UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIAS DA COMPUTAÇÃO

João Vicente Souto

An Inter-Cluster Communication Facility for Lightweight Manycore Processors in the Nanvix OS

Florianópolis
6 de dezembro de 2019

João Vicente Souto

An Inter-Cluster Communication Facility for Lightweight Manycore Processors in the Nanvix OS

Trabalho de Conclusão do Curso do Curso de Graduação em Ciências da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciências da Computação.
Orientador: Prof. Márcio Bastos Castro, Dr.
Coorientador: Pedro Henrique Penna, Me.

Florianópolis
6 de dezembro de 2019

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Souto, João Vicente
An Inter-Cluster Communication Facility for Lightweight Manycore Processors in the Nanvix OS / João Vicente Souto ; orientador, Márcio Bastos Castro , coorientador, Pedro Henrique Penna , 2019.
92 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2019.

Inclui referências.

1. Ciências da Computação. 2. Sistema Operacional Distribuído. 3. Camada de Abstração de Hardware. 4. Processador Lightweight Manycore. 5. Kalray MPPA-256. I. , Márcio Bastos Castro. II. , Pedro Henrique Penna. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

João Vicente Souto

An Inter-Cluster Communication Facility for Lightweight Manycore Processors in the Nanvix OS

Este Trabalho de Conclusão do Curso foi julgado adequado para obtenção do Título de Bacharel em Ciências da Computação e aprovado em sua forma final pelo curso de Graduação em Ciências da Computação.

Florianópolis, 6 de dezembro de 2019.

Prof. José Francisco Danilo De Guadalupe Correa Fletes, Me.
Coordenador do Curso

Banca Examinadora:

Prof. Márcio Bastos Castro, Dr.
Orientador
Universidade Federal de Santa Catarina

Pedro Henrique Penna, Me.
Coorientador
Université de Grenoble Alpes

Prof. Rômulo Silva de Oliveira, Dr.
Avaliador
Universidade Federal de Santa Catarina

Prof. Odorico Machado Mendizabal, Dr.
Avaliador
Universidade Federal de Santa Catarina

This work is dedicated to my colleagues, my siblings, and my dear parents.

ACKNOWLEDGEMENTS

I thank all who contributed in any way to the accomplishment of this undergraduate dissertation. In particular, I thank my advisor, Márcio Bastos Castro, my co-advisor, Pedro Henrique Penna, and the colleagues of the research group who were directly involved in the present work. I also thank the National Council for Scientific and Technological Development (CNPq) for granting the Scientific Initiation Scholarship (PIBIC), whose related activities fostered the development of this work.

If you wish to make an apple pie from scratch,
you must first invent the universe.
(SAGAN, C., 1980)

RESUMO

Em conjunto com a maior escalabilidade e eficiência energética, os processadores *lightweight manycores* trouxeram um novo conjunto de desafios no desenvolvimento de software provenientes de suas particularidades arquiteturais. Neste contexto, sistemas operacionais tornam o desenvolvimento de aplicações menos onerosos, menos suscetíveis a erros e mais eficientes. A camada de abstração provida pelos sistemas operacionais suprime as características do hardware sob uma perspectiva simplificada e eficaz. No entanto, parte dos desafios de desenvolvimento encontrados em *lightweight manycores* deriva diretamente de *runtimes* e sistemas operacionais existentes, que não lidam completamente com a complexidade arquitetural desses processadores. Acreditamos que sistemas operacionais para a próxima geração de *lightweight manycores* necessitam ser repensados a partir de seus conceitos básicos considerando as severas restrições arquiteturais. Em particular, as abstrações de comunicação desempenham um papel crucial na escalabilidade e desempenho das aplicações devido à natureza distribuída dos *manycores*. O objetivo deste trabalho é propor mecanismos de comunicação entre *clusters* para o processador *manycore* emergente MPPA-256. Estes mecanismos fazem parte de uma Camada de Abstração de Hardware (HAL) genérica e flexível para *lightweight manycores* que lida diretamente com os principais problemas encontrados no projeto de um sistema operacional para esses processadores. Sob estes mecanismos, serviços de comunicação também serão propostos para um sistema operacional baseado no modelo *microkernel*, que busca fornecer um esqueleto básico para as abstrações de comunicação. As contribuições deste trabalho estão inseridas em um contexto de pesquisa mais amplo, que procura investigar a criação de um sistema operacional distribuído baseado em uma abordagem *multikernel*, denominado *Nanvix OS*. O Nanvix OS se concentrará em questões de programabilidade e portabilidade através de um sistema operacional compatível com o padrão POSIX para *lightweight manycore*. Os resultados mostram como algoritmos distribuídos conhecidos podem ser eficientemente suportados pelo Nanvix OS e incentivam melhorias providas pelo uso adequado dos aceleradores de Acesso Direto à Memória (DMA).

Palavras-chave: HAL. Sistema Operacional Distribuído. *Lightweight Manycore*. Kalray MPPA-256.

ABSTRACT

Jointly with further scalability and energy efficiency, lightweight manycores brought a new set of challenges in software development coming from their architectural particularities. In this context, Operating Systems (OSs) make application development less costly, less error-prone, and more efficient. The abstraction layer provided by OSs suppresses hardware characteristics from a simplified and productive perspective. However, part of the development challenges encountered in lightweight manycores stems from the existing runtimes and OSs, which do not entirely address the complexity of these processors. We believe that OSs for the next generation of lightweight manycores must be redesigned from scratch to cope with their tight architectural constraints. In particular, communication abstractions play a crucial role in application scalability and performance due to the distributed nature of manycores. The purpose of this undergraduate dissertation is to propose an inter-cluster communication facility for the emerging manycore MPPA-256 processor. This facility is part of a generic and flexible Hardware Abstraction Layer (HAL) that deals directly with the key issues encountered in designing an OS for these processors. Above this facility, communication services will also be proposed for an OS based on the *microkernel* model, which seeks to provide a basic framework for communication abstractions. The contributions of this undergraduate dissertation are embedded in a broader research context that aims to investigate the creation of a distributed OS based on a multikernel approach, called *Nanvix OS*. Nanvix OS focuses on programmability and portability issues for manycores through a POSIX-compliant OS. The results present how well known distributed algorithms can be efficiently supported by Nanvix OS and encourage improvements provided by the proper use of Direct Memory Access (DMA) accelerators.

Keywords: HAL. Distributed Operating System. Lightweight Manycore. Kalray MPPA-256.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface. 50
C-NoC	Control Network-on-Chip. 41, 53, 54, 55, 58, 59, 60, 62, 63, 64
CMP	Chip Multiprocessor. 33
COW	Copy-On-Write. 39
CPU	Central Processing Unit. 31, 32, 33, 34, 35, 37, 38, 39, 49
D-NoC	Data Network-on-Chip. 41, 53, 54, 55, 60, 61, 62, 63, 64
DMA	Direct Memory Access. 13, 38, 41, 43, 49, 53, 54, 55, 56, 58, 63, 73, 75
DRAM	Dynamic Random Access Memory. 41
FLOPS	Floating-point Operations per Second. 27
FPGA	Field Programmable Gate Array. 49
GPU	Graphics Processing Unit. 33
HAL	Hardware Abstraction Layer. 13, 29, 30, 42, 43, 44, 51, 53, 54, 55, 56, 58, 60, 65, 66, 75, 89, 90
HPC	High-Performance Computing. 50
IEEE	Institute of Electrical and Electronics Engineers. 41
IO	Input/Output. 31, 35, 36, 43
IOCTL	Input/Output Control. 66, 69
IPC	Inter-Process Communication. 64
MIMD	Multiple Instruction Multiple Data. 27, 33, 40
MISD	Multiple Instruction Single Data. 33
MMIO	Memory-Mapped I/O. 43
MMU	Memory Management Unit. 31, 40, 43, 49
MOOSCA	Manycore Operating System for Safety-Critical Application. 50
mOS	multi Operating System. 50
MPI	Message Passing Interface. 30, 69, 70, 75
MPSoC	Multiprocessor System-on-Chip. 33
NoC	Network-on-Chip. 27, 28, 34, 41, 43, 49, 50, 51, 53, 54, 55, 60, 62, 90
NUMA	Non-Uniform Memory Access. 32, 33, 50, 51

CONTENTS

1 INTRODUCTION

For several years, the increase in the frequency of processors was employed as the main technique for achieving performance improvements. However, as a side effect, the temperature of processors started rising to high values, thus imposing a physical limit to the aforementioned technique. Alternatively, the constant improvement of semiconductor technology helped to mitigate the impact of this problem, allowing the industry to build more powerful processors with the same frequency. Therefore, knowing the frequency barrier and the imminent end of Moore's Law (MOORE, 1965), the academy and industry began to research and invest in alternatives to keep increasing the processing power of computer systems.

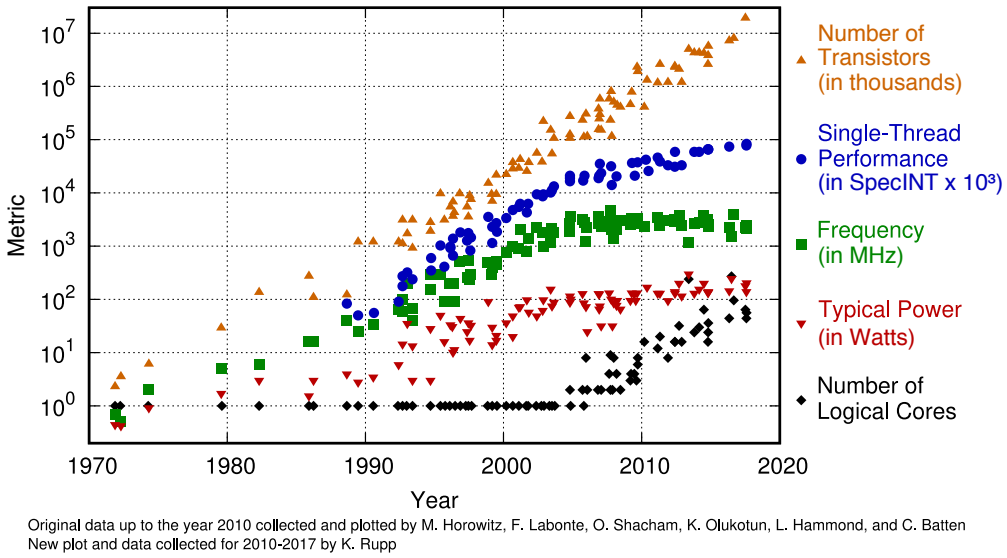
Figure 1 illustrates the paradigm shift that processors have gone through to the present day. From mid-2000, the frequency of processors tended to stagnate. The steady increase in transistors in the same chip area and the vast diversity of trade-offs to improve single-thread performance has softened the frequency impact on processors. Some significant trade-offs are different types of instruction sets, instruction parallelism, out-of-order processing techniques, branch prediction techniques, and various memory hierarchies. Then, in mid-2005, the performance of computer systems was pushed even further by increasing the number of processing cores in a single die. These architectures, called *multicores*, allowed the continuous rise of the computing performance.

The ever-increasing number of transistors and cores in a chip quickly led to the advent of manycores. Notwithstanding, the line between *multicores* and manycores is very tenuous. Some researchers argue that in the latter architectures, losing a core it will not significantly impact the performance of the platform. A system is classified as manycore when there is a need for distributed memory and on-chip networking (FREITAS, 2009).

Yet another classification for manycores is based on their ratio between processing speed, measured by the number of Floating-point Operations per Second (FLOPS), and power consumption, in Watts (W). Figure 1 pictures that even as the number of cores increasing, typical power has not grown uncontrollably. For instance, to achieve *exascale* (10^{18} FLOPS), the US Department of Defense issued a report stipulating the energy efficiency of a supercomputer should be around 50 GFLOPS/W (KOGGE et al., 2008). To cope with this energy constraint, a new class of parallel processors, called *lightweight manycores*, emerged to provide high parallelism with low power consumption. Lightweight manycores differ from traditional large-scale multicores and manycores in several points:

- They integrate thousands of low-power cores in a single die organized in clusters;
- They are designed to cope with Multiple Instruction Multiple Data (MIMD) workloads;
- They rely on a high-bandwidth Network-on-Chip (NoC) for fast and reliable message-passing communication;

Figure 1 – Multiprocessor evolution.



Source: Adapted from Rupp (2018).

- They have constrained memory systems; and
- They frequently feature a heterogeneous configuration.

Some industry-successful examples of lightweight manycores are the Kalray MPPA-256 (DINECHIN et al., 2013); the Adapteva Epiphany (OLOFSSON; NORDSTROM; UL-ABDIN, 2014); and the Sunway SW26010 (ZHENG et al., 2015). Together with superior performance scalability and energy efficiency, lightweight manycores brought a new set of challenges in software development coming from their architectural particularities. More precisely, these introduced the following difficulties:

- *Hybrid programming model:* due to the parallel and distributed nature of the architecture, engineers are frequently required to adopt a message-passing programming model to deal with the presence of rich NoCs (KELLY; GARDNER; KYO, 2013) that interconnects clusters and a shared-memory model inside the cluster;
- *Missing hardware support for cache coherency:* to reduce power consumption, theses processors do not feature cache coherency, which in turn forces programmers to handle it explicitly in software level and frequently calls out for a redesign in their applications (FRANCESQUINI et al., 2015);
- *Constrained memory system:* the frequent presence of multiple physical address spaces and small local memories require data tiling and prefetching to be handled by the software (CASTRO et al., 2016);
- *Heterogeneous configuration:* the different programmable components on lightweight manycores turns the actual deployment of applications in a complex task (BARBALACE et al., 2015).

Part of these challenges derives from existing runtimes and Operating Systems (OSs). On the one hand, runtimes do not hide the characteristics of hardware making software development more challenging and non-portable, e.g., they neither allow direct access to non-local data, nor the manipulation of them in a transparent way. Thus, fundamental OS mechanisms, such as core multiplexing, core partitioning, and process and data migration, may not be addressed. On the other hand, the complicated portability and scalability of traditional OSs with monolithic kernels, which were designed to homogeneous hardware, is leading to alternative OS designs (BAUMANN et al., 2009; KLUGE; GERDES; UNGERER, 2014; NIGHTINGALE et al., 2009; RHODEN et al., 2011).

We believe that OSs for the next-generation of lightweight manycores must be redesigned from scratch to cope with their tight architectural constraints. Based on this idea, a new fully-featured distributed OS based on a multikernel approach (BAUMANN et al., 2009) is under investigations (PENNA et al., 2017; PENNA et al., 2017; PENNA et al., 2019). The *Nanvix Multikernel* features a generic and flexible Hardware Abstraction Layer (HAL) for lightweight manycores that addresses the key issues encountered in the development for these processors. On top of the *Nanvix HAL*, a microkernel is being designed and implemented to provide the bare bones of the most important system abstractions.

1.1 GOALS

Based on the aforementioned motivations, the primary and specific goals of this work are detailed next.

1.1.1 Main Goal

The main goal of this undergraduate dissertation is to propose an *Inter-Cluster Communication Module* to the *Nanvix HAL* and port it to the Kalray MPPA-256 manycore processor (DINECHIN et al., 2013). This module exposes the essential abstractions that allow overlying layers to create richer communication services. Using this module, we also propose *Inter-Cluster Communication Services* to the *Nanvix Microkernel*. This work is part of the collaborative project between Federal University of Santa Catarina (UFSC), Pontifical Catholic University of Minas Gerais (PUC Minas), and University of Grenoble Alpes (UGA) to develop an OS for lightweight manycore platforms.

1.1.2 Specific Goals

- Definition and proposal of an *Inter-Cluster Communication Interface* for lightweight manycores;

- Implementation of the proposed interface in the *Nanvix HAL* for the Kalray MPPA-256 lightweight manycore processor;
- Integration of the Nanvix HAL interface with the *Nanvix Microkernel*;
- Performance evaluation of Nanvix Microkernel implementation using synthetic micro-benchmarks that reproduce the *collective communication routines* of the Message Passing Interface (MPI) programming model.

1.2 ORGANIZATION OF THE WORK

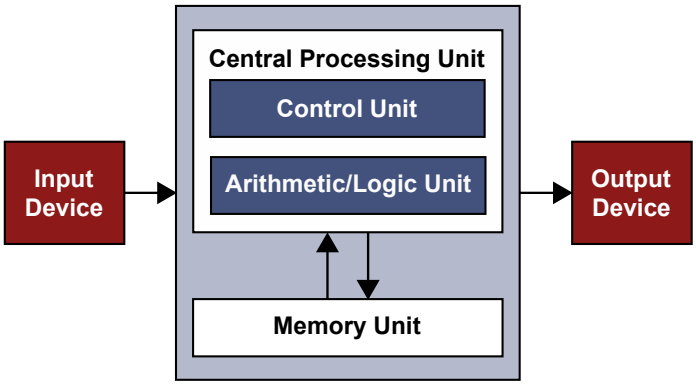
The remainder of this work is organized as follows. In Chapter 2, we present a background on OS and communication design for multicores and multicomputers, the MPPA-256 lightweight manycore processor and the Nanvix project. In Chapter 3, we discuss the principal related work. In Chapter 4, we discuss the design and implementation of the inter-cluster communication facility. In Chapter 5, we detail the evaluation methodology that we adopted and analyze our experimental results. Finally, in Chapter 6, we draw our conclusions and future work.

2 BACKGROUND

In the early days of electronic digital computing, John Von Neumann proposed an architectural model for computers to be easily programmable (NEUMANN, 1945). Figure 2 pictures the model scheme. The Central Processing Unit (CPU), also called core, loads instructions and data from an Memory Management Unit (MMU), dealing with inputs and generating outputs from/to Input/Output (IO) devices. Modern processors still follow this model, but some components and behaviors are specialized or replicated to increase performance.

According to Tanenbaum & Bos (2014), there are three models of modern multiple processor architectures. A shared-memory multiprocessor, a message-passing multicomputer, and a wide area distributed system. In this chapter, we present the background of multiple processor systems from a hardware and software perspectives. Specifically, Section 2.1 and Section 2.2 address details about shared-memory multiprocessors and message-passing multicomputers, respectively. Then, Section 2.3 presents the Kalray MPPA-256 processor. Finally, Section 2.4 shows an overview of Nanvix OS and the three levels of abstraction designed for lightweight manycores processors.

Figure 2 – Von Neumann architecture model.



Source: Adapted from Tanenbaum & Bos (2014).

2.1 MULTIPROCESSORS

A shared-memory multiprocessor is a computer system in which two or more CPUs share full access to the same Random Access Memory (RAM) (TANENBAUM; BOS, 2014). The ability to run execution streams in parallel intensifies the concurrency issues that already existed in multi-tasking single-core processors. The competition of different cores for shared resources may result in inconsistent outputs or even in an inconsistent OS state. For instance, this inconsistency occurs when a thread writes a value to a global variable and reads a different one. Thus, to solve this problem, special instructions

are required to enable synchronization between threads. Moreover, some architectures integrate heterogeneous cores introducing portability and programmability problems too. So, low-level software, such as OS kernels and runtimes, needs to handle those issues and provide management systems to user-level.

