João Vicente Souto

An Inter-Cluster Communication Module of a Hardware Abstraction Layer for Lightweight Manycore Processors

Trabalho de Conclusão de Curso submetido ao Departamento de Informática e Estatística em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de bacharel em Ciência da Computação.
Orientador: Dr. Márcio Bastos Castro
Coorientador: Me. Pedro Henrique Penna

Florianópolis
2019

João Vicente Souto

An Inter-Cluster Communication Module of a Hardware Abstraction Layer for Lightweight Manycore Processors

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "bacharel em Ciência da Computação", e aprovado em sua forma final pelo Departamento de Informática e Estatística em Ciência da Computação.

Florianópolis, 01 de Julho de 2019.

José Francisco D. de G. C. Fletes
Coordenador do Curso

Banca Examinadora:

Dr. Márcio Bastos Castro
Orientador
Universidade Federal de Santa Catarina

Nesta etapa do projeto o documento segue este formato mesmo?

Me. Pedro Henrique Penna
Coorientador
Universite Grenoble Alpes

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

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

arrumado

O que é desempenho? Será que escalabilidade não é uma das medidas de desempenho?

RESUMO

Em conjunto com a maior escalabilidade de desempenho e eficiência energética, os *lightweight manycores* trouxeram um novo conjunto de desafios no desenvolvimento de software, provenientes de suas particularidades arquitetônicas. Nesse contexto, os sistemas operacionais são essenciais porque permitem abstrair as características do hardware sob uma perspectiva simplificada e eficaz. Assim, os sistemas operacionais tornam o desenvolvimento de aplicações menos oneroso e mais eficiente. No entanto, parte dos desafios encontrados em *lightweight manycores* deriva dos runtimes e dos sistemas operacionais existentes que não lidam completamente com as complexidades desses processadores. Assim, acreditamos que os sistemas operacionais para a próxima geração de *lightweight manycores* devem ser reprojetados "do zero" para lidar com suas severas restrições arquitetônicas. Em particular, devido à natureza distribuída dos *manycores*, as abstrações de comunicação desempenham um papel crucial na escalabilidade e desempenho das aplicações. ~~Neste cenário,~~ o objetivo deste trabalho é desenvolver um módulo de comunicação entre clusters para o processador *manycore* emergente MPPA-256. Este módulo faz parte de um Camada de Abstração de Hardwarre genérico e flexível para *lightweight manycores* que lida com os principais problemas encontrados no projeto de um sistema operacional para esses processadores. Em cima desse módulo, serviços de comunicação também serão propostos para um sistema operacional baseado no modelo *microkernel*, que busca fornecer um esqueleto básico para as abstrações de sistema. Essas contribuições estão inseridas em um contexto de pesquisa mais amplo, que procura investigar um sistema operacional distribuído completo baseado em uma abordagem multikernel para esses processadores.

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

itálico
arrumado

coloquial

Sobre

Sugestão: Pode deixar o texto mais fluido usando menos conectores: "Assim, No entanto, Neste cenário, neste contexto, etc."

ABSTRACT

Jointly with further performance 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) are essential because they allow to abstract the characteristics of the hardware under a simplified and productive perspective. Thus, OSs make application development less costly and more efficient. However, part of the challenges encountered in lightweight manycores derives from existing runtimes and OSs that do not completely handle extant intricacies. So, 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, due to the distributed nature of the manycores, communication abstractions play a crucial role in the scalability and performance of these processors. In this scenario, the goal of this work is to develop an inter-cluster communication module for the emergent Kalray MPPA-256 Lightweight Manycore Processor. This module is part of a generic and flexible Hardware Abstraction Layer (HAL) for lightweight manycores that cope with the key issues encountered in designing an OS for these processors. On top of the module, communication services will also be proposed for a microkernel-based OS that seeks to provide bare bones for system abstractions. These contributions are embedded in a broader research context that seeks to investigate a fully-featured distributed OS based on a multikernel approach to these processors.

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

LIST OF FIGURES

LIST OF TABLES

CONTENTS

Você pode comparar temperatura e desempenho?

1 INTRODUCTION

① very high, limiting the adoption of this practice.

Great.

For some years, it was common to increase the frequency of processors to improve their processing power. However, as a side effect, the temperature rise was much higher than the performance, making this practice prohibitive. 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.

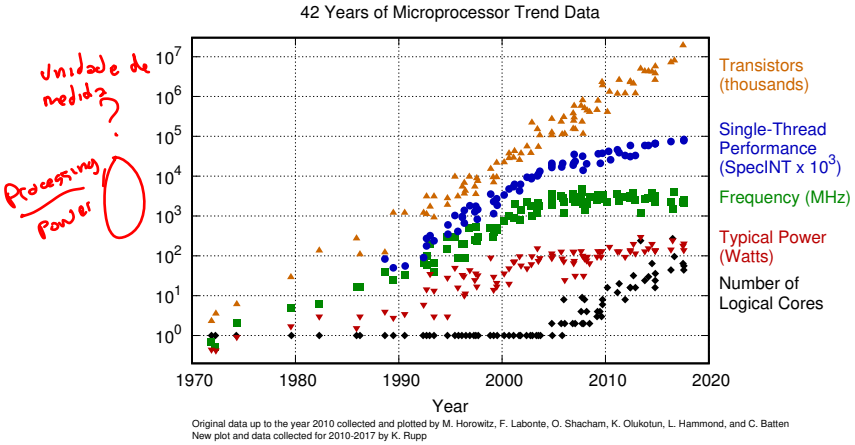
Such researches led to a wide diversity of trade-offs in modern architectures. For instance, different types of instruction sets, instruction parallelism, out-of-order processing techniques, detour prediction techniques, and various memory hierarchies were some of the key techniques proposed to improve the performance of a single core. Then, the performance of computer systems has been improved 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 lead to the advent of *manycores*. Notwithstanding, the line between *multicores* and *manycores* is very tenuous. Some researchers argue that the in 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 is based on the ratio between processing power (*Floating-point Operations per Second* (FLOPS)) and power consumption (*Watts* (W)) of a manycore, as shown in Figure 1. 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). Recently, a new class of parallel processors, called *lightweight manycores*, has emerged to provide high parallelism with low power consumption. These processors own the following characteristics:

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

Figure 1: 42 years of Multiprocessor Trend Data.



Source: (RUPP, 2018).

- Present constrained memory systems; and
- 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). Jointly with further performance scalability and energy efficiency, lightweight manycores brought a new set of challenges in software development coming from their architectural particularities. Precisely, these particularities introduce 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, Márcio 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 nonportable, e.g., do not 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 another hand, the complicated portability and scalability of traditional OSs with a monolithic kernels, which were designed to homogeneous hardwares, 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; COTA DE FREITAS, et al., 2017; PENNA; CASTRO, Márcio, et al., 2017; PENNA; SOUZA; JUNIOR; SOUTO, 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, we are simultaneously designing and implementing a *Microkernel* that provides bare bones system abstractions for each cluster.

1.1 GOALS

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

1.1.1 General Goals

The foremost goal of this undergraduate dissertation is to propose a *Inter-Cluster Communication Module* to the Nanvix HAL and port it to the Kalray MPPA-256 manycore processor (DINECHIN et al., 2013). This module

will expose the essential abstractions that permit upper layers to create more sophisticated communication services. Using this module, we also propose *Inter-Cluster Communication Services* to the Nanvix Microkernel. This work will be done in collaboration with Pedro Henrique Penna, who is a Ph.D. student at the University of Grenoble Alpes, ~~the main~~ developer of Nanvix and the co-advisor of this work.

1.1.2 Specific Goals

- Implement the proposed interfaces of the Inter-Cluster Communication Module of the HAL for the *Kalray MPPA-256 manycore processor*;
- Define Communication Services and their strategies in the use of the Inter-Cluster Communication Module in the context of a microkernel-based OS;
- Implement the Communication Services for the Nanvix Microkernel;
- Analyze the performance of the proposed implementation using specific micro-benchmarks.

