Nanvix : A Distributed Operating System for Lightweight Manycore Processors

Pedro Henrique Penna

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

préparée dans le cadre d'une cotutelle entre l'*Université Grenoble Alpes* et la *Pontifícia Universidade Católica de Minas Gerais*

Spécialité: **Informatique**

Arrêté ministériel: le 6 janvier 2005 – 25 mai 2016

Présentée par

Pedro Henrique DE MELLO MORADO PENNA

Thèse dirigée par **Jean-François MÉHAUT** et **Henrique FREITAS**

préparée au sein des **Laboratoires d'Informatique de Grenoble**

dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Nanvix: Un Système d'Exploitation Distribué pour les Processeurs Manycore Légers

Thèse soutenue publiquement le **23 Septembre 2021**
devant le jury composé de:

Abdoulaye GAMATIÉ
Directeur de Recherche, CNRS/LIRMM, Président
Pierre SENS
Professeur, Sorbonne Université, Rapporteur
Rodolfo AZEVEDO
Professeur, Universidade de Campinas, Rapporteur
Benoît DINECHIN
Ingéniur Docteur, Kalray Inc, Examinateur
François BROQUEDIS
Maître de Conferénce, Université Grenoble Alpes, Examinateur
Márcio CASTRO
Professeur Associé, Universidade Federal de Santa Catarina, Examinateur
Carlos Augusto MARTINS
Professeur, Pontifícia Universidade Católica de Minas Gerais, Invité

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Programa de Pós-Graduação em Informática

Pedro Henrique de Mello Morado Penna

NANVIX
A Distributed Operating System for Lightweight Manycore Processors

Belo Horizonte
2021

Pedro Henrique de Mello Morado Penna

NANVIX
A Distributed Operating System for Lightweight Manycore Processors

Doctorate thesis presented to the Graduate Program in Informatics of the Pontifícia Universidade Católica de Minas Gerais as partial requirements for earning the PhD degree in Informatics.

Advisor: Prof. Dr. Henrique Cota de Freitas (PUC Minas)

Advisor: Prof. Dr. Jean-François Méhaut (UGA)

Belo Horizonte
2021

Pedro Henrique de Mello Morado Penna

NANVIX
A Distributed Operating System for Lightweight Manycore Processors

Doctorate thesis presented to the Graduate Program in Informatics of the Pontifícia Universidade Católica de Minas Gerais as partial requirements for earning the PhD degree in Informatics.

Prof. Dr. Jean-François Méhaut – UGA (Advisor)

Prof. Dr. Henrique Freitas – PUC Minas (Advisor)

Prof. Dr. Abdoulaye Gamatié – LIRMM (Examining Committee)

Prof. Dr. Pierre Sens – Sorbonne Université (Examining Committee)

Prof. Rodolfo Azevedo – UNICAMP (Examining Committee)

Dr. Benoît Dinechin – Kalray Inc (Examining Committee)

Prof. Dr. Carlos Augusto Martins – PUC Minas (Examining Committee)

Prof. Dr. François Broquedis – UGA (Examining Committee)

Prof. Dr. Márcio Castro – UFSC (Examining Committee)

Belo Horizonte
2021

ABSTRACT

Lightweight Manycore (LW Manycore) processors were introduced to deliver performance scalability with low-power consumption. To address the former aspect, they rely on specific architectural characteristics, such as a distributed memory architecture and a rich Network-on-Chip (NoC). To achieve low-power consumption, they are built with simple low-power Multiple Instruction Multiple Data (MIMD) cores, they have a memory system based on Scratchpad Memories (SPM) and they exploit heterogeneity by featuring cores with different capabilities. Some industry-successful examples of these processors are the Kalray MPPA-256, the PULP and the Sunway SW26010. While this unique set of architectural features grant to LW Manycore performance scalability and energy efficiency, they also introduce multiple challenges in software programmability and portability. First, the *high density circuit integration* turns dark silicon into reality. Second, the *distributed memory architecture* requires data to be explicitly fetched/offloaded from remote memories to local ones. Third, the *small amount of on-chip memory* forces the software to partition its working data set into chunks and decide which of them should be kept local and which should be offloaded to remote memory. Fourth, the *on-chip interconnect* invites software engineers to embrace a message-passing programming model. Finally, the *on-chip heterogeneity* makes the deployment of applications complex. One approach for addressing these challenges is by means of an Operating System (OS). This type of solution craves to bridge intricacies of an architecture, by exposing rich abstractions and programming interfaces, as well as handling resource allocation, sharing and multiplexing. Unfortunately, existing OSes struggle to fully address programmability and portability challenges in LW Manycore, because they were not designed to cope with architectural features of these processors. In this context, the main goal of this work boils down to propose a novel OS for LW Manycore that specifically copes with these uncovered challenges. The main contribution of this work is a distributed OS that advances the resource management in LW Manycore processors. On the one hand, from the scientific perspective this main contribution may be unfolded in three specific contributions. First, a comprehensive Hardware Abstraction Layer (HAL) that makes the development and deployment of a fully-featured OS for LW Manycore easier, as well as it enables the portability of an OS across multiple of these processors. Second, a rich memory management approach that is based on Distributed Paging System (DPS). This is a novel system-level solution that we devised for managing memory of a LW Manycore. Third, a lightweight communication facility that manages the on-chip interconnect and exposes primitives with hardware channel multiplexing. On the other hand, as a technical contribution, this work introduces Nanvix. This is a concrete implementation of an OS for LW Manycore processor that features the aforementioned scientific advancements. Nanvix supports multiple architectures (Bostan, x86, OpenRISC, ARMv8 and RISC-V), runs on baremetal processors, exposes rich abstractions and high-level programming interfaces.

Keywords: Distributed Operating Systems. Lightweight Manycore Processors.

RESUMO

Processadores LW Manycore foram introduzidos para fornecer escalabilidade de desempenho a um baixo consumo de energia. Para abordar o primeiro aspecto, eles contam com características arquiteturais específicas, como uma arquitetura de memória distribuída e um rede-em-chip. Para atingir baixo consumo de energia, esses processadores são construídos com núcleos simples e de baixo consumo de energia, têm um sistema de memória baseado em SPM e exploram heterogeneidade. Alguns exemplos de sucesso desses processadores são o Kalray MPPA-256, o PULP e o Sunway SW26010. Embora este conjunto único de recursos arquiteturais conceda a LW Manycore desempenho escalabilidade e eficiência energética, eles também apresentam vários desafios na programação e portabilidade do software. Primeiro, a *integração do circuito de alta densidade* transforma o problema de *dark silicon* em realidade. Em segundo lugar, a *arquitetura de memória distribuída* requer que os dados sejam explicitamente buscados / descarregados de memórias remotas para memórias locais. Terceiro, a *pequena quantidade de memória on-chip* força o software a particionar seu conjunto de dados de trabalho em blocos e decidir qual deles deve ser mantido local e deve ser descarregado para a memória remota. Quarto, a *interconexão on-chip* convida engenheiros de software a abraçar um modelo de programação de passagem de mensagens. Finalmente, a *heterogeneidade no chip* torna complexa a implantação de aplicações. Uma abordagem para enfrentar esses desafios é por meio de um sistema operacional (SO). Esse tipo de solução anseia por superar as complexidades de uma arquitetura, por expor abstrações ricas e interfaces de programação, bem como manipular alocação, compartilhamento e multiplexação de recursos. Infelizmente, os sistemas existentes lutam para resolver totalmente a programabilidade e desafios de portabilidade em LW Manycore, porque eles não foram projetados para lidar com características arquitetônicas desses processadores. Nesse contexto, o principal objetivo deste trabalho se resume em propor um SO para LW Manycore que lida especificamente com esses desafios. A principal contribuição deste trabalho é um SO distribuído que avança o gerenciamento de recursos em processadores LW Manycore. Por um lado, do ponto de vista científico, esta contribuição principal pode ser desdobrada em três contribuições específicas. Em primeiro lugar, uma HAL abrangente que torna o desenvolvimento e implantação de um SO totalmente caracterizado para LW Manycore mais fácil, bem como permite a portabilidade de um SO em vários desses processadores. Em segundo lugar, uma abordagem de gerenciamento de memória rica que se baseia em DPS,uma nova solução em nível de sistema que criamos para gerenciar a memória de um LW Manycore. Terceiro, uma facilidade de comunicação leve que gerencia o on-chip interconecta e expõe primitivas com multiplexação de canal de hardware. Por outro lado, como contribuição técnica, este trabalho apresenta o Nanvix. Esta é uma implementação concreta de um processador SO para LW Manycore que apresenta os avanços científicos mencionados. Nanvix suporta múltiplos arquiteturas é executado em processadores baremetal, expõe abstrações ricas e programação de alto nível interfaces.

Palavras-chave: Sistemas Operacionais Distribuídos. Processadores Manycore leves.

LIST OF FIGURES

LIST OF TABLES

TABLE OF CONTENTS

1 INTRODUCTION

The scientific and industry communities are constantly seeking and developing solutions to address the ever-increasing performance demands of software applications. These solutions are often tailored to particular intricacies of a target application and platform, and they range from the software-level and compiler support (KAMIL et al., 2010), to the runtime system (BROQUEDIS et al., 2010) and the underlying hardware architecture (DINECHIN et al., 2013). In this way, specialized techniques may be applied and thus cutting-edge performance achieved.

On the one hand, in software-level, programming techniques are employed to optimally handle domain-specific properties (FRANCESQUINI et al., 2015). For instance, in graph-traversal applications, fine-grain locking schemes are widely employed to enable maximum concurrency and parallelism to the graph itself (MERRILL et al., 2012). Alternatively, in N-Body Simulations, density-guided domain decomposition is often used to minimize load imbalance, exploit data locality and enable faster simulation algorithms to be applied (BARNES; HUT, 1986). On the other hand, at compile time, code optimizations are employed, so that execution time, memory footprint and power usage are tuned. Some techniques such as Instruction Scheduling and Auto-Vectorization are architecture dependent and thus are required to be employed for extracting maximum performance from the hardware. In contrast, other techniques such as Register Allocation and Loop Unrolling are architecture independent, but they may significantly increase instruction throughput in a portable fashion.

Performance improvements in the runtime system level target challenges on the interactions between the application and the underlying platform. For instance, to overcome communication overheads, thread mapping heuristics (CRUZ et al., 2014), data placement techniques (DIENER et al., 2016) and asynchronous communication primitives (PASCOLO et al., 2016) are often employed. In this way, communication latencies between threads may be mitigated and overlapped with computation. Alternatively, to address processing heterogeneity and inherent load imbalance, runtime systems heavily rely on scheduling strategies (AUGONNET et al., 2011). Most often, these features are exported through a framework or an Application Programming Interface (API), and hence may be seamlessly invoked by applications (GROPP et al., 1996; DAGUM; MENON, 1998).

Finally, in hardware-level, tailored approaches are adopted to push even further performance bounds. To improve instruction throughout in a single core, techniques such as Instruction Pipelining, Superescalar Out of Order (OoO) execution and Very Long Instruction Word (VLIW) are widely used. Alternatively, thread-level parallelism and code acceleration are tackled with application-specific architectural solutions, such as multicore and manycore processors, Graphics Processing Unit (GPU), and more recently manycores and Field Programmable Gate Array (FPGA). Finally, there are solutions such as distributed memory architecture and Network-on-Chip (NoC) to enable a massive scalability on the number of cores in a single chip.

Specifically on the hardware-level efforts, Lightweight Manycore (LW Manycore) processors (ISHII et al., 2017) emerged as a promising alternative to deliver performance scalability with low-power consumption. To enable these features, these processors rely on a selected set of architectural characteristics:

- They are designed to cope with Multiple Instruction Multiple Data (MIMD) workloads.
- They integrate up to thousands of low-power cores in a single chip.
- They have their cores disposed in tightly-coupled groups called clusters.
- They feature a distributed memory architecture with multiple address spaces.
- They present a constrained memory, with small (few kB per core) and private local memories.
- They rely on NoC for fast and reliable message-passing communication.
- They have a heterogeneous configuration in terms of I/O and/or computing capabilities.

Some industry-successful examples of LW Manycore are the Kalray MPPA-256 (DINECHIN et al., 2013), the PULP (ROSSI et al., 2017), and the Sunway SW26010 (ZHENG et al., 2015), being the latter employed in Sunway TaihuLight (FU et al., 2016), currently the fourth most powerful supercomputer in the world, according to TOP500 raking[1].

1.1 Motivation

While a unique set of architectural features grant to LW Manycore high-performance and energy efficiency, they also introduce multiple challenges in programmability and software portability. First, the *high density circuit integration* turns dark silicon into reality (HAGHBAYAN et al., 2017). If all the cores of such type of a processor are powered on at the same time, the output heat cannot be effectively dissipated and the hardware may malfunction. Therefore to tackle this physical barrier, software is required to implement thermal-aware scheduling strategies.

Second, the *distributed memory architecture* leads to a non-trivial software design. To access and manipulate data across multiple physical address spaces, software should explicitly fetch/offload data from remote memories to local ones (FRANCESQUINI et al., 2015). Furthermore, the *small amount of on-chip memory* requires the software to partition its working data set into chunks and decide which chunks should be kept local and which should be offloaded to memory, at a given time. Furthermore, to achieve this efficiently, software should handle by its own data caching, replication and prefetching (CASTRO et al., 2016).

Third, the *on-chip interconnect* invites engineers to embrace a message-passing programming model and thus work on significant refactor on their software (SOUZA et al., 2017). Additionally, hardware exposes mechanisms for asynchronous programming and explicit message

[1] Available at: <https://www.top500.org>.

routing, which engineers should also handle. The former is a fundamental requirement when aiming at communication and computation overlapping (HASCOËT et al., 2017). The latter should be strongly considered in order to guarantee uniform communication latencies (DINECHIN et al., 2014).

Finally, the *on-chip heterogeneity* makes the deployment of applications in LW Manycore complex (BARBALACE et al., 2015). On the one hand, clusters of LW Manycore may feature different processing capabilities, and thus processor allocation should account for this factor, in order to extract the most of hardware. Allocating a cluster to a process that will not make use of all the hardware features that are available may negatively impact the overall system performance. On the other hand, clusters may also differ in respect to their connectivity to external devices, and thus application placement should also consider this aspect.

1.2 Problem

There are two main approaches for addressing software programmability and portability challenges in an architecture: baremetal runtime systems and Operating Systems (OSes). The first aims to expose a programming environment of top of the hardware to address the needs of a specific architecture (HO et al., 2015; Mohr; Tradowsky, 2017). In contrast, the second approach craves to bridge intricacies of the hardware in a broader sense, by exposing rich abstractions and programming interfaces, as well as handling resource allocation, sharing and multiplexing.

In the context of LW Manycore processors, unfortunately as of today, neither of the two previous approaches provide a complete solution to tackle programmability and portability challenges. On the one hand, existing baremetal runtime systems that aim these emerging processors (VARGHESE et al., 2014; RICHIE; ROSS; INFANTOLINO, 2017) inherently lack on covering all the issues that we stated in previously (Section 1.1). For instance, some runtime systems do provide rich abstractions to tackle on-chip communication (DINECHIN et al., 2013; AUBRY et al., 2013; HO et al., 2015), but overlook challenges that concern the distributed memory architecture. On the other hand, existing OSes do not to cope with architectural intricacies of LW Manycore as follows.

a) At the one end of the spectrum, centralized OSes intrinsically face important barriers:

   - They have scalability problems (WENTZLAFF; AGARWAL, 2009).
   - They do not cope with increasingly diverse hardware (BARBALACE et al., 2015).
   - They are not feasible for the context due to the small amount of on-chip memory.

b) On the other end of the spectrum, distributed OSes miss some architectural constraints of LW Manycore processors (PENNA et al., 2021), and thus it is not possible to deploy these OSes in the aforementioned architectures without a complete redesign and/or significant changes to their source code. The rationale for this lies in the following points:

- They do not rely on a decentralized view of the hardware. For instance, they assume that hardware resources are visible to all cores of the underlying processor.
- Although they do feature a distributed software organization, they do not cope with the multiple physical address spaces. They rely on shared memory areas to either transfer software messages and move data from one core to another.
- They do not have as one of their primary constraints their memory footprint, and thus inherently do not cope with small on-chip memories of LW Manycore.
- They overlook rich the on-chip interconnects and do not exploit additional features that are often available in LW Manycore, such as asynchronous message passing through Direct Memory Access (DMA) and message routing.

Based on these observations, we formulate the following problem, which remains unanswered by the state-of-the-art in Operating Systems:

> *How to design an operating system so as to address programmability and portability issues in lightweight manycore processors?*

1.3 Goals

Seeking for an answer to the previously stated problem, the main goal of this thesis boils down to *bridge architectural characteristics of lightweight manycore processors*. To this end, we propose a novel distributed OS that specifically aims the following:

- *Programmability*: to provide rich system abstractions for LW Manycore, such as process, files, virtual memory and communication primitives; as well as to enable resource sharing, multiplexing and isolation in these emerging architectures.
- *Portability*: to expose standard programming interfaces so as to enable existing software to be ported to LW Manycore processors.

1.4 Contributions

The main contribution of this thesis is a distributed OS that advances the resource management in LW Manycore processors. On the one hand, from the scientific perspective this main contribution may be unfolded in following specific contributions:

- A comprehensive Hardware Abstraction Layer (HAL) for LW Manycore (PENNA; FRANCIS; SOUTO, 2019) that: (i) bridges architectural differences across multiple architectures; and (ii) cope with key issues that are often encountered when designing an OS for these processors. With this abstraction layer, the development and deployment of a fully-featured OS becomes easier not only to a particular LW Manycore, but also the portability of an OS itself across multiple of these processors.

- A rich memory management approach that is based on a Distributed Paging System (DPS). This is a novel solution that we devised for managing memory of a LW Manycore processor and it works as follows. The local memories of a LW Manycore processor are considered page caches that can store data of whichever processes. Based on this, the OS manages this page cache by: (i) placing those pages that are being heavily used by a process in local memories; and (ii) swapping out those pages that are not used by a process to remote memories. Overall, this solution enables transparent data accessing and manipulation, as well as data sharing and coherence in LW Manycore. We believe that these issues are importantly related to programmability and portability challenges that we aim at overcoming.

- A lightweight communication facility that manages the on-chip interconnect and exposes rich primitives (PENNA et al., 2021). Overall, this facility exposes three communication abstractions: (i) syncs, for enabling a process to signal and unlock another process remotely; (ii) mailboxes, for sending fixed-size messages with low latency; and (iii) portals, for handling dense data transfers with high bandwidth. This communication infra-structure effectively enables the efficient implementation of our OS.

On the other hand, we highlight that this thesis also introduces an important technical contribution to the community:

- Nanvix, a concrete implementation of an OS that features the aforementioned scientific advancements for LW Manycore processors. Nanvix supports multiple architectures (i.e., k1b, x86, OpenRISC, ARMv8 and RISC-V), runs on silicon LW Manycore processors, as well as it exposes rich abstractions and high-level programming interfaces.

1.5 Thesis Outline

The remaining chapters of this thesis are organized as follows:

- Chapter 2 reviews the background of this work. We cover single-chip many core processors and precisely highlight the unique features of LW Manycore processors. We present an overview of the main aspects related to the design of OSes. We discuss about principles and problems in distributed systems.

- Chapter 3 discusses related work. We discuss about runtime systems and how they improve programmability by exposing API for narrowed domains. We present existing OSes for manycore processors and how they leverage software development in these architectures. We argue on what points these solutions lack for answering the problem stated in this thesis, and we summarize why our work advances the state-of-the-art on the programmability and portability of LW Manycore.

- Chapter 4 details the design of Nanvix and discusses how it addresses the programmability and portability challenges of LW Manycore.

- Chapter 5 presents the evaluation methodology that we employed to evaluate our OS. We relied on an experimental-based approach based on synthetic programs, system utilities and scientific programs.
- Chapter 6 presents and discuss the experimental results that we obtained. We evaluated several aspects of Nanvix in Kalray MPPA-256, a 288-core silicon LW Manycore processor.
- Chapter 7 presents our conclusions and perspectives for future works. We highlight the contributions of this thesis, as well as we stress three main follow-up research topics: kernel bypassing, process migration and hardware/software co-design.
- Appendix A presents an overview discussion on the platforms that are currently supported by Nanvix. We give insights on architectural intricacies of each of these platforms and how Nanvix was designed to cope with this challenges.

2 BACKGROUND

In this chapter we cover the foundations of our thesis. First, we take an in-depth look on single-chip manycore architectures and their main characteristics (Section 2.1). Next, we introduce runtime systems (Section 2.2). Then, we discuss about OSes (Section 2.3) and brief about distributed systems (Section 2.4). Finally, we conclude this chapter by positioning our work on this ground (Section 2.5).

2.1 Single-Chip Manycore Processors

During the past decades, performance improvements of computing platforms were mostly driven by techniques that aimed to increase instruction throughput in a single Central Processing Unit (CPU) (FRANCESQUINI et al., 2015). On the microarchitecture level, superescalar pipelines and speculative execution heuristics were employed to boost execution streaming. Conversely, on the electronic level, operating frequency increase was applied to effectively accelerate instruction execution. Unfortunately however, these techniques rapidly reached a scalability barrier on the material level known as the *Power Wall* (CASTRO et al., 2016). This problem comes down to the fact that instruction throughput in a single CPU cannot increase indefinitely because the output heat cannot be dissipated out of the circuit. As a side effect, the temperature of the processor sharply increases thus causing it to first malfunction and eventually get damaged.

Parallel processors were introduced to overcome this physical barrier. Instead of turning up instruction throughput in a single CPU, the alternative idea was to increase the actual number of cores in a CPU. This way, multiple execution streams could run at the same time, and thread level parallelism could be exploited to achieve performance scalability in an application. In this context, a well known categorization was proposed by Flynn (FLYNN, 1972), in which architectures may be grouped according to their instruction and data streams:

- Single Instruction Single Data (SISD): a single processor executes a single instruction stream to operate on data stored in a single memory.
- Single Instruction Multiple Data (SIMD): a single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data memory, so that each instruction is executed on a different set of data by the different processors.
- Multiple Instruction Single Data (MISD): a sequence of data is issued to a set of processors, each of which executes a different instruction sequence.
- MIMD: a set of processors simultaneously execute different instruction sequences on different data sets.

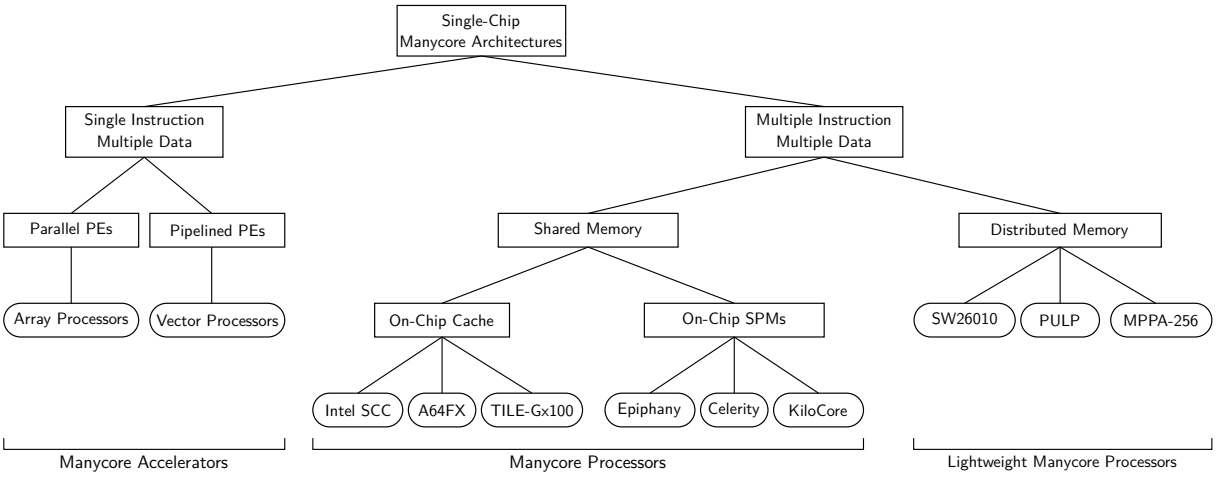


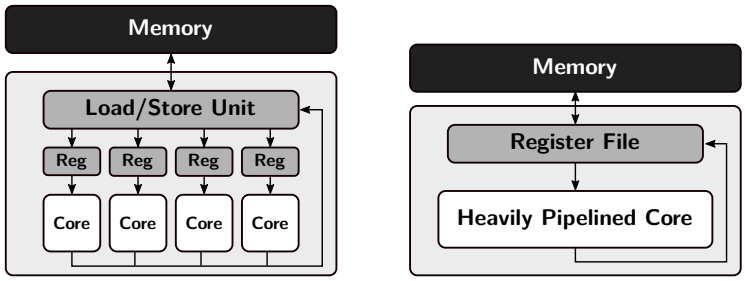
Figure 1 – An extended taxonomy for single-chip manycore architectures.

Despite the fact of being a solid taxonomy, with the recent increase of cores in architectures, one could also argue on extra architectural characteristics that may be accounted to categorize single-chip manycore processors. This extended classification is outlined in Figure 1 and it adds to Flynn's approach aspects concerning the memory system of single-chip manycore architectures. Based on this additional characteristic, we highlight three organizations that are relevant to the understanding of this thesis: *manycore accelerators*, *manycore processors* and *LW Manycore processors*. In the next sections, we present and discuss each of these classes.

2.1.1 Manycore Accelerators

Manycore accelerators were the first class of single-chip manycore processors to arise. They comprise those architectures that are narrowed for specific workloads, such as vector processing, computer vision and artificial intelligence. Overall, manycore accelerators often feature a SIMD organization and for this reason are often coupled to a general purpose processor to construct a heterogeneous system. Among manycore accelerators, we highlight two subclasses that differ in the way cores are organized: *array processors* and *vector processors*. These two approaches are outlined in Figure 2 and discussed next.

Array processors have several cores attached to a shared control unit and a common wide register file (a.k.a. vector register). All cores have the same internal configuration, but each of them operate on a different subword of the vector register. Figure 2a shows the execution flow for a set of instructions in an array processor. In this example, there are four different cores executing one instruction on a vector register file. At each clock cycle, all cores execute the same instruction, but each of them operate on a different range of the vector register. Array processors have the major advantage of being easy to implement, but at the same time, they face a limitation on the bit-width of operands. Typically, array processors are embedded in the same package of general purpose architectures, to accelerate some specific workloads such as those from multimedia domain (PELEG; WEISER, 1996). Some examples of standalone manycore accelerators of this subclass are discussed by (ISHIGAKI et al., 2015) and Shi (SHI et al., 2014).



(a) Array processor. (b) Vector processor.

Figure 2 – Manycore accelerators.

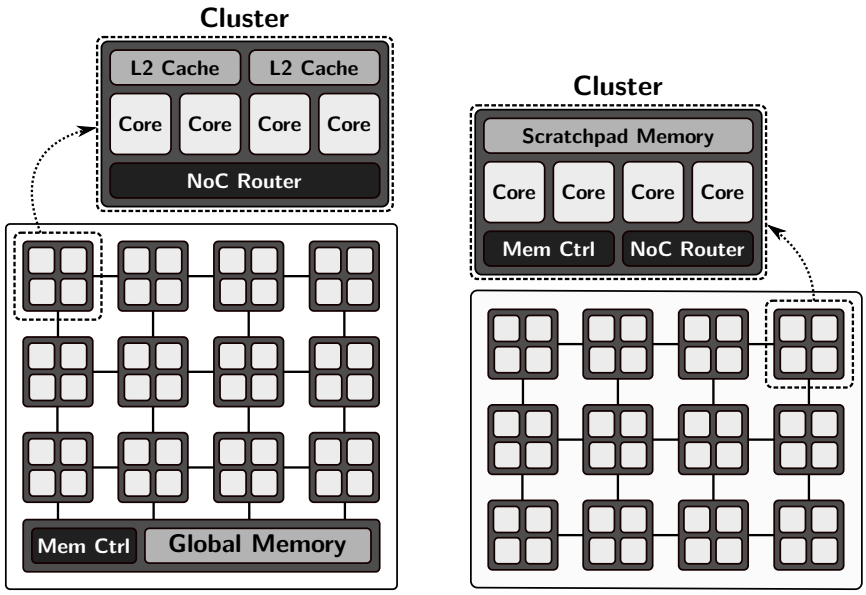
Vector processors take an orthogonal approach to array processors. Instead of implementing SIMD instructions with an array of identical cores and a vector register, vector processors bundle in the same chip several stream cores, each of which featuring several specialized functional units that run micro-operations on a vector throughout multiple consecutive cycles. With this alternative organization, the length of vector operands not only may be arbitrarily large, but also may they dynamically change. The execution flow in a vector processor is presented in Figure 2b. Some examples of vector processors are Nvidia Pascal (FOLEY; DANSKIN, 2017) and ARA (CAVALCANTE et al., 2020)

2.1.2 Manycore Processors

Manycore processors address the general-purpose High Performance Computing (HPC) domain and are often used as basic building blocks for supercomputers. These processors feature a shared memory system in which all cores may transparently read/write using common load/store instructions. Ultimately, manycore processors do not require great programming paradigm shifts and thus are currently the most known class among single-chip manycore architectures. Some examples of manycore processors are Intel Single-Cloud Computer (HOWARD et al., 2011), Tilera TILE-Gx100 (RAMEY, 2011), Adapteva Epiphany (OLOFSSON; NORDSTROM; ULABDIN, 2014), Celerity (DAVIDSON et al., 2018), Fujitsy A64FX (YOSHIDA, 2018) and Manticore (ZARUBA; SCHUIKI; BENINI, 2020).

Specifically concerning manycore processors, we highlight two different organizations that are often embraced. The first one is presented in Figure 3a. In this organization, cores are physically grouped in physical units called clusters (or tiles), which are altogether interconnected with the global shared memory system through a NoC. Cores in the same cluster are tightly coupled to a local memory controller, a local cache and a NoC router. Memory controllers enable transparent access to the global shared memory by forwarding load/store requests through the NoC. Furthermore, these memory controllers also provide cache coherence support, by either implementing it in hardware or exposing special instructions that enable this in software-level.

Figure 3b outlines a second organization for a manycore processor. Likewise in the previous approach, in this one cores are grouped into clusters which are interconnected through a NoC. Differently however, the memory system does not rely on a caching scheme. Instead, cores



(a) Cache-based. (b) SPM-based.

Figure 3 – Manycore processors.

in the same cluster share access to a local Scratchpad Memory (SPM) module. Moreover, the global address space is this statically partitioned across the several SPM and memory controllers forward remote memory accesses to the concerning SPM module.

Both organizations have their own advantages and disadvantages. On the one hand, cache-based manycore processors often feature larger physical address spaces than their counterparts. This is because memory modules are disposed outside the chip, and thus they may be as large as possible. However, as drawback, cache coherence becomes the main performance bottleneck in the system, once the number of cores in this type of design increases. On the other hand, SPM-based manycore processors do not face scalability issues concerning coherency traffic, but they do have a much more limited address space, once all memory modules are disposed on-chip.

2.1.3 Lightweight Manycore Processors

LW Manycore processors have an endeavor to deliver high performance with energy efficiency. To this end, they mostly rely on a distinguished set of architectural features. Overall, LW Manycore can be seen as “clusters on a chip” and present the following main characteristics:

- They are designed to cope with MIMD workloads.
- They integrate up to thousands of low-power cores in a single chip.
- They have their cores disposed in tightly-coupled groups called clusters.
- They feature a distributed memory architecture with multiple address spaces.
- They present a constrained memory, with small (few kB per core) and private local memories.

- They rely on NoC for fast and reliable message-passing communication.
- They have a heterogeneous configuration in terms of I/O and/or computing capabilities.

To exemplify these features, we consider a discussion on a concept processor (Figure 4). Noteworthy, this narrative applies to existing LW Manycore, such as PULP (ROSSI et al., 2017), Kalray MPPA-256 (DINECHIN et al., 2013) and Sunway SW26010 (ZHENG et al., 2015). This concept processor integrates 67 cores grouped into 17 clusters. In a cluster, cores share some local hardware resources, such as a Static Random Access Memory (SRAM) and NoC interfaces, and have uniform access latencies to these components. Clusters may have different characteristics, like processing power, local memory sizes, and communication capabilities, to better address different computing needs. For instance, in this concept processor, there are 16 Compute Clusters having 4 cores each and that are meant for computing workloads. In contrast, the I/O Cluster features connectivity to external devices as well as two NoC interfaces, thereby being intended to handle IO workloads.

Clusters have different address spaces, and they may communicate with one another by explicitly exchanging hardware-level messages through the NoC. Therefore, to write to external devices and the Dynamic Random Access Memory (DRAM) attached to the I/O Cluster, Compute Clusters must tile in software the output data into hardware-level messages and transfer them through the NoC to the I/O Cluster. Conversely, if two threads running on different Compute Cluster want to exchange information, they should do the same.

So far in this chapter, we discussed about sing-chip manycore processors and highlighted how LW Manycore processors are positioned in this context. In the next sections, we unveil software-related aspects that concern this thesis. First, we present an overview of runtime systems for parallel and distributed programming, which are often found in LW Manycore. Then, we cover the background on OSes and distributed systems, which we believe that are the key to enable rich resource management of LW Manycore. Finally, in the end of this chapter, we conclude about the aspects of OSes and distributed systems that are relevant for addressing portability and programmability in LW Manycore.

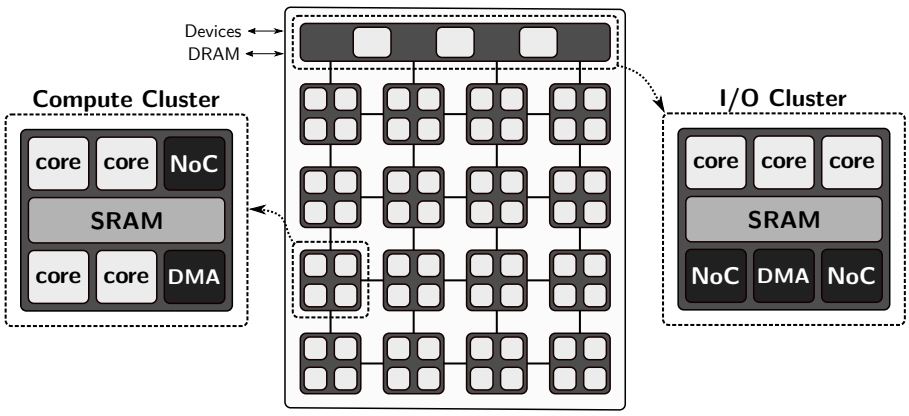


Figure 4 – A LW Manycore processor with 67 cores.

2.2 Runtime Systems for Parallel and Distributed Programming

Runtime systems provide rich programming environments for a particular execution model. To accomplish this, they expose specific API to applications; and they may interact directly with the hardware, OS and/or compiler. Some well-known examples of runtime systems are the crt0, the Android Runtime Environment (ART), and the Java Runtime Environment (JRE). These systems primarily enable a general-purpose programming environment for applications, by exposing routines and abstractions for manipulating resources such as data streams, software signals/exceptions and heap memory. Notwithstanding, in the context of this work, we next discuss about runtime systems that focus on manycore processors. Overall, the existing alternatives mostly concern parallel and distributed programming frameworks.

Parallel programming environments expose an infrastructure for applications to create, manipulate and synchronize multiple execution flows within a program. This way, an application may thoroughly exploit the performance that is provided by a parallel architecture. We highlight three environments that are often available for manycore architectures: POSIX Threads (PThreads), OpenMP (DAGUM; MENON, 1998) and OpenCL (MUNSHI, 2009). The first encapsulates an execution flow in an abstraction called *thread*, and it exposes a set of routines that enable applications to manipulate threads. The PThreads environment enables maximum flexibility in parallel programming and are typically available as a statically-linked library. Next, the OpenMP runtime focus on a transparent support for the fork-join parallel programming model. To this end, the runtime system interacts with the compiler infrastructure to enable the software engineer to annotate the source code in order to create parallel regions. Then, when building the application from an annotated source code, the compiler links low-level routines to spawn and terminate threads at the begin and end of a parallel region, respectively. Finally, the OpenCL environment aims at parallel programming in heterogeneous architectures. It defines a structure for writing programs where the application kernel executes in the OpenCL device, whereas the remainder portion of the code executes in the host device. To accomplish this, OpenCL works with the compiler infrastructure to generate code for both architectures, as well as it interacts with the hardware/OS to download/offload binary and data from/to the devices.

Distributed programming environments enable a complete exploitation of distributed architectures by focusing on providing communication abstractions between multiple computing nodes in a platform. We note two distributed programming environments that are typically encountered in manycore architectures: Message Passing Interface (MPI) (GROPP et al., 1996) and Partitioned Global Address Space (PGAS) (De Wael et al., 2015). The first exposes a set of routines that enable arbitrary data to be send/received using abstractions called messages. Messages can have a single or multiple recipients and may be send/received synchronously and/or asynchronously. On the other hand, PGAS enable one-sided communication by maintaining a global address space that is partitioned and spread across the several peers of the application. Peers may read/write data to this address space though special functions put/get, and the runtime system ensures that coherence is maintained.

2.3 Operating Systems

An Operating System (OS) is a software layer that lies on top of baremetal hardware, manages computing resources and provides common services for programs. Without an OS, engineers would have to handle hardware intricacies by their own, thereby increasing software complexity and development cost. An OS has two important roles in a computing platform:

(i) Abstract Resources: extend the functionalities of the hardware and expose to user programs services and rich API that are easier to deal with.

(ii) Manage Resources: multiplex access to hardware and software resources, ensuring data protection and integrity, as well as enforcing administration protocols and policies.

In this section, we take an in-depth look on the fundamentals of OSes. First, we present an overview on the main components of an OS (Section 2.3.1). Then, we discuss about the internals of each of these component (Section 2.3.2 to Section 2.3.5).

2.3.1 Overview

Figure 60 presents a schematic of an OS. It is composed by multiple components that interact with each other to provide rich resource abstraction and management. Noteworthy, there are many ways to couple these components. Therefore, in the following paragraphs we focus on discussing their roles, and in Section 2.3.2, we give insights on how they may be tighten together.

The HAL is the lowest-level component of an OS. It directly interacts with the hardware and exposes a common API for dealing with baremetal structures, such as interrupt vectors and Memory Management Unit (MMU) tables. The main role of this component is to hide as much as possible all hardware intricacies, as well as to provide a standard interface across multiple architectures. This way, the overlying OS components become less complex and easier to be ported from one hardware to another.

The Memory Management Subystem (MMS) provides a view of the underlying memory system. It typically does so by having two modules working together: the swapping and virtual memory modules. The first deals with paging, keeping in memory those pages that are more frequently used, and swapping out to secondary storage those that are not. The virtual memory system, on the other hand, relies on the paging module to enable advanced features such as shared memory regions, on-demand loading, lazy copying and page pinning. We further discuss the internals of a MMS in Section 2.3.3.

The Process Management Subsystem (PMS) handles all tasks that involves programs. To do this, it relies on an abstraction called process, which encapsulates the execution flow and the set of resources used by a program. Overall, the PMS exposes routines for creating, terminating and scheduling processes, as well as abstractions that allow processes to exchange information with one another and synchronize their activities. We present a detailed discussion of a PMS in Section 2.3.4.

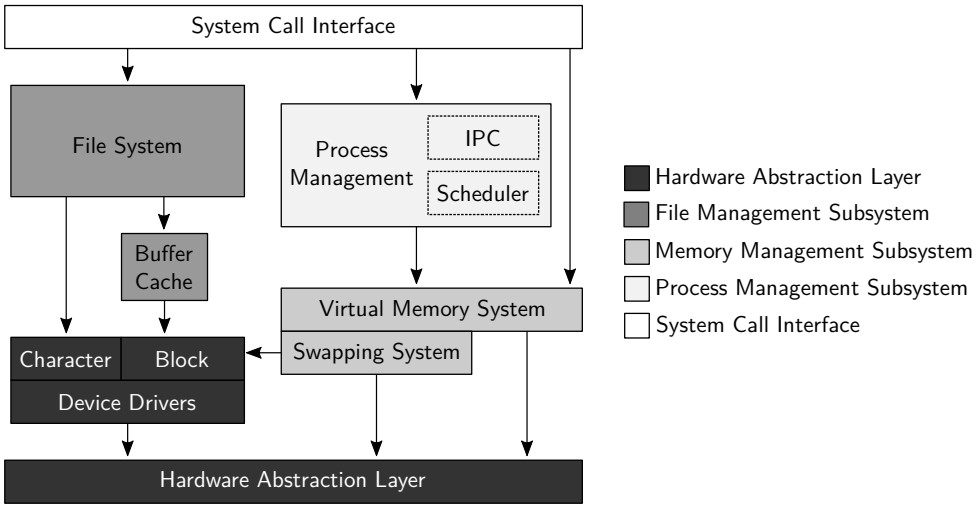


Figure 5 – A possible architecture for an OS.

The File Management Subsystem (FSM) provides a uniform interface for programs to store and retrieve data from secondary data storage, as well as to manipulate hardware devices and software resources. To accomplish this, it creates and maintains the file abstraction. Files may be organized into directories, can be accessed through a unique pathname, and may be shared among several processes transparently. We discuss about this component in Section 2.3.5.

The system libraries lie on top of all the components and expose to user programs a collection of routines known as system calls. This is a well-defined interface in which user programs rely on to interact with the system. Overall, system libraries do not do much work than sanity checking parameters and ensuring protection. Typically, user libraries wrap around system libraries to provide interfaces and abstractions that are even more friendly to software engineers.

2.3.2 Operating System Construction

In the previous section we presented a possible structure for an OS, but indeed such systems may be constructed in several different ways. In a nutshell, the structure of an OS is defined by the way in which their different subsystems and components are disposed and interact. Each structure has its own advantages and disadvantages, and in the next paragraphs we cover the major aspects of the most popular variants.

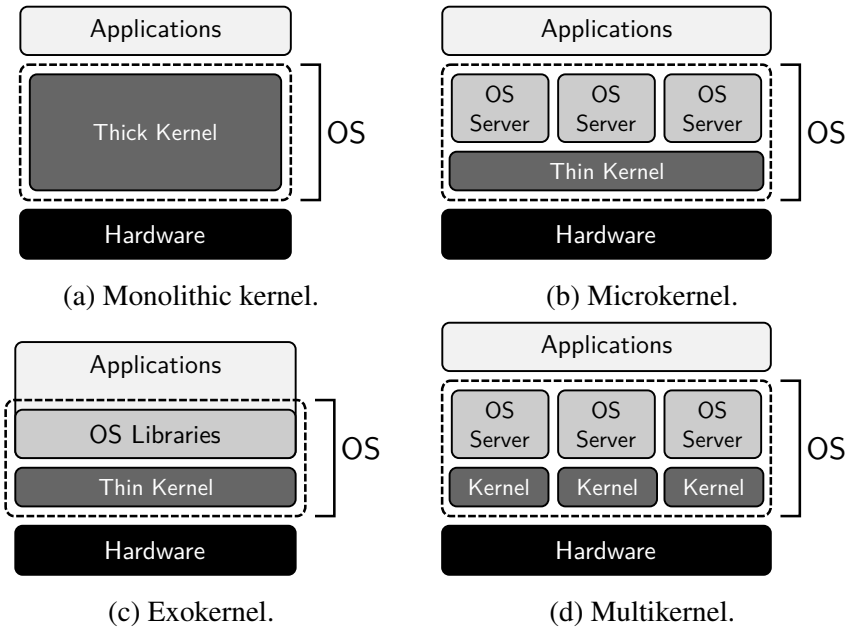
In a monolithic kernel OS (Figure 6a), the system is built as a single component that provides most system functionalities and runs entirely in privileged mode. Subsystems are implemented as a collection of procedures and data structures which are linked altogether into a single and large executable binary. The major advantage of a monolithic kernel system lies in the simplicity of its implementation. However, since it is constructed as a single unit it faces from both poor fault tolerance and scalability problems (BOYD-WICKIZER et al., 2010). This approach is typically embraced by commodity OSes, such as GNU/Linux, BSD and Windows.

In contrast to monolithic kernels, microkernel OSes factor the system in multiple components, as it is outlined in Figure 6b. On the one hand, there is a kernel component that

runs in privileged mode and ships bare minimum primitives for manipulating hardware resources (LIEDTKE, 1995). On the other hand, system servers run in unprivileged mode as stand alone programs and provide rich system functionalities, like process, memory and device management. A microkernel OS is challenging to implement, but it has two important advantages. First, it enforces a modular system structure, thereby improving fault tolerance and scalability. Second, it is flexible, because services may be narrowed to specific purposes, as well as multiple API and policies may coexist in the system. Examples of such OSes are L4 (LIEDTKE, 1995), Tornado (GAMSA et al., 1999) and K42 (APPAVOO et al., 2005).

Exokernel OSes take one step further towards minimalism and aim at a strong separation between security and abstraction. In this design (Figure 6c), the kernel is also the only software component that runs in privileged mode, but it does nothing more than partitioning the hardware and ensuring the secure multiplexing of these partitions(ENGLER; KAASHOEK; O'TOOLE, 1995). Unprivileged components run on top of these partitions and may either be user-applications, subsystems implementing common services or entire OSes. Exokernel OSes are complex to design, because their interface should be drawn so as to be comprehensive and flexible. The upside of this design lies on the fact that it enables programs to make efficient and intelligent use of hardware resources. Examples of exokernels are MIT's Exokernel (ENGLER; KAASHOEK; O'TOOLE, 1995) and Corey (BOYD-WICKIZER et al., 2008).

Multikernel OSes embrace a fully distributed design (Figure 6d). The system is factored in a set of independent OS kernels which are deployed across the cores of the underlying processor. These kernels communicate with one another via message-passing only; and they collaboratively implement traditional OS subsystems. Noteworthy, several designs for a multikernel are possible, once each OS kernel instance may follow either of the previous designs. The design and



(a) Monolithic kernel. (b) Microkernel.

(c) Exokernel. (d) Multikernel.

Figure 6 – Popular architectures for an OS.

implementation of a multikernel OS stands outs in high complexity. However, from all the current approaches known for OS construction, it achieves higher scalability, and for this reason it is increasingly under research, in the context of multicore and manycore systems. The multikernel approach is embraced by Barrelfish (BAUMANN et al., 2009), FOS (WENTZLAFF; AGARWAL, 2009), MOSSCA (KLUGE; GERDES; UNGERER, 2014), M^3 (ASMUSSEN et al., 2016) and Nanvix (PENNA et al., 2019).

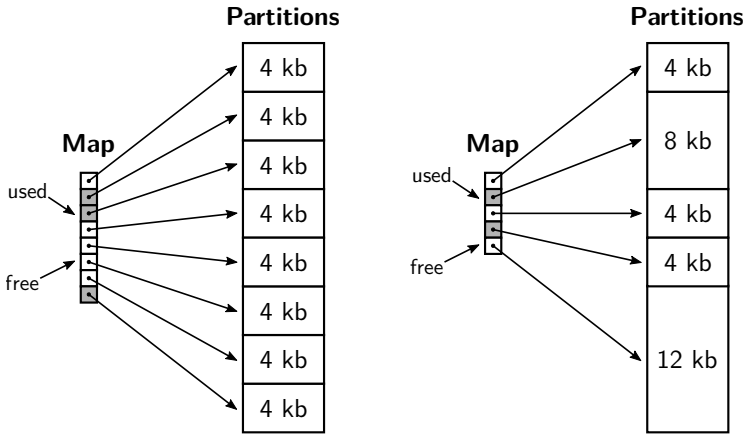
2.3.3 Memory Management

In a computing system, memory is the resource that is intended to accommodate running programs. In this context, the role of the OS is to: (i) keep track of which parts of memory are in use; (ii) allocate memory to programs; (iii) multiplex memory among programs; (iv) ensure protection on portions of memory that are owned by a program. These tasks are carried out by the Memory Management Subystem (MMS), and their efficient accomplishment is essential for the overall performance of a system. To this end, the MMS relies on a special abstraction called memory partition, which models a portion of physical memory. On top of this, the MMS provides four mechanisms: partitioning, memory allocation, virtualization and reallocation. In the next sections, we take a look on each of these mechanisms.

Partitioning

Partitioning comes down to dividing the physical memory in several sections, in order to either accommodate portions of running programs or programs themselves. Thanks to this mechanism, multiprogramming and virtualization are made possible. Overall, there are two approaches for partitioning: fixed-size partitions and dynamic partitioning. We outline these approaches in Figure 7 and detail them next.

In fixed-size partitioning, memory is statically divided into several sections that cannot grow nor shrink across execution. To keep track of partitions, the OS relies on a map data



(a) Fixed-size. (b) Variable-size.

Figure 7 – Memory partitioning techniques.

structure that records whether or not a partition is in use. Fixed-size partitioning eases memory allocation, but it introduces internal fragmentation. If a given partition is not entirely used, memory is wasted within a partition. In current architectures, fixed-size partitioning is natively supported by underlying hardware paging. Later in this section, we discuss how this technique is employed to implement paging.

In variable-size partitioning, memory is dynamically divided into several sections which can grow and shrink across execution. To keep track of variable size partitions, the OS relies on a map data structure that stores the location and size of partitions, as well as what is loaded into them. Variable size partitioning improves memory utilization, but it turns memory allocation into a more complex and expansive operation. In order to avoid external fragmentation (i.e., unused memory between partitions), the OS needs to constantly move partitions around and compact them. As a consequence, processor time is wasted. In current architectures, variable-size partitioning is natively supported by underlying hardware segments.

Allocation

Memory allocation concerns the strategy that the OS uses to allocate memory partitions. This is triggered whenever a new program is launched or when a running program requests for memory. To manage memory allocation, two approaches are commonly used: bitmaps and linked lists. These are outlined in Figure 8 and detailed in the next paragraphs.

The bitmap approach relies on an array of bits to keep track of which memory partitions are in use and which are free. With this representation, allocation of a single partition is straightforward: find the bit in the bitmap that is unset, set it to one and return the corresponding partition. Conversely, deallocation comes down to finding the corresponding bit in the bitmap and clearing it to zero. Bitmaps are suitable for managing fixed-size partitions and handling requests one at a time. In this scenario, the allocation procedure takes linear time, whereas the deallocation step takes constant time. Unfortunately however, if memory partitions have dynamic size the bitmap structure is not enough for storing the required information. Furthermore, if multiple partitions should be allocated at once, the complexity of this operation increases substantially, because the bitmap would need to be traversed several times to find a suitable place for allocation.

Memory management with linked lists is more adequate when either variable partitions

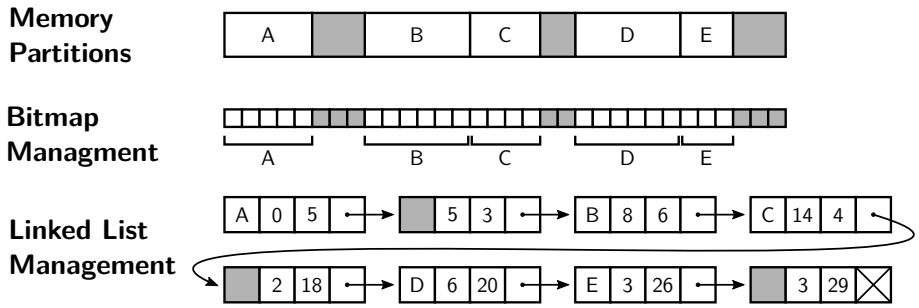


Figure 8 – Memory allocation techniques.

are in use or allocation requests have arbitrary sizes. In this alternative approach, the idea is to maintain a linked-list that stores in the information about free partitions, such as its base address and size. At system startup, this structure has a single node which starts at address zero and spans over the whole memory. Then, when an allocation is requested: (i) the list is searched for a free partition that is large enough; and (ii) the selected partition is resized accordingly.

On the other hand, when a partition is released, it is put back into the linked-list. Noteworthy, since allocation requests may have arbitrary sizes and partitions are resized during this operation, partitions that are too small are likely to appear in the linked list of free partitions. As a consequence, over the time, there will not be enough large partitions to fulfill allocation requests and the system starves. This problem is known as fragmentation and may be mitigated by coupling two orthogonal approaches. First, whenever a partition is released and is about to be placed backed in the linked list of free partitions, the memory allocator may attempt to merge unused partitions into larger ones. To this end, the system checks if either of the adjacent partitions are free, and if so it merges then. Second, when allocating partitions a suitable strategy may be employed, depending on the load characteristics in which the system is subjected. Some popular strategies are discussed below.

First-Fit The first memory partition that is large enough to fulfill the allocation request is returned. This favors performance and generates large fragments in average.

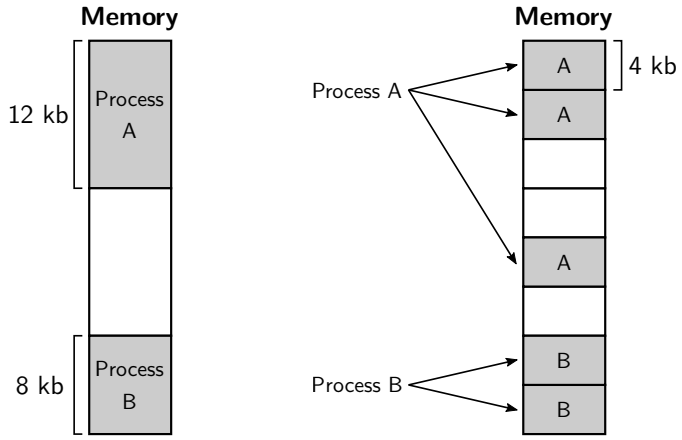
Best Fit The memory partition that best fulfills the request is returned. This has a slightly worse performance than First-Fit and generates small fragments in average.

Quick Fit Free memory partitions are hierarchically grouped into sublists according to their size. When a memory partition is requested, the concerned list is searched. Noteworthy, each sublist may have its own allocation policy. Overall, quick fit delivers the best performance but it imposes extra complexity for merging free partitions.

Virtualization

Virtualization enables programs to address more memory than there is physically available. To this end, the OS uses non-volatile storage devices to temporarily store unused memory portions of programs. There are two main approaches for virtualization, namely swapping and virtual memory. Both are outlined in Figure 9 and discussed in the next paragraphs.

In swapping, programs are placed in their entirety in physical memory and swapped back and forth from/to non-volatile storage. When a new program is launched, the OS first scans the physical memory for some unused partition that is large enough to accommodate the program. If such a partition can be promptly allocated, the new program is stored there and it starts executing. Otherwise, the new program is placed in the non-volatile storage until further space in physical memory becomes available. Later on during execution, whenever a running program gets preempted or terminates, the OS searches the non-volatile storage for another program to be brought into physical memory.



