizing the PISA (Protocol Independent Switch Architecture) provided by reconfigurable switches.

More recent works explore the use of FPGA prototyping in the DSM and memory disaggregation domain. Tempest [77], a user-level shared memory system, uses FPGAs to prototype their Typhoon design, which combines an off-shelf processor and network interface to support cache-line coherency. Clio [30] is a system that maintains disaggregated memory at hardware-based nodes. The authors use FPGAs to prototype these hardware-based memory nodes and demonstrate high throughputs. [109] explores a DSM-based multi-core NoC architecture and evaluates it using an H.264 video decoder program. It implements the multi-core H.264 decoder on an FPGA and compares the performance with centralized memory systems. Unlike these works, fDSM provides an FPGA-based DSM infrastructure for a host and FPGA-based SmartNIC accelerator setup.

3.3 Cache Coherent Interconnects

The emergence of complex memory and storage applications are a few driving factors for hardware acceleration. FPGA Accelerators with High Bandwidth Memory (HBM) used as co-processors can address these memory and performance requirements. Due to these heterogeneous acceleration setups, industries have developed cache-coherent interconnects to maintain memory coherency between a host CPU and an accelerator.

CXL [23] (Computer Express Link) is one such open industry-standard interconnect that offers high-bandwidth and low latency connectivity between the host processor and an accelerator. It maintains a coherent memory space between the CPU and an attached CXL device, allowing both the devices to share the same memory resources, thus reducing data movement and stack complexity. Accelerators like GPUs and FPGAs connect to the CXL 2.0 switch that supports fan-out to multiple devices with the help of a resource manager.

CCIX (Cache Coherent Interconnect for Accelerators) [18] is another industry standard for maintaining cache coherency in a heterogeneous multiprocessor system. Its architecture has three layers built on top of PCIe - Transaction (for coherence messaging), Protocol, and Link (to govern the coherency protocol and act upon it). The home agent (host CPU) passes data pointers to the request agent (accelerator) to maintain coherency between their respective caches. Xilinx provides cache-coherent Versal ACAP devices that enable the CCIX functionality [101].

Likewise, CAPI (Coherent Accelerator Processor Interface) [88] was also developed by IBM to conveniently attach FPGA-based accelerators as a peer to the host CPU through its PCIe interface. It allows accelerators to access user-space memory directly and has a CAPI-based accelerator that can transfer data from the application memory without DMA. Although these interconnects use programmable logic to implement the cache coherence protocol, they

are not flexible for experimenting with other consistency models. Devices with CCIX/CXL/-CAPI capabilities are locked from a user's point of view and add an extra layer of complexity to the programmer while running applications. The reconfigurability and ease of running applications give fDSM an edge over these interconnects in a host and SmartNIC setup. Whereas in a distributed system setup, these interconnects wouldn't be beneficial since they provide cache-line coherency and do not DMA the host memory and transfer pages between the nodes, which fDSM can provide.

P. Vogel *et al.*, present an interesting work by recreating these coherent interconnects to provide virtual memory consistency using FPGA-based SmartNICs [97]. For example, [97] enables a transparent and shared virtual memory for FPGA accelerators in embedded SoCs. It provides a soft-core IOMMU and avoids DMA/copy-based host to accelerator communication. They implement Translational Lookaside Buffers and page table walk mechanisms to do virtual-to-physical address translation allowing the programmer to only share virtual address pointers to the FPGA accelerator. Currently, fDSM doesn't have these advanced capabilities, and this is a direction that we could explore in the future iterations of our work.

3.4 Baselines for Evaluating fDSM

To reiterate, **fDSM** targets to provide DSM between a host CPU and the CPU of a smart accelerator like a SmartNIC. Since we design a prototype that emulates this setup, there are no head-on comparisons to our work. As discussed in the previous section, although cache-coherent interconnects are readily available, it wouldn't be appropriate to compare our work to them since they provide cache-line coherency and adopt a DMA-free approach. fDSM contrasts cache-coherent interconnects by providing virtual memory DSM at 4KB page granularity and moves the application data into the local memory of an accelerator (in

this case, a remote node) using a DMA-based approach.

In addition, fDSM also delivers a reconfigurability argument, meaning it can provide a customizable DSM that gives an end-user the privilege of choosing the memory coherency protocol based on the application's needs. For example, applications like Cholesky decomposition and FFT benefit from eager release consistency models, and Blocked-LU and Barnes-hut greatly benefit from lazy release consistency models [48]. Unfortunately, there hasn't been a lot of research exploring the advantages of having a customizable DSM framework.

Therefore, we compare fDSM to Popcorn Linux's software DSM, which employs a 10-Gigabit TCP-IP-based messaging layer and a state-of-the-art 40GBps Infiniband RDMA messaging layer. We present the speedups of fDSM over both the messaging layers and highlight the software latencies avoided by adopting the hardware-support DSM framework.

Chapter 4

FPGA Messaging Layer

The first part of our contribution is the FPGA Messaging Layer, a high-bandwidth and a low-latency communication interface between the nodes in Popcorn. In this chapter, we cover the design and implementation aspects of the FPGA-based Messaging layer, first by providing an overview of the setup in Section 4.1. Then, we go into the details of the messaging layer hardware logic in Section 4.2, followed by the details of the FPGA driver in Section 4.3.

4.1 Overview

As described earlier, in any distributed system, the communication/message-passing layer must be performance-critical [19]. The latency of inter-kernel communication is one of the most crucial aspects of our design that decides the performance benefits of our work. Moreover, it should provide flexibility and robustness to handle the complicated communication usage in Popcorn and the high migration traffic between the nodes. For that reason, we designed and implemented an FPGA-based messaging layer to facilitate the migration of messages and pages between the two nodes with minimal host intervention. The FPGA-based messaging layer has two integral components - Hardware logic and the FPGA driver (explained in Sections 4.2 and 4.3). They work in tandem to transfer the Popcorn kernel messages between the nodes.

Figure 4.1 shows our heterogeneous (x86 + ARM) hardware setup for two nodes, and Figure 4.2 shows a birds-eye view of the FPGA Messaging layer for the same setup. Similar to DeX [46], the primary objective of this work is to extend the execution boundary of an application over multiple nodes (node-0 and node-1 in this case) to achieve maximum performance.

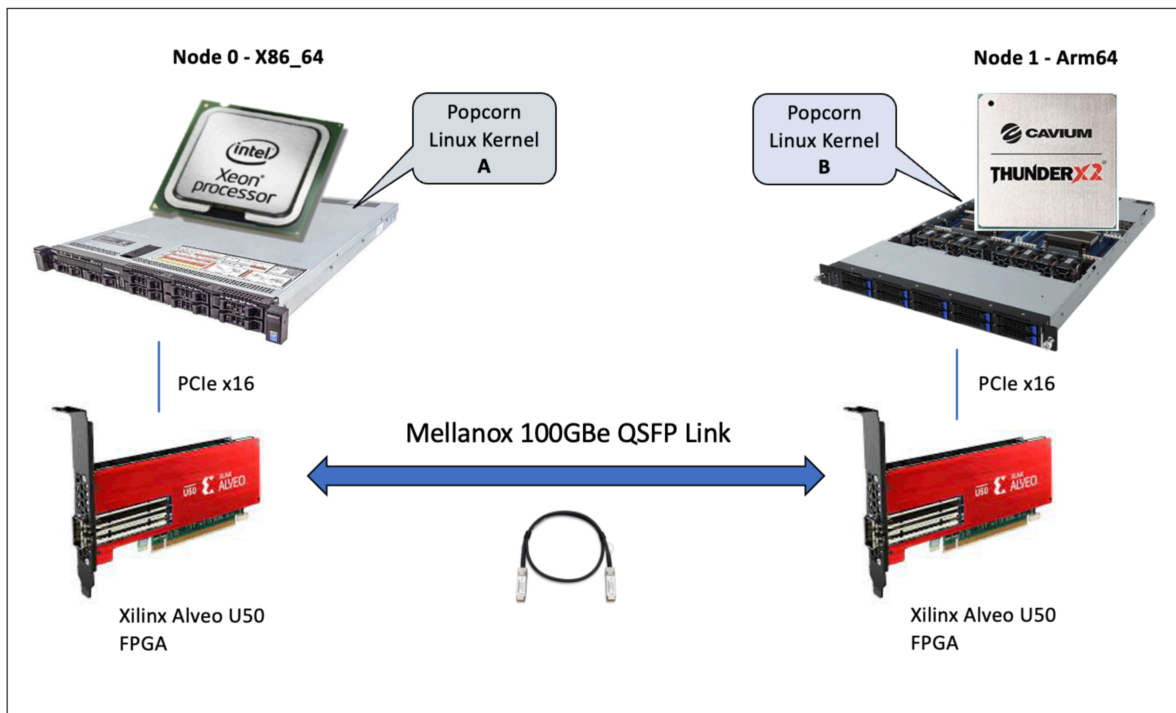


Figure 4.1: Hardware Setup

We chose the Xilinx Alveo U50 Accelerator Card with a Generation 3 X16 PCIe interconnect and 100Gbps QSFP28 port so that fDSM can leverage the high-bandwidth and low-latency communication provided by these interconnects. The Alveo U50 FPGA card is widely used in datacenters to accelerate memory-bound and compute-intensive applications in financial computing, data analytics, and machine learning inference workloads [35]. It also provides us with 872K Lookup tables, 1.7M register slices, and 200Mb of Block RAM [100], which we utilize to implement a hardware-accelerated sequential consistency protocol. As a result, we delegate the work of tracking the page ownership and the maintenance of the coherency protocol between the two nodes to the hardware. This is explained in detail in Chapter 5.

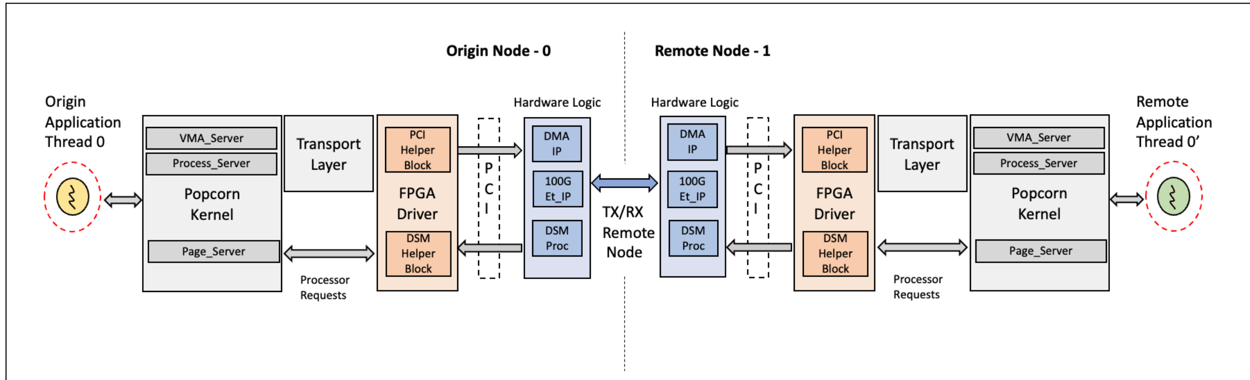


Figure 4.2: Overview of FPGA Messaging Layer

4.2 Hardware Logic

We design the hardware logic using the Xilinx Vivado Development Suite. The design comprises Xilinx™ IPs and custom hardware logic blocks to enable communication between the FPGAs connected to the hosts, as shown in Figure 4.3. In the following subsections we describe the IPs used in our design.

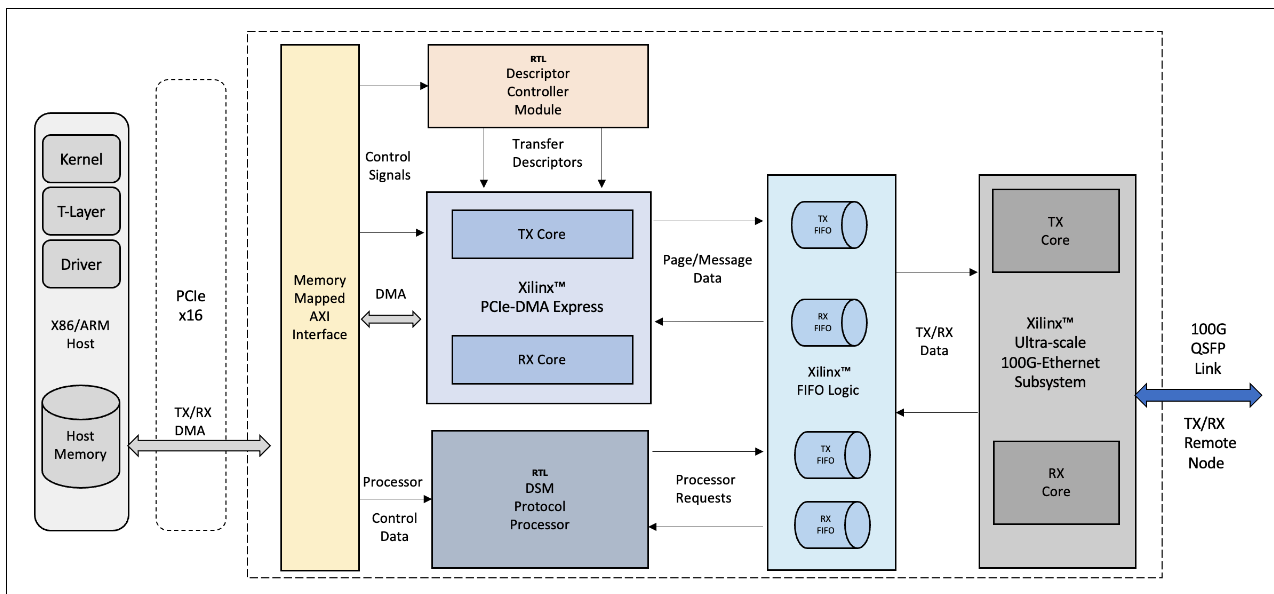


Figure 4.3: FPGA Hardware Logic

4.2.1 PCIe-DMA Express™

The DMA Subsystem for PCI Express implements a high-performance DMA Engine compatible with PCIe 3x, allowing data movement between the host memory and the DMA subsystem [107]. We chose AXI-Stream as the data-transfer protocol, which the whole design runs on. We select one channel of H2C (Host to Card)/TX and C2H (Card to Channel)/RX each. The DMA IP (XDMA) uses a list of descriptors that specify the source address, destination address, and length of the DMA transfer. As opposed to the descriptor fetch mode, in our design, the IP runs in the descriptor bypass mode, where the host provides the descriptors through the custom logic (Descriptor Bypass Controller). In addition to the DMA logic, the subsystem provides a PCI Configuration space to control the data-transfers and configure the IP. It also exposes specialized memory-mapped regions, called the Base Address Registers (BAR), which act as an interface for the host driver to access and configure the registers of other IPs in the design (shown in Figure 4.3).

4.2.2 Ultrascale 100G Ethernet Subsystem™

The 100G Ethernet Subsystem serves as an interface between the user hardware logic and the high performance and low latency 100 Gigabits per second port of the Xilinx Alveo FPGAs. We use it to transmit and receive the page data/processor requests from one node to another at very high speeds. In our design, the communication happens through the PCS + MAC interface, and a packet transmitted from a node's TX core arrives at its peer's RX core or vice versa [105]. We configure the Ethernet IP using its memory-mapped register space. For example, we set the `ctl_tx_enable` and `ctl_rx_enable` bits in the *ETH.CR* (Control Register with an offset of `0x00`) to enable TX/RX in the IP.

4.2.3 FIFO Logic™

FIFO generators were used in the design to provide a First In First Out memory queue for ordered storage and retrieval while utilizing minimum hardware resources [104] [106]. We use the RX FIFOs to store the page data/processor requests and the TX FIFOs to store and forward data to the Ethernet Subsystem. The XDMA IP writes and reads the data from the corresponding FIFOs (DMA Write → TX FIFO; DMA Read ← RX FIFO). We chose a depth of 1024 512-bit words, i.e., a capacity of 64 kB per FIFO, and the programmable threshold constant as 8 words, meaning the FIFO asserts the 'FIFO Full' signal when the DMA writes 8 512-bit words (512 bytes of data) in it. We use this signal to raise a user interrupt to the host to initiate the C2H transfer. Since the IPs in our design run at different clock frequencies (XDMA - 250 MHz; Ethernet Subsystem - 312.5 MHz; Initialization Clock - 100MHz), these FIFOs also act as IP synchronizers to avoid Clock Domain Crossing errors.

4.2.4 Descriptor Controller Module

The Descriptor Controller Module, or the Descriptor Bypass Controller, is a custom RTL module designed by [82] to forward the information of the DMA transfer from the host to the descriptor bypass port of the XDMA IP. The descriptor, as shown in Figure 4.4, is a 173-bit linked list to describe the memory transfer that the DMA subsystem has to perform. We use two controllers for the TX and the RX channels of the XDMA IP, respectively.