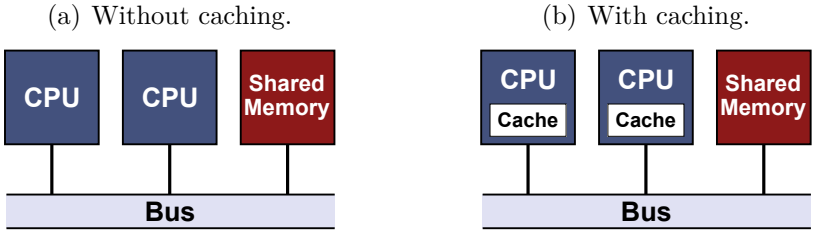
2.1.1 Multiprocessor Hardware

Multiprocessors can be usually classified according to memory access and workflow properties. In the first place, the access time to different memory addresses split multiprocessors into two groups. On the one hand, the group of systems that can read a memory word as fast as every other memory word are called Uniform Memory Access (UMA) multiprocessors. On the other hand, Non-Uniform Memory Access (NUMA) multiprocessors do not have this property.

The first UMA multiprocessors were bus-based architectures where the CPU waits for the bus channel to become free to perform a memory access, as illustrated in Figure 3(a). When the number of cores scale, the bus traffic becomes a bottleneck of the system. To solve this problem, a small but fast memory level, called cache, is added to each CPU, as depicted in Figure 3(b). The cache allowed successive readings to be resolved locally, reducing traffic to the main memory. However, many problems of inconsistency and ordering of operations on memory arose with the advent of caches. For instance, when a write operation modifies a memory address in a particular cache, this change must be notified to all other caches. Equally important, it is necessary to ensure a specific order in the concurrent operations on a given address through different caches. The protocols that guarantee these properties are called cache-coherence protocols (TANENBAUM; BOS, 2014).

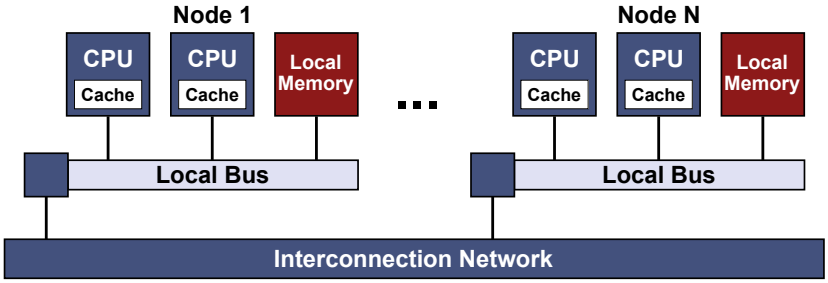
Nevertheless, the number of cores in UMA multiprocessors are limited to a few dozens of CPUs. NUMA multiprocessors provide much more scalability by employing a single address space visible to all CPUs through an interconnection network, as shown in Figure 4. The network is used to interconnect several memory blocks (also known as local memories) into a single address space, allowing hundreds of cores to communicate

Figure 3 – Two bus-based UMA multiprocessor examples.



Source: Adapted from Tanenbaum & Bos (2014).

Figure 4 – NUMA multiprocessor example.



Source: Adapted from Tanenbaum & Bos (2014).

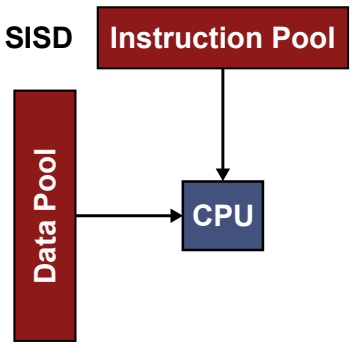
more efficiently. The side effect is that the interconnection network affects the latency observed by the processors to access the main memory, because the memory access time will depend on the memory location relative to the processor, which could be *local* or *remote*. The single address space provided by NUMA multiprocessors allows parallel programs that were originally developed for UMA processors to run without any source code changes. However, these applications usually achieve sub-optimal performance on NUMA multiprocessors, because there were not optimized to take into account local and remote memory accesses.

In the second place, the workflow classification proposed by Flynn (1972), split multiprocessors architecture based on the number of concurrent instruction and data streams available, as depicted in Figure 5. First, the most straightforward class, Single Instruction Single Data (SISD) describes a sequential machine which exploits no parallelism in either the instruction or data streams, like older uniprocessor machines. Second, Single Instruction Multiple Data (SIMD) uses multiple functional units to replicate and operate a single instruction over multiples different data streams, like Graphics Processing Unit (GPU). Third, the most uncommon class, Multiple Instruction Single Data (MISD) describe multiprocessors that apply multiple instructions streams over one data stream. Systems that need fault tolerance use theses multiprocessors, like modern flight control systems. Finally, a Multiple Instruction Multiple Data (MIMD) architecture has multiple processors simultaneously executing different instruction on different data, like modern processors from Intel.

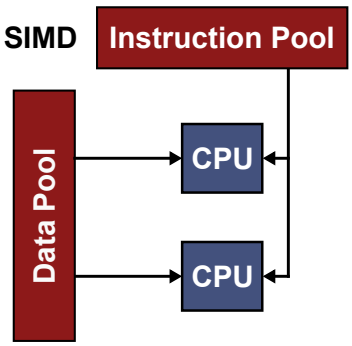
Currently, two categories of multiprocessors attract attention, the Chip Multiprocessor (CMP) and Multiprocessor System-on-Chip (MPSoC). CMPs are commercial multicores, which follow a symmetric architecture, integrating two or more identical cores into a single die. They can have private or shared cache levels, and always share access to the RAM. Alternatively, MPSoCs are designed with an asymmetric architecture, have in addition to the main cores, specialized CPUs in particular functions, e.g., audio and video encoders, encryption, becoming truly complete computer systems on a single chip. All these cores are linked to each other by an on-chip network-based communication sub-

Figure 5 – Flynn's taxonomy.

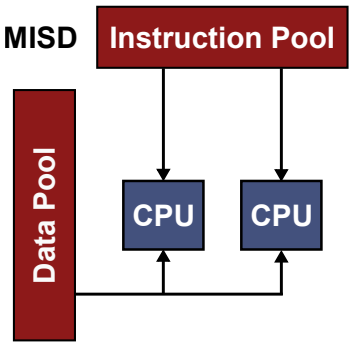
(a) Single Instruction Single Data.



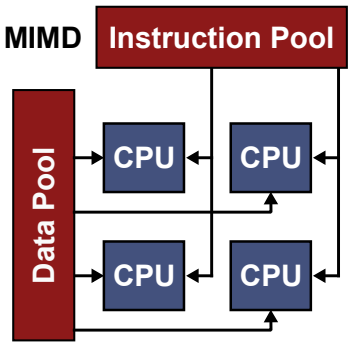
(b) Single Instruction Multiple Data.



(c) Multiple Instruction Single Data.



(d) Multiple Instruction Multiple Data.



Source: Adapted from Wikipedia (2019).

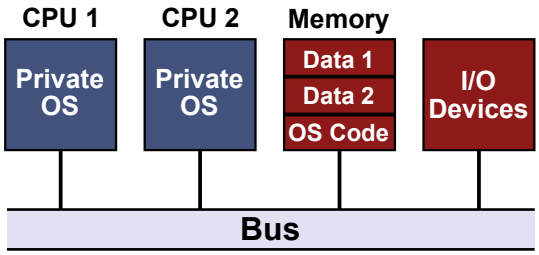
system, called NoC. The NoC improves scalability and power consumption compared to other communication subsystem designs.

2.1.2 Multiprocessor Operating Systems

OSs are a fundamental part of a computer system. They act as an intermediary between users and hardware, with the purpose to provide an environment in which users can run programs in a conveniently and efficiently manner (SILBERSCHATZ; GALVIN; GAGNE, 2012). Many OS approaches exist in multiprocessor systems. In particular, three of them express accurately the difficulties of developing OSs targeting the concurrency issues existing in such systems. Those models are called Replicated, Master-Slave, and Symmetric OS.

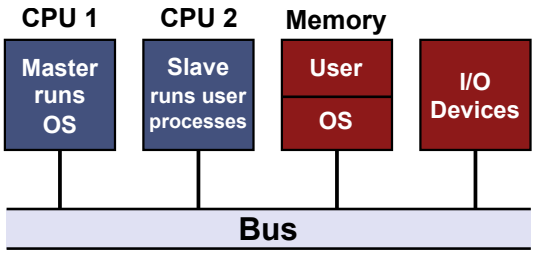
The Replicated Model is the simplest way to develop an OS for a parallel architecture. It only needs to replicate all the internal OS structures for each core. Figure 6 illustrates how this model allocates fixed memory spaces between the cores, giving each of them its private OS. The system calls are performed by the calling CPU, avoiding concurrency issues. Also, a producer-consumer model is sufficient for two different CPUs to communicate.

Figure 6 – Replicated OS model.



Source: Adapted from Tanenbaum & Bos (2014).

Figure 7 – Master-Slave OS model.



Source: Adapted from Tanenbaum & Bos (2014).

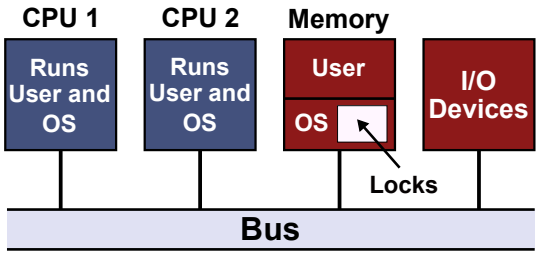
This model is still better than having separate computers. Nevertheless, the application of this model must first assess three aspects (TANENBAUM; BOS, 2014). First, since each CPU has its own process and page tables, it is impossible to optimize the use of resources. For instance, if many of processes are waiting for use an overloaded CPU, it is impossible to migrate them to an available CPU. Second, operations with IO devices can introduce inconsistency problems such as the same disk block operated by different CPUs. Finally, replication of the internal OS structures makes this model impractical for systems with memory constraints.

The Master-Slave model began to attract attention with the return of processors with no cache coherence. As Figure 7 pictures, there is only one copy of the internal OS structures, and they all belong to a single CPU, called master. In this way, all system calls performed by a worker CPU, called slave, are redirected to the master. With these changes, this model solves the problems of the previous model by using only one copy of the data structures. For illustration, processes and memory pages can be scheduled and distributed dynamically to any CPUs. However, when adopting a centralized approach, the master can become the bottleneck of the system if it can not handle the number of the incoming requests.

Finally, the Symmetric model, called Symmetric Multi-Processing (SMP), eliminates the centralization problem of the foregoing model, as illustrated in Figure 8. So, there is still only one copy of the OS structures but it is shared in memory. When a CPU issues a system call, it loads the structures and operates on them. Consequently, processes and memory pages also continue to be dynamically balanced. The difficulties introduced by this model lie in concurrency for OS structures. Depending on how the critical regions are managed, the performance of the system may be equivalent to the Master-Slave model. So the hardest part is breaking the OS into critical regions that will run on different CPUs, where one core does not affect the execution of another or fall into a deadlock (TANENBAUM; BOS, 2014). Besides, if the hardware does not support cache coherence, the process of invalidating the cache may also introduce serious performance problems in OSs of this type.

As it can be noted, the software is always lagging behind the constant hardware

Figure 8 – Symmetric OS model.



Source: Adapted from Tanenbaum & Bos (2014).

advances. Many solutions may work very well in specific contexts but should be chosen with care. In some cases, in order to extract the maximum performance from a system, it will be necessary to redesign the whole software stack from scratch.

2.2 MULTICOMPUTERS

Increasing the number of cores and still providing a shared memory in a single die is very expensive and challenging. However, it is more simple and cheap to interconnect more straightforward computers in a high-speed network. The result is a clustered architecture. Despite the problem of developing networks and high-speed interfaces for communication of the nodes, it is analogous to the problem of providing a shared memory in multiprocessors. Nevertheless, the expected communication times will be in the microseconds, as opposed to nanoseconds of the multiprocessors, making things simpler (TANENBAUM; BOS, 2014).

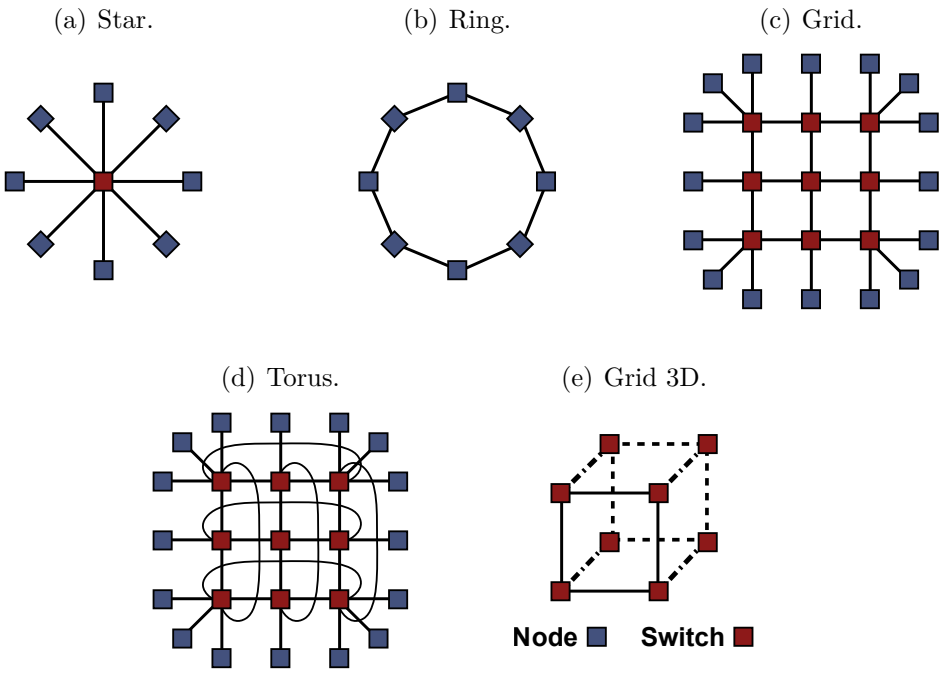
2.2.1 Multicomputer Hardware

A multicomputer node can be considered as an elementary computer, with one or more multiprocessors, local RAM and IO devices. In many cases, there is no need for monitors or keyboards, only the network interface. In this way, it is possible to integrate hundreds or even thousands of nodes providing the vision of a single computer.

A switch set is organized into different topologies to interconnect the nodes of a multicomputer. As illustrated in Figure 9, there are a variety of topologies with their own characteristics. For instance, commercial multicomputer usually uses bi-dimensional topologies such as *grid* or *mesh* because they present regular behavior and can scale easily. When the goal is to provide higher fault tolerance, in addition to the smaller path between two points, the *torus* variant implement connections between the extreme points of the *grid*. Even multi-dimensional topologies can be used, all depending on the characteristics expected from the network.

There are two types of switching schemes in the multicomputer network. The

Figure 9 – Network topologies examples.



Source: Adapted from Tanenbaum & Bos (2014).

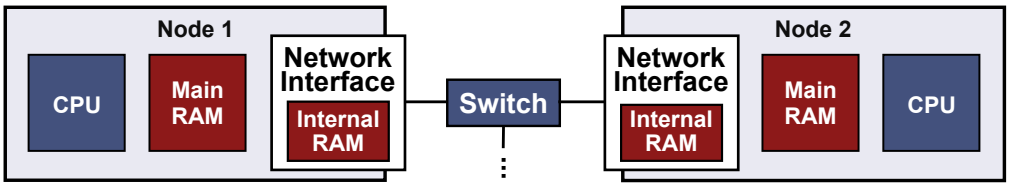
store-and-forward packet switching scheme breaks the message into fixed-size packets. The packets are copied and moved between the switches following a routing algorithm until they reach the destination. Although flexible and efficient, this scenario can generate a variable latency in packet delivery. The other scheme, called *circuit switching*, performs a resource allocation protocol through all path from source to the destination. This protocol ensures a steady communication stream, although the slow start and possible sub-utilization of the resources.

2.2.2 Low-Level Communication Software

Multicomputer nodes are interconnected to each other through network interfaces. Because these boards are built and connected to CPUs and RAM, they have substantial impacts on system performance and OS design. Virtually, interfaces have enough RAM space to receive/send packets. If this address space is actually in main memory, we fall into the same problem of multiprocessors in the struggle for the use of the bus channel. Thus, in general, network cards have a dedicated memory so as not to generate bottlenecks in access to main memory, as illustrated in Figure 10.

However, excessive packet copying can degrade the performance of the system. In an ideal scenario, four end-to-end copies would be needed: (i) from the RAM of the sender to the interface memory; (ii) from the interface to the network; (iii) from the network to the memory of the target interface, and, finally, (iv) to the RAM of the

Figure 10 – Simple Multicomputer Example.



Source: Adapted from Tanenbaum & Bos (2014).

recipient. Notwithstanding, the number of copies may increase further, depending on how the OS implements communication services on available hardware. For instance, mapping the interface into the kernel address space rather than the user-space, an extra copy to an internal kernel buffer is required. Thus, for performance reasons, modern systems already map the interfaces to user-space address even as new concurrency issues arise over communication resources.

Processors may also have one or more CPUs specialized in communication procedures, called Direct Memory Access (DMA). DMAs can make copies between system memories, send/receive packets without the main CPUs intervention. This reduces considerably wasted cycles due to network interfaces communication and/or main memory access bottlenecks. However, such intermediate copies lead to overhead on system structures, such as cache, Translation Lookaside Buffer (TLB), or page management. Furthermore, this introduces concurrency issues in the interaction between CPUs and existing DMA channels.

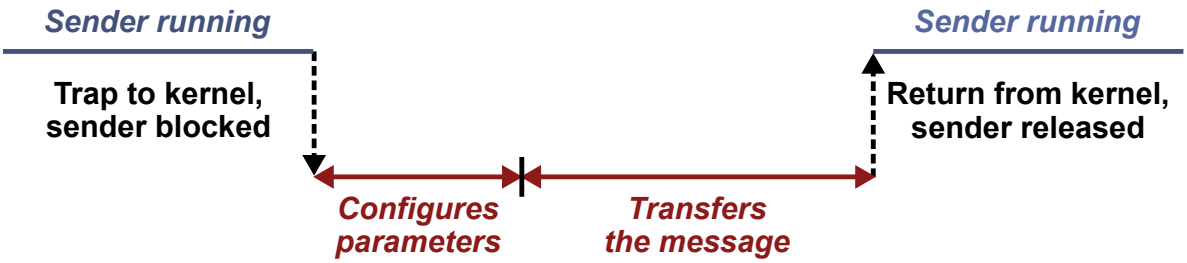
2.2.3 User-Level Communication Software

The low-level mechanisms discussed above allow cores on different computers to communicate through the messages exchange by send/receive primitives. However, it is still the responsibility of the user to define the required parameters to perform such operations. To send a message, one must inform the localization of the message, its size, and the identifier of the receiving interface. On the receiver side, the user must configure the interface with the location of sufficient memory space to receive the incoming message.