(a) Swapping. (b) Virtual memory.

Figure 9 – Memory virtualization techniques.

The main advantage of swapping lies on its simplicity. No special hardware is required, although current architectures do leverage swapping support with segment registers. Furthermore, the OS may manage partitions by itself using any combination of the approaches that we discussed previously for memory partitioning and allocation. Unfortunately however, swapping has an important limitation that prevents it from delivering a full virtualization solution by its own. For instance, if a single program requires more memory than there is available in the underlying platform, it cannot run even if there is no other program running in the system. Indeed, some OSes address this issue by applying the swapping approach only to program segments (i.e., stack, heap and code), but the problem still remains unsolved.

The second approach (virtual memory) provides a complete virtualization solution and works as follows. Physical memory is divided in fixed-size partitions, called frames. Conversely, programs are split into fixed-size partitions of same size, known as pages. Based on this, the OS takes care of: (i) assigning pages to frames; (ii) keeping in memory those pages that are currently in use; and (iii) swapping out to the non-volatile storage those pages that are not (currently) in use. Unlike swapping, virtual memory requires support from the underlying hardware and a tight cooperation with the OS. On the one hand, the OS maintains a special data structure, known as page table, that provides the mapping of pages. On the other hand, the hardware relies on this structure to dynamically translate page addresses into frame addresses.

In virtual memory, the main challenge that the OS faces concerns the decision of which pages it should keep in memory. Indeed, if physical memory is large enough, this is trivially handled, because there are more frames than pages. However, this is not the base case scenario. Often, physical memory operates under pressure, and the OS should constantly offload unused pages to non-volatile storage and bring in to memory those pages that are needed. This is known as page replacement and effectively happens whenever a program tries to access a page that is not currently mapped in physical memory. In this event, the hardware triggers a page fault exception and traps execution flow to the OS. In turn, the OS locates the target page in the underlying non-volatile storage and selects some in-memory page to be replaced. Overall, the replacement

algorithm has an important impact in the performance of the system and for this reason it is a wide subject of research. Among the various algorithms that exist, we highlight below those that are the most relevant to this work.

First-in First-Out (FIFO) This algorithm relies on the idea that pages that were accessed longer ago have less chance to be accessed in the near future. Pages are kept in a queue structure, which is sorted according to their arrival time in memory. Whenever a page should be swapped out to non-volatile memory, the head page in the queue is chosen to be replaced (i.e., the page that was first brought in). This algorithm is simple to implement, but often presents the anomaly of swapping out heavily-used pages.

Second Chance This is an improved version of FIFO page replacement. Likewise, in-memory pages are kept in a queue structure according to their arrival time. However, before selecting a page, the algorithm checks if that candidate was accessed since the last run of the algorithm. If this was the case, then the candidate page is put back in the queue and the search continues. As soon as a valid candidate is found, such page is selected. In the worst case, all in-memory pages are processed and Second Chance degenerates to FIFO.

Least Recently Used (LRU) This algorithm embraces the idea that pages that were heavily-used recently will likely be used in the near future. Conversely, those that were not heavily used for longer, will unlikely be used next. Based on this, whenever a replacement decision is triggered, the algorithm selects the page that was not used for longer. This strategy yields to near optimal solution, but faces an important challenge concerning performance. Effectively, to select the least recently used page, in-memory pages would have to be placed in a priority queue structure that is sorted according to the number of times that each page was accessed. Noteworthy, the hardware would need to update this structure at each memory reference, which is clearly prohibitive in terms of complexity and performance.

Aging This strategy consists in a feasible implementation of LRU. In this approach, each page is assigned to a software counter that is maintained by the OS. At startup, this counter is set to zero, and at each N clock interrupts it is updated as follows. If the page was accessed since the last time that the algorithm ran, the counter for that page is shifted right and its most significant bit is set to zero. Otherwise, the counter is simply shifted right. This way, whenever a replacement decision is triggered, the page with the smallest counter is selected to be swapped out. Noteworthy, the smaller N is the more accurate is the algorithm and the greater is the overhead.

Working Set (WSet) This algorithm was designed from the ground up based on the temporal locality principle. Bottom line, it strives to keep in memory the working set of pages of a program. To this end, the OS assigns to each page a timestamp which is initially set to startup time. Furthermore, at each N clock interrupts all page table entries of the running program are traversed and have their accessed bit cleared. Then, whenever a replacement

decision takes place, all in-memory pages are scanned and processed as follows. If the corresponding page table entry of that page has its accessed bit set, the current virtual time is written to the timestamp field that is assigned to that page. This indicates that the page is in the working set of the program and should not be selected to be replaced. However, if the accessed bit for that page is not set, two scenarios unfold. In the first one, the timestamp of the page is greater than a delta time τ from the current time, the page is no longer in the working set and may be selected to be replaced. In the second scenario, the timestamp of the page is less than τ, which means that the page is still in the working set. Nevertheless, the page is noted as a candidate for replacement. The algorithm scans all in-memory pages, and it replaces the page that was accessed the longest time ago.

2.3.4 Process Management

The Process Management Subsystem (PMS) collaborates with other subsystems of an OS to handle important tasks such as program launching, execution and termination. In this section, we take an in-depth look on the most important abstractions, policies and mechanisms policies of the PMS. First, we introduce process and threads. Then, we examine scheduling policies. Finally, we discuss about inter-process synchronization and communication.

Process and Threads

The overall functioning of the PMS relies on two abstractions: threads and processes. On the one hand, the former models an execution flow of a running program. Software engineers may rely on this abstraction to exploit parallelism in their software, by spawning multiple threads within a program. The actual structure of a thread is dependent on the OS, but it often encapsulates the following features: (i) an execution stack that stores thread-local temporary variables; (ii) a data segment that holds thread-local static variables; (iii) a execution context that describes the state of underlying hardware, which comprises the values that are stored in machine registers and interrupt vectors.

On the other hand, a process is a container that encapsulates resources of a running program. The PMS relies on this abstraction to alleviate the complexity of dealing with resource concurrency in a computing platform. Without processes, when developing a program, a software engineer would need to worry about what resources other programs might be using, and thus she/he would have to explicitly deal with resource multiplexing in her/his software. Usually, the following features are encompassed in a process: (i) a set of threads, which model the execution flows of a program; (ii) a range of addressable memory areas, which is called address space; and (iii) a list of hardware and/or software resources, such as files and peripheral devices. Noteworthy, threads of the same process share a the same set of resources and address space.

To manage the execution of processes and threads, the PMS relies on a state machine that models which algorithms should be triggered. A possible diagram for this is outlined in Figure 10 and discussed next. Whenever a new process is created, it is placed in the «new» state.

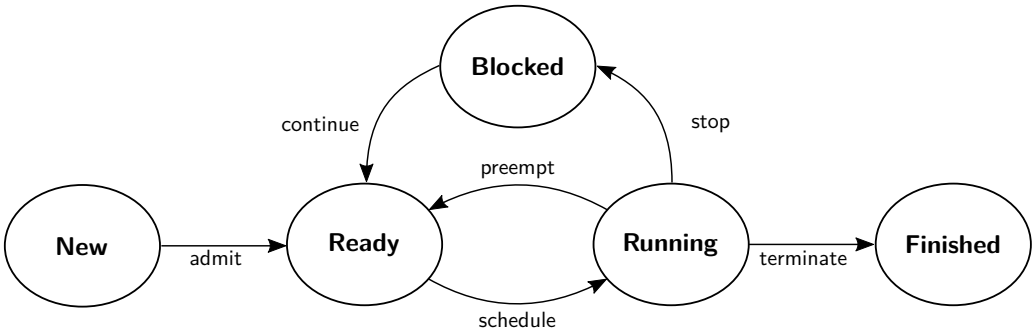


Figure 10 – States diagram for a process.

As soon as the process is ready to be executed, it is moved to the «ready» state and placed in a scheduling queue. Eventually, the PMS will select (a.k.a. schedule) this process to be executed and thus move it to the «running» state. The process will run until one of the following situation occurs. First, the process may stop because it has to wait for some resource. In this case, it is moved to the «blocked» state. In the second situation, it may get preempted by the PMS itself because it run for too long and the PMS decided that it is now time for another process to run. Finally, as soon as the process terminates its execution it is placed in the «finished» state.

Scheduling

The scheduling algorithm decides which process/thread should run and when. It is the core engine of the PMS and this algorithm is triggered whenever a process/thread is created, gets preempted or terminates. There are several factors that may be considered in a scheduling decision, such as processor utilization, response time and interrupt processing latency. Bottom line, a scheduling algorithm may be designed to meet different goals, but often three are aimed:

- Fairness: a process/thread should be granted a processor time according to its importance.
- Balance: scheduling decisions should maximize the utilization of resources.
- Policy Enforcement: scheduling decisions should follow system administration rules.

Table 1 outlines the main characteristics of the algorithms that are most relevant for this work, and next we detail their overall functioning. A more comprehensive discussion of scheduling algorithms is presented in (LEUNG; KELLY; ANDERSON, 2004).

Table 1 – Summary of popular scheduling algorithms.

Algorithm	**Use Case**	**Description**
FCFS	Batch Systems	Run processes/threads in the order that they come in
SJF	Batch Systems	Run the process/thread that least requires processor time
RR	Interactive Systems	Run for a quantum the ready process/thread that is waiting longer
Priority Scheduling	Interactive Systems	Run for a quantum the highest-priority ready process/thread
EDF	Real-Time Systems	Run for a quantum the process/thread with the earliest deadline

First-Come, First Served (FCFS) This is a non-preemptive scheduling algorithm that is typically employed in batch systems. It maintains a scheduling queue in which ready processes/threads are selected for execution in a FIFO fashion. When a process/thread comes in, it is placed in the end of this queue. When a scheduling decision should be made, the first process/thread in the queue is selected and runs until it terminates or blocks.

Shortest Job First (SJF) This is a non-preemptive scheduling algorithm that is often found in batch systems. It requires that run times of processes/threads are known in advance. Based on this, the algorithm maintains a priority queue in which ready processes/threads are selected for execution from the shortest to the longest one. When a process/thread comes in, it is placed in the scheduling queue according to its run time length. When a scheduling decision should be made, the shortest process/thread is selected and runs until it terminates or blocks.

Round-Robin (RR) This is a preemptive scheduling algorithm that may be used in interactive systems. Each process/thread is entitled to system-wide fixed time interval called *quantum*, during which it may use the processor. The scheduling algorithm maintains a queue in which ready processes/threads are selected for execution in a FIFO fashion. Whenever a scheduling decision happens, the first process/thread from the queue is picked up and runs until it terminates, blocks or runs out of quantum. When a new process/thread comes in, it is placed in the end of the queue.

Priority Scheduling This is a preemptive algorithm that is used in interactive systems. Likewise in RR, processes/threads are entitled a quantum. However, they are additionally assigned to a priority level, which is used to sort the scheduling queue. Priorities may be assigned statically upon process/thread creation; assigned dynamically by the OS itself; or by a mix of both. Either way, whenever a scheduling decision should be made, the process/thread with the highest priority is selected and runs until it terminates, blocks, runs out of quantum, or a process/thread with highest priority becomes ready. When a new process/thread comes in, it is placed in the scheduling queue according to its priority level.

Earliest Deadline First (EDF) This is a preemptive algorithm that is used in real-time systems. It requires that deadline of processes/threads are known in advance. Based in this information, the algorithm maintains a priority scheduling queue in which ready processes/threads with earliest deadlines are placed first in this queue. When a scheduling decision takes place, the process/thread that has the earliest deadline is selected to run.

Synchronization

Synchronization concerns the challenge of coordinating concurrent accesses to shared resources in such a way that semantic consistency is ensured across multiple interleaved operations on that resource. For instance, consider two threads, T_1 and T_2, that operate a common

global variable x. On the one hand, T_1 adds one to x. On the other hand, T_2 multiplies x by two. Therefore, depending on the order on which T_1 and T_2 execute these operations, x may end up with either of two values:

- If T_1 runs first we get $x = 2 \times (x+1)$.
- If T_2 runs first we have $x = (2 \times x) + 1$

The set of commands that operate on a shared resource is called a critical section. Having a non-deterministic execution of such sections may lead to potential side effects in the running program, which is clearly unwanted. This problem is known as race condition and is solved by providing mutual exclusion of running entities in a critical section. In this context, the job of the OS is to provide mechanisms to enable correct sharing of resources by exposing synchronization abstractions to programs. In the paragraphs that follow, we discuss three different solutions that are commonly featured by OSes: spinlocks, mutexes and semaphores.

Spinlocks are the simplest type of synchronization primitives. They are built on top of compare-and-swap (`cas`) instructions and have a binary state: *locked*, meaning that there is a thread running instructions of the critical section; or *unlocked*, meaning that no thread is running instructions of the critical section. Additionally, a spinlock is associated to two operations: `lock` which changes the state of spinlock from *unlocked* to *locked*; and `unlock`, which carries out the converse operation. Initially, a spinlock is *unlocked*, and when entering a critical section, `lock` should be called. This causes the calling thread to enter in a tight loop that relies in the `cas` instruction to atomically change the state of the spinlock to *locked*. As soon as this instruction succeeds, it returns from the call and the shared resource may be safely accessed. In contrast, when leaving a critical section, the thread calls `unlock` to atomically set the state of the spinlock to *unlocked*.

Spinlocks have minimum synchronization latency, because once a thread unlocks the spinlock, another thread may promptly enter the critical section. Unfortunately however, the active testing in the `lock` operation imposes two drawbacks. First, it wastes energy inherently, since threads that wait for entering the critical section constantly execute `cas` instructions. Second, as soon as the number of waiting threads increases it may potentially harm the performance of the system. The reason for this lies on the fact that `cas` instructions lock the bus between the processor and the memory. As a consequence even threads that are simply trying to read/write data from/to memory will have worsen latencies to access the memory. For these reasons, in practical situations spinlocks are often not used in programs. Instead, the OS uses them internally to carry out low-latency synchronization operations, but more importantly as the building block for constructing more complex operations.

Mutexes approach the aforementioned issues by coupling locking functionalities of a spinlock with the capabilities of the thread scheduler. The idea is simple: instead of making a thread to actively test if it may enter a critical section, this thread is preempted, placed in a waiting queue, and the underlying core is made available to another thread to run. As soon as the

thread in the critical region is done, it wakes up a thread that was waiting for such an event to happen, which can now execute. Noteworthy, with this solution, processing power is no longer wasted, because while a thread waits, the underlying core is available for other threads to run. The downside of this lies on the longer synchronization latencies, once the scheduler is involved.

While spinlocks and mutexes enable mutual exclusion in a critical region, often times more complex synchronization semantics are required. For instance, suppose a scenario where thread *A* and thread *B* operate on a fixed-length circular queue. Thread *A* places items in this structure, and thread *B* takes out items from it. Here not only the access to the circular queue should be secured, but also the number of free slots in this structure should be controlled somehow. If there are no slots available in the circular buffer, thread *A* should wait. Conversely, if there are no items to be consumed in the circular queue, thread *B* should wait. To specifically address synchronization scenarios where some sort of counting is involved, OSes provide a solution known as semaphores. In contrast to spinlocks and mutexes, semaphores have associated to them a multi-value state. This is stored in a counter, and essentially keeps track of the number of times running entities may enter the critical section without having to wait. Noteworthy, with these semantics it is not only possible to address the former use case scenario, but it also enables multiple read-only threads to be running in the same critical section, at the same time.

Semaphores have two operations associated to them: `down` and · The former, should be called when entering a critical section and it atomically checks the counter value stored in the semaphore. If it is greater than zero, the counter is decremented by one and the calling thread continues its execution. However, if the counter is less than or equal to zero, the calling thread gets preempted and is placed in a waiting queue. The ºperation carries out the converse operation of `down`, by atomically checking and updating the counter accordingly. This is called when leaving a critical region and makes the waiting queue of the semaphore to be inspected. If there is at least one thread there, the head thread is awakened and enabled to enter the critical section. Otherwise, the counter of the semaphore is simply incremented by one.

Inter-Process Communication

Threads of the same process work on a common address space. Therefore, if these threads want to exchange information, all they have to do is read/write to memory. However, when it comes to making processes to communicate, additional mechanisms are required, because processes do not share their address space. Therefore to address this, OSes provide Inter-Process Communication (IPC) primitives. Table 2 summarizes some solutions that are often available.

Shared memory regions consist in memory areas that are mapped to the address space of several processes. The OS creates them by carefully manipulating underlying memory management structures, such as page tables. When two or more processes attach the same memory region to their address space, they may all operate on common objects, as well as to exchange arbitrary information. In contrast to other IPC primitives, shared memory regions have one-sided semantics between processes, thereby enabling communication without any previous

Table 2 – List of IPC abstractions and primitives commonly found in OSes.

IPC Abstraction	Peers	Mechanism	Description
Shared Memory Region	M:N	Memory Sharing	Share variable-size objects.
Pipes	1:1	Memory Sharing	Transfer data streams.
Message Queues	1:N	Message Passing	Exchange fixed-size messages
Sockets	M:N	Message Passing	Exchange fixed-size messages between two hosts

agreement among the involved processes. For instance, one process may write data to a shared region, even if there are no other processes waiting for incoming data at that time.

Pipes are unidirectional channels that are used to connect data streams between processes. Typically, two processes use a pipe to exchange data as follows. At one end, one process (i.e., producer) puts data into the pipe; and at the other end, another process (i.e., consumer) takes data out from the pipe. Data flows from one end to another, and it is temporarily placed in a FIFO buffer. If there is no further space left in the buffer, the producer blocks waiting for data to be consumed. Conversely, if the consumer finds no data to be taken out from the pipe, it blocks waiting for data to be produced.

Message Queues enable processes to exchange fixed-size chunks of data with one another through messages. Processes are responsible for packing and unpacking information within a message, whereas the OS takes care of passing these messages around. Messages are assigned to priorities and the OS relies on this to process them. Message queues differ from pipes in two aspects. First, process may receive messages from several others. Second, messages are consumed from the highest to the lowest priority in receiver-side. Nevertheless, likewise in pipes, message queues feature a built-in synchronization scheme, in which consumer processes blocks when no messages are available in the queue, and producers blocks when there is no temporary space for messages.

Sockets are message-based IPC primitives that enable many-to-many communication between processes either residing in the same OS or spanning across different host OSes. In the latter case, they are routed through the network system, and are likely to rely on some network protocol, such as TCP/IP. Likewise message queues, information exchange with sockets is based on fixed-size data chunks, which are called *datagrams* in this context. Processes are responsible for packing and unpacking information, and the OS takes care of routing and delivering those datagrams. However, no priority processing of datagrams is imposed by the OS.

2.3.5 File System Management

The FSM provides a uniform interface for programs to store and retrieve data, as well as to manipulate hardware devices and software resources. To this end, the FSM relies on two main abstractions: files and file systems. In the following sections, we take an in-depth look on them. First, we present an overview of files, how they are organized and can be accessed. Then, we discuss about file allocation. Finally, we brief about the layout of a file system.

Files

Files enable the transparent interaction of programs with system resources. Typically, there are three types of file in a system. First, there are *regular files*, which are used by programs to store their data in the File System (FS). Second, there are *directories* which group files together in a three-like structure. Finally, there are *Special Files*, which model the access to hardware devices and software resources. More importantly, regardless the type of the file, the interface for operating on them is the same, thereby making files the fundamental abstraction for structuring data and manipulating devices. Some typical file operations are outlined in Table 3.

To operate on files, the OS relies on some meta information which we discuss next. First, each file in the system is assigned to a unique name, called filename. This is used to locate and access the file. The naming rules for files vary from system to system, but typically they may include letters and number and are composed by two parts: (i) a prefix, which stores some meaningful information for the end-user; and (ii) a suffix, which is also known as file extension and it is used by some systems to assign a program that handles that kind of file.

Besides the filename, files are also associated to some attributes, which the FS relies on to handle operations on a file. Often, there are four types of attributes. First, *access permissions* are used to determine which programs can access a given file and what operations may be performed. Second, *flags* control some specific properties of the FS. For instance, this can be a hidden flag, which indicates that the file should not appear in file listings, or an archive flag to specify that the FS should periodically backup that file. Third, *time stamps* record various times for the file, such as time of last access and modification. Finally, *structure* fields give fundamental information about the geometry of the file, such as its size, as well as length of records and keys.

File Organization and Access

File organization refers to the logical structuring of a file. Indeed, there are countless ways of structuring a file, each of which having its own advantages and disadvantages and targeting a particular set of requirements. However, we highlight three fundamental organizations that are typically found and also serve as building-blocks to implement more complex structures:

Table 3 – Common operations on files.

Operation	**Description**
`create(name, attr)`	creates an empty file with system-default attributes
`remove(name)`	releases all resources that are associated to a file
`rename(file, name)`	renames a file
`file = open(name)`	enables operations on a idle
`close(file)`	disables operations on a file
`read(file, data, ptr)`	reads data from a file
`write(file, data, ptr)`	writes data to a file
`seek(file, ptr)`	repositions the reading/writing pointer of a file
`chattr(file, attr)`	changes the attributes of a file
`attr = lsattr(file)`	retrieves attributes of a file

- *Byte sequence files* are structured as sequences of bytes (Figure 11a). From the perspective of the system itself, these files have no structure at all, and any particular structure should be imposed by programs. As a consequence, this approach enables maximum flexibility for the user, once data may be organized in the most convenient way for them.

- *Record sequence files* are structured as a sequence of fixed-size records, each of which having its own internal structure (Figure 11b). The main advantage of this organization lies on its performance, once reads/writes work on large chunks, thereby favoring I/O transfers.

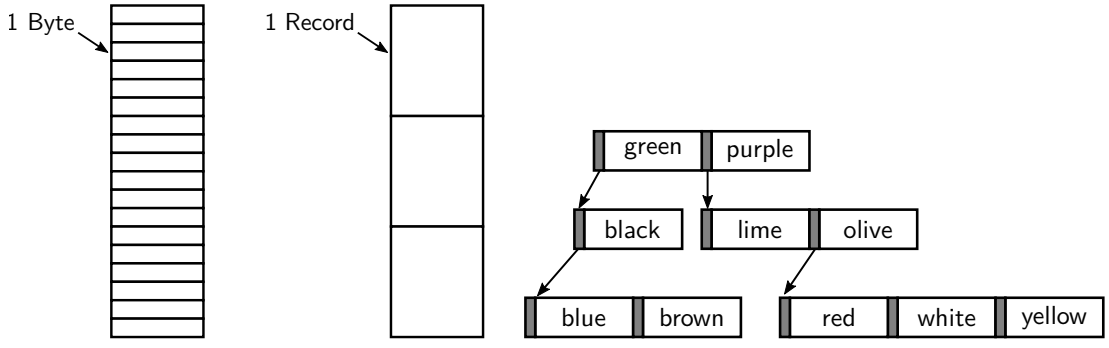
- *Tree files* are structured as a *m*-tree of records (Figure 11c). In this organization, records do not have necessarily the same size or internal structure, but all of them have a special key field at a specific position, which is used to sort the tree.

The organization of files directly impact on how file accesses perform. In byte sequence files, good performance is delivered in sequential reads or when appending data. However, for random access reading and writing, performance is highly dependent on the way user programs structure byte sequences. In record sequence files, reads and writes operate on records. As a consequence, sequential reads, data appending and arbitrary writes are efficiently supported. For arbitrary reads however, performance depends on the way that user programs structure their records. Finally, in tree files, random read accesses have an outstanding performance, since records may be quickly retrieved. Unfortunately sequential reads and write operations are not efficiently supported, once they may require the tree to be refactored.

File Allocation

File allocation concerns the scheme that the FSM uses to keep track of which data blocks belong to which files. There four main strategies, which we outline in Figure 12 and brief next.

Contiguous Allocation A file is stored as a contiguous set of data blocks, which is allocated at the time of file creation. The FSM maintains an allocation table with a single entry for



(a) Byte sequence. (b) Record sequence. (c) Tree structure.

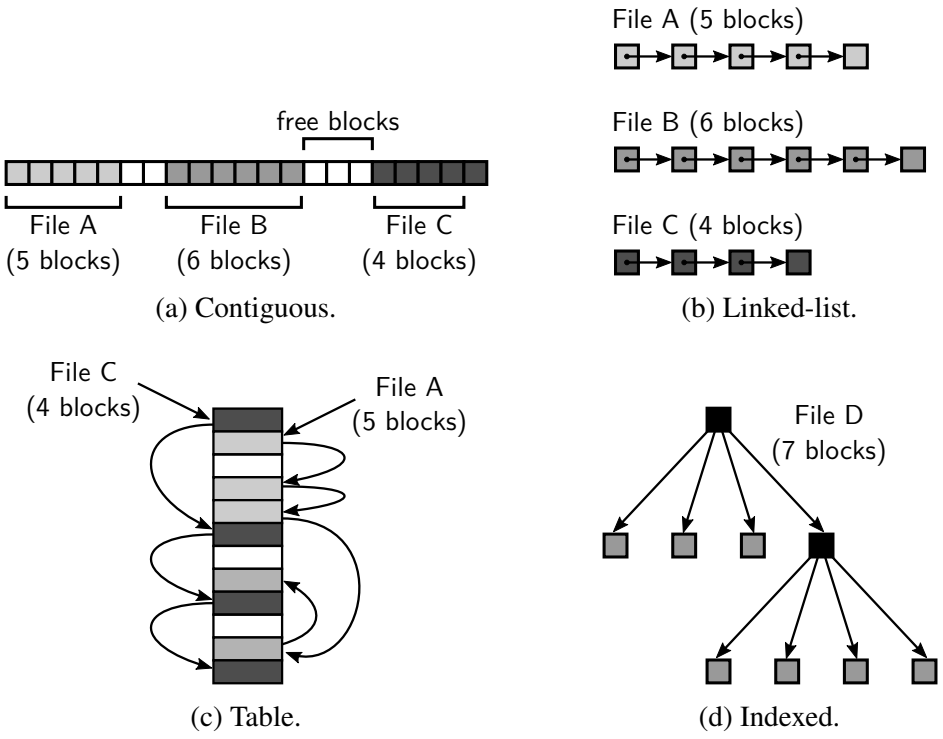
Figure 11 – Possible organizations for a file.

every existing file. Each entry holds the address of the first block and the length of the file. The main advantage of this strategy lies on its optimal read performance, because all blocks of a file are contiguously allocated. Unfortunately, this allocation scheme has an important drawback: over the time, the FS becomes fragmented. This happens because once files are removed and created, some disk space may be wasted between files.

Linked List Allocation Files are stored as a linked list of data blocks. To this end, the first word of each block is used as a pointer to the next data block in the file, and the remainder of the block is used for storing data. This scheme removes the problem of external fragmentation, but it delivers poor random access performance. In order to read/write to the n^{th} block of a file, the system has to start from the very first block and read the $n-1$ prior blocks.

File Allocation Table Files are stored as a linked list of data blocks, but pointers are stored in a special table, called File Allocation Table (FAT). With this approach, the entire block is available for data, and random access is improved. Although the chain must still be followed to find a given offset within the file, the FAT may be loaded into memory and thus the list may be traversed without making any disk references. The downside of this method is that it books a memory area that is proportional to size of the disk.

Indexed Allocation Each file is associated to a special data structure called Index-Node (I-Node), which stores the address of all disk blocks of a file. The main advantage of this scheme lies on its reduced memory consumption, since only I-Node for opened files are loaded in memory. Unfortunately, there are two disadvantages for indexed file allocation.



(a) Contiguous. (b) Linked-list.

(c) Table. (d) Indexed.

Figure 12 – File allocation schemes.

First, the number of files is limited to the number of I-Node in the file system. Second, the maximum size of a file is limited to the number of address pointers in an I-Node.

File Systems

File systems are complex structures that are used to store files. They typically reside in hard disks and solid-state drives, and thus they have a block-based geometry. The actual layout of a file system varies from one system to another but they often have some common fields in their structure. Figure 13 pictures a possible layout for a file system, which we discuss next.

The first block is known as boot block. It is reserved for the OS itself and it stores the boot program. The next block, is a special block that is called superblock and keeps important meta information about the FS, such as the file system identifier, the number of blocks in the file system and the starting offsets for other fields in the structure. Next, there is the block and file allocation tables, which may span over multiple blocks. The former keeps track of which blocks of the underlying device are free for allocation, and which are not. The latter holds information about files that reside in the file system. These information include access permissions, offset, file size. After these two tables, lies the root directory, which contains the top entry of the file system tree. Finally, the remainder of the blocks contains all the other directories and files.

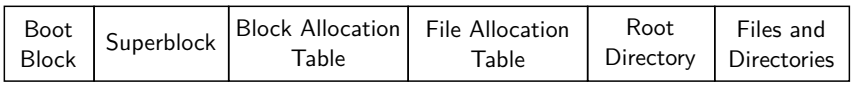


Figure 13 – A possible layout for a file system.

Directories

Directories are used to logically organize files in a FS. In the simplest systems, there is only the root directory, which lists all regular and special files in the FS. On the other hand, in more complex systems, there may be multiple directories, which are hierarchically organized.

Figure 14 outlines the structure of a directory. Overall, it is composed by a set of entries, one for each file that is therein listed. The maximum number of entries as well as their size depends on the file system implementation. For instance, directories may have a fixed number of entries or may dynamically shrink/grow; as well as entries may have fixed or variable sizes. Either way, more importantly is that entries correlate three information of a file: (i) the filename, which provides a logical identification for the file; (ii) the file attributes, which record access permissions, ownership and file size; (iii) disk pointers, which store the location of the file.

2.4 Distributed Systems

In Section 2.3 we detailed how OSes are engineered. However, we little discussed about specific issues that concern distributed architectures, which is the case of LW Manycore processors. Therefore, to narrow our discussion, in this section we cover the background

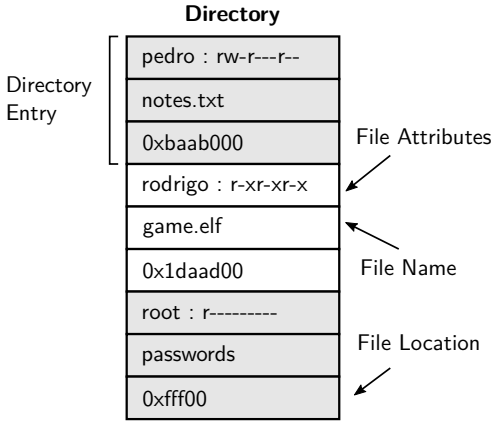


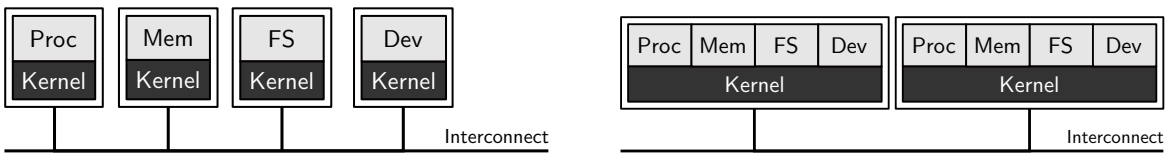
Figure 14 – Structure of a directory.

regarding distributed systems. Important to note, although our narrative extends to a broad system definition, the principles, mechanisms and algorithms that we present also apply to OSes. This section is organized as follows. In Section 2.4.1, we present some possible architectures for a distributed system. In Section 2.4.2, we discuss about communication mechanisms. In Section 2.4.3, we uncover algorithms for achieving coordination. In Section 2.4.4, we present an overview about data replication and management. In Section 2.4.5, we introduce a background for maintaining data consistency in a distributed system.

2.4.1 System Architecture

In distributed systems, an important issue concerns the way in which the system itself is structured. Overall, there are two orthogonal approaches: centralized and decentralized architectures. These alternatives are outlined in Figure 15 and discussed next.

In a centralized architecture, the system is organized as a collection of special processes called servers, each of which providing a different set functionalities to client processes. Servers may either implement entire or specific functionalities; as well as they may interact with one another to implement complex operations. This approach exploits vertical distribution and it is the simplest to implement. However, it has two problems, first, having a service centralized in one server may lead to scalability issues. Second, if a server fails, the underlying service becomes unavailable. Figure 61a outlines a centralized architecture for a Distributed OS. In this example, the OS is factored in four servers, which are in-turn spread across four different nodes. Each server implements a different subsystem and some of them cooperate.



(a) Centralized organization. (b) Decentralized organization.

Figure 15 – Architectures for distributed OSes.

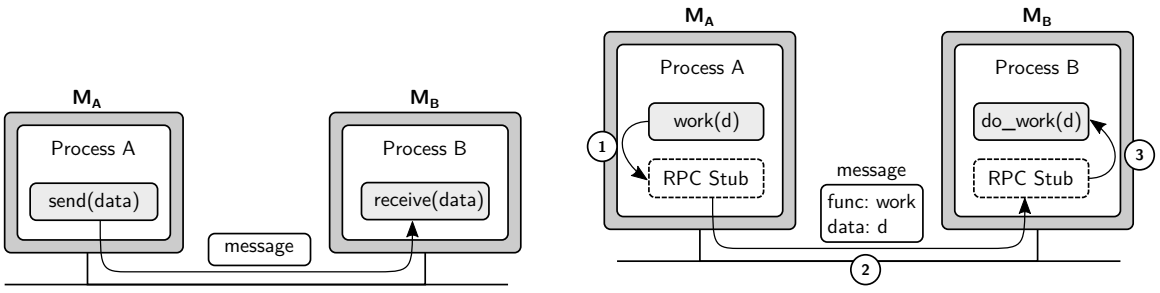
In a decentralized architecture the distributed system is structured as a set of servers, each of which implementing the complete set of functionalities. Each server provides the same set of interfaces, but serve different clients and potentially operate on different data. This architecture exploits horizontal distribution and offers better scalability, However, the downside lies on the fact that this architecture is harder to design and implement. Figure 61b presents a decentralized architecture for a Distributed OS. In this example, each node provides the full set of OS services and handles different client processes. In contrast to a centralized organization, components need to communicate less frequently, only to keep consistent shared data structures.

2.4.2 Communication

The foundations of distributed systems rely on mechanisms for inter-process communications. In this context, we examine the two most widely-used models for communication: message passing and Remote Procedure Call (RPC).

Message passing (Figure 16a) is the simplest communication model that one can imagine for a distributed system, because it is inherently supported by the underlying network itself. In a nutshell, there are two operations available for processes to make them communicate: `send` and `receive`. Based on these, on the one end a process sends a message (a sequence of bytes) to a destination; and on the other end, another process at the destination receives that message. Although simply put, there are two issues that concern message passing: (i) persistent *vs.* transient communication; and (ii) synchronous *vs.* asynchronous communication. Regarding the first issue, in persistent communication, messages are fully buffered in sender-side while they are not delivered to the receiver. In contrast, in transient communication, there is no buffering and data in sender-side is discarded as soon as it was submitted to the underlying network. Concerning the second issue, in synchronous communication, the sender is blocked until the message has been fully received at the destination. On the other hand, in asynchronous communication, the sender is unblocked and allowed to perform other tasks as soon as the message has been submitted for transmission.

RPC (Figure 16b) enables processes to call procedures across the network in a transparent fashion, that is, like they were local procedure calls. This relies on the message passing



(a) Message passing. (b) Remote Procedure Call (RPC).

Figure 16 – Communication mechanisms for distributed systems.

mechanism that we discussed previously, and it is handled by the communication system as follows. When a process running on network node M_A calls a procedure located on a network node M_B, execution flow in the caller process is transferred to a local stub procedure. In turn, this local stub: (i) packs parameters into a message, (ii) sends this message to M_B; and (iii) blocks the caller, waiting for a reply. When the message arrives at M_B, the message is forwarded to a server stub that unpacks the parameters contained in the message and runs the requested procedure. Once the procedure is complete, the result is packed into a message that is sent back to the network node where caller resides. Finally, when the result message arrives at M_A, the caller process unblocks, the local stub procedure unpacks the information on the return message, and normal execution flow is resumed in the caller. As a final remark, when it comes to RPC, an important issue is the way in which parameters/return values are packed into messages. While for primitive data types this is trivial, reference values should be carefully handled by the local stub and server stub procedures, because reference addresses are not valid across the different nodes in the network. To handle this, stub procedures are often employed to copy the entire data structures to which references are pointing to.

2.4.3 Coordination

Coordination concerns the management of interactions and dependencies between activities in a distributed system. In this Section we briefly examine aspects in this subject. First, we discuss about event ordering, then about mutual exclusion, and finally election algorithms.

Event Ordering

Distributed algorithms often depend on some method for ordering events in time. Promptly, one could suggest using oscillating circuits to keep a logical system clock and thus timestamp events. Unfortunately however, maintaining time in a distributed system using these devices may be ambiguous and the reason for this is two-fold. First, there are often multiple several oscillating circuits in a distributed system. Second, these oscillating circuits are likely to disagree in time.

Targeting this problem, several solutions were proposed, but here we highlight Lamport's work (LAMPORT, 1978), which we believe to have paved research on the area. Overall, instead of relying on physical time that is tracked by oscillating circuits, his solution works on logical time that is kept by software clocks, and an algorithm that maintains these abstract clocks synchronized across the peers of a distributed system. The synchronization algorithm itself is built on top of two premises:

(i) If two events occurred in the same peer P of a distributed system, then they have happened in the order in which that P observed them.

(ii) Whenever a message is sent between the peers of a distributed system, the event of sending the message occurs before the event of receiving the message.

Based on all this, peers may keep order of their events by updating their logical clocks as follows. First, the logical clock C_i is incremented before each time peer P_i issues an event. Then, whenever P_i sends a message to another peer P_j it timestamps the message with the current value of its logical clock C_i. On receiving this message, the logical clock C_j of P_j is updated to the greater between the two values: the current value of C_j and the timestamp included in the message. Finally, C_j is incremented and the message is considered as delivered to P_j, as well as that the two peers are synchronized.

As stated previously, Lamport's solution is able to keep partial order among peers of a distributed system. That is, it may timestamp with same logical times several events that occurred at different peers. Therefore, among this set of events, there is no way of establishing an order. Nevertheless, to keep total event ordering, his solution may be slightly modified and to also use the identifier of peers to break ties. Therefore, in this revised scheme timestamps are tuples formed up by both, the logical clock C_i and the identifier i of a peer P_i.

Mutual Exclusion

As in a single-node OS, processes in a Distributed OS frequently have to coordinate their activities. In order to ensure consistency when accessing shared resources, some mechanism for achieving mutual exclusion is required. To this end, either of two approaches may be taken: access to shared resources may be brokered by a server or carried out in a decentralized fashion by interest peers. In the paragraphs that follow we briefly examine some algorithms that fall into one of these two approaches.

The simplest way to achieve mutual exclusion in a distributed system is to have a server in charge of granting permissions to processes to enter critical sections. To enter a critical section C, a process A first sends a request to the server and waits for a reply. Upon receiving this request, the server checks if it has already granted permission to another process to enter critical section C. On the one hand, if this is not the case, then the server replies immediately and process A is granted access permission to enter critical section C. On the other hand, if access permission for running critical section C is currently assigned to other process B, then the server does not reply to process A, but it queues this request instead. As soon as process B exists the critical section C, it sends a release message to the server to communicate so. In turn, the server selects the oldest request that was queued and replies the corresponding process. Overall, this algorithm makes optimal use of bandwidth and requires minimum message exchanges for granting processes permission to enter a critical section. Nevertheless, the downside is that the server may become a performance bottleneck, specially in large-scale distributed systems.

One way to enable mutual exclusion without requiring a server to coordinate the concurrency between processes is to structure the order in which these processes may enter a critical section. This idea may be achieved by organizing processes in a logical ring. With this structure, mutual exclusion is achieved by having a special message (a.k.a. token) to be passed from process to process in a single direction around the ring. Conceptually, this token represents a

lock object, and thus the process that currently holds this object is allowed to enter the critical region. Conversely, to exit a critical region, a process has to send this token to its neighbor. If a process wants to enter the critical region but does not have the token, it waits until it gets the token, and retains it for how long it needs. If a process does not want to enter the critical section receives the token, it immediately forwards this object to its neighbor. The main strength of this algorithm lies in its simplicity. However, it constantly consumes network bandwidth and imposes a delay of up to N messages, where N is the number of processes in the ring.

An alternative solution for decentralized mutual exclusion relies on muticasting and event ordering. Whenever a process wants to enter a critical section, it broadcasts a request message to all other processes and awaits for their reply. While the process does not receive a reply from all others, it cannot progress. On the one hand, if no process is currently in the critical section, all of them will immediately reply the requester, which will in turn be granted entry to the critical section. On the other hand, if some process is currently in the critical section, then that process will postpone its reply until it is done its work in the critical section. Finally, to ensure that no more than one process requests access to the critical section at the same time, all messages in the algorithm are timestamped according to a global synchronized clock and the request with the lowest timestamp is always fulfilled first. The main advantage of this algorithm is its minimum synchronization delay. However, in contrast to the ring-based solution,it requires more messages to be exchanged.

Election Algorithms

Frequently in a distributed system, some process has to play some special role. For instance, this can be the one of a coordinator that grants processes access to critical regions, or a leader process that orchestrates consistency across replicated data. Election algorithms serve the purpose of choosing a unique process to play a particular role in a distributed system. In the next paragraphs we discuss two popular algorithms: Ring-Based Election and Bully Election.

Ring-Based Election was proposed by Chang and Roberts (CHANG; ROBERTS, 1979), and it works by picking up as coordinator of a distributed system the process that has the greatest identifier among all others. To this end, processes are disposed in a logical ring structure, in which each process has a communication channel to the next one and messages flow around the ring in one-way direction . Initially, there are no processes participating in an election, but any process can start one. As soon as a process P_i begins an election, it notes itself a participant in this tournament, places its identifier in a election message m_e and sends this message to its neighbor P_j. When P_j receives m_e, it compares the identifier in the election message with its own identifier. From this test, one out of two scenarios may arise:

- The arrived identifier is greater than the one of P_j. In this case, P_j marks itself as a participant in the election and forwards the message to its neighbor.

- The arrived identifier is smaller than the one of P_j. Therefore, P_j replaces the identifier in m_e with its own and carries out either of two actions, depending on whether or not P_j is a participant in the election. If it is, then P_j does not forward the edited message m_e. Otherwise, P_j marks itself as a participant in the election and forwards the edited message m_e to its neighbor.

Eventually, one process P_k receives an election message m_e which contains its own identifier. This happens because the identifier of this process is the greatest among all participants and this process should be elected as the coordinator. Therefore, P_k proceeds by announcing its victory as follows. It removes itself from the election and sends an announcement message m_a to its neighbor. When a process receives a message m_a, it removes itself from the election, records that P_k is the new coordinator and forwards the message to its neighbor, if it is not P_k. Overall, if a single process starts a ring-based election, then in the worst case, a linear number of messages concerning the number of participants will be exchanged in total. Despite this optimal performance, this algorithm has an important weakness, unfortunately: if one process crashes during an election, the whole algorithm is compromised.

The Bully Election was proposed by Garcia-Molina (GARCIA-MOLINA, 1982) to overcome this drawback. To this end, it relies on three assumptions: (i) processes know beforehand the identifiers of each one another; (ii) process execution speeds and communication times are bounded; and (iii) failures are locally detected with timeouts. Based on these premises, the election proceeds as follows:

- Whenever a process P_i detects that the coordinator has failed it starts an election.
- If P_i knows that it has the highest identifier among all others, then it elects itself as the coordinator and sends an announcement message m_a to all other process with lower identifiers.
- If P_i is not the process with the highest identifier, then it sends an election message m_e to those processes that have a higher identifier and waits for a reply message m_r to arrive.
- When a process receives an election message m_e, then it sends back a reply message m_r.
- If no m_r message arrives within the timeout, P_i assumes coordination and sends an announcement message m_a to all processes with lower identifiers. Otherwise, P_i waits for an extra timeout period for an announcement message m_a to arrive from the new coordinator. If none arrives it begins another election.
- When process P_i receives a coordinator message it sets its variable e_i to the identifier of the coordinator contained within it and treats that process as the coordinator.

Concerning the performance of the Bully algorithm, in the best case, the process with the second-highest identifier is elected as new coordinator and thus a linear number of messages is exchanged. On the other hand, in the worst case, the process with the lowest identifier is elected as coordinator and a quadratic number of messages is exchanged.

2.4.4 Replica Management

Replication of data is an important technique for distributed system that improves reliability and performance. Unfortunately however, these advantages come with the challenges of managing replicas and maintaining consistency across them. In this section, we cover the central issues that surround the management of replicas. Effectively, this is enabled by special processes of a distributed system known as replica-servers which deal with the following issues:

1. Replica Placement: decide where data should be placed.
2. Update Distribution: coordinate how modification on data should broadcast.
3. Update Publishing: orchestrate how updates are made available.

Replica Placement

There are three possible placements for replicas, as it is shown in Figure 17. Commonly, in a distributed system a combination of them is used to enable maximum reliability and performance. First, in the inner most level, there is the data storage in which a permanent replica reside. Often, this replica is used as a backup, in the case that all other replicas get corrupted; or as the only writable replica in the system, so as to guarantee data consistency. In the second inner most level, we have replica servers, which maintains server-initiated replicas. These are copies of permanent replicas and exist to improve performance. Finally, in the outer most level, there are client-initiated replicas residing in the client process itself. These are often read-only copies of server-initiated replicas and are provided to reduce even further access times to heavily used data items. Noteworthy, client-initiated replicas are also known as caches.

Update Distribution

Updates on data may be propagated in three different ways. First, data may be transferred from one replica to another. This is the simpler action to take, but it may consume too much bandwidth. Nevertheless, if read operations are more frequent than writes, the overall overhead is mitigated and performance is acceptable.

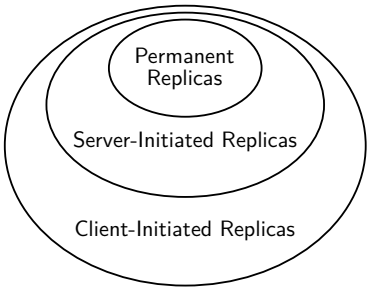


Figure 17 – Replica placement in a distribute system.

The second alternative is to propagate the notification of an update, which causes the content of other replicates to be invalidated. Later, when data is accessed on outdated replicas, valid data is fetched from the most updated copy. This requires an extra message to be passed around (i.e., invalidation message), but it saves some bandwidth, once it postpones data transfers to the moment when data is actually used.

Finally, the modification operation may be propagated to the other replicas. This operation carries all the information required to a replica to update its copy. This solution yields to the best performance among the three that we described, but it is the most challenging one to implement because operations may be arbitrary complex and hard to encode in a message. Furthermore, it is important to note that it consumes processing power in each replica, so that they can replay operations.

Update Publishing

Replica updates may be made available to clients using either of the following two approaches. First, in a push-based approach, updates are propagated to other replicas without those replicas even asking for the updates. Push-based approaches are often used between permanent and server-initiated replicas, but can also be used to push updates to client caches. Server-based protocols are generally applied when strong consistency is required.

In contrast, in a pull-based approach, a server or client requests another server to send it any updates it has at that moment. Pull-based protocols are often used by client caches. A pull-based approach is efficient when the read-to-update ratio is relatively low. This is often the case with (non-shared) client caches, which have only one client. However, even when a cache is shared by many clients, a pull-based approach may also prove to be efficient when the cached data items are rarely shared. The main drawback of a pull-based strategy in comparison to a push-based approach is that the response time increases in the case of a cache miss.

2.4.5 Data Consistency

When data is replicated in a distributed system, consistency across the copies must be ensured. That is, whenever a copy is modified, all others shall be updated accordingly. In this section we take a look at how consistency across replicas may be maintained. First we introduce consistency models that exist, and then we discuss about some relevant consistency protocols.

Consistency Models

A consistency model formulates a set of constraints that should be respected in order to guarantee a deterministic behavior when multiple processes simultaneously access shared data. Overall, there are two classes of consistency models which we uncover in the next paragraphs: data-centric and client centric. Models that lie in the first class aim at providing a consistent view of data to all processes in a system. These models assume that processes may be simultaneously

updating data, and thus it is necessary to impose constraints to counterattack this unwanted behavior. We highlight four types of consistency models:

- Sequential Consistency: the result of any execution is the same if the operations were executed in some sequential order; and the operations of each process appear in this sequence in the order specified by its program.
- Casual Consistency: operations that are casually related are observed by all processes in the same order, but processes may disagree on the order of operations that are casually unrelated.
- Eventual Consistency: write operations are performed by a single process, and if they do not take place for a long time then all the replicas will eventually become consistent.
- Entry Consistency: each data item x is assigned to a synchronization variable L, and operations on x are completed in order relative to the process that holds the lock of L.

On the other hand, client-centric consistency models restrict how processes can issue operations. These models are applied to systems that do not support simultaneous updates. Overall, there are four client-centric consistency models:

- Monotonic Read Consistency: if a process reads the value of a data item x, any successive read operation on x by that process will return that same value, or a more recent value.
- Monotonic Write Consistency: a write operation by a process on a data item x is completed before any successive write operation on x by the same process.
- Read-After-Write Consistency: the side-effect of a write operation by a process on data item x will be observed by a successive read operation on x by the same process.
- Write-After-Read Consistency: a write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.

Consistency Protocols

A consistency protocol comes down in an concrete implementation of a specific consistency model. Indeed, for each consistency model there is a concerning protocol. Notwithstanding, protocols that aim at sequential consistency are the most popular, hence this is the model that prevails in most distributed systems. Therefore, we focus our discussion in sequential consistency protocols.

Overall we highlight two classes of protocols that aim this consistency model: primary-based and replicated-write. In the former, data replicas are associated to a primary replica that is the only writable one and it is used for coordinating write operations. In the latter, write operations may be performed at multiple replica. We highlight the following protocols.

Remote-Write Protocol It is a primary-based protocol in which all writes are forwarded to a fixed server that retains the primary copy of the data, whereas read operations are carried out locally. The primary server order all incoming write operations and writes them back into data backup servers (if any). This is the simplest sequential consistency protocol to implement, but it imposes an important performance penalty on writes. As a workaround, non-blocking write operations may be used to slightly improve latencies.

Local-Write Protocol It is a primary-based protocol in which the primary copy of the data migrates between processes that wish to perform a write operation. On the other hand, reads are performed locally. Whenever a process wants to write on a data chunk, it locates the concerned primary copy and moves the chunk to its own location. Once the writer process is done, it propagates updates to the other replicas. The main advantage of this approach is that it enables multiple successive write operations to be carried out locally, and thus consequently write latencies mitigated.

Active Replication This is a replicated-write protocol in which update operations on a replica are actively and constantly propagated to the others, by relying on a totally ordered multicasting mechanism. This may be accomplished by means of a central coordinator that assigns a unique sequence number to each operation and broadcasts it to all replicas. Then, operations are carried out locally in each replica, in the order of their sequence number.

Quorum-Based Protocol This is a replicated-write protocol that requires processes to request and acquire permission of multiple replica-servers before either reading or writing from/to a replicated data item. If there are N replica-servers in the system, then a read operation is granted if a reader process assembles a quorum of at least N_R replica-servers. Conversely, a write operation is allowed if a writer process assembles a quorum of at least N_W replica-servers or more. Furthermore, N_R and N_W are also subject:

- $N_R + N_W > N$ to prevent read-write conflicts; and
- $N_W > N/2$ to prevent write-write conflicts.

2.5 Lightweight Manycores and Distributed Operating Systems

In this chapter we reviewed the background on which this thesis lies upon. In Section 2.1 we presented the landscape of single-chip manycore processors, highlighting differences between manycore accelerators, manycore processors and LW Manycore processors. In Section 2.3, we introduced the main concepts, principles, mechanisms and policies of OSes. We detailed how an OS is internally structured in different subsystems as well as their most important abstractions algorithms. Finally, in Section 2.4 we examined the main design and implementation aspects that concern distributed systems. In this section, we argue these areas are correlated and how we grounded our work on them.

First, it is important to emphasize that the unique set of architectural features of LW Manycore introduce multiple challenges in programmability and software portability:

- The *high density circuit integration* turns dark silicon into reality. All cores cannot be simply power on at the same time, otherwise the output heat in the processor cannot be effectively dissipated and the hardware may malfunction.

- The *distributed memory architecture* requires software to explicitly fetch/offload data from/to remote memories. Furthermore, the *small amount of on-chip memory* requires the software to partition its working data set into chunks and decide which chunks should be kept local and which should be offloaded to memory.

- The *on-chip interconnect* invites engineers to embrace a message-passing programming model and thus work on significant refactor on their software. Additionally, hardware exposes mechanisms for asynchronous programming and explicit message routing, which engineers should handle to achieve performance and efficiency.

- The *on-chip heterogeneity* makes the deployment of applications in LW Manycore complex. Clusters of LW Manycore may feature different processing capabilities as well as different connectivity to external devices. Therefore, processor allocation should account for these factors in order to extract most performance and efficiency of hardware.

One way to address the aforementioned programmability and software portability challenges in LW Manycore processors is through a fully featured OS. This is the approach that we explore on this thesis, and in summary it aims at bridging intricacies of the hardware by exposing rich abstractions and programming interfaces, as well as handling resource allocation, sharing and multiplexing. There are several possible ways to design and construct an OS. However, specifically for the context of LW Manycore processors we highlight a distributed OS architecture structure based on a multikernel design. This approach stands out when comes to scalability to massive architectures (WENTZLAFF; AGARWAL, 2009; BAUMANN et al., 2009) and we claim that it inherently best meets the unique architectural features of LW Manycore, specially when it comes to their constrained distributed memory architecture.

However, in this point it is important to stress the difference between a conventional distributed OS for networked environments (TANENBAUM; Van Renesse, 1985; PIKE et al., 1995) and distributed OS for LW Manycore processors (KLUGE; GERDES; UNGERER, 2014; ASMUSSEN et al., 2016). These differences concern design issues in distributed systems, which we discussed in section 2.4. Table 4 summarizes these concerns and next we discuss them.

The first and most important difference concerns the reliability in communication, which ends impacting other design aspects. On the one hand, conventional distributed OSes are designed to be deployed in network environments, which inherently face communication challenges in the physical layer, such as message corruption and loss. To address this problem, conventional

Table 4 – Distributed OSes for network environments vs LW Manycore.

Design Issue	Distributed OSes for Network Environments	Distributed OSes for LW Manycores
Communication Reliability	✓	✗
Fault Tolerance	✓	✗
Event Ordering	✓	✻
Mutual Exclusion	✓	✻
Replication Management	✓	✓
Leader Election	✓	✓
Update Distribution	✓	✓
Update Publishing	✓	✓

distributed OSes should implement message re-transmission protocols to enable reliable communication. On the other hand, distributed OSes for LW Manycore do not face this problem, once communication happens on-chip and it is reliable. The second aspect concerns fault tolerance. Either because communication links may fail or nodes may malfunction, conventional distributed OSes should account for this odds. In contrast, distributed OSes for LW Manycore may leverage the fact that a chip is a single hardware component which is either entirely working or not.