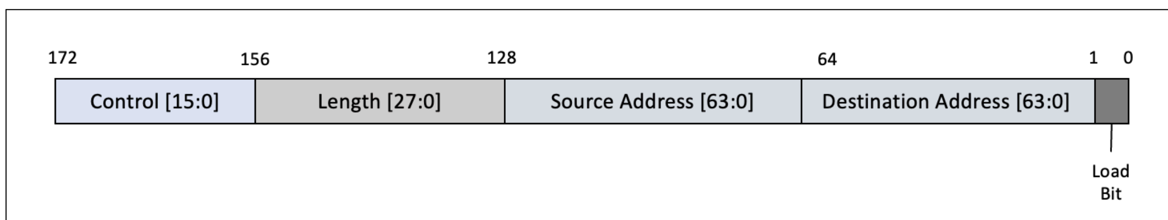


Figure 4.4: DMA Descriptor

The host driver writes the transfer length, MSB, and LSB of the source and destination DMA addresses (if any) and the descriptor control (to indicate End of Packet and interrupt the host upon completion) in the 32-bit register space using the offsets. Then, the driver sets the `start` bit in the control register that signals the controller to package this data into a 173-bit descriptor and load it in the descriptor buffer of the XDMA IP.

4.2.5 DSM Protocol Processor

The DSM protocol processor, or as we call it, the DSM Processor is the main contribution of our work and the pivotal part of the design. It is responsible for keeping track of the page ownership and maintaining the coherency between the nodes at a page-level granularity. The host communicates with the processor through the memory-mapped AXI interface and controls it by programming the register space. Chapter 5 explains the DSM processor in detail.

4.3 FPGA Driver

The FPGA driver acts as the interface between the Transport layer of Popcorn and the Xilinx Alveo interconnect. We implement the FPGA driver in the 5.2.21 version of the Linux kernel, with ≈*2.1K* lines of C code. We further break down the driver into many sub-components to describe them in detail. Figure 4.5 illustrates these sub-components of the FPGA driver.

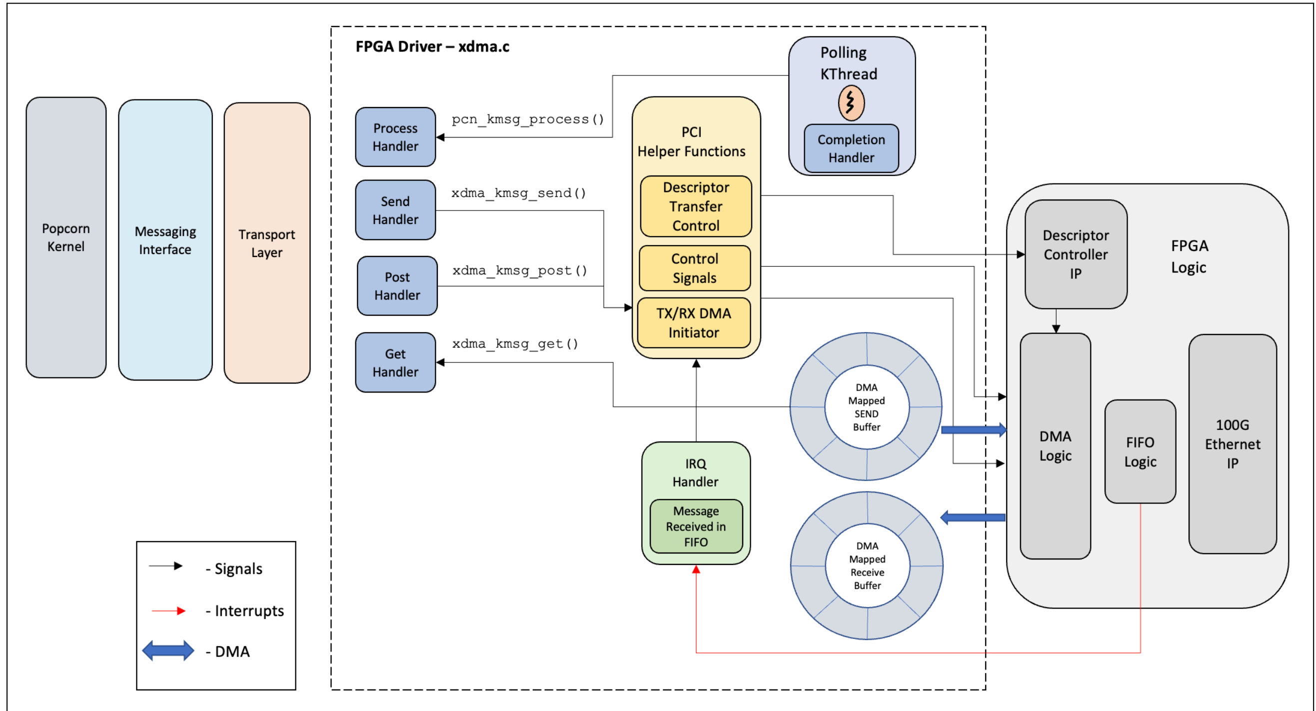


Figure 4.5: FPGA Driver

4.3.1 PCIe Helper Block

We implement PCI helper functions in the kernel (popcorn-kernel/kernel/popcorn/) and the driver to communicate with the hardware logic and initiate data transfer between the host and the FPGA. As described earlier, the DMA subsystem in the hardware exposes BARs (memory-mapped regions) to configure and control the logic in the FPGA. Using this region, we trigger the DMA by setting a few bits in the control registers of the XDMA IP and the Descriptor Controller with the help of these functions and PCI Configuration Space (explained in Section 4.2). We also implement a PCI header file that holds all the memory-mapped addresses of these control registers and their corresponding offsets. We also configure the Ethernet IP block using the PCI Configuration Space. We use IOMEM operations such as `ioread32()` and `iowrite32()` to communicate with the memory-mapped registers. Do note that, the helper block is synonymous with helper functions.

We use Linux APIs in the helper block to initialize and register the FPGA to the kernel's address space, enabling it to access the host memory and interrupt it. Also, these APIs help to expose the I/O resources of the FPGA, giving the developer full access to the PCI configuration space and control registers.

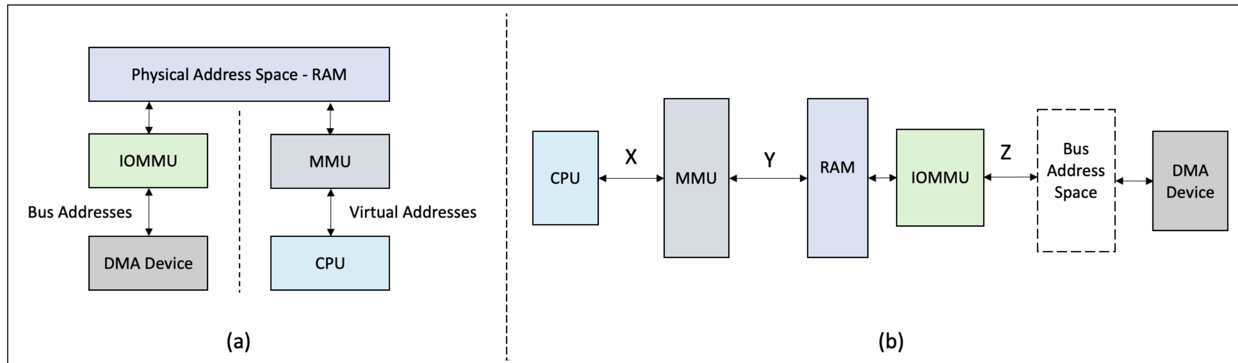


Figure 4.6: (a) Comparison of IOMMU and MMU (b) Kernel Memory Layout [24]

Furthermore, in our current implementation, the FPGA can't bypass the kernel's memory protection and access the VMA of a process by walking the PTEs. For instance, the driver allocates a buffer with `kmalloc()`, which returns a virtual address X. The kernel maps this address X to a physical address in the RAM, say Y. The driver can use the virtual address X, but the device can't because the DMA doesn't go through the virtual memory. In simple systems, the device can directly do a DMA to Y. But in our setup, the x86 IOMMU (Input Output Memory Management Unit) or the ARM64 SMMU (System Memory Management Unit), translates the physical address to a DMA-capable address Z for the external devices to use or vice versa as shown part (a) of Figure 4.6. To achieve this address translation, we use DMA Mapping APIs like `dma_alloc_coherent()/dma_free_coherent()` to convert a 64-bit virtual address X to a 64-bit DMA capable Bus Address Z. The kernel memory layout and the interaction between the IOMMU and the CPU is shown in part (b) of Figure 4.6.

Note: We include some helper functions in the kernel to make it accessible to the Popcorn page server (for fDSM).

4.3.2 Polling Thread Handler

The XDMA IP provides two ways of intimating the host CPU upon DMA completion - Polling and Interrupts. We choose the Polling method as it results in a better DMA data rate and avoids context switching and interrupt masking/enabling latencies on the software side. In this method, the XDMA writes the count of completed descriptors to a user-provided writeback address. We allocate two writeback addresses to monitor the completion of H2C (Host to Card) and C2H (Card to Host) DMA transfers. We also implemented a Kthread handler in the driver (Polling thread handler) to poll these addresses and take the appropriate actions upon a successful DMA. For instance, if a C2H operation is successful (i.e., the host receives a Popcorn message), the Kthread handler fetches the memory location of the receive buffer's tail and passes it to the `pcn_kmsg_process()` function, as shown in Listing 4.1.

```
static int poll_dma(void *arg0) {
    struct xdma_poll_writeback_address h2c_poll_bus, c2h_poll_bus;
    if (h2c_poll_bus->completed_desc_count != 0)
        /* Host to Device transfer done
         * Revert the count back to zero and resume polling */
        h2c_poll_bus->completed_desc_count = 0;
    else if (c2h_poll_bus->completed_desc_count != 0)
        /* Device to Host transfer done
         * Get the receive buffer tail and invoke process handler
         * Revert the count back to zero and resume polling */
        pcn_kmsg_process(receive_buffer->tail->addr);
        c2h_poll_bus->completed_desc_count = 0;
}
```

Listing 4.1: Polling Thread Handler

4.3.3 Send and Receive Buffers

As discussed earlier, the XDMA subsystem performs the DMA using bus addresses. Therefore, the DMA mapping of the virtual addresses of a myriad of messages can be costly and impact the overall performance of Popcorn. To avoid this, we implement Send and Receive buffers in the driver, which are DMA-ready so that Popcorn can communicate without the overhead of translating the virtual addresses of the messages. These buffers are Ring buffers consisting of physically continuous pages mapped to DMA-capable address ranges. A thread can get a pre-mapped memory region from the pool and compose an outgoing message.

For instance, before the Process server sends out a migration request to the remote node, it obtains a pre-mapped memory region from the buffer by calling the messaging interface `pcn_kmsg_get()` function (invokes `xdma_kmsg_get()` in the driver) and packages the request data into it. Then it sends the message by calling `pcn_kmsg_send()`, which invokes the send handler in the driver (`xdma_kmsg_send()`). Since the buffer containing the message is DMA-ready, the send handler transfers the memory region's descriptors and triggers the DMA, with the help of the descriptor control and DMA initiator blocks. Then the XDMA IP reads the memory location pointed by the descriptor and transfers the message contents to the remote node through the hardware logic.

Similarly, the driver holds a receive ring buffer to enqueue the incoming messages. The driver initializes this buffer during the setup phase with DMA-mapped memory regions with each chunk being the size of 8kB. The descriptors to these regions are transferred to the RX channel of the XDMA IP, which then writes the incoming data to the corresponding regions. Upon the reception of the messages in the buffer, the polling thread passes the virtual address of the memory region to the `pcn_kmsg_process()` (messaging interface handler), which then invokes the callback functions corresponding to the received message type.

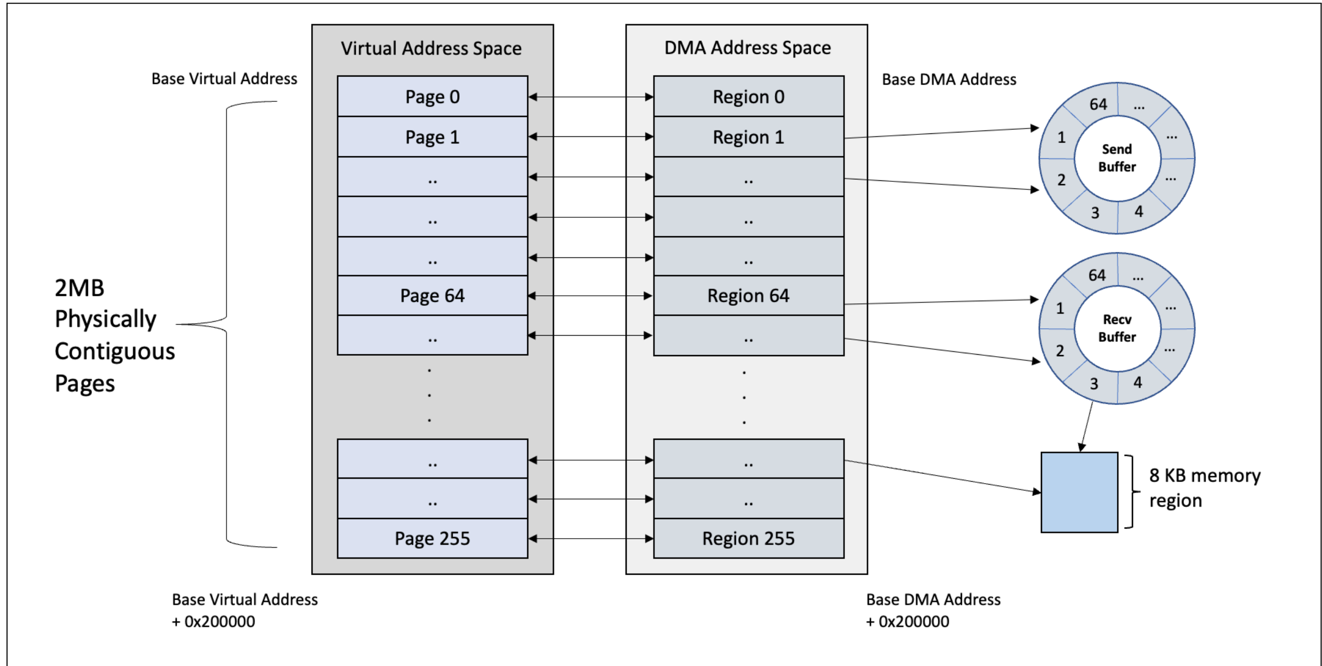


Figure 4.7: Send and Receive Buffers memory allocation

Typically, programmers go through a tedious process of obtaining the virtual address of a single memory region using `kmalloc()` and then mapping it to a DMA address using streaming Linux APIs like `dma_map_single()`. Since we use ring buffers, we allocated a massive physically-contiguous memory region of 2 MB (standard granule size in ARM64 machines) using the Linux API dma_alloc_coherent(), which automatically points to the base virtual and DMA address [39]. We divide this 2 MB region into 256 x 8 KB chunks and use the first 64 for the send ring buffer and the next 64 for the receive ring buffer, as shown in Figure 4.7.

4.3.4 IRQ Handler

The IRQ handler in the driver is an ISR (Interrupt Service Routine) that manages the interrupts from the FPGA hardware. Currently, our hardware design is not independent of

the host CPU involvement. There isn't a way for the XDMA subsystem to automatically realize that it has received data in the RX FIFOs, and it has to transfer that to the host memory. To enable the communication between the RX FIFOs and the XDMA system, we utilize the UI (User Interrupt) vector provided by the XDMA IP. The RX FIFO interrupts the host through the UI vector, and the IRQ handler in the driver traps it. The ISR masks the interrupt vector and transfers the descriptors of the receive buffer tail to the descriptor controller, and initiates the C2H (Device to Host) DMA. Then it re-enables the interrupt for future use and returns to the process context. The ISR handling a user interrupt is shown in Listing 4.2. The ISR also traps the fault interrupts raised by the DSM processor, which is explained in Section 5.1.

```
1  static irqreturn_t xdma_isr(int irq, void *dev_id) {
2      unsigned long read_usr_irq = ioread32(usr_irq_offset);
3      if (read_usr_irq)
4          /* User Interrupt Received - Mask it
5           *  Transfer descriptors and re-enable interrupt */
6          user_interrupt_disable(read_usr_irq);
7          transfer_descriptor(receive_buffer->tail->dma_addr);
8          initiate_transfer(C2H);
9      return IRQ_HANDLED;
10 }
```

Listing 4.2: Interrupt Service Routine

Chapter 5

fDSM Design

In this chapter, we explain the design and implementation of our main contribution, **fDSM**, an FPGA-Accelerated DSM framework for heterogeneous ISAs. **fDSM** combines the DSM processor and the FPGA messaging layer, which work together to maintain virtual memory coherency between two connected nodes.

In Section 5.1, we explain the design and implementation of the DSM processor that manages the Sequential Consistency DSM protocol. In Section 5.2, we describe the changes made to Popcorn Kernel to support fDSM, and contrast it with Popcorn's software version.

5.1 DSM Processor

Undoubtedly, the DSM processor is our main contribution, and we implement a hardware-accelerated, or as we call it, an FPGA-accelerated sequential consistency protocol with minimal host software intervention. It is responsible for keeping track of page owners, updating the status of the pages, and maintaining memory coherency between the nodes. We designed the DSM processor using mixed Hardware Description Languages ($\approx$ *1.6K* lines of Verilog and VHDL code) and performed functional verification using Verilog test benches in the Vivado Design Suite. As shown in Figure 5.1, the DSM Processor consists of the following sub-components.