1.2 ORGANIZATION OF THE WORK

The remainder of this work is organized as follows. Chapter 2 describes the background needed to accomplish this work. Chapter 3 discusses the principal related work. Chapter 4 presents the proposal for this work. Chapter 5 shows the schedule for the second phase of the development of the undergraduate dissertation. Finally, Chapter 6 concludes this work.

SUGESTÃO: Evitar este parágrafo de início de capítulo apenas pl explicar a estrutura do capítulo. Acho que pode ir mais direto ao ponto. Indico (em azul) sugestões de movimentação do texto que acredito ser mais adequado para abrir o capítulo

2 BACKGROUND

This chapter presents the background of multiple processor systems from a hardware and software perspective. Specifically, Section 2.1.1 and Section 2.1.2 ~~will~~ address details about multiprocessors and multicomputers, respectively. Subsequently, Section 2.2 will present the Kalray MPPA-256 processor. Next, Section 2.3 ~~will~~ show an overview of the Nanvix OS. Finally, Section 2.3.1.1 ~~will~~ cover the abstraction concepts in the Inter-Cluster Communication Module.

MOVIDO mas não foi reescrito.

~~2.1 MULTIPLE PROCESSOR SYSTEMS~~

According to Tanenbaum (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 systems. The sections below address the two first models presenting significant hardware and software concepts for the present undergraduate dissertation.

2.1

~~2.1.1~~ Multiprocessors

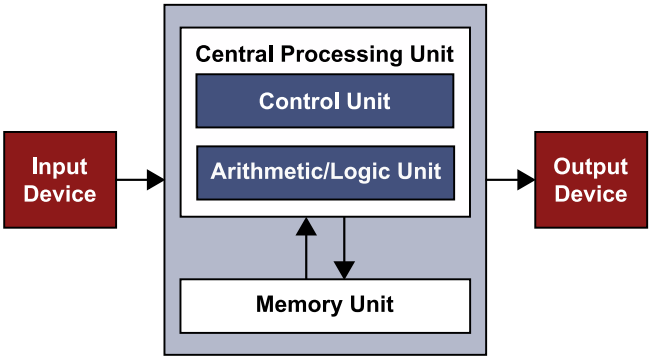
In the early days of electronic digital computing, John Von Neumann proposed an architectural model for computers to be easily programmable (NEUMANN, 1945). As illustrated in Figure 2, this model describes a Central Processing Unit (CPU), also called core, that loads instructions and data from a Memory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

In this context, a shared-memory multiprocessor is a computer system in which two or more CPUs share full access to a common Random Access Memory (RAM) (TANENBAUM; BOS, 2014). Concurrency issues begin to appear where are many CPUs competing for shared resources. For instance, when many threads of a process competing to read and write a global variable. 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.

Concorrência existe mesmo em sistemas com uma única CPU, basta o sistema ser multitarefa. Achei confusa a explicação dos problemas de concorrência.

vou reescrever

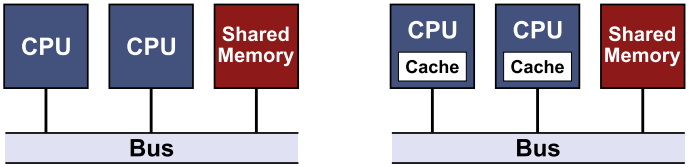
Figure 2: Von Neumann Architecture Model.



Source: Adapted from (TANENBAUM; BOS, 2014).

Figure 3: Two Bus-Based UMA Multiprocessor Examples.

(a) Without caching. (b) With caching.



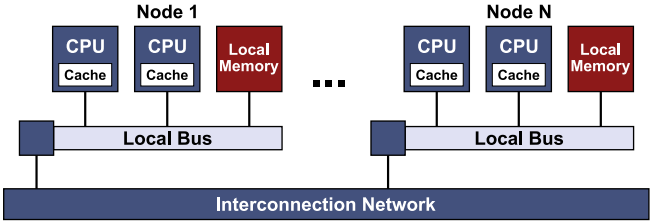
Source: Adapted from (TANENBAUM; BOS, 2014).

2.1.1.1 Multiprocessor Hardware

The multiprocessors can be usually classified using 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 firsts UMA multiprocessors were bus-based architectures where the CPU wait for the bus channel stays free to perform a memory access, as illustrated in Figure 3(a). When the number of cores scale, the bus traffic begin to be a bottleneck of the system. To solve this problem, a small but

Figure 4: NUMA Multiprocessor Example.



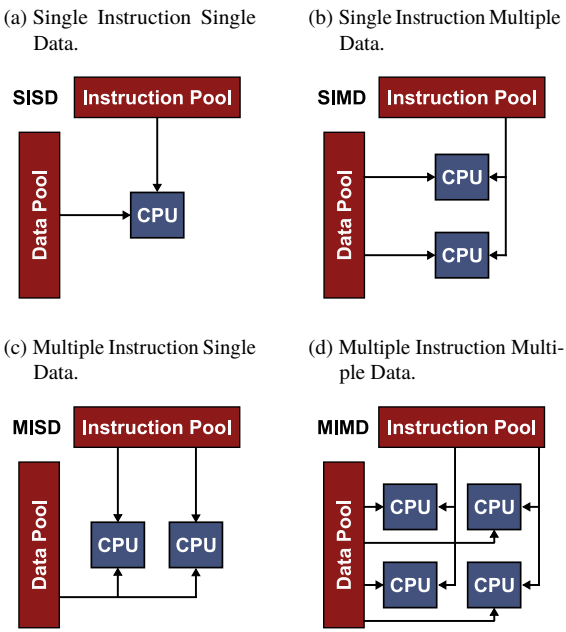
Source: Adapted from (TANENBAUM; BOS, 2014).

fast memory level, called cache, is added to each CPU, as depicted in Figure 3(b). The cache allowed 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 dirties 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.

Nevertheless, the number of cores in UMA multiprocessors are limited to a few dozens of CPUs. Thus, to allow hundreds of cores to communicate, Figure 4 illustrated as that NUMA machines provide a single address space visible to all CPUs through an interconnection network. Therefore, distributing a virtual memory space among local physics memories, the access is guaranteed via load and store instructions. Although the time to access to remote memory is slower than to local ones, this granted that all UMA programs will be able to run on NUMA machines but with worse performance.

In the second place, the workflow classification proposed by Michael J. Flynn (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 uses theses multiprocessors, like modern flight control systems.

Figure 5: Flynn's taxonomy.

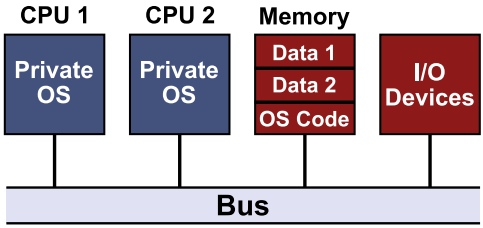


Source: Adapted from (WIKIPEDIA, 2019).

Finally, a Multiple Instruction Multiple Data (MIMD) architecture has multiple processors simultaneously executing different instruction on different data, like Intel Xeon Phi.

Currently, two categories of multiprocessors attract attention, the Chip Multiprocessor (CMP) and Multiprocessor System-on-Chip (MPSoC). CMPs are multicore commercials, 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 communications subsystem, called NoC. The NoC improve scalability and power consumption compared to other communication subsystem designs.

Figure 6: Replicated OS Model.



Source: Adapted from (TANENBAUM; BOS, 2014).