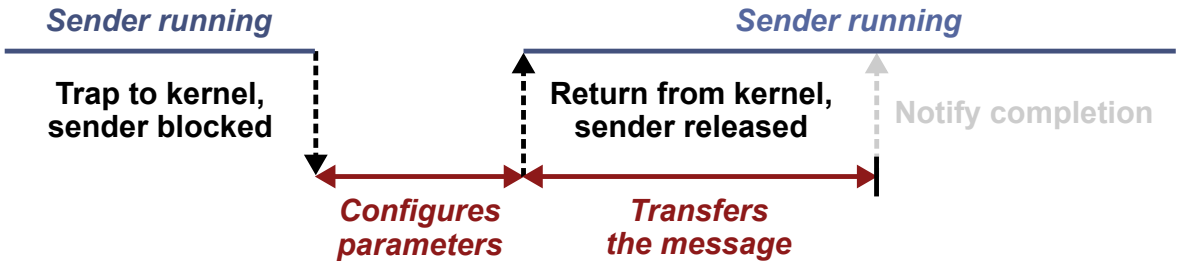
Figure 11 illustrates the two approaches to implement these primitives, either through blocking or non-blocking calls. Blocking calls, called *synchronous calls*, block the requesting CPU until complete the procedure. Non-blocking calls, called *asynchronous calls*, return control to the CPU while the procedure is still in progress. Although asynchronous calls provide better performance than synchronous ones, they introduce some disadvantages where the sender/receiver cannot use the message buffer before the operation is complete. According to Tanenbaum (TANENBAUM; BOS, 2014), there are four ways to implement a send primitive:

Figure 11 – Synchronous and asynchronous calls.

(a) Synchronous call on sender node.



(b) Asynchronous call on sender node.



Source: Adapted from Tanenbaum & Bos (2014).

- *Blocking sending:* CPU hibernates or schedules another process, while the message is transmitted;
- *Non-blocking sending with copying:* performs an extra copy of the message to a kernel buffer, degrading performance;
- *Non-blocking sending with interrupt:* notifies the CPU when the send finishes, where the buffer must remain untouchable, make more challenging software programming;
- *Copy-On-Write (COW):* management of buffers to make an extra copy only when needed, but can copy unnecessarily.

Analogously, there are other four forms to implement a receive primitive:

- *Blocking receive:* CPU hibernates or schedules another process until a message is received;
- *Non-Blocking receive with messages pool:* CPU creates a buffer to store incoming messages, then consumes from it when there is some message available, requiring synchronization;
- *Non-Blocking receive with Pop-up Threads:* creates a specific thread upon receiving a message to perform the necessary operations, but consumes resource for creating and destroying the thread;
- *Non-Blocking receive with interrupt handlers:* the receiver is interrupted to execute a handler when receiving a message, resulting in a better performance than creating a thread but makes programming more difficult.

Some of the implementation approaches may be hardware dependent. However, choosing the ideal approach is still the responsibility of the OS designer. Even so, the distributed nature of multicomputers forces a message-passing strategy regardless of what the hardware has to offer.

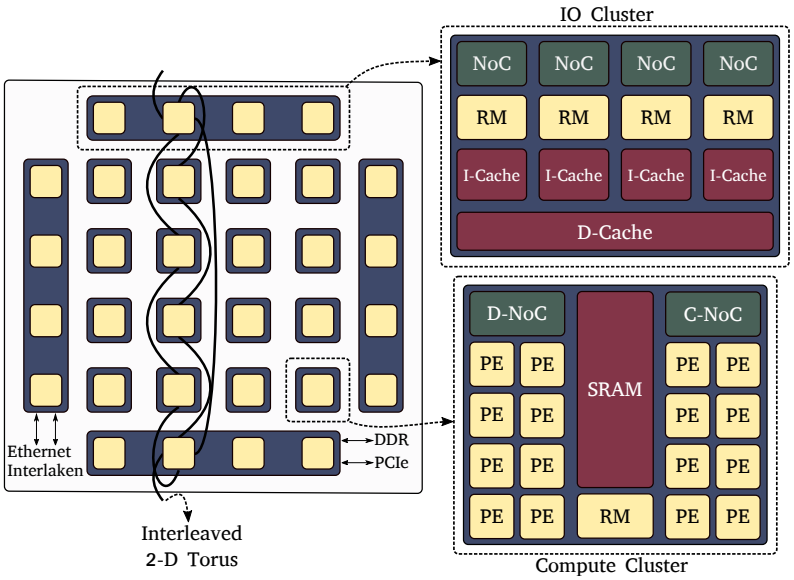
2.3 MPPA-256 LIGHTWEIGHT MANYCORE PROCESSOR

The lightweight manycores class features processors that have a vast number of cores and endeavor to be energy efficient. In general, they no longer provide global memory and work with distributed memory, and the communication among cores must be made explicitly through message exchanges. The Kalray MPPA-256 fits this processor profile.

Specifically, Kalray MPPA-256 is a high-performance, lightweight multicore processor developed by the French company Kalray. Developed to handle MIMD workloads, Kalray MPPA-256 mixes features of multiprocessors and multicomputers on a single chip. Precisely, the coordination of cores within a cluster utilizes multiprocessor concepts and inter-cluster communication employs multicomputer concepts.

Figure 12 illustrates the version of Kalray MPPA-256 architecture used, called Bostan. It has 256 general-purpose cores and 32 firmware cores, called Processing Elements (PEs) and Resource Managers (RMs), respectively. The processor use 28 mm CMOS technology and all cores run at 500 MHz. Besides, all cores have caches and MMUs with software-managed TLBs. Finally, the 288 cores are grouped into 16 Compute Clusters, dedicated to the payload, and 4 I/O Clusters, responsible for communicating with peripherals.

Figure 12 – Architectural overview of the Kalray MPPA-256 processor.



Source: Penna et al. (2018).

Each Compute Cluster features 16 PEs, an RM, an NoC interface and 2 MB of Static Random Access Memory (SRAM). The hardware does not support cache coherence to improve energy consumption. In contrast, I/O Clusters have only 4 RMs with cache coherence support, 4 NoC interfaces, and 4 MB of local SRAM added to 4 GB of Dynamic Random Access Memory (DRAM). The address space on each cluster is private, forcing exchange messages by one of two different interleaved 2-D Torus NoCs. On the one hand, the Control Network-on-Chip (C-NoC) is exclusive to 64-bit control messages, usually used for synchronization. On the other hand, intense exchange data occurs through the Data Network-on-Chip (D-NoC). Additionally, all clusters have available DMAs associated with each NoC interfaces to handle communication issues.

As discussed in Subsection 2.2.3, NoC interfaces expose communication resource to perform send and receive primitives like network interfaces. Accurately, they summarize the following resources:

- 128 slots for receiving commands;
- 256 slots for receiving data;
- 4 channels for sending commands;
- 8 channels for sending data, and;
- 8 μthreads for sending asynchronous data (each of which must to be associated with a data transfer channel).

The configuration of these features is accomplished by a mix between writing on DMA registers and performing syscalls to a hypervisor that virtualizes the Kalray MPPA-256 hardware.

2.4 NANVIX: AN OPERATING SYSTEM FOR LIGHTWEIGHT MANYCORES

During the 1970s and 1980s, several OS initiatives began to emerge. To prevent OSs from being incompatible with each other, the Institute of Electrical and Electronics Engineers (IEEE) creates the Portable Operating System Interface (POSIX) standardization. POSIX defines interfaces and behaviors expected from an OS, e.g., creation and control of processes. Many of these systems ceased to exist, and new ones emerged, but POSIX has consolidated and continues to extend to new concepts.

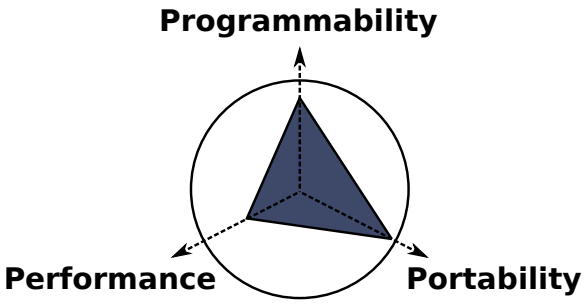
The vast majority of current OSs are designed to work with a small number of cores. As the number of cores increases, certain parts of the OS require redesign. At some point, this rework will be unfeasible. This assertion led researchers to study design alternatives for the new era of processors (WENTZLAFF; AGARWAL, 2009; BAUMANN et al., 2009; WISNIEWSKI et al., 2014). However, the focus of these OSs is on improving performance. Occasionally, aspects of hardware and design interfere with OS interfaces and behavior. Additionally, a lack of alternative OSs for lightweight manycores that

balance the development cost requires from the user a considerable development and debugging effort.

In this context, current research efforts on Nanvix OS focus on the programmability and portability challenges that have arisen with lightweight manycores (CHRISTGAU; SCHNOR, 2017; GAMELL et al., 2012; SERRES et al., 2011). We believe that significant barriers will still arise in this scenario, and the solution is to rethink OS design from scratch without losing back compatibility (PENNA; FRANCIS; SOUTO, 2019; PENNA et al., 2019). Figure 13 depicts the three main conceptual goals of Nanvix OS . The main efforts in Nanvix OS are focused on programmability and portability issues of lightweight manycores. This can be accomplished through a fully-featured POSIX-compliant OS (PENNA; FRANCIS; SOUTO, 2019).

Nanvix OS is composed of three distinct kernel layers. In a bottom-up approach of abstraction level, they are namely *Nanvix HAL* , *Nanvix Microkernel*, and *Nanvix Multikernel*. The next sections will introduce the concepts and problems addressed by each layer.

Figure 13 – Conceptual goals of the Nanvix OS.



Source: Adapted from Penna et al. (2019)

2.4.1 Nanvix Hardware Abstract Layer (HAL)

Nanvix OS proposes a generic and flexible HAL around the intrinsic architectural characteristics of the lightweight manycores. Currently, Nanvix HAL features all the essential modules to run standalone processes for the Kalray MPPA-256 (DINECHIN et al., 2013), OpTiMSoC (WALLENTOWITZ et al., 2013), and HERO (KURTH et al., 2017) platforms. In this work, we focus on proposing an inter-cluster communication module for the Nanvix HAL , and providing its implementation for the Kalray MPPA-256 processor.

Unlike other approaches that aim to design a fully-featured OS (BAUMANN et al., 2009; KLUGE; GERDES; UNGERER, 2014; NIGHTINGALE et al., 2009; RHODEN et al., 2011), the Nanvix HAL belongs to one level below. It is the first layer on top of the hardware and should provide a standard view of these emerging processors for a

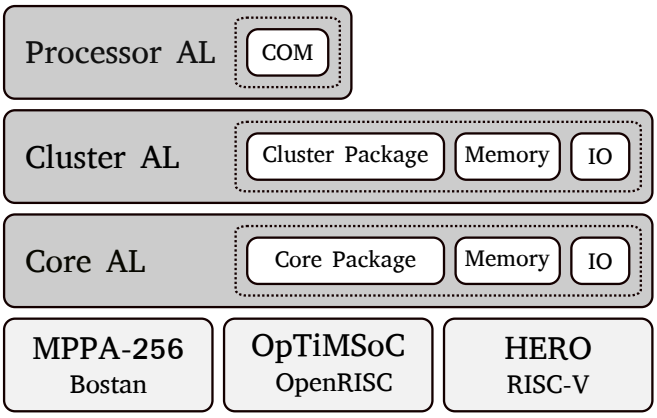
client application, e.g., OS. At this level, we do not protect internal structures with locks. Thus, the upper-level decides how to protect and multiplex HAL access. On the one hand, a traditional monolithic OS that implements a time-sharing behavior to access internal structures will need to apply locks manually. On the other hand, a master-slave OS not require such strict control, improving its performance. Figure 14 pictures the three logic layers of the Nanvix HAL , which are detailed below:

Core Abstraction Layer encapsulates the management of a single core. It provides a uniform view and maintenance routines for TLBs, MMU and cache, a rich support of handling exceptions/traps/interrupts, and access to Port-Mapped I/O (PMIO). Therefore, some design decisions are made to create interfaces that are not dependent on the underlying hardware. For example, a context switch mechanism was not provided in the *Core Package* because this would force the client OS to write code in assembly, hurting the conceptual idea of the Nanvix HAL .

Cluster Abstraction Layer bundles the resource management that concern all cores in a cluster. It provides support for virtual memory, cross-core notification through events, clock counter fetching, and access to IO resources such as Memory-Mapped I/O (MMIO) and DMA.

Processor Abstraction Layer embraces architectural features related to multiple clusters. The *inter-cluster communication module*, which is the focus of this undergraduate dissertation, provides NoC identification routines and exports three main abstractions to allow synchronization and data exchange among clusters, based on ideas proposed along with the NodeOS distributed runtime system (DINECHIN et al., 2013).

Figure 14 – Structural overview of the Nanvix HAL.



Source: Adapted from Penna, Francis & Souto (2019)

2.4.2 Nanvix Microkernel

The *Nanvix Microkernel*, running above the Nanvix HAL , provides bare bones system abstractions to the client applications (PENNA et al., 2019). Through a rich system call interface, it follows a Master-Slave OS model to avoid cache coherence issues present in manycores. The scope of the Nanvix Microkernel is within a single cluster. System calls and internal subsystems manage local and shared resources available to all cores to maintain the coherence of the OS. Figure 15 pictures the five logic layers of the Nanvix Microkernel, which are detailed below:

IPC Facility encapsulates the rest of this undergraduate dissertation, providing manipulation and protection of the inter-cluster communication abstractions.

Thread System implements a rich mechanism for creation, management, and protection of threads. This system features schedules, sleep, and wake up routines of the user threads.

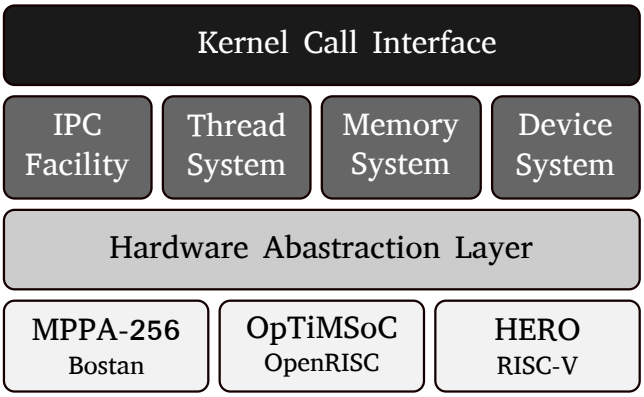
Memory System provides a rich memory management and virtual memory support at cluster level. It is based in a two-level paging scheme, supports pages of heterogeneous sizes and uses a capabilities system (BAUMANN et al., 2009) to keep track of permissions on pages.

Device System controls the access to memory and port mapped devices, and provides mechanisms to forward the implementation of rich device drivers to user space.

Kernel Call Interface isolates the microkernel internal structures through a rich system call interface. This interface implements the Master-Slave OS model and decides whether or not a kernel call should be executed locally or remotely.

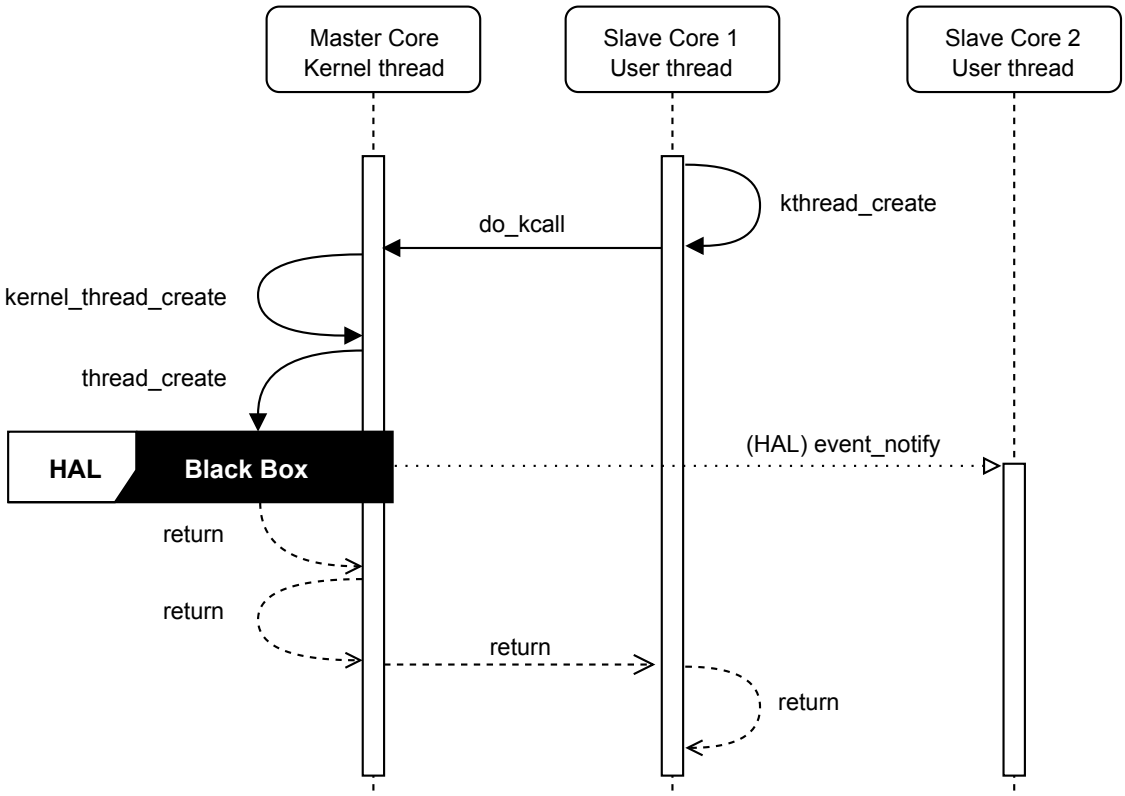
Figure 16 details how thread creation happens on Nanvix Microkernel. When a user thread makes a kernel call, a set of sanity checks are performed to ensure the integrity of the operation. After verifying the correctness of the parameters, the user thread notifies

Figure 15 – Concept Structural overview of the Nanvix Microkernel.



Source: Adapted from Penna et al. (2019)

Figure 16 – Execution example of the Nanvix Microkernel.



Source: Developed by the author.

the kernel thread and waits locked. The kernel thread implements a producer/consumer policy and executes requests one at a time. When consuming a request, the master identifies it and performs the necessary operations. In the end, the kernel thread releases the requesting slave. Note that the master core always performs all operations that change internal OS structures. Kernel calls that only query static parameters or structures (performing cache invalidation) are made locally.

2.4.3 Nanvix Multikernel

The *Nanvix Multikernel* follows a multikernel design, where OS services are scattered across clusters and interact with user processes through a Client-Server model. Ideally, OS services run in isolation from user processes. Via a message-passing approach, user processes request such services and cooperate with other processes.

Research that explores the parallel and distributed nature of multicore and many-core processors inspires the Nanvix Multikernel (WENTZLAFF; AGARWAL, 2009; BAUMANN et al., 2009; WISNIEWSKI et al., 2014). By treating the processor as a network of independent cores, it is possible to cover concepts of modern distributed systems. At this level of abstraction, the following distributed system concepts can be exploited:

Local Automation: Each node can be independent of the other nodes, providing security, locking, access, integrity, and failover mechanisms.

Replication Independence: Servers and data can be replicated across multiple nodes, where servers can service nearby nodes to decrease network traffic and perform a consistency and synchronization algorithm to ensure access to updated data.