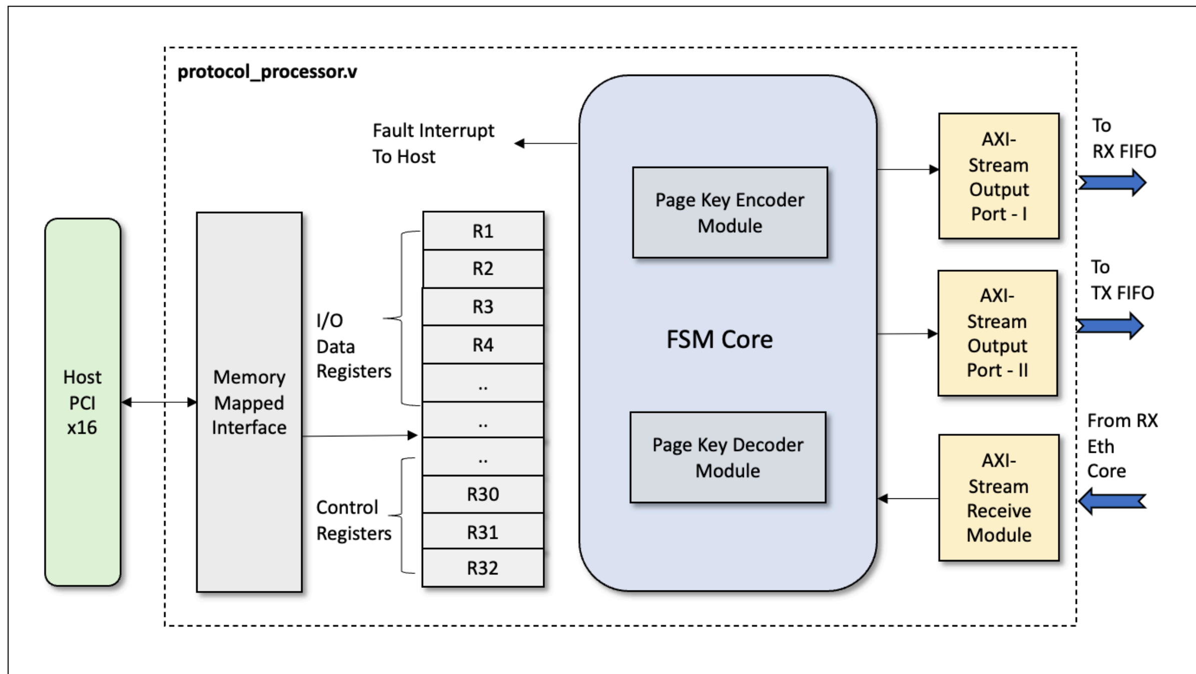


Figure 5.1: DSM Processor

5.1.1 Memory Mapped Interface

The Memory Mapped Interface connects to the XDMA AXI Interface that the host uses to communicate with the hardware logic in the FPGA. It uses a 32-bit addressing scheme and the AXI Read-Write protocol. As discussed earlier, this interface enables the host to access the Control and I/O registers in the processor's register space.

5.1.2 Register Space

The DSM processor has a 32-bit Register space configured by the host through the AXI memory-mapped interface. We use the 32-bit addressing scheme provided by the AXI protocol to access these registers. There are 32 registers, and we categorize them into Control and Input-Output. Table 5.1 shows a sample set of registers and their offsets.

32-bit Register Space			
Register Number	Register Name	Register Offset	Description
R0	`proc_pkey_msb`	0x00	Page Key MSB
R2	`proc_vaddr_lsb`	0x04	Page Virtual Address LSB
R6	`proc_fflags_lsb`	0x18	Page Fault Flags LSB
R8	`proc_iaddr_msb`	0x20	Page Instruction Address LSB
R12	`proc_mynid`	0x34	Local Node ID
R14	`proc_ctl`	0x38	Processor Control Register

Table 5.1: Sample Register Space

Since the registers are 32-bit, we split the 64-bit page addresses and keys into two 32-bit values and write them to corresponding LSB and MSB registers. Then, the DSM processor concatenates these 32-bit values to process and forward. We write these registers only when there is a local page fault on the local node. We avoid using IO operations to read these registers, as they are very time-expensive, with each ioread32 costing 0.8 microseconds. Instead, the DSM processor forwards the packets to RX FIFO, which signals the XDMA IP to transfer the contents to the host.

5.1.3 AXI-Stream Input/Output Modules

AXI Stream is a subset of the AMBA AXI protocol introduced by ARM in 1996 [7]. It is mainly used in the data-centric paradigm where the concept of memory-mapped address is not necessary [103]. Most of the specialized IPs from Xilinx, like the 100G Ethernet Subsystem, the FIFOs, and the XDMA IP, are designed to accommodate the AXI Streaming protocol. The protocol gives a developer more control over the transmission and reception of packets. For these reasons, we employ the streaming protocol for the processor packets to optimize the design performance.

To understand Figure 5.2, we need to know the AXI-Stream protocol. It mainly has 5 signals. The `TVALID` bit indicates the status of the data packet. A high value means that

the streaming packet is valid, else it is corrupted. The TDATA bus transmits the data packets between the IPs. In our implementation, the TDATA bus is 512 bits wide. The TKEEP signal also indicates the validity of the TDATA. All high means the data is valid. The TREADY bit informs the transmitting IP that the receiving module can accept the incoming packets. Until it's high, the transmitting IP waits to send the data. The LAST signal indicates the EOP (End of a packet stream). We use these signals to implement the Input-Output modules of the processor.

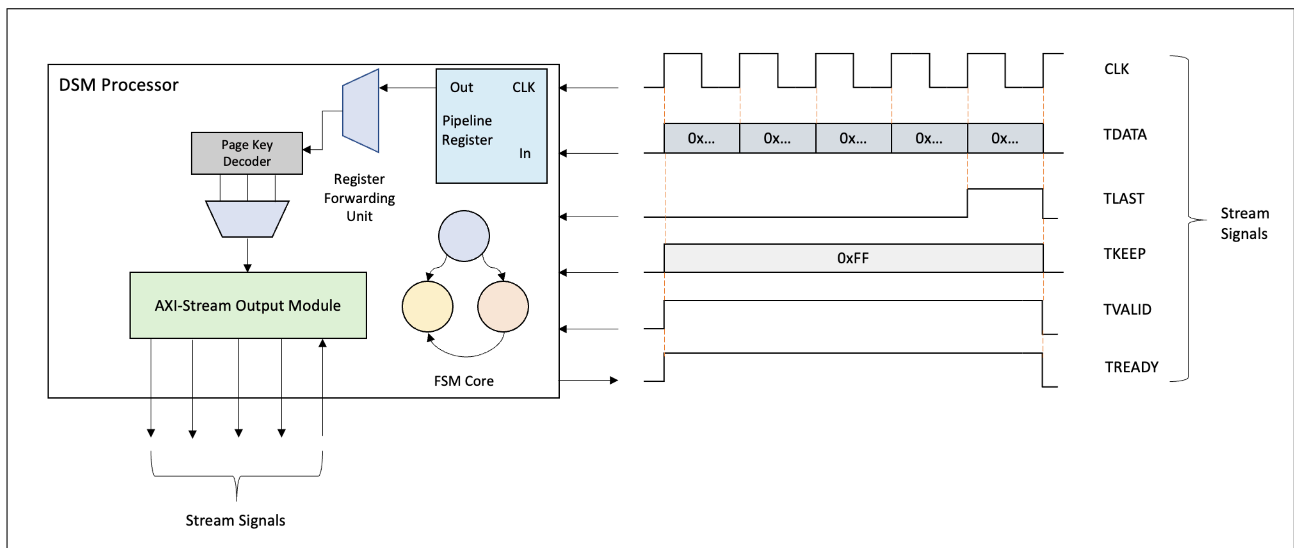


Figure 5.2: AXI-Stream Input Output modules

The Input-Output modules are external wires in the processor that receive and transmit the packets using the streaming protocol. They have pipeline registers that can store and forward the packet data for computation/transmission. For instance, as shown in Figure 5.2, upon packet reception, the receive pipeline register stores the first packet and forwards it to the page decoder module for processing. In the next clock cycle, the decoder module parses the data and instructs the host about the type of fault, potential owner, and the appropriate action to be taken (Shared or Invalidated).

5.1.4 FSM Core

In our implementation, the FPGA doesn't have the capability of walking the process PTEs to figure out the status/the owner of the VMA pages. In the software DSM version of Popcorn, we keep track of the pages using MMU, PTE APIs, and kernel data structures. Whereas, in fDSM we don't involve the host to do so. Instead, we introduce the concept of a Page Key that acts as a token and helps the DSM processor to understand the faulty page's details. The Page Key is a 64-bit UID (Unique Identifier) for all the pages in the VMA of the application, and Figure 5.3 shows its structure and fields. When a page fault occurs, the host writes the fault data (including the page key) into the register space of the DSM processor and triggers the Page Key Encoder module to formulate the key using the given data. We only involve the host to store these page keys in a Linux Radix tree for future reference. By doing this, we avoid having complex data structures in the hardware to store the page information.

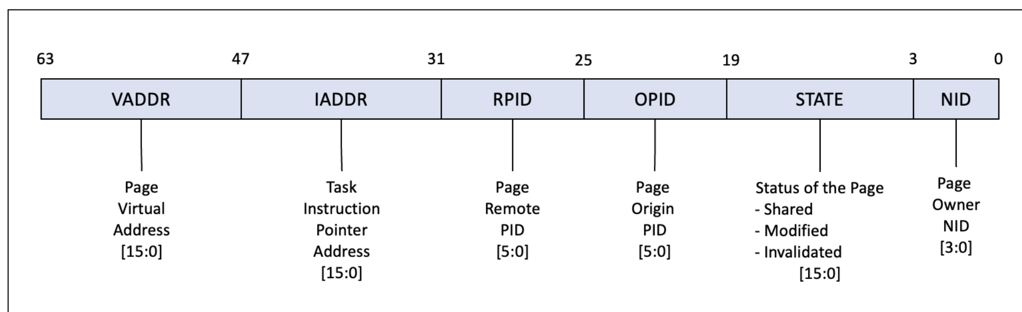


Figure 5.3: Page Key

The FSM core is implemented in Verilog and is a crucial part of the DSM Processor. It is responsible for governing the DSM SC (Sequential Consistency) protocol. We implement handlers for every type of fault (remote fault at origin, local fault at remote, etcetera) in the state machine. The host invokes the local fault handlers by writing the fault data and triggering the processor. Conversely, the remote fault handlers get going when the request

from the remote DSM processor arrives.

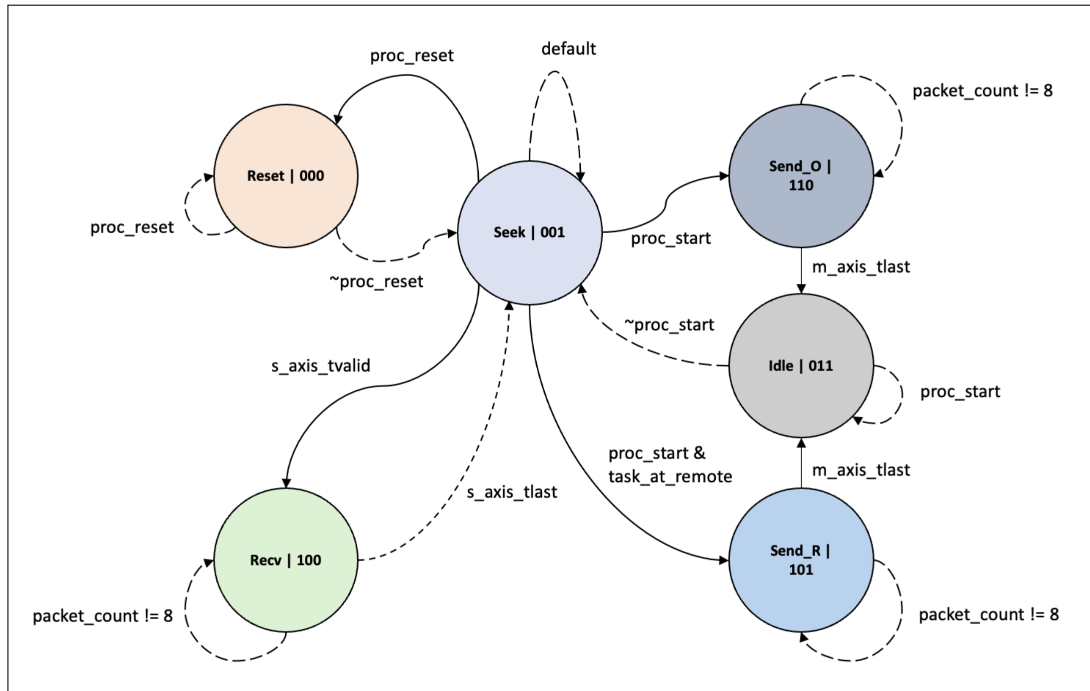


Figure 5.4: FSM Core

Seek is the default state of the processor, where the processor spins and waits for incoming data or some signal from the host. Upon a local GPF, the host kernel writes all the page information, and triggers the DSM processor by setting the `proc_start` bit in the control register (`proc_ctl`). As shown in Figure 5.4, the FSM changes states depending on the node (origin/remote node) the faulting task is running on. For instance, if a local page fault occurs on a task running on the remote node, the `local_fault_at_remote` handler is invoked in the DSM processor (we derive this concept from Popcorn's kernel code). In this case, the FSM switches to the Send Remote (**Send_R**) state, packages all the request information into an AXI Stream packet, and sends it to the destination node's DSM processor by setting the appropriate AXI-Stream signals. The FSM remains in the **Send_R** until it has sent out eight packets through the Output Port - 2. We chose 8 as the number of packets as the programmable full threshold constant of the FIFO is 8. Likewise, a local fault on the origin

node will invoke the `local_fault_at_origin` handler and the FSM switches to the Send Origin (**Send_O**) state. After sending the packets, the FSM goes into the **Idle** state and waits for the host to clear the `proc_start` bit. This is to avoid re-sending the packets till the `proc_start` bit is high.

Similarly, upon the reception of a packet, i.e., when the receiving module's `TVALID` bit is high, the FSM jumps to the **Recv** state. As discussed earlier, the pipeline registers forward the first packet of the stream to the decoder module, and it processes it. Once the processing completes, it starts forwarding the packets through the Output Port - 1 to the RX FIFO, which raises a user interrupt for the DMA to transfer the contents.

The FSM core encodes the transmitted packets with keywords to help the hardware logic distinguish them from the DMA packets. Since we use a single channel of the Ethernet Subsystem to send both the DMA and the processor packets, these keywords can help the arbiter switches and stream monitors we implement in our design. We implement these custom RTL modules to reroute the incoming processor packets to the DSM processor and avoid bus contention.

It is important to note that the Page Key Encoder module doesn't compose a unique key every time a page fault occurs on a particular address. It does so only the first time a page fault occurs on a particular address. For instance, if there is a page fault on a virtual address for the first time, say `0x1000_0000`, there isn't a page key stored for this address yet (all initial page faults happen in the remote node, where the developer migrates the task to). In this case, the host retrieves a NULL value from the radix tree and writes the same to the page key registers. The handlers in the FSM core compose a new key for `0x1000_0000`, and the host on both the nodes store this new value for future fault reference (if any).

We centralize the fDSM, meaning the origin node (the node that initiates task migration)

makes all the major protocol decisions. This implies that the fault handlers in the origin node are where the entire DSM action is. The modifications made by the origin on the page key are final, and the remote doesn't change it.

To avoid protocol errors, we implement fault inspection in the FSM Core. In the case of an outlier GPF, the FSM Core raises a fault interrupt to the host, and execution halts. For example, the remote node raising a read fault on a page it already owns is an exception. The fault inspection code in the FSM core detects this and raises an interrupt to the host.

5.2 Popcorn Kernel Support

To incorporate fDSM with Popcorn, we modify its kernel, especially the Page Server and Messaging Interface (with $\approx$*1.2K* lines of C code). We implement separate handlers in the Page Server to accommodate fDSM requests. To avoid maximum host-software intervention, we move the software DSM code in these handlers to the hardware and only retain the PTE and MMU update functionalities. We include the radix-tree functions for storing and updating the page key and IOMEM operations to configure the DSM processor. We also register them to the Popcorn callback function queue. By doing this, the messaging interface invokes these handlers upon the arrival of an fDSM remote page request. Figure 5.5 shows the flowchart of the kernel handling a remote page request.

As discussed in the previous chapters, the driver checks whether a callback function is associated with the received message type. If there isn't, the FPGA driver considers the received message as an fDSM request, as the DSM processor only sends RPR or invalidation requests to the host. Then, instead of the Popcorn process handler, the driver invokes our customized `xdma_pcn_kmsg_process()` function, which maps the received message to a processor request structure (`dsm_proc_request_t`). Then, the process function reads the

request and based on the type, invokes the corresponding handler, and passes control to the page server. In the case of an RPR, the page server then fetches the `task_struct` using the PID of the request and calls the remote fault handlers for the respective nodes. The remote fault handlers find the process VMA, prepare the PTEs, and take the action specified in the request (Share/Invalidate).

Whereas in software DSM, the kernel processes the page information, reads the fault flags, checks the page owner, and decides the course of action. We delegate this to the hardware in fDSM. Upon the DSM step completion, the kernel prepares the page, copies it to the remote page response, and sends it to the destination node. We derive this concept from the TCP-IP driver, and piggyback the page with the response to avoid two separate messages.