2.1.1.2 Multiprocessor OSs

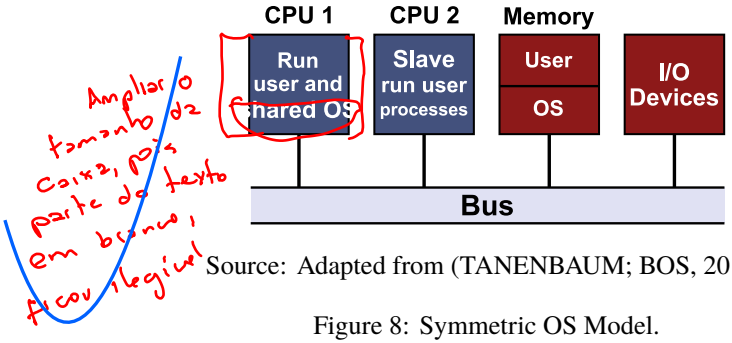
OSs are a fundamental part of any computer system. They act as an intermediary between users and hardware, with the purpose to provide an environment in which users can run programs in a conveniently and efficiently manner (SILBERSCHATZ; GALVIN; GAGNE, 2012). Many OS approaches exist in multiprocessor 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 shows 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.

However, this model has imperceptible 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 I/O 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.

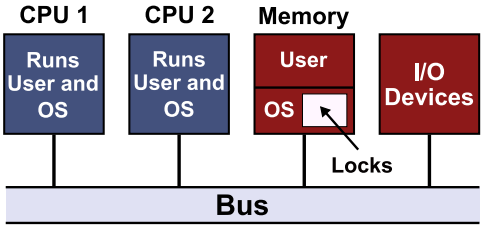
The Master-Slave model began to attract attention with the return of processors with no cache coherence. As Figure 7 shows, there is only one copy of the internal OS structures, and they all belong to a single CPU, called master. In this way, all system calls performed by a worker CPU, called slave, are redirected to the master. With these changes, this model solves the problems

Figure 7: Master-Slave OS Model.



Source: Adapted from (TANENBAUM; BOS, 2014).

Figure 8: Symmetric OS Model.



Source: Adapted from (TANENBAUM; BOS, 2014).

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 requisition.

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 shared in memory. When a CPU makes 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 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 process from scratch.

2.1.2 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 archietcture. 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).

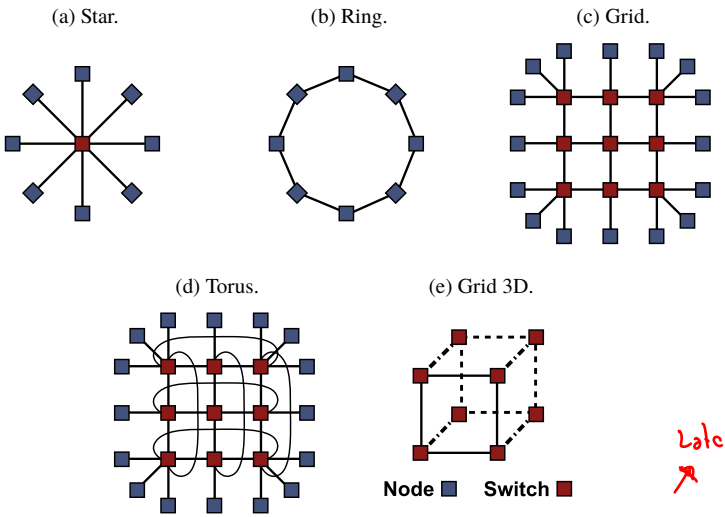
2.1.2.1 Multicomputer Hardware

A multicomputer node can be considered as an elementary computer, with one or more multiprocessors, local RAM and I/O 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.

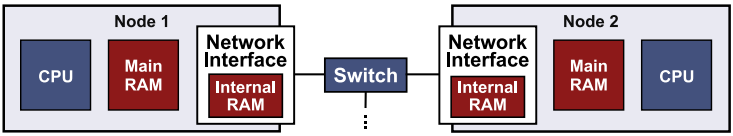
There are two types of switching schemes in the multicomputer network. The *store-and-forward packet* switching scheme breaks the message into fixed-size packets. The packets are copied and moved between the switches following a routing algorithm until they reach the destination. Although flexible and 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

Figure 9: Network Topologies Examples.



Source: Adapted from (TANENBAUM; BOS, 2014).

Figure 10: Simple Multicomputer Example.



Source: Adapted from (TANENBAUM; BOS, 2014).

communication stream, although the slow start and possible sub-utilization of the resources.

2.1.2.2 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 recipient. Although, depending on how the OS provides access to communication resources, that number can grow a lot. 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. Nonetheless, they introduce problems of concurrency between the various users by the 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.

2.1.2.3 User-Level Communication Software

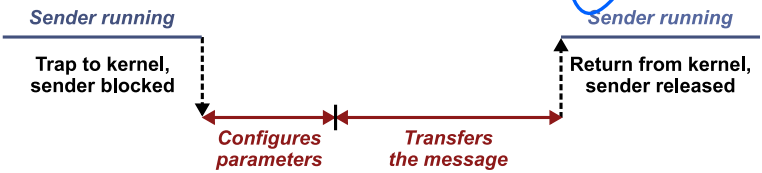
The low-level mechanisms discussed above allow CPUs on different computers to communicate through the messages exchange by send/receive primitives. The basic configuration needed for send primitive is knowing the recipient identifier and the message. Moreover, the receive primitive needs to identify which of the network interfaces it should configure and where to write the incoming data.

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:

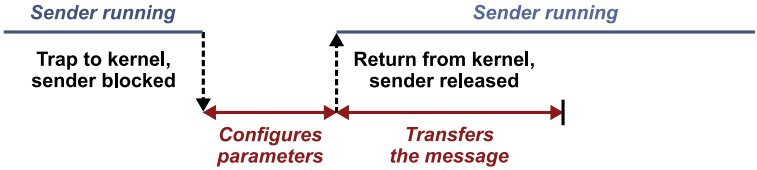
- *Blocking sending:* CPU hibernates or schedules another process, while

Figure 11: Calls Types.

(a) Synchronous Call on Sender Node.



(b) Asynchronous Call on Sender Node.



Source: Adapted from (TANENBAUM; BOS, 2014).

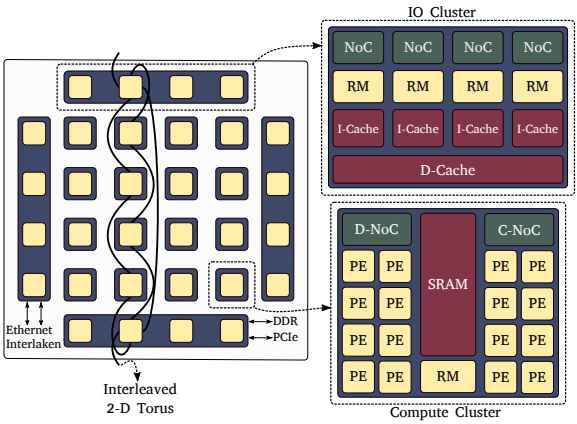
the message is transmitted;

- *Non-blocking sending with copying:* realizes 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, difficulting the programmability;
- *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;

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



Source: (PENNA; SOUZA; JUNIOR; NASCIMENTO, B. M. d., et al., 2018).

- *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 difficults the programmability.

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 developers to use a messaging strategy regardless of what the hardware has to offer.

2.2 THE MPPA-256 LIGHTWEIGHT MANYCORE PROCESSOR

The 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 multiprocessor model is used inside a cluster to coordinate the cores and the local resource balancing, and follows a multicomputer model in inter-cluster communication.

As illustrated by Figure 12, the used version of the architecture, called Bostan, 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.

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. On the other hand, 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 NoC (C-NoC) is exclusive to 64-bit control messages, usually used for synchronization. On the other hand, intense exchange data occurs through the Data NoC (D-NoC). Additionally, all clusters have available DMAs associated with each NoC interfaces to handle communication issues.

As discussed in Section 2.1.2.3, the 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 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.