Non-dependency on a central node: When replicating servers, the system no longer relies on a central node. It eliminates the risk of having a single point of failure that would affect all other nodes. A central node could also become overloaded, resulting in loss of system performance.

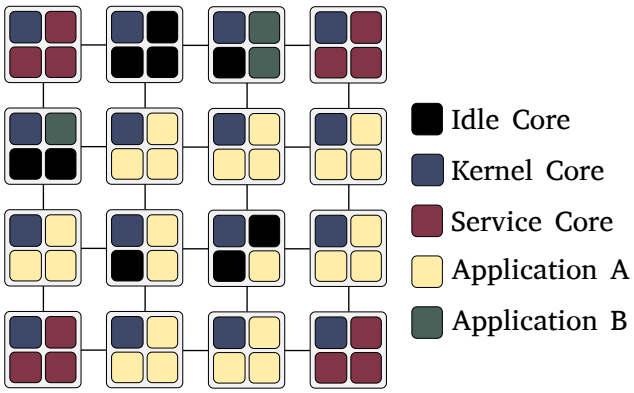
Location Transparency: Users should not need to know where the servers or data are located. For instance, a name server can control name resolution.

Figure 17 shows a possible configuration of a manycore using the Nanvix Multikernel. Clusters located at the corners of the processor run kernel services to better serve a nearby subset of clusters. In the other clusters, we can see the distribution of two distinct applications. An application does not necessarily need to use all cores in a cluster. Now, looking carefully at a cluster, we can see that Nanvix Microkernel always reserves a single core for kernel execution, making the other cores available for application execution.

To enable Nanvix OS compliance with POSIX, the Nanvix Multikernel is made up of high-level OS services that implement standardized interfaces. It exports both the client and server interfaces developed atop the Nanvix Microkernel services. From this perspective, user applications are easily ported to run on Nanvix Multikernel, where client-side interfaces abstract the communication with servers distributed on the processor.

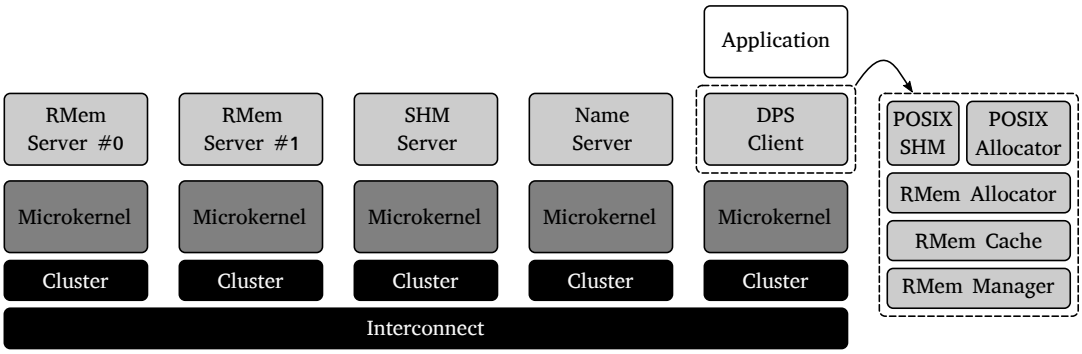
Figure 18 exemplifies the existing layers to provide the POSIX Shared Memory (SHM) service. When opening an SHM, the client does not have to worry about where data will be allocated or whether other cores in other clusters are operating over the same memory region. The SHM is divided into two blocks, the server block (SHM server) and

Figure 17 – Possible configurations on Nanvix Multikernel.



Source: Adapted from Penna et al. (2019)

Figure 18 – POSIX Compliance example of Nanvix Multikernel.



Source: Adapted from Penna et al. (2018) and Penna et al. (2019)

the client block (POSIX SHM). The server block runs in a cluster in isolation, receiving client requests, and triggering updates to shared regions in the clusters. The client block contains several interfaces for requesting services. The SHM is developed over the Remote Memory (RMem) service, which actually performed the memory upgrade. The name server maintains a mapping of server names to node IDs to provide server location independence. All servers and clients run on one instance of the Nanvix Microkernel, e.g., requesting communication services. Client interfaces, following the POSIX standard, abstract the control, location, and manipulation of distributed memory. Consequently, we provide greater programmability and software portability for manycores processors.

3 RELATED WORK

In this chapter, we discuss research efforts related to this undergraduate dissertation. First, we present an overview of the state-of-the-art on lightweight manycores. Then, due to the lack of OSs focused on lightweight manycores, we will cover different proposed OSs for manycores in general.

3.1 LIGHTWEIGHT MANYCORE PROCESSORS

Several research initiatives are focused on the design of lightweight manycores, aside from the Kalray MPPA-256 lightweight manycore. For instance, Olofsson, Nordstrom & Ul-Abdin (2014) introduce Adapteva Epiphany as a high-performance energy-efficient manycore architecture suitable for real-time embedded systems. The processor features multiple nodes interconnected through three 2D mesh NoCs with a distributed shared-memory model without coherence protocol. Each node has one Reduced Instruction Set Computer (RISC) CPU, multi-banked local memory, a DMA engine, an event monitor and a network interface. The three NoCs are independent, scalable, and implement a packet-switched model with four duplex links at every node.

Wallentowitz et al. (2013) presents the open-source OpTiMSoC framework to aid manycore processor design. The OpTiMSoC enables the rapid prototyping of a manycore, either via VHDL simulation or Field Programmable Gate Array (FPGA) synthesis. In this architecture, several OpenRISC cores[1] are bundled into tiles, which in turn communicate through a packet-switched NoC. The NoC supports various network topologies, depending only on how the tiles are disposed on chip. Precisely, a *network adapter* handles the memory transfers between a tile and its local memory and provides a message-passing communication model among tiles. Tiles can communicate by using message-exchange, partitioned global address space without cache coherence, or global memory with cache coherence via a write-through policy.

Similarly, Kurth et al. (2017) introduces HERO, which groups an ARM Cortex-A host processor with a fully modifiable RISC-V manycore implemented on an FPGA. The Programmable Manycore Accelerator (PMCA) uses a multi-cluster design and relies on multi-banked memory, called Software-managed Scratchpad Memory (SPM). Data transfer occurs between a local SPM and all remote SPMs or with shared global memory. Communication to main memory passes through software-managed lightweight Remapping Address Block (RAB). The RAB performs the translation of the virtual-to-physical address, similarly to traditional MMU, allowing clusters to share virtual address pointers.

[1] https://opencores.org/openrisc

3.2 OPERATING SYSTEMS FOR MANYCORES

Baumann et al. (2009) proposed a new OS design for scalable multicore systems, called Multikernel. In their perspective, the next-generation of OSs should embrace the networked nature of the machines, and thus borrow design ideas from large-scale distributed systems. Assuming that cores are independent nodes of a network, they build traditional OS functionalities as fully-featured processes on user space. These processes communicate via message-passing and do not share the internal structures of the OS. The work showed how expensive it is to maintain a state of the OS through shared-memory instead of exchanging messages and the subsequent increase of the complexity of cache-coherence protocols. The Multikernel implementation, named Barelfish, follows three design principles. The first principle is to *make all inter-core communication explicit*, which turns the system amenable to human or automated analysis because processes communicate only through well-defined interfaces. The second principle is to *make OS structures hardware-neutral*, so the kernel code can be easy to debug and optimize, facilitating the deployment of OS for new processor types and avoiding rework. The third principle is to *view OS state as replicated instead of shared*, which improves system scalability.

In Wisniewski et al. (2014), the concept of scalability was pushed to the extreme, towards High-Performance Computing (HPC). The principal motivation is the creation of an OS that simultaneously supports programmability through support GNU/Linux Application Programming Interface (API), and provides a lightweight kernel that achieves performance, scalability, and reliability. The OS, named multi Operating System (mOS), provides as much of the hardware resources as possible to the HPC applications and the GNU/Linux kernel component acts as a service that provides GNU/Linux functionalities.

Similarly, Kluge, Gerdes & Ungerer (2014) developed the Manycore Operating System for Safety-Critical Application (MOOSCA). With MOOSCA, they introduce abstractions that are easily composed, called Nodes, Channels, and Servers. Nodes represent execution resources, Channels represent communication resources, e.g., NoC resources, and lastly, Servers are nodes that provide services to client Nodes. To meet safety-critical requirements, the manycore is partitioned and each partition runs replicas of Servers, turning the whole system more predictable. However, in order to deal with interferences in shared resources, usage policies were introduced to make possible the prediction of system behavior.

Finally, Nightingale et al. (2009) presents Helios, aiming to simplify the task of writing, deploying, and optimizing an application across heterogeneous cores. They use the microkernel model, naming *satellite kernel*, to export a uniform and straightforward set of OS abstractions. The most important design decisions were to avoid unnecessary remote communication by thinking about the penalty they have in NUMA domains. Moreover, it requests a minimum set hardware resources to support architectures with little computational power or memory constraints.

3.3 DISCUSSION

In Section 3.1, we discussed how manycore architectures can be grouped over a common logic perspective. They all have one or more logical units distributed and incorporated on clusters. The clusters, interconnected through a network, communicate by message-exchange. However, due to the domain for which these processors were designed, they end up presenting several differences among them at the hardware level.

Additionally, Section 3.2 presented OSs studies that focus on the most efficient exploration of manycores processor characteristics. Many of them introduce entirely new concepts, reducing the programmability and portability of development environments. Some even seek to provide POSIX interfaces by porting an adapted version of known kernels, but this can lead to optimization losses at near-hardware levels. However, the OS and communication models presented fit well with the distributed nature of manycores. Finally, many of these OSs work with NUMA systems, where a complex bus system makes communication transparent. Therefore, there is, in fact, no network programming.

In this context, Nanvix HAL offers the ground OS abstractions needed to make lightweight manycores more easy to use and program. The exported interfaces sought to group lightweight manycores on a common and effective view. Above the Nanvix HAL , services will be developed that seek first and foremost to provide greater programmability and portability through a fully-featured POSIX-compliant OS. Nanvix OS is design specifically for manycores that require explicit programming of communication through NoC, have memory constraints and miss support for cache coherency. The combination of these features makes designing an OS for lightweight manycores challenging.

4 DEVELOPMENT

This work delivers two main contributions. Section 4.1 presents the most extensive part of the work and discusses how Kalray MPPA-256 hardware was used to provide three communication abstractions for *Nanvix HAL* and the challenges and solutions encountered during development. Next, Section 4.2 presents a standard view of the user services developed for the *Nanvix Microkernel* that make use of the low-level interface.

4.1 LOW-LEVEL COMMUNICATION

Nanvix HAL provides the inter-cluster communication module to allow separate clusters to exchange information. This module consists of three abstractions, named *Sync*, *Mailbox*, and *Portal*. These abstractions provide more precise, easy-to-use, scalable, and easily portable mechanisms for different architectures (WENTZLAFF et al., 2011). On top of them, it is possible to create simple facilities, such as those for synchronization and data exchange, as well as more elaborate services like SHM, POSIX Semaphores, and RMem (PENNA et al., 2018). These later semantics may be implemented on top of synchronous or asynchronous kernel calls, depending only on hardware support. Motivated to expose better Quality of Service (QoS) control to the upper layers, we decouple small data transfers from large ones, i.e., *Mailbox* and *Portal*. Note that it would be possible to use a NoC for each abstraction if the hardware supported it. We did not do this in Kalray MPPA-256 because C-NoC only provides the transfer of 64-bit values.

This section is organized as follows. Subsection 4.1.1 clarifies the use of Kalray MPPA-256 hardware resources. Subsection 4.1.2 covers commonalities between all abstractions. Subsection 4.1.3, Subsection 4.1.4, and Subsection 4.1.5 conceptually present each of the abstractions, encompassed problems, and implementation details.

4.1.1 Kalray MPPA-256 Hardware Resources

The realization of low-level communication mainly depends on two Kalray MPPA-256 features. First, the interrupt system allows the configuration of handlers for messages received and sent through NoC. Interrupts enable asynchronous operations, which is an important point in a microkernel-based OS where the master cannot be blocked waiting for a single communication to complete. If it were not possible to asynchronously receive data/signals, upper layers would face severe performance issues. Second, DMA is the mediator of all communications, either synchronous or asynchronous. At this point, a *hypervisor* virtualises the DMA, separating it into two global logical structures, one for C-NoC and one for D-NoC. Each structure groups registers for the send/receive configuration, and bit fields indicating which slots generated an interrupt. The *hypervisor* runs on RMs and asynchronously controls read/write permissions of virtualised registers.

The control made by the *hypervisor* does not include allocation or manipulation of resources. Consequently, we manually control allocation through bit fields. If an operation does not comply with this control, a negative value is returned, indicating the error, e.g., allocation of a resource that is already in use. Thus, we do not inflict undesired costs by shifting responsibility for handling the error to the upper layer, e.g., waiting for the release of a resource. Manipulation involves procedures for configuring the DMA with proper permissions and ensuring cache consistency of operations. Finally, it is noteworthy that there is no concurrency control over the commented structures so as not to inflict a cost over the microkernel approach. If the Nanvix HAL is used to develop a monolithic OS, it should be concerned with ensuring atomic operations.

The DMA coordinates three interruption lines. Two of these are used by D-NoC to notify the data receipt and completion of an asynchronous data sending. C-NoC only uses one line for receiving signals because sending a signal (64-bit value) is explicitly performed by the core. For each interrupt line, there is a specific handler, but they all execute a similar algorithm. Algorithm 1 outlines the behavior of the NoC handler. Because the line only notifies what type of interruption, it is responsibility of the handler to search the resources of each interface looking for who triggered the interrupt. The handlers are re-entrant to avoid loss of interrupts. Due to the asymmetric microkernel design, concurrency issues are softened because only the master handles interrupts.

NoC interfaces have two identifiers, one physical (physical ID) and the other logical (logical ID). The hardware uses the physical IDs in the process of data routing through NoC. Each physical ID is associated with a Logical ID to enable the identification of NoC nodes outside the HAL. Logical IDs primarily serve to disassociate the node identification from the architecture that implements the HAL. Table 1 presents the proposed mapping for Kalray MPPA-256 clusters. Each row has one of three groups of existing NoC nodes (first column), grouped by type of the clusters. Within each group, we have a set of physical identifiers (second column) that are mapped 1 to 1 to logical identifiers (third column). For example, the I/O Cluster 0 constitutes 4 NoC interfaces, where they are mapped as follows: $128 \rightarrow 0$, $129 \rightarrow 1$, $130 \rightarrow 2$, and $131 \rightarrow 3$. This mapping is a more

Algorithm 1 – Simplified NoC handler algorithm.

Require: $flags[M_{Interfaces}][N_{Resources}]$, Interruption flags of a resource.
Require: $handlers[M_{Interfaces}][N_{Resources}]$, Interruption handler of a resource.

```
1: procedure NOC_HANDLER
2:     for i ∈ [1, M_Interfaces] do
3:         for j ∈ [1, N_Resources] do
4:             if flags[i][j] == Interrupt_Triggered then
5:                 CLEAN_FLAGS(i, j)          ▷ Releases to receive new interrupts.
6:                 handlers[i][j](i, j)              ▷ Runs the associated handler.
```

Source: Developed by the author.

Table 1 – NoC Interface Identification.

	Physical ID	**Logical ID**
I/O Cluster 0	128-131	0-3
I/O Cluster 1	192-195	4-7
Compute Clusters	0-15	8-23

Source: Developed by the author.

Table 2 – Partitions of NoC resources by abstraction.

	C-NoC		**D-NoC**	
	RX Slot ID	**TX Channel ID**	**RX Slot ID**	**TX Channel ID**
Mailbox	0-23	0	0-23	1-3
Portal	24-47	1-2	24-47	4-7
Sync	48-71	3	-	-

Source: Developed by the author.

natural form to identify the NoC nodes of different architectures.

The sender needs to know which logical ID the receiver will use to perform a communication. For instance, if the receiver sets up receiving on one D-NoC resource and the sender sends it to another, even if the ID is correct, the DMA will not notify the receiver. For this reason, the receive slots of the C-NoC and D-NoC are partitioned by abstraction. Within a partition, each slot is statically mapped to a logical ID, e.g., 0 to 23 mappings. In contrast, transfer channels can be dynamically allocated. However, an essential concept of Nanvix HAL is to provide the resources required for a given operation without performing additional optimizations. Thus, the transfer channels were also partitioned by abstraction so that they are reserved for an entire job. Table 2 presents each abstraction, identified by the rows, and the partitioning of the resources of each NoC, identified by the columns. Since *Sync* does not exchange arbitrary data, no resources were reserved in D-NoC. It is worth noting that even without using all existing slots, improving programmability by not having to specify communication features is a strength of abstraction design.

Finally, we were able to remove almost all dependency on the software stack provided by Kalray. However, by eliminating communication libraries, the lack of documentation of hardware virtualisation, and functional examples of communication, limitations arose in the transfer of data through D-NoC. In summary, it was not possible to correctly configure existing μthreads in the DMA for asynchronous transfer, thus making it a future work. However, this limitation did not impact on the design of abstractions, requiring only that the master core waste time transferring data manually.

4.1.2 General Concepts of Communication Abstractions

Technically, Nanvix HAL is not just designed for use by a microkernel OS. Thus, we must ensure a standard behavior that does not affect the functionality of the upper layer, whether it is a replicated, microkernel or shared OS. The microkernel specifically assigns the master core the task of handling requests (i.e., kernel calls) from slave cores. Therefore, interfaces export only asynchronous calls. This decision forces the upper layer, if desired, to create synchronous calls that call the wait function right after the asynchronous operation. Thus, at the microkernel level, we can ensure that the master core sets or executes asynchronous functions and notifies blocked slaves. In Kalray MPPA-256, spinlocks perform synchronization between the master and the slaves. Upon completion of an operation, the master releases the lock for the slave to continue its execution.

However, the limitation of the DMA, described in Subsection 4.1.1, adds a challenge to the transfer operations of the mailbox and portal abstractions. In these abstractions, there is a flow control where the receiver must notify the sender, permitting him to transfer the data. This behavior can cause the master to lock while waiting for permission. The concept of lazy sending was introduced to circumvent this problem. Algorithm 2 illustrates the behavior of lazy transfer, the algorithm defines that the master saves the parameters of transmission if it does not have desired permission and will accomplish other requests. Upon receiving permission from the receiver, the interrupt handler identifies the resource, actually sends the data and releases the slave that requested the send. This algorithm ensures that the master is always doing useful operations and never crashes the entire system.

Finally, abstract interfaces follow a convention to distinguish between receiver and sender roles. Receivers use functions with `create`, `unlink`, `aread`, and `wait` suffixes. Senders, in turn, use functions with `open`, `close`, `awrite/signal`, and `wait` suffixes. Because the `wait` function is shared, the abstraction must distinguish the role by the resource identifier. Discriminating the nature of operations helps both the user, being entirely intuitive, and implementing HAL by explaining what features will be needed.

4.1.3 Sync Abstraction

Synchronization Abstraction, called *Sync*, provides the basis for cluster synchronization across distributed barriers. Its behavior is analogous to POSIX Signals abstraction, but notifications do not carry information, they are only for synchronization. *Sync* defines two synchronization modes, `ALL_TO_ONE` and `ONE_TO_ALL`. In both modes, there is a single master node (`ONE`) and one or more slave nodes (`ALL`) involved in the communication. Figure 19(a) illustrates the `ALL_TO_ONE` mode, where the master node waits blocked for the N notifications coming from the slaves. In contrast, Figure 19(b) pictures the `ONE_TO_ALL` mode, where the master notifies the N slaves,

releasing them from the lock. The sender nodes are responsible for sending a signal and will never block. Receiver nodes are responsible for waiting for all notifications to arrive.