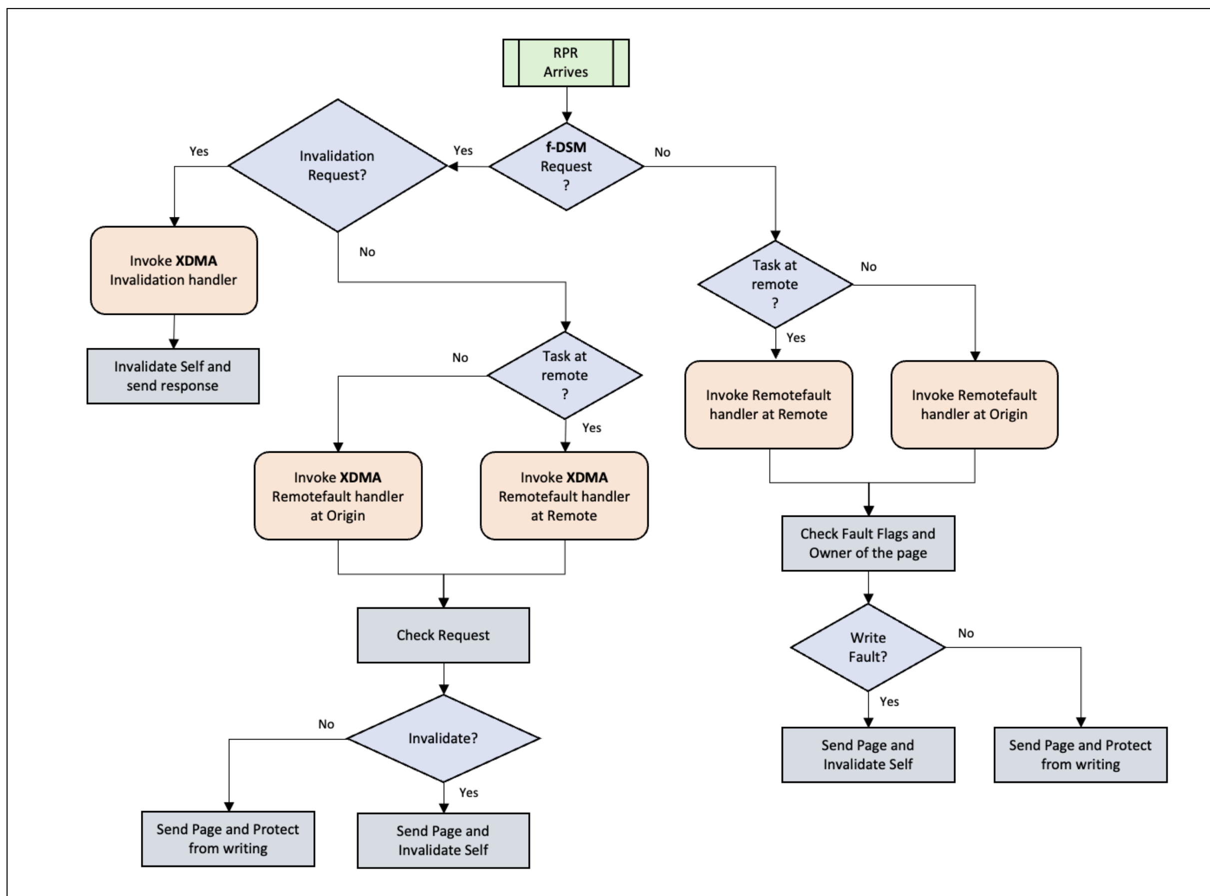


Figure 5.5: RPR Handling Flowchart

Figure 5.6 shows the flowchart of a local fault handling in fDSM and Popcorn software DSM. In Popcorn, the kernel decides whether to send an invalidation message or an RPR. It has to go through a tedious process of parsing the fault flags, checking the page owner and status, and updating the information accordingly. Whereas in fDSM, the kernel fetches the page key and writes all the page information to the DSM processor's register space, as shown in Listing 5.1. By doing so, we offload all the above-mentioned steps to the hardware. The similarities between the two are that in both cases, the kernel prepares the PTE, creates a wait station, and waits for a response. Upon response reception, the fDSM handlers process it, and based on the type of response (invalidation or remote page response), it modifies the PTE. We do this to distance the host from the decision-making procedure.

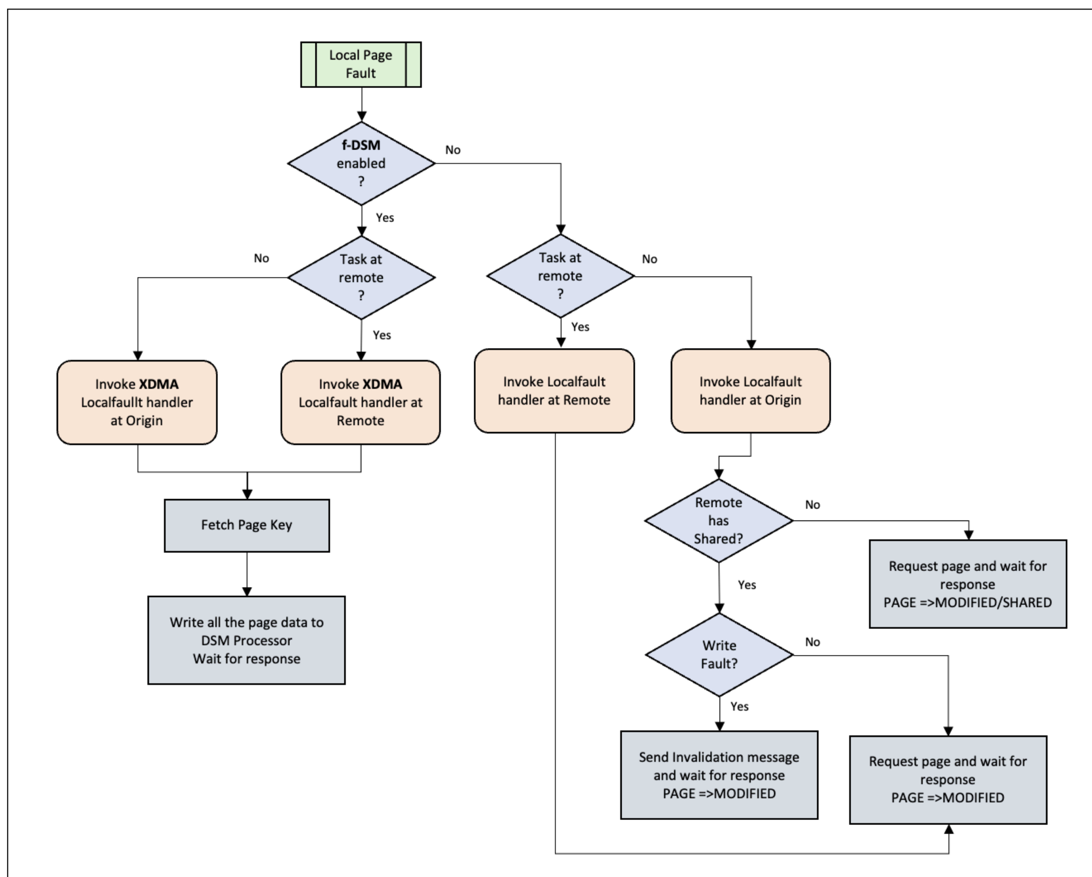


Figure 5.6: Local fault Handling Flowchart

```
/* fDSM Local Fault Handler */
void xdma_handle_localfault_at_origin(struct vmfault *vmf) {
    /* Fetch page key from radix tree for fault address */
    fetch_pkey(vmf->addr) => radix_tree_lookup(addr);
    /* Send remote page request by writing the processor registers */
    iowrite32(proc_vaddr_msb, msb(vmf->addr));
    /* Write other page information */
    /* Wait for remote page/invalidation response */
    wait_at_station(station_id);
    return 0;
}
```

Listing 5.1: fDSM Local fault handler pseudocode

Chapter 6

Experimental Evaluation

In this chapter, we describe the experimental evaluation of fDSM on the heterogeneous and the homogeneous setup and investigate the following questions:

1. *How does the performance of fDSM compare to software DSM implementations?*
2. *Can our FPGA-based messaging layer outperform high-speed conventional interconnects like Infiniband RDMA? If yes, what and where are the savings?*
3. *What type of applications benefit from our framework?*

We provide an overview of our experimental setup and the benchmarks we use in Section 6.1. In Section 6.2, we present the comparison of our heterogeneous results, followed by our comparison of the homogeneous results. In Section 6.3, we examine our performance and discuss it with the help of additional metrics. We summarize our results in Section 6.4.

6.1 Overview

The heterogeneous machines comprise a 16-core (2 threads each) Intel(R) Xeon(R) 4110 CPU (64-bit x86 ISA) server with 192 GB of RAM running at 2.10 GHz and a 28-core (4 threads each) 2 x Cavium ThunderX2 CPU (64-bit ARM ISA) server with 128 GB of RAM

running at 2.0 GHz. Our homogeneous setup includes two of the x86 machines mentioned above. All these machines run Popcorn Linux v5.2.21.

To evaluate the fDSM's performance, we chose the serial version of the NASA Parallel Benchmarks (NPB v3.3) [61]. The NPB suite is a set of applications derived from Computational Fluid Dynamics (CFD) problems, and they help evaluate the performance of parallel supercomputers. It consists of five kernels (IS, MG, FT, EP, and CG), three pseudo applications (BT, SP, and LU) that mimic the data movement in CFD applications, and one unstructured computation benchmark (UA). We selected three standard classes of problem size (A, B, and C).

We also chose the Blackscholes (BLK) partial differentiation application from the PARSEC benchmark suite [36], which focuses on shared-memory programs. As discussed earlier (Chapter 3), we use the performance of Popcorn with TCP-IP and Infiniband RDMA messaging layers as our baselines.

6.2 Experimental Results

Figure 6.1 demonstrates the heterogeneous performance comparison of fDSM against Popcorn with TCP-IP and Infiniband RDMA messaging layers for the A class of benchmarks. We normalize the speedup of applications to the execution time of Popcorn with the TCP-IP messaging layer. For each application, the different colored bars on the x-axis represent the speedups for software DSM Popcorn with TCP-IP (always 1), software DSM Popcorn with Infiniband RDMA, and fDSM. The speedup formula is shown in Equation 6.1.

$$Speedup\ of\ fDSM\ in\ IS = \frac{Execution\ time\ of\ IS\ using\ Popcorn\ TCP - IP}{Execution\ time\ of\ IS\ using\ fDSM} \quad (6.1)$$

Along with significantly surpassing Popcorn with TCP-IP by ≈ 2.3X in IS, 2.82X in MG, 1.42X in FT, and 1.3X in CG, fDSM also outperforms Popcorn with RDMA by ≈ 7%, 6%, 7%, and 3% in benchmarks MG, CG, IS, and FT, respectively. In the rest of the applications, neither RDMA nor fDSM performs better, with the speedup being is close to zero. We observe a similar trend with the B class of benchmarks where fDSM outperforms RDMA and TCP-IP in 4 out of 10 applications. As seen in Figure 6.2, which demonstrates the speedups for the B class, fDSM shows a performance benefit of ≈ 2X, 1.45X, 1.12X, and 1.07X over TCP-IP, and gains of 5%, 3%, 1% and 0.5% over RDMA in IS, MG, FT, and CG, respectively, whereas, in the rest of the applications, there is virtually no speedup.

We ran into memory issues while compiling the MG and FT applications in the C class of benchmarks. The data set for the C class of these applications is enormous, and the program consists of statically allocated arrays requiring a large amount of stack space. This produces a linking and compilation (”relocation truncated to fit”) error on Linux in the case of FT. GCC uses a small code model, where all of the program’s code and statically allocated data has to fit into the lower 2GB of the address space. In this case, these static arrays consume the entire memory for themselves. As a result, the address mapping of other variables in the application exceeds the 2GB limit. To resolve this issue, programmers must refactor the application to avoid static arrays and dynamically allocate data. Whereas in the case of MG, the application compilation was successful. However, during execution, the portion of the application couldn’t reach these variables and raised exceptions like segmentation faults.

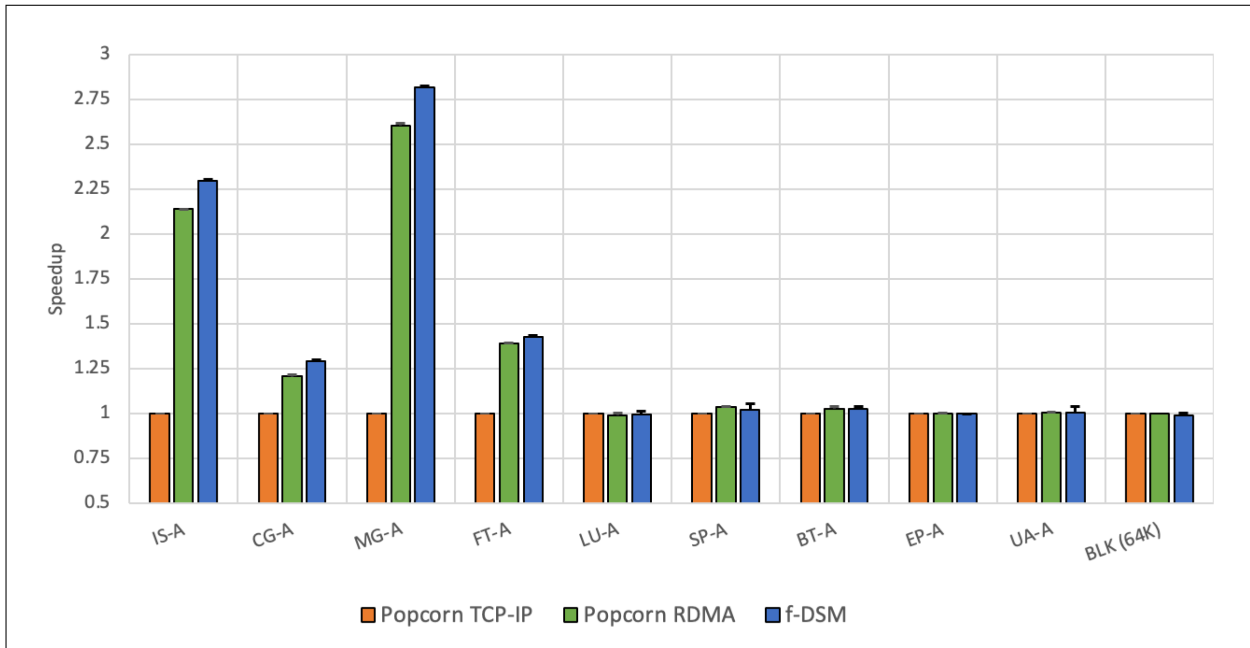


Figure 6.1: Speedup of fDSM over Popcorn with TCP-IP and Popcorn with RDMA in A class of NPB and PARSEC Benchmarks

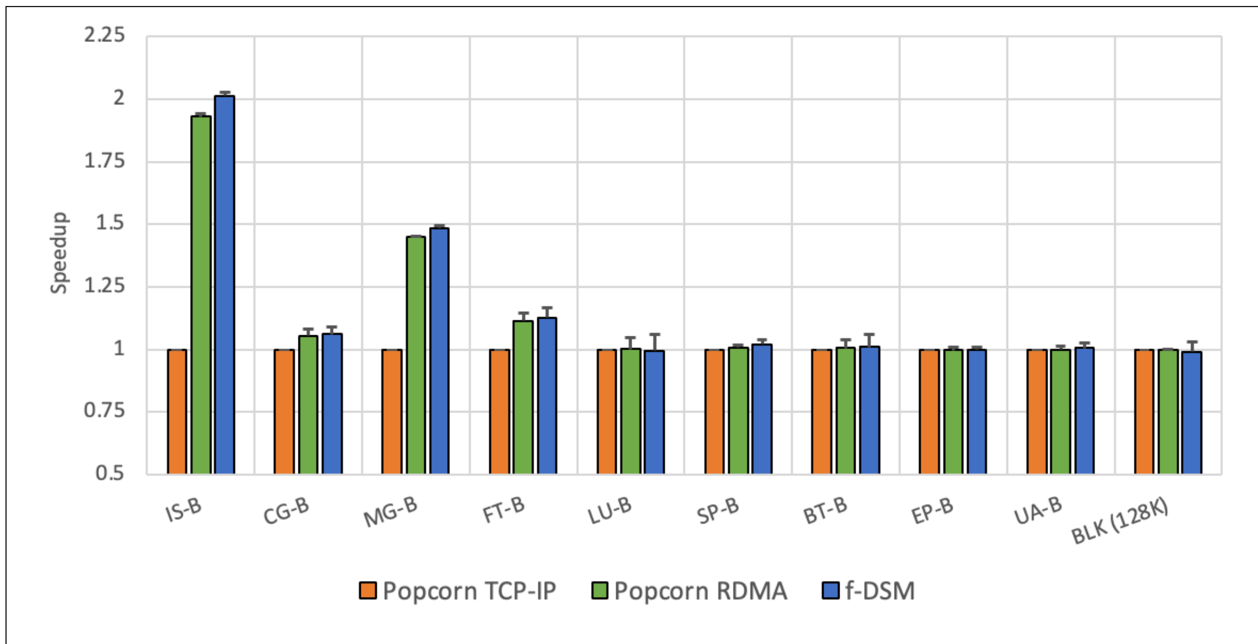


Figure 6.2: Speedup of fDSM over Popcorn with TCP-IP and Popcorn with RDMA in B class of NPB and PARSEC Benchmarks

Therefore, we exclude them from the C class plots and measure the speedups for the rest of the applications.