The third difference concerns event ordering which is an important concern to ensure data coherence as well as correct execution semantics. While conventional distributed OSes should implement some logical clocking algorithm, this is not always true for distributed OSes for LW Manycore. The reason for this lies on the observation that some of these processors have a global clock domain that coordinate all cores with no clock drift nor skew. This observation also greatly simplifies the design complexity of a distributed OS for LW Manycore. Similarly, regarding mutual exclusion, conventional distributed OSes have to provide these mechanisms entirely in software level. In contrast, some LW Manycore processors feature hardware synchronization mechanisms (i.e., hardware barriers and mutexes) in which distributed OSes for these processors may rely upon. Finally, and as a side remark, it is worth noting that other design aspects such as leader election and replica placement, as well as update distribution and publishing are equally challenging for both types of distributed OSes.

In the next chapter, we present existing distributed OSes that target manycore processors. We cover their most important design aspects and we argue why they are not enough to address the programmability and software portability challenges of LW Manycore processors.

3 RELATED WORK

In this chapter we summarize some relevant works that aim at supporting software development in single-chip manycore processors. We discuss about two complementary approaches:

- *Runtime Systems*, which interact with the hardware and/or compiler to expose API that enable narrowed programmability for a specific class of applications.
- *Operating Systems (OSes)*, which provide rich abstractions and interfaces to leverage software development, portability and deployment, in a wide range.

In Section 3.1, we present existing runtime systems and make considerations about the programming aspects on which they focus. Then in Section 3.2, we take an in-depth look on current OS solutions. Finally, in Section 3.3 we discuss about the current limitations of both approaches, and we summarize why our work advances the state-of-the-art on the programmability and portability of LW Manycore processors.

3.1 Runtime Systems for Single-Chip Manycore Processors

Runtime systems for single-chip manycore processors are often focused on facilitating parallel and/or distributed programming models. In this way, performance-demanding applications may be deployed more easily on these processors and thus benefit from the computing power that it is provided. In this context, we note four groups of runtime systems: Vendor Specific Libraries, Template Programming Libraries, Partitioned Global Address Space Libraries; and Message Passing Interface Libraries. In the next section, we discuss works on each of these approaches.

3.1.1 Vendor Specific Libraries

Vendor Specific Libraries are engineered to specifically target a particular processor. These libraries are constructed from scratch to expose the main features of the underlying architecture and thereby provide a high-performance programming environment. For instance, the Intel Single-Cloud Computer processor has the RCCE library (WIJNGAART; MATTSON; HAAS, 2011), which is a lightweight library for developing message passing parallel applications. RCCE makes extensive use of hardware Message Passing Buffer (MPB) to enable on-chip communication with minimum latencies. Similarly, Kalray MPPA-256 features libasync (HASCOËT et al., 2017), a library that efficiently manages on-chip DMA engines to enable high-performance on-chip communication with asynchronous and one-sided capabilities. Finally, the Adapteva Epiphany processor (VARGHESE et al., 2014) is shipped with a Software Development Toolkit (SDK) that exposes primitives for facilitating data transfers among the on-chip SPM, programming DMA channels and synchronizing execution across the cores.

3.1.2 Templates Programming Libraries

Templates Programming Libraries rely on programming language features to provide high-performance abstractions narrowed for specific purposes.

Richie *et al.* (RICHIE; ROSS; INFANTOLINO, 2017) proposed such a type of solution for enabling a Distributed Shared Memory (DSM) programming model in Adapteva Epiphany. They developed their solution using C++ containers and variadic templates, as well as they relied on CLETE-2 as back-end engine for data placement and parallel loop ordering. They evaluated their solution using a N-Body benchmark and a Finite-Difference Time-Domain (FDTD) solver, and they concluded that their infrastructure eases and leverages programming in the target processor.

Podestá *et al.* (PODESTÁ; NASCIMENTO; CASTRO, 2018) ported the back-end of PSkel (PEREIRA; RAMOS; GóES, 2015) to the Kalray MPPA-256 processor. This is a framework that provides a high-level programming abstraction for the stencil programming model. Based on this framework, software engineers may develop highly optimized and portable applications from various domains such as image filtering and dense linear algebra. The PSkel port for Kalray MPPA-256 was made on top of a vendor library (libasync). On-chip communications relied on DMA capabilities and techniques such as asynchronous communication and data prefetching were extensively exploited. Podestá *et al.* evaluated their solution using three different application kernels, and results unveiled that better performance and energy efficiency than state-of-the-art Symmetric Multiprocessing (SMP) processors and GPU may be achieved.

3.1.3 Message Passing Interface (MPI) Libraries

Message Passing Interface (MPI) Libraries expose rich communication primitives to enable efficient distributed computing in manycore processors.

Richie *et al.* (RICHIE et al., 2015) proposed a threaded version prototype of the MPI specification for the Adapteva Epiphany processor. In their runtime system, the multithreading scheme was implemented on top of COPRTHR SDK (PRONGNUCH; WIANGTONG, 2015) and on-chip communication relied on hardware DMA channels. They assessed their solution using a matrix multiplication code, and they concluded that good scalability may be achieved. However, they remarked a great challenge on designing their runtime system for this processor, due to the constrained architectural characteristics.

Quan Ho *et al.* (HO et al., 2015) introduced a proof-of-concept implementation of the MPI specification for the Kalray MPPA-256 processor. They grounded their implementation on top of a vendor library (libmppaipc) and considered optimizations such as eager send and lazy messaging, as well as the use of DMA channels to asynchronous transfers. Quan Ho *et al.* evaluated their solution using synthetic and scientific benchmark kernels. In summary, they reached three important conclusions. First, 75% of peak communication throughput could be achieved in the synthetic kernel benchmark, but this result depends on the capabilities of the

libmppaipc. Second, in the scientific kernel benchmark the best performance was achieved when using only 16 out of 288 cores of Kalray MPPA-256, due to the limited on-chip-memory available in the processor. Finally, they acknowledged that efficiently supporting the MPI programming model on LW Manycore processors is not trivial, once hardware is increasingly diverse and complex, and thus architectural optimizations are hard to exploit in a portable way.

Comprés Ureña *et al.* (Comprés Ureña; RIEPEN; KONOW, 2011) presented an implementation of the MPI specification for the Intel Single-Cloud Computer processor. Their solution was implemented on top of the Rckmb driver, and it embraced a hybrid communication scheme based on on-chip MPB and off-chip DRAM shared memory. The former was used for transferring small messages, whereas the latter was employed for sending/receiving large chunks of data. Comprés Ureña *et al.* evaluated their solution using synthetic programs as well as scientific application kernel benchmarks. In summary, their solution showed up to be scalable and to achieve better performance than a MPB/DRAM-only approach.

3.1.4 Partitioned Global Address Space (PGAS) Libraries

Partitioned Global Address Space (PGAS) Libraries aim at providing one-sided communication abstractions to enable a shared-memory-like programming model over a distributed memory architecture. To this end, these systems expose an API to enable applications to get/put data in a software-managed shared memory space. The PGAS runtime partitions this address space across physical memories and it handles data transfers, as well as it ensures coherency.

Serres *et al.* (SERRES et al., 2011) shared their experience on porting an Unified Parallel C (UPC) compiler to the Tilera TILE64 processor. Their approach relied on an in-house source-to-source translator from UPC to ANSI C, a narrowed implementation of Global Address Space Networking (GASNet) runtime system; and the Tilera C compiler. They evaluated their solution using synthetic microbenchmarks and application kernel benchmarks of the NAS Parallel Benchmarks (NPB) (BAILEY et al., 1991). In the end, they concluded that while good performance and scalability could be achieved, there are many optimization opportunities in what regards the performance gap accessing local and remote on-chip memories.

Ross and Richie (ROSS; RICHIE, 2016) presented an implementation of OpenSHMEM for Adapteva Epiphany. Their solution was constructed from scratch to the aforementioned processor and best exploit on specific architectural features such as dual-channel DMA engines, NoC topology and hardware barriers. Overall their implementation achieves 87% peak performance in a synthetic benchmark, thereby showing that it could improve programmability support in Adapteva Epiphany while delivering high-performance.

3.2 Operating Systems for Single-Chip Manycore Processors

In this section, we present existing OSes for single-chip manycores. We structured our discussion in two groups: OSes for manycore processors (shared memory), and OSes that target

LW Manycore processors (distributed memory).

3.2.1 Operating Systems for Manycore Processors

Manycore processors feature a high-number of cores and a globally-shared address space. OSes that target these architectures leverage both of the aforementioned characteristics: they embrace a distributed design to achieve scalability; and they rely on the shared memory to enable efficient communication in data intensive operations. In the next sections, we highlight three OSes for manycore processors that have pushed the stat-of-the-art: FOS, Barrelfish and HeliOS.

Factored Operating System (FOS)

FOS (WENTZLAFF; AGARWAL, 2009) was designed to cope with existing scalability problems of commodity OSes for high-core count architectures. To this end, FOS relied on two main design principles:

(i) Factoring: structure the OS in a collection of services. Each service is implemented by a set of collaborating system servers, which communicate via message passing.

(ii) *Space Multiplexing*: run servers in dedicated cores. In this way, the working set of the OS and the applications are spatially isolated and cache interference is mitigated.

FOS is structured in three layers. At the lowest one, a microkernel runs on each core of the architecture and it controls access to resources, as well as it exposes an API for enabling communication and process spawning. The second layer (called the OS layer) is composed by a collection of system servers that provide common OS functionalities such as core and memory management, system abstractions and file systems. Servers may cooperate with one another to provide these functionalities, but more importantly they all communicate using message passing primitives exposed by the microkernel. Finally, at the highest layer of FOS, a proxy library provides a common interface for applications to access and use system functionalities.

Beyond this structure, we stress two characteristics that were introduced alongside with FOS and we believe that are important in the context of LW Manycore processors as well. First, FOS employs ideas that are commonly used in Internet servers, such as caching, replication and lazy updates. We believe that these principles are fundamental to enable OS scalability. Second, FOS exploits spatial multiplexing by reserving some cores of the processor to run system servers on them. In the context of our research, we not only believe that this reduces OS interference on the execution of applications, but also it enables a reduced memory footprint for the OS, which is a strong limiting factor in LW Manycore.

Barrelfish

Barrelfish (BAUMANN et al., 2009) was introduced to address: (i) scalability to hundreds of cores; and (ii) adaptability to increasingly hardware diversity. To achieve this, it embraces

a multikernel design which lies upon three fundamental principles. First, all inter-core communication is made explicit, because the on-chip interconnect is gradually becoming richer and resembling a message passing network. Second, the OS structure is constructed to be hardware-neutral, once hardware is increasingly diverse and it is unfeasible to rely on specific hardware characteristics to deliver high-performance at system level. Finally, Barrelfish relies on state replication instead of sharing, once the former may be used to improve system scalability in a distributed design.

Barrelfish is composed by a set of independent OS instances, each of which being deployed on a different core of the underlying processor. In turn, OS instances are structured in two components: a CPU driver and a Monitor. On the one hand, the former runs in privileged-mode and provides fundamental system functions such as scheduling, communication and low-level resource allocation. On the other hand, the latter runs in user-mode as special process and encapsulate most of the mechanisms provided by an OS, like memory allocation, device drivers and file system. In order to maintain coherence across all the CPU drivers and Monitors, Barrelfish relies on two inter-core communication mechanisms. For state replication between CPU Drivers, it relies on Inter Processor Interrupt (IPI). In contrast, for ensuring coherent replication across Monitors it relies on RPC implementation that uses shared memory regions to transfer cache-line-sized messages around.

Andrew *et al.* implemented a prototype of Barrelfish for the x86_64 architecture, and they assessed their system running on different platforms, with diversity in both memory cache configuration and processor off-chip processor topology. They considered both synthetic benchmarks and scientific application kernels to assess their solution. Overall, their results unveiled that their multikernel OS may scale to a high number of cores, as well as to efficiently support multiple communication patterns. As a remark on their work, we stress the idea of replicating some system structures, rather than sharing them. We believe that this may greatly improve performance while simplifying the design of the system. We employed the principle of replication to devise the DPS system of Nanvix.

HeliOS

HeliOS (NIGHTINGALE et al., 2009) was proposed to address heterogeneous SPM systems, but the concepts and ideas that it introduced may be applied to manycore processors. Overall, HeliOS is based on four design principles:

(i) Require minimal hardware resources, once Processing Element (PE) may greatly differ in both processing and memory capabilities.

(ii) Require minimal hardware primitives so that HeliOS may be more easily supported by a number of PE.

(iii) Avoid unnecessary remote communication so as to avoid a high-performance penalty.

(iv) Avoid unnecessary local IPC and rely on local system calls as much as possible to improve performance.

Based on these points, HeliOS embraces a distributed structure that layouts on top of a set of satellite kernels. These are kind of microkernels and feature the same set of functionalities: a local memory manager, a thread scheduler and a namespace component for IPC communication. On top of these microkernels, traditional system services are provided in user-space as special processes, like in FOS. An interesting feature of HeliOS is the way that it provides process isolation: instead of relying on MMU and virtual address spaces, it ensures this requirement in software-level. All processes run in the same address space and at the most-privileged hardware execution level, and communication happens exclusively via a message passing mechanism (backed up by shared memory). As a consequence, the overall system design is simpler as well as context switches between processes are as fast as they are for threads. The implementation of HeliOS was based on Singularity (HUNT; LARUS, 2007) and a proof-of-concept version of the system was deployed in an Intel XScale and AMD Non-Uniform Memory Access (NUMA) platforms.

In contrast to the solution that we seek, HeliOS does not provide any Portable Operating System Interface (POSIX) IPC functionalities nor was designed to cope with a specific system interface standard. As a consequence, porting software to HeliOS is not a trivial task. Notwithstanding, in Nanvix we also aimed at covering the same design principles that HeliOS listed.

3.2.2 Operating Systems for Lightweight Manycore Processors

OSes for LW Manycore embrace a distributed design, they rely on message passing communication only, and they do not assume a global view of the hardware. Thanks to these characteristics, these OSes may cope with architectural intricacies of the underlying processor. In this section, we examine the existing OSes that are most relevant to our work.

Manycore Operating System for Safety-Critical Applications (MOSSCA)

MOSSCA is a distributed OS devised by (KLUGE; GERDES; UNGERER, 2014) to address safety-critical application domains. This system is designed based on three properties: (i) predictability; (ii) partitioning; and (iii) fine-grained communication. The first concerns the fact that the OS is analyzable and a predictable timing behavior. The second property regards the requirement that space and time partitioning of the hardware incurs in no interference between applications. Finally, the third property states that threads of the same application may communicate using a fine-grained local scheme, whereas communication between different applications are carried out by special OS primitives.

MOSSCA features a three-tier structure. On the bottom layer, a symmetric microkernel instance is deployed on each cluster of the underlying LW Manycore. This kernel manages all

cluster-local hardware resources, ensures protection and provides system mechanisms such as inter-cluster communication, application scheduling and local memory management. Moving to the second layer, there are system servers running on dedicated clusters of the LW Manycore processor. These servers provide services that cannot be executed locally on application clusters. There are four types of servers in MOSSCA:

- The OS Server provides higher-level system functionalities as well as is responsible for orchestrating access to system servers. In MOSSCA, there are multiple instances of OS Servers, each of which interacting with a specific partition. Applications cannot directly interact with this type of server.

- The I/O server intermediates the access to a specific peripheral device that is connected to the processor. There may be multiple instances of this type of server running, each of which specifically handling a device, by time multiplexing accesses and shaping input/output.

- The IPC Server stores inter-partition messages until they are consumed. In this way, the system predictability may be improved. There is one instance of this server in MOSSCA, and user applications may directly interact with this server to use the service that it provides.

- The Application Library Server hosts user-libraries that are commonly used by applications. In this way, some cluster-local memory may be saved and thus be available for the application. Furthermore, this server may provide some specific user API.

Finally, the third layer of MOSSCA consists in a stub library that resides on the same cluster of applications and intermediates the access to system servers. In summary, this component invokes system services and functionalities via the NoC using an RPC mechanism.

Kluge *et al.* implemented a proof-of-concept version of MOSSCA on top of an in-house simulator. Specifically concerning their contributions, we highlight that MOSSCA has greatly inspired our work two main points. First, on the idea of reserving dedicated clusters to run either applications or system servers. Second, in the aspect Application Library Servers may be used to host implementation of standard interfaces, such as POSIX, in our case.

Microkernel-Based System for Heterogeneous Manycores (M^3)

M^3 is a distributed OS proposed by (ASMUSSEN et al., 2016) to address the ever increasing heterogeneity in LW Manycore. Overall, it was devised based on three claims:

(i) The hardware heterogeneity can be hidden behind a common hardware interface;

(ii) The NoC may be used to isolate the execution of untrusted code in arbitrary cores;

(iii) The OS functionalities may be made available to arbitrary cores with RPC via the NoC.

M^3 embraces a hardware/software co-design approach to exploit these observations. On the hardware side, it relies on a specialized hardware component, called Data Transfer Unit (DTU). This component is attached to each PE of the underlying manycore processor and serves two purposes: (i) to provide a connectivity between PE and PE-external resources, such as memories, devices and other PE; and (ii) expose a uniform interface that enables the control of different PE remotely (i.e., outside of a PE). On the software side, M^3 bases its structure on a microkernel design. This kernel is deployed in exclusive PE of the processor, and it is solely responsible for driving DTU to ensure protection. System services and functionalities are entirely implemented as special applications in user-space and are deployed on their own PE. Finally, communication with kernels and system service applications is enabled through a RPC library stub library running alongside user applications.

Asmussen *et al.* implemented a proof-of-concept M^3 system in Tomahawk, a Multi-processor System-on-Chip (MPSoC) for mobile communication applications. Based on this prototype implementation, they used synthetic programs to benchmark specific characteristics of their solution, as well as they employed system-level utilities to evaluate in a broader extent the performance of M^3. Overall, Asmussen *et al.* reached important conclusions, which in turn supported some aspects of our work. First, they validated the idea that the clustered layout of LW Manycore processors and the NoC may be used to provide resource and performance isolation. Second, they showed that RPC system calls may be as fast as local system calls, thereby further supporting the idea that running system services in a remote cluster is feasible. Finally, they showed that a single kernel instance is enough to serve requests of multiple user applications at once (4 in their design), thereby suggesting that an asymmetric kernel design may be a feasible approach to deliver performance isolation.

3.3 Discussion

Runtime systems interact with the hardware and/or compiler to expose API that facilitate programming in a specific application domain. Specifically in the case of single-chip manycore processors, runtime systems mostly aim at delivering high-performance parallel and/or distributed programming environments. In this this context, we note why these runtime systems do not cover the open-challenges on LW Manycore processors:

- *Vendor Specific Libraries* are engineered for a specific architecture and inherently do not provide any support for software portability.

- *Template Meta Programming Libraries* offer portability only to those applications and platforms that they target, which is limited.

- *MPI and PGAS Libraries* do not cope with the small amount of on-chip memory nor they enable transparent memory access over multiple physical address spaces.

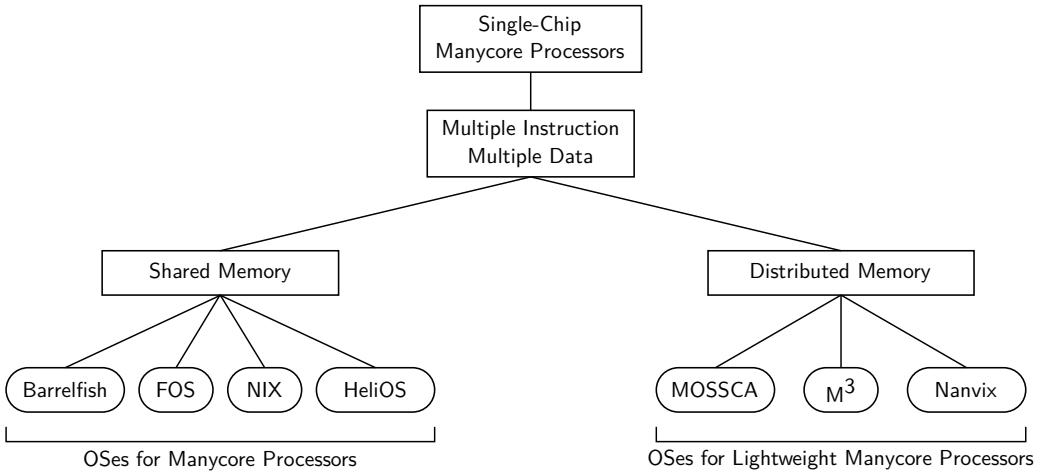


Figure 18 – Operating systems for single-chip manycore architectures.

In addition, it is important to highlight that runtime systems inherently lack on providing rich resource management like allocation, isolation, sharing and multiplexing. Putting it differently, they are not enough for delivering a complete solution for programmability and portability for LW Manycore processors.

OSes crave to bridge intricacies of an architecture, by exposing rich abstractions, programming interfaces and resource management. Unfortunately, existing OSes struggle to fully address programmability and portability challenges in LW Manycore processors. Figure 18 outlines the rationale for this and next we carry out a further discussion on the subject.

On the one hand, fully OSes such as Barrelfish, FOS and HeliOS do not primarily target LW Manycore, and were not designed to address architectural constraints of these processors. For instance, they do not account for the multiple physical address spaces nor cope with the small amount of on-chip memories. In summary, it is not possible to deploy these OSes on LW Manycore without a complete redesign and significant changes on their code.

On the other hand, MOSSCA and M^3 do target LW Manycore, but still lack on delivering programmability and portability to these processors. Overall, the rationale for this lies on the fact that both OSes do not offer support for virtual memory, nor any means for transparent accesses over multiple address spaces and address space expansion. Applications written for these systems should rely on specific API and handle data accessing by their own, thereby incurring in non-portability and poor programmability. Furthermore, it is worth noting that that MOSSCA and M^3 targeted home-grown simulated LW Manycore processors, with very special features. Therefore, it is not trivial to determine to which extent they can be ported to other similar architectures, including silicon chips.

In contrast to these OSes, Nanvix provides a complete solution for programmability and portability for LW Manycore processors. To this end, it relies on a DPS system to overcome memory-related challenges, it features a lightweight communication facility to enable efficient on-chip communication, and it is grounded on a comprehensive HAL that enables system portability to a variety of architectures. In the next chapter, we detail our solution.

4 THE NANVIX OPERATING SYSTEM

Nanvix is a distributed OS that aims to address programmability and portability issues in LW Manycore processors. In summary, it is the concrete implementation of the contributions of this thesis. Nanvix is open-source [1] and is the result of a joint collaboration between Pontifícia Universidade Católica de Minas Gerais (PUC Minas), Universidade Federal de Santa Catarina (UFSC) and Université Grenoble Alpes (UGA).

In this chapter, we present the internals of Nanvix, as well as we discuss how it addresses the problems that we target. First, we summary our design goals (Section 4.1) and we present an overview of the system (Section 4.2). Next, we detail the internals of Nanvix using a bottom-up approach (Section 4.3 to Section 4.6). Finally, we give some insights on the implementation of Nanvix (4.7) and conclude this chapter by stressing how our OS improves programmability and portability in LW Manycore processors (4.8).

4.1 Design Requirements and Goals

LW Manycore processors are well known for their performance, scalability and low-power consumption. Unfortunately however, these emerging architectures struggle when comes to portability and programmability. On the one hand, the former concerns the difficulty for getting existing software to run on a novel architecture. The easier this is, the more likely an architecture will be employed in more real-world contexts. On the other hand, programmability regards how challenging it is to design and develop software for a given system/architecture. If software development is too complex, it is unlikely that an architecture will be widely adopted. To target these problems, we aimed the following design goals for Nanvix:

- Programmability
  - Enable resource multiplexing to make it possible to concurrently deploy multiple applications in the processor and thus increase the overall utilization of the platform.
  - Ensure resource isolation to prevent applications to have either they performance impacted or have their execution tempered.
- Portability
  - Provide rich abstractions, such as processes and files. This way, software engineers do not need to worry about implementing, embedding and maintaining them in their software
  - Expose standard programming interfaces, thereby enabling software to be ported with minimum syntax changes.

[1] Publicly available at: `https://github.com/nanvix`.

– Support multiple programming paradigms (i.e., shared and distributed memory), thereby enabling a smooth shift from traditional multicore and manycore architectures.

4.2 System Overview

Nanvix features a multikernel design (BAUMANN et al., 2009; WENTZLAFF; AGARWAL, 2009) that is structured on top multiple instances of a microkernel. We chose this organization because we believe that it best meets the characteristics of LW Manycore. On the one hand, the multikernel approach copes with the distributed memory architecture of these processors. On the other hand, the microkernel-based structure enables a lightweight and modular implementation of system services, which is a fundamental requirement given the scarcity of hardware resources. Figure 19 depicts an architectural overview of Nanvix. It has a four-tier layout which we introduce next.

The HAL exposes a comprehensive abstraction of the underlying hardware (PENNA; FRANCIS; SOUTO, 2019). It is implemented as a baremetal library, and it ships common routines and data structures to manipulate hardware structures, such as cores, MMU and NoC. In summary, the HAL enables the portability of our OS across multiple LW Manycore processors, such as Kalray MPPA-256 (DINECHIN et al., 2013), OpTiMSoC (WALLENTOWITZ et al., 2012) and PULP (ROSSI et al., 2017). In Section 4.3, we further discuss about this layer.

The microkernel provides resource sharing, multiplexing, isolation and protection, within a cluster of a LW Manycore (PENNA et al., 2019). It runs in privileged mode and exposes bare minimum OS abstractions and primitives, such as threads, virtual address space and inter-process communication mechanisms. Our microkernel features an asymmetric design, which means that it exclusively runs in one core of the cluster, and it leaves the remaining cores to general-purpose use. This design mitigates the interference of the kernel in user-level software, and thus improves performance. In Section 4.4 we detail the microkernel of Nanvix.

System servers are deployed on top of the microkernel and implement services commonly found in an OS, such as process scheduling, memory mappings, and file allocation. In turn, subsystems are implemented by having a set of these system servers that work together, in a distributed fashion. In Section 4.5 we uncover the services of Nanvix. Finally, on top of system servers, the runtime libraries are disposed. These libraries are linked to the user applications and

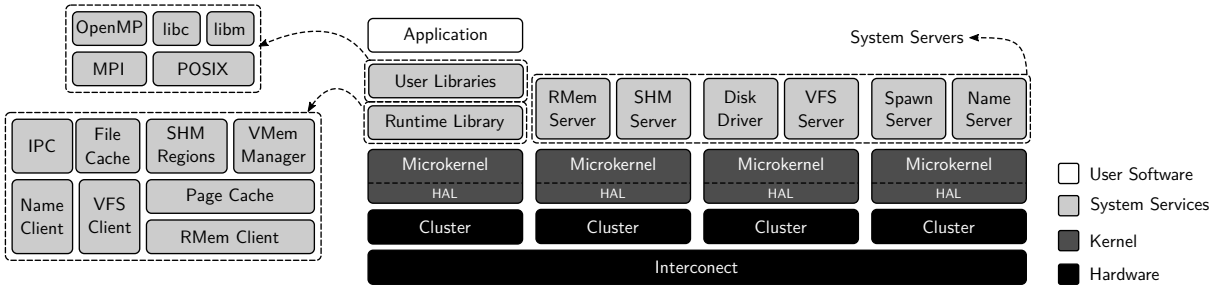


Figure 19 – A structural overview of Nanvix.

provide a standard interface for user-level software to interact with the OS. In Section 4.6 we briefly discuss about the libraries and runtimes featured by Nanvix.

4.3 Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) enables the portability of Nanvix to several architectures. In this section, we discuss the internals of this layer. First, we summary our design goals for this layer 4.3.1. Next, we present an overview of its structure (Section 4.3.2). Finally, we detail its functioning (Section 4.3.3 to Section 4.3.5).

4.3.1 Design Goals

LW Manycore processors may greatly differ from one to another. Some may have software-managed MMU, while others have these structures managed by the hardware; the NoC topology and communication capabilities may change; and the specific processing capabilities of each clusters may be unique to each architecture. Therefore, to cope with this intricacies, the HAL of Nanvix is designed to achieve the following three goals:

(i) *Comprehensibility*: enable portability across multiple LW Manycore processors;

(ii) *Extendability*: be easy to modify and thus to support new architectures; and

(iii) *Flexibility*: enable resource management to be implemented in overlying system layers.

First, to approach *comprehensibility*, we conditioned our design to account for differences in all software-driven components of a LW Manycore processor, such as cores, interrupt system, memory caches, MMU and on-chip interconnects. Second, to deliver *extendability*, our HAL was required to have a modular design. This way, a new architecture could be supported by re-implementing a set of modules for the new target, instead of demanding a complete re-write of the HAL. Finally, to enable *flexibility*, our HAL had to be minimum and expose primitives for manipulating hardware components, rather than providing rich software abstractions.

4.3.2 Overview

Aiming these goals, we structured our HAL in three layers, as it is presented in Figure 20:

- The *Core Abstraction Layer* provides common routines for manipulating structures of a core, such as interrupt system, registers, memory caches and MMU.
- The *Cluster Abstraction Layer* abstracts intricacies of the underlying cluster, like inter-core signals and integrated peripherals.
- The *Processor Abstraction Layer* exposes a uniform interface for the dealing with the on-chip interconnect, such as topology, message routing and asynchronous operations.

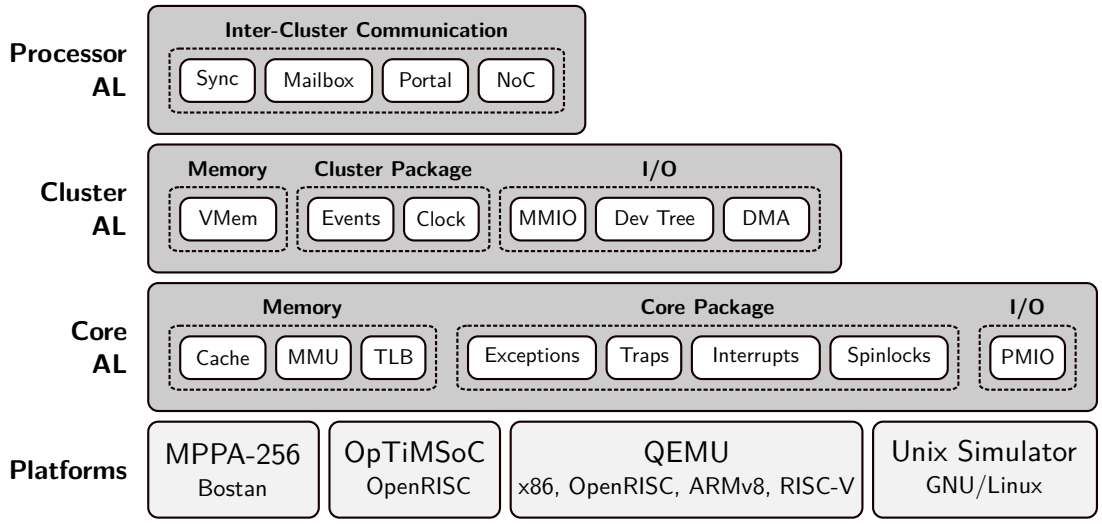


Figure 20 – Structural overview of the HAL of Nanvix.

Thanks to this decoupled software structure, we are able to expose a common interface across a diverse range of architectures. For instance, the current HAL of Nanvix supports the following platforms:

- Bostan: the second-generation of the silicon Kalray MPPA-256 processor.
- OpTiMSoC: a FPGA-emulated, OpenRISC manycore processor.
- QEMU: a virtualized multicore processor supporting x86, OpenRISC, ARMv8 and RISC-V.

In addition to the aforementioned platforms, which have baremetal counterparts, Nanvix may also run on a simulated environment on a host Unix system. Our OS is shipped with a tool that we developed to simulate the behavior and main characteristics of a LW Manycore. This utility uses POSIX abstractions to mimic the main characteristics of the processors that Nanvix targets. For instance, processes are employed to simulate clusters, threads used to model cores and message queues to reproduce message-passing communication through NoC links. The reason why we developed such a tool was two-fold: (i) existing virtual machines and architectural simulators missed support for emulating/simulating LW Manycore processors; and (ii) implementing an OS from scratch and directly on top of baremetal is very challenging technical task. Thanks to this Unix simulator we could speedup the development of Nanvix. Furthermore, one may rely on this tool to run Nanvix even without access to dedicated hardware, as well as to and prototype new OS ideas for LW Manycore processors.

4.3.3 Core Abstraction Layer

The *Core Abstraction Layer* has two modules: the *Memory Package*; and the *Core Package*. Table 5 presents the interface that they exposes, and next we detail each of them in turn. On the one hand, the former module exposes a uniform view of Translation Lookaside Buffer (TLB) and MMU structures. Furthermore, it exports routines for: (i) flushing, reloading

Table 5 – Interface exposed by the *Core Abstraction Layer*.

Operation	Description
`tlbe_write()`, `tlbe_read()`	Encodes/Reads a TLB entry.
`tlbe_inval()`, `tlbe_flush()`	Invalidates/Flushes a TLB entry.
`dcache_inval()`, `dcache_flush()`	Invalidates/Flushes the data cache.
`mmu_page_map()`	Maps a page into a frame.
`mmu_page_walk()`	Searches for a page mapping
`spinlock_lock()`, `spinlock_unlock()`	Locks/Unlocks a spinlock.
`exception_register()`, `exception_unregister()`	Registers/Unregisters an exception handler.
`interrupt_register()`, `interrupt_unregister()`	Registers/Unregisters an interrupt handler.
`kcall()`	Issues a trap call.

and invalidating caches; (ii) encoding, reading, invalidating and flushing TLB entries; and (iii) looking up and changing paging structures.

We relied on two decisions to design this interface. First, by providing a standard interface for the TLB and paging structures, an overlying memory management system does not need to worry about the intricacies of these hardware components. For instance, in Kalray MPPA-256, the TLB is split into two smaller structures, each of which with different lengths and associativity capabilities. If this complexity was exposed, the overlying kernel would be platform-dependent. Second, we chose to expose a software-managed view of the memory hierarchy, even though it may be fully-managed by the hardware in some targets. In summary, our motivation for this decision was two-fold. In architectural families that TLB are software-managed, an important performance improvement can be achieved when the actual management takes place in higher software levels. Whereas in architectural families which manage caches and TLB at the hardware-level, no performance drawback is observed if such interface is exposed, because indeed the underlying implementation is linked to dummy wrappers (i.e., do nothing).

On the other hand, the latter module (*Core Package*) provides a locking interface, and a uniform dispatching scheme for hardware interrupts, exceptions and traps. Overall, this abstraction module was designed so that a thread synchronization system, a rich interrupt/exception facility and a system call interface could be built on its top, without requiring any additional architecture-dependent code to be written.

4.3.4 Cluster Abstraction Layer

The *Cluster Abstraction Layer* provides an interface for managing the cores of a cluster. It exposes routines for powering on/off cores, as well as starting, stopping and suspending instruction execution in them. Furthermore, this module also ships functions to send and wait inter-core notifications, thereby paving the basics for thread synchronization. Bottom line, these features enable the construction of a fully featured thread management system on its top.

In Table 6, we summary the most relevant operations exposed by the *Cluster Abstraction Layer*. The `core_startup()` routine is called to power on a core and initialize all underlying architectural structures, such as the interrupt vector tables, TLB and page tables. Conversely,

Table 6 – Interface exposed by the *Cluster Abstraction Layer*.

Operation	Description
`core_startup()`	Power on the underlying core.
`core_shutdown(status)`	Shutdowns the underlying core.
`core_start(coreid, routine)`	Starts instruction execution in a core.
`core_reset()`	Stops and resets execution in the underlying core.
`core_sleep()`	Suspends execution in the underlying core.
`core_wakeup(coreid)`	Resumes execution in the underlying core.

the `core_shutdown()` de-initializes hardware structures and powers off the underlying core. The next three functions, `core_start()`, `core_reset()`, `core_sleep()` and `core_wakeup()`, may be used to start, stop, suspend and resume instruction execution in a core, respectively.

More concretely, a thread management system would work as follows. To create a thread, it would invoke `core_start()`, supply as a parameter the identifier of the core in which instruction execution should start. Conversely, whenever the thread finishes its execution, it would call `core_reset()`. Finally, thread switching as well as sleep/wakeup synchronization primitives would be implemented on top of `core_sleep()` and `core_wakeup()`.

4.3.5 Processor Abstraction Layer

The *Processor Abstraction Layer* (Table 7) provides supporting functions that enable clusters to exchange data. Overall, three abstractions are exposed: *sync*, *mailbox*, and *portal*. A *sync* enables a set of clusters to notify or wait for a notification from a set of clusters, and thus it provides the bare bones for inter-cluster synchronization. The *mailbox* abstraction enables clusters to exchange fixed-size messages with one another. A message may encode small operations and system control signals, and can have one or many recipients. Finally, the *portal* abstraction enables dense data transfers between clusters, either synchronously or asynchronously. A *portal* can be opened just between a pair of clusters and it features a built-in flow control semantics.

Table 7 – Interface exposed by the *Processor Abstraction Layer*.

Operation	Description
`sync_open()` / `sync_close()`	Open/Close a write-only sync.
`sync_create()` / `sync_unlock()`	Create/Destroy a read-only sync.
`sync_signal()` / `sync_wait()`	Send/Wait a signal to/in a sync.
`mailbox_open()` / `mailbox_close()`	Open/Close a write-only mailbox.
`mailbox_create()` / `mailbox_unlock()`	Create/Destroy a read-only mailbox.
`mailbox_read()` / `mailbox_write()`	Read/Write messages to a mailbox.
`mailbox_wait()`	Wait for the completion of an operation in a mailbox.
`portal_open()` / `portal_close()`	Open/Close a write-only portal.
`portal_create()` / `portal_unlock()`	Create/Destroy a read-only portal.
`portal_read()` / `portal_write()`	Read/Write data to a portal.
`portal_wait()`	Wait for the completion of an operation in a portal.

Note that in the design of this interface, we intentionally decoupled small data transfers from large ones by exporting two abstractions (i.e., *syncs* and *mailboxes* and *portals*). The rationale for this decision was two-fold. First, we enable a narrower implementation of these different transfers, and thus we are able to exploit even further the characteristics of the underlying architecture. For instance, in LW Manycore that feature multiple NoC with different bandwidths, such as Kalray MPPA-256, one NoC may be exclusively used for *syncs* and *mailboxes* and another one for *portals*. Second, with multiple interfaces, we push even further *flexibility* of our HAL. This is because the overlying kernel and system services may choose which types of data transfer they want to rely on to build complex communication protocols.

4.4 Asymmetric Microkernel

The kernel of Nanvix provides resource management, sharing and isolation at cluster level of a LW Manycore processors. In this section, we present how we designed this OS layer. In Section 4.4.1, we summary our goals. In Section 4.4.2, we present a structural overview of the kernel. Finally, from Section 4.4.3 to Section 4.4.5 we uncover the internals of our kernel.

4.4.1 Design Goals

LW Manycore processors face an important duality in their design. While they integrate hundreds of cores in a single die, at the same time they have a limited amount of on-chip memory available. To cope with this aspect, we aimed at the following design goals:

(i) *Minimalism*: only provide essential functionalities, so that memory footprint is minimum.

(ii) *Flexibility*: enable a wide-range of OS services to be deployed on top of our kernel.

(iii) *Efficiency*: make available the most of hardware resources to applications.

We believe that an asymmetric microkernel best addresses these goals, thus we embraced this design in Nanvix. Our claim is based on three observations. First, a microkernel architecture is inherently minimalist. Second, a microkernel architecture enables maximum flexibility. Finally, an asymmetric structure delivers better performance isolation (PENNA et al., 2019).

4.4.2 Overview

The kernel of Nanvix features an asymmetric microkernel design, which means that: (i) it only provides essential functionalities that are needed to implement richer OS services in user space (microkernel architecture); and (ii) the kernel itself exclusively runs in one core of the underlying cluster of a LW Manycore processor (asymmetric structure). Figure 21 presents an overview of the Nanvix kernel. It lies on top of the Nanvix HAL (see Section 4.3) and it is organized in two logical layers, which we describe next: the *Modules Layer* and the *Kernel Call Layer*.

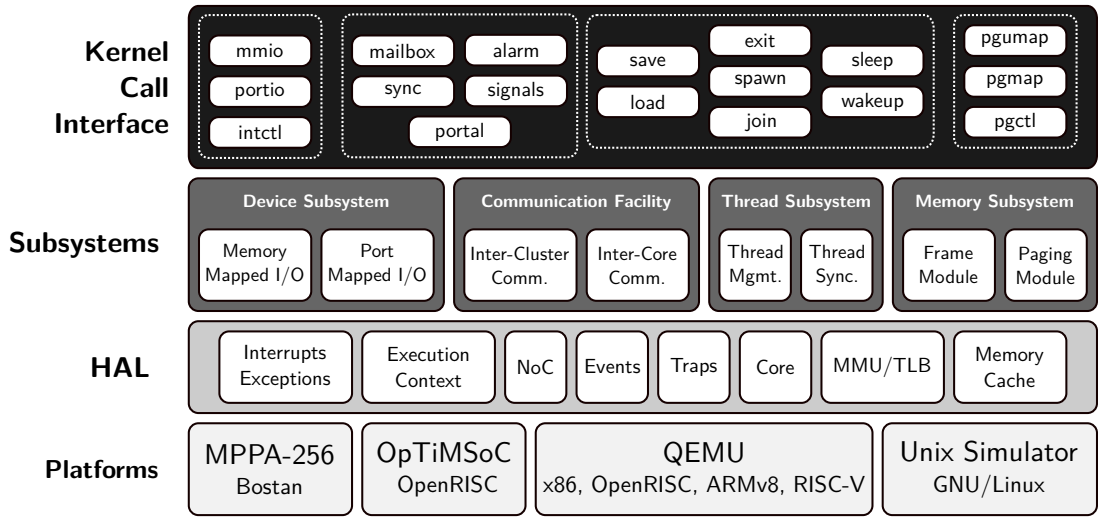


Figure 21 – Structural overview of the Nanvix microkernel.

The *Modules Layer* hosts the implementation of the functionalities and capabilities of the microkernel, and it currently features four modules. The *Device Module* manages access permissions to memory and port mapped devices, as well as it exports uniform routines for reading and writing to both device types. Furthermore, it provides the required mechanisms to forward the implementation of rich device drivers to user space. The *Thread Module* provides a thread abstraction in kernel space. Kernel threads run in uninterruptible mode and have exclusive access to a core. This module schedules kernel threads to cores in a FIFO fashion, and it supports the multiplexing of several user threads on top of kernel threads through programmable software alarms. Also, the *Thread Module* features sleep and wakeup routines, to suspend and resume the execution of a thread, respectively.

The *Communication Facility* provides three inter-cluster communication primitives: (i) *syncs* which enable mutual exclusion; (ii) *mailboxes* which are meant to exchange small and fixed-size messages; and (iii) *portals* which enable one-sided dense data transfers. All these primitives have a synchronous behavior. However, for *mailboxes* and *portals*, asynchronous mode of operation is also supported, in LW Manycore that feature DMA engines. Finally, the *Memory Management Module* provides a virtual memory extension to 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. Furthermore, to enable address space management in user space, this module exposes routines for inspecting and updating paging structures.

The *Kernel Call Layer* lies on top of the *Modules Layer* and effectively exposes the functionalities of each module to user space. This topmost layer performs security checking and controls execution flow. Overall, kernel calls fall either one of the following two types:

(i) *Local Kernel Calls* that execute in the same core in which they were issued; and

(ii) *Remote Kernel Calls* that are forwarded to the kernel core, which in turn handles them.

This distinction on where kernel calls are executed identifies the asymmetry of the kernel. The controller module of the *kernel Call Layer* decides whether or not a given kernel call should be executed locally (i.e., in the slave core), or remotely (i.e., in the master core) based on a two-fold condition. If the kernel call changes structures of the requesting core, or if it accesses only read-only data structures of the kernel, then this kernel call executes locally, else it executes remotely. For serving remote kernel calls, the master core processes them in a FIFO fashion, and it relies on a kernel-land semaphore to control the requests coming from user-threads.

4.4.3 Memory System Module

The memory system module of the kernel wraps up the local memory system of a cluster in the underlying LW Manycore processor, to provide a flat address space abstraction to the overlying software layers. This module is based on a two-level paging scheme, and exposes operations that enable transparent manipulating of page structures. Figure 22 depicts the internals of the memory system module of the Nanvix microkernel. Overall, it has two components: the *Physical Memory Allocator* and the *Page Allocator*.

The Physical Memory Allocator manages allocation of physical memory. It operates on a page frame granularity and uses a bitmap data structure to keep track of which page frames are allocated/freed. Whenever a page frame is requested, the bitmap is searched for the first unset bit. Once found, this bit is set and it is used to compute the address of the now-allocated page frame. In contrast, when a page frame is released, the corresponding bit in the bitmap structure is unset, to indicate that the now-freed page frame is available for future allocations. We chose this scheme because it offers a compromise between performance and memory consumption. From the perspective of time performance, the Physical Memory Allocator takes linear time for allocating page frames and constant time for releasing them. From the memory consumption

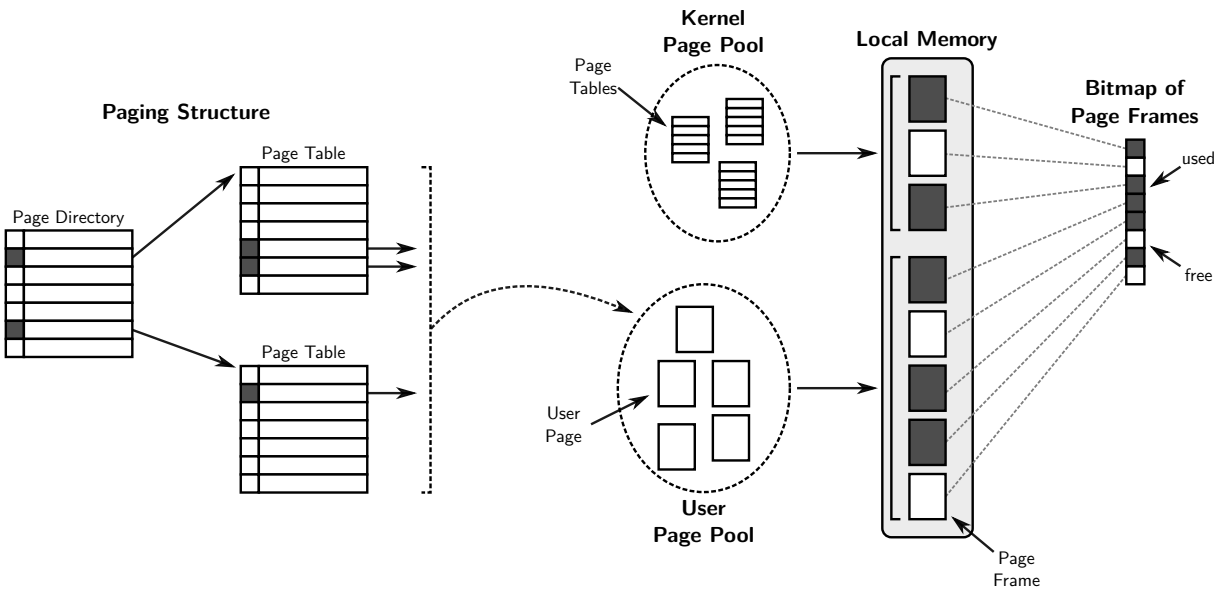


Figure 22 – Memory system module.

view, this allocator requires logarithmic space to maintain the bitmap data structure.

The Page Allocator effectively provides the flat address space abstraction as well as operation on it, to the overlying software layers. To achieve this, the Page Allocator relies on: (i) a two-level page table data structure that records the physical-to-virtual address mappings; and (ii) two logical pools of page frames (i.e., Kernel Page Pool and User Page Pool) that are used to satisfy user-space operations on the flat address space. When a page is allocated, The Kernel Page Pool is used for satisfying internal requests of memory allocations of the kernel. The User Page Pool is used for satisfying requests for allocating user-space memory.

The microkernel of Nanvix exposes four kernel calls to operate with the memory system module. The `upage_alloc` and `upage_free` enable user pages to be allocated and released, respectively. This enables user-level allocators. The `upage_map` and `upage_unmap` calls enable pages to be mapped and unmapped from virtual address ranges, and thus enable user-space address space management.

4.4.4 Thread System Module

The *Thread System Module* of the Nanvix microkernel has three main components: (i) the Thread Engine, which maintains threads; (ii) the Synchronization Facility, which exposes synchronization primitives for threads; and (iii) the Exception Engine, which enables exceptions to be handled at user-level. In the paragraphs that follow we detail the functioning of each of these components.

Thread Engine

The Thread Engine provides and manages threads at kernel-level. To this end, it relies on two per-thread data structures: a Thread Control Block (TCB) to store meta information of a thread, such as its identifier, state, starting routine and pending signals; and a stack to hold temporary variables and data of the thread itself. Both structures are dynamically allocated/released when threads are created/terminated.

The Thread Engine was designed from the ground up based on the design goals of the kernel (see Section 4.4.1). First, to keep the minimalism, the Thread Engine may spawn up to one thread on top of each core of the underlying cluster of the LW Manycore. This way, it does not need to handle thread preemption and scheduling. Still, core multiplexing may be provided by runtime libraries in user-level, because enough support is offered to enable multiple user threads to be mapped to a kernel thread. Second, to promote efficiency, the Thread Engine features two types of threads: interruptible and non-interruptible threads. The former are hooked up to the interrupt system of the underlying architecture, and thus may catch and handle hardware interrupts. On the other hand, the latter threads run with all hardware interrupt lines masked, and thus may run CPU-intensive code at full-speed, without getting interrupted. The number of interruptible and non-interruptible threads may be statically configured, and they are meant to run kernel routines and user-level code on their top, respectively.

The Thread Engine exposes a POSIX compatible interface that enables one to operate on kernel threads. The `thread_create` spawns a thread on an idle core of the cluster to run a target routine. In contrast, the `thread_exit` function enables one thread to terminate its execution and release its underlying resources (i.e., TCB and stack). The `thread_get_id` enables a thread to retrieve its unique identifier, which can be useful for indexing software resources. Finally, one thread may wait the execution of another by calling `thread_join`.

Thread Synchronization Facility

The Thread Synchronization Facility provides essential mechanisms to enable threads to synchronize their work. In summary, this facility exposes three abstractions, as well primitives to operate on them: spinlocks, condition variables and signals. The runtime system of Nanvix relies on this interface to build more complex constructs such as mutexes, semaphores and barriers.

Spinlocks wrap up atomic instructions of the underlying architecture and effectively enable mutual exclusion in a critical region. This is a non-blocking and non-reentrant abstraction, and it should be used when short waiting times are expected, otherwise CPU cycles are wasted. The kernel extensively relies on spinlocks internally, but user-space software typically use spinlocks in conjunction with condition variables, to enable blocking synchronization. The microkernel of Nanvix has three operations for spinlocks: (i) `spinlock_init` for initializing a spinlock; (ii) `spinlock_lock`, which locks a spinlock and thus prevents other threads to enter a critical region; and (iii) `spinlock_unlock`, which conversely unlocks a spinlock and hence allows other threads to enter a critical region.

Condition variables enable threads to wait for the occurrence of a specific event, thereby providing means for blocking constructs to be implemented. Condition variables on Nanvix present a strong similarity to POSIX ones and they works as follows. First, a condition variable is initialized by calling `cond_init`. Then, whenever a thread A wants to sleep on a condition α, is calls `cond_wait` with a locked spinlock L. This causes the thread to atomically release the lock of L and sleep in a queue until the target condition α happens. Then when another thread B calls `cond_signal` on α, thread A atomically reacquires L and resumes its execution.

Signals provide means for a thread to deal with asynchronous events. In a nutshell, it works as follows. First, a thread registers a callback function for a signal by calling `sigctl`. Then, whenever this this signal is triggered, the Thread Engine tweaks the execution flow of the recipient thread to invoke the callback function. Finally, the callback function does whatever it should do to handle the signal, and once it is finished it calls `sigreturn` to resume previous execution flow. Although simply put, this execution flow is challenge to support in Nanvix. The reason for this lies on the fact that the microkernel features non-interruptible threads. Therefore, the kernel gets a change to tweak the execution flow of the recipient thread only when the recipient thread voluntarily transfers execution control to the kernel, for instance when issuing a kernel call. Hence, to avoid long latencies in the delivery of signals, the kernel relies on software interrupts to notify recipient threads about signals, and thus force them to enter the

kernel. Noteworthy, software interrupts occur less frequently than hardware interrupts, and hence they have softer performance impacts on the execution of user-level threads.

Exception Engine

The Exception Engine wraps up the exception system of the underlying processor to enable both hardware interrupts (i.e., timer and network interrupts) and software exceptions (i.e., page faults and arithmetic logic) to be handled in user space. Runtime libraries of Nanvix rely on this feature to support system services that run across the clusters of the LW Manycore.

This module maintains a table of exception handlers in kernel space. User-level handler threads may register themselves in this structure by calling `excp_ctrl`. Either the same handler may be registered to multiple exceptions, thereby enabling custom dispatch logic to be implemented in user space; or one handler may be registered for each existing exception, thus allowing lower latency exception handling. Either way, once the handler is registered, it may call `excp_pause` to block until the target exception is raised.

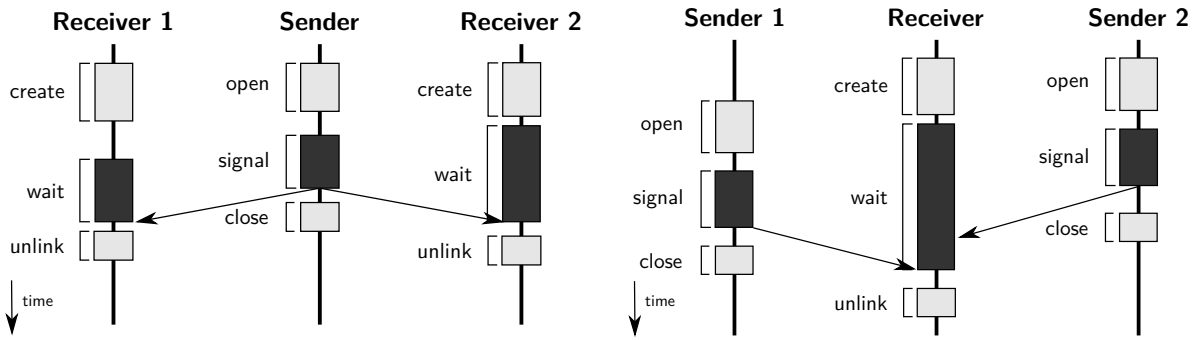
As soon as this event happens in any thread, hardware first traps to a exception dispatcher in kernel space. In turn, this routine checks the exception table and realizes that a user-level handler thread is registered for the exception that was just triggered, and thus it unblocks the concerning handler. When the user-space handler resumes, it uses information returned from `excp_pause` to parse and effectively do whatever it needs to process the exception. Once the handler completes its work, it calls `excp_resume` to return control to ask the kernel to resume the execution of the faulting user-level thread. Hence, the kernel, wipes out the user-level exception stack of the faulting thread, which resumes its execution flow.