Algorithm 2 – Simplified lazy transfer algorithm.

Require: *resources*, Abstraction Resource Table

Configures data transfer.

```
1:  procedure ASYNC_WRITE(id, message, size)
2:      resources[id].message ← message
3:      resources[id].size ← size
4:      if resources[id].has_permission then
5:          DO_LAZY_TRANSFER(id)
6:      else
7:          resources[id].is_waiting ← True
```

Receives permission.

```
8:  procedure ABSTRACTION_HANDLER(id)
9:      if resources[id].is_waiting then
10:         DO_LAZY_TRANSFER(id)
11:     else
12:         resources[id].has_permission ← True
```

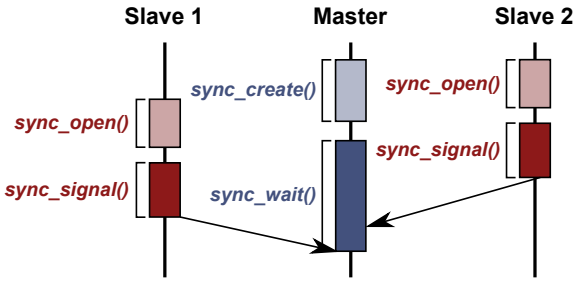
Transfers the data.

```
13: procedure DO_LAZY_TRANSFER(id)
14:     resources[id].is_waiting ← False
15:     resources[id].has_permission ← False
16:     TRANSFER_DATA(resources[id].message, resources[id].size)
17:     UNLOCK(resources[id].lock)                ▷ Releases slave core.
```

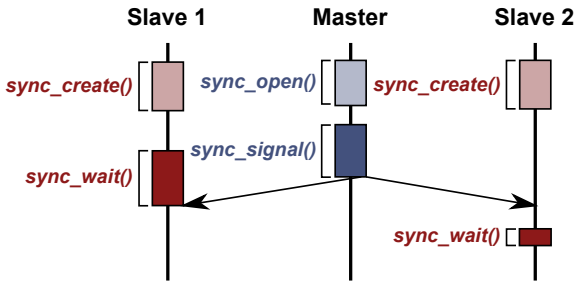
Source: Developed by the author.

Figure 19 – Synchronization abstraction example.

(a) N slaves notify the master (ALL_TO_ONE).



(b) The master notifies N slaves (ONE_TO_ALL).



Source: Developed by the author.

4.1.3.1 Receiver-side Implementation

Listing 1 introduces the *Sync* interface for receiver nodes proposed for the *Nanvix HAL*. The parameters required for creating a synchronization point are a list of logical IDs, list size, and mode of synchronization. The list must always be initialized with the master node ID, regardless of the mode. The remaining identifiers, provided they have no repetition, can be in any order. The other functions use the abstraction identifier returned by the create function. If any parameters are invalid or have wrong semantics, a negative value is returned indicating the error, e.g., nodes pointer equal to `NULL`.

Receipt of notifications requires booking one C-NoC receiving slot related to the Master ID. This relation eliminates the conflict of using the same slot across different synchronization settings. Consequently, a node cannot be the master in two simultaneous operations. Thus, the total of *Sync* operations created simultaneously is equal to the number of existing nodes, i.e., 24 in Kalray MPPA-256. In I/O Clusters, this total is multiplied by the number of available NoC interfaces, i.e., 24 per DMA. A 64-bit mask, created from sender node IDs, configures the receiving slot. Bits positioned on nodes IDs are 0. When receiving a signal, DMA performs an *OR-bitwise* with the previous value. When all bits are set to 1, DMA clears the register and triggers an interrupt.

A vector of internal structures controls the operations. Each structure is reserved for a physical slot and holds control flags, the initial mask, and a spinlock. HAL checks for discrepancies in IDs, control flags, or parameters when creating, waiting, or unlinking a *Sync*. Finally, the spinlock is used to synchronize the operation with slave cores. On our microkernel-based system, the master core configures *Sync*, and the slave waits for the spinlock release. The interrupt handler of the *Sync* identifies the structure and releases the lock. The lock of cache coherence does not affect spinlocks because instructions that guarantee the atomicity are used to implement the lock.

Listing 1 – Nanvix HAL: Sync interface for receiver node.

```
1  /**
2   * @brief Allocates and configures the receiving side of
3   * the synchronization point.
4   */
5  int sync_create(const int *nodes, int nnodes, int mode);
6
7  /* @brief Releases and cleans receiver slot. */
8  int sync_unlink(int syncid);
9
10 /* @brief Waits a signal. */
11 int sync_wait(int syncid);
```

Source: Developed by the author.

4.1.3.2 Sender-side Implementation

The *Sync* interface for sender nodes, presented in Listing 2, uses the same create parameters for opening a *Sync* point. The standardization of parameters simplifies the role of a cluster in a synchronization. However, at both creation and opening, the local ID must be included in the list. For instance, a problem occurs if a *Sync* is opened with the local ID as master and the mode is set to `ALL_TO_ONE`. This discrepancy will return an error because the master should be the notification receiver and not the sender. The rest of the implementation follows what was already explained in the previous section.

Unlike the receiver, the sender needs one C-NoC transfer channel to open a *Sync* point. Due to the separation of channels described in the Table 2, a node can only open one *Sync* at a time. The node must identify the target ID and receiving slot of the master to emit a signal. A 64-bit value composes the mask with the sender node bit set to 1. The sender control structure also has flags to ensure the semantics of the operations. Besides, the sender stores, in an array of integers, all IDs of the receivers. When performing the notification, a signal will be sent to each target of this list.

Listing 2 – Nanvix HAL: Sync interface for sender node.

```
1  /**
2   * @brief Allocates and configures the sending side of
3   * the synchronization point.
4   */
5  int sync_open(const int *nodes, int nnodes, int mode);
6
7  /* @brief Releases the transfer channel. */
8  int sync_close(int syncid);
9
10 /* @brief Sends a signal. */
11 int sync_signal(int syncid);
```

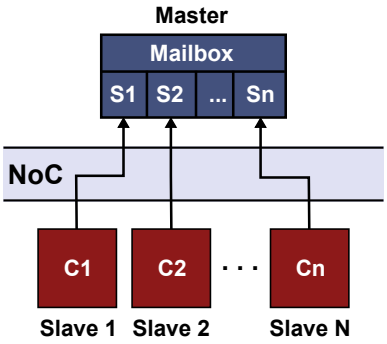
Source: Developed by the author.

4.1.4 Mailbox Abstraction

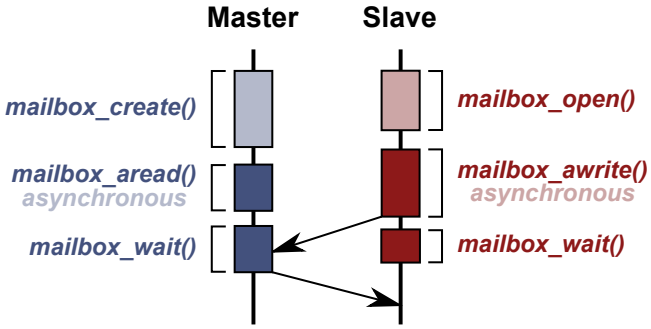
Message Queue Abstraction, called *Mailbox*, allows clusters to exchange fixed-length messages with each other. The message size is designed to be relatively small, usually a few hundreds of bytes. The recipient consumes these messages without needing to know who sent them. Similarly, mailbox operations follow the behavior of the POSIX message queue. Figure 20(a) conceptually illustrates one of the ways to implement a *Mailbox*. The receiver allocates enough space to receive one message from each possible sender. The sender transfers the message to a predefined location.

Figure 20 – Mailbox abstraction concept.

(a) Conceptual overview.



(b) Flow of execution: the slave sends a message whereas the master reads and notifies the sender.



Source: Developed by the author.

Figure 20(b) outlines the communication flow between a receiver and a sender node. The receiver creates an empty message queue where senders are free to send the first message. Subsequently, to avoid overwriting old messages, all transmissions require the permission of the receiver. For this reason, when the receiver consumes a message, it copies the message to the user buffer, releases the queue space, and notifies the sender.

In theory, the number of messages allowed per sender can be from 1 to N. However, Nanvix HAL statically allocates sufficient memory for the message queue inside the kernel. Therefore, we chose to allow only one message due to memory constraints presented by lightweight manycores. Furthermore, using one message is sufficient for servers to handle requests at the Nanvix Multikernel level. For instance, the message can be used to encode small operations and system control signals.

4.1.4.1 Receiver-side Implementation

Listing 3 presents the Mailbox interface for receiver nodes. Since the I/O Cluster has multiple nodes, it is necessary to inform the local node ID on the creation of the *Mailbox* (`mailbox_create`). The other operations use the file descriptor returned by `mailbox_create`. To consume a message (`mailbox_aread` function), the application must inform a `buffer` and a message `size`. Although the `size` is constant, it is used to verify the integrity of the operation. Successful copying of a message will release the slave that performs the `mailbox_wait` function. In the release protocol, the master core flushes the message to the SRAM so that the slave, when invalidating its cache, can read the message.

Mailbox is more complicated than *Sync*, in terms of hardware resources. Specifically, the receiver requires one D-NoC receiving slot and one C-NoC transfer channel. The need for one transfer channel for the lifetime of the receiver limits the creation of only one mailbox per NoC node. The receiving slot is configured using two sizes. One for the size

Listing 3 – Nanvix HAL: mailbox interface for receiver node.

```
1  /* @brief Creates a mailbox. */
2  int mailbox_create(int nodenum);
3
4  /* @brief Destroys a mailbox. */
5  int mailbox_unlink(int mbxid);
6
7  /* @brief Reads data asynchronously from a mailbox. */
8  ssize_t mailbox_aread(int mbxid, void * buffer, size_t size);
9
10 /* @brief Waits for an asynchronous operation to complete. */
11 int mailbox_wait(int mbxid);
```

Source: Developed by the author.

of a message, which will generate interrupts, and one for the total buffer size allocated for protection. The buffer is allocated within kernel memory with sufficient space to receive 24 messages. The message itself is composed of a header identifying the sender, a body containing the useful message, and a footer for handler control. The transfer channel, in turn, is used after copying the useful message to the user buffer, notifying the header ID.

Parallelism in receiving messages introduced some challenges in the asynchronous reading of Mailbox because: (i) each message generates an interrupt, (ii) interrupts suspended by new incoming messages may not be able to find the D-NoC resources that issued the interrupt, and (iii) the hardware does not report the total interrupts generated by a resource. A behavior similar to lazy transfer was implemented to circumvent these difficulties. First, a global counter containing the total amount of messages received allows the receiver to copy received messages on the asynchronous read call. Second, if no messages is available, copying will be performed by the next triggered handler. Third, to solve the problem of preemptive interrupts, whenever a handler is triggered, it will traverse the message queue checking headers and footers. When identifying a valid message, the handler increments the counter and changes the footer to a specific code. This way, no message is lost because a single handler identifies messages not yet counted.

4.1.4.2 Sender-side Implementation

Listing 4 displays the Mailbox interface for sender nodes. Opening a mailbox requires the application to inform the receiver ID to allocate related resources. The `mailbox_awrite` function, in particular, implements the concept of lazy transfer. So, the master core never blocks if the mailbox is not allowed to transfer the message. The `mailbox_wait` function is the same function used by the receiver node where the slave core waits blocked until receiving the transfer permission.

The sender implementation also requires the allocation of resources from both

Listing 4 – Nanvix HAL: Mailbox interface for sender node.

```
1  /* @brief Opens a mailbox. */
2  int mailbox_open(int nodenum);
3
4  /* @brief Closes a mailbox. */
5  int mailbox_close(int mbxid);
6
7  /* @brief Writes data asynchronously to a mailbox. */
8  ssize_t mailbox_awrite(int mbxid, const void * buffer, size_t size);
9
10 /* @brief Waits for an asynchronous operation to complete. */
11 int mailbox_wait(int mbxid);
```

Source: Developed by the author.

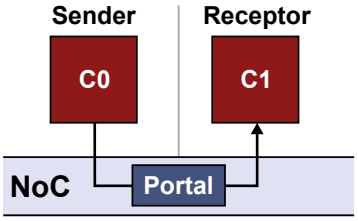
NoCs. However, the resources are opposite to the receiver, where it allocates one C-NoC receiving slot and one D-NoC transfer channel. The allocated C-NoC receiving slot is relative to the receiver ID to prevent conflicts among openings for distinct receivers. The transfer channel is dynamically allocated. Because it has fewer transfer channels than receiving slots, transfer channels limit mailbox openings to 4 per node.

4.1.5 Portal Abstraction

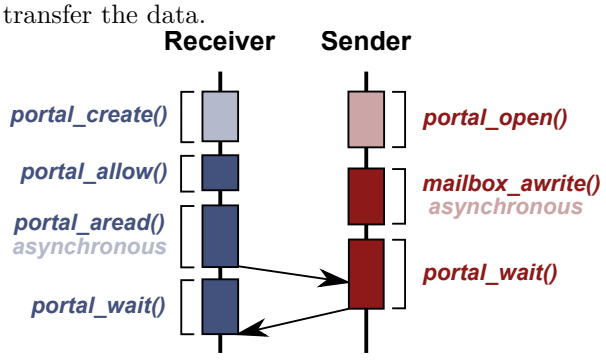
Finally, *Portal Abstraction* allows two nodes to exchange arbitrary amounts of data. Figure 21(a) presents the conceptual idea of the *Portal*, which resembles that of POSIX Pipes with flow control. The cardinality of operations is always 1 : 1, where a pair of nodes opens a one-way channel for data transfer. However, unlike POSIX Pipes, which defines that the pipe exists only between two processes, the *Portal* allows the receiver to communicate with other nodes through the same channel. The sender, in turn, can only communicate with one node.

Figure 21 – Portal abstraction concept.

(a) Conceptual Overview.



(b) Receiver creates a portal and notifies Sender to transfer the data.



Source: Developed by the author.

Figure 21(b) outlines the flow control of the *Portal*. When attempting to transmit data to the receiver, the sender will block until the receiver can accept it. By enabling one data exchange at a time, the receiver configures the read settings and notifies the sender. In this way, the flow control ensures that the receiver will not be overloaded, will not receive data without properly configured DMA, and will not overwrite previous data. Allowing communication empowers the receiver to choose which communications to prioritize.

4.1.5.1 Receiver-side Implementation

Listing 5 presents the Portal interface for receiver nodes. Like the *Mailbox*, the application must identify the local node to create a *Portal*. The receiver must always perform three operations to perform communication, i.e., `portal_allow`, `portal_aread`, and `portal_wait`. The `portal_allow` function allocates one receiving slot relative to the given remote node, limiting one channel per pair of nodes. However, the permission will only be sent after DMA configuration by the `portal_aread` function. Finally, the node will block in the `portal_wait` function until receiving the informed data size.

A receiver allocates one D-NoC receiving slot and one C-NoC transfer channel. Unlike *Mailbox*, the *Portal* has two transfer channels available, which allows the creation of two simultaneous portals. Such portals cannot communicate with the same node at the same time because they use the same physical slot. Read sets the receiving slot to the buffer and size informed by the application. The DMA eliminates intermediate copies like Mailbox, because it writes data directly to the application buffer. Consequently, control structures have also been simplified, containing only control flags and the spinlock for asynchronous operations.

Listing 5 – Nanvix HAL: Portal interface for receiver node.

```
1  /* @brief Creates a portal. */
2  int portal_create(int local);
3
4  /* @brief Destroys a portal. */
5  int portal_unlink(int portalid);
6
7  /* @brief Allow sender to transfer data. */
8  int portal_allow(int portalid, int remote);
9
10 /* @brief Reads data asynchronously from a portal. */
11 ssize_t portal_aread(int portalid, void * buffer, size_t size);
12
13 /* @brief Waits for an asynchronous operation to complete. */
14 int portal_wait(int portalid);
```

Source: Developed by the author.

4.1.5.2 Sender-side Implementation

Listing 6 presents the Portal interface for sender nodes. When opening a *Portal*, the application is required to inform the local node ID and the receiver node ID. The local ID serves to distinguish the NoC interface on I/O Clusters. The receiver node ID identifies the C-NoC receiving slot that will catch the transfer permission. The early allocation ensures that the permission will not be lost even if the permission arrives before the sender sets up the transfer. The transfer configuration follows the lazy transfer algorithm.

The hardware resources required to open a portal are the opposite of resources to create. Specifically, one C-NoC receiving slot and one D-NoC transfer channel are required. The transfer channel is reserved but only used when the transfer is allowed. Due to the limitation of four transfer channels to the portal, a node can open only four portals at a time. The control structures for the sender portals contain the parameters needed to perform the lazy transfer and a spinlock to asynchronous operations.

Listing 6 – Nanvix HAL: Portal interface for sender node.

```
1  /* @brief Opens a portal. */
2  int portal_open(int local, int remote);
3
4  /* @brief Closes a portal. */
5  int portal_close(int portalid);
6
7  /* @brief Writes data asynchronously to a portal. */
8  int portal_awrite(int portalid, const void * buffer, size_t size);
9
10 /* @brief Waits for an asynchronous operation to complete. */
11 int portal_wait(int portalid);
```

Source: Developed by the author.

4.2 USER-LEVEL COMMUNICATION

The inter-cluster communication module, described in Section 4.1, is designed to export a standard and straightforward communication primitives to different lightweight manycores. These primitives can be used by various types of OSs and applications. Thus, the module is flexible enough not to impact the performance of the upper layers negatively. For this, it does not provide rich management of the exposed abstractions.

In this scenario, the communication services of Nanvix Microkernel seek to provide Inter-Process Communication (IPC) between distinct clusters. Specifically, these services perform the multiplexing of the hardware resources and the verification of the parameters that will be passed on the communication primitives. Due to the Master-Slave

model, the master core is responsible for protecting, manipulating, and configuring HAL resources. The slave core will request operations through the kernel call interface, passing the necessary information to the master.

Considering that the abstractions make up the fundamental elements of the construction of more complex services, the Nanvix Microkernel services were responsible for the management and multiplexing of the finite resources for the many cores of a cluster. In total, there are three communication services in the Nanvix Microkernel, each associated with an abstraction of the communication module, analogously named *Sync*, *Mailbox*, and *Portal* services. These services must take into account the memory constraints and the Master-Slave model chosen for the Nanvix Microkernel.

The impacts of the Master-Slave model, protection, management and manipulation operations are similar to all services. They will be provided through interfaces that function as wrappers for the HAL abstraction functions. In the implementation of these interfaces, there will be a mapping between low-level identifiers, associated with HAL resources, and high-level identifiers, associated with resource protection structures.

The following sections provide an overview of these topics punctuating the differences of each service.