2.3 NANVIX: AN OPERATING SYSTEM FOR LIGHTWEIGHT MANYCORES

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 (PENNA; FRANCIS; SOUTO, 2019; PENNA; SOUZA; JUNIOR; SOUTO, et al., 2019). In this context, the Nanvix OS aims improving programmability and software portability in lightweight manycores by means of a fully-featured Portable Operating System Interface (POSIX)-compliant OS (PENNA; FRANCIS; SOUTO, 2019). It

follows a multi-kernel design where OS services are scattered across cores and interacting with user processes through a Client-Server approach.

Nanvix OS can be divided into three distinct kernels, *Multikernel*, *Microkernel*, and HAL. The *Multikernel* is made up of high-level OS services, e.g., shared memory regions and name resolution. It exports both the client and server interfaces, both developed above the *Microkernel* services. On top of the HAL, the *Microkernel* follows the Master-Slave OS model to avoid the problem of the lack of coherence of many manycores. It shall run in each cluster and provide bare bones system abstractions, such as thread management, thread synchronization, virtual memory support and communication services. And finally, the next section will introduce HAL.

2.3.1 Hardware Abstract Layer (HAL)

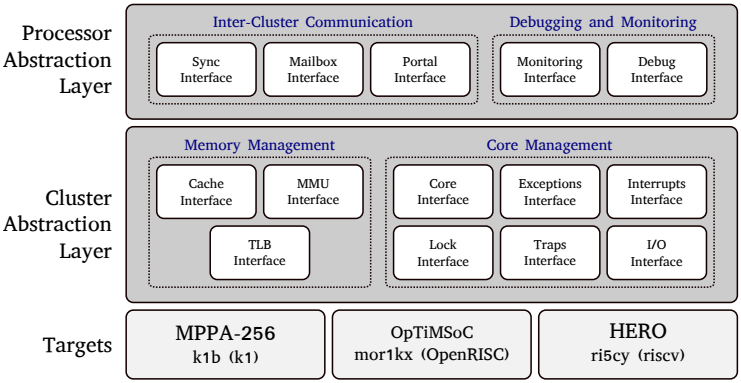
Nanvix OS proposes a generic and flexible HAL around the intrinsic architectural characteristics of the *lightweight manycores*. In this undergraduate dissertation, we focus on the development of HAL to Kalray MPPA-256 (DINECHIN et al., 2013). However, other efforts have also been done for other platforms such as OpTiMSoC (WALLENTOWITZ et al., 2013).

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 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 applications, e.g., OS. As illustrated in Figure 13, this HAL is structured in two major logic layers: *Cluster Abstraction Layer* and *Processor Abstraction Layer*.

The *Cluster Abstraction Layer* encapsulates the management of a single cluster. It provides two modules, *Core Management*, and *Memory Management*. The *Core Management Module* aims to provide to a client application a complete thread synchronization and management system, rich support of handling interrupts/exceptions, and an adequate system call interface. The *Memory Management Module* provides a uniform view of the TLBs and paging systems with maintenance routines for them and the cache. 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 Management module because this would force the client OS to write code in assembly, hurting the conceptual idea of the HAL.

The *Processor Abstraction Layer*, in particular, embraces architectural features related to multiple clusters. The *Inter-Cluster Communication Module*, the focus of this undergraduate dissertation, exports three main abstractions to allow the cluster to exchange data between them, based on ideas proposed along with the NodeOS distributed runtime system (DINECHIN et al., 2013).

Figure 13: Structural overview of the proposed HAL.



Source: (PENNA; FRANCIS; SOUTO, 2019).

Lastly, the *Monitoring and Debugging Module*, as the name shows, exports routines to aid debugging and extraction of diagnostics about hardware metrics, such as the number of page-fault or detour taken.

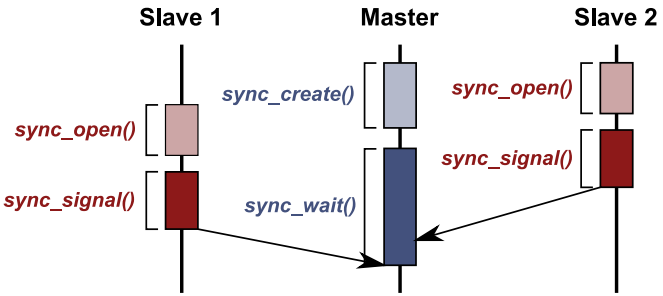
2.3.1.1 Inter-Cluster Communication Module

The following sections conceptually present the three abstractions exported by HAL and which will be implemented on the Kalray MPPA-256. They are the *Sync*, *Mailbox*, and *Portal* abstraction. The purpose of developing more specific abstractions than send and receive primitives, described in Section 2.1.2.3, is to provide more precise and easy-to-use mechanisms for OS and client applications. On top of them, it is possible to create all kinds of essential services such as message passing and synchronization, to more elaborate services such as shared memory regions (PENNA; SOUZA; NASCIMENTO, B., et al., 2018).

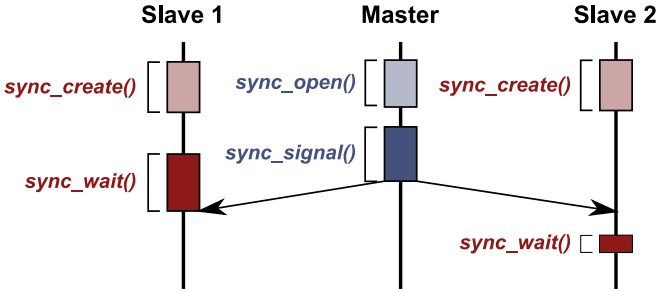
A related point to consider is that mechanisms described below refer to interactions between distinct clusters only. As described in Section ??, these system abstractions can use the communication primitives, i.e., send and receive primitives, to export more elaborate services. The behavior expected can be simulated both through synchronous calls and asynchronous calls, depending only on hardware support. In the same way, it is worth be noted that the small messaging exchange is decoupled from abstraction to intense data transfer. The motivation for this, as other details, is to exhibit better control over the Quality of Service (QoS) to the higher layers. For instance, the use of distinct NoCs for each service.

Figure 14: Synchronization Abstraction Concept.

(a) N Slaves notify the Master.



(b) The Master notify N Slaves.



Source: Developed by the Author.

Sync Abstraction

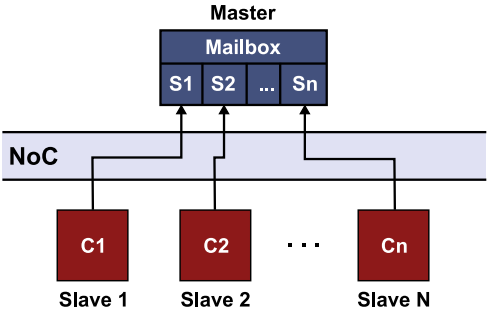
The *Synchronization Abstraction*, called *Sync*, proves bare bones for cluster synchronization. From this, it is possible to create distributed barriers. For instance, a set of clusters can wait for a notification coming from a specific cluster. The behavior is analogous to the POSIX signal abstraction, but the *Sync* provides only the case of notification. Specifically, as can be seen in Figure 14, the cardinality of the operation is N:1, where N clusters wait for 1 Cluster to send a notification.

Mailbox Abstraction

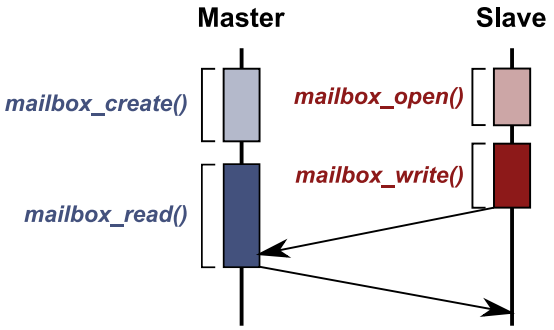
The *Mailbox Abstraction* allows clusters to exchange fixed-size messages with each other. The message was thought to be a relatively small size, usually

Figure 15: Mailbox Abstraction Concept.