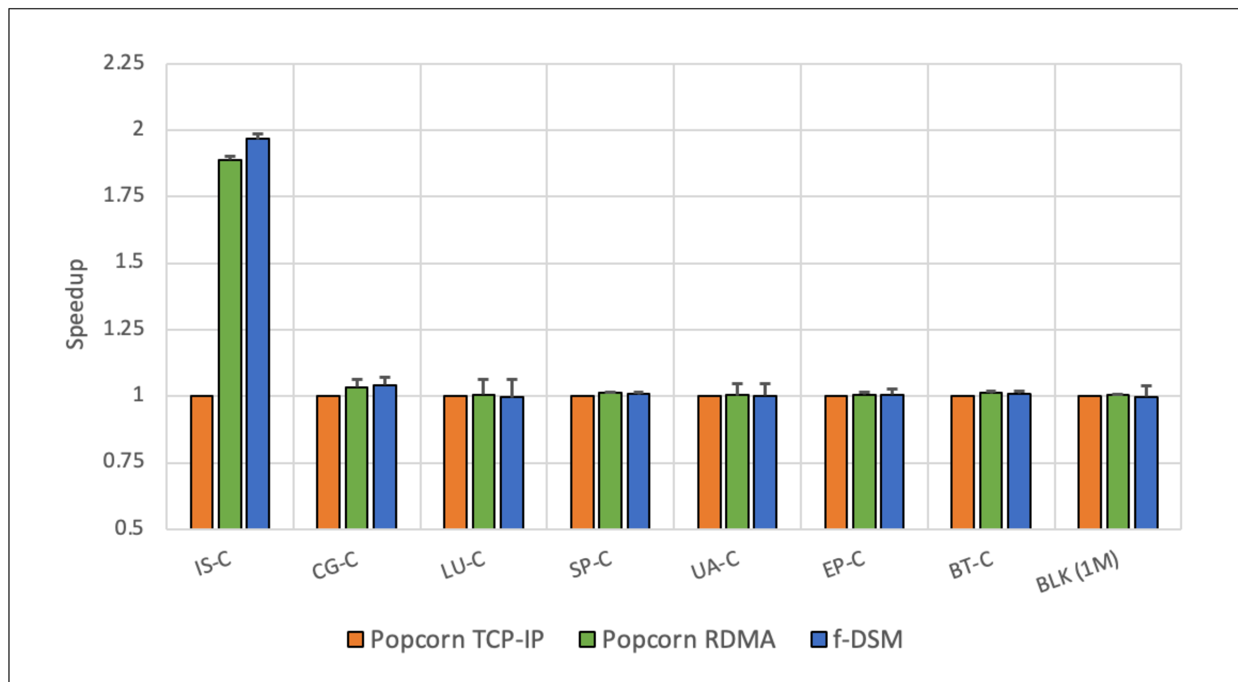


Figure 6.3: Speedup of fDSM over Popcorn with TCP-IP and Popcorn with RDMA in C class of NPB and PARSEC Benchmarks

Regardless of the above-mentioned problems, we see a similar tendency in the C class of benchmarks, as shown in Figure 6.3. fDSM outperforms Popcorn TCP-IP by 96% and 5%, and Popcorn RDMA by 5% and 1% in IS and CG, respectively.

6.2.1 Speedup Analysis

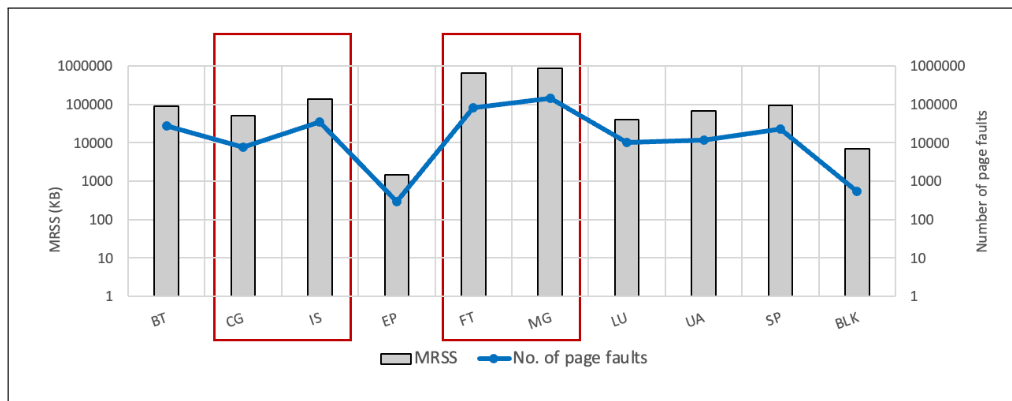
We attribute the speedup gains to the application's memory intensity and the number of page faults triggered during the execution, as shown in Figure 6.4a. The primary y-axis represents the MRSS (Maximum Resident Set Size) of the application, i.e., the application's RAM usage in KB. The secondary y-axis represents the number of page faults triggered during the benchmark execution. The benchmarks highlighted by red boxes are the ones

that outperform Popcorn with TCP-IP and RDMA. We measured the memory utilization using a Linux kernel API (`getrusage()`) which returns a structure consisting of all the resource usage and fault metrics of an application using its `task_struct`.

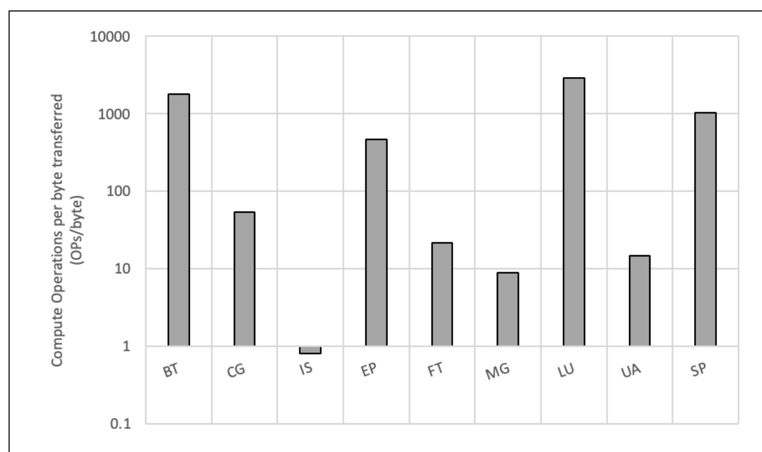
It is interesting to note that despite some applications (BT, LU, SP, and UA) being memory-intensive and raising many page faults, do not show any speedups. To understand this phenomenon better, we measure the computational density, also known as Arithmetic Intensity, of the benchmarks, as shown in Figure 6.4b. The y-axis represents the Compute operations performed by the host CPU per byte of data transferred. We determine this using the number of pages transferred between the nodes, execution time, and MOP/s (Million Operations per second) obtained from the application. Equation 6.2 shows the formula for calculating the Arithmetic Intensity of an application.

$$Arithmetic\ Intensity = \frac{Execution\ time \times ApplicationMOPs}{Number\ of\ pages\ transferred \times Page\ size} \tag{6.2}$$

As evidenced by the figure, BT, LU, and SP have high compute operations per byte compared to the other applications, and this is a clear indicator that these benchmarks are CPU-intensive. CPU-bound applications don't perform well with our implementation and Popcorn using RDMA since they fail to leverage the interconnect's bandwidth and speed. To better understand the underperformance of CPU-bound applications, we need to consider the advantage an interconnect has over the other. For instance, the time taken by Popcorn RDMA to resolve a page fault is 37 microseconds (as shown in Figure 6.7), while Popcorn TCP-IP spends 120 microseconds to do the same, indicating that RDMA has an edge of 83 microseconds over TCP-IP. This time delay is related to the driver implementation and the capabilities of the hardware interconnect. It is important to note that this difference in page resolution time is the most crucial aspect and solely determines the performance benefits of



(a) Memory utilized and number of page faults raised during benchmark execution



(b) Computational Density of each benchmark

Figure 6.4: Compute Intensity and Memory Intensity and number of page faults raised in the benchmarks

any messaging layer.

To further explain the relation between speedup and the number of page faults, let us consider the case of EP. The execution time of EP-A is 92 seconds with the Popcorn TCP-IP layer. Throughout its runtime, it utilizes 1444kB of RAM and raises 280-page faults. This is very low compared to the number of page faults triggered by IS (34580).

Note: The memory utilized and the number of page faults raised by any application are always constant and mutually exclusive of the underlying hardware.

Since RDMA has a benefit of 83 microseconds over TCP-IP, the speedup in execution time would be directly proportional to the product of the number of page faults and the interconnect time difference. Multiplying 83 microseconds with 280-page faults results in a time difference of 0.023 seconds, i.e., the execution time of EP-A using RDMA is close to 91.978 seconds (92 - 0.028). By using Equation 6.1, we can say Popcorn RDMA has a speedup of 0.02% over Popcorn TCP-IP, which is inconsequential. This implies that the number of page faults is a critical indicator of speedup, and EP-A spends a negligible fraction of its runtime utilizing the I/O (where the speedup lies). We can extrapolate the performance benefits for the other benchmarks using the same concept and conclude that processor-bound applications like BT, SP, LU, and EP do not obtain any performance benefits with our implementation.

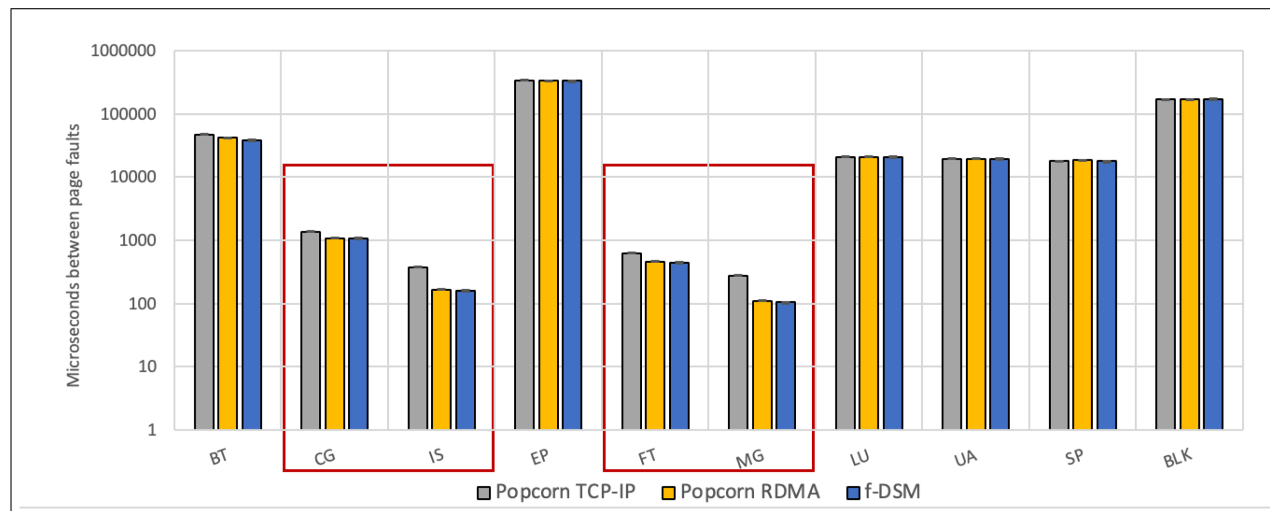


Figure 6.5: Page fault periods with respect to messaging layers

However, the UA application is an anomaly here. It has a comparatively low computational density, uses a significant RAM volume, and triggers a good amount of page faults but shows no speedup. We explain this outlier using the page fault period, i.e., the time elapsed between subsequent page faults, as shown in Figure 6.5. The y-axis represents the microseconds between consecutive page faults during the application runtime. The benchmarks highlighted

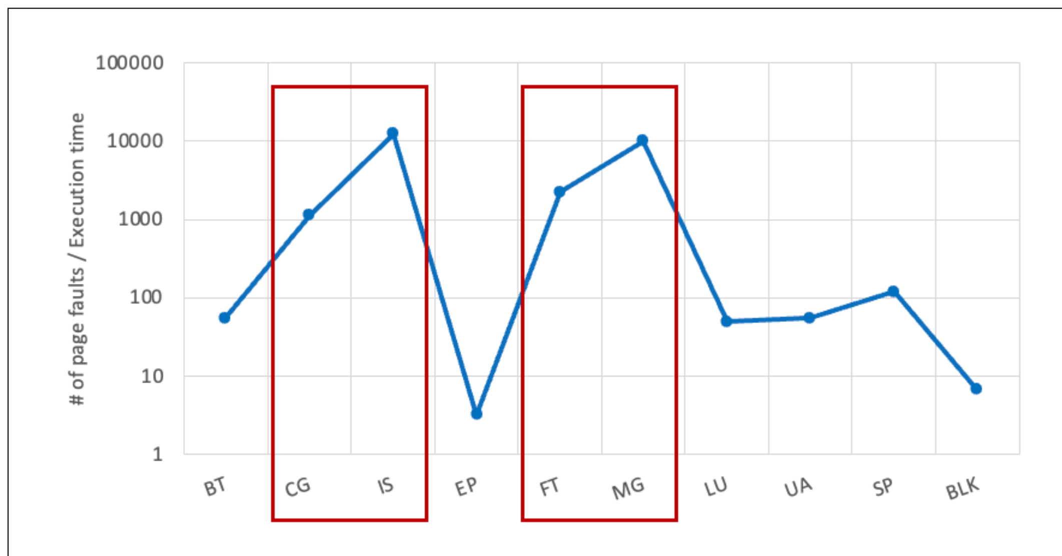


Figure 6.6: Ratio of number of page faults to execution time of benchmark

with red boxes are the ones that gain benefits over Popcorn RDMA. This is another essential metric, which is directly proportional to the execution time and inversely proportional to the speedup. For instance, UA-A has an execution time of 211.6 seconds and 210.49 seconds using Popcorn with TCP-IP and fDSM, respectively. fDSM takes 28 microseconds to resolve a page fault (as shown in Figure 6.8), indicating it is 92 microseconds faster than TCP-IP and 9 microseconds faster than RDMA. The total number of page faults for UA-A is 11576, i.e., the execution time difference is 11576 times 92 microseconds, which is 1.06 seconds. By using Equation 6.1, we observe that the speedup of fDSM over Popcorn TCP-IP is 0.5% for UA-A, which is inconsequential. In this case, since the execution time of an application is high, the page fault period is also high leading to minimal performance benefits.

In addition, as evidenced by Figure 6.6, which depicts the page fault to execution time ratio in the y-axis and the applications in x-axis, it is critical that the page faults raised by an application is correlated with its execution time. We can support this argument by revisiting the UA-A case. UA-A runs for 211.6 seconds with only $\approx$ 11K page faults, whereas IS-A runs for 2.8 seconds with $\approx$ 36K page faults. This is an indicator that unlike IS-A, UA-A spends

most of its execution time looping in the CPU rather than accessing the I/O or memory, and conclude that speedup is also proportional to the page faults raised per second of execution time.

6.2.2 Timing Breakdown

Figures 6.7 and 6.8 illustrate the breakdown of the time taken to resolve a page fault in Popcorn with RDMA and fDSM, respectively. As mentioned earlier, Popcorn with RDMA takes 37 microseconds to fix a page fault, irrespective of the node or application. It has 14.4 microseconds of software latency (with 4 microseconds to run the software DSM steps) and 22.6 microseconds of hardware latency. Whereas, fDSM spends 28 microseconds to resolve a page fault. It has 13 microseconds of software latency and 15 microseconds of hardware latency. We use Linux kernel time accessors like `ktime_get_ns()` to get the breakdown of these numbers. In FPGAs, it is tedious to measure the time taken by each IP in the hardware logic. Therefore, we determine the average time to resolve a page fault using the `ktime_get_ns()` function and subtract the software latencies to obtain the hardware timing.