4.2.1 Impacts of the Master-Slave Model

Master-Slave OS model defines that the master core must exclusively manipulate the internal structures of the OS. For this end, each service has generic and simple structures that hold control flags, parameters for resource identification, and storage of physical descriptors returned by HAL. Thus, the kernel call interface separates the functions of the services into two sets of functions. The first set contains the kernel calls that request a particular operation. The second set contains functions that operate on the internal structures and communicate with the HAL level. The master executes almost exclusively the second set, except for the wait functions, which the slave performs entirely.

The kernel call set follows the `kernel_`*`abstraction_operation`* notation. Its responsibilities are to find out which logical ID is attached to the local node, if there is more than one, and perform the possible sanity checks on the slave side. The internal set of functions follows the `k`*`abstraction_operation`* notation. Internally, it is used by the kernel call engine to perform protection, handling, and multiplexing services.

The requirement of the slave to perform wait functions is because the master cannot block indefinitely waiting for an operation. At this point, the slave must access the internal structure of the OS to query the physical descriptor for access to the HAL spinlock. The spinlock is used to wait for an asynchronous operation to complete. So, the master updates local memory whenever a modification is made, and the slave invalidates its cache to ensure structure coherence.

4.2.2 Protection and Management

Protection and management operations involve two phases, the slave and the master phase. The slave phase performs several checks to identify all possible problems with the arguments before wasting master time with an invalid operation. Examples of theses verifications are: (i) valid file descriptors, (ii) non-null buffer pointers, (iii) buffer sizes within the stipulated limit, and (iv) the semantics of the arguments, e.g., synchronization with itself.

The master phase performs more robust checks than those performed by the slave phase, where it is possible to verify the semantics of operations on a given resource. Specifically, the master: (i) identifies multiple creations/openings (ii) checks for conflicting operations, e.g., writing to read-only resources, (iii) measures communication time and total number of bytes transmitted/received, and (iv) interacts with Nanvix HAL detecting errors.

4.2.3 Multiplexing

The identification of creations/openings with the same arguments allows multiple slaves to use the same resource at different times. For this, the internal structures of the OS keep, as simply as possible, the arguments used to create/open a service. When the same arguments are identified, a reference counter is incremented. The master sets the resource to busy when prompted for a read/write. A second slave who wants to read/write will be prevented until the previous operation is completed. The resources of the HAL will only be released when all references are removed.

4.2.4 Input/Output Control

The *Mailbox* and *Portal* services have a particular kernel call named Input/Output Control (IOCTL). The IOCTL grants the implementation of operations that cannot be expressed by regular kernel calls. In the case of communication services, we implement two types of operations to query some information about the transmissions performed. The first operation, named `IOCTL_GET_VOLUME`, queries the current number of bytes transmitted/received from a service. The second operation, named `IOCTL_GET_LATENCY`, queries the sum of measured communication latencies by the difference of two clock readings. This kernel call may be extended in the future to introduce new features without causing changes to current interfaces.

4.2.5 Validation and Correctness Tests

Extra effort has been made to ensure that deployment across the multiple supported architectures exhibits the same expected behavior. To this end, two sets of tests ensure the validation and correctness of the implementation, named *API* and *FAULT* tests. On the one hand, *API* tests create, open, and stimulate services with arguments within valid value ranges and the correct semantics of the operations. On the other hand, *FAULT* tests use arguments outside the domain of functions and incorrect semantics of operations. The failure of an operation also should generate a previously known error value. These sets of tests sought to cover the most common and predictable errors that occur when using communication services. However, a robust coverage test should be performed to ensure that the tests verify all possible errors.

5 EXPERIMENTS

This chapter evaluates the performance of communication services of the Nanvix Microkernel running on the Kalray MPPA-256 processor, i.e., *Mailbox* and *Portal*. The impacts of the synchronization mechanism were not analyzed because it is a simple service that does not directly influence node communication, depending greatly on the workload of each cluster. Noteworthy, the *Sync* was used in all benchmarks to synchronize the nodes involved due to the different boot times and the distinct node roles. The evaluation is divided into two sections. First, Section 5.1 describes the micro-benchmarks, their motivations, and the parameters used. Second, Section 5.2 unveil and discusses our experimental results.

5.1 EVALUATION METHODOLOGY

To deliver a comprehensive assessment of the communication service, we stimulate the services with usual collective communication configurations. These configurations are usually found in distributed systems and present in the high-level services exported by Nanvix Multikernel, such as message exchanging between servers and clients, work distribution, and gathering results.

Micro-benchmarks measure the data volume and communication latency through the IOCTL interface. In manycores, the nodes that communicate with peripherals are the bridge between the user and applications. Therefore, in our experiments, I/O Cluster plays the master role when a communication routine requires a master-slave behavior. I/O Clusters also manages only one of the available interfaces to simplify communication. In all micro-benchmarks, only one PE was used to request microkernel services.

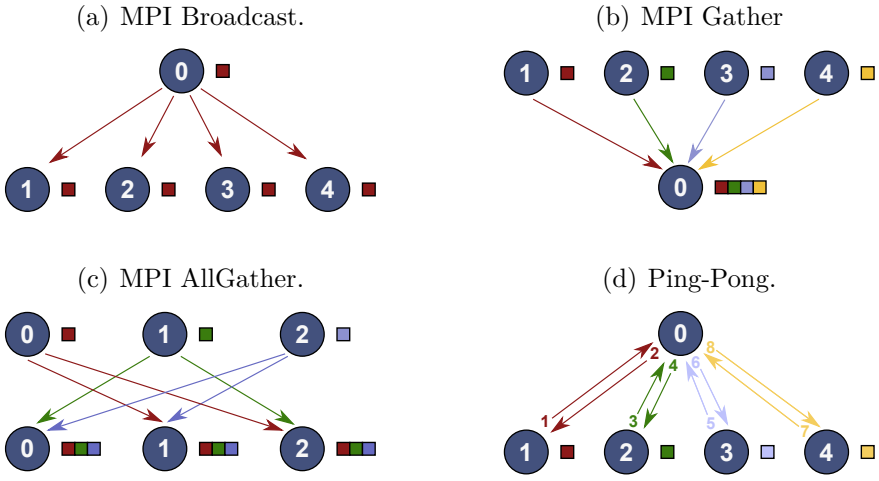
5.1.1 Micro-benchmarks

To analyze the performance of the communication services, we relied in collective communication patters of MPI, as well as common behaviors between clients and servers. The following subsections conceptually introduce each of these routines and behaviors.

5.1.1.1 Broadcast

Broadcast is the most widely used communication pattern in MPI. In this routine, a node sends the same data to all existing nodes. This process may be implemented in several ways, such as, Flat Tree, Binary Tree, Double Tree, and Chain (WICKRAMASINGHE; LUMSDAINE, 2016). Figure 22(a) presents the *Flat Tree* algorithm used in the benchmark. The Flat Tree defines that the root node should send data to everyone without delegating this function to other nodes. This routine can be used to send user

Figure 22 – Collective Communication Routines.



Source: Adapted from Kendall, Nath & Bland (2019).

inputs to a parallel program or to send configuration parameters to all nodes (KENDALL; NATH; BLAND, 2019).

5.1.1.2 Gather

Gather is the inverse operation of a broadcast variant called scatter. Figure 22(b) illustrates the reverse data flow, where this routine gathers the data distributed on a single node (KENDALL; NATH; BLAND, 2019). Similarly to broadcast, a Flat Tree was implemented where all root nodes send their parts directly to the root node.

5.1.1.3 AllGather

AllGather is a routine that does not have a root node, illustrated by Figure 22(c). As the name suggests, the routine performs several Gather operations so that all participating nodes end with all pieces of data gathered. Some possible algorithms are Ring Algorithm, Recursive Doubling, Gather followed by Broadcast Algorithm. The benchmark implements the *Bruck Algorithm* where each node will send its data to a node with distance i and receive data from a distance $-i$ until all nodes contain the complete data.

5.1.1.4 Ping-Pong

Ping-Pong is not an MPI collective communication routine but represents communication from a server answering requests from client nodes. Figure 22(d) illustrates communication by focusing on the master node, where the master receives and answers one request at a time.

5.1.2 Experimental Design

The parameters that we used for each micro-benchmark are detailed in Table 3. The first set of experiments sought to analyze the throughput provided by the Portal service. Throughput displays the performance of the Portal in communicating different amounts of data and thereby highlighting the best amounts. All micro-benchmarks involve 1 I/O Cluster and 16 Compute Clusters, varying the size of the buffer to be transmitted from 4 KB to 64 KB. Larger values were not studied due to limitation on physical memory size in Compute Clusters (i.e., 2 MB). For instance, AllGather requires approximately a total space of 1 MB (17 nodes $\times$ 64 KB). The second set aimed to analyze the latency of the Mailbox service. Latency allows us to analyze the communication time between different components of a distributed system. The micro-benchmarks executed were practically the same as the Portal. However, the buffer size to be transmitted became constant, 120 Bytes. The variable parameter of the experiments was the number of Compute Clusters involved in the routines. Thus, I/O Cluster is always the master of routines, and the number of Compute Cluster is changed between 1 and 16.

The Kalray MPPA-256 is designed to provide low energy per operation and time predictability which guarantees low variability between runs (DINECHIN et al., 2013). Thus, 50 iterations of each benchmark were performed. For each experiment, the first ten iterations were discarded to eliminate undesired warm-up effects. Finally, all results discussed bellow present a standard error inferior to 1%.

Table 3 – Micro-benchmark parameters for experiments.

	Portal		**Mailbox**	
	#Clusters	**Data Size**	**#Clusters**	**Data Size**
Broadcast	1 IO, 16 CC	4, 8, 16, 32, 64 KB	1 IO, 1 to 16 CC	120 B
Gather	1 IO, 16 CC	4, 8, 16, 32, 64 KB	1 IO, 1 to 16 CC	120 B
AllGather	1 IO, 16 CC	4, 8, 16, 32, 64 KB	1 IO, 1 to 16 CC	120 B
Ping-Pong	1 IO, 16 CC	4, 8, 16, 32, 64 KB	1 IO, 1 to 16 CC	120 B

Source: Developed by the author.

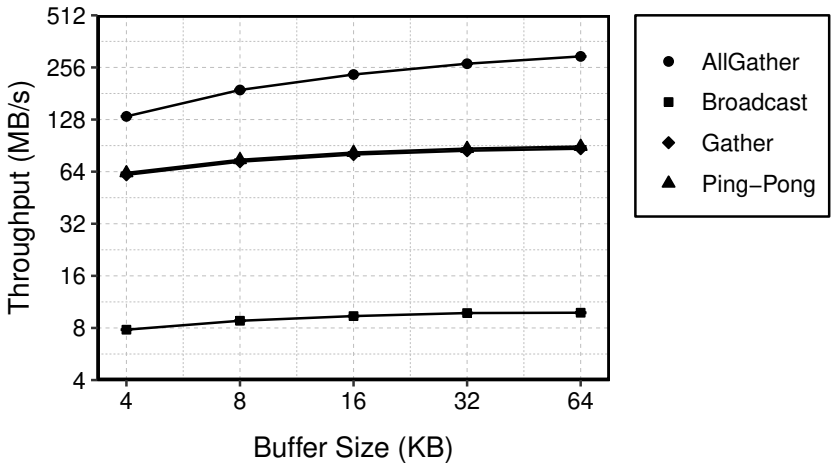
5.2 EXPERIMENTAL RESULTS

In this section, we present our experimental results. First, we analyze outcomes for *Portal service*, and next we move to a discussion on the results for the *Mailbox service*.

5.2.1 Portal Throughput Analysis

Figure 23 presents the throughput of the Portal in MB/s relative to the different amounts of data transmitted. Results exhibit three distinct behaviors in the experiments.

Figure 23 – Throughput of the Portal.



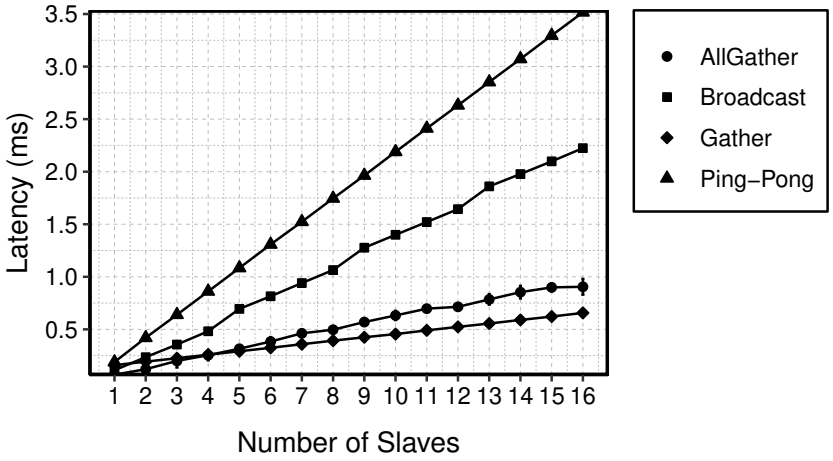
Source: Develop by the Author.

First, the Broadcast was expected to have the worst transmission rate due to the use of a single data transmitter. Since the measurement was done on the receiver side, the last slave had to wait for the master to transmit data to all other nodes, considerably reducing the transfer rate in the Broadcast. Second, the Gather and Ping-Pong routines exhibited similar results, overlapping each other on the chart. This similarity is because the master node receives multiple requests and handles them serially one by one. The master node dictated the data flow in both benchmarks because transmission is only performed when allowed by the receiver. Finally, the AllGather routine exhibited the best results because of the parallelism of communications. Each communication pair is done in parallel and multiple read/write requests do not happen at the same time on a node, softening the interruption of the master core. In the context of OSs, we have subsystems requiring large data transfers, such as file and paging systems. In this case, observing the slope of the lines, we can infer that the 8 KB and 16 KB sizes favor Portal throughput. Overall, the results were as expected, but we believe that solving the problem using DMA accelerators described in Subsection 4.1.1 could significantly improve Portal throughput.

5.2.2 Mailbox Latency Analysis

Figure 24 presents the latency of the Mailbox in milliseconds relative to the number of clusters involved. First, Gather routine, one of the essential routines, had the best results because receiving the messages occurs in parallel. Thus, the cost after the first message is the overhead of the kernel call itself, not the communication. Second, the AllGather routine exhibited similar behavior to Gather because sending messages occurs before the reading begins. So, when the clusters start reading, the latest messages are already coming. The Broadcast routine also suffers from the same problem as the Portal routine, because there is only one node sending the messages. Finally, the overload of

Figure 24 – Latency of the Mailbox.



Source: Develop by the Author.

sending messages to requesters caused the linear behavior of the Ping-Pong routine. In this case, Ping-Pong has a slightly higher cost than the sum of Gather and Broadcast costs, because, despite the benefits of receiving requests in parallel, the master spends most of its time handling requests sequentially. Overall, Broadcast, Gather, and AllGather showed how message-passing distributed algorithms can be efficiently supported in Nanvix Multikernel. Ping-Pong, in turn, highlights the latency of communication between Nanvix subsystems with remote kernels, encouraging the potential for improvement using DMA accelerators.

6 CONCLUSIONS

Initially, this work presented a historical evolution of processors from single core to manycore. By demonstrating the relationship between the growth of the number of cores and energy consumption, it was discussed how academia and industry began to develop alternatives to alleviate the technological barriers that have emerged. However, even new processors that emerge and stand out because of their performance and power consumption lack on programmability and portability, because of their architectural features, such as hybrid programming model, constrained memory subsystems, no cache coherency, and heterogeneous configurations. Part of the difficulty stems from the incompleteness of existing OSs and runtimes in dealing with severe architectural constraints.

In this work, we present an inter-cluster communication facility designed around the main points in the development of an OS for *lightweight manycores*. As a basis, we discussed hardware and software aspects of parallel and distributed architectures. Different models of OS approaches that can use the communication facility have been presented. Thus, to provide the essential functionalities for such OSs, three communication abstractions have been proposed for the Nanvix HAL with the concern of providing QoS: *Sync*, useful to create distributed barriers; *Mailbox*, which provides the exchange of small messages with flow control; and *Portal*, which allows the exchange of arbitrary amounts of data between two clusters.

Another contribution of this work was the communication services for an OS based on the microkernel approach (*Nanvix*). These services can multiplex the resources exposed by HAL and perform the verification of the parameters for each abstraction. In general, these services securely export the communication abstractions to the user, benefiting from the non-competition of OS internal structures, because of the separation of master and slave responsibilities.

The contributions of this dissertation are included in the investment of a distributed operating system. Several works are underway in this context, including a full port of the MPI and implementation of a distributed paging system that will rely on the proposed abstractions. As future work at the HAL level, we will seek to properly utilize existing DMA μthreads to perform asynchronous submissions and, consequently, to increase the performance of the abstractions. At the microkernel level, we will study the virtualization of the structures of each service, and improve multiplexing algorithms, enriching the programmability of applications. The results present how well-known distributed algorithms can be efficiently supported by Nanvix OS and encourage improvements provided by the proper use of DMA accelerators.

BIBLIOGRAPHY

BARBALACE, A. et al. Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms. In: **Proceedings of the 10th European Conference on Computer Systems**. Bordeaux, France: ACM, 2015. (EuroSys '15), p. 1–16. ISBN 978-1-4503-3238-5. Disponível em: http://dl.acm.org/citation.cfm?doid=2741948.2741962.

BAUMANN, A. et al. The multikernel: A new OS architecture for scalable multicore systems. In: **SOSP '09 Proceedings of the 22nd ACM Symposium on Operating Systems Principles**. ACM, 2009. (SOSP '09), p. 29–44. ISBN 978-1-60558-752-3. Disponível em: https://dl.acm.org/citation.cfm?doid=1629575.1629579.

CASTRO, M. et al. Seismic wave propagation simulations on low-power and performance-centric manycores. **Parallel Computing**, v. 54, p. 108–120, 2016. ISSN 01678191.

CHRISTGAU, S.; SCHNOR, B. Exploring One-Sided Communication and Synchronization on a Non-Cache-Coherent Many-Core Architecture. **Concurrency and Computation: Practice and Experience (CCPE)**, v. 29, n. 15, p. e4113, mar. 2017. ISSN 1532-0626. Disponível em: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4113.

DINECHIN, B. D. de et al. A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. In: **Procedia Computer Science**. Barcelona, Spain: Elsevier, 2013. (ICCS '13, v. 18), p. 1654–1663. ISBN 1877-0509. Disponível em: http://linkinghub.elsevier.com/retrieve/pii/S1877050913004766.

FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE Trans. Comput.**, IEEE Computer Society, Washington, DC, USA, v. 21, n. 9, p. 948–960, set. 1972. ISSN 0018-9340. Disponível em: http://dx.doi.org/10.1109/TC.1972.5009071.

FRANCESQUINI, E. et al. On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms. **Journal of Parallel and Distributed Computing (JPDC)**, v. 76, n. C, p. 32–48, fev. 2015. ISSN 0743-7315. Disponível em: http://linkinghub.elsevier.com/retrieve/pii/S0743731514002093.

FREITAS, H. C. de. **Arquitetura de NoC Programável Baseada em Múltiplos Clusters de Cores para Suporte e Padrões de Comunicação Coletiva**. Tese (Doutorado) — Programa de Pós-Graduação em Computação, UFRGS, Porto Alegre, 6 2009. An optional note.