(a) Conceptual Overview.



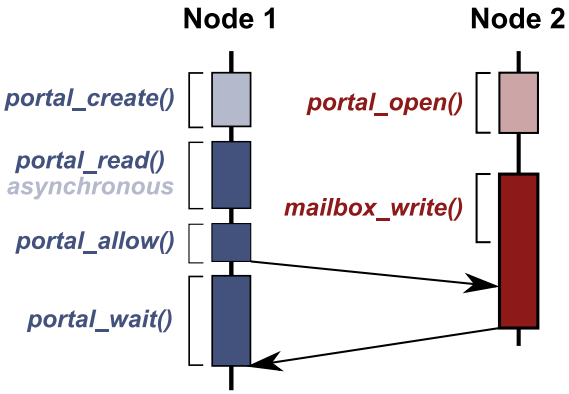
(b) Flow of execution: Slave sends a message, Master reads and notifies the sender.



Source: Developed by the Author.

a few hundred bytes. Similarly, the operation of the *Mailbox* follows POSIX message queue behavior. For example, the message can be used to encode small operations and system control signals. As illustrated in Figure 15, the operation cardinality is N:1, where N senders can transfer one message at a time to a receiver queue. When the receiver consumes a message, it notifies the sender to ensure control of the flow.

Figure 16: Portal Abstraction Concept: Node 1 create a portal and notify Node 2 to transfer the data.



Source: Developed by the Author.

Portal Abstraction

Lastly, the *Portal Abstraction* enables two clusters to exchange arbitrary amounts of data. The conceptual idea of the *Portal* is similar to that implemented in POSIX pipes. Figure 16 shows the *Portal* operation, with cardinality 1:1, a cluster pair opens a channel to transfer data. However, the operation is only allowed in one direction with a flow control mechanism, where the receiver cluster warns the sender cluster when it is ready to receive.

2.3.2 Communication Services for the Nanvix Microkernel

The inter-cluster communication module, described in Section 2.3.1.1, is designed to export a standard and straightforward communication primitives to different lightweight manycores. These primitives can be used by various types of operating systems 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 responsibility of protecting, manipulating, and configuring HAL resources is of the master core. The slave

core will request operations through a meta-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 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.

3 RELATED WORK

The proposal of this work is related to several other research works. First, some research papers describing state-of-the-art *lightweight manycores* processors will be cited. Further, research on different OSs proposed for such processors will be highlighted.

3.1 LIGHTWEIGHT MANYCORE PROCESSORS

In addition to Kalray MPPA-256, many works exemplify the wide variety of architectural possibilities of *lightweight manycores*. For instance, Olofsson *et al.* (OLOFSSON; NORDSTROM; UL-ABDIN, 2014) introduce Adapteva Epiphany as a high-performance energy-efficient *manycore* architecture suitable for real-time embedded systems. The architecture consists of nodes communicating 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, named rMesh, cMesh, and xMesh, are independent and scalable. They implements a packet-switched model with four duplex links at every node. When a node want read a remote data, first, it sends a read request over the rMesh and wait a signal to arrive on cMesh. On the another hand, the write operation allows the node to continue running while data is tranfered through over xMesh.

On the other hand, for help and facilitate on the *manycore* processor design, Wallentowitz *et al.* (WALLENTOWITZ et al., 2013) presents the open-source framework OpTiMSoC which allows build *manycore* System-on-a-Chip (SoC) and simulate them on a computer or synthesize them on a Field Programmable Gate Array (FPGA). The PE are OpenRISC [1] processor organized in tiles. The central architectural element is the LISNoC, a *packet-switched NoC* that implements virtual channels to avoid message-dependent deadlocks. The LISNoC support various network topologies, depending only on the tiles organization. Precisely, a *network adapter* handles the memory transfers between tile and the memory and provide hardware means to a message-passing communication model among tiles. The tiles organization and the network topology also allows handle communication in three different ways. First, when using distributed memory, communication is performed through the message-exchange between the tiles. Second, by using a partitioned global address space scattered through the local memories of the tiles, the network adapter can perform Memory Protection Unit (MPU)-like memory

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

address translation hiding the exchange of messages but without providing cache coherence. Finally, Wallentowitz *et al.* study policy of write-through snooping to provide a global memory shared among tiles with cache coherence.

Similarly, Kurth *et al.* (KURTH et al., 2017-10) introduce the HERO, which unites an ARM Cortex-A host processor with a fully modifiable RISC-V *manycore* implemented on a FPGA. The Programmable Manycore Accelerator (PMCA) uses a multi-cluster design and also relies on multi-banked memory, called Software-managed Scratchpad Memories (SPM). The data caches had substituted to a multi-channel DMA engine that copy data between a shared L1 SPM and remote SPMs or shared main 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 similar to an MMU. In this way, clusters can share virtual address pointers. Besides, exists different designs for the shared instruction caches and top-level interconnection such as bus or NoC.

3.2 OPERATING SYSTEMS FOR LIGHTWEIGHT MANYCORES

Baumann *et al.* (BAUMANN et al., 2009) proposed a new OS architecture for scalable multicore systems, called *Multikernel*. In their perspective, the future of the OSs is on embracing the networked nature of the machines based on distributed systems ideas. Assuming the cores are independent nodes of a network, they build the tradicional OS functionalities as fully-featured processes. The processes communicate via message-passing and does 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, follow three design principles. First, *Make all inter-core communication explicit* turns the system amenable to human or automated analysis because processes communicate only through well-defined interfaces. Second, *Make OS structures hardware-neutral* makes the hardware-independent code easy to debug, optimize, and facilitates the deploy the OS for new processor types, avoiding rework. And lastly, *View OS state as replicated instead of shared* improve system scalability.

In Wisniewski (WISNIEWSKI et al., 2014) *et al.*, the concept of scalability was pushed to the extreme, thinking on 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 to performance, scalability, and reliability. The OS, named multi Operating System

(mOS), provide as much of the compute hardware resources as possible to the HPC applications. On the other hand, the Linux kernel component acts as a service that provides Linux functionalities.

In like manner, Kluge *et al.* (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. Where 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, they partition *manycore* and distribute replicas of Servers, turning the whole system more predictable. However, in order to deal with interferences in shared resources, usage policies are introduced to make possible the prediction of system behavior.

Finally, Nightingale *et al.* (NIGHTINGALE et al., 2009) presents the Helios OS to simplify the process of writing, deploy, and optimize 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. Also, request the minimum of hardware primitives so that architectures with many constraints can be ported. Moreover, request the minimum hardware resources to support architectures with little computational power or memory constraints.

3.3 DISCUSSION

Section 3.1 exemplifies 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.

On the other hand, Section 3.2 presents studies of OSs that focus on mitigating these differences in order to provide greater programmability and portability. Nevertheless, they span the entire development spectrum, delivering to end-users high-level abstractions for increase productivity. In this way, it is possible to notice the extensive rework that all of them had in dealing with the closest layer of hardware that, as observed, can be abstracted to a common perspective.

In this context, the proposed HAL deal with the lowest level possible with a focus on the aspects that make it challenging to work with *manycores*. Thus, different OSs and services can be developed and disseminated by several

architectures that implement such the perspective. So consequently, the applications that run on top of them.

4 PROPOSAL

The proposal for this undergraduate dissertation is made up of two main contributions. The first contribution will be the port of the *Inter-Cluster Communication Module*, described in Section 2.3.1.1, for the Kalray MPPA-256 manycore processor. The second contribution will be the design and implementation of communication services of a master-slave OS, described in Section 2.3.2, on top of the communication module.

4.1 INTER-CLUSTER COMMUNICATION MODULE

Ideally, the HAL implementation should not use any other layer of software to deal with the hardware. However, the implementation of HAL in Kalray MPPA-256 uses the software stack made available by Kalray. In order to replace them, a great effort would be necessary, fleeing the proposal of this work and the scope of Pedro H. Penna's Ph.D. The main reasons for this decision were due to poor documentation and the inability to execute kernel mode code freely.