4.4.5 Inter-Kernel Communication Facility

The Inter-Kernel Communication (IKC) Facility abstracts on-chip communication capabilities of LW Manycore by providing virtualization and multiplexing of the NoC links. Overall, this facility exposes three communication abstractions: (i) *syncs*, for enabling a process to signal and unlock another process remotely; (ii) *mailboxes*, for sending fixed-size messages with low latency; and (iii) *portals*, for handling dense data transfers with high bandwidth.

Synchronization Points

The *sync* abstraction (shorthand for synchronization point) provides the basis for peer synchronization. It works by having on one side multiple peers (i.e., receivers) to block and wait for peers on the other side (i.e., senders) to issue a notification. The notification itself does not carry any information other than the required to wake up the receivers, thus this abstraction works with fine-grain data. At system-level, *syncs* are used at system startup to synchronize subsystems and when a distributed application is launched. Furthermore, *syncs* may be used to build more complex synchronization structures such as distributed mutexes, semaphores and



(a) `1:N` mode. (b) `N:1` mode.

Figure 23 – Execution flow of *sync* abstraction.

barriers. The rationale for providing this abstraction in our facility is twofold. First, small amounts of data should be transferred around (i.e., tens of bytes) to synchronize peers. Thus, using a coarser-grain abstraction such as *mailboxes* to this purpose would be inefficient. Second, the on-chip interconnect of some LW Manycore include special hardware to enable low-latency inter-cluster synchronization (FU et al., 2016; DINECHIN et al., 2013). If the synchronization semantic is explicit, our facility may be implemented so as to better exploit the capabilities of the underlying hardware.

Figure 23 outlines the semantics of the *sync* abstraction. We propose two operating modes for it: `1:N` and `N:1`, which in turn define senders and receivers. In `1:N` mode (Figure 23a), there is a single sender that issues wake up notifications to multiple receivers waiting for them. Conversely, in `N:1` mode (Figure 23b), multiple senders issue notification signals to a single receiver. The set of operations available for each side of the communication is different. On the one hand, senders are allowed to `open`, `signal` and `close` a *sync*. On the other hand, receivers can `create`, `wait` and `unlink` a *sync*.

Mailboxes

The *mailbox* abstraction enables peers to exchange fixed-size messages with each other. The size of a message is designed to be small (i.e., hundreds of bytes), so that communication latency is reduced. This abstraction features an `N:1` semantic and works as follows. On one endpoint, a receiver owns a *mailbox* from which it reads messages. On the other endpoint, multiple senders may write messages to this *mailbox*. At system-level, *mailboxes* are used for exchanging control messages, which either encode simple operations or encapsulate meta-information of more complex tasks. For instance, the memory management subsystem may use a single *mailbox* message to request a remote peer to invalidate its page cache. On the other hand, one peer of the file system manager may rely on a message to pack information concerning a file read/write operation (i.e., name of the file, offset and read/write size). The dense data transfer is then carried out with a *portal* abstraction (see Section 4.4.5). Overall, we included this fixed-size *mailbox* abstraction in our facility to decouple small message exchanges from

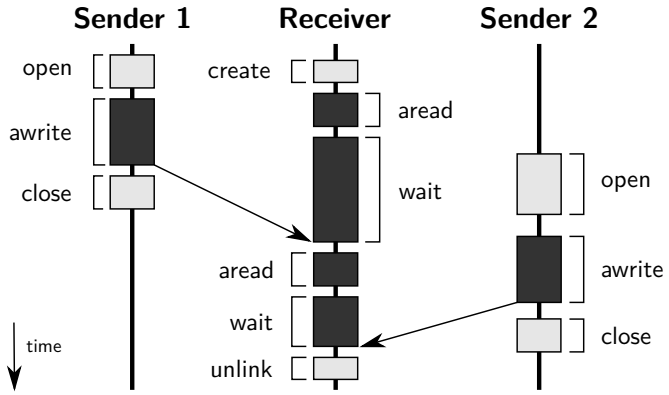


Figure 24 – Execution flow of *mailbox* abstraction (`N:1`).

dense data transfers. In this way, we enable low-latency communication among the peers of a distributed OS for LW Manycore.

Figure 24 details the semantics of a *mailbox*. On the receiver side, four operations are available: (i) `create`, for creating a *mailbox*; (ii) `aread`, for asynchronously reading incoming messages; (iii) `wait`, for waiting for any income message; and (iv) `unlink`, for destroying a *mailbox*. Conversely, the sender features three operations: (i) `open`, for establishing a connection with a remote *mailbox*; (ii) `awrite`, for asynchronously posting a message to a remote *mailbox*; and (iii) `close`, for terminating the connection with a remote *mailbox*. Noteworthy, the *mailbox* abstraction features an asynchronous read/write semantic. Nevertheless, synchronous reads may be achieved by issuing an `aread` followed by a `wait` operation. On the other hand, synchronous writes may be accomplished by having each side owning a *mailbox* and a send-acknowledge protocol implemented.

Portals

The *portal* abstraction allows two peers to exchange arbitrarily large amounts of data (i.e., thousands of bytes) with each other, with built-in support for receiver-side control flow. This abstraction presents an `1:1` semantic and works as follows. On one endpoint, a receiver owns a *portal*, from which it reads incoming data. On the other endpoint, a sender may write

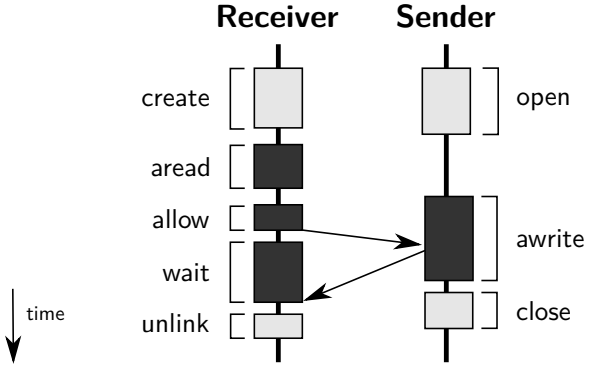


Figure 25 – Execution flow of *portal* abstraction (`1:1`).

data to this *portal*, once a connection with the remote receiver is established. This connection is explicitly established by the receiver itself by allowing a write on its *portal* from the sender. At system-level, *portals* may be used in a wide range of scenarios, specially when dense data transfers are needed. For instance, the process management may rely on this abstraction to deploy a binary file in a remote cluster or the file system manager may use *portals* for transferring file system blocks around. We included this abstraction in our communication facility to semantically support efficient dense data transfers. In this way, if the NoC of the underlying LW Manycore features special hardware to take care of this, system performance may be improved.

Figure 25 outlines the semantics of a *portal*. On the receiver side, five operations are available: (i) `create`, for creating a *portal*; (ii) `allow`, for authorizing a remote sender to write to the *portal*; (iii) `aread`, for asynchronously reading data; (iv) `wait`, for waiting incoming data; and (v) `unlink`, for destroying a *portal*. On the other hand, the sender features three operations: (i) `open`, for establishing a connection with a remote *portal*; (ii) `awrite`, for asynchronously writing data to a *portal*; and (iii) `close`, for terminating the connection with a remote *portal*.

4.5 System Services

The asymmetric microkernel of Nanvix handles resource management within the cluster of a LW Manycore and provides bare minimum abstractions such as threads and local memory management. In contrast, richer OS functionalities are provided by a set of distributed services that run on top of all running instances of this kernel. In this section, we turn our discussion to these system services. First, we list the design goals that we target (Section 4.5.1). Then, we present an overview of all system services that are provided (Section 4.5.2). Finally, we detail the internals of the services related to process management (Section 4.5.3), memory management (Section 4.5.4) and file system management ((Section 4.5.5).

4.5.1 Design Goals

System servers of Nanvix were designed to address the following goals:

(i) *Modularity*: be factored in minimum components; and

(ii) *Scalability*: be capable of handling requests from multiple processes, at the same time.

On the one hand, to approach modularity we factored system services in several standalone specialized components with non overlapping sets of functionalities and responsibilities. On the other hand, to deliver scalability, Nanvix features static elasticity. That is, the system may be configured to feature multiple instances of a given component and thus improve performance of a given system service.

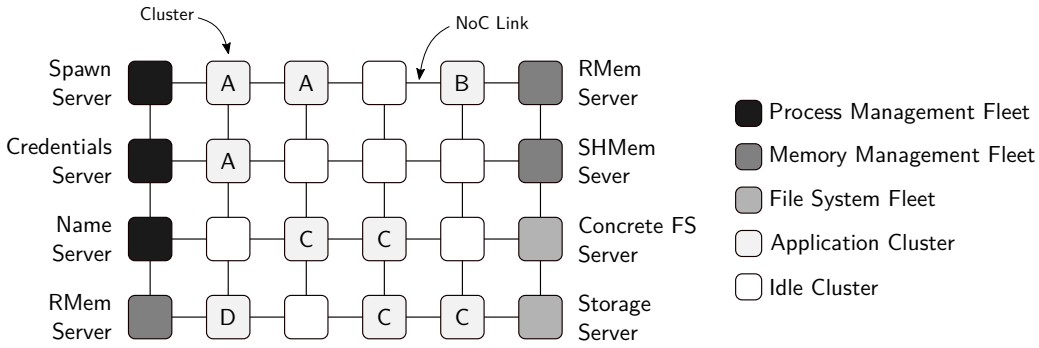


Figure 26 – Services of Nanvix running on a lightweight manycore processor.

4.5.2 Overview

In Nanvix, each OS functionality is implemented by a special system process called system server. These processes run in user-space, are spatially distributed across the processor and handle specific tasks, such as process scheduling, page allocation and file operations. In turn, fully-featured resource management capabilities are provided by a collection of system services, or fleets, that mutually cooperate. Figure 26 pictures a snapshot of Nanvix running in a LW Manycore processor. Overall, there are three fleets, which we introduce next: the process management fleet, the memory management fleet and the file management fleet.

The *Process Management Fleet* fleet exposes the process abstraction and handles a variety of concerned tasks, such as process creation, termination, scheduling and migration. Furthermore, this fleet also deals with authentication and provides high-level inter-process communication mechanisms like unnamed pipes and messages queues. The *Process Management Fleet* is composed by three servers: (i) the *Credentials Server*, which assigns access various system identifiers to processes; (ii) the *Spawn Server*, which orchestrates the utilization of clusters; and (iii) the *Name Server*, which maintains the disposal of processes in the processor. We detail the internals of this fleet in Section 4.5.3.

The *Memory Management Fleet* deals with physical memory utilization. Additionally, it provides virtual memory and shared memory abstractions, which are an important concern for programmability in LW Manycore processors. This fleet features two servers: (i) the *Remote Memory (RMem) Server*, which handles page allocation and swapping; and (ii) the *Shared Memory (SHMem) Server*, which keeps track of shared memory mappings. We examine the main functionalities provided by this fleet in Section 4.5.4.

The *File System Fleet* provides the abstraction of files and a set of operations on them. This fleet is composed by multiple servers: (i) the various *Storage Servers* which keep data; (ii) the *Concrete File System Servers* provide the implementation of a given file system, like Ext4, NTS and NFS; and (iii) the *Virtual File System* provides a uniform interface for accessing all concrete file system servers as well as well driver servers. We cover this fleet in Section 4.5.5.

4.5.3 Process Management Fleet

The *Process Management Fleet* of Nanvix manages processes and deals with their creation, scheduling, termination and authentication. This fleet is composed by three system servers: the *Spawn Server*, the *Credentials Server* and the *Name Server*.

The *Spawn Server* is the coordinator server of the *Process Management Fleet*: it orchestrates the utilization of clusters. Overall, it is responsible for launching user processes, as well as scheduling them. Furthermore, this server is also in charge of spawning other system servers at startup time and restarting them, in the event that any failure is encountered. The *Spawn Server* provides two important system calls: `spawn` `kill`. The former, enables one process to launch a new one, which can either be a system server or a user process. On the other hand, the `kill` system enables one process to send some signal to another process. This signal is directly delivered to the microkernel of the remote cluster and encodes some scheduling operation that causes the process to either abort, stop, suspend, pause or resume its execution flow.

The *Credentials Server* maintains authentication information about a processes, such as their its Process Identifier (PID), Group Identifier (GID) and User Identifier (UID). This server is queried by third party system servers and user processes whenever they should decide whether to authorize or block some operation. In summary, the *Credentials Server* provides fundamentals of security in Nanvix, by exposing two fundamental operations: `setids`, which sets credential information of a process; and `getids`, which does the converse operation and retrieves the concerned authentication information.

The *Name Server* tracks the placement of processes in the clusters of the underlying LW Manycore. To this end, it maintains a table of aliases that maps logical identifiers of a process to the cluster where this process is running. Logical identifiers (aliases) can either be the PID of the process or a string, but more importantly they uniquely identify a process in the system. Based on aliases, the *Name Server* exposes a name resolution protocol that enables other system servers as well as users processes to transparently communicate in Nanvix. The *Name Server* exposes three system calls to enable third parties to operate with aliases: (i) `name_link` links an alias to a process; (ii) `name_unlink` unlinks an alias from a process; and (iii) `name_lookup` resolves an alias.

In the upcoming sections, we examine how these servers cooperate to implement some important system functionalities, mechanisms, algorithms and protocols.

System Startup

System startup is handled by the *Spawn Server* and happens as follows. When Nanvix is built, the image of the system is crafted so as the binary of the *Spawn Server* gets loaded and started up by the hardware itself at boot time. Then, once the *Spawn Server* is up and running, this server relies on an internal table to launch other system servers. This table is called *inittab* and stores all the information that is required to carry out the aforementioned task, such as where

the concerned binary files are placed in the system image, where each server should be deployed (i.e., in which cluster) and in what order servers should be brought up.

The actual startup procedure comes down to traversing this table and loading servers one after another. Unfortunately however, this process is not straightforward, because depending on the characteristics of the underlying LW Manycore processor, multiple instances of a *Spawn Server* are loaded boot time. For instance, in some architectures that are supported by Nanvix, I/O Clusters should be started up individually. Therefore, to cope with this intricacy, the *Spawn Server* is designed to work as it is detailed next.

At compile time when constructing the *inittab*, system servers are associated with a ring level. Servers within the same ring level do not have any runtime dependencies among themselves. In contrast, servers with a given ring level depend on functionalities provided by servers of a lower ring level. Based on this scheme, at system startup, instances of a *Spawn Server* work in a lockdown step, each one loading a subset of servers in the *inittab* starting from the lowest ring level towards the highest one. Whenever a *Spawn Server* spawns all process in a certain ring level, it blocks on a global barrier and waits for the other instances of *Spawn Server* to complete their job. System startup completes when all the ring levels are traversed.

Process Spawn

In Nanvix, new processes are created with the `spawn` system call. This operation constructs a process image from a given binary executable file and schedules this process for later execution. In comparison to POSIX system calls, `spawn` is semantically equivalent to a `fork` followed by an `exec`. Overall, the *Spawn Server* leads the execution of the `spawn` system call in three steps, which are outlined in Figure 27 and detailed in the next paragraphs.

First, as soon as the *Spawn Server* receives a `spawn` request from a process (parent), it queries the *File Management Fleet* for the given binary file. If such file exists, then it parses the executable headers to determine how much core memory the process image takes. This size is given by the total length of the code plus the static data segments and provides an estimation of how much memory should be allocated for the process, in the first place.

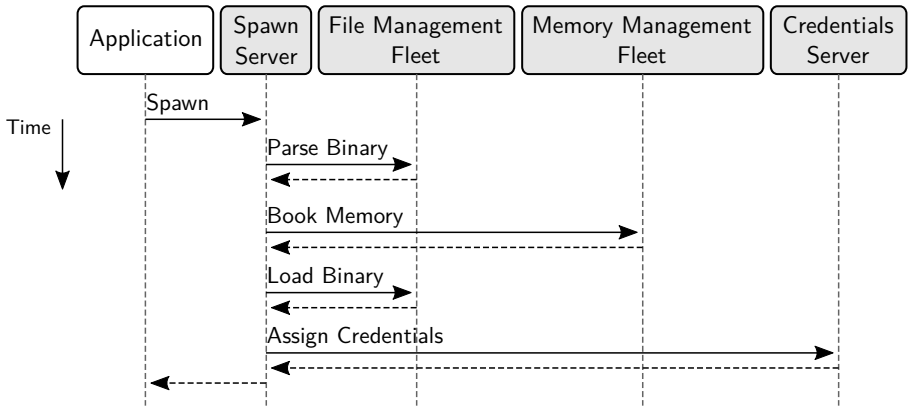


Figure 27 – Process spawn in Nanvix.

Second, with the information on the length of the process image, the *Spawn Server* proceeds with the construction of the process address space. For this, it asks the *Memory Management Fleet* for some to memory to accommodate the page tables, and it loads the executable file as follows. First, the *Spawn Server* books enough memory to accommodate the process image. This prevents the complexity of rolling back the operation in the case of not enough memory is available. Next, starting from the very first data block towards the end of the file, the *Spawn Server*: (i) asks the *Memory Management Fleet* for a previously-booked page; (ii) loads the loads the contents of the executable file to this page; and (iii) updates the page tables of the process accordingly.

Finally, with the file loading completed, the *Spawn Server* contacts the *Credentials Server* to assign credentials to the newly-created process (child). By default, the child process is assigned to a unique PID, but it has its GID and UID set to the same value of its parent. Nevertheless, if the process wants, it can change later its GID and UID with the `setids` system call. Anyway, with credentials set, the *Spawn Server* registers this PID in the *Name Server* as a canonical alias for the process, and places the process in the scheduling queue. Eventually, the scheduling algorithm will run and select this process to deploy for execution in a cluster.

Process Scheduling

Process scheduling is accomplished by a collaboration between the *Spawn Server*, the *Name Server* and the *Memory Management Fleet*. The first server is responsible for choosing which processes to execute, as well as where they should run (i.e., which cluster). The second server maintains a map of where processes are running, and it is updated accordingly whenever the placement of processes in the manycore processor changes. Finally, the *Memory Management Fleet* is used as backup storage for the image of processes that are not running.

The scheduling protocol is triggered under multiple circumstances, whenever a process changes its state. Figure 28 summaries this flow in a state diagram, and we next discuss how the scheduling protocol behaves in each of these state transitions. As soon as a new process is created, it is placed in the scheduling queue of ready processes. This state indicates that the process image is backed up in the *Memory Management Fleet* and it is not running, but the process is eligible for execution. When the scheduler selects a ready process to run, it instructs

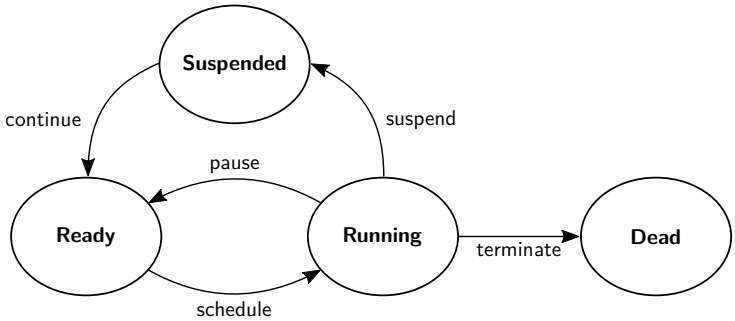


Figure 28 – States of a process in Nanvix.

the *Memory Management Fleet* to offload the process image to a cluster, it assigns this process a running time quota and resumes execution.

In turn, the process runs until either of the following situations occur: (i) its running time quota has run out; (ii) it receives a suspend signal; or (iii) it terminates. On the one hand, when the first situation happens, the *Spawn Server* checks the scheduling queue of ready processes. If this queue is empty, the server simply reassigns more running time quota to the process and let it continue executing. Otherwise, the *Spawn Server* issues a pause signal to preempt the process. This freezes the process image, stores it in the *Memory Management Fleet*, releases all underlying hardware resources that were being used and places back the process in the scheduling queue of ready processes.

On the other hand, the second situation arises in a variety of cases. For instance, when a system administration utility running asks for suspension to make some cluster available for other process; or when a system server asks the suspension of that process because some resource is not yet available. Anyway, the process image frozen and stored in the *Memory Management Fleet*, similar as when it is preempted. However, instead of placing the process in the scheduling queue of ready processes, the *Spawn Server* places the process in the scheduling queue of suspended ones. Processes in this latter queue stay there until they receive a `CONTINUE` signal from some other process, which moves the suspended process to the scheduling queue of ready processes

Finally, the third situation happens either when the process has completed what it had to do, or when it receives an abort signal from an authorized process. In the first scenario, termination voluntary happens by having the process calling the `_exit` system call. The second scenario is triggered whenever the process receives an `ABORT` signal. This may be sent any other process with enough privileges and it is often triggered because the process has performed some erroneous operation. Anyway, regardless the reason for which the `ABORT` signal was sent, the *Spawn Server* triggers the process termination protocol, which we detail later in this section.

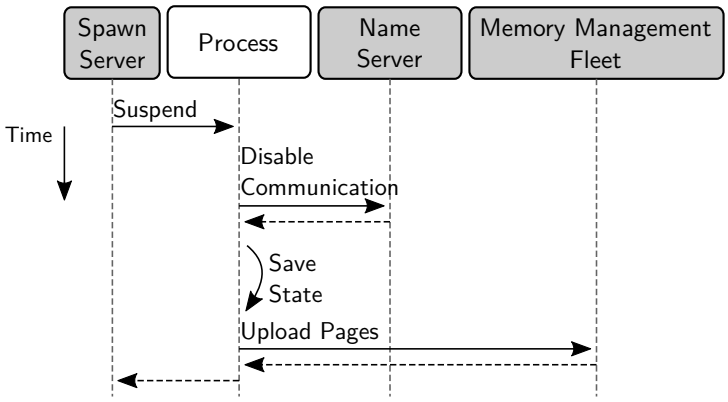


Figure 29 – Process migration in Nanvix.

Process Migration

Process scheduling requires that processes are migrated from one cluster to another. Nanvix has two operations to address this need in a distributed memory architecture, as is the case for LW Manycore processors: the offloading of a process image from the *Memory Management Fleet* to a cluster; and the uploading of a process image from a cluster to the *Memory Management Fleet*. These operations introduce an important distinction in the process scheduling of Nanvix, and are only possible thanks to the features that are exposed by the underlying asymmetric microkernel.

Overall the kernel of Nanvix has two calls to specifically support process scheduling in a distributed memory architecture: (i) `save`, which suspends the execution of all user threads and dumps the kernel state in a well-formatted structure called `kstate`; and (ii) `restore`, which does the converse operation by restoring the kernel state from a `kstate` structure and resuming the execution of user threads.

Based on these two kernel calls, offloading and uploading of process images are possible as follows (Figure 29). For uploading a process image, the *Process Spawner* sends a `SUSPEND` signal to the target process. This signal cannot be ignored nor dropped, and it is immediately caught by the runtime system of Nanvix. The signal handler of the runtime proceeds by running a suspension protocol. First, it disables communication to all system services by notifying the *Name Server* that the process is being suspended. This causes any other processes or system servers that address the process the suspending process to block. Next, the runtime system of Nanvix calls `save`, which suspends the execution of all user threads, freezes the kernel state and saves this to the `kstate` structure. Finally, the runtime system contacts the *Memory Management Fleet* and uploads to it all pages holding user-space data of the process, such as the code, static data, stack and heap, as well as the `kstate` structure.

Process Termination

Process termination happens when the process completes ts work or when it receives an `ABORT` signal. When either of these situations occur, the runtime system first shutdowns all system daemons that are running alongside with the process. This operation causes each system daemon to contact its respective system server and release resources that are allocated to that process. For instance, for closing all files that are opened by the terminating process, the File System Daemon contacts the *File System Server* to handle the operation. Conversely, the Memory Management daemon contacts the *Memory Management Server* to close any shared memory mappings and release all physical memory that is allocated to the process. More importantly, during the service shutdown, the runtime system proceeds from the outermost ring of execution, towards the innermost. This is the reverse order in which daemons were launched at process spawning and it is required since system services in Nanvix may be composed by a collection of services in turn. As soon as all the system services were shutdown, the runtime system

invokes the `_exit` system call to release all resources that are allocated in the microkernel for the terminating process.

Name Resolution

The *Name Resolution* protocol is handled by the *Name Server* and it is invoked whenever a process alias must be resolved to the logical cluster identifier. This happens when two processes are communicating via any named IKC primitive, such as *mailboxes* and *portals*. Thanks to this protocol processes may communicate regardless their physical placement in the processor. The *Name Resolution* server exposes three operations to deal with names:

(i) `name_link` assigns a symbolic name to the calling process;

(ii) `name_unlink` removes a symbolic name of the calling process; and

(iii) `name_lookup` which resolves a symbolic name into the location of a process.

Based on these operations, name resolution happens as follows. First, process *A* invokes the `name_link` call to link the symbolic name *process-a* to itself. This operation is locally routed to the name resolution daemon, which in turn performs some sanity checks and handle the request to the *Name Server*. Once the server receives the request and ensures that the symbolic name is unique in the system, it links the symbolic name to process *A*. From this moment on, all processes in the system may refer to *A* using the name *process-a*. For instance, if process *B* sends a mailbox message to process *A* using the symbolic name *process-a* the location of *A* if firstly resolved by the name server and then forwarded to the concerned cluster. When process *A* terminates, it invokes the `name_unlink` operation to remove the symbolic name *process-a* from the name lookup tables.

4.5.4 Memory Management

The Memory Management Fleet administrates physical memory and exposes a flatten memory view over the underlying distributed memory architecture. Furthermore, this fleet provides: (i) virtual memory capabilities, thereby allowing processes to work on address spaces that are larger than the physical one; (ii) and shared memory abstractions, which enables one-sided communication between processes. The Memory Management Fleet features three major components: (i) MMS Client; (ii) RMem Server; and (iii) SHMem Server.

The MMS Client provides the required support to the overlying POSIX libraries so as a standard API may be exposed to user-level programs. The MMS Client itself comprises three subsystems. The Virtual Memory Allocator (VMem Allocator) manages the address space of the program, by explicitly manipulating underlying page tables. The Virtual Memory Cache (VMem Cache) maintains a local cache of those pages that are mostly used by the program. Finally, the Virtual Memory Stub (VMem Stub) interfaces with the RMem and SHMem servers, forwarding to them requests from the overlying subsystems. The RMem Server act as remote backup storage

for pages which are not used by the programs in the system. This server takes care of physical memory allocation and well as ownership and access permission tracking. Finally, the SHMem Server maintains the shared page mappings in the system and plays the role of coordinator of VMem Cache. To further discuss the interactions between these three components, we take a close look in some fundamental protocols of the *Memory Management Fleet* next.

Page Allocation

When a process is created, the system allocates and loads pages with the code, static data and stack segments of the program in one cluster of the LW Manycore. Then, whenever the process requests more memory by calling `malloc()` the page allocation protocol is triggered.

At this time, as soon as the C memory allocator starts and finds out that more heap space should be allocated, it invokes the VMem Allocator. In turn, the VMem Allocator checks the virtual layout of the program to decide whether or not it may increase the heap size. If by expanding the heap no other segment of the program gets overlapped (i.e., stack, static data or text), then the VMem Allocator proceeds by increasing the heap break of the program and booking the underlying pages. In this latter step, the VMem Allocator does not actually allocate memory, but it rather tag underlying pages as «demand zero». These pages will be allocated later, on-demand when handling page faults, as soon as the program starts writing/reading to/from them. Once the booking step finishes, the VMem Allocator returns the new break value for the heap to the C allocator, which in turn works out to fulfill the request made by the program.

Page Sharing

Page sharing is achieved by having processes to create shared memory regions and attach these regions to their address space. Figures 36 details this execution flow. First, the program invokes shm_open() to open/create a shared memory region. The POSIX SHMem module proceeds by searching in its table of locally opened shared memory regions for a region that matches the target one. If such memory region is found, then then its identifier is returned. Otherwise, the opening/creation request is handled to the VMem Stub, which in turn forwards the task to the SHMem Server.

Once the SHMem Server receives the request, it searches in its table of globally opened SHMem regions for a region that matches the requested one. If such region is found, then it asserts access permissions access on that region, and it returns an identifier for that region for the requesting VMem Stub. However, if the requested SHMem region is not found, it means that the region does not exist yet, and thus it should be created. Therefore, the SHMem Server allocates an entry in its table for accommodating the new region, sets the access permissions accordingly, truncates the size of the SHMem region to zero and returns an identifier.

With the identifier of the newly created SHMem region, the program calls `ftruncate()` to change the size of the region. To do so, the POSIX SHMem module updates the meta-information about the region and handles the task to the SHMem Server, through the VMem

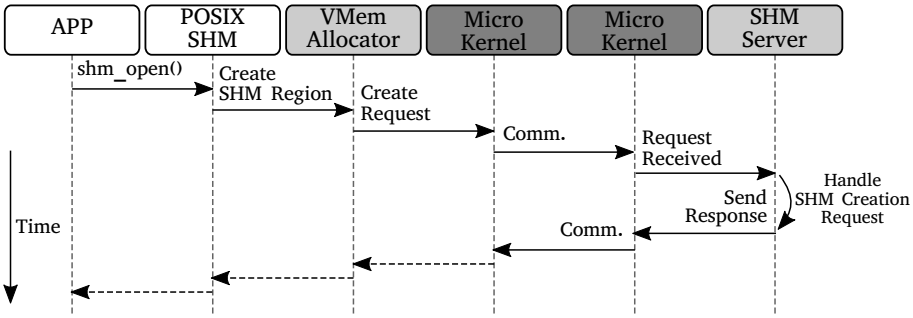


Figure 30 – Shared memory region creation in Nanvix.

Stub. As soon as the server receives this request, it locates the corresponding entry in its table of global opened SHMem regions and updates it accordingly:

- If the new size is larger than the current size of the SHMem region, which is the case for recently created regions, then its size is updated, but underlying page frames are not allocated. They will be allocated on-demand (see Section (4.5.4)).
- If the new size is smaller than the current size of the SHMem region, then the region size is updated and the underlying page frames are released, by requesting so to the RMem Server.

Finally, the program calls `mmap()` to attach the newly created SHMem region to its address. The POSIX SHMem module itself does no more than checking the parameters of the call, before handling handles the mapping task to the VMem Allocator. This later subsystem starts by first finding a valid address range (i.e., does not overlap other segments of the program) in which the target SHMem region may be attached. Once the address range is allocated, than the VMem Allocator proceeds by booking all the underlying pages, by tagging them as «demand fill» and “demand fill”. Page frames will be allocated or loaded from the RMem Server on-demand, as soon as the process writes to this SHMem region and a page fault occurs.

Page Fault Handling

Figure 31 outlines the execution of a program that raises a page fault exception. This may happen due to the one of the following three conditions:

(i) the process tried to access a protected memory address;

(ii) the process attempted to read/write from/to a page that is not yet allocated; or

(iii) the process tried to access a memory address that lies in a page that is not locally mapped.

Either way, as soon as a page fault is triggered, execution traps to the kernel, which in turn performs sanity checks in the address range. If the process is unauthorized to access that memory address (i.e., first case), then the kernel aborts execution. Otherwise, the exception is forwarded to the user-space page fault handler that lies in the VMem Allocator. When this

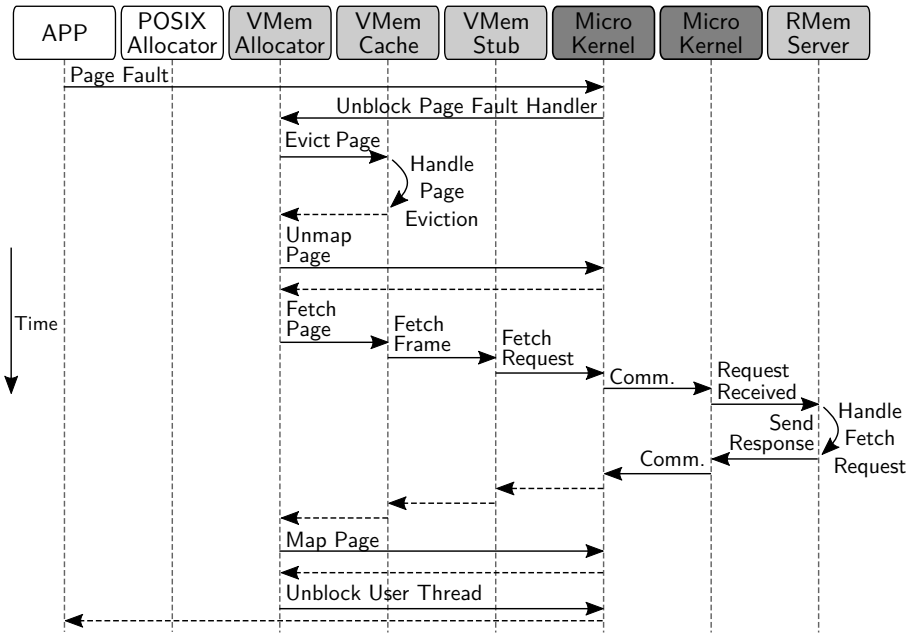


Figure 31 – Remote page fetch in Nanvix.

handler is launched, it searches the page tables of the process and realizes that the faulting address lies in a page that is tagged as «demand zero» or «demand fill» (i.e., case two and three, respectively). Therefore, the VMem Allocator for a slot for the VMem Cache to accommodate the page that is about to be brought in and handles the request to the VMem Stub.

On the server side, as soon as the request arrives on that server, the underlying physical memory is searched for an unused page frame. In the case where such a frame is found, then the RMem Server computes a logical remote frame number for it, and sends this number back to the VMem Stub. Otherwise, it sends an abort reply which will cause the faulting program to terminate. When the VMem Stub receives the positive response from the server, with the newly allocated remote page frame number, it returns this information to the cache. In turn, the VMem Cache stores this number in the cache slot and returns a local pointer for that page to the VMem Stub. The VMem Allocator, with the local pointer, computes the logical physical frame and interacts with the kernel to manipulate page tables and link the physical address returned and finally returns from the page fault handler to user space.

4.5.5 File Management

The File System Fleet exposes, manages and organizes files. In Nanvix, files consist in a central abstraction for applications and are used for structuring information, as well as for modeling physical devices and software resources. Figure 38 presents an overview of the File System Fleet. It has three components, which we introduce next.

The Storage Servers drive storage devices and expose a uniform function interface to other components of the system. For each existing type of storage device in the architecture, there is a corresponding Storage Server that handles that device type. Therefore, more importantly than

detailing the internals of Storage Servers (which may greatly vary) we highlight the interface that is exposed. Overall, two system calls are available: (i) blkread, which fetches a block from the underlying storage device; and (ii) blkwrite, which does the converse operation.

The Concrete File System Servers interact directly with the Storage Servers to provide the implementation of a given file system specification, like Ext4, NTFS or FAT. Concrete File System Servers expose a set of programming-friendly system calls that enables the creation and deletion of files, as well as reading and writing to them. Finally, the Virtual File System Server wraps up all the Concrete File System Servers to provide a single and uniform file system tree to the system. Thanks to this processes and applications may interchangeably manipulate files without any knowledge about the geometry and operations of the underlying concrete file system. Noteworthy, since Storage Servers and Concrete File System Servers depend on the technical specifications of the device and file system respectively, in the remainder of this section we focus our discussion to detail the internals of the Virtual File System Server.

Virtual File System

The Virtual File System of Nanvix features an hierarchical structure and it is constructed on top of two central abstractions, as it is shown in Figure 39: superblocks and I-Nodes. The implementation of these abstractions are delegated to Concrete File System Servers, and the Virtual File System Server relies on them to expose a fully-fledged file operations to applications, such as creation, deletion, listing, reading and writing of files.

On the one hand, superblocks keep essential information of a concrete file system such as residing device, number of files, maximum file length and size of data blocks. In summary, there

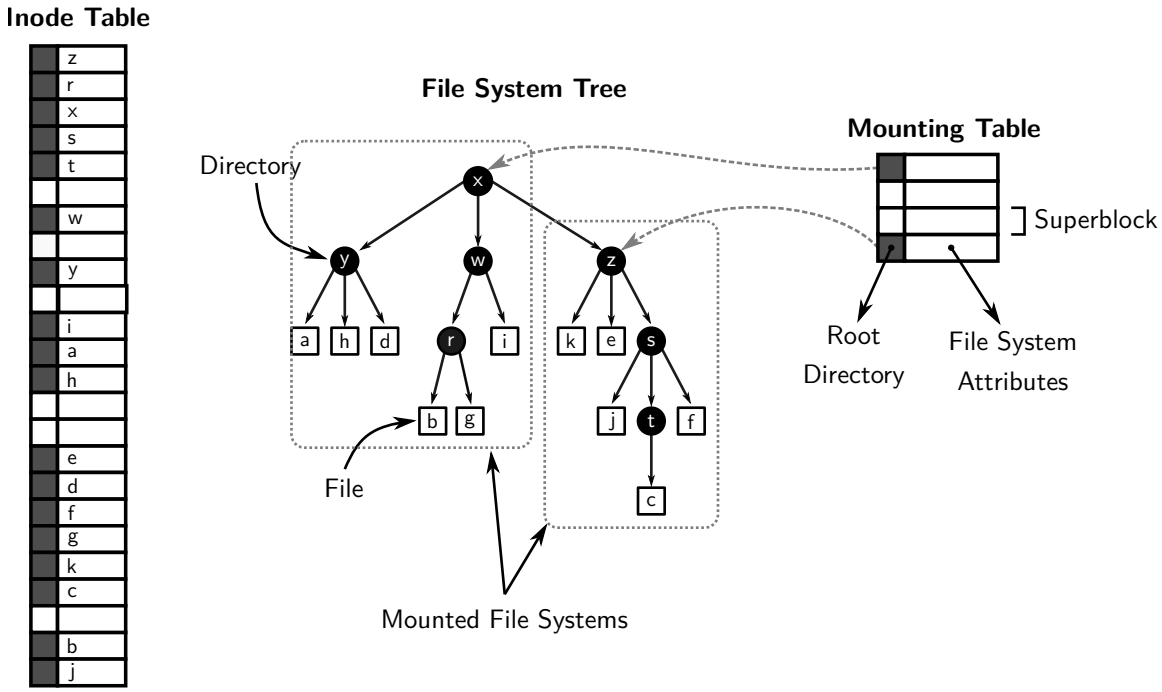


Figure 32 – Virtual file system of Nanvix.

are three sets of operations associated to superblocks. First there are mount and unmount, which causes the root of a file system to be mapped/unmapped on a node of the virtual file system. Second, there are ialloc and ifree, which allocates/releases an I-Node in the underlying concrete file system. Finally, there is iname, which traverses a concrete file system and converts a path name into an I-Node.

On the other hand, I-Nodes store all fundamental information concerning a file, like its type, owner, access permissions, current size, time of last modification and pointers to data blocks. Data blocks are assigned using an hierarchical addressing scheme, where the first seven data blocks are directly addressed, and next ones are addressed via either double or triple indirections. The way that blocks pointers are actually used depend on the type of the file itself. If it is a regular file, then blocks pointers point to data blocks filled with user data. If the file is a directory, however, block pointers point to data blocks that store directory entries. Finally, if the file is a special file, the block pointers hold the minor and major numbers of the referred device

There are three main operations associated to I-Nodes. The first operation (inode_read), fetches the I-Node from the underlying Concrete File System Server to the Virtual File System Server. To this end it takes as an argument the serial number of the target I-Node and the device on which it resides. The Virtual File System Server uses the information to contact the concerned Concrete File System Server. Once the I-Node is brought in, file operations such as read/write are enabled and take place on the Virtual File System Server. Conversely, the second operation (i.e., inode_write) operation writes back an I-Node to the concrete file system in which it resides. This operation takes as a parameter a pointer to the target I-Node and effectively flushes the metadata of in the I-Node to the underlying storage and it is typically invoked when a file is closed on explicitly flushed by a process. Finally, the inode_off operation translates a file offset into a data block pointer, and effectively enables the Virtual File System to read/write data to the concerned portion of a file, that is stored in the Storage Server.

4.6 Libraries and Runtimes

On top of the system services, Nanvix provides a collection of libraries and runtime systems, to provide a rich and standard programming environment in LW Manycore. Bellow we present an overview of them.

libNanvix This is a low-level runtime system that wraps up system calls and it is conceived to be the common ground to other user-space libraries and runtime systems. Furthermore, it provides the user-level thread implementation as well as important extensions to the thread system, such as thread local storage, semaphores, local barriers and condition variables.

uLibc This is a lightweight C library extracted from Red Hat Newlib Project citeNewLib. It provides essential operations for dealing with strings, file streams, memory allocation and more. Noteworthy, the programming environment that is exposed enables standard C applications to be transparently ported to Nanvix, without any significant changes.

libOMP This a parallel programming environment ported from the GNU libGOMP Project (GNU, 2020) that enables one to easily exploit data and task parallelisms using the fork-join model. This environment was engineered on top of the thread system of the Nanvix microkernel and exposes directives for creating parallel regions and loops, critical sections and atomic operations. With this OpenMP runtime system, software engineers may rapidly exploit and parallelism with clusters of a LW Manycore.

lwMPI This is a narrowed implementation of the MPI specification (version 3.1) to Nanvix (ULLER et al., 2020a). Our runtime system is provided on top of inter-cluster communication abstractions exposed by Nanvix (i.e., mailboxes, portals and sync) and currently supports for communicators, groups of communication, error handlers, and point-to-point synchronous communication. More importantly, with this interface, performance of all clusters in the LW Manycore may be thoroughly exploited by an application.

4.7 Implementation Insights

The source code of Nanvix is available at <https://github.com/nanvix> under the MIT License and is organized in multiple repositories. We highlight the following:

- In <https://github.com/nanvix/hal> is the implementation of the HAL.
- In <https://github.com/nanvix/microkernel> is the implementation of the microkernel.
- In <https://github.com/nanvix/libnanvix> is the implementation of the libNanvix.
- In <https://github.com/nanvix/ulibc> is the implementation of the uLibc.
- In <https://github.com/nanvix/multikernel> is the implementation of system services.
- In <https://github.com/nanvix/libmpi> is the implementation of the lwMPI.

All the repositories have a similar directory structure. The `build` directory hosts the build system. The `contrib` directory holds references to external modules. The `include` directory keeps all header files. The `src` store source files. Nanvix was written in C and assembly and has about 150 k and 1 k lines of code, in each language respectively[2]. The GNU Compiler (GCC) infrastructure is the toolchain that we officially support, and it leads to a memory footprint of about 128 Kilobytes (kB) for our OS. Notwithstanding, since Nanvix was written in compliant ANSI C, it should not be hard to enable compilation with other toolchains, such as the LLVM compiler infrastructure. The source code is shipped with a comprehensive regression infra-structure to enable one to rapidly test for enhancements and bug fixes made to Nanvix. In addition, based on this testing infra-structure, upstream source code repositories are linked to a continuous integration system, which runs nightly.

[2] Measured with `cloc` utility which is available at <https://github.com/AlDanial/cloc>.

Concerning the deployment of Nanvix, it relies on a configuration file to describe how the OS and applications should be disposed on the target hardware. In a nutshell, this works as follows. First, when the system and applications are compiled, a set of binary files is generated. Some of these contain the core image of system services, and other were generated from the source code of a user application. Next, based on the deployment configuration file, the build system of Nanvix generates a system-image file that precises where each binary should be placed. Finally, once the platform is powered on, the system-image file is loaded into the hardware to with the that configuration.

The layout of this system-image file as well as the procedure for loading it is hardware-dependent and thus varies from one platform to another. Nevertheless, it is important to remark that default configuration files are written so as to most exploit efficiency of each platform. By default, those clusters that have more I/O and memory connectivity (i.e., I/O Clusters) are reserved for system binaries, whereas those with more computing capabilities (i.e., Compute Clusters) are used to deploy user binaries. Furthermore, in some platforms that feature a few counts of I/O Clusters, but with multiple cores, we bundle several services on the same service in a single binary so as to maximize hardware utilization. For instance, in Kalray MPPA-256, there are four quad-core I/O Clusters which are used as follows. The east and west clusters runs the Memory Management Fleet. The north cluster runs the Process Management Fleet. The South Cluster runs the File System Fleet. In Appendix we present an overview discussion on the platforms that are currently supported by Nanvix. We give insights on architectural intricacies of each of these platforms and how Nanvix was designed to cope with this challenges.

As a final remark, we highlight some challenges and limitations on the current implementation of Nanvix. First, among the platforms that are currently supported by our OS, Kalray MPPA-256 is the only one that has a silicon implementation, and precisely for this reason we chose it as the main platform for our experimental assessment (see Section 6). Unfortunately however, this platform imposes some technical limitations in software design that do not enable the implementation of some protocols that we discussed in this chapter, such as process migration and dynamic process spawning. Indeed, we left this as a future work, as discussed in Section 7.2. Second, we oftentimes lacked on enough technical information on the hardware of the underlying platforms, thereby significantly slowing down the development of Nanvix. For instance, in Kalray MPPA-256 we faced important challenges when trying to directly access DMA hardware, and in OpTiMSoC we struggled with building arbitrarily large processor configurations.

4.8 Concluding Remarks

In this chapter we presented Nanvix: a distributed OS that we designed from scratch to target programmability and portability challenges in LW Manycore processors. Our system supports multiple architectures (i.e., k1b, x86, OpenRISC, ARMv8 and RISC-V), runs on silicon LW Manycore processors, exposes rich abstractions and high-level programming interfaces to

user applications. Nanvix is an open-source software [3] and is the result of a joint collaboration between PUC Minas, UFSC and UGA.

To cope with the architectural challenges of LW Manycore processors, Nanvix embraces a multikernel design that is structure on top of multiple instances of an asymmetric microkernel. Traditional system functionalities, such as memory management, process management and file system are provided as special processes (system servers) that run in user-space. User-level applications may invoke these functionalities by calling the RPC interface that is exposed by the Nanvix runtime system (libNanvix).

Nanvix enhances programmability in LW Manycore in several ways. First, to address the limitations that concern of the constrained memory system of these processors (i.e., distributed memory architecture and limited on-chip memory), our system relies on a rich memory management approach that is based on a DPS. This is a novel solution that we devised for managing memory of a LW Manycore processor and it works as follows. The local memories of a LW Manycore processor are considered page caches that can store data of whichever processes. Thanks to this DPS, virtual memory is enabled, as well as memory sharing over different physical address spaces.

Second, to reduce kernel interference in user-applications and thus negatively impact performance, our system embraces an asymmetric kernel design. With this approach, local caches are not polluted with kernel data structures and execution flows, and user threads may better perform. Furthermore, the additional overhead of user-kernel transitions is mitigated.

Third, to enable multi-application deployment in LW Manycore, as well as to provide resource isolation between them, Nanvix relies on the principle of hardware-level isolation (ASMUSSEN et al., 2016). In this approach, only one process is deployed at a time in a cluster and NoC inherently provides resource isolation. Furthermore, it is worth noting that by removing multiprogramming support in clusters, we additionally cope with the reduced amount of memory that is there available. With small local memories, it is unfeasible to simultaneously deploy multiple processes in a cluster at once.

Finally, Nanvix features a lightweight communication facility that manages the on-chip interconnect and exposes primitives capabilities (PENNA et al., 2021). Three communication abstractions are exposed, which of which providing efficient support for a different communication granularity. Thanks to these primitives, we are able to efficiently support multiple communication patterns and mechanisms, such as fine- and coarse-grain transfers, one-to-one, all-to-one and one-to-all, RPC and message passing.

In additional to these improvements in programmability, Nanvix also enhancements portability in LW Manycore processors. On the one hand, the HAL of Nanvix enables system-level portability to a wide-range of architectures On the other hand, at application-level, Nanvix supports multiple standard programming interfaces and runtime libraries, such as POSIX, MPI and OpenMP.

[3] Publicly available at: `https://github.com/nanvix`.

We specifically assess these improvements introduced by Nanvix in the upcoming chapters. In Chapter 5 we present an evaluation methodology for quantifying the portability, programmability and performance of our system. In Chapter 6 we present and discuss our experimental results.

5 EVALUATION METHODOLOGY

In this chapter, we present the evaluation methodology in which we relied to assess Nanvix. First, we summary the goals that we aimed (Section 5.1). Then, we present the experimental programs that we considered (Section 5.2). Next, we detail the experimental LW Manycore processor that we used (Section 5.3). Finally, we describe the experimental design that we followed (Section 5.4).

5.1 Evaluation Goals

Our evaluation methodology was elaborated so as to answer the following three main questions, which are aligned to the goals of this work (see Section 1.3):

Q-1 How does Nanvix improve programmability in lightweight manycore processors?

a) How does it abstract hardware intricacies?

b) How does it enable resource multiplexing?

c) How does it handle resource balancing?

d) How does it provide resource isolation?

e) How does it enable resource sharing?

Q-2 To what extent does Nanvix enhance portability?

a) How does Nanvix enable portability in system level?

b) How does Nanvix enable portability in application level?

Q-3 What is the performance of Nanvix?

a) What is the performance of the communication management subsystem?

b) What is the performance of the memory management subsystem?

c) What is the performance of the process management subsystem?

d) What is the performance of user-level applications?

5.2 Experimental Programs

We considered several experimental programs to specifically assess the performance of Nanvix and thus answer evaluation question Q-1. Overall, these programs may be grouped according to their characteristics and evaluation purposes as follows:

Table 8 – Experimental programs used for assessing Nanvix

Group	Program	Brief Description
Synthetic Programs	`fence`	benchmarks latency of *syncs*
	`mail`	assesses latency of *mailboxes*
	`cargo`	measures throughput of *portals*
	`lkcall`	clocks latency of local kernel calls
	`rkcall`	times latency of remote kernel calls
	`forkjoin`	evaluates the cost for creating and terminating threads
	`knoise`	assesses the interference introduced by the kernel in a user application
	`lookup`	measures the latency for determining the physical location of a process
	`heartbeat`	benchmarks the latency for sending an alive message to the process manager
	`stream`	evaluates the read/write bandwidth from/to local memory
	`rstream`	assesses the read/write bandwidth from/to remote memory
	`pginval`	clocks the latency for invalidating an entry of the page cache
	`pgfetch`	times the latency for transferring a page from the remote memory server
System Utilities	`diff`	computes the least subsequence between two strings
	`grep`	searches for a patter in a text
	`sed`	finds and replaces patterns in a text
	`sort`	sorts numerical data
Scientific Programs	`km`	clusters a set of points in several groups
	`gf`	runs a Gaussian filter on a 2D image
	`fn`	computes the friendly numbers over an interval range

(i) *Synthetic Programs* are implemented on top of the the runtime system of Nanvix (libNanvix) and they are designed to make the most efficient use of the system. These programs focus on the measurement of specific system features and we employed them to understand the performance of the individual subsystems of Nanvix.

(ii) *System Utilities* consist of a set of Unix utilities that were ported to Nanvix, on top its C library (uLibc). These programs force the interaction among subsystems as well as between user and system libraries. We relied on these programs to understand what performance to expect when running typical system applications in Nanvix.

(iii) *Scientific Programs* are composed by kernels of scientific benchmark suites that were ported to Nanvix. These programs use the distributed and parallel programming libraries that are available in our OS (lwMPI and PThreads), and enabled us to determine the performance that Nanvix may deliver to domain-specific applications.

Table 8 shows the main characteristics of all programs that we considered, and in the next sections, we present each of them. All programs are available at <https://github.com/nanvix>.

5.2.1 Synthetic Programs

Synthetic Programs exercise specific features of our OS and were implemented on top of libNanvix. There are 13 programs in this group, and they may be further classified regarding to the subsystem that they aim to assess: Communication Management System, Process Management Subsystem, Memory Management Subsystem and File Management Subsystem.

Communication System

`fence` It assesses the latency of synchronization with our *sync* abstraction. It launches up to N processes that repeatedly sync up with each other.

`mail` This program assesses the latency for exchanging messages with our *mailbox* abstraction. It launches up to $P = N + M$ processes, and makes each one of the N processes to send messages to each one of the M processes.

`cargo` This program measures the throughput for transferring dense data blocks with our *portal* abstraction. It launches up to $P = N + M$ processes, and makes each one of the N processes to transfer data blocks to each one of the M processes.

Process System

`lkcall` This program clocks the latency of a local kernel call, which executes in the same core that the requesting user thread is running. To this end, we used the `thread_get_id` kernel call, which retrieves the identifier of the underlying thread.

`rkcall` This program times the performance of a remote kernel call. Remote kernel calls execute in a dedicated core (i.e., master core) of the underlying cluster, and they consist in the key feature of our asymmetric microkernel. In this benchmark, we considered a void remote kernel call of Nanvix.

`forkjoin` This program measures the overhead for creating and terminating threads. The cost for supporting these operations is important, if the OS aims at efficient support for parallel programming frameworks.

`knoise` This program evaluates the interference introduced by the kernel in the execution of a user application. It works as follows. Let n be the number of cores available in the underlying cluster. Therefore, this benchmark launches x compute-intensive threads in x of these cores, and in the $n - x$ remainder cores it launches kernel-intensive tasks (i.e., threads that issue remote kernel calls in a tight loop).

`lookup` This program benchmarks the latency for resolving the location of a process, which is the cluster in which that process is running. To this end, this program launches a process that dispatches several name lookup requests to the name server. This program relies on a `1:1` ping-pong communication protocol using a *mailbox*.

`heartbeat` This program assesses the latency for informing the process manager that a process is alive. For this, it launches multiple processes that iteratively send heartbeat signals to the name server. This program relies on a `N:1` all-gather communication using a *mailbox*.

Memory System

`stream` This program evaluates the read/write bandwidth of local memories. It launches multiple threads that concurrently read/write from/to a local memory region.

`rstream` This program assesses the read/write bandwidth of remote memories. It launches multiple processes that concurrently read/write from/to a remote memory region.

`pginval` This program benchmarks the latency for invalidating page cache entries of a remote process. It launches multiple processes that iteratively read data from the same shared page, and then forces a page cache invalidation on these processes by writing to this page. This program relies on a `1:N` broadcast communication protocol using our *mailbox* abstraction.