Note: The question marks in Figure 6.8 indicate that the time taken by that process isn't measured. For instance, time taken by the FIFO logic to raise an interrupt is unknown.

fDSM avoids the latency of $\approx$ 4 microseconds involved in handling software DSM actions by delegating that work to the DSM processor. The DSM processor only takes 15 clock cycles, i.e., $\approx$ 50 nanoseconds to process the request (DSM processor runs at 312.5 MHz). Instead of sending a message that takes 0.3 microseconds, it writes to the registers (costs $\approx$ 0.2 microseconds). Although it avoids software DSM latencies, it incurs a delay of 2.6 microseconds due to interrupt servicing and misfiring latencies. Also, the XDMA IP often misfires interrupts upon completion of C2H transfers. This is a minor defect in the XDMA

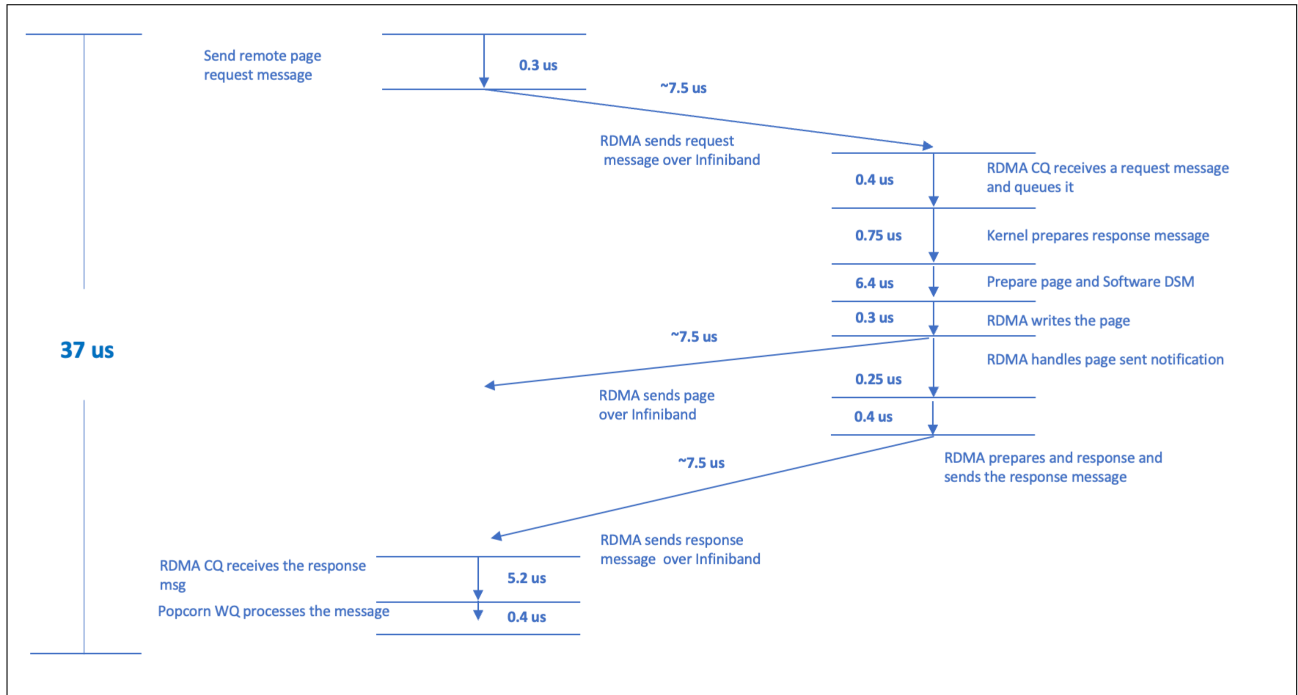


Figure 6.7: Timing breakdown of the resolution of a page fault using Popcorn with RDMA

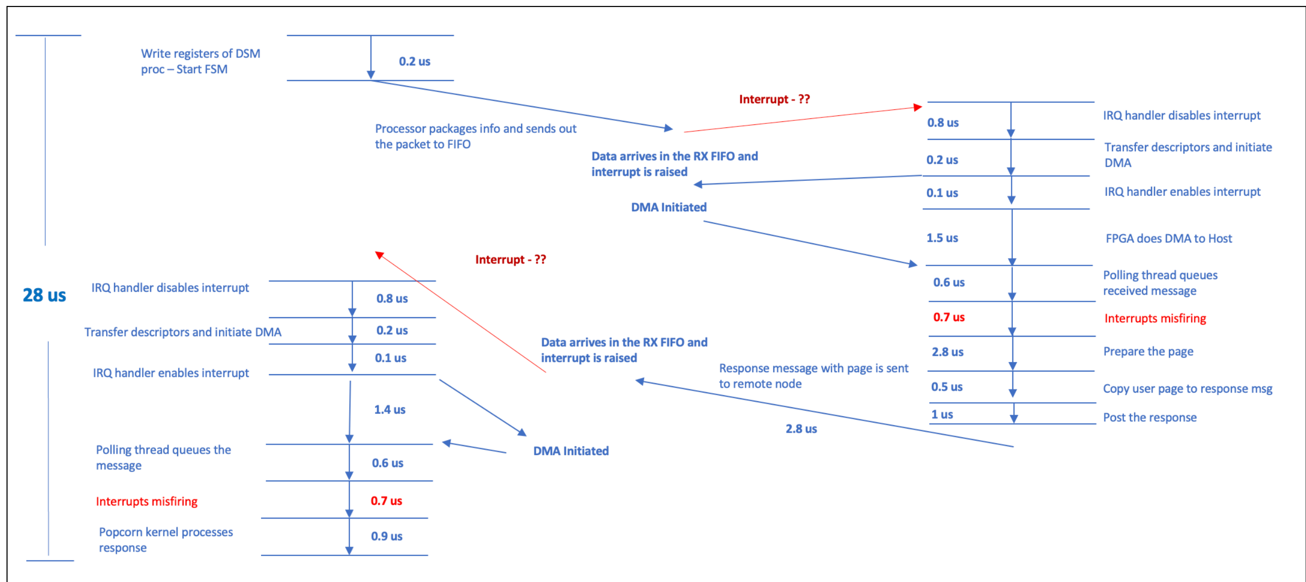


Figure 6.8: Timing breakdown of the resolution of a page fault using fDSM

IP. These misfires constitute 30-40% of the interrupt latencies and add moderate delays to the over execution time. Therefore, the net speedup in software is ≈ 1.4 microseconds (4 - 2.6). Additionally, fDSM hardware logic is 7 microseconds faster than RDMA hardware resulting in an overall benefit of ≈ 8.4 microseconds (7 + 1.4).

The savings of 8.4 microseconds of fDSM over RDMA explains why the former's speedups are close to the latter. We can use these savings to document another calculation of a theoretical speedup. For instance, IS-A runs for ≈3 seconds using Popcorn with RDMA and has ≈36K page faults. As a result, the savings in execution time is 0.3 seconds (8.6 * 36000), and the theoretical execution time of IS-A using fDSM is 2.7 seconds. Using equation 6.1, we can say that fDSM has a theoretical speedup of 10% over Popcorn with RDMA (3/2.7), which is directly proportional to the savings (actual speedup is ≈7%). To further increase the performance of fDSM, the 8.6 microseconds gap has to widen, meaning the time taken to resolve a page fault by fDSM has to decrease. We can do so by further avoiding host and interrupt latencies. f-DSM can avoid these delays by completely removing host intervention and disabling the XDMA interrupt vector. This is one of the optimizations that we plan to do in the future.

6.2.3 Homogeneous Setup Evaluation

In our homogeneous evaluation, we observe the same trend in results compared to the heterogeneous setup. In Figures 6.9, 6.10, 6.11, we demonstrate the homogeneous results of fDSM compared against Popcorn with TCP-IP and RDMA with A, B, and C classes of benchmarks.

In the A-class, we notice speedups of 2.7X, 1.7X, 2.47X, and 1.56X over Popcorn TCP-IP and 5%, 4%, 3%, and 2% over Popcorn with RDMA, in IS, FT, MG, and CG, respectively,

and no visible speedups in the other benchmarks.

Likewise, in the B-class, we obtain speedups of 2.5X, 1.44X, 1.28X, and 1.14X over Popcorn TCP-IP and 4%, 1%, 1% over Popcorn with RDMA, in IS, MG, and FT, respectively. In the C class, we observe speedups of 1.76X, 1.57X over Popcorn TCP-IP, and 5%, 1% over Popcorn RDMA in IS and CG, respectively.

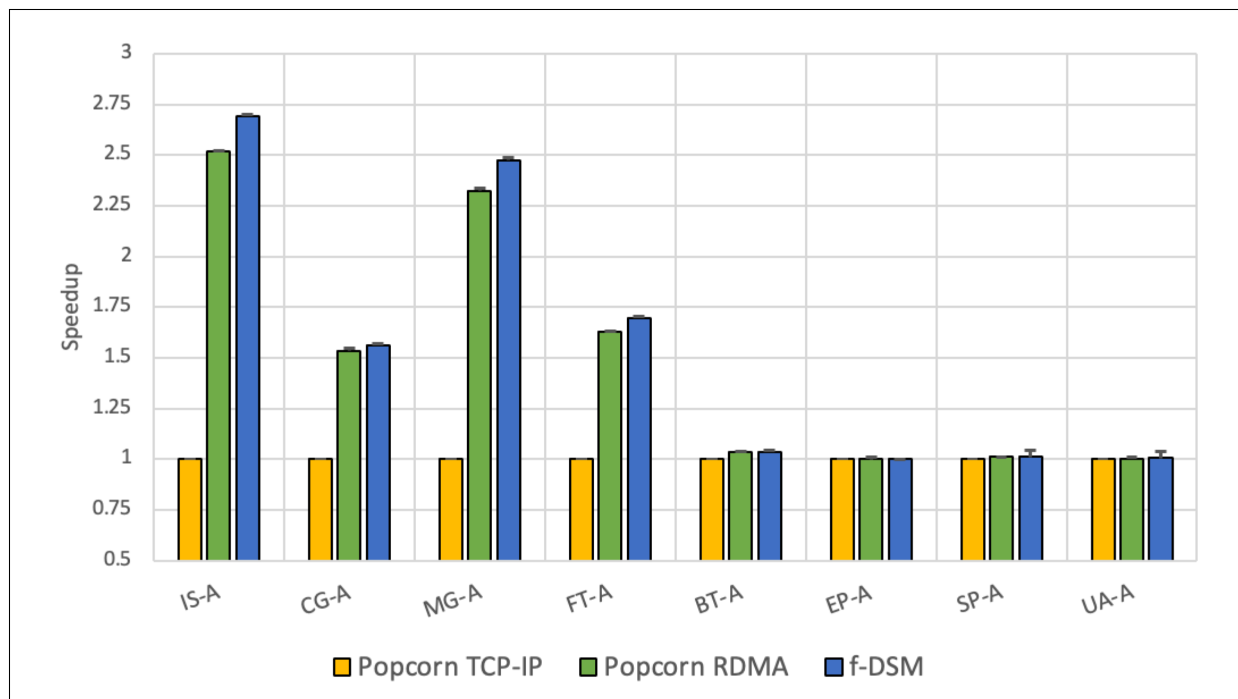


Figure 6.9: Speedup of fDSM over Popcorn with TCP-IP and Popcorn with RDMA in B class of NPB and PARSEC Benchmarks - Homogeneous Setup

6.3 Discussion

We observed a disparity in the speedups of different classes of the benchmarks, despite the high number of page faults, as shown in Figure 6.12a. The primary y-axis represents the speedup, the second y-axis represents the number of page faults, and the x-axis represents the A and B classes of benchmarks. For instance, MG shows a speedup of 2.8X over Popcorn

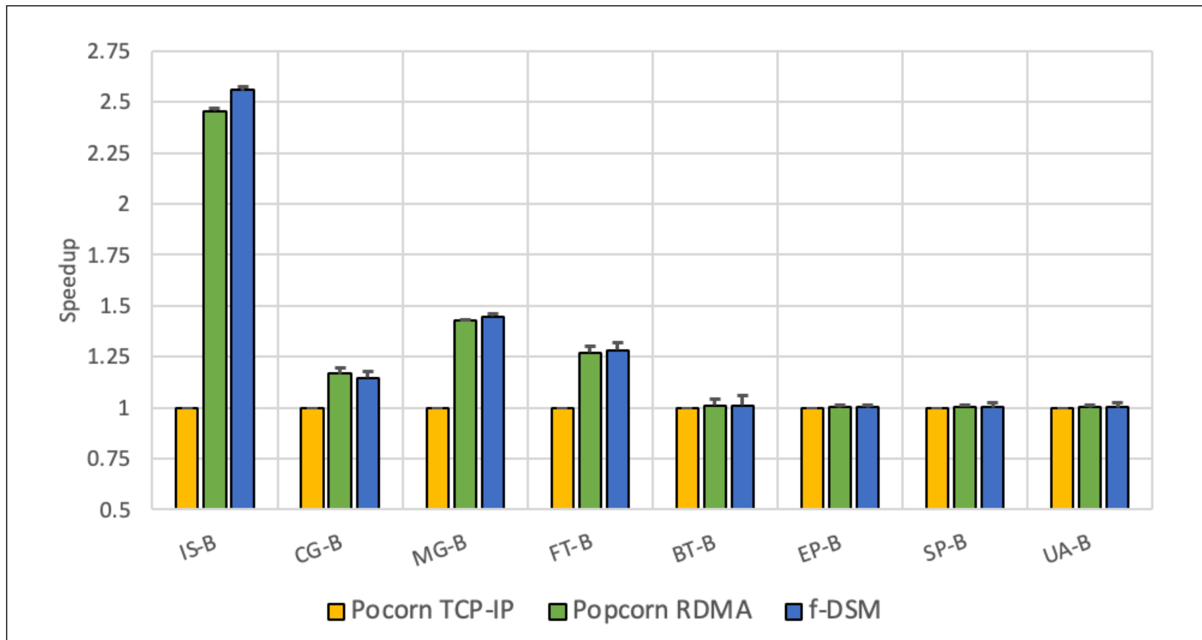


Figure 6.10: Speedup of fDSM over Popcorn with TCP-IP and Popcorn with RDMA in B class of NPB and PARSEC Benchmarks - Homogeneous Setup

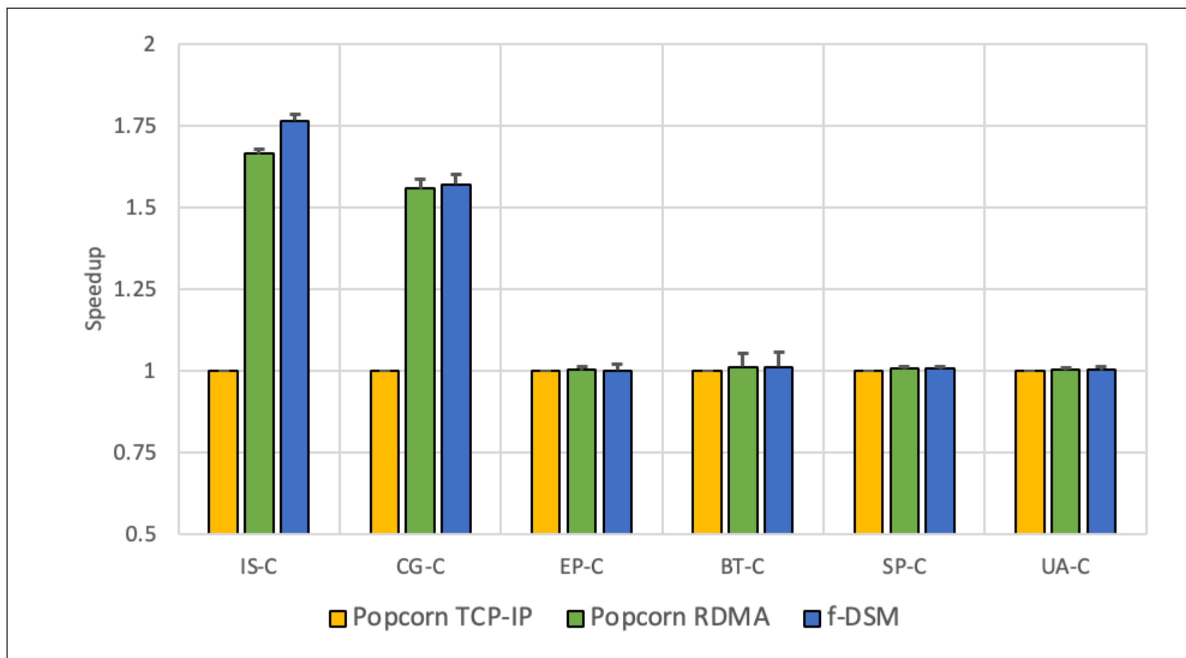


Figure 6.11: Speedup of fDSM over Popcorn with TCP-IP and Popcorn with RDMA in B class of NPB and PARSEC Benchmarks - Homogeneous Setup

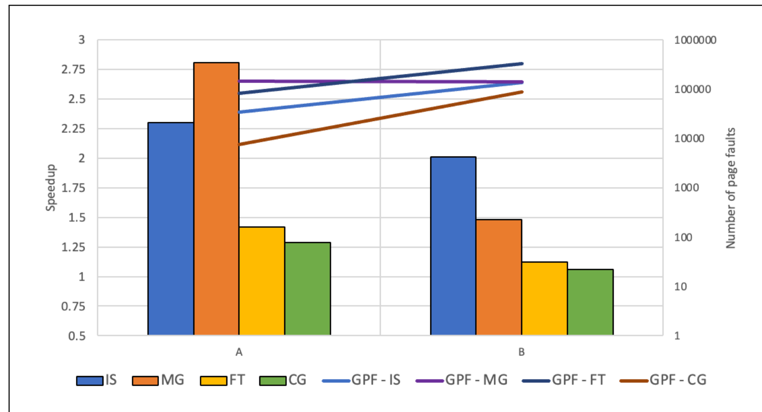
TCP-IP in the A-class but steeply declines to 1.48X in the B-class. Upon investigation, we found that the benchmarks (MG, FT, and CG) which show inconsistency in speedup tend to have higher computational densities as the data size increases, as shown in Figure 6.12b. The primary y-axis represents the computational density, the second y-axis represents the page fault period, and the x-axis represents the A and B classes of benchmarks. MG-A has a computational density of 8.05 Compute operations per byte, whereas MG-B has 41.4 operations per byte. This incline causes the page fault period to increase and speed up to drop. Only the speedup of IS remains consistent irrespective of the class due to its high memory access and low computational density.