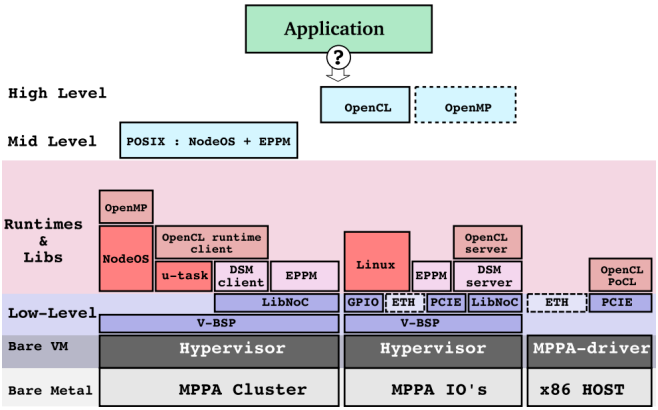
Figure 17 shows the software stack present for the Kalray MPPA-256. From it, only the *hypervisor* and the *vbsp* library will be used. The hypervisor is used to virtualize the hardware by separating it into logical parts, e.g., core virtualization, C-NoC virtualization, and D-NoC virtualization. It also exports routines to manage, configure, and allow access to virtual resources. The vbsp library, in turn, provides primitives for setting interrupts.

The inter-cluster communication module will use HAL's interfaces and the virtualizations of the C-NoC and D-NoC interfaces directly. Virtual NoC interfaces export asynchronous calls to acquire read and write permission from registers that configure the hardware for a given task. For instance, the DMA configuration for asynchronous sending is abstracted by μthreads that are configured through global structures provided by the hypervisor.

Clusters have two identifiers, one physical (physical ID) and the other logical (logical ID). The physical ID's are the numbers in hardware that identically them during the process of routing the data through NoC. The logical ID's are numbers associated with the physical ID's so that it is possible to identify the clusters outside the HAL. Logical ID's primarily serve to disassociate the cluster identification from the architecture in which HAL is implemented. Table 1 shows the physical and logical identification performed for Kalray MPPA-256.

To perform the communication between two clusters, it is necessary that the sender knows which resource the receiver will use. For this reason,

Figure 17: Software Stack of the Kalray MPPA-256.



Source: Kalray MPPA-256 Processor Documentation.

Table 1: Cluster Identification.

	Physical ID	Logical ID
Compute Cluster	0-15	0-15
I/O Cluster 0	128-131	16-19
I/O Cluster 1	192-195	20-23

Table 2: Partitions of NoC resources by abstraction.

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

the receive slot range of C-NoC and D-NoC are partitioned by abstraction, as can be seen in Table 2. Within a partition, each slot is associated with a cluster's logical ID. On the other hand, the transmitters use sending channels that need to be reserved during the entire operation. Thus, Table 2 also shows the partition of the sending channels for each abstraction.

Pode explicar 2 tabelas

Code 4.1: HAL Sync Interface for Receiver Cluster

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

Code 4.2: HAL Sync Interface for Sender Cluster

```
1 /*  @brief Allocates and configures the sending side of
2  *  the synchronization point.                          */
3 int sync_open(void);
4
5 /* @brief Releases the sender resources on a specific DMA channel. */
6 int sync_close(int syncid);
7
8 /* @brief Send signal on a specific synchronization point. */
9 int sync_signal(int syncid, const int *nodes, int nnodes, int type);
```

4.1.1 Sync

As described in Section 2.3.1.1, the *Sync* abstraction allows the creation of distributed barriers. The *Sync* can be decomposed into two sets of functions, where the Code 4.1 shows the functions to one clusters that will receive signals, and Code 4.2 shows the ones that will send. Each set allocates different types of resources. Accurately, the receiver must allocate a receive slot of the frame, and the transmitter will allocate a transmitter channel of the frame. Because of this, 24 synchronization points can be created (`sync_create()`), and only 1 can be opened (`sync_open()`) per cluster.

Sync functions implement two different types, depending on who cluster should be the master of the synchronization operation. Specifically, every synchronization operation is separated into a master cluster and one or more slave clusters defined by the `ONE_TO_ALL` and `ALL_TO_ONE` constants. The master always takes the role "ONE" and the slaves the "ALL" role. The following subsections describe the behavior and implementation details of each type of synchronization.

`ONE_TO_ALL` Synchronization Type

The slave clusters involved in this type of synchronization expect a signal sent from the master. To do this, each slave will perform the function

Não está referenciando este código no texto

Code 4.3: HAL Mailbox Interface for Receiver Cluster

```
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 from a mailbox. */
8 ssize_t mailbox_read(int mbxid, void * buffer, size_t size);
```

`sync_create()` by allocating the resource associated with the master's logical ID. After allocating and configuring the resource, the cluster can continue to run without worrying about whether the signal was received. When the time comes to perform the synchronization, it is enough for the slave to run the function. Upon receiving the signal, the sync will auto reconfigure the resource and free the cluster.

The cluster master, in turn, must perform the complementary functions. That is, first, the master will allocate a sending resource by calling the `sync_open()` function. After that, it should inform the set of clusters that will receive the signal on its logical ID. Finally, both can release sync resources by calling release functions, `sync_unlink()` for the slave and `sync_close()` to the master.

`ALL_TO_ONE` Synchronization Type

Analogously to the previous type, the only changes are just on the roles in which they are reversed. The master must perform the functions of creation, configuration of the resources, and waiting for the numerous signals. The resource allocated by the master must be the resource associated with its logical ID. On the other hand, the slaves will perform the functions of opening and sending a signal on the master's logical ID.

4.1.2 Mailbox

As explained in Section 2.3.1.1, the *Mailbox* abstraction allows the exchange of small messages of sizes between clusters similar to a message queue. The *Mailbox* is more complex than the previous abstraction because it uses both D-NoC and C-NoC resources. When executing the function `mailbox_create()`, the receiver will only use one D-NoC receive slot and configures it with a kernel memory space, sufficient to receive 24 messages. The messages are composed of the header identifying the sender and a body containing the useful message. When consuming a message (`mailbox_read()`),

Não está referenciando este código no texto

Code 4.4: HAL Mailbox Interface for Sender Cluster

```
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 to a mailbox. */
8 ssize_t mailbox_write(int mbxid, const void * buffer, size_t size);
```

the receiver will copy the message to the user's buffer and send a signal to the sender informing him that it can send another message. If there is no message in the buffer, the receiver is blocked until a message is received.

On the other hand, the sender will allocate a receive signal slot (`mailbox_open()`) before sending its first message to the receiver. If the sender attempts to send a message before the receiver has consumed the previous message, the sender will be blocked waiting for the sender's notification. In this way, flow control is guaranteed, and the sender will not overwrite messages unread by the receiver. Sending the message will always be executed asynchronously because it will always be necessary to copy the message to a kernel buffer that contains the header. Thus, the sender will never be blocked waiting for the message to be sent. In this configuration, the number of mailbox creations (`mailbox_create()`) within a cluster is limited to 1 because of the C-NoC sending channel. On the other hand, the maximum number of opens (`mailbox_open()`) is 4 because of the limitation of the available D-NoC sending channels.

4.1.3 Portal

Section 2.3.1.1 showed that the portal abstraction is similar to a POSIX pipe with flow control. So, the *Portal* analogously follows the ideas implemented in the *Mailbox* with the difference of the option to write, that can be synchronously or asynchronously. The total number of receive data operations (`portal_create()`) is limited to 2 because of the number of available signal sending channels. On the other hand, up to 4 send operations (`portal_open()`) can be performed simultaneously because there are 4 available send channels for the *Portal*.