`pgfetch` This program assesses the time for transferring a page from the memory server to a process. It launches a process that fetches several pages from the remote server by iteratively allocating memory chunk, reading this chunk and releasing it. This program relies on a `1:1` ping-pong communication protocol. A *mailbox* is used for exchanging meta-information concerning pages and a *portal* is used for transferring pages.

5.2.2 System Utilities

System utilities were ported on top of the uLibc and they exercise realistic interactions among subsystems of Nanvix as well as between user and system libraries. This group has the following programs.

`diff` This Unix utility computes the difference between two strings with the least common subsequence algorithm. It relies on a memoization table and features a nearest neighbor memory access pattern.

`grep` This Unix utility searches for a pattern in a text using a dictionary-based string matching algorithm. It relies on a pre-computed table to efficiently skip mismatched strings and presents a sequential memory access pattern.

`sed` This Unix utility finds and replaces all patterns in a text. It relies on a similar algorithm used by `grep` to find patterns and uses an auxiliary buffer to place the resulting text. It features a sequential access pattern.

`sort` This Unix utility sorts numerical data using the quicksort algorithm. It presents random access pattern.

5.2.3 Scientific Programs

Scientific programs are built on top of the lwMPI in Nanvix. These applications were extracted from the CAP Benchmarks suite (SOUZA et al., 2017) and have different characteristics. The following programs are in this group.

`fn` This program finds all subsets of numbers in a range $[n,m]$ that share the same *abundance*. The abundance of n is the ratio between the sum of divisors of n by n itself. FN implements the *MapReduce* parallel pattern and has tasks with regular loads. The problem is predominantly CPU-bound.

`gf` This program reduces the noise of an image by applying a matrix convolution operation with a special two-dimensional Gaussian mask to the image pixels. GF performs the *Stencil* parallel pattern to equal-sized parts of the image, thus being CPU-intensive and having a medium communication intensity.

`km` This program implements the k-means clustering technique, which is often employed in data analysis. KM gets a set of n points in real d-dimensional space and randomly split them into k partitions. Then, it applies the *Map* parallel pattern to distribute points and replicate data centroids between the Compute Clusters. The irregular workload is both CPU- and memory-bound. Since each iteration must update data centroids, this kernel operates with high communication intensity.

5.3 Experimental Platform

Among the architectures supported by Nanvix, we chose Kalray MPPA-256 as the experimental platform in this work. This is a single-chip commercial LW Manycore processor that features most of the characteristics discussed in Section 2.1.

The Kalray MPPA-256 processor (Figure 50) features 272 general-purpose cores and 16 firmware-cores, called PE and Resource Manager (RM), respectively. The processor is built

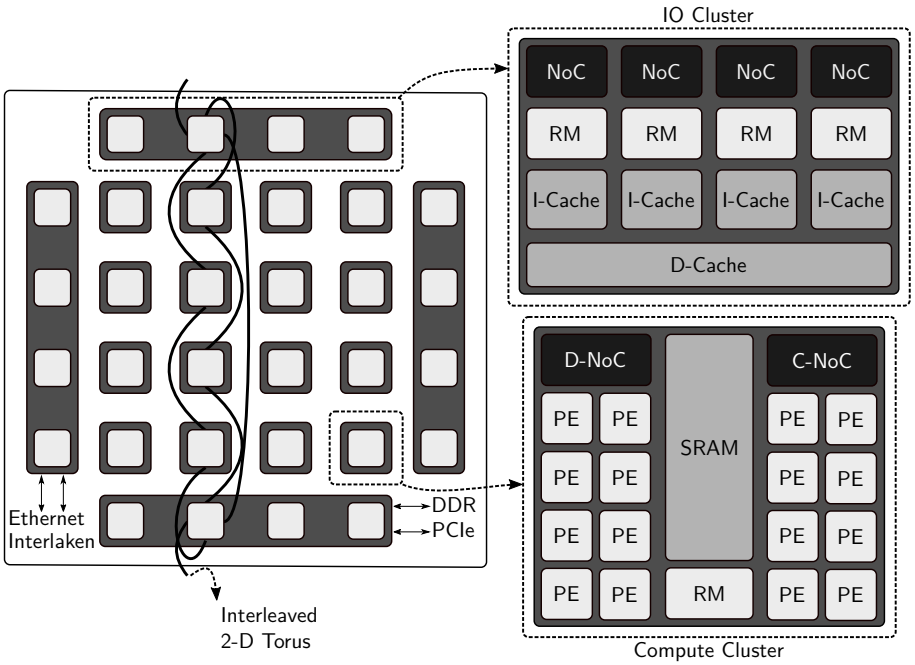


Figure 33 – Architectural overview of the MPPA-256 LW Manycore processor.

with 28 nm CMOS technology and it runs at 400 MHz. All cores implement a 64-bit proprietary instruction set, present a 5-issue VLIW pipeline, 8 kB Instruction Cache (I-Cache) and Data Cache (D-Cache), and feature a software-managed MMU.

The 288 cores of Kalray MPPA-256 are grouped into 16 Compute Clusters, which are intended for computation, and 4 I/O Clusters, which are designed to provide connectivity to peripherals. Each Compute Cluster bundles 16 PE, one RM, two NoC interfaces and a 2 Megabytes (MB) of local SRAM (128 kB per core). In these clusters, hardware cache coherence is not supported. In contrast, I/O Clusters have 4 RM, 8 NoC interfaces and 4 MB of SRAM (1 MB per core). Two of these clusters are connected to a different DRAM controller, and the other two are attached to PCI and Ethernet controllers. Compute Clusters are not attached to a global memory and they all have private address spaces. Thus, Compute Clusters have to exchange hardware messages by one of two different interleaved 2-D torus NoC to carry out communications: (i) a Control NoC (C-NoC) that features low bandwidth and is intended for small data transfers; and (ii) a Data NoC (D-NoC) that presents high bandwidth and is dedicated to dense data transfers.

What concerns software development in Kalray MPPA-256, this processor is shipped with a patched version of GCC 4.9.4 and Binutils 2.11.0. No OS is provided by the vendor, and software engineers should rely on a proprietary and non-compliant runtime environment to write their applications (DINECHIN et al., 2013), if not using Nanvix. Furthermore, regarding OS kernel implementation, system engineers are required to rely on a proprietary hypervisor from Kalray. This hypervisor runs on the firmware cores of the processor and intermediates all low-level operations. Noteworthy, Kalray hypervisor cannot be changed nor configured, and thus it imposes additional challenges in OS construction for this LW Manycore processor.

5.4 Evaluation Methods

We relied on quantitative and qualitative approaches to answer the evaluation questions that we aimed (Section 5.1). For evaluating programmability and portability we carry out a discussion that unveils how these aspects are enabled by Nanvix. In contrast, to assess the performance of Nanvix we run the programs that we detailed previously (Section 5.2) on the Kalray MPPA-256 LW Manycore processor (Section 5.3). Noteworthy, by discussing experimental results of system utilities and scientific programs we are effectively showing that Nanvix delivers programmability (Q-1) and portability (Q-2) to LW Manycore.

In experimental evaluation, we gathered time measurements using hardware performance counters to enable monitoring with minimum interference. On the other hand, to retrieve energy consumption measurements, we relied on a device that is externally attached to the board of the processor. This device measures power dissipation on the board and comprises statistics for all cores, NoC and other on-chip resources. Furthermore, for each experiment, we carried out 10 trials to eliminate undesired warm-up effects, and then executed 30 trials to collect results. All results that are presented in this work are based on a confidence interval threshold of 95%.

6 EXPERIMENTAL RESULTS

In this chapter, we present the results of our evaluation. We first discuss outcomes from the specific assessment that we carried for subsystems of Nanvix, and then we move our analysis to the results of broader evaluation that encompasses the interaction among subsystems, as well as user and system libraries. This chapter is organized as follows:

- In Section 6.1, we present our evaluation on the communication management subsystem.
- In Section 6.2, we discuss our assessment outcomes for the memory subsystem.
- In Section 6.3, we analyze our evaluation results for the process subsystem.
- In Section 6.4, we present our assessment in user-level.
- In Section 6.5, we summarize the most important findings of our experiments.

6.1 Communication System

In this section we evaluate the communication management subsystem of Nanvix. First in Section 6.1.1, we discuss how it improves programmability and portability by contrasting the interface exposed by Nanvix with the one provided by vendor libraries that are shipped with Kalray MPPA-256. Also, we discuss how *syncs*, *mailboxes* and *portals* are used to construct POSIX compliant communication/synchronization abstractions. Then in Section 6.1.2, we examine the performance and scalability of our communication primitives.

6.1.1 Programmability and Portability

Nanvix exposes three main primitives that may be used for process communication and synchronization: (i) *syncs* let processes to wait for signals and thus synchronize with one another; (ii) *mailboxes* allow processes to exchange fixed-size messages (iii) *portal* enable processes to exchange arbitrary amounts of data. Overall, these primitives improve programmability in several points. First, they do not require the programmer to know in which cluster a given process

```
1  void send_message(int dest, int rxbuf, char (*msg)[128])
2  {
3    char path[64];
4    sprintf(path, "/mppa/channel/128:%d/%d:%d", rxbuf, dest, rxbuf);
5    int ch = mppa_open(path, O_WRONLY);
6    mppa_write(ch, msg, 128);
7    mmppa_close(ch)
8  }
```

Snippet 6.1 – Sending fixed-size messages in MPPA using with Kalray's baremetal libraries.

```
1  void send(pid_t dest, char *msg, size_t n)
2  {
3    int outbox = kmailbox_open(dest);
4    kmailbox_write(outbox, msg, n);
5    kmailbox_close(outbox)
6  }
```

Snippet 6.2 – Sending fixed-size messages in MPPA using Nanvix.

is running. Second, they handle communication flow control. Third, they handle allocation of low-level hardware resources. Finally, they enable the multiplexing of NoC interfaces.

Code Snippets C.1 and C.2 illustrates some of these improvements by showing how fixed-size message exchanging may be achieved in Nanvix, in contrast to vendor libraries that are shipped with Kalray MPPA-256. When using the latter library C.1, the following steps are involved. First, the path for remote endpoint should be constructed accordingly (line 4). This specifies what hardware transmitter buffer to use (`rxbuf`)), the size of messages that will be sent by and the physical ID of the destination cluster. Once this setup is done, a communication channel to the endpoint is opened (line 5), the message is sent (line 6) and the end communication channel is closed (line 7). Noteworthy, in this flow the software engineer has to deal with low-level aspects of the hardware, when setting up the endpoint. In contrast, when using the communication system of Nanvix (Code Snippet C.2) three simpler steps are required: (i) open a mailbox to the target process by supplying its identification number (line 3); (ii) write the message through the mailbox (line 4); and (iii) close the mailbox (line 5).

In respect of portability, the primitives that are exposed enable the constriction of POSIX synchronization/communication abstractions as follows:

- Signals are implemented on top of *mailboxes*. In this context, messages carry information of a concerned signal, such as its number.
- Semaphores are implemented on top of *mailboxes* and *syncs*. The former is used to transmit increments/decrements values, whereas the latter is used to send wake up notifications.
- Message queues are implemented on top of *mailboxes*. In this case, messages encapsulate the payload that is being passed around, as well as the priority of a message.
- Pipes are implemented using *portals* and *sync*. The former is used transfer data around, whereas the latter is used to send/wait wake up notifications.

6.1.2 Performance

To analyze the performance of communication system of Nanvix, we considered as our baseline an implementation of communication primitives using *mailboxes* only. That is, an implementation in which *syncs* and *portals* are backed up by *mailboxes*. The rationale for this

decision lies on the fact that other OSes that target LW Manycore rely only in this type of abstraction to carry out all kinds of on-chip communication (PENNA et al., 2021).

Mailbox

Figure 34 presents the latency for reading/writing to a *mailbox*. These results were obtained with the `mail` program, when varying the size of a message payload from 64 to 1024 Bytes (B). Overall, we observed the following: (i) latency proportionally increases with the size of the message payload; and (ii) the latency grows slower for reads than for writes. These results uncover an important design aspect: the message size should be kept small to minimize the communication latency. For 64 B payloads, this yields to 26 µs reads and 27 µs writes. In contrast, for 128 B payloads, hits yields to 29 µs reads and 31 µs writes.

At first, one would argue in favor of 64 B payload messages, because *mailboxes* are meant for low-latency communication and payloads of this size yield to minimum latency. However, we decided for a different configuration due to the following important observation. In addition to the payload of a message, we should transfer a message header that has a fixed size of 16 B. Therefore, for a 64 B payload we transfer 80 B (20% overhead); and for 128 B payload we transfer 144 B (12% overhead). Hence, when considering both the size of the payload and the communication latency, a 128 B payload yields to the optimum configuration – minimum latency with the smallest overhead. For this reason, from now onward, all results that picture *mailboxes* are based on a 128 B message payload configuration.

Figure 35 presents the throughput for reading and writing fixed-size messages from/to a *mailbox*. These results were obtained with the `mail` program, when increasing the number of communicating processes. We noted that both operations deliver linear throughput scalability, with a capacity for reading and writing about 3.2 k and 3.8 k messages per second, respectively. These results show the inherent scalability of NoC, in contrast to traditional interconnects such

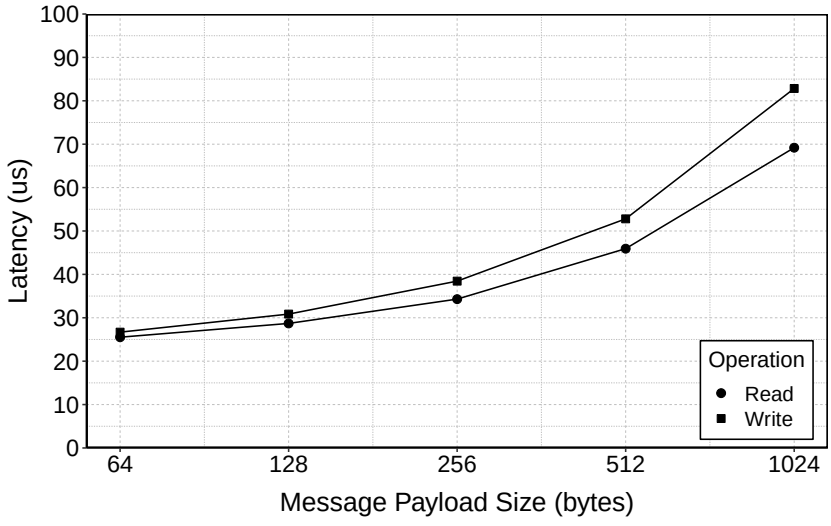


Figure 34 – Mailbox latency when varying message size.

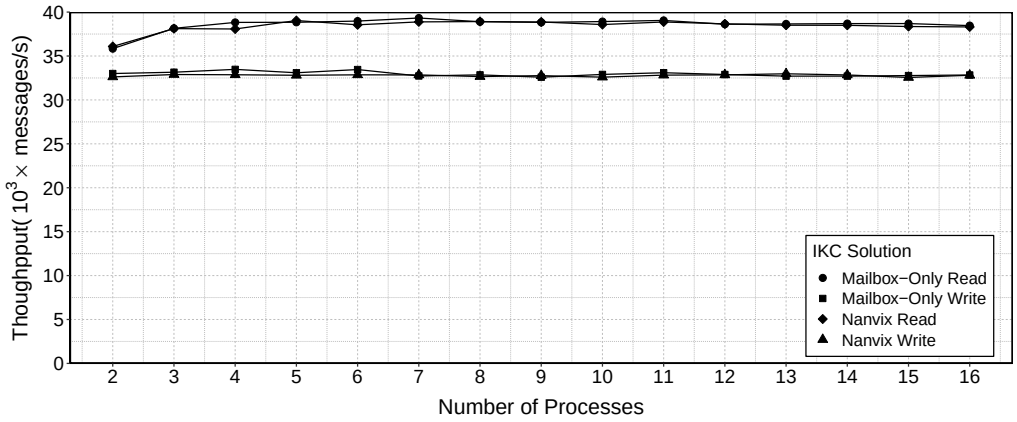


Figure 35 – Mailbox throughput for fixed-size messages.

as buses and crossbar switches. Notwithstanding, the throughput gap between the two operations is justified by technical limitations imposed by the hypervisor of Kalray MPPA-256. We were not able to fully utilize DMA engines and thus our implementation relied on polling for writes.

Synchronization Point

Figure 36 shows the latency for sending and receiving synchronization signals between multiple processes using either *mailbox* (baseline) or *sync*. These results were obtained with the `fence` program, when varying the number of processes from 1 to 16. Overall, reached the following two important conclusions. First, we observed an important difference between *mailbox* and *sync*. Synchronization points handle signals 64× faster than the baseline solution. The rationale for this is three-fold: (i) synchronization signals require a few bytes of data to be transferred; (ii) *mailboxes* work with a coarser transfer granularity (128 B) in contrast to *sync* (4 B); and (iii) our abstraction relies on the C-NoC to transfer data, which delivers lower latencies than the D-NoC, which is narrowed for bandwidth and is used by *mailboxes*.

Second, when we contrasted the latency of the two operations while increasing the number of processes, we observed that the *wait* latency stays constant at 8 µs, whereas the *signal*

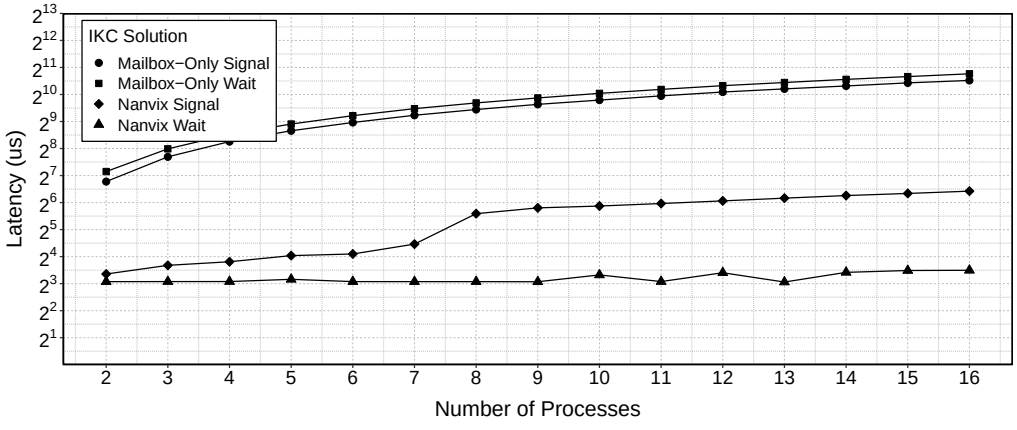


Figure 36 – Sync latency scalability for synchronization signals.

latency increases from 10 µs to 64 µs. The rationale for this comes from the fact that *syncs* are backed up by the C-NoC of Kalray MPPA-256, which can receive multiple signals in parallel. Bottom line, this result shows that synchronization points enable one to better exploit fine grain communication in a LW Manycore processor.

Portal

Figure 37 presents the read and write bandwidth when using either *mailbox* (baseline) or *portal*. These results were obtained with the `cargo` benchmark, while varying the number of communicating processes. In general, we observed that both abstractions provide linear bandwidth scalability, with reads being more efficient than writes. This observation is aligned with the previous conclusions on latency scalability for fixed-size message passing with *mailboxes*.

When contrasting the throughput performance of the two implementations for the *portal* abstraction for working with dense data transfers, however, we spotted an important difference between them: *portals* achieved roughly 60 MB/s and 40 MB/s for read and write operations, respectively, as opposed to 8 MB/s and 5 MB/s achieved by *mailbox*-based *portals*. The rationale for this lies on the fact that *portals* are backed up by larger transfer buffers than *mailboxes*. Thanks to this, in dense data transfers, fewer chunks are needed to carry out the transfer, and thus the overall communication overhead is smoothed out and higher bandwidth is achieved.

Figure 38 pictures the throughput for reading data when using either a fixed-size *mailbox* of 128 B (baseline) or a optimally-sized *portal*. These results were obtained with the `cargo` program and show up the peak bandwidth of our *portal* abstraction, and were used to dimension the size of buffers for this abstraction.

Overall, we observed that the throughput delivered by *mailbox* is constant, whereas the throughput achieved by *portal* increases with the transfer size. A three-phase behavior can be noticed in the plot: (i) for transfer sizes ranging from 64 B to 1024 B, a linear increase in throughput is observed; (ii) for transfer sizes ranging from 1024 B to 8192 B, a sub-linear increase in throughput is observed; and (iii) for transfer sizes bigger than 8192 B, a constant

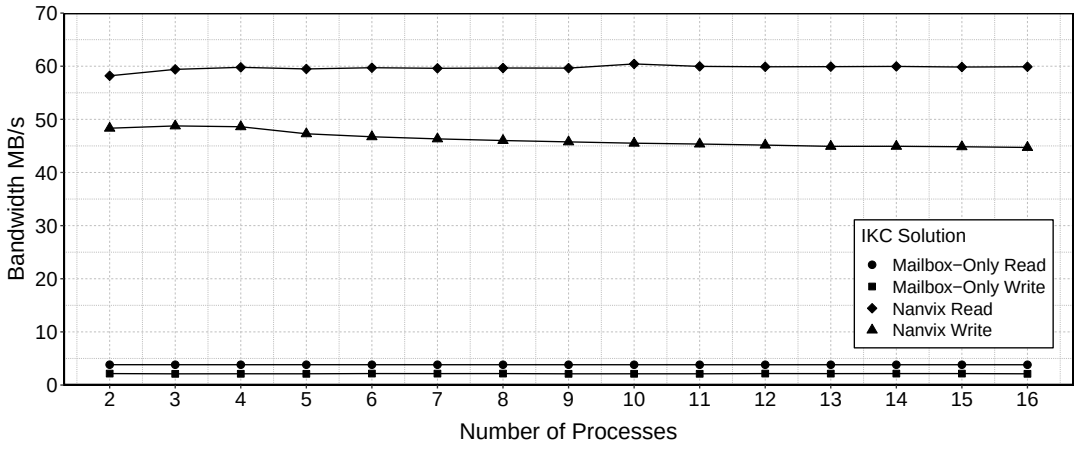


Figure 37 – Portal bandwidth scalability for dense data transfers.

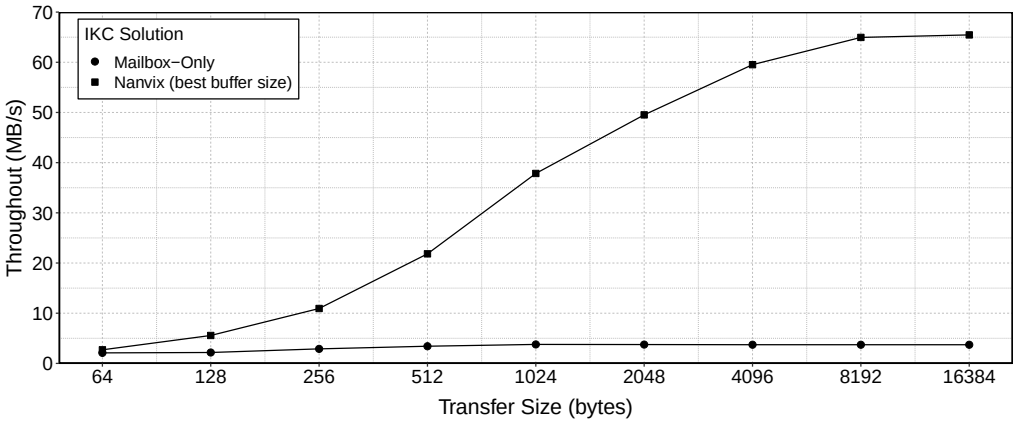


Figure 38 – Mailbox and portal throughput when varying transfer size.

throughput is achieved. The rationale for this is that transfer sizes of up to 1024 B are efficiently handled by the NoC, which becomes saturated beyond that point.

Bottom line, we relied on these conclusions to dimension the buffer size of *portals* in Kalray MPPA-256. We set this to 4 kB, in order to achieve a best performance for page transfers involving the memory management subsystem.

6.2 Memory System

In this section we evaluate the memory system of Nanvix. First, we discuss how it improves programmability and portability in LW Manycore processors through the DPS (Section 6.2.1). Then, we uncover the performance of memory accesses, as well as we analyze the latency for invalidating and fetching pages (Section 6.2.2).

6.2.1 Programmability and Portability

Recall that LW Manycore processors have important constraints concerning their memory system. First, they feature a distributed memory architecture: memory modules are physically spread across the processor, the address space is not unique, and the memory is not globally accessible by all cores. Access to remote memory is only possible with message passing communication through a on-chip network. Second, access times to memory modules are not uniform. It is faster to access a local memory bank than to access data (via the NoC) that is

```
1  size_t n = (1 << 30);          /* 1 GB                          */
2  char *mem = malloc(n);         /* Allocate a memory area.       */
3
4  for (size_t i = 0; i < n; i++) /* Manipulate the memory area. */
5    mem[i] = 0xff;
6
7  free(mem);                     /* Free the memory area.         */
```

Snippet 6.3 – Use case for address expansion.

```
1  /* Allocate some shared memory. */
2  int fd = shm_open("shared-memory", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
3  ftruncate(fd, 4096);
4  char *shmem = mmap(NULL, n, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)
5
6  /* Write to shared memory. */
7  for (size_t i = 0; i < 4096; i++)
8    shmem[i] = 0xff;
9
10 munmap(shm, size);
11 shm_unlink(fd);
```

Snippet 6.4 – Use case for address space sharing.

stored on a remote memory module. Finally, the amount of on-chip memory is very limited. For instance, in the Kalray MPPA-256 processor, on-chip memories are 2 to 4 MB big. In summary, these architectural characteristics impose challenges in both software programmability and portability. We further discuss how our memory system overcomes these issues in the next paragraphs by examining two representative use-case examples.

In Snippet C.3, we present some C code that dynamically allocates a big memory area (1 GB) using the standard memory allocator from the C library (line 2), manipulates this memory area (lines 4 and 5) and releases it (line 7). This example would work in most modern systems which have enough virtual memory. However, OSes for LW Manycore currently miss rich system-level memory management and neither these processors have enough on-chip physical memory to fulfill this operation. As a consequence, only programs that have memory footprints small enough to fit in the physical memories of a LW Manycore cluster may be ported without introducing significant changes to their source code. Nanvix enables this use-case example to run without any modifications in Kalray MPPA-256.

In Snippet 6.4, we present a fragment of a C program that pictures the basics for address space sharing between two processes. It works as follows. First, it allocates a 4 kB shared memory region, which is typed as readable and writable by programs of the same user (line 2). Then, it attaches this shared memory region to the address space of the calling program (line 4). Next, it writes to this shared memory region (lines 7 and 8). Finally, it releases this shared memory region (lines 10 and 11). Unfortunately, current OSes for LW Manycore do not support this execution flow. The only way that programs have to communicate is through message passing. This greatly changes the design of programs and thus compromises programmability. Nanvix supports shared memory regions and enables processes to use them to communicate.

6.2.2 *Performance*

The greatest feature exposed by the memory management subsystem of Nanvix is the virtual address space through the distributed memory architecture of a LW Manycore processor. To this end, the DPS takes care of keeping in local memories those pages that are mostly accessed by processes, while offloading to remote memory those pages that are not heavily used. Due

to this, there is an important distinction between accessing local pages and remote ones: the former is faster to access than the latter. Therefore, to examine the performance of the memory management subsystem we analyze the bandwidth for raw reads/writes to the virtual address space, considering both cached and uncached accesses. Then, we analyze the performance of the page invalidation and fetching protocols, which play an important role in our DPS solution.

Uncached Memory Access

Figure 39 outlines the uncached read/write bandwidth achieved by our memory system, when varying the number of processes that concurrently access the remote memory. In addition, we also present the bandwidth for reads/writes to local memory, when varying the number of threads that concurrently access the local memory. These results were obtained with the `stream` and `rstream` programs, with the page cache disabled.

Overall, we observed that both local and remote read/write bandwidth scale up with the number of concurrent streams (i.e., threads or processes). In remote memory accesses, the bandwidth achieved by a single process is 9.38 MB/s and 18.76 MB/s, when performing reads and writes, respectively. In contrast, for local memory access, the bandwidth achieved by a single thread is 58.30 MB/s, for both reads and writes. In terms of memory latency, this represents 53.31 ns for remote accesses in contrast to 17.15 ns local ones.

When contrasting the bandwidth between local and remote memory accesses, we noted an important performance gap. Local memory accesses are about 10× and 5× faster than remote reads and remote writes, respectively. The rationale for this lies on the architectural characteristics of LW Manycore processors. On the one hand, local memory is directly accessed with trivial machine instructions (e.g., `load`/`store`). On the other hand, remote memory is accessed through the NoC, which involves software-level message exchanging as well as data buffering. This leads to two important observations, in regards to the performance of memory-intensive applications:

(i) Local memory must be efficiently managed.

(ii) Data locality should be extensively exploited.

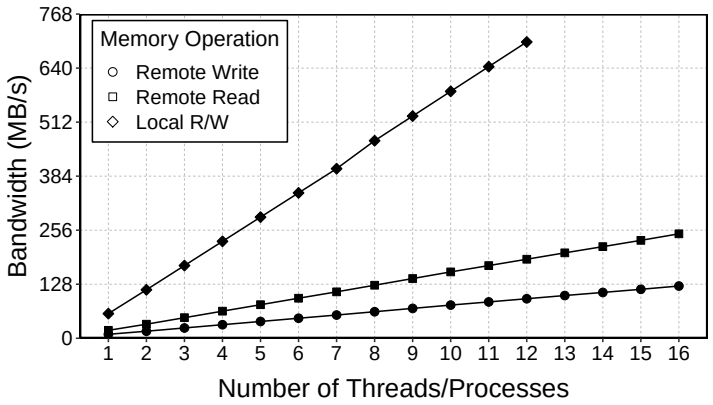


Figure 39 – Uncached read/write bandwidth.

Bottom line, these findings support the design of our memory management subsystem. It relies on a page cache to cope with the performance gap for accessing local and remote memories.

Cached Memory Access

Figure 40 presents the bandwidth for cached memory accesses in our memory management subsystem. These results were obtained with the `pgfetch` program, when varying both the number of concurrent processes and the working set size. Besides, note that we highlight the bandwidth achieved in uncached writes which we observed with the `rstream` program.

When analyzing the scalability of cached memory accesses, we observed that bandwidth increases with the number of concurrent processes, specially when up to 4 concurrent processes are running. This behavior is related to the architectural characteristics of Kalray MPPA-256: processes run in Compute Clusters, which are disposed in a 4×4 2-D torus NoC. Since in this experiment we have one process running per Compute Cluster, running more than 4 concurrent processes with this configuration implies on NoC saturation. Furthermore, we observed that the memory system cache may significantly improve the memory bandwidth in a scenario with working sets of 32 pages, achieving up to 1751.73 MB/s (16 processes) in contrast to 135.93 MB/s with uncached accesses.

However, we noted that if the working set is larger than 32 pages, performance degradation follows. The reason for this is two-fold. First, we configured the size of our memory system cache to 32 pages, thus any workload bigger than this implies capacity misses. Second, due to hardware limitations in the MMU of the Kalray MPPA-256 processor, whenever a page fault occurs, the VMem Cache cannot determine whether or not a page was modified. Consequently, the VMem Cache must perform a write-back operation to ensure data consistency. Therefore, each page fetch incurs in an additional copying that does not happen when the cache is bypassed. Overall, the result of this experiment yields to two important conclusions:

(i) Thanks to the page cache, an application running on top of our memory system may have similar performance as if it would manage by itself data chunking and offloading.

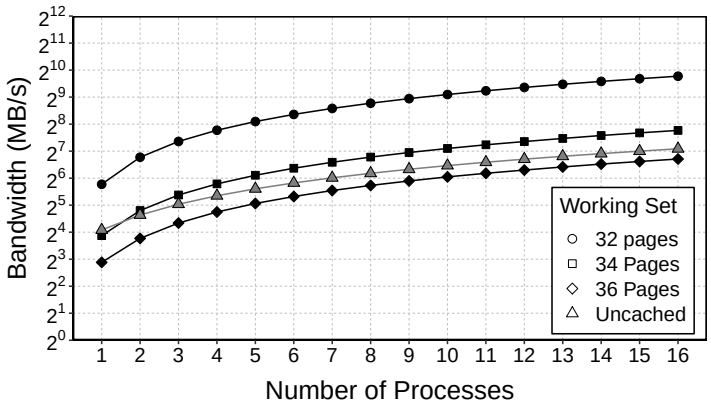


Figure 40 – Cached read/write bandwidth.

(ii) The performance of the page cache, may be significantly improved if the underlying MMU hardware provides enough information to distinguish clean pages from dirty ones.

Memory Management Protocols

Table 9 presents the latency and energy consumption for the page fetch and invalidation. These results were obtained with the `pgfetch` and `pginval` programs, respectively. The first protocol is triggered whenever a fault occurs in the page cache of a process and a page must be fetched from a remote memory server to the local memory of a cluster. On the other hand, the second protocol is fired when a shared page is written back to the remote memory and cached copies of that page should be invalidated. In summary, we present this results to uncover the overheads of our DPS and thus to guide future works on this.

Overall, we noted that both protocols have important time requirements. This corroborates to our claim that the page cache should be efficiently managed in order to deliver reasonable performance to an application relying on either features: virtual memory and/or shared-memory programming. Nevertheless, in Section 6.3 we show that the page cache of our DPS is enough to deliver reasonable performance when applications use virtual memory features provided Nanvix. Furthermore in (JUNIOR et al., 2021) we evaluate how different page replacement strategies may positively impact on the overall performance of our DPS. As a final remark, it is worthy noting that the page fetch latency that we measured is aligned to what may be currently observed in current multiprocessor computers. Putting it differently, our memory management subsystem enables LW Manycore processors to achieve comparable performance to commodity processors, in respect to page fault latencies.

Table 9 – Performance of memory management protocols.

Protocol	Total Energy	Total Latency	Kernel Time	Processing Time	Network Time
Page Invalidation	9.55 mJ	1370 µs	30.75 µs	1248.25 µs	91 µs
Page Fetch	31.4 mJ	4700 µs	30.75 µs	4544.15 µs	125 µs

6.3 Process System

In this section, we present our evaluation results on the process management subsystem of Nanvix. First, we discuss how it improves programmability and portability in LW Manycore processors (Section 6.3.1). Then we analyze the performance and scalability of various tasks and protocols that are carried out by this subsystem (Section 6.3.2).

6.3.1 Programmability and Portability

The process subsystem of Nanvix improves programmability and portability in LW Manycore processors in two axes. First, it provides the process abstraction, which encapsulates

the execution flow of a program, as well as the set of software and hardware resources that are currently assigned to that program. Thanks to the process abstraction, programs may be isolated from each other, and thus multiple applications may be concurrently deployed in a LW Manycore processor.

Second, the process management subsystem bridges geometry and topology differences across LW Manycore processors. On the one hand, in respect to the former property, the microkernel running on each cluster of the processor enables multithreading through preemptive threads. With this solution, a process may spawn as much threads as it needs, and the software engineer does not need to worry about multiplexing these threads of top of the underlying cores.

On the other hand, concerning the processor topology, the Nanvix provides a uniform numbering scheme for addressing clusters and processes. Thanks to the Name Service, each process is assigned to a unique identifier and a name. In turn, processes may use these to communicate with one another, without knowing the specific numbering scheme of the processor. Furthermore, process identifiers and names enable transparent on-chip communication: processes are not required to know the allocation of one another (i.e., in which cluster they are running). In this way, process migration may be also enabled, without impacting communication.

6.3.2 Performance

In this section, we present the performance results for the process subsystem. First, we uncover the latencies involved in kernel calls. Then, we examine the overheads for creating and terminating threads. Finally, we unveil the performance of some protocols of this subsystem.

Kernel Calls

Figure 41 shows the breakthrough of execution cycles for local kernel calls in Nanvix. Recall that these calls are those that execute in the same core and cluster in which they were issued. The results that we present were obtained with the `lkcall` program. Important to remark, hardware events that are depicted in this plot are not exclusive with each other. For example, an *I-Cache Stall* may incur in a *Register Stall*. Therefore, the bar labeled *Total Cycles* is not the

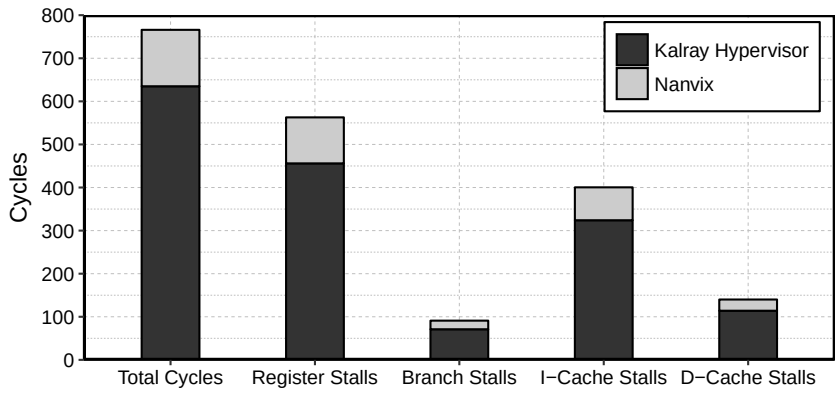


Figure 41 – Execution breakthrough for local kernel calls.

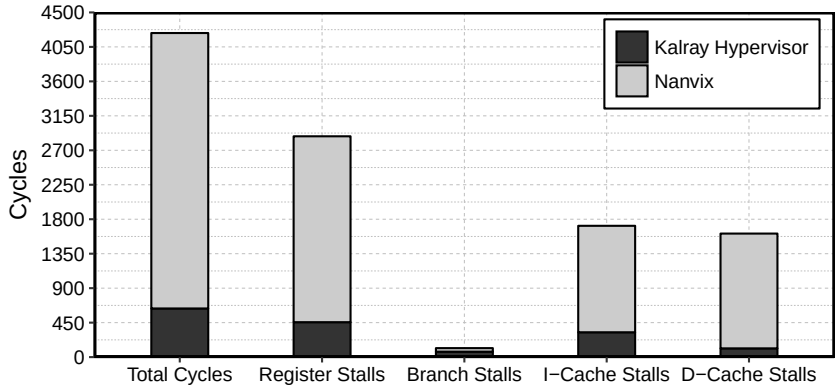


Figure 42 – Execution breakthrough for remote kernel calls.

aggregation of the other bars. Furthermore, note that we point out the amount of cycles that were spent in local kernel calls while Nanvix was running in hypervisor mode.

In general, we observed that this hypervisor accounts for an important amount of execution statistics, thereby uncovering a potential hot spot for improvement. Besides this, we noted that the cost for local kernel calls is about 766 cycles, from which 164 cycles are required for saving and restoring machine registers upon mode switches. Execution stalls caused by the branch unit and D-Cache account for 11.88% and 18.28% of total execution cycles, respectively. Together, these metrics suggest that the overhead of local kernel calls is low. The complex execution flow does not cause the branch unit to perform badly, nor the implied mode switch introduces enough stalls in the D-Cache so as to turn this into the the execution bottleneck. On the other hand, stalls caused by the register file and the I-Cache account for 73.50% and 52.22% of execution cycles, respectively. This behavior unveils that local kernel calls impose substantial pressure in the I-Cache. We analyzed the code of the `lkcall` program and we found out that it is small enough to fit in the I-Cache of the underlying core. Therefore, we conclude that the poor performance behavior is either because: (i) the I-Cache has a small associativity (2-way); or (ii) the prefetching buffer unit of the I-Cache is badly performing.

Figure 42 presents the breakthrough of execution cycles for remote kernel calls in Nanvix. These results were obtained with the `rkcall` program. Note that the bar labeled *Total Cycles* is not the aggregation of the other bars. Likewise in local kernel calls, we observed that stalls on register file accounted for an important amount of total cycles (68.4%). However, in this context, we found out that this behavior was caused by stalls on both I-Cache and D-Cache, which added up to 40.51% and 38.12% of execution cycles, respectively. In order to understand the aforementioned behavior for the I-Cache, we carried out the same code analysis that we did when analyzing results for local kernel calls, and we reached out the same conclusions.

However, for stalls caused by the D-Cache we found out a different reason. By design, a remote kernel call requires inter-core communication. Therefore, from the perspective of the D-Cache, coherence traffic arises. In this scenario, the performance issue emerges due to the fact that hardware cache coherence is not supported in Kalray MPPA-256. To deal with this

problem, the processor exposes instructions to flush, discard and wait for changes to the D-Cache. Unfortunately, this is not enough to implement an optimal coherency protocol in kernel space, due to a lack of context information. Hence, to ensure correct semantics, the kernel issues a cache shoot-down, just before starting and returning from kernel calls, thereby dropping performance.

Fork Join

Figure 43 presents scalability results for creating and terminating threads. These results were obtained with the `forkjoin` program, and the plot depicts the the costs for creating and terminating threads, in terms of cycles. Overall, our results unveiled optimal performance for creating and terminating threads: when increasing the number of threads that we spawn and join, overheads grow linearly. On average, for spawning and terminating a thread, we observed an overhead of 5.14 k cycles (12.86 µs) and 3.42 k cycles (8.57 µs), respectively.

With these results, we also uncovered three observations that are worthy to point out. First, the latency for spawning and terminating a thread, which is 21.43 µs, gives insights on the spawn/join cost for supporting frameworks based on the thread pop-up model, such as web servers. Second, we observed a performance gap of about 1.5× between the two operations. We found out that the reason for this lies on the synchronous acquisition of resources during thread creation, in contrast to the asynchronous release of resources in termination. To decrease this performance gap, in future works, we intend to introduce thread recycling in Nanvix. Finally, the linear spawn/join scalability further shows that our asymmetric design is scalable, since both operations are implemented as remote kernel calls.

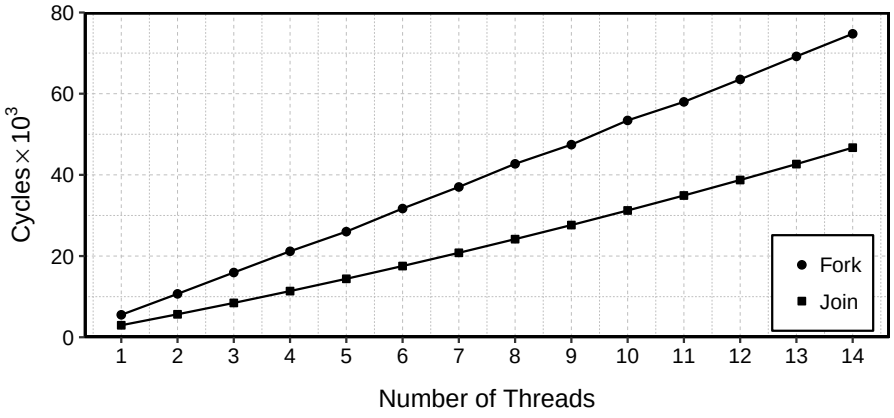


Figure 43 – Performance scalability for creating and terminating threads.

Performance Isolation

Figure 44 shows results for performance isolation of applications in Nanvix. These statistics were obtained with the `knoise` program, and in the plot we present the efficiency that is sustained when increasing the number of compute-intensive threads in a process when the kernel is subjected to heavy load (*with noise*) and it is not (*without noise*). Putting it differently,

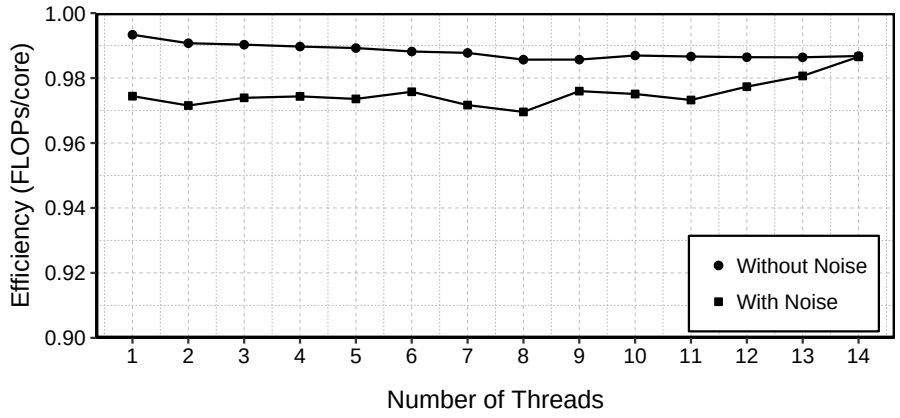


Figure 44 – Execution efficiency for the `knoise` Benchmark.

This experiment provides the worst-case evaluation for the execution interference of our OS in an application: when threads run the compute-intensive workload, all the remainder threads execute the kernel-intensive one.

Overall, our results uncover the strongest advantage of the asymmetric kernel that Nanvix features: low interference in the execution of user-level threads. When a single thread runs a compute-intensive workload, and the remainder 13 threads a kernel-intensive one, the efficiency achieved is 0.9% below from the one that we could achieve with no kernel-intensive tasks. Furthermore, we observed that when the number of compute-intensive threads increases, the performance interference gradually drops down to zero.

Process Management Protocols

Table 10 presents the latency and energy consumption for the heartbeat and name lookup protocols, which were obtained with the `lookup` and `heartbeat` programs, respectively. The process management server runs the former protocol periodically to know whether or not a given process is healthfully running. In contrast, the latter protocol is explicitly called by a process whenever it wants to resolve a process identifier or name into a physical location, for instance to communicate with that process. In terms of the characteristics of each protocol, both rely on *mailboxes* to pass information around and are sent by processes to the server. However, heartbeats involve minimum processing, whereas name lookup messages require some server-side processing for name resolution.

Overall, we noted that most of the time is spent on processing time outside the kernel, thereby unveiling the potential spot for improvement. For the heartbeat and name lookup

Table 10 – Performance of process management protocols.

Protocol	**Total Energy**	**Total Latency**	**Kernel Time**	**Processing Time**	**Network Time**
Heartbeat	881 µJ	130 µs	12 µs	87 µs	31 µs
Name Lookup	4.5 µJ	737 µs	20.5 µs	656.5 µs	60 µs

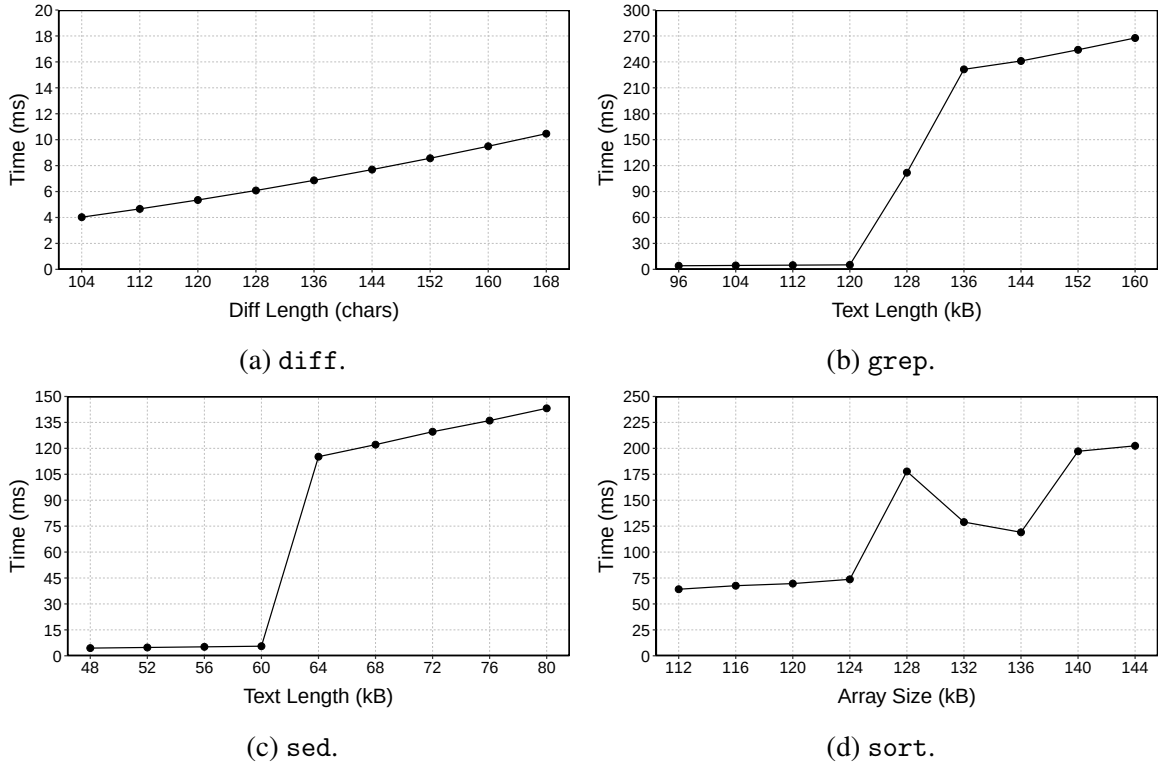
protocols, this statistic accounts for 66.92% and 89.07%, respectively. These results may guide the parameterization of Nanvix for meeting specific performance criteria and energy budgets. On the one hand, the frequency of heartbeats may be fine-tuned to decrease passive energy consumption. On the other hand, the length of the name lookup cache may be sized accordingly to reduce the number of name resolution misses.

6.4 User Applications

So far we analyzed the isolated performance of different subsystems of Nanvix. In this section we urn our discussion to a broader evaluation that encompasses the interaction among subsystems, as well as user and system libraries. First, we uncover the performance results for system utilities (Section 6.4.1), and then we unveil performance of scientific programs (Section 6.4.2). Important to note, by discussing these experimental results we are effectively showing that Nanvix delivers programmability and portability to LW Manycore.

6.4.1 System Utilities

Figure 65 presents the execution time of the four Unix utilities considered, when varying the size of the input working set. Noteworthy, each utility was exclusively executed in Nanvix. In other words, aside from OS services, there was no other user-level application running. Therefore, there are no external interference, and thus reported results reflect the upper-bound performance.



(a) `diff`. (b) `grep`.

(c) `sed`. (d) `sort`.

Figure 45 – Performance for system utilities.

When observing results for the `diff` utility (Figure 65a), we noted a linear increase in execution time when increasing the size of the working set. The reason for this behavior is explained by the characteristics of the application. The least common subsequence algorithm processes the memoization table onward and each new access to this data structure incurs in a page fault. Consequently, the bigger the data set, the higher is the number of page faults, resulting in longer execution times. Since the time for fetching pages is constant, linear execution time follows.

Figure 65b and Figure 65c present the execution times for the `grep` and `sed` programs, respectively. In both cases, we observed a stepped performance behavior that leads to complementary conclusions for `diff`. These two utilities (i.e., grep and `sed`) feature access patterns that greatly benefit from the spatial locality. For this reason, when the working set is small enough to fit in the page cache of our memory system, we observed small execution times. However, as soon as the working set does not entirely fit in the cache, we observed that execution time increases significantly. Noteworthy, when this latter situation happens, increases in execution time are linear, as we observed for `diff`. Finally, it is important to highlight that the results of these experiments show how important the page cache is our system. Thanks to it, applications may achieve up to 10$\times$ better performance than directly accessing remote memories.

Finally, we present in Figure 65d results obtained with the `sort` utility. In this experiment, we noticed a more irregular behavior with three main phases. First, when the size of the working set is small enough to fit in the page cache (i.e., up to 128 kB), execution time increases slightly. Then, when the size of the working set varies from 100% to about 105% of the size of the page cache, performance presents a great variation. Third, when the working set is large enough, we observe linear increases in execution time from a higher baseline. The results for the first and third phases are aligned to the behaviors noted in the other utilities. In contrast, the reasoning for the behavior observed in the second phase comes from the random access pattern of the program itself. In this scenario, the page replacement policy performs at most as bad as a pathological pattern (peak in 128 kB) or as good as when locality is exploited (hollow in 136 kB).

Figure 66 pictures the performance of Nanvix when it is subjected to heavy load. In

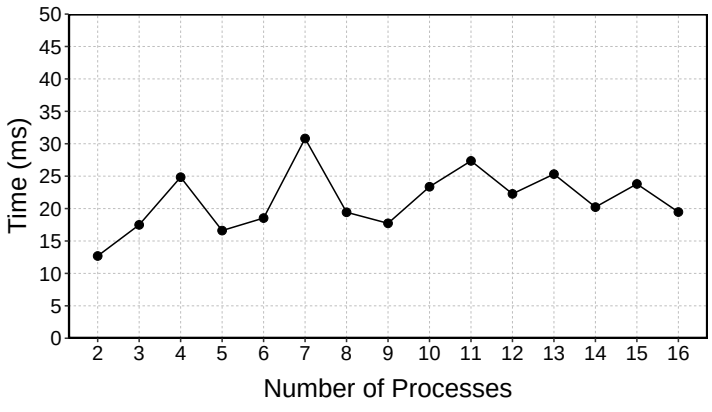


Figure 46 – Performance of Nanvix under heavy-load.

this experiment, we deployed multiple instances of the four Unix utilities at the same time. System images were randomly generated, varying the number of process in the system.. More importantly, this experiment shows that Nanvix enables resource multiplexing, isolation and sharing in LW Manycore processors.

Overall, we noted that execution time presents some irregularity, varying from 12 ms to up to 30 ms. However, this behavior was expected since system images were randomly generated and the characteristics of each utility vary from one to another. Nevertheless, the execution time does not significantly increases when moving from a lightweight scenario, in which a single utility is deployed, to a heavyweight one, in which sixteen utilities are concurrently running in Nanvix. The performance decreases sub-linearly with the load in the system: the overall performance degrades by only 38% when increasing system load by a factor of 16×.

6.4.2 *Scientific Programs*

Figure 47 presents results for `fn`, `gf`, and `km` programs based on two metrics: execution times and weak scaling efficiency. In these plots, `IPC` corresponds to an implementation of each program using Nanvix IPC; and lwMPI. In the following paragraphs, we highlight our main findings.

In `fn`, a *global master* MPI process distributes equal-sized ranges of numbers to *local master* MPI processes, which in turn divide these ranges equally among its associated *slaves* to compute the abundance values. `fn` is a CPU-bound application and has a low communication demand. Because of that, results obtained with `IPC`, and lwMPI solutions are fairly similar. This result is expected since most of the differences between these solutions come from the way they manage communications.