GAMELL, M. et al. Exploring Cross-Layer Power Management for PGAS Applications on the SCC Platform. In: **Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing**. Delft, The Netherlands: ACM, 2012. (HPDC '12), p. 235–246. ISBN 978-1-4503-0805-2. Disponível em: http://dl.acm.org/citation.cfm?doid=2287076.2287113.

KELLY, B.; GARDNER, W.; KYO, S. AutoPilot: Message Passing Parallel Programming for a Cache Incoherent Embedded Manycore Processor. In: **Proceedings of the 1st International Workshop on Many-core Embedded Systems**. Tel-Aviv, Israel: ACM, 2013. (MES '13), p. 62–65. ISBN 978-1-4503-2063-4. Disponível em: http://dl.acm.org/citation.cfm?doid=2489068.2491624.

KENDALL, W.; NATH, D.; BLAND, W. **A Comprehensive MPI Tutorial Resource**. 2019. https://mpitutorial.com/, Last accessed on 2019-10-22.

KLUGE, F.; GERDES, M.; UNGERER, T. An Operating System for Safety-Critical Applications on Manycore Processors. In: **2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing**. Reno, Nevada, USA: IEEE, 2014. (ISORC '14), p. 238–245. ISBN 978-1-4799-4430-9. Disponível em: http://ieeexplore.ieee.org/document/6899155/.

KOGGE, P. et al. Exascale computing study: Technology challenges in achieving exascale systems. **Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Techinal Representative**, v. 15, 01 2008.

KURTH, A. et al. Hero: Heterogeneous embedded research platform for exploring risc-v manycore accelerators on fpga. In: **Proceedings of Computer Architecture Research with RISC-V Workshop (CARRV' 17)**. [S.l.: s.n.], 2017. First Workshop on Computer Architecture Research with RISC-V (CARRV 2017); Conference Location: Boston, MA, USA; Conference Date: October 14, 2017.

MOORE, G. E. Cramming more components onto integrated circuits. **Electronics**, v. 38, n. 8, April 1965.

NEUMANN, J. v. **First Draft of a Report on the EDVAC**. [S.l.], 1945.

NIGHTINGALE, E. et al. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In: **Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles**. Big Sky, Montana, USA: ACM, 2009. (SOSP '09), p. 221–234. ISBN 978-1-60558-752-3. Disponível em: http://doi.acm.org/10.1145/1629575.1629597.

OLOFSSON, A.; NORDSTROM, T.; UL-ABDIN, Z. Kickstarting High-Performance Energy-Efficient Manycore Architectures with Epiphany. In: **2014 48th Asilomar Conference on Signals, Systems and Computers**. Pacific Grove, California, USA: IEEE, 2014. (ACSSC '14), p. 1719–1726. ISBN 978-1-4799-8297-4. Disponível em: http://ieeexplore.ieee.org/document/7094761/.

PENNA, P. H. et al. Using the Nanvix Operating System in Undergraduate Operating System Courses. In: **2017 VII Brazilian Symposium on Computing Systems Engineering**. Curitiba, Brazil: IEEE, 2017. (SBESC '17), p. 193–198. ISBN 978-1-5386-3590-2. Disponível em: http://ieeexplore.ieee.org/document/8116579/.

PENNA, P. H.; FRANCIS, D.; SOUTO, J. The Hardware Abstraction Layer of Nanvix for the Kalray MPPA-256 Lightweight Manycore Processor. In: **Conférence d'Informatique en Parallélisme, Architecture et Système**. Anglet, France: [s.n.], 2019. Disponível em: https://hal.archives-ouvertes.fr/hal-02151274.

PENNA, P. H. et al. Using The Nanvix Operating System in Undergraduate Operating System Courses. In: **VII Brazilian Symposium on Computing Systems Engineering**. Curitiba, Brazil: [s.n.], 2017. Disponível em: https://hal.archives-ouvertes.fr/hal-01635880.

PENNA, P. H. et al. On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores. In: **SBESC 2019 - IX Brazilian Symposium**

on Computing Systems Engineering. Natal, Brazil: [s.n.], 2019. Disponível em: https://hal.archives-ouvertes.fr/hal-02297637.

PENNA, P. H. et al. An Operating System Service for Remote Memory Accesses in Low-Power NoC-Based Manycores. n. October, 2018.

PENNA, P. H. et al. RMem: An OS Service for Transparent Remote Memory Access in Lightweight Manycores. In: **MultiProg 2019 - 25th International Workshop on Programmability and Architectures for Heterogeneous Multicores**. Valencia, Spain: [s.n.], 2019. (High-Performance and Embedded Architectures and Compilers Workshops (HiPEAC Workshops)), p. 1–16. Disponível em: https://hal.archives-ouvertes.fr/hal-01986366.

PENNA, P. H. et al. An os service for transparent remote memory accesses in noc-based lightweight manycores. Poster. 2018.

RHODEN, B. et al. Improving Per-Node Efficiency in the Datacenter with New OS Abstractions. In: **Proceedings of the 2nd ACM Symposium on Cloud Computing**. Cascais, Portugal: ACM, 2011. (SoCC '11), p. 1–8. ISBN 978-1-4503-0976-9. Disponível em: https://dl.acm.org/citation.cfm?id=2038941.

RUPP, K. **Microprocessor Trend Data**. 2018. https://github.com/karlrupp/microprocessor-trend-data, Last accessed on 2019-06-26.

SERRES, O. et al. Experiences with UPC on TILE-64 Processor. In: **Aerospace Conference**. Big Sky, Montana, USA: IEEE, 2011. (AERO '11), p. 1–9. ISBN 978-1-4244-7350-2. Disponível em: http://ieeexplore.ieee.org/document/5747452/.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. 9th. ed. [S.l.]: Wiley Publishing, 2012. ISBN 1118063333, 9781118063330.

TANENBAUM, A. S.; BOS, H. **Modern Operating Systems**. 4th. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. ISBN 013359162X, 9780133591620.

WALLENTOWITZ, S. et al. **Open Tiled Manycore System-on-Chip**. [S.l.], 2013. 1–7 p. Disponível em: http://arxiv.org/abs/1304.5081.

WENTZLAFF, D.; AGARWAL, A. Factored Operating Systems (FOS): The Case for a Scalable Operating System for Multicores. **ACM SIGOPS Operating Systems Review**, v. 43, n. 2, p. 76–85, abr. 2009. ISSN 0163-5980. Disponível em: https://dl.acm.org/citation.cfm?doid=1531793.1531805.

WENTZLAFF, D. et al. **Fleets: Scalable Services in a Factored Operating System**. 2011. 1–13 p. Disponível em: https://dspace.mit.edu/handle/1721.1/61640.

WICKRAMASINGHE, U.; LUMSDAINE, A. A survey of methods for collective communication optimization and tuning. **CoRR**, abs/1611.06334, 2016. Disponível em: http://arxiv.org/abs/1611.06334.

WIKIPEDIA. **Flynn's taxonomy**. 2019. https://en.wikipedia.org/wiki/Flynn%27s_taxonomy, Last accessed on 2019-06-30.

WISNIEWSKI, R. et al. mOS: An Architecture for Extreme-Scale Operating Systems. In: **ROSS '14 Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers**. Munich, Germany: ACM, 2014. (ROSS '14), p. 1–8. ISBN 978-1-4503-2950-7. Disponível em: http://dl.acm.org/citation.cfm?doid=2612262.2612263.

ZHENG, F. et al. Cooperative Computing Techniques for a Deeply Fused and Heterogeneous Many-Core Processor Architecture. **Journal of Computer Science and Technology (JCST)**, v. 30, n. 1, p. 145–162, jan. 2015. ISSN 1000-9000. Disponível em: https://link.springer.com/article/10.1007%2Fs11390-015-1510-9.

APPENDIX A – SCIENTIFIC ARTICLE

Mecanismos de Comunicação entre *Clusters* para *Lightweight Manycores* no Nanvix OS

João Vicente Souto[1], Pedro H. Penna[2], Márcio Castro[1], Henrique Freitas[3]

[1] Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD)
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, Brasil

[2]Laboratoire d'Informatique de Grenoble (LIG)
Université Grenoble Alpes (UGA) – Grenoble, França

[3]Grupo de Arquitetura de Computadores e Processamento Paralelo (CArT)
Pontifícia Universidade Católica de Minas Gerais (PUC Minas) – Belo Horizonte, Brasil

joao.vicente.souto@grad.ufsc.br,pedro.penna@univ-grenoble-alpes.fr,
marcio.castro@ufsc.br,cota@pucminas.br

Abstract. *Development environments for lightweight manycores lack to provide a good relationship between programmability and portability. In this context, this work proposes mechanisms of communication between clusters for a distributed operating system that are accurate, easy-to-use, scalable, and easily portable. The results show that it is possible to support collective communication algorithms efficiently.*

Resumo. *Ambientes de desenvolvimento para lightweight manycores pecam em prover uma boa relação entre programabilidade e portabilidade. Neste contexto, este artigo propõe mecanismos de comunicação entre clusters para um sistema operacional distribuído que sejam precisos, fáceis de usar, escalonáveis e facilmente portáveis. Os resultados mostram ser possível suportar algoritmos de comunicação colectiva de forma eficientemente.*

1. Introdução

A próxima grande barreira de desempenho dos sistemas computacionais modernos será proveniente da relação entre poder de processamento e consumo energético [Kogge et al. 2008]. Neste âmbito, processadores *lightweight manycore* surgiram para prover alto nível de paralelismo com baixo consumo energético. Entretanto, o desenvolvimento de aplicações para essa classe de processadores exibe diversos desafios [Castro et al. 2016].

A Figura 1 ilustra as particularidades que diferem os *lightweight manycores* dos *multicores* e *manycores* tradicionais. Especificamente, eles: (i) integram centenas de núcleos de baixo potência agrupados em *clusters*, (ii) lidam com cargas de trabalho *Multiple Instruction Multiple Data* (MIMD), (iii) apresentam memória distribuída através de restritivas memórias locais e falta de coerência de cache, (iv) dependem de uma *Network-on-Chip* (NoC) para comunicação entre *clusters*, forçando uma programação híbrida entre memória compartilhada e troca de mensagens e (v) possuem componentes heterogêneos.

Parte dos desafios encontrados ao se trabalhar com *lightweight manycores* deriva diretamente dos *runtimes* e Sistemas Operacionais (SOs) existentes que não lidam corretamente com suas particularidades. Deste modo, eles tornam o ambiente de desenvolvimento mais oneroso e suscetível a erros. No contexto de comunicações em SOs para *lightweight manycores*, *Kluge et al.* [Kluge et al. 2014] projetaram um SO que provê

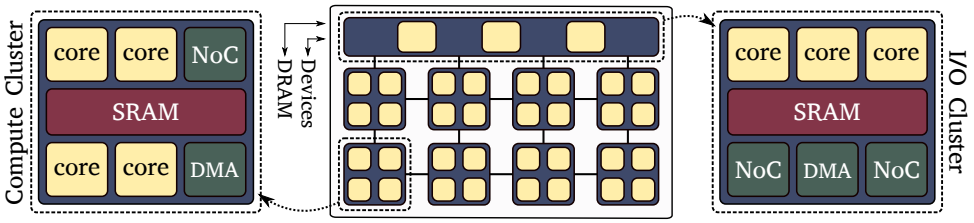


Figura 1. Visão geral de um *lightweight manycore* [Penna et al. 2019b].

canais de comunicação unidirecionais com suporte a configuração de políticas de Qualidade de Serviço (QoS). *Wentzlaff et al.* [Wentzlaff et al. 2011], por sua vez, propuseram uma abordagem de comunicação entre processos através da troca de mensagens baseado na abstração de *mailbox*. Os trabalhos mostraram que estes mecanismos básicos de comunicação podem ser facilmente portados para diferentes arquiteturas de *lightweight manycores*, além de apresentarem boa escalabilidade.

O presente trabalho se insere neste contexto de comunicação em *chip* e está incluso na pesquisa e desenvolvimento de um novo SO distribuído para *lightweight manycores*. A principal contribuição deste artigo é a proposta de mecanismos de comunicação entre *clusters* que buscam ser mais precisos, fáceis de usar, escalonáveis, e facilmente portáveis.

O restante do trabalho está organizado da seguinte maneira. A Seção 2 apresenta o Nanvix, um SO com foco em *lightweight manycores*, e o processador Kalray MPPA-256, o qual foi utilizado como caso de estudo. A Seção 3 e Seção 4 apresentam os mecanismos propostos e a sua avaliação, respectivamente. Por fim, a Seção 5 apresenta as conclusões.

2. Fundamentação Teórica

Esta seção apresenta uma breve descrição do SO e do *lightweight manycore* utilizados neste trabalho.

2.1. O Sistema Operacional Nanvix

O Nanvix[1] é um projeto de código aberto e colaborativo que atende a ausência de SOs que lidem com as particularidades dos *lightweight manycores*. Ele é um SO de propósito geral que busca uma boa relação entre programabilidade e portabilidade, além de ser compatível com o padrão *Portable Operating System Interface* (POSIX). Ele adota uma estrutura *multikernel*, onde os serviços do SO são processos que rodam isoladamente atendendo requisições de processos de usuário através da troca de mensagens. Dentro de um *cluster* é utilizado um *microkernel* assíncrono para amenizar a interferência entre os núcleos [Penna et al. 2019b]. Na camada mais baixa, o Nanvix exporta uma visão padronizada e concisa desses processadores através de uma Camada de Abstração de Hardware (HAL) [Penna et al. 2019a]. O presente trabalho está incluso nas camadas do *microkernel* e HAL.

2.2. Processador *Lightweight Manycore* MPPA-256

O Kalray MPPA-256 [de Dinechin et al. 2013] é uma das arquiteturas suportadas pelo Nanvix OS e será utilizada neste artigo como caso de estudo. Especificamente, ele integra 288 núcleos de propósito geral, agrupados em 16 Clusters de Computação (CCs), destinados a computação útil, e 4 Clusters de I/O (IOs) responsáveis pela comunicação com

[1]https://github.com/nanvix/

periféricos. Para comunicação entre *clusters*, o Kalray MPPA-256 apresenta uma NoC para dados e outra para comandos, denominadas *Data Network-on-Chip* (D-NoC) e *Control Network-on-Chip* (C-NoC), respectivamente. Cada uma das NoCs apresenta características distintas, destacando-se as diferentes quantidades de recursos de comunicação (RX e TX) e a quantidade de dados transmitidos (apenas 64 bits de cada vez pela C-NoC).

3. Proposta de Mecanismos de Comunicação entre *Clusters*

Os mecanismos de comunicação entre *clusters* propostos neste trabalho são constituídos de três abstrações: *Sync*, *Mailbox* e *Portal*. A definição dessas abstrações generalizam três comportamentos que frequentemente aparecem em sistemas distribuídos, i.e., sincronizações, trocas de mensagens de controle e grandes trocas de dados. Desta forma, é possível exportar uma visão abstrata e padronizada dos recursos de comunicação existentes nas mais diversas arquiteturas.

Primeiramente, a abstração *Sync* provê a criação de barreiras distribuídas. Seu comportamento se assemelha ao POSIX *Signals*. Existem dois modos de sincronização. Primeiro, o modo `ALL_TO_ONE` define que um *cluster* mestre deve aguardar N notificações de N escravos. Segundo, no modo `ONE_TO_ALL` o mestre notifica N escravos, liberando-os do bloqueio. A implementação no Kalray MPPA-256 utilizou apenas recursos da C-NoC, onde o principal argumento é a lista dos *clusters* envolvidos. A primeira posição nesta lista deve ser ocupada obrigatoriamente pelo mestre, a partir do qual será inferido quais recursos de *hardware* deverão ser utilizados. Deste modo, é possível abstrair o conhecimento e manipulação do *hardware* pelo usuário.

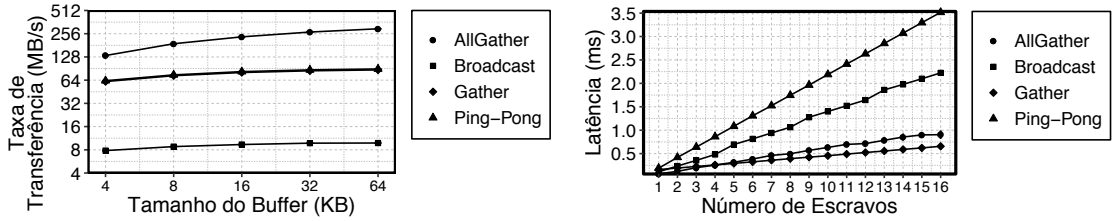
Segundo, a abstração *Mailbox* provê a troca de mensagens de tamanho fixo. Ela é similar ao POSIX *Message Queue*, onde o receptor aloca espaço suficiente para receber N mensagens. O emissor, por sua vez, envia uma mensagem para um local pré-determinado. O comportamento da *Mailbox* define um controle de fluxo permitindo a emissão de apenas uma mensagem de cada vez. Quando o receptor consumir uma mensagem, ele notificará o emissor que a enviou. A implementação utilizou recursos da C-NoC e da D-NoC. A fila de mensagens foi alocada no espaço de memória do *kernel* para abstrair o controle e manipulação das mensagens do usuário.

Por fim, a abstração *Portal* permite a troca de quantidades arbitrárias de dados entre dois *clusters*. Similar ao POSIX *Pipe*, a comunicação é unidirecional. O *Portal*, assim como a *Mailbox*, implementa um controle de fluxo para garantir a QoS. Neste controle, o emissor só enviará os dados quando o receptor estiver apto a receber. Deste modo, a implementação do *Portal*, utilizando recursos da C-NoC e D-NoC, eliminou a necessidade de cópias intermediárias, onde a comunicação é configurada com endereços de memória do próprio espaço de usuário.

4. Resultados Experimentais

Para avaliar os serviços de transferência de dados, foram elaboradas quatro rotinas de comunicação coletiva que reproduzem comportamentos existentes no Nanvix OS, i.e., *Broadcast*, *Gather*, *AllGather* e *Ping-Pong* [Wickramasinghe and Lumsdaine 2016]. Para garantir 95% de confiança, foram realizadas 50 replicações, descartando-se as primeiras 10 para ignorar o período de aquecimento e conduzindo a um erro padrão inferior a 1%.

Os experimentos do *Portal* avaliaram a taxa de transferência da comunicação para um número constante de *clusters* envolvidos (1 IO e 16 CCs), variando-se a quantidade de dados transmitidos (4 KB até 64 KB). Os resultados na Figura 2(a) exibem três comportamentos. Primeiro, devido ao gargalo do único emissor, o *Broadcast* apresentou os



(a) *Portal*. (b) *Mailbox*.

Figura 2. Resultados experimentais.

piores resultados. Segundo, o *Gather* e *Ping-Pong* mostram resultados similares porque o mestre da operação dita o fluxo das comunicações devido ao QoS. Por último, o *AllGather* obteve os melhores resultados porque se benificia do paralelismo das comunicações. No contexto do Nanvix OS, é possível inferir que os tamanhos entre 8 KB e 16 KB favorecem a taxa de transferência do *Portal* no uso pelos subsistemas do SO.