Function `portal_open()` starts the data send operation. In it will be allocated a channel of data sending associated with a μthread. Also, a signal receiving slot will be allocated to receive the signal that will release the sender to transfer data to the receiver. In asynchronous send operation (`portal_awrite()`), the cluster cannot modify or release the buffer until the

Não está referenciando este código no texto

Code 4.5: HAL Portal Interface for Receiver Cluster

```
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_read(int portalid, void * buffer, size_t size);
12
13 /* @brief Waits for an asynchronous operation on a
14  *        portal to complete.                    */
15 int portal_wait(int portalid);
```

Code 4.6: HAL Portal Interface for Sender Cluster

```
1 /* @brief Opens a portal. */
2 int portal_open(int remote);
3
4 /* @brief Closes a portal. */
5 int portal_close(int portalid);
6
7 /* @brief Writes data to a portal. */
8 ssize_t portal_write(int portalid, const void * buffer, size_t size);
9
10 /* @brief Writes data asynchronously to a portal. */
11 int portal_awrite(int portalid, const void * buffer, size_t size);
12
13 /* @brief Waits for an asynchronous operation on a
14  *        portal to complete.                    */
15 int portal_wait(int portalid);
```

operation is completed. To ensure that the buffer can use, the cluster must call function `portal_wait()`.

The receiver will allocate a receive slot of the D-NoC and a send channel (`portal_create()`) to receive data from another cluster. After setting up the resources (`portal_read()`), the receiver can notify a sender (`portal_allow()`), enabling it to transfer data. For this reason, the read operation is always performed asynchronously. After receiving the set amount of data, the receiver can use the buffer securely.

4.2 COMMUNICATION SERVICES

As described in Section 2.3.2, three communication services will be developed for the Nanvix Microkernel, named *Sync*, *Mailbox*, and *Portal* services.

Each service will be responsible for protecting, managing, manipulating, and multiplexing the resources exposed by the HAL communication module. These services must take into account the memory constraints and the master-slave model chosen for the Microkernel.

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 protection operations are mostly similar. For instance, the use of unallocated resources, sanitizing entries, checking valid identifiers, non-null pointers, and checking for conflicting operations (reading in write-only resources). In the meantime, there are exceptional cases in some services that must be taken. For instance, in the *Sync* service, a cluster cannot synchronize with itself, or there is a repetition of identifiers in the stipulated set of clusters.

Finally, some aspects of services and implementation still need to be analyzed and will be better detailed in another version of the dissertation. For example, what resource multiplexing methods will be used and their impacts on the Nanvix Microkernel services.

Para o final do trabalho, é importante destacar:
- como trata as operações de proteção
- como pretende testar/verificar a corretude dos serviços implementados.

5 SCHEDULE

This chapter presents the schedule for the next activities planned for the development of the undergraduate dissertation.

5.1 ACTIVITIES

Figure 18: Chart Gantt of the Schedule.

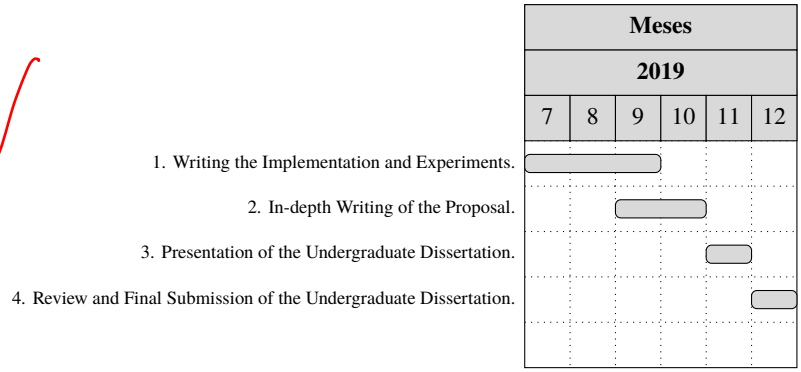


Figure 18 shows the planned activities and their durations visually. Beginning in July 2019, the final submission is planned for December 2019. In detail, the activities are described below:

- *Writing the Implementation and Experiments:* Currently, the inter-cluster communication module has the *Sync* abstraction completed and part of the *Mailbox* abstraction. Since the *Portal* abstraction uses the same low-level mechanisms of the others abstractions, its implementation will be facilitated. The communication services already have prototypes developed by the author for a symmetric OS. In this way, the prototypes will need to be modified to use the HAL and modified for a master-slave model. Finally, micro-benchmarks will be developed to perform an analysis of the performance of the implemented services.

- *In-depth Writing of the Proposal:* During September and October, this activity will be committed to improving this draft and detailing the project decisions chose and implementations produced. We will also describe the experiments performed and discuss the results obtained.

- *Presentation of the Undergraduate Dissertation:* November will be dedicated to the development and preparation of the presentation of the work and the results achieved. So finally, to present the dissertation to the evaluators.
- *Review and Final Submission of the Undergraduate Dissertation:* Finally, December will be dedicated to the correction of the issues indicated by the evaluators and finalized with the final submission of the dissertation.

6 CONCLUSIONS

Initially, this work presented a historical context of multicore processors to the nowadays. By demonstrating the relationship between the growth of the number of core 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, they sin in programmability and portability because of their architectural features, such as hybrid programming model, restrictive memory subsystems, lack of cache coherence, 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 a inter-cluster communication module 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 have been presented that can use the communication module. Thus, to provide the basic functionalities for such OSs, three communication abstractions have been proposed for HAL with the concern of providing QoS. Among them is the *Sync* abstraction to create distributed barriers. The *Mailbox* abstraction provides the exchange of small messages with flow control. So finally, the *Portal* abstraction allows the exchange of arbitrary amounts of data between two clusters.

Another contribution of this work was the communication services for an operating system based on the microkernel approach. These services provide for the multiplexing of the resources exposed by HAL and the verification of the parameters required 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. Lastly, the proposal detailed, in general, several aspects of the implementations. Because the communication services depend on the HAL communication module to be developed, the topics associated with the module have become more detailed and better explored. However, the next version of the undergraduate dissertation will clearly and objectively specify both contributions of this work.

BIBLIOGRAPHY

BARBALACE, Antonio et al. Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms. English. In: PROCEEDINGS of the 10th European Conference on Computer Systems. Bordeaux, France: ACM, Apr. 2015. (EuroSys '15), p. 1–16. ISBN 978-1-4503-3238-5. DOI: 10.1145/2741948.2741962. Available from: <http://dl.acm.org/citation.cfm?doid=2741948.2741962>.

BAUMANN, Andrew et al. The Multikernel: A New OS Architecture for Scalable Multicore Systems. English. In: PROCEEDINGS of the 22nd ACM Symposium on Operating Systems Principles. Big Sky, Montana, USA: ACM, Oct. 2009. (SOSP '09), p. 29–44. ISBN 978-1-60558-752-3. DOI: 10.1145/1629575.1629579. Available from: <https://dl.acm.org/citation.cfm?doid=1629575.1629579>.

CASTRO, Márcio et al. Seismic Wave Propagation Simulations on Low-power and Performance-centric Manycores. **Parallel Computing**, v. 54, p. 108–120, 2016. ISSN 01678191. DOI: 10.1016/j.parco.2016.01.011.

CHRISTGAU, Steffen; SCHNOR, Bettina. Exploring One-Sided Communication and Synchronization on a Non-Cache-Coherent Many-Core Architecture. English. **Concurrency and Computation: Practice and Experience (CCPE)**, v. 29, n. 15, e4113, Mar. 2017. ISSN 1532-0626. DOI: 10.1002/cpe.4113. Available from: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4113>.

DINECHIN, Benoît Dupont de et al. A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. English. In: PROCEDIA Computer Science. Barcelona, Spain: Elsevier, June 2013. v. 18. (ICCS '13), p. 1654–1663. ISBN 1877-0509. DOI: 10.1016/j.procs.2013.05.333. Available from: <http://linkinghub.elsevier.com/retrieve/pii/S1877050913004766>.