In `gf`, the *global master* MPI process distributes images to be processed by *local master* MPI processes. Then, each *local master* splits the image into equal-sized chunks and distributes them to its associated *slaves* to perform matrix computations using a Gaussian mask. The time spent in communications is not negligible in `gf`. We observed that lwMPI achieved better execution times. The results show that the performance gains achieved by lwMPI tend to decrease as we increase the number of MPI processes. We believe that this performance degradation observed with lwMPI is related to the communications between the *global master* and the *local masters*. This completely synchronous communication tends to hide the benefits of the optimized

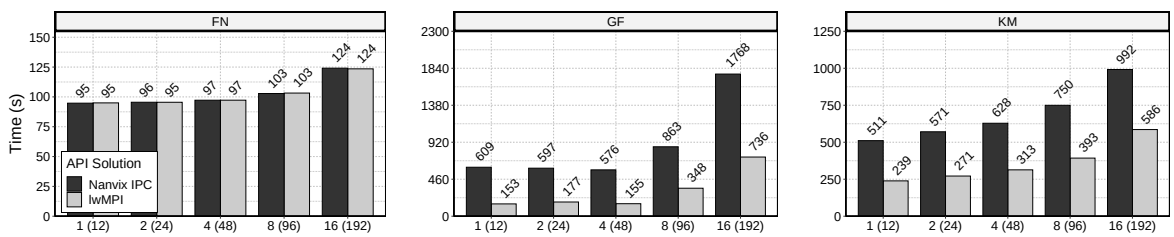


Figure 47 – Execution times for `fn`, `gf` and `km`.

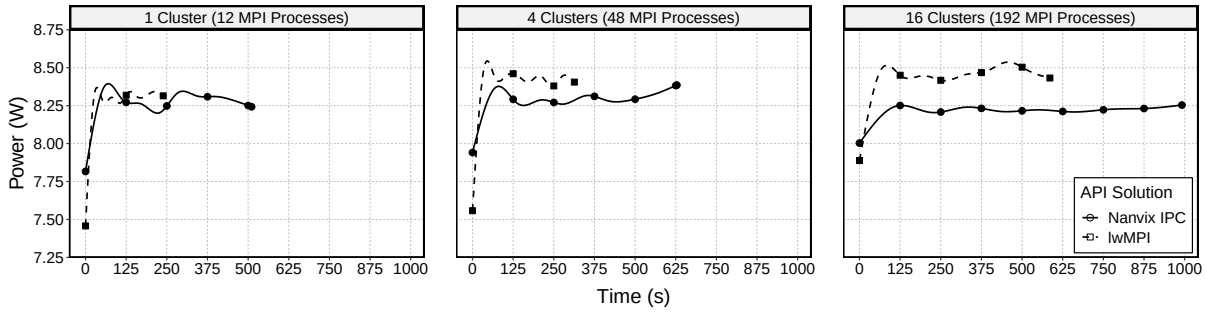


Figure 48 – Power consumption for `km` when varying the number of clusters/problem sizes.

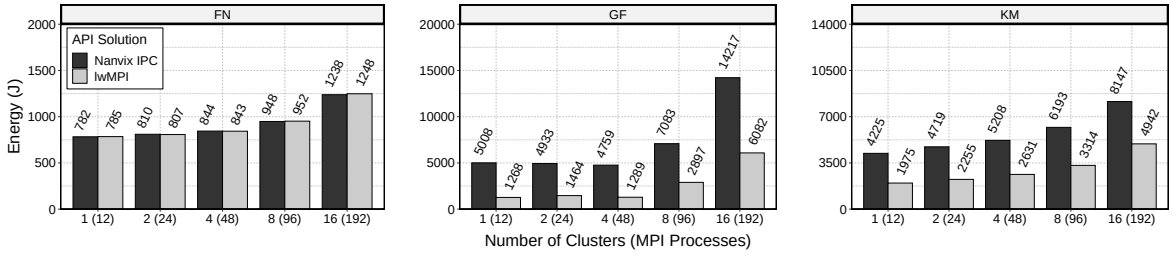


Figure 49 – Energy consumption for `fn`, `gf` and `km`.

local communications in Compute Clusters, resulting in *local masters* waiting for their turn to communicate with the *global master*.

In `km`, the *global master* MPI process iteratively orchestrates the parallel execution by gathering and broadcasting centroids to *local master* MPI processes, which then forward the data to *slaves* to perform the actual computation. Similarly, lwMPI achieved the best execution times for `km`. We observed a fairly consistent growth in execution times of all solutions as we increased the number of MPI processes. Since `km` is a communication-bound application, it is ideal for evaluating the performance gains that can be achieved with the local communication optimization implemented in lwMPI. Overall, the lowest performance improvement achieved by lwMPI was with 192 MPI processes.

Figure 48 shows the power consumption when running `km` with 12, 48, and 192 MPI processes. As it can be noticed, the power consumption of Kalray MPPA-256 when running `km` with lwMPI is slightly higher than with `IPC`. This increase in power consumption is due to the optimizations in local memory communications, which allow on-chip resources to be better exploited. A similar behavior was also observed with `gf`. Since execution times of `gf` and `km` are drastically decreased with lwMPI, their overall energy consumption is also reduced as shown in Figure 49. As expected, the energy consumption of `fn` was the same for all solutions because it has very few communications. Overall, execution times and energy consumption follow the same trend on all applications considered.

6.5 Concluding Remarks

In this chapter, we presented and discussed the experimental results that we obtained for Nanvix. In the next sections we summarize the most important findings that we reached out in our analysis, as we review answers for the evaluation questions that we formulated in Section 5.1.

6.5.1 Programmability and Portability with Nanvix

Concerning programmability (Q-1) and portability (Q-2), we carried out a detailed analysis as follows. In Section 6.1.1, we examined how the IKC primitives of Nanvix enable more efficient and transparent on-chip communication. To this end, each primitive was designed to work with a specific communication granularity and handle flow control. Furthermore, our OS deals with low-level allocation of hardware resources as well as multiplexing of NoC interfaces. In Section 6.2.1, we uncovered how the DPS of Nanvix enables address space expansion and memory sharing over a distributed memory architecture with small local memories. We leverage the capabilities of this system by exposing a POSIX interface on top of it. Finally, in Section 6.3.1, we discuss how that the process abstraction of Nanvix enables multiples applications to be simultaneously deployed in a LW Manycore processor. Also we argue on how the Name Service enables transparent communication and process identification, regardless the processor topology.

Finally, as an important remark on software portability we highlight two points that are implicitly present in the way that Nanvix is engineered and the results that we discussed. On the one hand, the HAL of Nanvix enables system-level portability to multiple targets (i.e., k1b, x86 OpenRISC ARMv8 and RISC-V), which in turn not only differ in Instruction Set Architecture (ISA)-level but also in computing capabilities. In Section 7.1 we briefly present some works that evaluate Nanvix on other platforms. On the other hand, at application-level, Nanvix enables the portability of applications by supporting standard programming interfaces and runtime libraries, such as POSIX, MPI and OpenMP. This outcome is further sustained by the fact that in Section 6.4 we present and discuss experiment results for system-level utilities and scientific programs that were ported to Nanvix on top of the libraries and runtime systems that are supported by our OS.

6.5.2 Performance in Nanvix

Regarding the performance assessment of Nanvix (Q-3) we highlight that extensive analysis of our experimental results was presented. In Section 6.1.2, we analyzed the latency, throughput and scalability of our IKC primitives. Overall, we unveiled that: (i) 128-B *mailboxes* sustain about 3.8 k message throughput. (ii) 4 kB *portals* enable 7.5× superior performance than a baseline communication primitive. (iii) *syncs* efficiently exploit fine-grain on-chip communication and deliver a solution that is 64× fast than the baseline.

In Section 6.2.2, we discussed about the performance of the Memory Management System of Nanvix. For uncached memory accesses, we uncovered that single-stream local memory bandwidth is about 9.38 MB/s and 18.76 MB/s, when performing reads and writes, respectively. In contrast, for local memory access, the bandwidth achieved by a single thread is 58.30 MB/s, for both reads and writes. Furthermore we verified that local memory accesses are up to 10× than remote reads and remote writes, respectively, thereby confirming the design of our memory management system which: (i) efficiently manages local memory; (ii) and extensively exploits data locality. For cached memory accesses, up to 1751.73 MB/s throughput may be achieved, in contrast to 135.93 MB/s with uncached accesses. This outcome hols when considering a working set that fits in the cache of the DPS, and when 16 processes are running concurrently. Furthermore, we noted that if the working set is larger than the dimensions of the page cache, performance drops below the baseline (uncached accesses) due to the non-optimal decisions made by the page replacement algorithm. Importantly to note, we found out that this is due to a hardware limitation on the underlying platform, which if it is overcome, superior performance may be achieved. We further discuss results that we obtained on this in (JUNIOR et al., 2021). Finally, when analyzing latencies of important protocols of the memory management system we found out important time requirements, thereby stressing our claim that the page cache of the DPS should be efficiently managed to achieve high-performance. For page invalidations and page fetches, about 1370 µs and 4700 µs are required, respectively.

In Section 6.3.2, we untangle results of the Process Management System of Nanvix. Overall we reached important conclusions. First, concerning the performance of kernel calls, we confirmed that local ones are substantially faster (5×) than remote ones. Furthermore, we observed that performance of both can be significantly improved if limitations related to the underlying platform are overcome – hypervisor and coherence traffic. In (PENNA et al., 2020) we further discuss the unfolding of these limitations, once they are surpassed. Second, we noted that the time for creating and terminating threads is constant and thus linearly grows with the number of threads that are currently running in a process. Putting it differently, Nanvix may efficiently support libraries that rely in thread-pup or fork-join models. Third, we observed that the asymmetric microkernel design that Nanvix features leads to low interference in the execution of user-level threads. Even when considering a stressing scenario, 99.1% efficiency may be achieved. Finally, when analyzing the performance of protocols of the Process Management System, we observed that they spent most of their time is spent in processing time outside the kernel. These results may guide the parameterization of Nanvix for meeting specific performance criteria and energy budgets

In Section 6.4, we unveil the performance results for user-level applications running on top of Nanvix. First, when analyzing the execution results of individual system-level utilities, we noted that the DPS solution of Nanvix enable these programs to transparently manipulate data sets larger than local memories of the underlying platform, while delivering up to 10× better performance than directly accessing remote memories. Second, when we examined performance

results of Nanvix when it is subjected to heavy load, we found out that this metric does not significantly decreases when moving from a lightweight scenario to a heavyweight one. The performance decreases sub-linearly with the load in the system, thereby showing the capabilities of Nanvix to grant performance isolation, while providing resource multiplexing and sharing. Finally, when the execution of scientific programs, which make us of lwMPI and span over multiple clusters of the underlying LW Manycore process, we reached two important conclusions. On the one hand, we observed that performance scalability may be delivered to these applications, and thus show that Nanvix may also be used to speedup the development of HPC applications in LW Manycore. On the other hand, we verified that performance may be leveraged due to the fact that specific improvements may be applied in user-level, which cannot be done at system-level, such as local communication with zero-copying.

7 CONCLUSIONS

LW Manycore were introduced to deliver performance scalability with low-power consumption. To address the former aspect, these processors rely on specific architectural characteristics, such as a distributed memory architecture and a rich NoC. To achieve low-power consumption, these manycores are built with simple low-power MIMD cores, they have a memory system based on SPM and they exploit heterogeneity by featuring cores with different capabilities. Some industry-successful examples of these processors are the Kalray MPPA-256, the PULP and the Sunway SW26010.

While this unique set of architectural features grant to LW Manycore performance scalability and energy efficiency, they also introduce multiple challenges in software development and deployment. First, the *high density circuit integration* turns dark silicon into reality. Second, the *distributed memory architecture* requires data to be explicitly fetched/offloaded from remote memories to local ones. Third, the *small amount of on-chip memory* forces the software to partition its working data set into chunks and decide which of them should be kept local and which should be offloaded to remote memory. Fourth, the *on-chip interconnect* invites software engineers to embrace a message-passing programming model. Finally, the *on-chip heterogeneity* makes the deployment of applications complex.

To tackle the aforementioned problems, there are two main approaches: runtime systems and OSes. The first aims to expose a programming environment to address the needs of a specific class of applications (HO et al., 2015; Mohr; Tradowsky, 2017). In contrast, the second approach craves to bridge intricacies of the hardware in a broader sense, by exposing rich abstractions and programming interfaces, as well as handling resource allocation, sharing and multiplexing.

In the context of LW Manycore processors, unfortunately today, neither of the two previous approaches provide a complete solution to tackle programmability and portability challenges. On the one hand, runtime systems inherently lack on covering all the issues that we stated in previously. For instance, some runtime systems do provide rich abstractions tackle on-chip communication, but overlook challenges that concern the distributed memory architecture. On the other hand, existing OSes are not designed to cope with all the architectural constraints of LW Manycore processors. In summary, it is not possible to deploy these OSes in LW Manycore without a complete redesign and/or significant changes to their source code. Overall, we identified the following unanswered problem in the the state-of-the-art Operating Systems, which we aim to address in this thesis:

How to design an operating system so as to address programmability and portability issues in lightweight manycore processors?

7.1 Contributions

This thesis proposes a novel distributed OS that advances the resource management in LW Manycore processors and specifically stresses the following points:

- *Programmability*: to provide rich system abstractions for LW Manycore, such as process, files, virtual memory and communication primitives; as well as to enable resource sharing, multiplexing and isolation in these emerging architectures.
- *Portability*: to expose standard programming interfaces so as to enable existing software to be ported to LW Manycore processors.

In the following sections we recall scientific and technical contributions that unfold from the these points. Additionally, we also highlight the academic contribution of this thesis on the supervision of students and interns.

7.1.1 Scientific Contributions

The main contribution of this thesis – *a novel distributed OS that advances the resource management in LW Manycore processors* – may be unfolded in the following specific scientific contributions:

- A comprehensive HAL for LW Manycore that bridges architectural differences across multiple architectures, and copes with key issues that are often encountered when designing an OSes for these processors. With this abstraction layer, the development and deployment of a fully-featured OS becomes easier not only to a particular LW Manycore, but also enables the portability of a specific OS across multiple of these emerging processors.
- A rich memory management approach that is based DPS. This is a novel solution that we devised for managing memory of a LW Manycore processor and it works as follows. The local memories of a LW Manycore processor are considered page caches that can store data of whichever processes. Overall, this solution enables transparent data accessing and manipulation, as well as data sharing and coherence in LW Manycore;
- A lightweight communication facility that manages the on-chip interconnect and exposes primitives with channel multiplexing.

Important to note, these contributions were published in multiple scientific communication medium, during the preparation of this thesis. Bellow, we briefly highlight these works.

- In (PENNA et al., 2019) we focus on the open challenges that arise from constrained memory subsystems of LW Manycore, such as the presence of multiple address spaces and limited on-chip memory. To cope with transparent data access in this scenario, we introduce an OS service, named RMem. This service provides a shared memory abstraction over multiple

address spaces and exposes system calls that enable one-sided communication on top of this abstraction. We implemented a prototype of our service in the Nanvix research OS, and we deployed the system the Kalray MPPA-256 LW Manycore. Our experimental results with a microbenchmark unveiled that, while exposing an easier-to-program interface, the RMem Service may deliver about 91% of the write performance and up to 2.4× better read performance than the primitives in the libraries of the experimental platform.

- In (PENNA; FRANCIS; SOUTO, 2019) we introduce a Hardware Abstraction Layer (HAL) for LW Manycore that cope with key issues that are often encountered when designing an OS for these processors. We present the interface exposed by our HAL, as well as a discussion about its implementation for the Kalray MPPA-256 manycore.

- In (PENNA et al., 2019) we argue distributed OS architecture based on asymmetric microkernels for LW Manycore manycores. We deliver an open-source implementation of an OS kernel with these characteristics, and we provide a comprehensive assessment using a representative benchmark suite. Our results show that an asymmetric microkernel design is scalable and introduces at most 0.9% of performance interference in an application execution. Also, our results unveil co-design aspects between an OS kernel and the architecture of LW Manycore, concerning the memory system and core grouping.

- In (ULLER et al., 2020b) we propose a portable and lightweight MPI library (lwMPI) designed from scratch to cope with restrictions and intricacies of LW Manycore. We integrated lwMPI into a distributed OS that targets these processors and evaluated it on the Kalray MPPA-256 processor. Results obtained with three applications from a representative benchmark suite unveiled that lwMPI achieves similar performance scalability in comparison with the low-level vendor-specific API narrowed for MPPA-256, while exposing a richer programming interface.

- In (JUNIOR et al., 2021) we propose a trace-driven methodology to evaluate and optimize features of a memory management service of distributed OSes for LW Manycore. By using a compact representation of the page access pattern of the applications, our methodology is capable of mimicking the memory access pattern of the original applications on the target distributed OS running on a LW Manycore. We integrated our methodology in a distributed OS (Nanvix) and validated it using three applications from a specific benchmark for LW Manycore (CAP Bench). Then, we applied our methodology to carry out a case study using a software-managed cache implementation available in Nanvix. Our methodology enabled us to evaluate different page replacement policies on Kalray MPPA-256, even without the required support from the architecture to implement them.

- In (PENNA et al., 2020) we propose a co-design solution between the cluster of a LW Manycore and the OS kernel to cope with the overhead of the asymmetric microkernel. We designed a 4-core heterogeneous cluster with one core tuned for the OS kernel and we

patched the OS kernel to better match the characteristics of this core. Our experiments unveiled that our solution consumes 14.1% less power than the baseline and also improves the OS kernel performance by up to 6.5%.

- In (PENNA et al., 2021) we propose an Inter-Kernel Communication (IKC) facility that exposes two kernel-level communication abstractions in addition to mailboxes: syncs, for enabling a process to signal and unlock another process remotely, and portals, for handling dense data transfers with high bandwidth. We implemented the proposed facility in Nanvix, the only open-source distributed OS that runs on a silicon LW Manycore, and we evaluated our solution on a 288-core processor (Kalray MPPA-256). Our results showed that our IKC facility achieves up to 16.87× and 1.68× better performance than a mailbox-only solution, in synchronization and dense data transfers, respectively.

7.1.2 Technical Production

From the technical perspective, this thesis also introduces an important contribution, which is Nanvix. This is a concrete implementation of an OS for LW Manycore processor that features the aforementioned scientific advancements. Nanvix supports multiple architectures (i.e., x86, OpenRISC, ARMv8 and RISC-V), runs on silicon LW Manycore processors, exposes rich abstractions and high-level programming interfaces.

Nanvix features a multikernel design that is structured on top multiple instances of a microkernel. We chose this organization because we believe that it best meets the characteristics of LW Manycore. On the one hand, the multikernel approach copes with the distributed memory architecture of these processors. On the other hand, the microkernel-based structure enables a lightweight and modular implementation of system services, which is a fundamental requirement given the scarcity of hardware resources. It has a four-tier layout which we introduce next.

The HAL exposes a comprehensive abstraction of the underlying hardware (PENNA; FRANCIS; SOUTO, 2019). It is implemented as a baremetal library, and it ships common routines and data structures to manipulate hardware structures, such as cores, MMU and NoC. In summary, the HAL enables the portability of our OS across multiple LW Manycore processors, such as Kalray MPPA-256, OpTiMSoC and PULP.

The microkernel provides resource sharing, multiplexing, isolation and protection, within a cluster of a LW Manycore (PENNA et al., 2019). It runs in privileged mode and exposes bare minimum OS abstractions and primitives, such as threads, virtual address space and inter-process communication mechanisms. Our microkernel features an asymmetric design, which means that it exclusively runs in one core of the cluster, and it leaves the remaining cores to general-purpose use. This design mitigates the interference of the kernel in user-level software, and thus improves performance.

System servers are deployed on top of the microkernel and implement services commonly found in an OS, such as process scheduling, memory mappings, and file allocation. In turn,

subsystems are implemented by having a set of these system servers that work together, in a distributed fashion. Finally, on top of system servers, the runtime libraries are disposed. These libraries are linked to the user applications and provide a standard interface for user-level software to interact with the OS.

7.1.3 Academic Supervisions

During the preparation of this research, the main author of this work supervised several students and interns in the subject of this thesis:

- Clement Rouquier (2017 – UGA, Grenoble, France) worked during his engineering internship (10 weeks, full-time) in the setup of the automated regression test infrastructure for Nanvix.
- Romane Gallier (2017 – UGA, Grenoble, France) worked during her engineering internship (10 weeks, full-time) on the implementation of the virtual file system of Nanvix.
- Vincent Turrin (2017 – UGA, Grenoble, France) worked during his engineering internship (10 weeks, full-time) on the implementation of the virtual file system of Nanvix.
- Davidson Francis Lima (2017-2019 – PUC Minas, Belo Horizonte, Brazil) worked during his BSc (partial-time) in the port of the HAL of Nanvix for the OpenRISC architecture and OpTiMSoC platform.
- Guillaume Besnard (2018 – UGA, Grenoble, France) worked during his engineering internship (10 weeks, full-time) in the implementation of the PThreads library for Nanvix.
- Jordan Jean (2018 – UGA, Grenoble, France) worked during his engineering internship (10 weeks, full-time) on the implementation IPC services in Nanvix.
- Antoine Saget (2019 – UGA, Grenoble, France) worked during his engineering internship (10 weeks, full-time) in the port of a lightweight TCP/IP stack (LwIP) to Nanvix.
- Daniel Coscia (2019 – UGA, Grenoble, France) worked during his engineering internship (10 weeks, full-time) in the initial development of the built-in Unix Simulator of Nanvix.
- Emmanuel Podestá Júnior (2019 – UFSC, Florianópolis, Brazil) worked during his MSc internship (10 weeks, full-time) on the investigation of page replacement strategies of the DPS in Nanvix.
- Lucas Andrade Maciel (2019 – PUC Minas, Belo Horizonte, Brazil) worked during his MSc internship (10 weeks, full-time) in hardware/software co-design aspects between the Nanvix microkernel and the cluster of a LW Manycore processor.
- João Vicente Souto (2018-2021 – UFSC, Florianópolis, Brazil) worked during his BSc and MSc dissertation (partial-time) on the low-level implementation of the HAL of Nanvix

for Kalray MPPA-256. Also, he made important contributions to the implementation of distributed services of our OS.

- João Fellipe Uller (2019-2021, UFSC, Florianópolis, Brazil) worked during his BSc dissertation (partial-time) on the design and implementation of the lwMPI. Furthermore, he has greatly contributed to the development of the Nanvix microkernel.

- Lucca Augusto Santos (2020-2021 – PUC Minas, Belo Horizonte, Brazil) worked during his BSc dissertation (partial-time) on the implementation of the file system service for Nanvix.

- Alexis Lanquetin (2021 – UGA, Grenoble, France) worked during his engineering internship (10 weeks, full-time) in the port of the HAL of Nanvix for the ARMv8 architecture.

In addition, it is important to stress that the research pushed by this thesis leveraged academic and scientific collaboration between PUC Minas, UGA and UFSC.

7.2 Future Works

This thesis enables the research and investigation in several aspects that concern OS support for LW Manycore processors. Bellow we give some insights on those research topics that we find more relevant.

Kernel Bypassing By design in Nanvix, processes run exclusively in a cluster. While this can make the overall system structure simpler, we believe that this characteristic could be leveraged to deliver even better performance to applications. For instance, NoC channels within a cluster are not multiplexed between several processes. As a consequence, the kernel does not to buffer data nor intermediate information flow int he I/O data path. Instead, it could hand direct buffering control to user-applications and thus enable zero-copying transfers. In this context, all that the kernel would do was to validate flows on the control path. The idea of kernel bypass is currently employed in OSes for data centers (ZHANG et al., 2019), to achieve high-performance communication I/O. We believe that this principle may also be studied in the context of LW Manycore, and potentially yield to outstanding performance improvements.

Process Migration Nanvix does not multiprogram clusters nor time-share them between several processes. Therefore, once a process is deployed in a cluster, the underlying cluster will be booked until the process terminates or gets killed. As a consequence, if the process is not make full utilization of all cores in the cluster, or it spends most of its time blocked, waiting for I/O operations, then the processor will be underutilized and the overall system performance will decrease. Indeed, this problem would not arise if the aforementioned limitations existed in the first place. While multiprogramming support is not an option due to the small on-chip local-memories of LW Manycore, the latter alternative (time-sharing)

can be enabled. In a conventional shared-memory architecture, this could be achieved by having processes to be either suspended or migrated to other cores. Unfortunately however, this is not simply possible in LW Manycore, due to the distributed memory architecture that they feature. Putting it differently, to effectively support time-sharing in this scenario, process migration between different address spaces would have to be supported. To achieve this, the OS would have to feature mechanisms to save/restore machine and kernel states remotely. Furthermore and to make this even more challenging, with the limited amount of on-chip-memory this should be enabled with minimum memory footprints. We believe that if process migration is enabled, not only research on load balancing and fault tolerance may be carried out, but also strategies for dynamic scheduling, placement of processes could be enabled in the context of LW Manycore.

Hardware/Software Co-Design Thanks to the fact that Nanvix runs on multiple silicon chip LW Manycore processors, we were able to spot some architectural bottlenecks. It turns out that, if were able to re-design hardware to better address these limitations, then we would potentially push the OS performance and programmability. Indeed, we initially carried out some hardware/software co-design research initiative, where we aimed at designing the cluster of a LW Manycore to better support an asymmetric microkernel design (PENNA et al., 2020). Nevertheless, we believe that in this direction there is still a broad ground to be covered. For instance, the layout and capabilities of an MMU can be rethought to support in a lighter fashion the execution of a single process per cluster and/or offer additional capabilities to implement advanced paging replacement schemes. Furthermore, we believe that interrupt and exception delegation mechanisms exposed by the hardware could be investigated to leverage the design of microkernel-based distributed OS.

REFERENCES

APPAVOO, J. et al. Experience with K42, An Open-Source, Linux-compatible, Scalable Operating-System Kernel. *IBM Systems Journal*, v. 44, n. 2, p. 427–440, 2005.

ASMUSSEN, N. et al. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In: *International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, Georgia: ACM, 2016. (ASPLOS '16), p. 189–203. ISBN 9781450340915. Disponível em: <http://dl.acm.org/citation.cfm?doid=2954680.2872371>.

AUBRY, P. et al. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. *Procedia Computer Science*, v. 18, p. 1624 – 1633, 2013. ISSN 1877-0509. 2013 International Conference on Computational Science. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050913004730>.

AUGONNET, C. et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience (CCPE)*, John Wiley & Sons, Ltd., v. 23, n. 2, p. 187–198, feb 2011. Disponível em: <http://doi.wiley.com/10.1002/cpe.1631>.

BAILEY, D. H. et al. The NAS Parallel Benchmarks: Summary and Preliminary Results. *International Conference on Supercomputing (ICS)*, p. 158–165, 1991. Disponível em: <http://doi.acm.org/10.1145/125826.125925>.

BARBALACE, A. et al. Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms. In: *International European Conference on Computer Systems (EuroSys)*. Bordeaux, France: ACM, 2015. p. 1–16. ISBN 9781450332385.

BARNES, J.; HUT, P. A hierarchical O(N log N) force-calculation algorithm. *Nature*, Nature Publishing Group, v. 324, n. 6096, p. 446–449, 1986. ISSN 0028-0836. Disponível em: <http://www.nature.com/doifinder/10.1038/324446a0>.

BAUMANN, A. et al. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In: *ACM SIGOPS Symposium on Operating Systems Principles*. Big Sky, Montana: ACM, 2009. (SOSP '09), p. 29–44. ISBN 978-1-60558-752-3. Disponível em: <http://portal.acm.org/citation.cfm?doid=1629575.1629579>.

BOYD-WICKIZER, S. et al. Corey: An Operating System for Many Cores. In: *USENIX Conference on Operating Systems Design and Implementation (OSDI)*. San Diego, USA: USENIX Association, 2008. p. 43–57.

BOYD-WICKIZER, S. et al. An analysis of linux scalability to many cores. In: *USENIX Conference on Operating Systems Design and Implementation*. Vancouver, Canada: [s.n.], 2010. (OSDI '10), p. 1–16.

BROQUEDIS, F. et al. ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures. *International Journal of Parallel Programming*, Springer US, v. 38, n. 5-6, p. 418–439, 2010.

CASTRO, M. et al. Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing (PARCO)*, Elsevier, v. 54, p. 108–120, may 2016. ISSN 01678191. Disponível em: <https://linkinghub.elsevier.com/retrieve/pii/S0167819116000417>.

CAVALCANTE, M. et al. Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 28, n. 2, p. 530–543, feb 2020. ISSN 1063-8210. Disponível em: <https://ieeexplore.ieee.org/document/8918510/>.

CHANG, E.; ROBERTS, R. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Communications of the ACM*, v. 22, n. 5, p. 281–283, may 1979. ISSN 0001-0782. Disponível em: <https://dl.acm.org/doi/10.1145/359104.359108>.

Comprés Ureña, I. A.; RIEPEN, M.; KONOW, M. RCKMPI – Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC). In: . Springer, 2011. p. 208–217. ISBN Online ISBN 978-3-642-24449-0. Disponível em: <http://link.springer.com/10.1007/978-3-642-24449-0_24>.

CRUZ, E. et al. Dynamic Thread Mapping of Shared Memory Applications by Exploiting Cache Coherence Protocols. *Journal of Parallel and Distributed Computing (JPDC)*, v. 74, n. 3, p. 2215–2228, 2014.

DAGUM, L.; MENON, R. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, v. 5, n. 1, p. 46–55, 1998.

DAVIDSON, S. et al. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro*, IEEE, v. 38, n. 2, p. 30–41, mar 2018. Disponível em: <https://ieeexplore.ieee.org/document/8344478/>.

De Wael, M. et al. Partitioned Global Address Space Languages. *ACM Computing Surveys (CSUR)*, ACM, v. 47, n. 4, p. 1–27, may 2015. Disponível em: <http://dl.acm.org/citation.cfm?doid=2775083.2716320>.

DIENER, M. et al. Kernel-Based Thread and Data Mapping for Improved Memory Affinity. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, IEEE, v. 27, n. 9, p. 2653–2666, 2016.

DINECHIN, B. D. de et al. A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications. In: *IEEE High Performance Extreme Computing Conference*. Waltham, USA: IEEE, 2013. (HPEC '13), p. 1–6. ISBN 978-1-4799-1365-7. Disponível em: <http://ieeexplore.ieee.org/document/6670342/>.

DINECHIN, B. D. de et al. Guaranteed services of the noc of a manycore processor. In: *International Workshop on Network on Chip Architectures*. Cambridge: ACM Press, 2014. (NoCArc '14), p. 11–16. ISBN 9781450330640. Disponível em: <http://dl.acm.org/citation.cfm?doid=2685342.2685344>.

DINECHIN, B. D. de et al. A distributed run-time environment for the kalray mppa-256 integrated manycore processor. *Procedia Computer Science*, Elsevier, v. 18, n. International Conference on Computational Science, p. 1654–1663, jan 2013. ISSN 1877-0509. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050913004766?via\%3Dihub>.

ENGLER, D. R.; KAASHOEK, F.; O'TOOLE, J. Exokernel: An operating System Architecture for Application-Level Resource Management . In: *ACM SIGOPS Symposium on Operating Systems Principles*. [S.l.]: ACM, 1995. (SOSP '95), p. 251–266. ISBN 0897917154.

FLYNN, M. J. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21, n. 9, p. 948–960, sep 1972. ISSN 0018-9340. Disponível em: <http://ieeexplore.ieee.org/document/5009071/>.

FOLEY, D.; DANSKIN, J. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro*, Elsevier Ltd, v. 37, n. 2, p. 7–17, mar 2017. ISSN 02721732. Disponível em: <http://dx.doi.org/10.1016/j.sse.2015.11.015http://ieeexplore.ieee.org/document/7924274/>.

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)*, Elsevier - Academic Press, Orlando, v. 76, n. C, p. 32–48, february 2015. ISSN 0743-7315. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0743731514002093>.

FU, H. et al. The Sunway TaihuLight Supercomputer: System and Applications. *Science China Information Sciences*, Science China Press, v. 59, n. 7, p. 072001–0720016, jul 2016. ISSN 1674-733X. Disponível em: <http://link.springer.com/10.1007/s11432-016-5588-7>.

GAMSA, B. et al. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In: *USENIX Conference on Operating Systems Design and Implementation (OSDI)*. New Orleans, USA: USENIX, 1999. Disponível em: <https://www.usenix.org/legacy/events/osdi99/full_papers/gamsa/gamsa.pdf>.

GARCIA-MOLINA. Elections in a Distributed Computing System. *IEEE Transactions on Computers*, C-31, n. 1, p. 48–59, jan 1982. ISSN 0018-9340. Disponível em: <http://ieeexplore.ieee.org/document/1675885/>.

GNU. *GOMP – An Implemetation for GCC – GNU*. 2020. Disponível em: <https://gcc.gnu.org/projects/gomp/>.

GROPP, W. et al. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, v. 22, n. 6, p. 789–828, sep 1996. ISSN 01678191. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/0167819196000245>.

HAGHBAYAN, M.-H. et al. Performance/reliability-aware resource management for many-cores in dark silicon era. *IEEE Transactions on Computers (TC)*, v. 66, n. 9, p. 1599–1612, sep 2017. Disponível em: <http://ieeexplore.ieee.org/document/7892847/>.

HASCOËT, J. et al. Asynchronous One-Sided Communications and Synchronizations for a Clustered Manycore Processor. In: *Symposium on Embedded Systems for Real-Time Multimedia*. Seoul: ACM Press, 2017. (ESTIMedia '17), p. 51–60. ISBN 9781450351171. Disponível em: <http://dl.acm.org/citation.cfm?doid=3139315.3139318>.

HO, M. Q. et al. MPI communication on MPPA many-core NoC: Design, modeling and performance issues. In: *International Conference on Parallel Computing*. Edinburgh, UK: IOS Press, 2015. (ParCo '15, v. 27), p. 113–122. Disponível em: <https://doi.org/10.3233/978-1-61499-621-7-113>.

HOWARD, J. et al. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *IEEE Journal of Solid-State Circuits (JSSC)*, v. 46, n. 1, p. 173–183, jan 2011. ISSN 0018-9200. Disponível em: <http://ieeexplore.ieee.org/document/5621843/>.

HUNT, G. C.; LARUS, J. R. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, Association for Computing Machinery, New York, NY, USA, v. 41, n. 2, p. 37–49, abr. 2007. ISSN 0163-5980. Disponível em: <https://doi.org/10.1145/1243418.1243424>.

ISHIGAKI, Y. et al. An FPGA implementation of 3D numerical simulations on a 2D SIMD array processor. In: *International Symposium on Circuits and Systems (ISCAS)*. Lisbon, Portugal: IEEE, 2015. (ISCAS '15), p. 938–941. ISBN 978-1-4799-8391-9. Disponível em: <http://ieeexplore.ieee.org/document/7168789/>.

ISHII, M. et al. Fast modular arithmetic on the kalray mppa-256 processor for an energy-efficient implementation of ecm. *IEEE Transactions on Computers*, v. 66, n. 12, p. 2019–2030, 2017.

JUNIOR, E. P. et al. A Trace-Driven Methodology to Evaluate and Optimize Memory Management Services of Distributed Operating Systems for Lightweight Manycores. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2021. (SAC '21), p. 1190–1198. ISBN 9781450381048. Disponível em: <https://doi.org/10.1145/3412841.3441994>.

KAMIL, S. et al. An auto-tuning framework for parallel multicore stencil computations. In: *International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2010. p. 1–12. ISBN 978-1-4244-6442-5. Disponível em: <http://ieeexplore.ieee.org/document/5470421/>.

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

LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, Association for Computing Machinery, v. 21, n. 7, p. 558–565, jul 1978. ISSN 0001-0782. Disponível em: <https://dl.acm.org/doi/10.1145/359545.359563>.

LEUNG, J.; KELLY, L.; ANDERSON, J. H. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. USA: CRC Press, Inc., 2004. ISBN 1584883979.

LIEDTKE, J. On micro-kernel construction. In: *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. New York, USA: ACM Press, 1995. v. 29, n. 5, p. 237–250. ISBN 0897917154.

MERRILL, D. et al. Scalable GPU Graph Traversal. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. New York, USA: ACM Press, 2012. v. 47, n. 8, p. 117. ISBN 9781450311601.

Mohr, M.; Tradowsky, C. Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores. In: *Design, Automation & Test in Europe Conference & Exhibition*. Lausanne, Switzerland: IEEE, 2017. (DATE '17), p. 1781–1786. ISBN 1558-1101.

MUNSHI, A. The OpenCL specification. In: *IEEE Hot Chips Symposium (HCS)*. Stanford, USA: IEEE, 2009. (HotChips '09), p. 1–314. ISBN 978-1-4673-8873-3. Disponível em: <http://ieeexplore.ieee.org/document/7478342/>.

NIGHTINGALE, E. B. et al. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In: *ACM SIGOPS Symposium on Operating Systems Principles*. Big Sky, Montana: ACM Press, 2009. (SOSP '09), p. 221–234. ISBN 978-1-60558-752-3. Disponível em: <http://portal.acm.org/citation.cfm?doid=1629575.1629597>.

OLOFSSON, A.; NORDSTROM, T.; UL-ABDIN, Z. Kickstarting high-performance energy-efficient manycore architectures with epiphany. In: *Asilomar Conference on Signals, Systems and Computers*. Pacific Grove, USA: IEEE, 2014. (Asilomar '14), p. 1719–1726. ISBN 978-1-4799-8297-4. Disponível em: <http://ieeexplore.ieee.org/document/7094761/>.

PASCOLO, E. et al. OpenMP tasks: Asynchronous programming made easy. In: *International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2016. p. 901–907. ISBN 978-1-5090-2088-1. Disponível em: <http://ieeexplore.ieee.org/document/7568430/>.

PELEG, A.; WEISER, U. MMX technology extension to the Intel architecture. *IEEE Micro*, v. 16, n. 4, p. 42–50, 1996. ISSN 02721732. Disponível em: <http://ieeexplore.ieee.org/document/526924/>.

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. p. 1–11.

PENNA, P. H. et al. Co-Designing Clusters of Lightweight Manycores and Asymmetric Operating System Kernels. *IEEE Embedded Systems Letters*, To appear., p. 1–4, 2020.

PENNA, P. H. et al. On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores. In: *Brazilian Symposium on Computing Systems Engineering*. Natal, Brazil: SBC, 2019. (SBESC '19), p. 1–8. ISSN 2324-7894. Disponível em: <https://hal.archives-ouvertes.fr/hal-02297637>.

PENNA, P. H. et al. Inter-Kernel Communication Facility of a Distributed Operating System for NoC-Based Lightweight Manycores. *Journal of Parallel and Distributed Computing*, 2021. ISSN 0743-7315. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0743731521000794>.

PENNA, P. H. et al. Rmem: An os service for transparent remote memory access in lightweight manycores. In: *International Workshop on Programmability and Architectures for Heterogeneous Multicores*. Valencia, Spain: [s.n.], 2019. (MultiProg '19), p. 1–16. Disponível em: <https://hal.archives-ouvertes.fr/hal-01986366>.

PEREIRA, A. D.; RAMOS, L.; GóES, L. F. W. Pskel: A stencil programming framework for cpu-gpu systems. *Concurrency and Computation: Practice and Experience*, v. 27, n. 17, p. 4938–4953, 2015. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3479>.

PIKE, R. et al. Plan 9 from bell labs. *Computing systems*, Berkeley, CA: University of California Press., v. 8, n. 3, p. 221–254, 1995.

PODESTÁ, E.; NASCIMENTO, B. M. do; CASTRO, M. Energy Efficient Stencil Computations on the Low-Power Manycore MPPA-256 Processor. In: . [s.n.], 2018. p. 642–655. Disponível em: <http://link.springer.com/10.1007/978-3-319-96983-1_46>.

PRONGNUCH, S.; WIANGTONG, T. Performance comparison of coprthr and epiphany sdk on heterogeneous computing platform. In: . [S.l.: s.n.], 2015. (EECON '15), p. 1–4.

RAMEY, C. TILE-Gx100 ManyCore processor: Acceleration interfaces and architecture. In: *Hot Chips Symposium (HCS)*. Stanford: IEEE, 2011. (HCS '11), p. 1–21. ISBN 978-1-4673-8877-1. Disponível em: <http://ieeexplore.ieee.org/document/7477491/>.

RICHIE, D.; ROSS, J.; INFANTOLINO, J. A Distributed Shared Memory Model and C++ Templated Meta-Programming Interface for the Epiphany RISC Array Processor. *Procedia Computer Science*, Elsevier, v. 108, p. 1093–1102, jan 2017. ISSN 1877-0509. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050917308293>.

RICHIE, D. et al. Threaded MPI Programming Model for the Epiphany RISC Array Processor. *Journal of Computational Science (JCS)*, Elsevier, v. 9, n. Computational Science at the Gates of Nature, p. 94–100, jul 2015. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877750315000617>.

ROSS, J.; RICHIE, D. Implementing openshmem for the adapteva epiphany risc array processor. *Procedia Computer Science*, Elsevier, v. 80, n. C, p. 2353–2356, jan 2016. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1877050916309206>.

ROSSI, D. et al. Energy-Efficient Near-Threshold Parallel Computing: The PULPv2 Cluster. *IEEE Micro*, IEEE, v. 37, n. 5, p. 20–31, sep 2017. Disponível em: <http://ieeexplore.ieee.org/document/8065010/>.

SERRES, O. et al. Experiences with UPC on TILE-64 processor. In: *Aerospace Conference*. IEEE, 2011. p. 1–9. ISBN 978-1-4244-7350-2. Disponível em: <http://ieeexplore.ieee.org/document/5747452/>.

SHI, C. et al. A 1000 fps Vision Chip Based on a Dynamically Reconfigurable Hybrid Architecture Comprising a PE Array Processor and Self-Organizing Map Neural Network. *IEEE Journal of Solid-State Circuits (JSSC)*, v. 49, n. 9, p. 2067–2082, sep 2014. ISSN 0018-9200. Disponível em: <https://ieeexplore.ieee.org/document/6853420>.

SOUZA, M. et al. Cap bench: A benchmark suite for performance and energy evaluation of low-power many-core processors. *Concurrency and Computation: Practice and Experience (CCPE)*, Wiley Online Library, v. 29, n. 4, p. 1–18, february 2017. ISSN 1532-0626. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3892>.

TANENBAUM, A. S.; Van Renesse, R. Distributed Operating Systems. *ACM Computing Surveys*, v. 17, n. 4, p. 419–470, dec 1985. ISSN 0360-0300. Disponível em: <https://dl.acm.org/doi/10.1145/6041.6074>.

ULLER, J. F. et al. Enhancing programmability in noc-based lightweight manycore processors with a portable mpi library. In: *Simpósio em Sistemas Computacionais de Alto Desempenho*. Online: SBC, 2020. (WSCAD '20), p. 1–12. ISSN 2358-6613.

ULLER, J. F. et al. Proposta de suporte ao padrão mpi sobre infraestrutura de comunicação de baixo nível no nanvix. In: *Escola Regional de Alto Desempenho da Região Sul*. Porto Alegre, RS, Brasil: SBC, 2020. (ERAD/RS '20), p. 121–124. ISSN 2595-4164. Disponível em: <https://ojs.sbc.org.br/index.php/eradrs/article/view/10771>.

VARGHESE, A. et al. Programming the adapteva epiphany 64-core network-on-chip coprocessor. In: *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Phoenix, USA: IEEE, 2014. (IPDPSW '14), p. 984–992. ISBN 978-1-4799-4116-2. Disponível em: <http://ieeexplore.ieee.org/document/6969488/>.

WALLENTOWITZ, S. et al. A Framework for Open Tiled Manycore System-On-Chip. In: *International Conference on Field Programmable Logic and Applications*. Oslo:

IEEE, 2012. (FPL '2012), p. 535–538. ISBN 978-1-4673-2256-0. Disponível em: <http://ieeexplore.ieee.org/document/6339273/>.

WENTZLAFF, D.; AGARWAL, A. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, ACM, v. 43, n. 2, p. 76–85, apr 2009. ISSN 0163-5980. Disponível em: <http://portal.acm.org/citation.cfm?doid=1531793.1531805>.

WIJNGAART, R. F. van der; MATTSON, T. G.; HAAS, W. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Operating Systems Review (OSR)*, Association for Computing Machinery, New York, NY, USA, v. 45, n. 1, p. 73–83, feb 2011. ISSN 0163-5980. Disponível em: <https://doi.org/10.1145/1945023.1945033>.

YOSHIDA, T. *Fujitsu High Performance CPU for the Post-K Computer*. Cupertino, USA: [s.n.], 2018. 21 p.

ZARUBA, F.; SCHUIKI, F.; BENINI, L. Manticore: A 4096-core RISC-V Chiplet Architecture for Ultra-Efficient Floating-point Computing. *IEEE Micro*, p. 1–1, 2020. ISSN 0272-1732. Disponível em: <https://ieeexplore.ieee.org/document/9296802/>.

ZHANG, I. et al. I'm Not Dead Yet!: The Role of the Operating System in a Kernel-Bypass Era. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. Bertinoro, Italy: ACM, 2019. (HotOS '19), p. 73–80. ISBN 9781450367271. Disponível em: <https://dl.acm.org/doi/10.1145/3317550.3321422>.

ZHENG, F. et al. Cooperative Computing Techniques for a Deeply Fused and Heterogeneous Many-Core Processor Architecture. *Journal of Computer Science and Technology*, Springer US, v. 30, n. 1, p. 145–162, jan 2015. ISSN 1000-9000. Disponível em: <https://link.springer.com/article/10.1007/s11390-015-1510-9>.

APPENDIX A – SUPPORTED PLATFORMS

Nanvix was engineered from the ground up to be portable across different platforms. The current version of our OS supports the following platforms:

- QEMU: a multicore processor emulator that supports dozen of different ISA.
- Unix Simulator: a built-in simulator that runs on top of a host Unix system and uses POSIX abstractions to mimic the main characteristics of a lightweight manycore processor.
- Kalray MPPA-256 Bostan: the second-generation of the silicon Kalray MPPA-256 lightweight manycore processor.
- OpTiMSoC: a FPGA-emulated OpenRISC manycore processor.

In this appendix we stress the portability of Nanvix by providing details on the aforementioned platforms.

A.1 QEMU

QEMU[1] is a fast full-system emulator that provides support for a diverse set of hardware and devices. It relies on dynamic binary translation and hardware-assisted vitalization extensions to leverage a fast emulation experience. QEMU is well-known in the OS development community for multiple reasons. First, it is supported by many general purpose OSes, including Microsoft Windows, Linux and macOS, thereby making it accessible for most of OS engineers. Second, QEMU offer supports for most conventional architectures and hardware, such as x86, PowerPC, ARM, SPARC and RISC-V. Finally, QEMU is open source and integrates with powerful development utilities such as external debuggers.

We used QEMU in the early porting stages of Nanvix to new architectures. Currently, our system runs on the emulated platforms:

- QEMU x86: Single-core 408086 processor (32-bit) with 32 MB Random Access Memory (RAM). In this architecture, the MMU is hardware-managed and address translation is based on a two-level paging scheme. This was the first architecture supported by Nanvix, but is no longer supported by the development team.
- QEMU OpenRISC: Dual-core or1k processor (32-bit) with 16 MB RAM. In this architecture, the MMU is software-managed. In this target, Nanvix embraces a two level-paging scheme and pins on TLB entries those memory pages belonging to the kernel. Due to the lack of an inter-core-interrupt hardware unit, in this target Nanvix couples atomic instructions and fine-grain clock interrupts to enable events to enable inter-core notifications to be sent.

[1] Available at: <https://www.qemu.org/>

- QEMU RISC-V: Quad-core rv32gc processor (32-bit) with 16 MB RAM. In this architecture, the MMU is hardware-managed and address translation is based on a two-level paging scheme. One interesting feature on this target concerns the three-level privileged mode which enables interrupt and exception delegations. Currently, Nanvix runs in supervisor mode, but hardware interrupts are delegated from machine mode to supervisor mode. In future works, we intend as well to delegate software interrupts and exceptions from supervisor to user mode, and thus better match our microkernel-based design.

- QEMU ARMv8: Quad-core ARMv8 processor with 32 MB RAM. In this architecture, the MMU is software-managed. In this target, Nanvix embraces a two level-paging scheme and pins on TLB entries those memory pages belonging to the kernel. Due to the lack of an inter-core-interrupt hardware unit, in this target Nanvix couples atomic instructions and fine-grain clock interrupts to enable events to enable inter-core notifications to be sent.

As a final remark, it is interesting to share our experience over the years using QEMU as an important development utility for getting Nanvix ported across different and diver platforms. For writing the first version of Nanvix for the x86 architecture, it took one work year of an experienced engineer. Then, for OpenRISC 6 months, and for RISC-V about 2 months. Finally, for getting Nanvix running into ARMv8 an intern narrowed down the task in 3 months. In summary, this findings unveil that Nanvix evolved to a easy and portable OS, thereby corroborating to the contributions of our work, as well.

A.2 Unix Simulator

The Unix Simulator is a utility that we wrote to support the initial development of Nanvix. This utility is now embedded into the HAL of Nanvix and it enables one with no access to a LW Manycore processor to run and experiment new ideas.

The Unix Simulator uses POSIX abstractions to model the main characteristics of the processors that Nanvix targets. Specifically we simulated a LW Manycore processor as follows:

- Processes were used to model clusters;
- POSIX Threads were used to model cores;
- Message Queues were used to model low-latency NoC links
- Shared Memory Regions were used to model high-throughput NoC links;
- Statically-Allocated memory chunks were used to model the private local memories.

It is important to stress that the primary goal for developing this utility was to have a behavioral simulator that would enable us to carry out system development. That is, we did not aim performance modeling. Furthermore, the reason why we wrote the Unix Simulator was two fold: (i) existing virtual machines and architectural simulators missed support for

emulating/simulating LW Manycore processors; and (ii) implementing an OS from scratch and directly on top of a silicon processor would be a very challenging task to tackle within the frame of our research.

A.3 MPPA-256 Bostan

The Kalray MPPA-256 Bostan is a silicon LW Manycore processor that features 272 general-purpose cores and 16 firmware-cores, called PE and RM, respectively. The processor is built with 28 nm CMOS technology and it runs at 400 MHz. All cores implement a 64-bit proprietary instruction set, present a 5-issue VLIW pipeline, 8 kB I-Cache and D-Cache, and feature a software-managed MMU.

The 288 cores of Kalray MPPA-256 are grouped into 16 Compute Clusters, which are intended for computation, and 4 I/O Clusters, which are designed to provide connectivity to peripherals. Each Compute Cluster bundles 16 PE, one RM, two NoC interfaces and a 2 MB of local SRAM (128 kB per core). In these clusters, hardware cache coherence is not supported. In contrast, I/O Clusters have 4 RM, 8 NoC interfaces and 4 MB of SRAM (1 MB per core). Two of these clusters are connected to a different DRAM controller, and the other two are attached to PCI and Ethernet controllers. Compute Clusters are not attached to a global memory and they all have private address spaces. Thus, Compute Clusters have to exchange hardware messages by one of two different interleaved 2-D torus NoC to carry out communications: (i) a C-NoC that features low bandwidth and is intended for small data transfers; and (ii) a D-NoC that presents high bandwidth and is dedicated to dense data transfers.

What concerns software development in Kalray MPPA-256, this processor is shipped with a patched version of GCC 4.9.4 and Binutils 2.11.0. No OS is provided by the vendor, and software engineers should rely on a proprietary and non-compliant runtime environment to

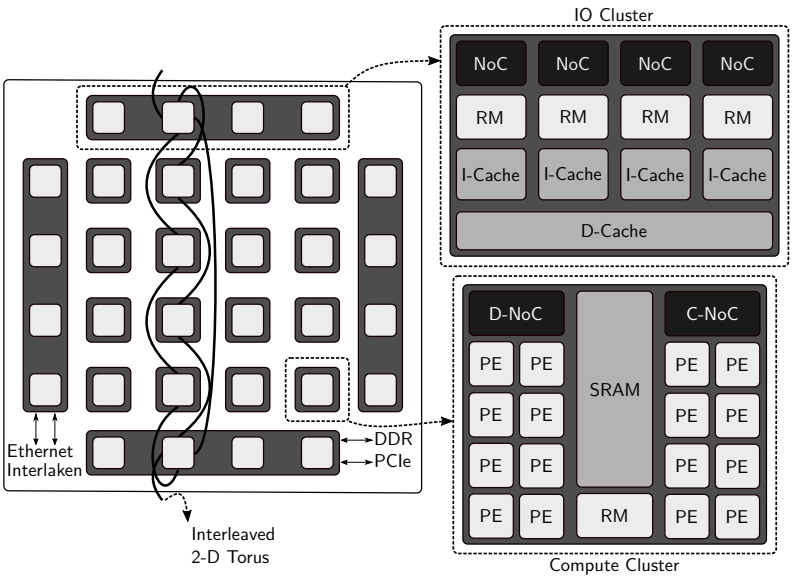


Figure 50 – Architectural overview of the MPPA-256 LW Manycore processor.

write their applications (DINECHIN et al., 2013), if not using Nanvix. Furthermore, regarding OS kernel implementation, system engineers are required to rely on a proprietary hypervisor from Kalray. This hypervisor runs on the firmware cores of the processor and intermediates all low-level operations. Noteworthy, Kalray hypervisor cannot be changed nor configured, and thus it imposes additional challenges in OS construction for this LW Manycore processor.

Kalray MPPA-256 Bostan is the primary architecture supported by Nanvix, because it features most of the system design challenges that we discussed in Section 2.1. For this reason, we present a comprehensive evaluation of OS on this platform, in this work. Specifically concerning the features of Nanvix to cope with the characteristics of Kalray MPPA-256 we highlight the following:

- Decoupled communication abstractions for low-latency high-throughput high-throughput data transfers exposed by the C-NoC and D-NoC. The implementation of Nanvix on Kalray MPPA-256 uses the C-NoC to implement *syncs*, whereas the D-NoC to provide *mailboxes* and *portals*.

A.4 OpTiMSoC

OpTiMSoC is an open-source framework that enables on to prototype OpenRISC-based manycores processors in FPGA (WALLENTOWITZ et al., 2012). Overall, the framework supports a cluster-based organization with three different memory layouts:

- Shared Memory: there is a global and coherent memory is shared clusters. Each cluster features a L2 Cache and a directory coherency controller.

- PGAS: there is a global memory that is partitioned in multiple chunks, each of which is assigned to a cluster. Each cluster has a scratchpad memory, an address translation unit and load store unit that translate memory access addresses.

- Distributed Memory: there is no global memory, but multiple address spaces instead. Each cluster has its own private local memory, which is shared by all cores in that cluster. Data transfers are carried out with message passing via an on-chip interconnect.

In addition to chose one out of the aforementioned organizations, OpTiMSoC enables one to also change the parameters of a cluster, and thus experiment with different hardware configurations. In the base configuration, each cluster has four cores, each of which with a 32-bit, 6-stage OpenRISC pipeline and 8 kB I-Cache. The pipeline features a simple branch predictor; a barrel shifter; a serial divider; a three-stage multiplier; and no Floating Point Unit (FPU). On the other hand, the I-Cache presents a 2-way, 4k-set associativity. Each of these hardware units can be rapidly changed by switching on compilation flags for VHDL code. Table 11 presents those options that are supported out-of-the-box by the project.