Os experimentos da *Mailbox* avaliaram a latência da comunicação. Foi mantido constante o tamanho da mensagem (120 B) e variou-se o número de *clusters* de computação envolvidos (de 1 à 16). A Figura 2(b) também apresentaram três comportamentos. Primeiro, o *Gather* e *AllGather* se benificiaram do recebimento paralelo das mensagens e obtiveram as menores latências. Segundo, o comportamento do *Broadcast* sofre do mesmo problema do *Portal*. Por último, apesar do *Ping-Pong* também se beneficiar do recebimento paralelo, o mestre deve atender sequencialmente cada uma das mensagens, por isso apresentou os piores resultados.

5. Conclusão

Neste trabalho, foi proposto mecanismos de comunicação entre *cluster* para processadores *lightweight manycores* no Nanvix OS. Os resultados mostraram como algoritmos distribuídos bem conhecidos podem ser eficientemente suportados pelo Nanvix OS. Como trabalhos futuros no contexto do Nanvix OS, pretende-se realizar o porte do *Message Passing Interface* (MPI) sobre os mecanismos de comunicação propostos.

Referências

Castro, M., Francesquini, E., Dupros, F., Aochi, H., Navaux, P. O., and Méhaut, J.-F. (2016). Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, 54:108–120.

de Dinechin, B. D., de Massas, P. G., Lager, G., Léger, C., Orgogozo, B., Reybert, J., and Strudel, T. (2013). A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. In *Procedia Computer Science*, volume 18 of *ICCS '13*, pages 1654–1663, Barcelona, Spain. Elsevier.

Kluge, F., Gerdes, M., and Ungerer, T. (2014). An Operating System for Safety-Critical Applications on Manycore Processors. In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '14, pages 238–245, Reno, Nevada, USA. IEEE.

Kogge, P., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Keckler, S., Klein, D., and Lucas, R. (2008). Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced*

Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Techinal Representative, 15.

Penna, P. H., Francis, D., and Souto, J. (2019a). The Hardware Abstraction Layer of Nanvix for the Kalray MPPA-256 Lightweight Manycore Processor. In *Conférence d'Informatique en Parallélisme, Architecture et Système*, Anglet, France.

Penna, P. H., Souto, J., Lima, D. F., Castro, M., Broquedis, F., Freitas, H., and Mehaut, J.-F. (2019b). On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores. In *SBESC 2019 - IX Brazilian Symposium on Computing Systems Engineering*, Natal, Brazil.

Wentzlaff, D., Gruenwald, C., Beckmann, N., Belay, A., Kasture, H., Modzelewski, K., Youseff, L., Miller, J., and Agarwal, A. (2011). Fleets: Scalable services in a factored operating system.

Wickramasinghe, U. and Lumsdaine, A. (2016). A survey of methods for collective communication optimization and tuning. *CoRR*, abs/1611.06334.

APPENDIX B – SOURCE CODE

B.1 NANVIX PROJECT STRUCTURE

The development of a general-purpose distributed operating system for lightweight manycores processors, called Nanvix OS, includes the source code for this undergraduate dissertation. Nanvix OS is the result of an open-source, collaborative project made available on the Github platform. Since it is not semantically interesting to have only part of the source code and it is impossible to insert all OS code into this document, this appendix details where find and test the developed code. The Nanvix Project is detailed in Section 2.4.

Specifically, there is a separate *Github* repository for each abstraction layer that is maintained and updated by Nanvix contributors. Submodules, supported by the *git* tool, create an implicit dependency hierarchy between the Nanvix repositories. Thus, each layer that depends on another has guarantees of its operation and is exempt from its implementation. Theses guarantees make the codes better manageable, modular, and better portable. All repositories contain test sets (validation and fault tests) to ensure the correct implementation of exported interfaces on the several supported architectures.

All repositories follow a branch naming convention, where only two branches are, in fact, essential. First, the master branch contains the most stable version of the system and marks the significant releases of Nanvix versions. Second, the unstable branch contains the most current version, but there may still exist bugs or required parts missing. The other branches are intended to implement new features, improve existing ones, or fix bugs. These other branches are merged into the unstable branch when passing all regression tests.

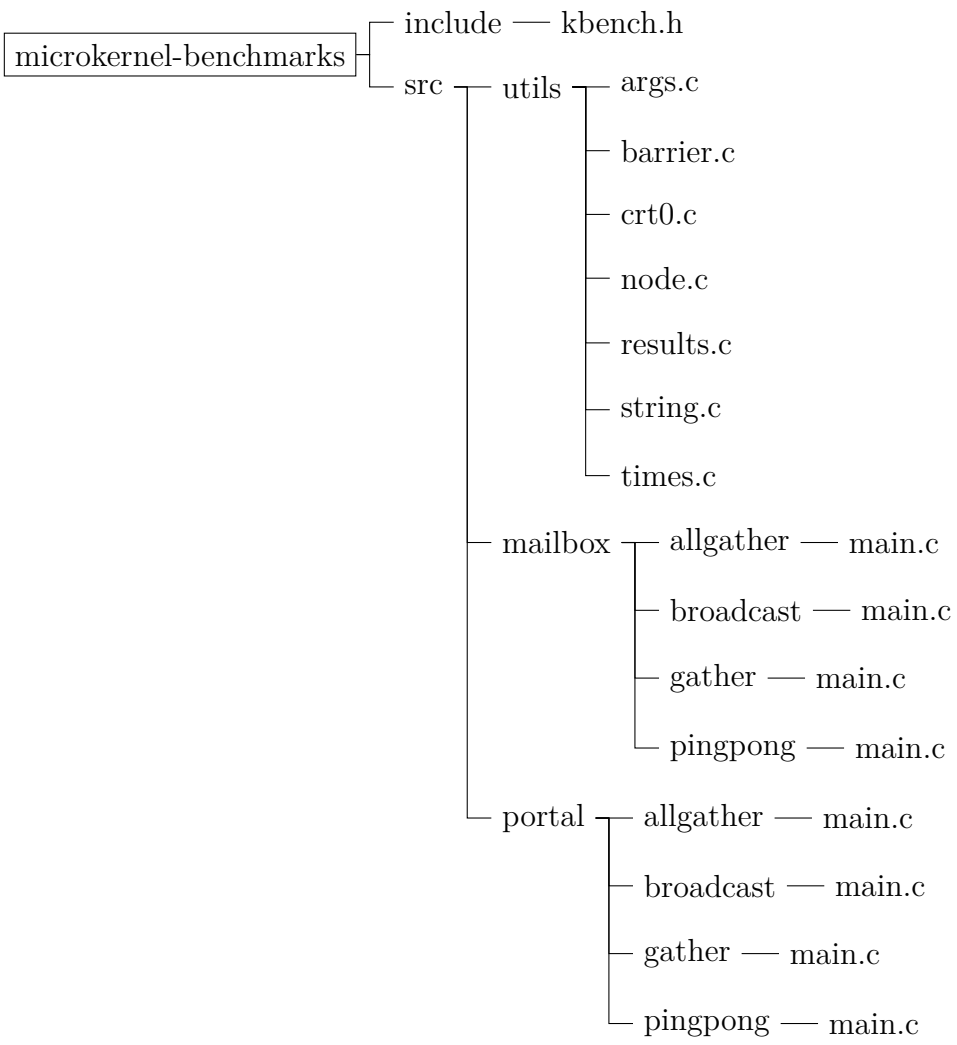
The following subsections, ordered by dependency, detail the four repositories that contain the source code of this undergraduate dissertation.

B.1.1 Microkernel-Benchmarks Repository

The *Microkernel-Benchmarks* implements the micro-benchmarks that stimulate communication services. Specifically, there are four micro-benchmarks for each data transfer service, detailed in Section 5.1. In this repository, there are also scripts for compiling and running experiments on the Kalray MPPA-256 platform.

The developed code version are available at https://github.com/joaovicentesouto/microkernel-benchmarks, in the *collective-comm-routines branch* or, specifically, in the *commit aafd9a70f8188105efabd651050bc7cafc39d343*. Figure B-1 presents, in the form of a directory tree, all files that are fully, or partially, developed. The `include` folder contains only one file with static experiment settings. The `src` folder is made up of auxiliary files and the micro-benchmarks themselves.

Figure B-1 – Directory tree with developed source codes in Microkernel-Benchmarks repository.



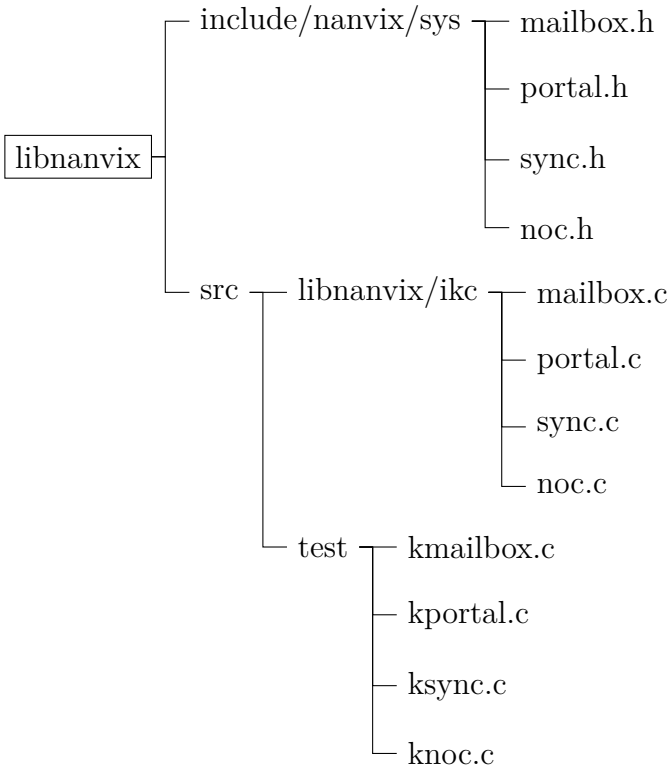
Source: Developed by the author.

B.1.2 LibNanvix Repository

LibNanvix defines and exports user interfaces for Nanvix Microkernel services. Implementations, in turn, have the responsibility of making requests to the master core through the kernel call interface exported by the microkernel repository, detailed in Subsection 2.4.2. Briefly, this repository is the complement of the Microkernel repository.

Figure B-2 illustrates, as a directory tree, all files developed of the communication abstraction user interface. First, the files in the *include* folder define the user interfaces. Second, files within the *ikc* folder perform checks to identify potential problems. Finally, the test folder contains the validation and correctness tests of user abstractions. The code version is available in *commit a9dcb35dd8727aefe41d316ac2609c88073e160e* at https://github.com/nanvix/libnanvix.

Figure B-2 – Directory tree with developed source codes in LibNanvix repository.



Source: Developed by the author.

B.1.3 Microkernel Repository

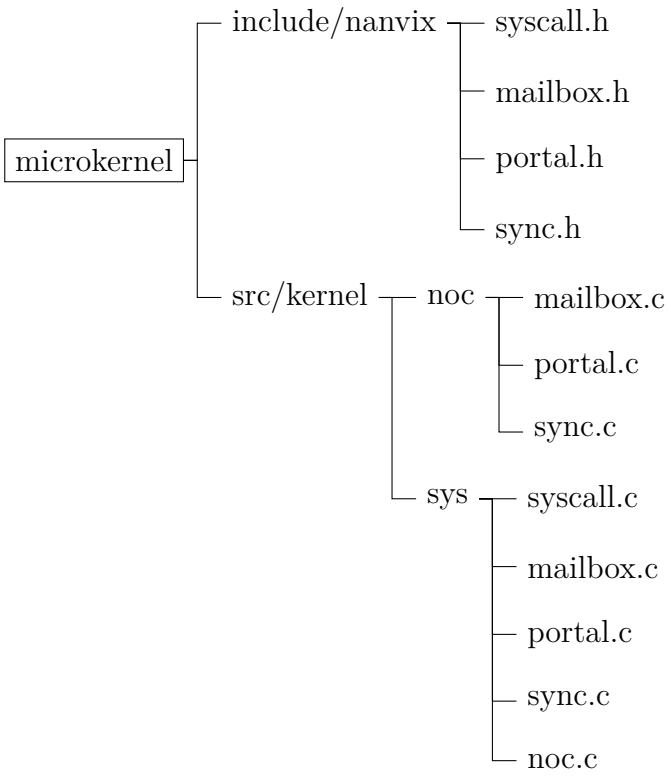
Microkernel, in the context of the asymmetric microkernel, covers the responsibilities of the master core. This repository contains all internal microkernel structures, manipulation functions, and master-side kernel calls. In association with LibNanvix, they provide the skeletons of system abstractions within the cluster of a lightweight manycores. For more details, see Subsection 2.4.2.

Figure B-3 exemplifies the three developed file sets. First, the *include* folder defines static kernel configurations; exports internal call interfaces, i.e., executed by the master; and external calls, i.e., the kernel call system. Second, the files within the *noc* folder integrate the internal structures and proper protection, manipulation, and multiplexing functions. Finally, the *sys* folder contains the implementation of kernel calls. The code version is available in *commit a9826dec62baa3fe47ab3a77b15f3ccfdd84b79a* at https://github.com/nanvix/microkernel.

B.1.4 Hardware Abstraction Layer (HAL) Repository

HAL defines and exports the lowest-level abstraction interfaces, detailed in Subsection 2.4.1. This repository deals directly with multiple supported architectures, cur-

Figure B-3 – Directory tree with developed source codes in Microkernel repository.



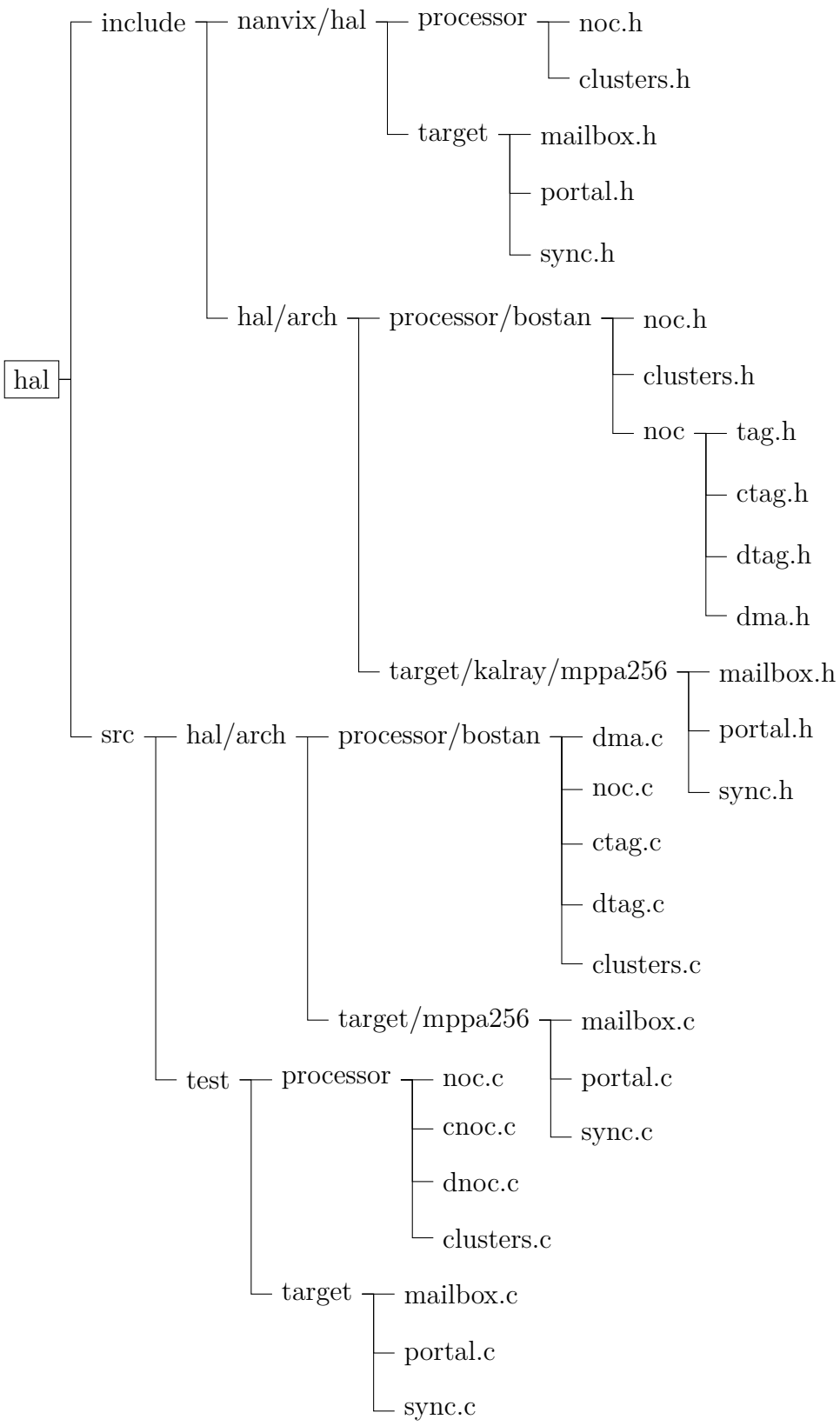
Source: Developed by the author.

rently including Kalray MPPA-256, OpTiMSoC, and HERO. An implementation on a Unix OS has also been developed to allow easy debugging of the developed systems.

Figure B-4 illustrates the logical structure designed to facilitate HAL portability. In this framework, there are levels that abstract the hardware layers from the essential element (*core*) to the level that abstracts the architecture as a whole (*target*). Expressly, this work was restricted to *processor* and *target* levels. The file hierarchy follows what is described in the previous subsections, dividing the files into interfaces and implementations. The code version is available in *commit 1e7d3bc64decff023ac91cdecc2e0ac6c53ac946* at https://github.com/nanvix/hal.

The interfaces are included in the include folder. However, there is a distinction between exported interfaces and architecture-specific interfaces. The `nanvix/hal` folder covers interfaces exported to other repositories, as well as static checking of specific interfaces. The `hal/arch` folder actually exports what architecture has hardware capabilities. If it does not implement a hardware limitation feature, a default implementation will be used. For each hardware interface defined in the `hal/arch`, there is a corresponding source file (`src` folder). Specific implementations of Kalray MPPA-256 architecture, for example, contain implementations of low-level communications, including handling and manipulation of different existing NoCs. Finally, the test folder constitutes the integration tests that validate the various implementations.

Figure B-4 – Directory tree with developed source codes in HAL repository.



Source: Developed by the author.

B.2 REGRESSION TESTING EXAMPLE

Listing B-1 exemplifies the running of user interfaces regression tests on the Kalray MPPA-256 platform. A server that contains an installed Kalray MPPA-256 card compiles and executes the tests remotely. The tests can still run locally using the *Docker* tool, where a virtual machine simulates one of the architectures supported by Nanvix OS.

Listing B-1 – Bash script for regression testing on the MPPA-256 platform.

```
#!/bin/bash
# A Shell Script To Runs Regression Tests on MPPA-256
# Nanvix Project - 2019

# Defines the target architecture.
export TARGET=mppa256

# Download the LibNanvix repository.
git clone https://github.com/nanvix/libnanvix.git

# Enter the folder containing the source code.
cd libnanvix

# Switch to the version developed in this work.
git checkout a9dcb35dd8727aefe41d316ac2609c88073e160e

# Download submodules recursively.
git submodule update --init --recursive

# Compile the dependencies.
make contrib

# Compile, link to libraries, and create the executable.
make all

# Runs the regression tests.
make test
```

Source: Developed by the author.