Figure 6.13a shows the standard deviation of the execution time of the benchmarks in the heterogeneous setup. It is evident that memory-intensive applications are very stable and have a low variance compared to CPU-bound applications, which are more dispersed.

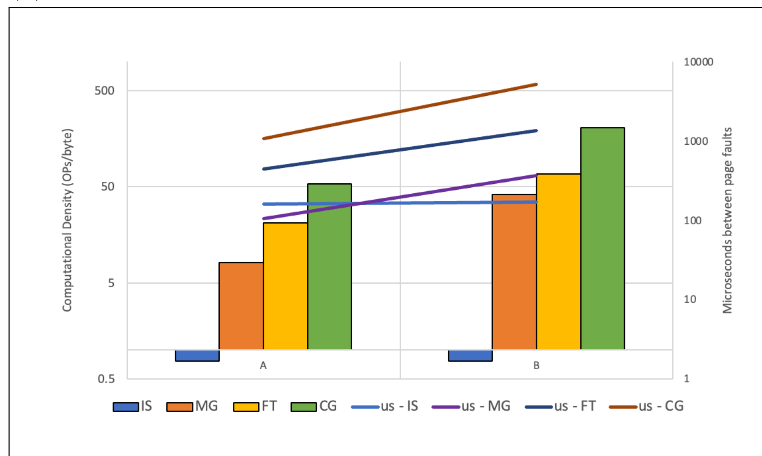
We also observed a considerable difference between the application runtime in the heterogeneous and homogeneous setup. Figure 6.13b depicts this execution time difference for only the memory-intensive applications (to give an idea). The time jump in the heterogeneous setup is due to the stack transformation latency during the initialization of the task migration. The popcorn kernel does the stack transformation because of the non-uniformity of the stack layout in different ISAs.

6.3.1 Ideal Application Behavior

As discussed earlier, the speedup is directly proportional to the number of page faults and I/O memory usage. It is also inversely proportional to the computational density of the benchmarks, and the page fault period, as shown in Equation 6.3.

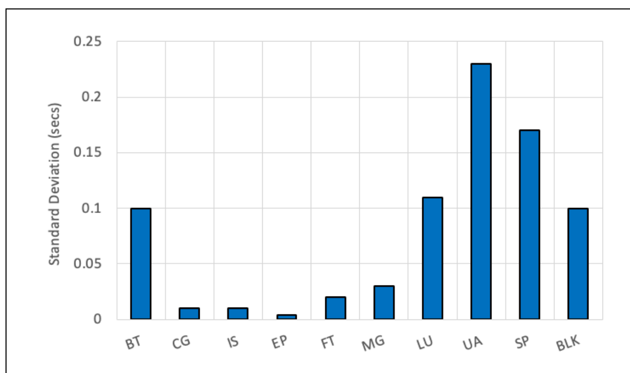


(a) Declining speedup over classes and number of page faults

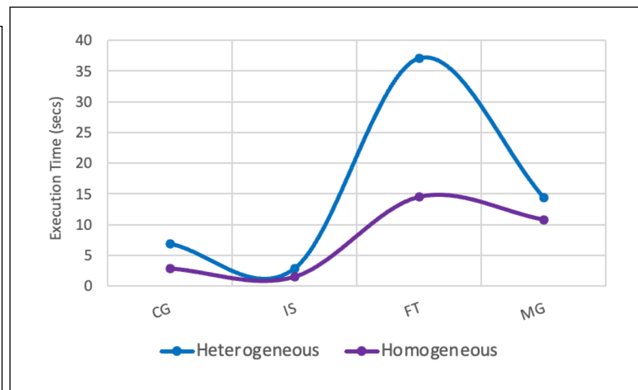


(b) Increasing computational density and page fault period

Figure 6.12: Reason for decreasing speedup over bigger classes of benchmarks



(a) Execution Standard Deviation



(b) Execution time difference between setups

Figure 6.13: Execution time observations in both heterogeneous and homogeneous setups

FPGA Resource Utilization			
Resource	Utilization	Available Resources	Utilization Percentage
LUT (Lookup Tables)	56013	871680	6.43
LUTRAM (Lookup Table RAM)	3985	403200	1
FF (Flip Flops)	63726	1743360	3.66
BRAM (Block RAM)	97.50	1344	7.25
IO (Input/Output)	2	416	0.48
GT (Gigabit Trans-receivers)	20	20	100
BUFG	28	672	4.17
MMCM (Multi-Mode Clock Module)	1	8	12.5

Table 6.1: FPGA Resource Utilization

$$Speedup \propto \frac{Pagefaults \times Memory\ usage}{Computational\ Density \times Page\ fault\ period} \tag{6.3}$$

Therefore, we can conclude that an ideal application for fDSM to show substantial performance benefits would be similar to IS and MG-A. It should be a memory-bound application with the combination of low computational density like IS, high consumption of RAM, and a minimal page fault period like MG-A.

6.3.2 FPGA resource utilization

Table 6.1 shows the FPGA resource utilization. Since we use the 100GB Ethernet Subsystem and there are 4 x 25G GT channels in the FPGA we use, it consumes all the Gigabit Transceivers (GT) in the FPGA. Apart from GT, our design takes up minimal resources, and there is sufficient space to implement additional logic. Also, since our design is compact, it can be incorporated in state-of-the-art FPGA-based SmartNICs.

6.4 Evaluation Summary

In this chapter, we presented the experimental results of fDSM in the heterogeneous setup and compared its performance against Popcorn, a software DSM infrastructure with TCP-IP and Infiniband RDMA based messaging layers. Upon evaluation with NPB and PARSEC benchmarks, fDSM performs up to 2.8X and 7% better than Popcorn with TCP-IP and RDMA, respectively, in some memory-bound applications. We found out using the Computational Density that compute-intensive applications spend most of their execution time in the CPU rather than utilizing I/O. This increases the page fault period and computing operations per byte of data transfer of the benchmarks.

Following that, we break down the latency to resolve a page fault and compare it against Popcorn using Infiniband RDMA and highlight the areas where fDSM outperforms the state-of-the-art interconnect (Mellanox Infiniband RDMA adapters), and explains how fDSM avoids software overhead. We also examine the areas of concern and avoidable latencies and discuss further improvements.

We also present the homogeneous setup results to solidify our evaluation. It confirms that fDSM can provide FPGA support DSM for a homogeneous setup.

Furthermore, we discuss the declining speedup of certain benchmarks (MG, CG, and FT) throughout the data classes and attribute it to the increasing computational density and the page fault period as the data size increases. Based on these observations, we explain the ideal behavior of an application that can benefit from employing the fDSM framework. We conclude with some other metrics like Standard Deviation and FPGA resource utilization, proving that fDSM is stable and has a compact design that utilizes minimal resources leaving space to implement additional logic, possibly other DSM consistency protocols.

Chapter 7

Conclusions and Future Work

Distributed Shared Memory (DSM) is a high-performance computing architecture that combines the advantages of the shared-memory, provides an illusion of "shared" memory, and the low cost and scalability of the distributed-memory paradigms [52]. DSM has been a thriving area of research since the early 1980s and has seen exponential progress through numerous works, all of which target low latency shared memory abstraction provided to multiprocessors or physically distributed nodes. Software DSM systems incur the overhead of managing consistency and coherency in the OS [9, 45, 54]. Hardware-supported DSM frameworks rely on custom chips [1, 49, 51] and cache-coherent interconnects [18, 23, 88] that lack the reconfigurability to experiment with different consistency models and DSM algorithms, and the scalability to provide page-level memory coherency between physically distributed systems.

To address these issues, this thesis presents **fDSM**, an FPGA-accelerated DSM framework for heterogeneous compute architectures. fDSM employs a high-speed FPGA messaging layer to enable inter-kernel communication and a DSM protocol processor that implements a sequential consistency model to provide a shared virtual memory between physically distributed heterogeneous/homogeneous nodes.

fDSM utilizes Xilinx Alveo data center accelerator FPGAs and a QSFP link to implement the low-latency messaging layer. These FPGAs connect to the PCIe slots of the nodes. fDSM also uses the commodity IPs provided by Xilinx, like the PCIe DMA subsystem,

Ultrascale+ Ethernet subsystem, and FIFO generators, along with our custom RTL modules to enable the transfer of page and message data between the nodes. We build the OS support for the messaging layer on top of DeX [46], and we implement all the messaging passing functionalities in a plug-in FPGA driver. Any software DSM framework can use fDSM's messaging layer by leveraging the APIs provided by the driver and can seamlessly migrate tasks and pages between clusters with minimal OS modifications.

Furthermore, fDSM provides an FPGA-supported shared virtual memory between the connected nodes, the core contribution of this thesis. We design a DSM protocol processor comprising an FSM core, responsible for maintaining sequential consistency at a 4KB page-level granularity. The processor implements a multiple-reader single-writer coherency protocol, and is also equipped with a 32-bit register space and memory-mapped interface for host interaction. It also has pipelined registers, data forwarding units, and AXI-stream input/output modules to transmit, receive and process page requests and packets. fDSM utilizes this DSM processor in conjunction with the messaging layer to manage the page status, track the page owners, and manage the memory consistency and coherency as a whole.

Although fDSM targets to provide DSM between a host CPU and an FPGA-based accelerator (for example, SmartNICs), we design a scaled-up prototype using two server-grade heterogeneous machines connected through Xilinx FPGAs and 100 GB QSFP cables. Since we adopt an atypical approach of providing virtual memory coherency at a 4KB page granularity, there aren't any concrete open source comparisons to our work.

Therefore, we compare fDSM against Popcorn, a heterogeneous software DSM framework utilizing 10 Gigabit-ethernet (TCP-IP) and state-of-the-art 40 GBps Infiniband RDMA interconnects. We experiment with NAS parallel benchmarks and PARSEC benchmarks that mimic computational fluid dynamics applications and mathematical problems, respectively. Experimental studies reveal that, fDSM performs 2.8X and 7% better than Popcorn with

TCP-IP and Popcorn with Infiniband RDMA, respectively, in memory-intensive applications. However, in CPU-bound applications, fDSM shows marginal speedups. It also provides reconfigurability to experiment with other consistency models and DSM algorithms with some hardware logic changes.

7.1 Limitations

Currently, fDSM doesn't offer concurrent application support. All the benchmarks we run are single-threaded, and a significant amount of engineering in the hardware logic is required to sustain multi-threaded page faults and requests. For instance, we tried the KMeans application from the Rodinia benchmark suite. KMeans spawns multiple threads, thereby raising simultaneous page faults which access the DSM processor at the same instant. Although we implement locking mechanisms in the software, there are no synchronization primitives available in the hardware that can delay a page fault until the previous one is processed. Similarly, when 2-page requests arrive at the receiving end, the FSM is unable to process both and ends up servicing the latest message. Therefore, the DSM processor requires a send and a receive buffer to queue the page faulting handling requests and incoming messages, respectively. With the help of these buffers, the DSM processor can avoid race conditions and other synchronization issues.

Unlike, Infiniband RDMA adapters, our current version of fDSM requires software/host intervention to initiate the DMA between the host and the device memory. This adds significant latencies while receiving a page from the remote node, resulting in fDSM performing slightly better (7%) than Popcorn with RDMA. These latencies can be avoided by the complete removal of host intervention using custom hardware logic to initiate the RX DMA. This optimization can further augment the performance of fDSM and widen the small gap

between the speedups of fDSM and Popcorn with RDMA.

7.2 Future Work

As discussed earlier, fDSM intends to provide DSM between a host CPU and the CPU of an FPGA-based accelerator like a SmartNIC with an ARM CPU. This would require us to implement fDSM's hardware logic in the FPGA of the SmartNIC and run a Linux kernel in the SmartNIC's CPU to provide OS support. Work is in progress to do the same on a Bitware 250 SoC [14], which has a 64-bit ARM CPU and x16 PCIe interconnects. A 100 GB ethernet link and the Ethernet subsystem IP would be unnecessary since the two nodes are connected through the PCIe slot. The FPGA driver has to be changed to accommodate this setup.

Furthermore, we can implement multiple consistency models like Lazy-Release, Aggressive-Release, and other strict or weak consistency models [29, 43] in the DSM processor. Having multiple consistency models would give an end-user the privilege of choosing the protocol based on the application requirement. or example, applications like Cholesky decomposition and FFT benefit from eager release consistency models, and Blocked-LU and Barnes-hut greatly benefit from lazy release consistency models due to the high false-page sharing [48]. This would require additional engineering to implement a flexible DSM processor, which can be controlled by the end-user to switch between different consistency protocols.

Bibliography

[1] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, Beng-Hong Lim, K. Mackenzie, and D. Yeung. The mit alewife machine: architecture and performance. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 2–13, 1995. doi: 10.1109/ISCA.1995.524544.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: A simple abstraction for remote memory. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 775–787, USA, 2018. USENIX Association. ISBN 9781931971447.

[3] Mohammad I. AlAli, Khaldoon M. Mhaidat, and Inad A. Aljarrah. Implementing image processing algorithms in fpga hardware. In *2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, pages 1–5, 2013. doi: 10.1109/AEECT.2013.6716446.

[4] Alpha Data. ADM-PCIE-9V3 User Manual. https://www.alpha-data.com/product/adm-pcie-9v3/.

[5] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. Lazy release consistency for gpus. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, 2016. doi: 10.1109/MICRO.2016.7783729.

[6] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, Honghui Lu, R. Rajamony, Weimin

Yu, and W. Zwaenepoel. Treadmarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996. doi: 10.1109/2.485843.

[7] ARM, 2020. AMBA AXI and ACE Protocol Specification Version H.c. https://developer.arm.com/documentation/ihi0022/latest/.

[8] M. Banikazemi, J. Liu, D.K. Panda, and P. Sadayappan. Implementing treadmarks over virtual interface architecture on myrinet and gigabit ethernet: Challenges, design experience, and performance evaluation. In *International Conference on Parallel Processing, 2001.*, pages 167–174, 2001. doi: 10.1109/ICPP.2001.952060.

[9] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: a replicated-kernel os based on linux. pages 123–138, July 2014. Linux Symposium 2014, OLS 2014 ; Conference date: 14-07-2014 Through 16-07-2014.

[10] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332385. doi: 10.1145/2741948.2741962. URL https://doi.org/10.1145/2741948.2741962.

[11] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-isa datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 645–659, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344654. doi: 10.1145/3037697.3037738. URL https://doi.org/10.1145/3037697.3037738.

[12] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. *SIGPLAN Not.*, 25(3):168–176, feb 1990. ISSN 0362-1340. doi: 10.1145/99164.99182. URL https://doi.org/10.1145/99164.99182.

[13] Saman Biookaghazadeh, Pravin Kumar Ravi, and Ming Zhao. Toward multi-fpga acceleration of the neural networks. *J. Emerg. Technol. Comput. Syst.*, 17(2), apr 2021. ISSN 1550-4832. doi: 10.1145/3432816. URL https://doi.org/10.1145/3432816.

[14] Bittware. BittWare 250-SoC Accelerator Card Data Sheet. https://www.bittware.com/files/ds-250-soc.pdf.

[15] D.L. Black, A. Gupta, and W.-D. Weber. Competitive management of distributed shared memory. In *Digest of Papers. COMPCON Spring 89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage*, pages 184–190, 1989. doi: 10.1109/CMPCON.1989.301925.

[16] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/brunella.

[17] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with rdma and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, jul 2018.

ISSN 2150-8097. doi: 10.14778/3236187.3236209. URL https://doi.org/10.14778/3236187.3236209.

[18] CCIX Consortium. An Introduction to CCIX. https://www.ccixconsortium.com/wp-content/uploads/2019/11/CCIX-White-Paper-Rev111219.pdf.

[19] Benny Cheung, Cho-Li Wang, and Kai Hwang. Jump-dp: A software dsm system with low-latency communication support. 07 2000.

[20] Ho-Ren Chuang, Robert Lyerly, Stefan Lankes, and Binoy Ravindran. Scaling shared memory multiprocessing applications in non-cache-coherent domains. In *Proceedings of the 13th ACM International Systems and Storage Conference*, SYSTOR '20, page 13–24, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375887. doi: 10.1145/3383669.3398278. URL https://doi.org/10.1145/3383669.3398278.

[21] A.L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the performance of software distributed shared memory as a target for parallelizing compilers. In *Proceedings 11th International Parallel Processing Symposium*, pages 474–482, 1997. doi: 10.1109/IPPS.1997.580943.

[22] A.L. Cox, E. de Lara, C. Hu, and W. Zwaenepoel. A performance comparison of homeless and home-based lazy release consistency protocols in software shared memory. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 279–283, 1999. doi: 10.1109/HPCA.1999.744380.

[23] CXL Consortium. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/about-cxl.

[24] David S. Miller, Richard Henderson, Jakub Jelinek. Dynamic DMA mapping Guide -

The Linux Kernel's User and Administrator Guide. https://www.kernel.org/doc/html/latest/admin-guide/index.html.

[25] G. Delp, A. Sethi, and D. Farber. An analysis of memnet—an experiment in high-speed shared-memory local networking. *SIGCOMM Comput. Commun. Rev.*, 18(4): 165–174, aug 1988. ISSN 0146-4833. doi: 10.1145/52325.52342. URL https://doi.org/10.1145/52325.52342.