Through the development of this thesis, we used OpTiMSoC to evaluate some hardware-software co-design approach for Nanvix. We unveiled the most relevant findings on this alternative research axis in (PENNA et al., 2020). In summary, these are the following key takeaways of our investigation:

- An heterogeneous cluster may leverage the strengths of a microkernel OS design.
- Nanvix barely uses floating point operations, multiplications and divisions, and the source code could be easily patched to not use them at all, without hurting performance.
- A simple directed-mapped cache better handles the workload of the Nanvix kernel, in contrast to a set-associative cache for targeting user workload.
- When running Nanvix on a co-designed cluster we reduce power consumption on up to 14.1% and improve performance by up to 6.5%.

Table 11 – Design space for cores in OpTiMSoC.

Hardware Unit	**Parameters**
Branch Predictor	Simple, SatCounter and GShare
Shift Unit	Serial and Barrel
Divider	None and Serial
Multiplier	None and Three-Stage
FPU	None, IEEE-754
I-Cache	1-way 8k-set, 2-way 4k-set, 4-way 2k-set, 8-way 1k-set

APPENDIX B – EXTENDED ABSTRACT IN PORTUGUESE

B.1 Introdução

As comunidades científicas e industriais estão constantemente buscando e desenvolver soluções para atender às crescentes demandas de desempenho de aplicativos de software. Essas soluções são frequentemente adaptadas para complexidades específicas de um aplicativo e plataforma de destino, e variam do nível de software e suporte do compilador (KAMIL et al., 2010), para o sistema de tempo de execução (BROQUEDIS et al., 2010) e o hardware subjacente arquitetura (DINECHIN et al., 2013). Desta forma, técnicas especializadas podem ser aplicado e, assim, alcançado um desempenho de ponta.

B.1.1 Motivação

Especificamente nos esforços de nível de hardware, LW Manycore processadores (ISHII et al., 2017) surgiu como uma alternativa promissora para fornecer escalabilidade de desempenho com Baixo consumo de energia. Para habilitar esses recursos, esses processadores contam com um conjunto selecionado de característica arquitetônica que apresenta múltiplos desafios em programabilidade e portabilidade de software. Primeiro, a *integração do circuito de alta densidade* transforma o silício escuro em realidade (HAGHBAYAN et al., 2017). Em segundo lugar, a *arquitetura de memória distribuída* leva a um não-trivial design de software (CASTRO et al., 2016). Terceiro, a *interconexão no chip* convida os engenheiros a abraçar um modelo de programação de passagem de mensagens e, portanto, trabalhar em refatoração significativa em seu software (SOUZA et al., 2017). Finalmente, a *heterogeneidade no chip* torna a implantação de aplicações em LW Manycore complex (BARBALACE et al., 2015).

B.1.2 Problema

Existem duas abordagens principais para abordar a programação de software e desafios de portabilidade em uma arquitetura: baremetal runtime systems e SOs. O primeiro visa expor um ambiente de programação de ponta do hardware para atender às necessidades de uma arquitetura específica (HO et al., 2015; Mohr; Tradowsky, 2017). No contraste, a segunda abordagem anseia por superar as complexidades do hardware em um sentido mais amplo, expondo abstrações e programação ricas interfaces, bem como lidar com a alocação de recursos, compartilhamento e multiplexação.

No contexto de processadores LW Manycore, infelizmente hoje, nenhuma das duas abordagens anteriores oferece uma solução completa para enfrentar os desafios de programação e portabilidade. Por um lado, existente baremetal runtime systems que visam esses processadores emergentes (VARGHESE et al., 2014; RICHIE; ROSS; INFANTOLINO, 2017) inerentemente carece de cobertura de todos os problemas que afirmamos em anteriormente (Seção 1.1). No

outro Por outro lado, os sistemas existentes não lidam com a arquitetura complexidades de LW Manycore como segue.

a) Em uma extremidade do espectro, SOs centralizados intrinsecamente enfrentam barreiras importantes:
   - Eles têm problemas de escalabilidade (WENTZLAFF; AGARWAL, 2009).
   - Eles não lidam com cada vez mais diversificada hardware (BARBALACE et al., 2015).
   - Não são viáveis para o contexto devido à pequena quantidade de memória on-chip.

b) Na outra extremidade do espectro, SOs distribuídos perdem alguns restrições arquitetônicas dos processadores LW Manycore (PENNA et al., 2021), e assim não é possível implantar estes SOs nas arquiteturas supracitadas, sem um redesenho completo e / ou mudanças significativas para seu código-fonte. A justificativa para isso reside nos seguintes pontos:
   - Eles não dependem de uma visão descentralizada do hardware. Por exemplo, eles assumem que os recursos de hardware são visíveis para todos os núcleos do processador subjacente.
   - Embora apresentem uma organização de software distribuída, eles não lidam com os vários espaços de endereço físico. Elas dependem de áreas de memória compartilhada para transferir mensagens de software e mover dados de um núcleo para outro.
   - Eles não têm como uma de suas principais restrições seus pegada de memória e, portanto, inerentemente não lidar com pequenas memórias on-chip de LW Manycore.
   - Eles ignoram as interconexões on-chip e não exploram recursos adicionais que geralmente estão disponíveis em LW Manycore, como mensagem assíncrona passando por DMA e roteamento de mensagem.

Com base nessas observações, formulamos o seguinte problema, que permanece sem resposta pelo estado da arte em Sistemas Operacionais:

> *Como projetar um sistema operacional de modo a abordar problemas de programação e portabilidade em processadores lightweight manycore?*

B.1.3 Objetivos

Em busca de uma resposta para o problema apontado anteriormente, o principal objetivo desta tese se resume a *arquitetura de ponte características de processadores leves de muitos núcleos*. Para este fim, nós propomos um SO distribuído que visa especificamente o seguinte:

- *Programação*: para fornecer abstrações de sistema ricas para processadores LW Manycore, como processo, arquivos, memória virtual e primitivas de comunicação; bem como para permitir o compartilhamento de recursos, multiplexação e isolamento nessas arquiteturas emergentes.

- *Portabilidade*: para expor a programação padrão interfaces de modo a permitir que o software existente seja transferido para LW Manycore processadores.

B.1.4 Contribuições

A principal contribuição desta tese é um OS distribuído que avança o gerenciamento de recursos em processadores LW Manycore. Por um lado, do ponto de vista científico, esta contribuição principal pode ser desdobrada nas seguintes contribuições específicas:

- Uma HAL abrangente para LW Manycore (PENNA; FRANCIS; SOUTO, 2019) em que: (i) une as diferenças arquitetônicas em vários arquiteturas; e (ii) lidar com os principais problemas que são frequentemente encontrados ao projetar an OS para esses processadores. Com esta camada de abstração, o desenvolvimento e implantação de um OS completo se torna mais fácil não apenas para um LW Manycore específico, mas também a portabilidade de um próprio OS em vários desses processadores.

- Uma abordagem de gerenciamento de memória rica que é baseada em a DPS. Isto é uma nova solução que criamos para gerenciar a memória de um processador LW Manycore e funciona da seguinte maneira. As memórias locais de um processador LW Manycore são considerados caches de página que podem armazenar dados de quaisquer processos. Sediada sobre isso, o OS gerencia este cache de página por: (i) colocar aquelas páginas que estão sendo muito utilizadas por um processo nas memórias locais; e (ii) trocando aquelas páginas que não são usadas por um processo para memórias remotas. No geral, esta solução permite o acesso transparente aos dados e manipulação, bem como compartilhamento de dados e coerência em LW Manycore. Nós acreditamos que esses problemas estão relacionados de maneira importante à programação e desafios de portabilidade que pretendemos superar.

- Uma facilidade de comunicação leve que gerencia a interconexão em chip e expõe primitivos ricos (PENNA et al., 2021). Geral, esta facilidade expõe três abstrações de comunicação: (i) sincronizações, para habilitar um processo para sinalizar e desbloquear outro processo remotamente; (ii) caixas de correio, para envio de mensagens de tamanho fixo com baixa latência; e (iii) portais, para lidar com transferências de dados densos com alta largura de banda. Esse infra-estrutura de comunicação permite efetivamente o eficiente implementação de nosso OS.

Por outro lado, destacamos que esta tese também introduz um importante contribuição técnica para a comunidade:

- Nanvix, uma implementação concreta de um OS que apresenta o avanços científicos mencionados para processadores LW Manycore. Nanvix suporta múltiplas arquiteturas (i.e., k1b, x86, OpenRISC, ARMv8 e RISC-V), é executado em processadores LW Manycore de silício, bem como expõe abstrações e interfaces de programação de alto nível.

B.2 Referencial Teórico

Neste capítulo, cobrimos os fundamentos de nossa tese. Primeiramente, nos aprofundamos nas arquiteturas de muitos núcleos de chip único e suas principais características. A seguir, apresentamos o runtime systems. Então, discutimos sobre OSes e resumimos sobre sistemas distribuídos. Finalmente, concluímos este capítulo posicionando nosso trabalho neste terreno.

B.2.1 Processadores Manycore Single-Chip

Processadores paralelos foram introduzidos para superar a escalabilidade barreira no nível do material conhecida como *Power Wall* (CASTRO et al., 2016). Em vez de aumentar a instrução rendimento em um único CPU, a ideia alternativa era aumentar o número real de núcleos em a CPU. Desta forma, execuções de múltiplas streams podem ser realizadas ao mesmo tempo, e paralelismo de nível de thread poderia ser explorado para alcançar escalabilidade de desempenho em um aplicativo. Neste contexto, uma categorização bem conhecida foi proposto por Flynn (FLYNN, 1972), em que as arquiteturas podem ser agrupados de acordo com suas instruções e fluxos de dados:

- SISD: um único processador executa um único fluxo de instrução para operar em dados armazenados em uma única memória.
- SIMD: uma única instrução de máquina controla o simultâneo execução de uma série de elementos de processamento em uma base travada. Cada elemento de processamento tem uma memória de dados associada, de modo que cada a instrução é executada em um conjunto diferente de dados por diferentes processadores.
- MISD: uma sequência de dados é emitida para um conjunto de processadores, cada um dos quais executa uma sequência de instruções diferente.
- MIMD: um conjunto de processadores simultaneamente executa diferentes sequências de instruções em diferentes conjuntos de dados.

Apesar de ser uma taxonomia sólida, com o recente aumento de núcleos em arquiteturas, pode-se também argumentar sobre arquitetura extra características que podem ser contabilizadas para categorizar manycore de chip único processadores. Esta classificação estendida é descrita em Figura 51 e adiciona a Aspectos da abordagem de Flynn sobre o sistema de memória de chip único arquiteturas manycore. Com base nesta característica adicional, nós destacar três organizações que são relevantes para a compreensão deste tese: *aceleradores de muitos núcleos*, *processadores de muitos núcleos* e *LW Manycore processadores*.

Os aceleradores Manycore foram a primeira classe de processadores manycore de chip único a surgir. Eles compreendem aquelas arquiteturas que são estreitadas para aplicações específicas cargas de trabalho, como processamento vetorial, visão computacional e inteligência. No geral, aceleradores manycore costumam apresentar uma organização SIMD e por este

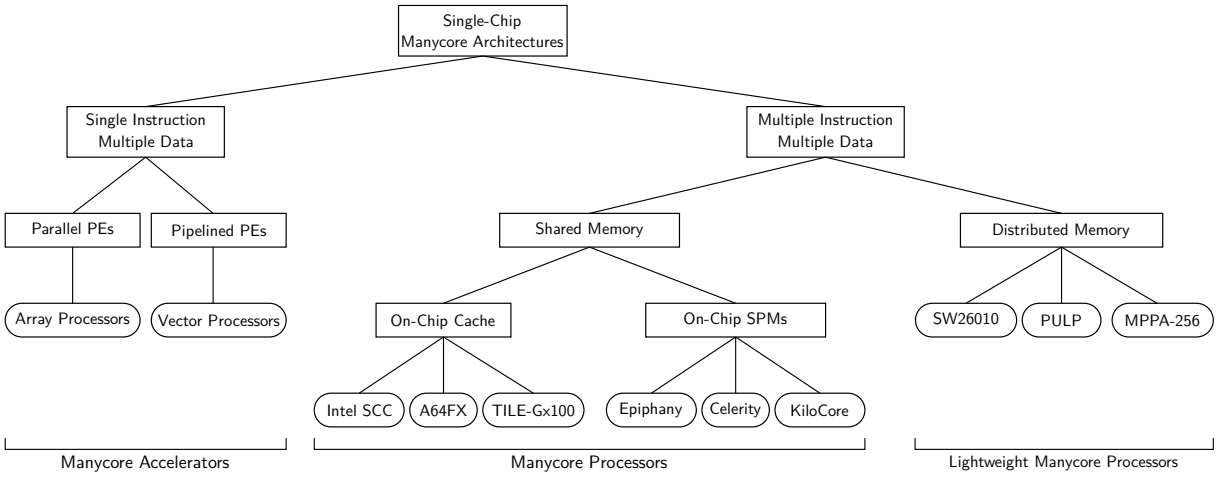


Figure 51 – Uma taxonomia estendida para arquiteturas manycore de chip único.

motivo são frequentemente acoplados a um processador de uso geral para construir um sistema heterogêneo. Entre muitos aceleradores principais, destacamos dois subclasses que diferem na forma como os núcleos são organizados: *processadores de matriz* e *processadores vetoriais*.

Processadores Manycore endereçam o domínio HPC de propósito geral e são freqüentemente usados como blocos de construção básicos para supercomputadores. Esses processadores apresentam um sistema de memória em que todos os núcleos podem ler/escrever de forma transparente usando carregar/armazenar instruções. Em última análise, processadores manycore não exigem grande mudanças de paradigma de programação e, portanto, são atualmente a classe mais conhecida entre arquiteturas manycore de chip único. Alguns exemplos de processadores manycore são Intel Single-Cloud Computer (HOWARD et al., 2011), Tilera TILE-Gx100 (RAMEY, 2011), Adapteva Epiphany (OLOFSSON; NORDSTROM; UL-ABDIN, 2014), Celerity (DAVIDSON et al., 2018), Fujitsy A64FX (YOSHIDA, 2018) e Manticore (ZARUBA; SCHUIKI; BENINI, 2020).

Especificamente em relação aos processadores manycore, destacamos dois diferentes organizações que são frequentemente adotadas. No primeiro, núcleos são fisicamente agrupados em unidades físicas chamadas clusters (ou blocos), que são totalmente interconectado com o sistema global de memória compartilhada por meio de um NoC. Núcleos no mesmo cluster são fortemente acoplados a um controlador de memória local, um cache local e um roteador NoC. Os controladores de memória permitem acesso transparente a a memória compartilhada global encaminhando solicitações de carregamento/armazenamento por meio do NoC. Além disso, esses controladores de memória também fornecem suporte de coerência de cache, por implementando-o em hardware ou expondo instruções especiais que permitem isso em nível de software. Em uma segunda organização para um processador manycore, os núcleos também são agrupados em clusters que são interconectados por meio de um NoC. No entanto, o sistema de memória não depende de um esquema de cache. Em vez disso, os núcleos no mesmo cluster compartilham o acesso a um SPM local módulo. Além disso, o espaço de endereço global é estaticamente particionado entre os vários SPM e controladores de memória para a frente acessos

de memória remota ao módulo SPM em questão.

LW Manycore processadores têm um esforço para fornecer alto desempenho com energia eficiência. Para este fim, eles contam principalmente com um conjunto distinto de características arquitetônicas. No geral, LW Manycore pode ser visto como " clusters em um chip " e apresentar as seguintes características principais:

- Eles são projetados para lidar com cargas de trabalho MIMD.
- Eles integram até milhares de núcleos de baixo consumo de energia em um único lasca.
- Eles têm seus núcleos dispostos em grupos fortemente acoplados chamados clusters.
- Eles apresentam uma arquitetura de memória distribuída com vários espaços de endereço.
- Apresentam memória restrita, com pequena (poucos kB por núcleo) e memórias locais privadas.
- Eles contam com NoC para uma passagem rápida e confiável de mensagens comunicação.
- Eles têm uma configuração heterogênea em termos de I/O e/ou recursos de computação.

B.2.2 Ambientes de Execução

Runtime systems fornece ambientes de programação ricos para uma execução particular modelo. Para fazer isso, eles expõem API específicos para aplicativos; e eles podem interagir diretamente com o hardware, OS e/ou compilador. Algum exemplos bem conhecidos de runtime systems são o crt0, o ART e o JRE. Esses sistemas permitem principalmente um ambiente de programação de propósito geral para aplicativos, expondo rotinas e abstrações para manipular recursos como fluxos de dados, sinais/exceções de software e memória heap. Não obstante, no contexto deste trabalho, discutiremos a seguir sobre runtime systems que se concentra em processadores manycore. No geral, as alternativas exstentes dizem respeito principalmente a frameworks de programação paralela e distribuída.

Ambientes de programação paralela expõem uma infraestrutura para aplicativos para criar, manipular e sincronizar vários fluxos de execução dentro de um programa. Dessa forma, um aplicativo pode explorar completamente o desempenho que é fornecido por uma arquitetura paralela. Destacamos três ambientes que estão frequentemente disponíveis para arquiteturas manycore: PThreads, OpenMP (DAGUM; MENON, 1998) e OpenCL (MUNSHI, 2009). O primeiro encapsula um fluxo de execução em uma abstração chamada *thread*, e ele expõe um conjunto de rotinas que permitem aos aplicativos manipular threads. O ambiente PThreads permite flexibilidade máxima em paralelo programação e estão normalmente disponíveis como uma biblioteca vinculada estaticamente. Em seguida, o OpenMP runtime foca em um suporte transparente para o fork-join modelo de programação paralela. Para este fim, o sistema de tempo de execução interage com a infraestrutura do compilador para permitir que o engenheiro de

software anote o código-fonte para criar regiões paralelas. Então, ao construir o aplicativo a partir de um código-fonte anotado, o compilador faz a ligação de baixo nível rotinas para gerar e encerrar threads no início e no final de um paralelo região, respectivamente. Finalmente, o ambiente OpenCL visa o paralelo programação em arquiteturas heterogêneas. Ele define uma estrutura para escrever programas onde o kernel do aplicativo é executado no OpenCL dispositivo, enquanto a parte restante do código é executada no host dispositivo. Para fazer isso, o OpenCL trabalha com a infraestrutura do compilador para gerar código para ambas as arquiteturas, bem como interage com o hardware/OS para baixar/descarregar binários e dados de/para os dispositivos.

Ambientes de programação distribuída permitem uma exploração completa de arquiteturas distribuídas com foco no fornecimento de comunicação abstrações entre vários nós de computação em uma plataforma. Notamos dois ambientes de programação distribuída que são normalmente encontrados em arquiteturas manycore: MPI (GROPP et al., 1996) e PGAS (De Wael et al., 2015). O primeiro expõe um conjunto de rotinas que permitem que dados arbitrários sejam enviar/receber usando abstrações chamadas mensagens. As mensagens podem ter um destinatários únicos ou múltiplos e podem ser enviados/recebidos de forma síncrona e/ou de forma assíncrona. Por outro lado, PGAS permite a comunicação unilateral mantendo um espaço de endereço global que é particionado e espalhado por todo o vários pares do aplicativo. Os pares podem ler/gravar dados neste endereço espaço por meio de funções especiais put/get, e o sistema de tempo de execução garante que a coerência é mantida.

B.2.3 Sistemas Operacionais

Um Operating System (OS) é uma camada de software que fica no topo do hardware baremetal, gerencia recursos de computação e fornece serviços comuns para programas. Sem um OS, os engenheiros teriam que lidar com as complexidades de hardware por meio de seus própria, aumentando assim a complexidade do software e o custo de desenvolvimento. Um OS tem duas funções importantes em uma plataforma de computação:

(i) Recursos abstratos: amplia as funcionalidades do hardware e expor a serviços de programas do usuário e rich API que são mais fáceis de lidar com.

(ii) Gerenciar Recursos: acesso multiplex a hardware e software recursos, garantindo a proteção e integridade dos dados, bem como fazendo cumprir protocolos e políticas de administração.

A Figura 60 apresenta um esquema de um OS. Isto é composto por vários componentes que interagem entre si para fornecer rica abstração e gerenciamento de recursos. Digno de nota, existem muitas maneiras de acoplar esses componentes. Portanto, nos parágrafos a seguir nos concentramos em discutindo suas funções, e em Seção 2.3.2, damos percepções sobre como eles podem ser unidos.

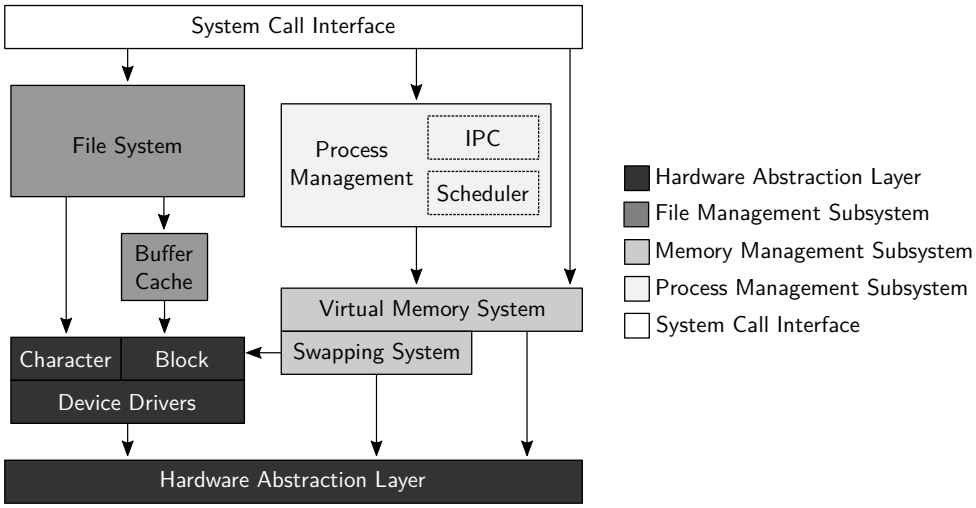


Figure 52 – Uma possível arquitetura para um sistema operacional.

O HAL é o componente de nível mais baixo de um OS. Ele interage diretamente com o hardware e expõe um API comum para lidar com baremetal estruturas, como vetores de interrupção e tabelas MMU. O papel principal deste componente é ocultar o máximo possível todas as complexidades de hardware, também para fornecer uma interface padrão em várias arquiteturas. Por aqui, os componentes OS sobrejacentes tornam-se menos complexos e mais fáceis de serem transportados de um hardware para outro.

O MMS fornece uma visão do sistema de memória subjacente. Normalmente faz então, tendo dois módulos trabalhando juntos: a troca e a memória virtual módulos. O primeiro trata da paginação, mantendo na memória as páginas que são usadas. O sistema de memória virtual, por outro lado, depende da paginação usados com mais frequência, e trocando para armazenamento secundário aqueles módulo para habilitar recursos avançados, como regiões de memória compartilhada, sob demanda carregamento, cópia lenta e fixação de página. Discutimos ainda os aspectos internos de a MMS na Seção 2.3.3.

O PMS lida com todas as tarefas que envolvem programas. Para fazer isso, ele depende em uma abstração chamada processo, que encapsula o fluxo de execução e o conjunto de recursos usados por um programa. No geral, o PMS expõe rotinas para criar, encerrar e agendar processos, bem como abstrações que permitem que os processos troquem informações entre si e sincronizar suas atividades. Apresentamos uma discussão detalhada de um PMS na seção 2.3.4.

O FSM fornece uma interface uniforme para programas para armazenar e recuperar dados de armazenamento de dados secundários, bem como para manipular dispositivos de hardware e recursos de software. Para conseguir isso, ele cria e mantém o abstração de arquivo. Os arquivos podem ser organizados em diretórios, podem ser acessados através de um caminho único, podendo ser compartilhado entre vários processos de forma transparente. Discutimos sobre este componente em Seção 2.3.5.

As bibliotecas do sistema ficam por cima de todos os componentes e são expostas ao usuário programa uma coleção de rotinas conhecidas como chamadas de sistema. Isto é um

interface bem definida na qual os programas do usuário dependem para interagir com o sistema. No geral, as bibliotecas do sistema não funcionam muito além da sanidade verificar os parâmetros e garantir a proteção. Normalmente, bibliotecas de usuário envolver bibliotecas do sistema para fornecer interfaces e abstrações que são ainda mais amigáveis a engenheiros de software.

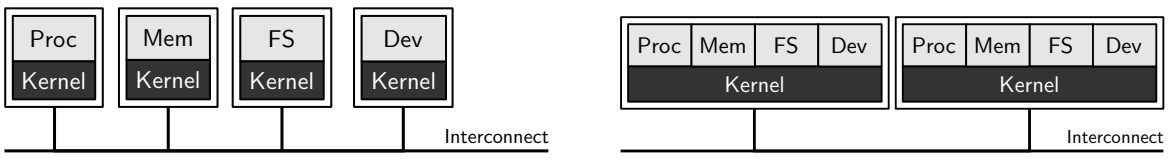
B.2.4 Sistemas Distribuídos

Até agora, pouco discutimos sobre questões específicas que dizem respeito ao sistema design para arquiteturas distribuídas, que é o caso de processadores LW Manycore. Portanto, para estreitar nossa discussão, nesta seção nós cobrir os antecedentes relativos aos sistemas distribuídos.

Em sistemas distribuídos, uma questão importante diz respeito à maneira como o o próprio sistema é estruturado. No geral, existem duas abordagens ortogonais: arquiteturas centralizadas e descentralizadas. Essas alternativas são delineadas na Figura 53 e discutido a seguir.

Em uma arquitetura centralizada, o sistema é organizado como uma coleção de processos especiais chamados servidores, cada um dos quais fornecendo um conjunto de funcionalidades diferenciadas aos processos do cliente. Os servidores podem implementar funcionalidades inteiras ou específicas; assim como eles podem interagir uns com os outros para implementar operações complexas. Esse abordagem explora a distribuição vertical e é a mais simples de implementar. Porém, ele tem dois problemas, primeiro, ter um serviço centralizado em um servidor pode levar a problemas de escalabilidade. Em segundo lugar, se um servidor falhar, o serviço subjacente torna-se indisponível. Figura 53a descreve um arquitetura centralizada para a Distributed OS. Neste exemplo, o OS é fatorado em quatro servidores, que por sua vez estão espalhados por quatro nós diferentes. Cada servidor implementa um subsistema diferente e alguns deles cooperam.

Em uma arquitetura descentralizada, o sistema distribuído é estruturado como um conjunto de servidores, cada um dos quais implementando o conjunto completo de funcionalidades. Cada servidor fornece o mesmo conjunto de interfaces, mas atende clientes diferentes e potencialmente operar em dados diferentes. Esta arquitetura explora distribuição horizontal e oferece melhor escalabilidade, no entanto, a desvantagem está no fato de que essa arquitetura é mais difícil de projetar e implementar. Figura 53b apresenta um arquitetura descentralizada para a Distributed OS. Neste exemplo, cada nó fornece o conjunto completo de serviços OS



(a) Centralizado organização. (b) Organização descentralizada.

Figure 53 – Arquiteturas para sistemas operacionais distribuídos.

e lida com diferentes processos do cliente. Ao contrário de uma organização centralizada, os componentes precisam se comunicar menos freqüentemente, apenas para manter estruturas de dados compartilhadas consistentes.

B.3 Trabalhos Relacionados

Existem duas abordagens complementares que visam apoiar desenvolvimento de software em processadores manycore de chip único: *Runtime Systems* e *Operating Systems (OSes)*

Runtime systems interage com o hardware e/ou compilador para expor API que facilitar a programação em um domínio de aplicativo específico. Especificamente no caso de processadores manycore de chip único, runtime systems visa principalmente entregando programação paralela e/ou distribuída de alto desempenho ambientes. Neste contexto, notamos porque esses runtime systems não cobrem o desafios abertos em processadores LW Manycore:

- *Bibliotecas específicas de fornecedores* são projetadas para um arquitetura e, inerentemente, não fornecem qualquer suporte para software portabilidade.
- *Bibliotecas de metaprogramação de modelo* oferecem portabilidade apenas para os aplicativos e plataformas que eles visam, o que é limitado.
- *MPI and PGAS Libraries* não lidam com as pequenas quantidade de memória on-chip nem permitem acesso transparente à memória vários espaços de endereço físico.

Além disso, é importante destacar que runtime systems inerentemente falta de fornecimento de gerenciamento de recursos ricos como alocação, isolamento, compartilhamento e multiplexação. Colocando de outra forma, eles não são o suficiente para fornecer uma solução completa para programação e portabilidade para processadores LW Manycore.

Eles anseiam por superar as complexidades de uma arquitetura, expondo abstrações, interfaces de programação e gerenciamento de recursos. Infelizmente, os sistemas existentes lutam para resolver totalmente a programabilidade e desafios de portabilidade em processadores LW Manycore. A Figura 54 descreve a razão para isso e em seguida, fazemos uma discussão mais aprofundada sobre o assunto.

Por um lado, totalmente OSes como Barrelfish, FOS e HeliOS não visam principalmente LW Manycore e não foram projetados para abordar restrições arquitetônicas desses processadores. Por exemplo, eles não consideram os vários espaços de endereço físico, nem lidam com os pequenos quantidade de memórias no chip. Em resumo, não é possível implantar esses OSes on LW Manycore sem um redesenho completo e significativo mudanças em seu código.

Por outro lado, MOSSCA e M^3 têm como alvo LW Manycore, mas ainda falta em fornecer programabilidade e portabilidade para estes processadores. No geral, a justificativa para isso reside no fato de que ambos não oferecem suporte para memória virtual, nem qualquer meio de transparência acessos em vários espaços de endereço e expansão do espaço de endereço. Os aplicativos escritos para esses sistemas devem contar com API e lidar com o acesso de dados por

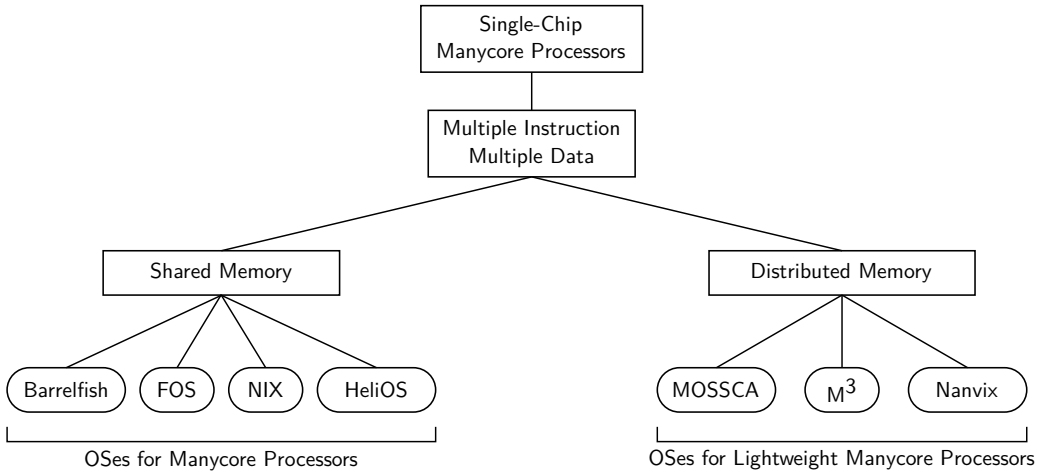


Figure 54 – Sistemas operacionais para arquiteturas multicore de chip único.

conta própria, incorrendo em não portabilidade e má programação. Além disso, é importante notar que MOSSCA e M^3 processadores caseiros simulados LW Manycore, com características muito especiais. Portanto, não é trivial determinar a qual medida que eles podem ser transferidos para outras arquiteturas semelhantes, incluindo silício salgadinhos.

Em contraste com esses OSes, Nanvix fornece uma solução completa para programabilidade e portabilidade para processadores LW Manycore. Para isso final, ele depende de um sistema DPS para superar os desafios relacionados à memória, apresenta uma facilidade de comunicação leve para permitir um on-chip eficiente comunicação, e é baseado em um abrangente HAL que permite portabilidade do sistema para uma variedade de arquiteturas. No próximo capítulo, detalhamos nossa solução.

B.4 O Sistema Operacional Nanvix

Nanvix é um OS distribuído que visa abordar a programabilidade e problemas de portabilidade em processadores LW Manycore. Em resumo, esse SO concretiza as contribuições apresentadas nessa tese. Nanvix é de código aberto [1] e é resultado de uma colaboração conjunta entre PUC Minas, UFSC e UGA.

B.4.1 Objetivos de Projeto

Para abordar a programabilidade e portabilidade em LW Manycore, buscamos o seguintes objetivos de design para Nanvix:

- Programação
  - Habilita a multiplexação de recursos para tornar possível implantar simultaneamente vários aplicativos no processador e, assim, aumentar a utilização geral da plataforma.

[1] Disponível publicamente em: `https://github.com/nanvix`.

  - Garanta o isolamento dos recursos para evitar que os aplicativos tenham têm desempenho impactado ou sua execução temperado.

- Portabilidade
  - Fornece abstrações ricas, como processos e arquivos. Dessa forma, os engenheiros de software não precisam se preocupar com implementar, incorporar e mantê-los em seu software
  - Expor interfaces de programação padrão, permitindo assim software a ser portado com mudanças mínimas de sintaxe.
  - Suporta múltiplos paradigmas de programação (i.e., compartilhada e memória distribuída), permitindo assim uma mudança suave de arquiteturas multicore e manycore tradicionais.

B.4.2 Arquitetura do Sistema

Nanvix apresenta um design multikernel (BAUMANN et al., 2009; WENTZLAFF; AGARWAL, 2009) que é estruturado em várias instâncias superiores de um microkernel. Nós escolhemos esta organização porque acreditamos que ela atende melhor a características de LW Manycore. Por um lado, a abordagem multikernel lida com a arquitetura de memória distribuída de esses processadores. Por outro lado, a estrutura baseada em microkernel permite uma implementação leve e modular de serviços de sistema, que é um requisito fundamental dada a escassez de hardware Recursos. A Figura 55 descreve uma arquitetura visão geral de Nanvix. Ele tem um layout de quatro camadas que apresentaremos a seguir.

O HAL expõe uma abstração abrangente do subjacente hardware (PENNA; FRANCIS; SOUTO, 2019). É implementado como uma biblioteca baremetal, e ele envia rotinas comuns e estruturas de dados para manipular hardware estruturas, como núcleos, MMU e NoC. Em resumo, o HAL habilita o portabilidade de nosso OS em vários processadores LW Manycore, como Kalray MPPA-256 (DINECHIN et al., 2013), OpTiMSoC (WALLENTOWITZ et al., 2012) e PULP (ROSSI et al., 2017). Na seção 4.3, discutimos sobre esta camada.

O microkernel fornece compartilhamento de recursos, multiplexação, isolamento e proteção, dentro de um cluster de um LW Manycore (PENNA et al., 2019). Corre em modo

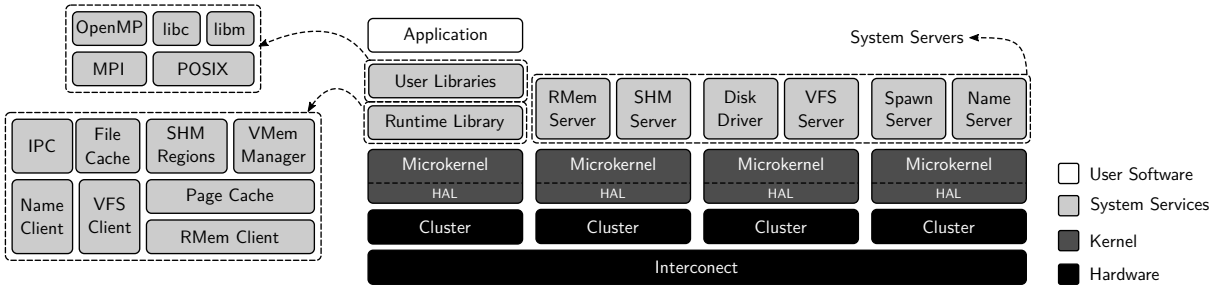


Figure 55 – Uma visão geral estrutural de Nanvix.

privilegiado e expõe abstrações e primitivas mínimas de OS, como threads, espaço de endereço virtual e comunicação entre processos mecanismos. Nosso microkernel apresenta um design assimétrico, o que significa que ele é executado exclusivamente em um núcleo do cluster e deixa o restante núcleos para uso geral. Este projeto atenua a interferência do kernel no software de nível de usuário e, portanto, melhora o desempenho. Na Seção 4.4 detalhamos o microkernel de Nanvix.

Os servidores do sistema são implantados no topo do microkernel e implementam serviços comumente encontrados em um OS, como agendamento de processos, memória mapeamentos e alocação de arquivos. Por sua vez, os subsistemas são implementados tendo um conjunto desses servidores de sistema que funcionam juntos, de forma distribuída moda. Na seção 4.5, descobrimos o serviços de Nanvix. Finalmente, no topo dos servidores do sistema, as bibliotecas de tempo de execução são descartadas. Esses bibliotecas estão vinculadas aos aplicativos do usuário e fornecem um padrão interface para software de nível de usuário interagir com o OS. Na Seção 4.6 discutimos brevemente sobre as bibliotecas e tempos de execução apresentados por Nanvix.

B.5 Metodologia de Avaliação

Nossa metodologia de avaliação foi elaborada de forma a responder o seguinte três questões principais, que estão alinhadas aos objetivos deste trabalho (ver Seção 1.3):

Q-1 Como o Nanvix melhora a programabilidade em processadores manycore leves?

Q-2 Até que ponto o Nanvix melhora a portabilidade?

Q-3 Qual é o desempenho do Nanvix?

B.5.1 Programas de Experimentação

Nós consideramos vários programas experimentais para avaliar especificamente o desempenho de Nanvix e, portanto, responder a pergunta de avaliação Q-1. No geral, esses programas podem ser agrupados de acordo com suas características e propósitos de avaliação da seguinte forma:

(i) *Programas Sintéticos* são implementados no topo do sistema de tempo de execução de Nanvix (libNanvix) e eles são projetados para fazer o uso mais eficiente do sistema. Esses programas se concentram no medição de recursos específicos do sistema e os empregamos para compreender o desempenho dos subsistemas individuais de Nanvix.

(ii) *Utilitários do Sistema* consiste em um conjunto de utilitários Unix que foram portados para Nanvix, além de sua biblioteca C (uLibc). Esses programas forçar a interação entre os subsistemas, bem como entre o usuário e bibliotecas do sistema. Contamos com esses

Table 12 – Programas experimentais usados para avaliar Nanvix

Grupo	Programa	Breve Descrição
Programas Sintéticos	`fence`	benchmarks latency of *syncs*
	`mail`	avalia a latência de *mailboxes* de correio
	`cargo`	mede a taxa de transferência de *portals*
	`lkcall`	clocks latência das chamadas locais do kernel
	`rkcall`	times de latência das chamadas remotas do kernel
	`forkjoin`	avalia o custo para criar e encerrar threads
	`knoise`	avalia a interferência introduzida pelo kernel em um aplicativo do usuário
	`lookup`	mede a latência para determinar a localização física de um processo
	`heartbeat`	avalia a latência para enviar uma mensagem viva para o gerenciador de processo
	`stream`	avalia a width de banda de leitura/gravação de/para a memória local
	`rstream`	avalia a width de banda de leitura/gravação de/para a memória remota
	`pginval`	clica a latência para invalidar uma entrada do cache de página
	`pgfetch`	vezes a latência para transferir uma página do servidor de memória remoto
Utilitários de Sistema	`diff`	calcula a menor subsequência entre duas strings
	`grep`	procura um padrão em um texto
	`sed`	encontra e substitui padrões em um texto
	`sort`	ordena dados numéricos
Programas Científicos	`km`	agrupa um conjunto de pontos
	`gf`	executa um filtro gaussiano em uma imagem 2D
	`fn`	calcula os números amigáveis em um intervalo de intervalo

programas para entender o que desempenho esperado ao executar aplicativos de sistema típicos em Nanvix.

(iii) *Programas Científicos* são compostos por núcleos de suítes de benchmark que foram portadas para Nanvix. Esses programas usam o bibliotecas de programação distribuída e paralela que estão disponíveis em nosso OS (lwMPI e PThreads), e nos permitiu determinar o desempenho que Nanvix pode entregar a aplicativos específicos de domínio.

A tabela 12 mostra as principais características de todos os programas que consideramos, e nas próximas seções, apresentamos cada um deles. Todos os programas estão disponíveis em <https://github.com/nanvix>.

B.5.2 Plataforma Experimental

Entre as arquiteturas suportadas por Nanvix, escolhemos Kalray MPPA-256 como o plataforma experimental neste trabalho. Este é um processador LW Manycore comercial que apresenta a maioria das características discutido em Seção 2.1.

O processador Kalray MPPA-256 (Figura 56) apresenta 272 núcleos de uso geral e 16 núcleos de firmware, chamados PE e RM, respectivamente. O processador é construído com tecnologia CMOS de 28 nm e funciona a 400 MHz. Todos os núcleos implementam um conjunto de instruções proprietárias de 64 bits, apresentar um pipeline VLIW de 5 questões, 8 kB I-Cache e D-Cache, e apresentam um MMU gerenciado por software.

Os 288 núcleos de Kalray MPPA-256 são agrupados em 16 Compute Clusters, que são destinados para computação, e 4 I/O Clusters, que são projetados para fornecer conectividade com periféricos. Cada Compute Cluster agrupa 16 PE, um RM, dois interfaces NoC e 2 MB de SRAM local (128 kB por núcleo). Nesses clusters, a coerência do cache de hardware não é suportada. Em contraste, I/O Clusters tem 4 RM, 8 NoC interfaces e 4 MB de SRAM (1 MB por núcleo). Dois desses clusters estão conectados a um DRAM controlador, e os outros dois estão conectados a PCI e Ethernet controladores. Compute Clusters não estão ligados a uma memória global e todos eles têm espaços de endereços privados. Assim, Compute Clusters têm que trocar o hardware mensagens por um de duas NoC torus 2-D intercaladas, para realizar comunicações: (i) a C-NoC que apresenta baixa width de banda e se destina a pequenas transferências de dados; e (ii) a D-NoC que apresenta alta width de banda e é dedicado a transferências de dados densos.

No que diz respeito ao desenvolvimento de software em Kalray MPPA-256, este processador é fornecido com uma versão corrigida do GCC 4.9.4 e Binutils 2.11.0. Nenhum OS é fornecido por o fornecedor e os engenheiros de software devem contar com um ambiente de execução proprietário não compatível para escrever seu aplicativos (DINECHIN et al., 2013), se não estiver usando Nanvix. Além disso, em relação à implementação do kernel OS, os engenheiros de sistema são obrigados a confiar em um hipervisor proprietário da Kalray. Este hipervisor é executado no núcleos de firmware do processador e intermediários de todas as operações de baixo nível. Digno de nota, o hipervisor Kalray não pode ser alterado nem configurado e, portanto, impõe desafios adicionais na construção de OS para este processador LW Manycore.

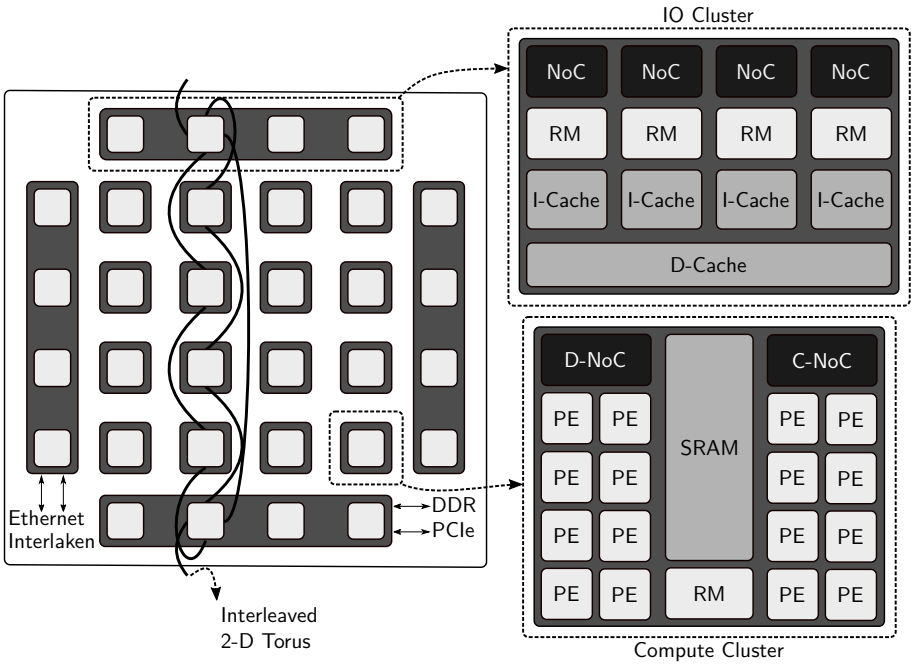


Figure 56 – Visão geral da arquitetura do processador MPPA-256 LW Manycore.

B.5.3 Projeto Experimental

Contamos com abordagens quantitativas e qualitativas para responder às perguntas de avaliação que almejamos (Seção 5.1). Para avaliando a programabilidade e a portabilidade, realizamos uma discussão que revela como esses aspectos são ativados por Nanvix. Em contraste, para avaliar o desempenho de Nanvix executamos os programas que detalhamos anteriormente (Seção 5.2) no processador Kalray MPPA-256 LW Manycore (Seção 5.3). Digno de nota, ao discutir resultados experimentais de utilitários de sistema e programas científicos, estamos efetivamente mostrando que o Nanvix oferece programação (Q-1) e portabilidade (Q-1) para LW Manycore.

Na avaliação experimental, reunimos medidas de tempo usando hardware contadores de desempenho para permitir o monitoramento com o mínimo de interferência. Por outro lado, para recuperar as medições de consumo de energia, contamos com um dispositivo que está externamente conectado à placa do processador. Esse dispositivo mede a dissipação de energia na placa e compreende estatísticas para todos os núcleos, NoC e outros recursos no chip. Além disso, para cada experimento, realizamos 10 testes para eliminar os efeitos indesejados do aquecimento, e, em seguida, 30 tentativas para coletar os resultados foram executadas. Todos os resultados que são apresentados neste trabalho são baseados em um limite de intervalo de confiança de 95 %.

B.6 Resultados Experimentais

Neste capítulo, apresentamos os resultados de nossa avaliação. Nós primeiro discutimos resultados da avaliação específica que realizamos para subsistemas de Nanvix, e então passamos nossa análise para os resultados de uma avaliação mais ampla que engloba a interação entre subsistemas, bem como usuário e bibliotecas do sistema.

B.6.1 Programabilidade e Portabilidade

Nanvix expõe três primitivas principais que podem ser usadas para o processo comunicação e sincronização: (i) *syncs* permite que os processos esperem por sinais e, assim, sin-

```
1  void send_message(int dest, int rxbuf, char (*msg)[128])
2  {
3    char path[64];
4    sprintf(path, "/mppa/channel/128:%d/%d:%d", rxbuf, dest, rxbuf);
5    int ch = mppa_open(path, O_WRONLY);
6    mppa_write(ch, msg, 128);
7    mmppa_close(ch)
8  }
```

Snippet B.1 – Envio de mensagens de tamanho fixo em MPPA usando as bibliotecas de metal bar da Kalray.

cronizem um com o outro; (ii) *mailboxes* de correio permitem que processos troquem mensagens de tamanho fixo (iii) *portal* permite que processos troquem quantidades arbitrárias de dados. No geral, essas primitivas melhoram a programabilidade em vários pontos. Primeiro, eles não exigem que o programador saiba em qual cluster um determinado processo está correndo. Em segundo lugar, eles lidam com o controle do fluxo de comunicação. Terceiro, eles lidam com a alocação de recursos de hardware de baixo nível. Finalmente, eles permitem o multiplexação de interfaces NoC.

Snippets de código B.1 e B.2 ilustram algumas dessas melhorias, mostrando como a troca de mensagens de tamanho fixo pode ser alcançada em Nanvix, em contraste com as bibliotecas do fornecedor enviadas com o Kalray MPPA-256. Ao usar a última biblioteca B.1, o seguinte etapas estão envolvidas. Primeiro, o caminho para o endpoint remoto deve ser construída em conformidade (linha 4). Isso especifica qual hardware buffer do transmissor a ser usado (`rxbuf`)), o tamanho das mensagens que será enviado por e o ID físico do destino cacho. Uma vez que esta configuração é feita, um canal de comunicação para o endpoint é aberto (linha 5), a mensagem é enviada (linha 6) e o o canal de comunicação final está fechado (linha 7). Digno de nota, neste fluxo o engenheiro de software tem que lidar com baixo nível aspectos do hardware, ao configurer o terminal. Em contraste, ao usar o sistema de comunicação de Nanvix (Código Snippet B.2) três etapas mais simples são obrigatório: (i) abre uma caixa de correio para o processo de destino, fornecendo seu número de identificação (linha 3); (ii) escrever a mensagem através da caixa postal (linha 4); e (iii) fecha a caixa de correio (linha 5).

No que diz respeito à portabilidade, as primitivas expostas permitem a implementação de abstrações de sincronização/comunicação POSIX como segue:

- Os sinais são implementados em cima das *mailboxes* de correio. Nesse contexto, as mensagens transportam informações de um sinal em questão, como seu número.

- Os semáforos são implementados em cima de *mailboxes* de correio e *syncs*. O primeiro é usado para transmitir valores de incrementos/decrementos, enquanto o último é usado para enviar notificações de ativação.

- As filas de mensagens são implementadas em cima das *mailboxes* de correio. Nesse caso, as mensagens encapsulam a carga útil que está sendo transmitida, bem como a prioridade de uma mensagem.

- Pipes são implementados usando *portals* e *sync*. O primeiro é usado para transferir dados, enquanto o último é usado para enviar/esperar ativar notificações.

Nesta seção, avaliamos o sistema de memória do Nanvix. Primeiro, nós discutimos como melhora a programação e portabilidade em LW Manycore processadores através do DPS (Seção 6.2.1). Em seguida, descobrimos o desempenho dos acessos à memória, bem como analisamos a latência para invalidar e buscar páginas (Seção 6.2.2).

```
1  void send(pid_t dest, char *msg, size_t n)
2  {
3    int outbox = kmailbox_open(dest);
4    kmailbox_write(outbox, msg, n);
5    kmailbox_close(outbox)
6  }
```

Snippet B.2 – Enviando mensagens de tamanho fixo em MPPA usando Nanvix.

Lembre-se de que processadores LW Manycore têm importantes restrições relativas ao seu sistema de memória. Primeiro, eles apresentam um arquitetura de memória distribuída: os módulos de memória são fisicamente espalhados pelo processador, o espaço de endereço não é único, e a memória não é acessível globalmente por todos os núcleos. Acesso a a memória remota só é possível com a comunicação de passagem de mensagem por meio de uma rede no chip. Em segundo lugar, tempos de acesso aos módulos de memória não são uniformes. É mais rápido acessar um banco de memória local do que dados de acesso (via NoC) que são armazenados em um módulo de memória remoto. Finalmente, a quantidade de memória no chip é muito limitada. Para Por exemplo, no processador Kalray MPPA-256, as memórias no chip são de 2 a 4 MB grande. Em resumo, essas características arquitetônicas impõem desafios na programação e portabilidade do software. Nós discutiremos mais como nosso sistema de memória supera esses problemas nos próximos parágrafos, examinando dois exemplos de casos de uso representativos.

No Snippet B.3, apresentamos alguns códigos C que aloca dinamicamente uma grande área de memória (1 GB) usando o padrão alocador de memória da biblioteca C (linha 2), manipula esta memória área (linhas 4 e 5) e libera (linha 7). Este exemplo funcionaria na maioria dos sistemas modernos que possuem memória virtual suficiente. No entanto, OSes for LW Manycore atualmente falta memória rica em nível de sistema gerenciamento e nenhum desses processadores tem o suficiente físico no chip memória para cumprir esta operação. Como consequência, apenas programas que têm pegadas de memória pequenas o suficiente para caber nas memórias físicas de um LW Manycore cluster pode ser portado sem introdução mudanças significativas em seu código-fonte. Nanvix possibilita que esse exemplo de caso de uso execute sem nenhuma modificação no Kalray MPPA-256.

No Snippet B.4, apresentamos um fragmento de um C programa que retrata o básico

```
1  size_t n = (1 << 30);          /* 1 GB                          */
2  char *mem = malloc(n);         /* Allocate a memory area.       */
3
4  for (size_t i = 0; i < n; i++) /* Manipulate the memory area. */
5    mem[i] = 0xff;
6
7  free(mem);                     /* Free the memory area.         */
```

Snippet B.3 – Caso de uso para expansão de endereço.

```
1  /* Allocate some shared memory. */
2  int fd = shm_open("shared-memory", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
3  ftruncate(fd, 4096);
4  char *shmem = mmap(NULL, n, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)
5
6  /* Write to shared memory. */
7  for (size_t i = 0; i < 4096; i++)
8    shmem[i] = 0xff;
9
10 munmap(shm, size);
11 shm_unlink(fd);
```

Snippet B.4 – Caso de uso para compartilhamento de espaço de endereço.

para o compartilhamento do espaço de endereço entre dois processos. Funciona da seguinte maneira. Primeiro, ele aloca uma memória compartilhada de 4 kB região, que é digitada como legível e gravável por programas do mesmo usuário (linha 2). Em seguida, ele anexa esta região de memória compartilhada ao espaço de endereço do programa de chamada (linha 4). Em seguida, ele grava nesta memória compartilhada região (linhas 7 e 8). Finalmente, ele libera esta região de memória compartilhada (linhas 10 e 11). Infelizmente, os OSes atuais para LW Manycore não oferecem suporte a este fluxo de execução. A única maneira que os programas têm para comunicarrem-se é através da passagem de mensagens. Isso muito altera o design de programas e, portanto, compromete a capacidade de programação. Nanvix oferece suporte a regiões de memória compartilhada e permite que processos os usem para comunicar.