FLYNN, Michael J. Some Computer Organizations and Their Effectiveness. **IEEE Trans. Comput.**, IEEE Computer Society, Washington, DC, USA, v. 21, n. 9, p. 948–960, Sept. 1972. ISSN 0018-9340. DOI: 10.1109/TC.1972.5009071. Available from: <http://dx.doi.org/10.1109/TC.1972.5009071>.

FRANCESQUINI, Emilio et al. On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms. English. **Journal of Parallel and Distributed Computing (JPDC)**, v. 76, n. 100, p. 32–48, Feb. 2015. ISSN 0743-7315. DOI: 10.1016/j.jpdc.2014.11.002. Available from: <http://linkinghub.elsevier.com/retrieve/pii/S0743731514002093>.

FREITAS, Henrique Cota de. **Arquitetura de NoC Programável Baseada em Múltiplos Clusters de Cores para Suporte e Padrões de Comunicação Coletiva**. June 2009. PhD thesis – Programa de Pós-Graduação em Computação, UFRGS, Porto Alegre. An optional note.

GAMELL, Marc et al. Exploring Cross-Layer Power Management for PGAS Applications on the SCC Platform. English. In: PROCEEDINGS of the 21st international symposium on High-Performance Parallel and Distributed Computing. Delft, The Netherlands: ACM, June 2012. (HPDC '12), p. 235–246. ISBN 978-1-4503-0805-2. DOI: 10.1145/2287076.2287113. Available from: <http://dl.acm.org/citation.cfm?doid=2287076.2287113>.

KELLY, Ben; GARDNER, William; KYO, Shorin. AutoPilot: Message Passing Parallel Programming for a Cache Incoherent Embedded Manycore Processor. English. In: PROCEEDINGS of the 1st International Workshop on Many-core Embedded Systems. Tel-Aviv, Israel: ACM, June 2013. (MES '13), p. 62–65. ISBN 978-1-4503-2063-4. DOI: 10.1145/2489068.2491624. Available from: <http://dl.acm.org/citation.cfm?doid=2489068.2491624>.

KLUGE, Florian; GERDES, Mike; UNGERER, Theo. An Operating System for Safety-Critical Applications on Manycore Processors. English. In: 2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. Reno, Nevada, USA: IEEE, June 2014. (ISORC '14), p. 238–245. ISBN 978-1-4799-4430-9. DOI: 10.1109/ISORC.2014.30. Available from: <http://ieeexplore.ieee.org/document/6899155/>.

KOGGE, Peter 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, Jan. 2008.

KURTH, Andreas et al. HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA. en. In: PROCEEDINGS of Computer Architecture Research with RISC-V Workshop (CARRV' 17). 2017-10. First Workshop on Computer Architecture Research with RISC-V (CARRV 2017); Conference Location: Boston, MA, USA; Conference Date: October 14, 2017. DOI: 10.3929/ethz-b-000219249.

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

NEUMANN, John von. **First Draft of a Report on the EDVAC**. 1945.

NIGHTINGALE, Edmund et al. Helios: Heterogeneous Multiprocessing with Satellite Kernels. English. In: PROCEEDINGS of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. Big Sky, Montana, USA: ACM, Oct. 2009. (SOSP '09), p. 221–234. ISBN 978-1-60558-752-3. DOI: 10.1145/1629575.1629597. Available from: <http://doi.acm.org/10.1145/1629575.1629597>.

OLOFSSON, Andreas; NORDSTROM, Tomas; UL-ABDIN, Zain. Kickstarting High-Performance Energy-Efficient Manycore Architectures with Epiphany. English. In: 2014 48th Asilomar Conference on Signals, Systems and Computers. Pacific Grove, California, USA: IEEE, Nov. 2014. (ACSSC '14), p. 1719–1726. ISBN 978-1-4799-8297-4. DOI: 10.1109/ACSSC.2014.7094761. Available from: <http://ieeexplore.ieee.org/document/7094761/>.

PENNA, Pedro Henrique; CASTRO, Márcio, et al. Using the Nanvix Operating System in Undergraduate Operating System Courses. English. In: 2017 VII Brazilian Symposium on Computing Systems Engineering. Curitiba, Brazil: IEEE, Nov. 2017. (SBESC '17), p. 193–198. ISBN 978-1-5386-3590-2. DOI: 10.1109/SBESC.2017.33. Available from: <http://ieeexplore.ieee.org/document/8116579/>.

PENNA, Pedro Henrique; COTA DE FREITAS, Henrique, et al. Using The Nanvix Operating System in Undergraduate Operating System Courses. In: VII BRAZILIAN SYMPOSIUM ON COMPUTING SYSTEMS ENGINEERING. Curitiba, Brazil, Nov. 2017. Available from: <https://hal.archives-ouvertes.fr/hal-01635880>.

PENNA, Pedro Henrique; FRANCIS, Davidson; SOUTO, João. 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, June 2019. Available from: <https://hal.archives-ouvertes.fr/hal-02151274>.

PENNA, Pedro Henrique; SOUZA, Matheus; JUNIOR, Emmanuel Podestá; NASCIMENTO, Bruno Marques do, et al. An Operating System Service for Remote Memory Accesses in Low-Power NoC-Based Manycores. October, 2018. DOI: 10.13140/RG.2.2.19732.96649.

PENNA, Pedro Henrique; SOUZA, Matheus; JUNIOR, Emmanuel Podestá; SOUTO, João, 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, Jan. 2019. (High-Performance and Embedded Architectures and

Compilers Workshops (HiPEAC Workshops)), p. 1–16. Available from: <https://hal.archives-ouvertes.fr/hal-01986366>.

PENNA, Pedro Henrique; SOUZA, Matheus; NASCIMENTO, Bruno, et al. An OS Service for Transparent Remote Memory Accesses in NoC-Based Lightweight Manycores. Poster, Oct. 2018.

RHODEN, Barret et al. Improving Per-Node Efficiency in the Datacenter with New OS Abstractions. English. In: PROCEEDINGS of the 2nd ACM Symposium on Cloud Computing. Cascais, Portugal: ACM, Oct. 2011. (SoCC '11), p. 1–8. ISBN 978-1-4503-0976-9. DOI: 10.1145/2038916.2038941. Available from: <https://dl.acm.org/citation.cfm?id=2038941>.

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

SERRES, Olivier et al. Experiences with UPC on TILE-64 Processor. English. In: AEROSPACE Conference. Big Sky, Montana, USA: IEEE, Mar. 2011. (AERO '11), p. 1–9. ISBN 978-1-4244-7350-2. DOI: 10.1109/AERO.2011.5747452. Available from: <http://ieeexplore.ieee.org/document/5747452/>.

SILBERSCHATZ, Abraham; GALVIN, Peter B.; GAGNE, Greg. **Operating System Concepts**. 9th: Wiley Publishing, 2012. ISBN 1118063333, 9781118063330.

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

WALLENTOWITZ, Stefan et al. **Open Tiled Manycore System-on-Chip**. English. Apr. 2013. p. 1–7. Available from: <http://arxiv.org/abs/1304.5081>.

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

WISNIEWSKI, Robert et al. mOS: An Architecture for Extreme-Scale Operating Systems. English. In: ROSS '14 Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers. Munich, Germany: ACM, June 2014. (ROSS '14), p. 1–8. ISBN 978-1-4503-2950-7. DOI: 10.1145/2612262.2612263. Available from: <http://dl.acm.org/citation.cfm?doid=2612262.2612263>.

ZHENG, Fang et al. Cooperative Computing Techniques for a Deeply Fused and Heterogeneous Many-Core Processor Architecture. English. **Journal of Computer Science and Technology (JCST)**, v. 30, n. 1, p. 145–162, Jan. 2015. ISSN 1000-9000. DOI: 10.1007/s11390-015-1510-9. Available from: <https://link.springer.com/article/10.1007%2Fs11390-015-1510-9>.