[26] H. Eichner, C. Trinitis, and T. Klug. Implementation of a dsm-system on top of infiniband. In *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 6 pp.–, 2006. doi: 10.1109/PDP.2006.43.

[27] Wataru Endo, Shigeyuki Sato, and Kenjiro Taura. Menps: A decentralized distributed shared memory exploiting rdma. In *2020 IEEE/ACM Fourth Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*, pages 9–16, 2020. doi: 10.1109/IPDRM51949.2020.00006.

[28] S.S. Fu and Nian-Feng Tzeng. Lock improvement technique for release consistency in distributed shared memory systems. In *Proceedings of 6th Symposium on the Frontiers of Massively Parallel Computation (Frontiers '96)*, pages 255–262, 1996. doi: 10.1109/FMPC.1996.558093.

[29] S.S. Fu and Nian-Feng Tzeng. Aggressive release consistency for software distributed shared memory. In *Proceedings of 17th International Conference on Distributed Computing Systems*, pages 288–295, 1997. doi: 10.1109/ICDCS.1997.598054.

[30] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system, 2022.

[31] M. Heinrich, V. Soundararajan, J. Hennessy, and A. Gupta. A quantitative analysis of

the performance and scalability of distributed shared memory cache coherence protocols. *IEEE Transactions on Computers*, 48(2):205–217, 1999. doi: 10.1109/12.752662.

[32] J. Hennessy, M. Heinrich, and A. Gupta. Cache-coherent distributed shared memory: perspectives on its development and future challenges. *Proceedings of the IEEE*, 87(3): 418–429, 1999. doi: 10.1109/5.747863.

[33] Yang Hong, Yang Zheng, Fan Yang, Binyu Zang, Haibing Guan, and Haibo Chen. Scaling out numa-aware applications with rdma-based distributed shared memory. *Journal of Computer Science and Technology*, 34:94–112, 2019.

[34] Edson Horta, Ho-Ren Chuang, Naarayanan Rao VSathish, Cesar Philippidis, Antonio Barbalace, Pierre Olivier, and Binoy Ravindran. Xar-trek: Run-time execution migration among fpgas and heterogeneous-isa cpus. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 104–118, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385343. doi: 10.1145/3464298.3493388. URL https://doi.org/10.1145/3464298.3493388.

[35] InAccel, 2020. InAccel Accelerated Machine Learning Suite on the Nimbix Cloud with Xilinx Alveo. https://www.nimbix.net/accelerated-machine-learning-cloud-inaccel.

[36] Intel Princeton University. Princeton Application Repository for Shared-Memory Computers (PARSEC) . https://parsec.cs.princeton.edu/index.html.

[37] James K Archibald. The cache coherence problem in shared-memory multiprocessors. https://digital.lib.washington.edu/researchworks/handle/1773/6955.

[38] John Russell, 2020. Nvidia (Mellanox) Debuts NDR 400 Gigabit InfiniBand

at SC20. https://www.hpcwire.com/2020/11/16/nvidia-mellanox-debuts-ndr-400-gigabit-infiniband-at-sc20/.

[39] Greg Kroah-Hartman Jonathan Corbet, Alessandro Rubini. *Chapter 15. Memory Mapping and DMA - Linux Device Drivers, 3rd Edition*. O'Reilly Media Inc., 2005.

[40] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with Two-Sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia.

[41] David Katz, Antonio Barbalace, Saif Ansary, Akshay Ravichandran, and Binoy Ravindran. Thread migration in a replicated-kernel os. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 278–287, 2015. doi: 10.1109/ICDCS.2015.36.

[42] Stefanos Kaxiras, David Klaftenegger, Magnus Norgren, Alberto Ros, and Konstantinos Sagonas. Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 3–14, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335508. doi: 10.1145/2749246.2749250. URL https://doi.org/10.1145/2749246.2749250.

[43] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, pages 13–21, 1992. doi: 10.1109/ISCA.1992.753300.

[44] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, page 13–21, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897915097. doi: 10.1145/139669.139676. URL https://doi.org/10.1145/139669.139676.

[45] Peter Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Tread marks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter 1994 Technical Conference (USENIX Winter 1994 Technical Conference)*, San Francisco, CA, January 1994. USENIX Association. URL https://www.usenix.org/conference/usenix-winter-1994-technical-conference/tread-marks-distributed-shared-memory-standard.

[46] Sang-Hoon Kim, Ho-Ren Chuang, Robert Lyerly, Pierre Olivier, Changwoo Min, and Binoy Ravindran. Dex: Scaling applications beyond machine boundaries. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 864–876, 2020. doi: 10.1109/ICDCS47774.2020.00021.

[47] A. K. Kodi and A. Louri. Design of a high-speed optical interconnect for scalable shared memory multiprocessors. In *Proceedings of the High Performance Interconnects, 2004. on Proceedings. 12th Annual IEEE Symposium*, HOTI '04, page 92–97, USA, 2004. IEEE Computer Society. ISBN 0780386868. doi: 10.1109/CONECT.2004.1375210. URL https://doi.org/10.1109/CONECT.2004.1375210.

[48] Leonidas I. Kontothanassis, Michael L. Scott, and Ricardo Bianchini. Lazy release consistency for hardware-coherent multiprocessors. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, page 61–es, New

York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897918169. doi: 10.1145/224170.224398. URL https://doi.org/10.1145/224170.224398.

[49] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. *SIGARCH Comput. Archit. News*, 22(2):302–313, apr 1994. ISSN 0163-5964. doi: 10.1145/192007.192056. URL https://doi.org/10.1145/192007.192056.

[50] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979. doi: 10.1109/TC.1979.1675439.

[51] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The dash prototype: Implementation and performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, page 92–103, New York, NY, USA, 1992. Association for Computing Machinery. ISBN 0897915097. doi: 10.1145/139669.139706. URL https://doi.org/10.1145/139669.139706.

[52] K Li. Shared virtual memory on loosely coupled multiprocessors. URL https://www.osti.gov/biblio/6674218.

[53] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, nov 1989. ISSN 0734-2071. doi: 10.1145/75104.75105. URL https://doi.org/10.1145/75104.75105.

[54] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. *ACM Trans. Comput. Syst.*, 33(2), jun 2015. ISSN 0734-2071. doi: 10.1145/2699676. URL https://doi.org/10.1145/2699676.

[55] L. Liss, Y. Birk, and A. Schuster. Efficient exploitation of kernel access to infiniband: a software dsm example. In *11th Symposium on High Performance Interconnects, 2003. Proceedings.*, pages 130–135, 2003. doi: 10.1109/CONECT.2003.1231489.

[56] L. Liss, Y. Birk, and A. Schuster. In-kernel integration of operating system and infiniband functions for high performance computing clusters: a dsm example. *IEEE Transactions on Parallel and Distributed Systems*, 16(9):830–840, 2005. doi: 10.1109/TPDS.2005.111.

[57] Virginia Mary Lo. Operating systems enhancements for distributed shared memory. *Adv. Comput.*, 39:191–237, 1994.

[58] Robert Lyerly, Antonio Barbalace, Christopher Jelesnianski, Vincent Legout, Anthony Carno, and Binoy Ravindran. Operating system process and thread migration in heterogeneous platforms. April 2016. URL http://www.cs.utexas.edu/~mars2016/workshop-program/. The 2016 Workshop on Multicore and Rack-scale Systems, MaRS 2016 ; Conference date: 18-04-2016 Through 18-04-2016.

[59] A.M. Matsunaga, M.O. Tsugawa, and S.T. Kofuji. Evaluation of emerging high speed networks on brazos software dsm system. In *Proceedings. Eighth International Conference on Parallel and Distributed Systems. ICPADS 2001*, pages 405–411, 2001. doi: 10.1109/ICPADS.2001.934847.

[60] H. Midorikawa. The evaluation of user-level software based distributed shared memory systems. In *1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, PACRIM. 10 Years Networking the Pacific Rim, 1987-1997*, volume 2, pages 920–923 vol.2, 1997. doi: 10.1109/PACRIM.1997.620410.

[61] NASA Advanced Supercomputing (NAS) Division. NAS Parallel Benchmarks . https://www.nas.nasa.gov/software/npb.html.

[62] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL https://www.usenix.org/conference/atc15/technical-session/presentation/nelson.

[63] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, aug 1991. ISSN 0018-9162. doi: 10.1109/2.84877. URL https://doi.org/10.1109/2.84877.

[64] R. Noronha and D.K. Panda. Implementing treadmarks over gm on myrinet: challenges, design experience, and performance evaluation. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 8 pp.–, 2003. doi: 10.1109/IPDPS.2003.1213368.

[65] R. Noronha and D.K. Panda. Designing high performance dsm systems using infiniband features. In *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, pages 467–474, 2004. doi: 10.1109/CCGrid.2004.1336602.

[66] R. Noronha and D.K. Panda. Can high performance software dsm systems designed with infiniband features benefit from pci-express? In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, volume 2, pages 945–952 Vol. 2, 2005. doi: 10.1109/CCGRID.2005.1558663.

[67] NVIDIA. ConnectX-5: Advanced Offload Capabilities for the Most Demanding Applications. https://www.nvidia.com/en-us/networking/ethernet/connectx-5/, .

[68] NVIDIA. Innova-2 Flex: Advanced Programmability Delivers Acceleration and Performance. https://www.nvidia.com/en-us/networking/ethernet/innova-2-flex/, .

[69] NVIDIA. NVIDIA BlueField Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, .

[70] Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. Os support for thread migration and distribution in the fully heterogeneous datacenter. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 174–179, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350686. doi: 10.1145/3102980.3103009. URL https://doi.org/10.1145/3102980.3103009.

[71] E. W. Parsons, M. Brorsson, and K. C. Sevcik. Predicting the performance of distributed virtual shared-memory applications. *IBM Systems Journal*, 36(4):527–549, 1997. doi: 10.1147/sj.364.0527.

[72] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2017. ISBN 0128122757.

[73] Morten B. Petersen, Anthon V. Riber, Simon T. Andersen, and Martin Schoeberl. Time-predictable distributed shared memory for multi-core processors. In *2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–7, 2018. doi: 10.1109/NORCHIP.2018.8573463.

[74] R.W. Pfile, D.A. Wood, and S.K. Reinhardt. Decoupled hardware support for distributed shared memory. In *23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 34–34, 1996. doi: 10.1109/ISCA.1996.10010.

[75] J. Protic, M. Tomasevic, and V. Milutinovic. A survey of distributed shared memory systems. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference*

on System Sciences, volume 1, pages 74–84 vol.1, 1995. doi: 10.1109/HICSS.1995.375407.

[76] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: concepts and systems. *IEEE Parallel Distributed Technology: Systems Applications*, 4(2):63–71, 1996. doi: 10.1109/88.494605.

[77] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and typhoon: user-level shared memory. In *Proceedings of 21 International Symposium on Computer Architecture*, pages 325–336, 1994. doi: 10.1109/ISCA.1994.288138.

[78] S.K. Reinhardt, R.W. Pfile, and D.A. Wood. Hardware support for flexible distributed shared memory. *IEEE Transactions on Computers*, 47(10):1056–1072, 1998. doi: 10.1109/12.729790.

[79] Xiaowei Ren and Mieszko Lis. Efficient sequential consistency in gpus via relativistic cache coherence. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 625–636, 2017. doi: 10.1109/HPCA.2017.40.

[80] Robert Lyerly. Popcorn Linux: A Compiler and Runtime for State Transformation Between Heterogeneous-ISA Architectures. https://vtechworks.lib.vt.edu/bitstream/handle/10919/100599/Lyerly_RF_D_2019.pdf?sequence=1&isAllowed=y.

[81] RoCE Initiative, 2017. RoCE Accelerates Data Center Performance, Cost Efficiency, and Scalability. https://www.roceinitiative.org/wp-content/uploads/2017/01/RoCE-Accelerates-DC-performance$_F$*inal.pdf*.

[82] rsarwar87. Descriptor bypass controller for Xilinx XDMA IP for PCIe. https://github.com/rsarwar87/xdma_dsc_byp_cltr.

[83] Marina Sadini, Antonio Barbalace, Binoy Ravindran, and Francesco Quaglia. A page coherency protocol for popcorn replicated-kernel operating system. October 2013. URL http://www.mscs.mu.edu/~brylow/SPLASH-MARC-2013/. Many-Core Architecture Research Community (MARC) Symposium
at SPLASH 2013, MARC 2013 ; Conference date: 28-10-2013 Through 28-10-2013.

[84] Samsung. SmartSSD: Computational Storage Drive. https://samsungsemiconductor-us.com/smartssd/.

[85] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 323–337, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350280. doi: 10.1145/3127479.3128610. URL https://doi.org/10.1145/3127479.3128610.

[86] Weisong Shi, Weiwu Hu, and Zhimin Tang. An interaction of coherence protocols and memory consistency models in dsm systems. *SIGOPS Oper. Syst. Rev.*, 31(4):41–54, oct 1997. ISSN 0163-5980. doi: 10.1145/271019.271027. URL https://doi.org/10.1145/271019.271027.

[87] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. Cashmere-2l: Software coherent shared memory on a clustered remote-write network. *SIGOPS Oper. Syst. Rev.*, 31(5):170–183, oct 1997. ISSN 0163-5980. doi: 10.1145/269005.266675. URL https://doi.org/10.1145/269005.266675.

[88] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. Capi: A coherent accelerator processor interface. *IBM J. Res. Dev.*, 59(1):7:1–7:7, jan 2015. ISSN 0018-8646. doi: 10.1147/JRD.2014.2380198. URL https://doi.org/10.1147/JRD.2014.2380198.

[89] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *Computer*, 23:54 – 64, 06 1990. doi: 10.1109/2.53355.

[90] Systems Software Research Group. Popcorn Linux Compiler Toolchain. https://github.com/ssrg-vt/popcorn-compiler.git, .

[91] Systems Software Research Group. Popcorn Linux for Distributed Thread Execution. https://github.com/ssrg-vt/popcorn-kernel, .

[92] M.C. Tam and D. Farber. Capnet-an approach to ultra high speed network. In *IEEE International Conference on Communications, Including Supercomm Technical Sessions*, pages 955–961 vol.3, 1990. doi: 10.1109/ICC.1990.117216.

[93] M. Tomasevic and V. Milutinovic. Hardware approaches to cache coherence in shared-memory multiprocessors, part 1. *IEEE Micro*, 14(5):52–, 1994. doi: 10.1109/MM.1994.363067.

[94] M. Tomasevic and V. Milutinovic. Hardware approaches to cache coherence in shared-memory multiprocessors. 2. *IEEE Micro*, 14(6):61–66, 1994. doi: 10.1109/40.331392.

[95] Stephen M. Trimberger. Three ages of fpgas: A retrospective on the first thirty years of fpga technology. *Proceedings of the IEEE*, 103(3):318–331, 2015. doi: 10.1109/JPROC.2015.2392104.

[96] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 306–324, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132762. URL https://doi.org/10.1145/3132747.3132762.

[97] Pirmin Vogel, Andrea Marongiu, and Luca Benini. Exploring shared virtual memory for fpga accelerators with a configurable iommu. *IEEE Transactions on Computers*, 68(4):510–525, 2019. doi: 10.1109/TC.2018.2879080.

[98] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with In-Network cache coherence. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 277–292. USENIX Association, February 2021. ISBN 978-1-939133-20-5. URL https://www.usenix.org/conference/fast21/presentation/wang.

[99] Jagath Weerasinghe, Raphael Polig, Francois Abel, and Christoph Hagleitner. Network-attached fpgas for data center applications. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 36–43, 2016. doi: 10.1109/FPT.2016.7929186.

[100] Xilinx. Alveo U50 Data Accelerator Card Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds965-u50.pdf, .

[101] Xilinx. Versal ACAP CPM CCIX. https://www.xilinx.com/support/documentation/architecture-manuals/am016-versal-cpm-ccix.pdf, .

[102] Xilinx. Vivado Design Suite User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug910-vivado-getting-started.pdf, .

[103] Xilinx. AXI Reference Guide - UG761. https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, .

[104] Xilinx. AXI4-Stream Infrastructure IP Suite v3.0 - PG085. https://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf, .

[105] Xilinx. UltraScale+ Devices Integrated 100G Ethernet Subsystem v2.4 - PG203. https://www.xilinx.com/support/documentation/ip_documentation/cmac_usplus/v2_4/pg203-cmac-usplus.pdf, .

[106] Xilinx. FIFO Generator v13.2 - PG057. https://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v13_2/pg057-fifo-generator.pdf, .

[107] Xilinx. DMA/Bridge Subsystem for PCI Express v4.1 - PG195. https://www.xilinx.com/support/documentation/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf, .

[108] Xiangyao Yu and Srinivas Devadas. Tardis: Time traveling coherence algorithm for distributed shared memory. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 227–240, 2015. doi: 10.1109/PACT.2015.12.

[109] Jiajie Zhang, Zheng Yu, Zhiyi Yu, Kexin Zhang, Zhonghai Lu, and Axel Jantsch. Efficient distributed memory management in a multi-core h.264 decoder on fpga. In *2013 International Symposium on System on Chip (SoC)*, pages 1–4, 2013. doi: 10.1109/ISSoC.2013.6675256.

[110] J.Z. Zhou, M. Mizuno, and G. Singh. A sequentially consistent distributed shared memory. In *Proceedings of ICCI'93: 5th International Conference on Computing and Information*, pages 165–169, 1993. doi: 10.1109/ICCI.1993.315385.