B.6.2 Desempenho de Utilitários de Sistema

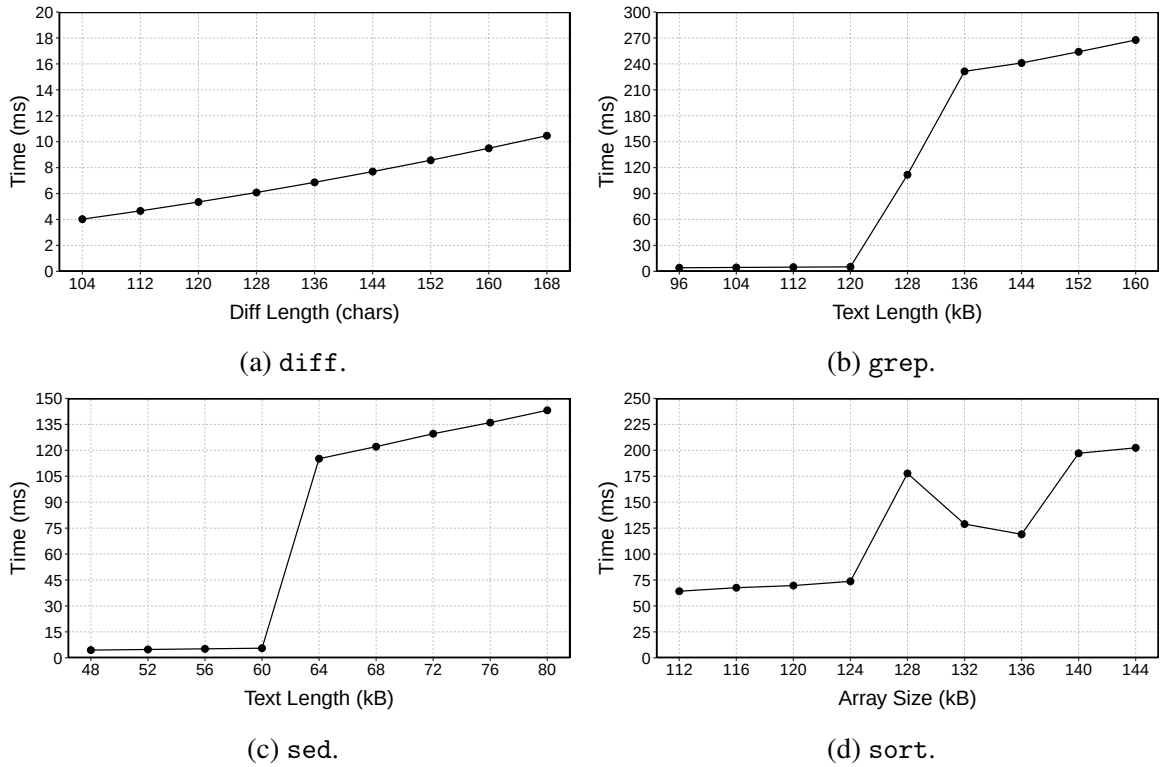
Figura 57 apresenta o tempo de execução dos quatro utilitários unix considerados, ao variar o tamanho da entrada de trabalho definir. Vale ressaltar que cada utilitário foi executado exclusivamente em Nanvix. Em outras palavras, além dos serviços OS, não havia nenhum outro nível de usuário aplicativo em execução. Portanto, não há interferência externa, e assim, os resultados relatados refletem o desempenho do limite superior.

Ao observar os resultados para o utilitário `diff` (Figura 57a), notamos um aumento linear em tempo de execução ao aumentar o tamanho do conjunto de trabalho. O motivo desse comportamento é explicado pelas características do aplicativo. O algoritmo de subsequência menos comum processa a tabela de memoização em diante e cada novo acesso a esta estrutura de dados incorre em uma falta página . Consequentemente, quanto maior o conjunto de dados, maior é o número de falhas de página, resultando em tempos de execução mais longos. Uma vez que o tempo para buscar páginas é constante, o tempo de execução linear.

Figura 57b e Figura 57c apresenta os tempos de execução dos programas `grep` e `sed`, respectivamente. Em ambos casos, observamos um comportamento de desempenho escalonado que leva a conclusões complementares para `diff`. Esses dois utilitários (i.e., grep e `sed`) apresentam padrões de acesso que se beneficiam muito do ambiente localidade. Por este motivo, quando o conjunto de trabalho é pequeno o suficiente para caber o cache de página

do nosso sistema de memória, observamos pequenos tempos de execução. No entanto, assim que o conjunto de trabalho não couber inteiramente no cache, nós observamos que o tempo de execução aumenta significativamente. Digno de nota, quando este última situação acontece, os aumentos no tempo de execução são lineares, à medida que observado para `diff`. Por fim, é importante destacar que os resultados dessas experiências mostram a importância do cache de página para o nosso sistema. Graças a isso, os aplicativos podem atingir até $10\times$ melhor desempenho do que acessando diretamente as memórias remotas.

Finalmente, apresentamos na Figura 57d os resultados obtidos com o utilitário `sort`. Neste experimento, notamos um comportamento mais irregular com três fases principais. Primeiro, quando o tamanho do conjunto de trabalho é pequeno o suficiente para caber no cache da página (ou seja, até 128 kB), tempo de execução aumenta ligeiramente. Então, quando o tamanho do conjunto de trabalho varia de 100% a cerca de 105% do tamanho do cache da página, o desempenho apresenta uma grande variação. Terceiro, quando o conjunto de trabalho é grande o suficiente, observamos aumentos lineares no tempo de execução a partir de uma linha de base mais alta. Os resultados para a primeira e a terceira fases estão alinhadas aos comportamentos observados na outros utilitários. Em contraste, o raciocínio para o comportamento observado em a segunda fase vem do padrão de acesso aleatório do próprio programa. Neste cenário, a política de substituição de página tem um desempenho tão ruim quanto um padrão patológico (pico em 128 kB) ou tão bom quanto quando a localidade é explorado (vazio em 136 kB).



(a) `diff`. (b) `grep`.

(c) `sed`. (d) `sort`.

Figure 57 – Desempenho para utilitários do sistema.

Figura 58 retrata o desempenho de Nanvix quando está sujeito a cargas pesadas. Neste experimento, implantamos vários instâncias dos quatro utilitários Unix ao mesmo tempo. As imagens do sistema eram gerados aleatoriamente, variando o número de processos no sistema. Mais importante, este experimento mostra que Nanvix ativa o recurso multiplexação, isolamento e compartilhamento em processadores LW Manycore.

De forma geral, notamos que o tempo de execução apresenta alguma irregularidade, variando de 12 ms a até 30 ms. No entanto, este comportamento era esperado, uma vez que as imagens do sistema foram geradas aleatoriamente e o as características de cada utilidade variam de uma para outra. No entanto, o o tempo de execução não aumenta significativamente ao passar de um cenário leve, em que um único utilitário é implantado, para um um pesado, no qual dezesseis utilitários são executados simultaneamente em Nanvix. O desempenho diminui sublinearmente com a carga no sistema: o desempenho geral degrada em apenas 38% ao aumentar carga do sistema por um fator de 16×.

B.7 Conclusões

LW Manycore foram introduzidos para fornecer escalabilidade de desempenho com baixo consumo de energia consumo. Para lidar com o primeiro aspecto, esses processadores contam com sistemas específicos características arquitetônicas, como uma arquitetura de memória distribuída e um rich NoC. Para atingir o baixo consumo de energia, esses muitos pontos são construídos com núcleos simples de baixo consumo MIMD, eles têm um sistema de memória baseado em SPM e eles explorar a heterogeneidade apresentando núcleos com recursos diferentes. Algum exemplos de sucesso da indústria desses processadores são o Kalray MPPA-256, o PULP e o Sunway SW26010.

Embora este conjunto único de recursos arquitetônicos conceda a LW Manycore desempenho escalabilidade e eficiência energética, eles também apresentam vários desafios em desenvolvimento e implantação de software. Primeiro, a *integração do circuito de alta densidade* transforma o silício escuro em realidade. Em segundo lugar, a *arquitetura de memória distribuída*

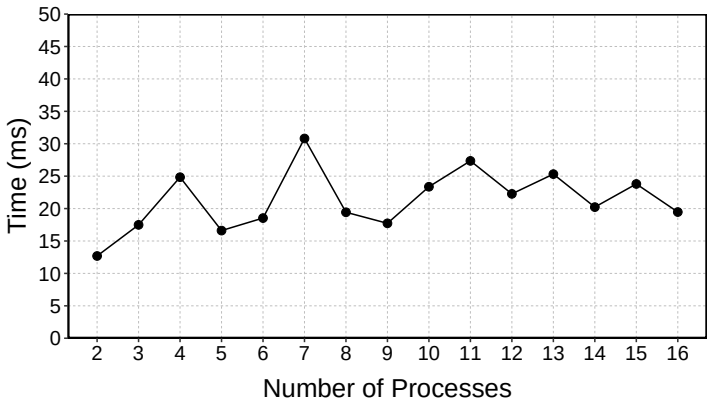


Figure 58 – Desempenho do Nanvix sob carga pesada.

requer que os dados sejam explicitamente buscado/descarregado de memórias remotas para memórias locais. Terceiro, o *pequena quantidade de memória on-chip* força o software para particionar seu conjunto de dados de trabalho em blocos e decidir qual deles deve ser mantido local e deve ser descarregado para a memória remota. Quarto, a *interconexão on-chip* convida engenheiros de software a abraçar um modelo de programação de passagem de mensagens. Finalmente, a *heterogeneidade no chip* torna a implantação de aplicações complexas.

Para resolver os problemas mencionados, existem duas abordagens principais: runtime systems e OSes. O primeiro visa expor um ambiente de programação para atender às necessidades de uma classe específica de aplicativos (HO et al., 2015; Mohr; Tradowsky, 2017). Em contraste, a segunda abordagem anseia por superar as complexidades do hardware em um sentido mais amplo, expondo abstrações ricas e interfaces de programação, como bem como lidar com a alocação, compartilhamento e multiplexação de recursos.

No contexto de processadores LW Manycore, infelizmente hoje, nenhum dos dois abordagens anteriores fornecem uma solução completa para lidar com a programabilidade e desafios de portabilidade. Por um lado, runtime systems inerentemente carece de cobrir todos os problemas que nós declarado anteriormente. Por exemplo, alguns runtime systems fornecem abstrações abordam a comunicação no chip, mas negligenciam os desafios que preocupam a arquitetura de memória distribuída. Por outro lado, os sistemas existentes não são projetados para lidar com todos os restrições arquitetônicas dos processadores LW Manycore. Em resumo, não é possível implantar esses OSes em LW Manycore sem um redesenho completo e/ou significativo mudanças em seu código-fonte. No geral, identificamos o seguinte problema sem resposta no Sistemas Operacionais de última geração, que pretendemos abordar nesta tese:

> *Como projetar um sistema operacional de modo a lidar com a programabilidade e problemas de portabilidade em processadores manycore leves?*

Esta tese propõe um SO distribuído que avança o recurso. gerenciamento em processadores LW Manycore e especificamente enfatiza o seguinte pontos:

- *Programação*: para fornecer abstrações de sistema ricas para LW Manycore, como processo, arquivos, memória virtual e comunicação primitiva; bem como para permitir o compartilhamento de recursos, multiplexação e isolamento nessas arquiteturas emergentes.

- *Portabilidade*: para expor interfaces de programação padrão para para permitir que o software existente seja transferido para processadores LW Manycore.

Nas seções a seguir, relembramos as contribuições científicas e técnicas que desdobram a partir desses pontos. Além disso, também destacamos a parte acadêmica contribuição desta tese na orientação de alunos e estagiários.

B.7.1 Contribuições Científicas

A principal contribuição desta tese - *um romance distribuído OS que avança o gerenciamento de recursos em processadores LW Manycore* - pode ser desdobrada nas seguintes contribuições científicas específicas:

- Um abrangente HAL para LW Manycore que une as diferenças arquitetônicas em várias arquiteturas e lida com os principais problemas que são frequentemente encontrados ao projetar um OSes para esses processadores. Com esta camada de abstração, o desenvolvimento e implantação de um OS completo se torna mais fácil não apenas para um LW Manycore específico, mas também permite a portabilidade de um OS específico em vários destes processadores emergentes.

- Uma abordagem de gerenciamento de memória rica que se baseia em DPS. Isto é um nova solução que criamos para gerenciar a memória de um processador LW Manycore e funciona da seguinte maneira. As memórias locais de um processador LW Manycore são considerados caches de página que podem armazenar dados de quaisquer processos. No geral, esta solução permite o acesso transparente aos dados e manipulação, bem como compartilhamento de dados e coerência em LW Manycore;

- Uma facilidade de comunicação leve que gerencia o on-chip interconexão e exposições primitivas com multiplexação de canal.

B.7.2 Produção Técnica

Do ponto de vista técnico, esta tese também apresenta um importante contribuição, que é Nanvix. Esta é uma implementação concreta de um OS para LW Manycore processador que apresenta o supramencionado avanços científicos. Nanvix suporta múltiplas arquiteturas (i.e., x86, OpenRISC, ARMv8 e RISC-V), roda em silício LW Manycore processadores, expõe abstrações ricas e interfaces de programação de alto nível.

Nanvix apresenta um design multikernel que é estruturado em várias instâncias superiores de um microkernel. Nós achamos esta organização porque acreditamos que ela atende melhor a características de LW Manycore. Por um lado, o abordagem multikernel lida com a arquitetura de memória distribuída de esses processadores. Por outro lado, a estrutura baseada em microkernel permite uma implementação leve e modular de serviços de sistema, que é um requisito fundamental dada a escassez de recursos de hardware. Ele tem um layout de quatro camadas que apresentaremos a seguir.

O HAL expõe uma abstração abrangente do subjacente hardware (PENNA; FRANCIS; SOUTO, 2019). É implementado como uma biblioteca baremetal, e envia rotinas e estruturas de dados comuns para manipular estruturas de hardware, como núcleos, MMU e NoC. Em resumo, o HAL permite a portabilidade de nosso OS em vários processadores LW Manycore, como Kalray MPPA-256, OpTiMSoC e PULP.

O microkernel fornece compartilhamento de recursos, multiplexação, isolamento e proteção, dentro de um cluster de um LW Manycore (PENNA et al., 2019). Corre em modo privilegiado e abstrações e primitivas mínimas expostas, como threads, espaço de endereço virtual e comunicação entre processos mecanismos. Nosso microkernel apresenta um design assimétrico, o que significa que ele é executado exclusivamente em um núcleo do cluster e deixa o restante núcleos para uso geral. Este projeto atenua a interferência do kernel no software de nível de usuário e, portanto, melhora o desempenho.

Os servidores do sistema são implantados no topo do microkernel e implementam serviços comumente encontrados em um OS, como agendamento de processos, memória mapeamentos e alocação de arquivos. Por sua vez, os subsistemas são implementados tendo um conjunto desses servidores de sistema que funcionam juntos, de forma distribuída moda. Finalmente, no topo dos servidores do sistema, as bibliotecas de tempo de execução são descartadas. Estas bibliotecas são vinculadas aos aplicativos do usuário e fornecem uma interface padrão para software de nível de usuário interagir com o OS.

APPENDIX C – EXTENDED ABSTRACT IN FRENCH

C.1 Introduction

Les communautés scientifiques et industrielles recherchent et développer des solutions pour répondre aux exigences de performance toujours croissantes d'applications logicielles. Ces solutions sont souvent adaptées complexités particulières d'une application et d'une plate-forme cibles, et elles vont du niveau logiciel et du support du compilateur (KAMIL et al., 2010), à le système d'exécution (BROQUEDIS et al., 2010) et le matériel sous-jacent architecture (DINECHIN et al., 2013). Ainsi, des techniques spécialisées peuvent être appliqués et ainsi obtenir des performances de pointe.

C.1.1 Motivation

Plus précisément sur les efforts au niveau matériel, LW Manycore processeurs (ISHII et al., 2017) a émergé comme une alternative prometteuse pour offrir une évolutivité des performances avec Basse consommation énergétique. Pour activer ces fonctionnalités, ces processeurs s'appuient sur un ensemble sélectionné de caractéristique architecturale qui introduisent de multiples défis dans la programmabilité et la portabilité des logiciels. Premièrement, le *intégration de circuit haute densité* transforme le silicium noir en réalité (HAGHBAYAN et al., 2017). Deuxièmement, la *architecture de mémoire distribuée* conduit à une conception de logiciels (CASTRO et al., 2016). Troisièmement, le *on-chip interconnect* invite les ingénieurs à adopter une modèle de programmation de passage de message et ainsi travailler sur un refactor important sur leur logiciel (SOUZA et al., 2017). Enfin, la *hétérogénéité sur puce* rend le déploiement de applications dans LW Manycore complex (BARBALACE et al., 2015).

C.1.2 Problème

Il existe deux approches principales pour aborder la programmabilité logicielle et défis de portabilité dans une architecture : baremetal runtime systems et OSes. La première vise à exposer un environnement de programmation de haut niveau pour répondre aux besoins d'une architecture spécifique (HO et al., 2015; Mohr; Tradowsky, 2017). Dans en revanche, la deuxième approche aspire à combler les subtilités du matériel dans un sens plus large, en exposant de riches abstractions et programmations interfaces, ainsi que la gestion de l'allocation des ressources, le partage et multiplexage.

Dans le cadre des processeurs LW Manycore, malheureusement à partir de aujourd'hui, aucune des deux approches précédentes n'offre une solution complète à relever les défis de la programmabilité et de la portabilité. D'une part, l'existant baremetal runtime systems qui visent ces processeurs émergents (VARGHESE et al., 2014; RICHIE; ROSS; INFANTOLINO, 2017) manque intrinsèquement de couvrir tous les problèmes que nous avons énoncés dans

précédemment (Section 1.1). De l'autre part, les OSes existants ne font pas face aux contraintes architecturales complexités de LW Manycore comme suit.

a) À une extrémité du spectre, centralisé OSes intrinsèquement faire face à des obstacles importants :

- Ils ont des problèmes d'évolutivité (WENTZLAFF; AGARWAL, 2009).
- Ils ne font pas face à des matériel (BARBALACE et al., 2015).
- Ils ne sont pas réalisables pour le contexte en raison de la faible quantité de mémoire sur puce.

b) À l'autre extrémité du spectre, les OSes distribuées manquent contraintes architecturales des LW Manycore processeurs (PENNA et al., 2021), et donc il n'est pas possible de déployer ces OSes dans les architectures sans refonte complète et/ou changements significatifs leur code source. La justification réside dans les points suivants :

- Ils ne reposent pas sur une vue décentralisée du matériel. Par exemple, ils supposent que les ressources matérielles sont visibles pour tous les cœurs du processeur sous-jacent.
- Bien qu'ils présentent une organisation logicielle distribuée, ils ne gèrent pas les multiples espaces d'adressage physiques. Ils s'appuyer sur des zones de mémoire partagée pour transférer des messages logiciels et déplacer des données d'un cœur à un autre.
- Ils n'ont pas comme contrainte principale leur empreinte mémoire, et ne supportent donc pas intrinsèquement les petites mémoires sur puce de LW Manycore.
- Ils négligent les riches interconnexions sur puce et n'exploitent pas des fonctionnalités supplémentaires qui sont souvent disponibles dans LW Manycore, telles que message asynchrone passant par DMA et routage de message.

Sur la base de ces observations, nous formulons le problème suivant, qui reste sans réponse par l'état de l'art en matière de systèmes d'exploitation :

Comment concevoir un système d'exploitation pour répondre problèmes de programmabilité et de portabilité dans le manycore léger processeurs ?

C.1.3 Contributions

La principale contribution de cette thèse est un OS distribué qui fait progresser la gestion des ressources dans les processeurs LW Manycore. D'un côté, du point de vue scientifique, cette contribution principale peut être s'est déroulée dans les contributions spécifiques suivantes :

- Un HAL complet pour LW Manycore (PENNA; FRANCIS; SOUTO, 2019) cette: (i) comble les différences architecturales entre plusieurs architectures; et (ii) faire face aux problèmes clés qui sont souvent rencontrés lors de la conception un OS pour ces processeurs. Avec cette couche d'abstraction, le développement et le déploiement d'un OS complet devient plus facile non seulement pour un LW Manycore particulier, mais aussi la portabilité d'un OS lui-même sur plusieurs de ces processeurs.

- Une approche de gestion de mémoire riche basée sur un DPS. C'est une nouvelle solution que nous avons conçue pour gérer la mémoire d'un processeur LW Manycore et cela fonctionne comme suit. Les mémoires locales d'un processeur LW Manycore sont considérés comme des caches de pages pouvant stocker des données de n'importe quel processus. Basé sur ce, le OS gère ce cache de page en : (i) plaçant ces pages qui sont fortement utilisées par un processus dans les mémoires locales ; et (ii) échangeant les pages qui ne sont pas utilisées par un processus pour souvenirs lointains. Globalement, cette solution permet un accès transparent aux données et manipulation, ainsi que le partage et la cohérence des données dans LW Manycore. Nous croyons que ces problèmes sont liés de manière importante à la programmabilité et défis de portabilité que nous visons à surmonter.

- Une installation de communication légère qui gère le interconnecter et expose des primitives riches (PENNA et al., 2021). Globalement, cette fonction expose trois abstractions de communication : (i) les synchronisations, par permettre à un processus de signaler et de déverrouiller un autre processus à distance ; (ii) boîtes aux lettres, pour l'envoi de messages de taille fixe avec une faible latence ; et (iii) portails, pour gérer des transferts de données denses avec une bande passante élevée. Cette l'infrastructure de communication permet effectivement l'efficace implémentation de notre OS.

D'autre part, nous soulignons que cette thèse introduit également un contribution technique à la communauté :

- Nanvix, une implémentation concrète d'un OS qui présente le les avancées scientifiques susmentionnées pour les processeurs LW Manycore. Nanvix prend en charge plusieurs architectures (i.e., k1b, x86, OpenRISC, ARMv8 et RISC-V), fonctionne sur des processeurs au silicium LW Manycore, ainsi qu'il expose de riches abstractions et interfaces de programmation de haut niveau.

C.2 Background

Dans ce chapitre, nous abordons les fondements de notre thèse. Premièrement, nous examinez en profondeur les architectures multicœurs à puce unique et leurs principales caractéristiques. Ensuite, nous introduisons runtime systems. Puis, nous discutons de OSes et des brèves sur les systèmes distribués. Finalement, nous concluons ce chapitre en positionnant notre travail sur ce terrain.

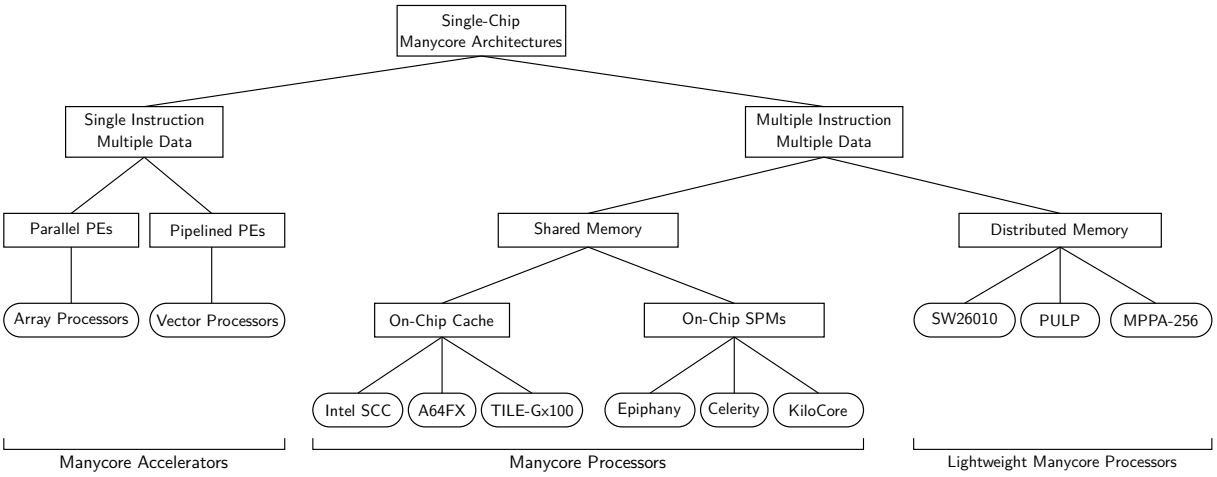


Figure 59 – Une taxonomie étendue pour les architectures multicœurs à puce unique.

C.2.1 Processeurs à puce unique Manycore

Des processeurs parallèles ont été introduits pour surmonter une évolutivité barrière au niveau matériel connue sous le nom de *Power Mur* (CASTRO et al., 2016). Au lieu de monter l'instruction débit dans un seul CPU, l'idée alternative était d'augmenter le nombre réel de cœurs dans un CPU. De cette façon, l'exécution multiple les flux pourraient s'exécuter en même temps et le parallélisme au niveau des threads pourrait être exploité pour atteindre une évolutivité des performances dans un application. Dans ce contexte, une catégorisation bien connue a été proposé par Flynn (FLYNN, 1972), dans lequel les architectures peuvent être regroupés selon leurs instructions et flux de données:

- SISD: un seul processeur exécute un seul flux d'instructions pour fonctionner sur des données stockées dans une seule mémoire.
- SIMD: une seule instruction machine contrôle la simultanéité l'exécution d'un certain nombre d'éléments de traitement sur une base synchrone. Chaque élément de traitement a une mémoire de données associée, de sorte que chaque l'instruction est exécutée sur un ensemble différent de données par les différents processeurs.
- MISD: une séquence de données est envoyée à un ensemble de processeurs, chacun dont exécute une séquence d'instructions différente.
- MIMD : un ensemble de processeurs exécutent simultanément différents séquences d'instructions sur différents ensembles de données.

Bien qu'il s'agisse d'une taxonomie solide, avec l'augmentation récente de noyaux dans les architectures, on pourrait aussi argumenter sur des éléments architecturaux supplémentaires caractéristiques qui peuvent être prises en compte pour catégoriser plusieurs cœurs à puce unique processeurs. Cette classification étendue est décrite dans Figure 1 et cela ajoute à Aspects de l'approche de Flynn concernant le système de mémoire des monopuces architectures à

plusieurs cœurs. Sur la base de cette caractéristique supplémentaire, nous mettre en évidence trois organisations pertinentes pour la compréhension de cette thèse: *manycore accélérateurs*, *manycore processeurs* et *LW Manycore processeurs*.

Les accélérateurs multicœurs ont été la première classe de processeurs multicœurs à puce unique à surgir. Ils comprennent les architectures qui sont rétrécies pour des charges de travail, telles que le traitement vectoriel, la vision par ordinateur et intelligence. Dans l'ensemble, les accélérateurs manycore comportent souvent une organisation SIMD et pour cette raison sont souvent couplés à un processeur à usage général pour construire un système hétérogène. Parmi de nombreux accélérateurs de cœur, nous soulignons deux sous-classes qui diffèrent dans la façon dont les cœurs sont organisés: *array processors* et *processeurs vectoriels*.

Les processeurs manycore s'adressent au domaine HPC à usage général et sont souvent utilisés comme blocs de construction de base pour les superordinateurs. Ces processeurs disposent d'un partage système de mémoire dans lequel tous les cœurs peuvent lire/écrire de manière transparente en utilisant instructions de chargement/stockage. En fin de compte, les processeurs manycore ne nécessitent pas une grande changements de paradigme de programmation et sont donc actuellement la classe la plus connue parmi architectures multicœurs à puce unique. Quelques exemples de processeurs manycore sont Intel Single-Cloud Computer (HOWARD et al., 2011), Tilera TILE-Gx100 (RAMEY, 2011), Adapteva Epiphany (OLOFSSON; NORDSTROM; UL-ABDIN, 2014), Celerity (DAVIDSON et al., 2018), Fujitsy A64FX (YOSHIDA, 2018) et Manticore (ZARUBA; SCHUIKI; BENINI, 2020). Concernant spécifiquement les processeurs manycore, nous soulignons deux organisations qui sont souvent adoptées. Dans le premier, les noyaux sont physiquement regroupés en unités physiques appelées clusters (ou tuiles), qui sont entièrement interconnecté avec le système global de mémoire partagée via un NoC. Les cœurs d'un même cluster sont étroitement couplés à un contrôleur de mémoire local, un cache local et un routeur NoC. Les contrôleurs de mémoire permettent un accès transparent à la mémoire partagée globale en transférant les demandes de chargement/stockage via le NoC. De plus, ces contrôleurs de mémoire fournissent également un support de cohérence de cache, en soit en l'implémentant dans le matériel, soit en exposant des instructions spéciales qui permettent ceci au niveau logiciel. Dans une seconde organisation pour un processeur multicœur, les cœurs sont également regroupés en clusters interconnectés via un NoC. Cependant, le système de mémoire ne repose pas sur un schéma de mise en cache. Au lieu de cela, les cœurs du même cluster partagent l'accès à un SPM local module. De plus, l'espace d'adressage global est ce statiquement partitionné sur plusieurs SPM et contrôleurs de mémoire en avant accès mémoire distant au module SPM concerné.

LW Manycore les processeurs s'efforcent de fournir des performances élevées avec de l'énergie Efficacité. À cette fin, ils s'appuient principalement sur un ensemble distingué de caractéristiques architecturales. Dans l'ensemble, LW Manycore peut être vu comme des "clusters sur une puce" et présentent les principales caractéristiques suivantes:

- Ils sont conçus pour faire face aux charges de travail MIMD.

- Ils intègrent jusqu'à des milliers de cœurs basse consommation dans un seul ébrécher.
- Ils ont leurs noyaux disposés en groupes étroitement couplés appelés groupes.
- Ils présentent une architecture de mémoire distribuée avec plusieurs espaces d'adressage.
- Ils présentent une mémoire contrainte, avec de petits (quelques kB per core) et des mémoires locales privées.
- Ils s'appuient sur NoC pour une transmission de messages rapide et fiable la communication.
- Ils ont une configuration hétérogène en termes d'E/S et/ou capacités de calcul.

C.2.2 Environnements d'exécution

Runtime systems fournit des environnements de programmation riches pour une exécution particulière maquette. Pour ce faire, ils exposent des API spécifiques aux applications ; et ils peuvent interagir directement avec le matériel, OS et/ou le compilateur. Certains des exemples bien connus de runtime systems sont le crt0, le ART et le JRE. Ces systèmes permettent principalement un environnement de programmation à usage général pour applications, en exposant des routines et des abstractions pour manipuler ressources telles que les flux de données, les signaux/exceptions logiciels et la mémoire de tas. Néanmoins, dans le cadre de ce travail, nous discutons ensuite de runtime systems qui se concentre sur les processeurs manycore. Dans l'ensemble, l'existant les alternatives concernent principalement les cadres de programmation parallèles et distribués.

Les environnements de programmation parallèle exposent une infrastructure pour les applications pour créer, manipuler et synchroniser plusieurs flux d'exécution au sein d'un programme. De cette façon, une application peut exploiter à fond les performances qui est fourni par une architecture parallèle. Nous mettons en évidence trois environnements qui sont souvent disponibles pour les architectures multicœurs : PThreads, OpenMP (DAGUM; MENON, 1998) et OpenCL (MUNSHI, 2009). La première encapsule un flux d'exécution dans une abstraction appelée *thread*, et il expose un ensemble de routines qui permettent aux applications de manipuler des threads. L'environnement PThreads permet une flexibilité maximale en parallèle programmation et sont généralement disponibles sous forme de bibliothèque liée de manière statique. Ensuite, le runtime OpenMP se concentre sur un support transparent pour le fork-join modèle de programmation parallèle. À cette fin, le système d'exécution interagit avec l'infrastructure du compilateur pour permettre à l'ingénieur logiciel d'annoter le code source afin de créer des régions parallèles. Ensuite, lors de la construction du application à partir d'un code source annoté, le compilateur établit des liens de bas niveau routines pour générer et terminer des threads au début et à la fin d'un parallèle région, respectivement. Enfin, l'environnement OpenCL vise à parallèle programmation dans des architectures hétérogènes. Il définit une structure pour écrire des programmes où le noyau de l'application s'exécute dans le OpenCL périphérique,

tandis que la partie restante du code s'exécute dans l'hôte dispositif. Pour ce faire, OpenCL fonctionne avec l'infrastructure du compilateur pour générer du code pour les deux architectures, ainsi qu'il interagit avec le hardware/OS pour télécharger/décharger le binaire et les données depuis/vers les appareils.

Les environnements de programmation distribués permettent une exploitation complète de architectures distribuées en se concentrant sur la communication abstractions entre plusieurs nœuds de calcul dans une plate-forme. On note deux environnements de programmation distribués qui sont généralement rencontrés dans architectures multicœurs : MPI (GROPP et al., 1996) et PGAS (De Wael et al., 2015). Le premier expose un ensemble de routines qui permettent à des données arbitraires d'être envoyer/reçu en utilisant des abstractions appelées messages. Les messages peuvent avoir un destinataires uniques ou multiples et peuvent être envoyés/reçus de manière synchrone et/ou de manière asynchrone. D'autre part, PGAS permet une communication unilatérale en le maintien d'un espace d'adressage global qui est partitionné et réparti sur le plusieurs pairs de l'application. Les pairs peuvent lire/écrire des données à cette adresse l'espace à travers des fonctions spéciales put/get, et le système d'exécution garantit que la cohérence est maintenue.

C.2.3 Systèmes Opérationnels

Un Operating System (OS) est une couche logicielle qui se trouve au-dessus du matériel baremetal, gère les ressources informatiques et fournit des services communs pour les programmes. Sans OS, les ingénieurs devraient gérer les complexités matérielles par leur propre, augmentant ainsi la complexité du logiciel et le coût de développement. Un OS a deux rôles importants dans une plate-forme informatique:

(i) Ressource abstraite: étendre les fonctionnalités du matériel et exposer aux programmes utilisateurs des services et des API riches plus faciles à traiter avec.

(ii) Gérer les ressources : accès multiplex au matériel et aux logiciels ressources, garantissant la protection et l'intégrité des données, ainsi que l'application protocoles et politiques d'administration.

La figure 60 présente un schéma d'un OS. Il est composé de plusieurs composants qui interagissent les uns avec les autres pour fournir abstraction et gestion des ressources riches. Il est à noter qu'il existe de nombreuses façons de coupler ces composants. Par conséquent, dans les paragraphes suivants, nous nous concentrons sur discuter de leurs rôles, et en Section 2.3.2, nous donnons des idées sur la façon dont ils peuvent être serrés les uns contre les autres.

Le HAL est le composant de niveau le plus bas d'un OS. Il interagit directement avec le matériel et expose un API commun pour traiter le baremetal structures, telles que les vecteurs d'interruption et les tables MMU. Le rôle principal de ce composant est de masquer autant que possible toutes les subtilités matérielles, ainsi pour fournir une interface standard à travers

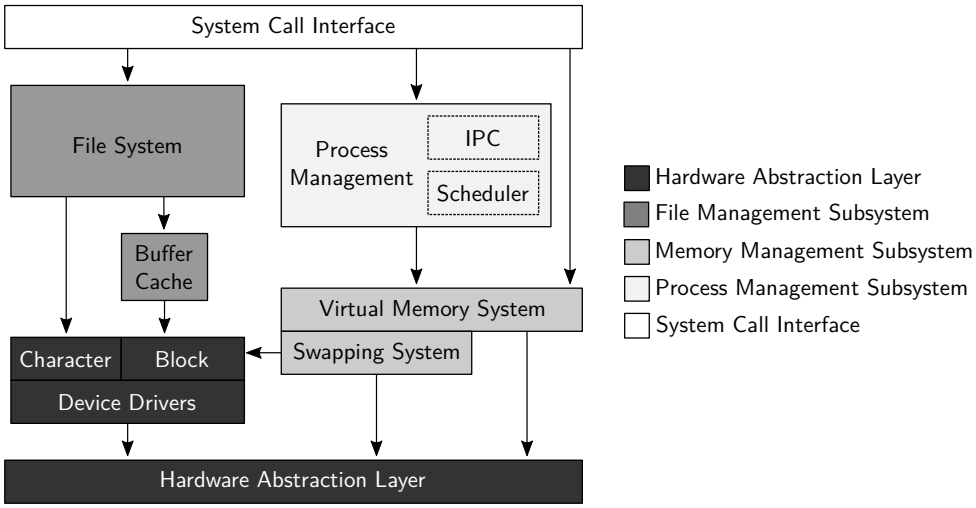


Figure 60 – Une architecture possible pour un OS.

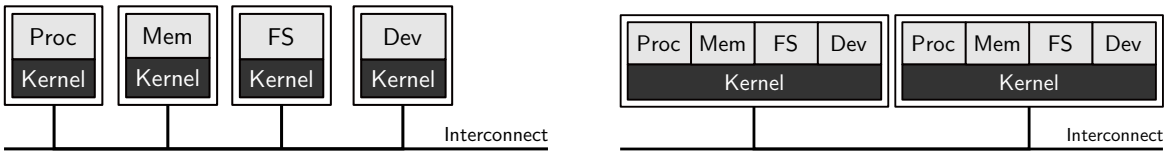
plusieurs architectures. Par ici, les composants OS sus-jacents deviennent moins complexes et plus faciles à porter d'un matériel à un autre.

Le MMS fournit une vue du système de mémoire sous-jacent. Il fait généralement donc en faisant travailler ensemble deux modules: le swapping et la mémoire virtuelle modules. La première traite de la pagination, en gardant en mémoire les pages qui sont plus fréquemment utilisés, et en remplaçant vers le stockage secondaire ceux qui sont ne pas. Le système de mémoire virtuelle, quant à lui, repose sur la pagination module pour activer des fonctionnalités avancées telles que les régions de mémoire partagée, à la demande chargement, copie paresseuse et épinglage de page. Nous discutons plus en détail de l'intérieur de a MMS dans la section 2.3.3.

Jusqu'à présent, nous avons peu discuté des problèmes spécifiques qui concernent le système conception d'architectures distribuées, ce qui est le cas de LW Manycore processeurs. Par conséquent, pour affiner notre discussion, dans cette section, nous couvrir le contexte concernant les systèmes distribués.

Dans les systèmes distribués, une question importante concerne la manière dont le système lui-même est structuré. Globalement, il existe deux approches orthogonales: architectures centralisées et décentralisées. Ces alternatives sont décrites dans Figure 15 et discuté ensuite.

Dans une architecture centralisée, le système est organisé comme un ensemble de processus spéciaux appelés serveurs, dont chacun fournir un ensemble de fonctionnalités différentes aux processus clients. Les serveurs peuvent soit mettre en œuvre des fonctionnalités entières ou spécifiques ; ainsi qu'eux peuvent interagir les uns avec les autres pour mettre en œuvre



(a) Centralisé organisation. (b) Organisation décentralisée.

Figure 61 – Architecture pour les systèmes d'exploitation distribués.

des opérations complexes. Cette L'approche exploite la distribution verticale et c'est la plus simple à mettre en œuvre. Cependant, il a deux problèmes, d'abord, avoir un service centralisé dans un serveur peut entraîner des problèmes d'évolutivité. Deuxièmement, si un serveur tombe en panne, le le service sous-jacent devient indisponible. La figure 61a décrit un architecture centralisée pour un Distributed OS. Dans cet exemple, le OS est pris en compte dans quatre serveurs, qui sont à leur tour répartis sur quatre nœuds différents. Chaque serveur implémente un sous-système différent et certains d'entre eux coopèrent.

Dans une architecture décentralisée, le système distribué est structuré comme un ensemble de serveurs, chacun mettant en œuvre l'ensemble complet des fonctionnalités. Chaque serveur fournit le même ensemble d'interfaces, mais sert des clients différents et potentiellement fonctionner sur des données différentes. Cette architecture exploite l'horizontale distribution et offre une meilleure évolutivité, Cependant, l'inconvénient réside fait que cette architecture est plus difficile à concevoir et à mettre en œuvre. La figure 61b présente un architecture décentralisée pour un Distributed OS. Dans cet exemple, chaque nœud fournit l'ensemble complet des services OS et gère différents processus clients. Dans contrairement à une organisation centralisée, les composants doivent moins communiquer fréquemment, uniquement pour conserver des structures de données partagées cohérentes.

C.3 Travaux Connexes

Il existe deux approches complémentaires qui visent à soutenir développement de logiciels dans des processeurs multicœurs monopuces : *Systèmes d'exécution* et *Systèmes d'exploitation (SE)*

Runtime systems interagit avec le matériel et/ou le compilateur pour exposer API que faciliter la programmation dans un domaine d'application spécifique. Plus précisément dans le cas des processeurs multicœurs à puce unique, runtime systems vise principalement à fournir une programmation parallèle et/ou distribuée de haute performance environnements. Dans ce contexte, nous notons pourquoi ces runtime systems ne couvrent pas le défis ouverts sur les processeurs LW Manycore :

- *Les bibliothèques spécifiques au fournisseur* sont conçues pour un l'architecture et, par nature, ne fournissent aucun support pour les logiciels portabilité.
- *Template Meta Programming Libraries* offrent la portabilité uniquement aux applications et plates-formes qu'ils ciblent, ce qui est limité.
- *MPI et PGAS Libraries* ne gèrent pas le petit quantité de mémoire sur puce ni ils permettent un accès transparent à la mémoire sur plusieurs espaces d'adressage physiques.

De plus, il est important de souligner que runtime systems manque intrinsèquement de fournir une gestion riche des ressources comme l'allocation, l'isolement, le partage et le

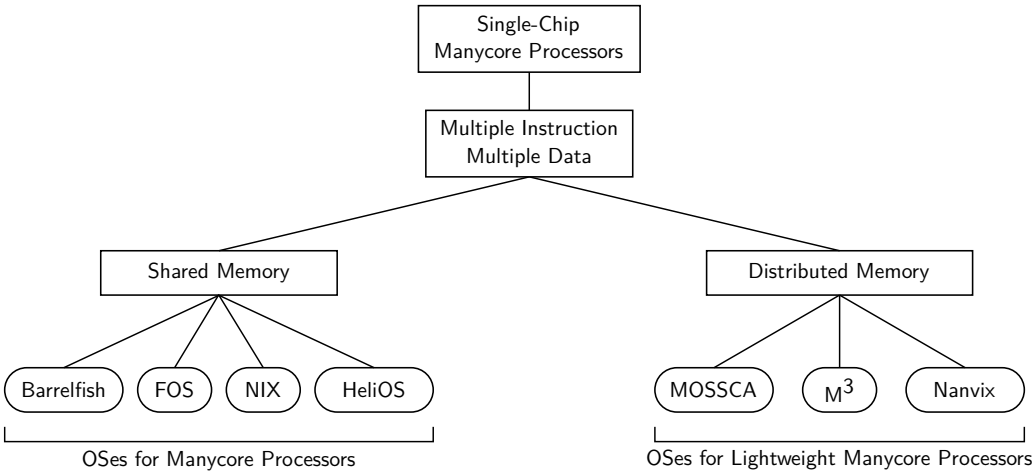


Figure 62 – Systèmes d'exploitation pour architectures multicœurs à puce unique.

multiplexage. En d'autres termes, ils ne sont pas suffisant pour fournir une solution complète de programmabilité et portabilité pour les processeurs LW Manycore.

OSes aspirent à combler les complexités d'une architecture, en exposant de riches abstractions, interfaces de programmation et gestion des ressources. Malheureusement, les OSes existants ont du mal à gérer pleinement la programmabilité et défis de portabilité dans les processeurs LW Manycore. La figure 62 en explique la raison et Ensuite, nous procédons à une autre discussion sur le sujet.

D'une part, les OSes pleinement tels que Barrelfish, FOS et HeliOS ne ciblent principalement LW Manycore, et n'ont pas été conçus pour traiter contraintes architecturales de ces processeurs. Par exemple, ils ne tenir compte des multiples espaces d'adressage physiques ni faire face au petit quantité de mémoires sur puce. En résumé, il n'est pas possible de déployer ces OSes sur LW Manycore sans refonte complète et significative changements sur leur code.

D'un autre côté, MOSSCA et M^3 ciblent LW Manycore, mais manque encore de capacité de programmation et de portabilité à ces processeurs. Dans l'ensemble, la justification de ceci réside dans le fait que les deux OSes n'offrent pas de support pour la mémoire virtuelle, ni aucun moyen pour transparent accès sur plusieurs espaces d'adressage et expansion de l'espace d'adressage. Les applications écrites pour ces systèmes doivent s'appuyer sur des API et gérer les données en accédant par eux-mêmes, entraînant ainsi une non-portabilité et mauvaise programmabilité. De plus, il convient de noter que MOSSCA et M^3 processeurs LW Manycore simulés localement ciblés, avec caractéristiques très particulières. Il n'est donc pas anodin de déterminer à quel dans la mesure où ils peuvent être portés sur d'autres architectures similaires, y compris le silicium frites.

Contrairement à ces OSes, Nanvix fournit une solution complète pour programmabilité et portabilité pour les processeurs LW Manycore. Pour ça fin, il s'appuie sur un système DPS pour surmonter les défis liés à la mémoire, il dispose d'une fonction de communication légère pour permettre une puce efficace communication, et il est fondé sur un HAL complet qui permet portabilité du système à une variété d'architectures. Dans le chapitre suivant, nous détaillons

notre solution.

C.4 Le système d'Exploitation Nanvix

Nanvix est un système d'exploitation distribué qui vise à gérer la programmabilité et problèmes de portabilité dans les processeurs LW Manycore. En résumé, c'est le concret mise en œuvre des contributions de cette thèse. Nanvix est open-source [1] et est le résultat d'une collaboration entre PUC Minas, UFSC et UGA.

C.4.1 Objectifs du Projet

Pour aborder la programmabilité et la portabilité dans LW Manycore, nous avons visé le objectifs de conception suivants pour Nanvix :

- Programmabilité
  - Activer le multiplexage des ressources pour permettre déployer simultanément plusieurs applications dans le processeur et ainsi augmenter l'utilisation globale de la plate-forme.
  - Assurer l'isolement des ressources pour empêcher les applications d'avoir soit ils ont un impact sur les performances, soit ont leur exécution tempéré.
- Portabilité
  - Fournit des abstractions riches, telles que des processus et des fichiers. De cette façon, les ingénieurs logiciels n'ont pas à se soucier de les implémenter, les intégrer et les maintenir dans leur logiciel
  - Expose des interfaces de programmation standard, permettant ainsi logiciel à porter avec un minimum de changements de syntaxe.
  - Prend en charge plusieurs paradigmes de programmation (¸est-à-dire partagé et mémoire distribuée), permettant ainsi un passage en douceur de architectures multicœurs et multicœurs traditionnelles.

[1] Disponible en public sur : `https://github.com/nanvix`.

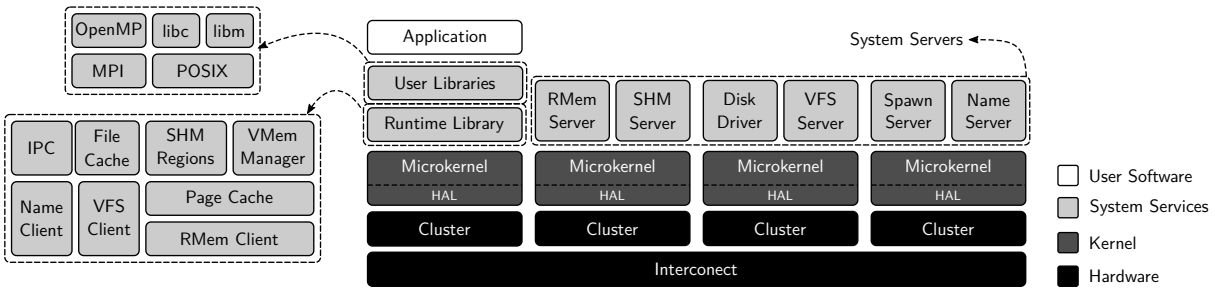


Figure 63 – Un aperçu structurel de Nanvix.

C.4.2 Architecture du Système

Nanvix présente une conception multikernel (BAUMANN et al., 2009; WENTZLAFF; AGARWAL, 2009) qui est structuré sur plusieurs instances supérieures d'un microkernel. Nous avons choisi cette organisation parce que nous pensons qu'elle répond le mieux aux caractéristiques de LW Manycore. D'une part, le ł'approche multinoyau s'adapte à l'architecture de mémoire distribuée de ces processeurs. D'autre part, la structure basée sur microkernel permet une mise en œuvre légère et modulaire des services système, ce qui est une exigence fondamentale compte tenu de la rareté du matériel Ressources. Figure 63 représente une architecture aperçu de Nanvix. Il a une disposition à quatre niveaux que nous présentons ensuite.

Le HAL expose une abstraction complète du sous-jacent matériel (PENNA; FRANCIS; SOUTO, 2019). Il est implémenté comme une bibliothèque baremetal, et il embarque des routines et des structures de données communes pour manipuler le matériel structures, telles que les noyaux, MMU et NoC. En résumé, le HAL permet au la portabilité de notre OS sur plusieurs processeurs LW Manycore, tels que Kalray MPPA-256 (DINECHIN et al., 2013), OpTiMSoC (WALLENTOWITZ et al., 2012) et PULP (ROSSI et al., 2017). Dans la section 4.3, nous approfondissons discuter de cette couche.

Le microkernel fournit le partage de ressources, le multiplexage, l'isolation et protection, au sein d'un cluster d'un LW Manycore (PENNA et al., 2019). Il court dans mode privilégié et expose les abstractions et primitives OS minimales, tels que les threads, l'espace d'adressage virtuel et la communication inter-processus mécanismes. Notre microkernel présente une conception asymétrique, ce qui signifie que il s'exécute exclusivement dans un cœur du cluster et laisse le reste noyaux à usage général. Cette conception atténue les interférences du noyau dans les logiciels de niveau utilisateur, et améliore ainsi les performances. Dans Section 4.4 nous détaillons le microkernel de Nanvix.

Les serveurs système sont déployés au-dessus du microkernel et implémentent services couramment trouvés dans un OS, tels que la planification de processus, la mémoire mappages et allocation de fichiers. À leur tour, les sous-systèmes sont mis en œuvre en ayant un ensemble de ces serveurs système qui fonctionnent ensemble, dans un mode. Dans la section 4.5, nous découvrons le services de Nanvix. Enfin, en plus des serveurs système, les bibliothèques d'exécution sont supprimées. Ces les bibliothèques sont liées aux applications utilisateur et fournissent un standard interface pour le logiciel de niveau utilisateur pour interagir avec le OS. Dans Section 4.6 nous discutons brièvement de les bibliothèques et les environnements d'exécution présentés par Nanvix.

C.5 Méthodologie d'évaluation

Notre méthodologie d'évaluation a été élaborée de manière à répondre aux trois questions principales, alignées sur les objectifs de ce travail (voir Section 1.3) :

Q-1 Comment Nanvix améliore-t-il la programmabilité dans les processeurs multicœurs légers ?

Table 13 – Programmes expérimentaux utilisés pour évaluer Nanvix

Groupe	Programme	Brève description
Synthetic Programs	`fence`	benchmarks latence de *syncs*
	`mail`	évalue la latence des *mailboxes*
	`cargo`	mesure le débit de *portals*
	`lkcall`	montre la latence des appels locaux au noyau
	`rkcall`	temps de latence des appels au noyau distant
	`forkjoin`	évalue le coût de création et de terminaison des threads
	`knoise`	évalue les interférences introduites par le noyau dans une application utilisateur
	`lookup`	mesure la latence pour déterminer l'emplacement physique d'un processus
	`heartbeat`	évalue la latence pour l'envoi d'un message vivant au gestionnaire de processus
	`stream`	évalue la bande passante en lecture/écriture de/vers la mémoire locale
	`rstream`	évalue la bande passante en lecture/écriture de/vers la mémoire distante
	`pginval`	cadence la latence pour invalider une entrée du cache de page
	`pgfetch`	fois la latence pour le transfert d'une page depuis le serveur de mémoire distant
System Utilities	`diff`	calcule la moindre sous-séquence entre deux chaînes
	`grep`	recherche un motif dans un texte
	`sed`	recherche et remplace des motifs dans un texte
	`sort`	trie les données numériques
Scientifique Programmes	`km`	regroupe un ensemble de points en plusieurs groupes
	`gf`	exécute un filtre gaussien sur une image 2D
	`fn`	calcule les nombres conviviaux sur une plage d'intervalles

Q-2 Dans quelle mesure Nanvix améliore-t-il la portabilité ?

Q-3 Quelles sont les performances de Nanvix ?

C.5.1 Programmes d'expérimentation

Nous avons envisagé plusieurs programmes expérimentaux pour évaluer spécifiquement la performance de Nanvix et ainsi répondre à la question d'évaluation Q-1. Dans l'ensemble, ces programmes peuvent être regroupés selon leurs caractéristiques et objectifs d'évaluation comme suit :

(i) *Synthetic Programs* sont implémentés au-dessus du système d'exécution de Nanvix (libNanvix) et ils sont conçus pour l'utilisation la plus efficace du système. Ces programmes se concentrent sur la mesure des caractéristiques spécifiques du système et nous les avons utilisées pour comprendre les performances des sous-systèmes individuels de Nanvix.

(ii) *System Utilities* consiste en un ensemble d'utilitaires Unix qui ont été portés sur Nanvix, en plus de sa bibliothèque C (uLibc). ces programmes forcer l'interaction entre les sous-systèmes ainsi qu'entre l'utilisateur et bibliothèques système. Nous nous appuyons sur ces programmes pour comprendre ce que performances à attendre lors de l'exécution d'applications système typiques dans Nanvix.

(iii) *Programmes scientifiques* sont composés de noyaux de suites de référence qui ont été portées vers Nanvix. Ces programmes utilisent le bibliothèques de programmation dis-

tribuées et parallèles disponibles dans notre OS (lwMPI et PThreads), et nous a permis de déterminer les performances que Nanvix peut fournir à des applications spécifiques à un domaine.

Le tableau 8 montre les principales caractéristiques de tous les programmes que nous avons considérés, et dans les sections suivantes, nous présentons chacune d'entre elles. Tous les programmes sont disponibles sur <https://github.com/nanvix>.

C.5.2 Plateforme expérimentale

Parmi les architectures supportées par Nanvix, nous avons choisi Kalray MPPA-256 comme plate-forme expérimentale dans ce travail. Il s'agit d'une publicité monopuce LW Manycore processeur qui présente la plupart des caractéristiques discuté dans Section 2.1.

Le processeur Kalray MPPA-256 (Figure 64) comporte 272 couleurs à usage général et 16 couleurs de firmware, appelées PE et RM, respectivement. Le processeur est construit avec la technologie CMOS 28 nm et il fonctionne jusqu'à 400 MHz. Toutes les couleurs implémentent un jeu d'instructions propriétaire 64 bits, présenter un pipeline VLIW à 5 problèmes, 8 kB I-Cache et D-Cache, et disposent d'un MMU géré par logiciel.

Les 288 couleurs de Kalray MPPA-256 sont regroupées en 16 Compute Clusters, qui sont destinés pour le calcul, et 4 I/O Clusters, qui sont conçus pour fournir connectivité aux périphériques. Chaque Compute Cluster regroupe 16 PE, un RM, deux NoC interfaces et 2 MB de SRAM local (128 kB par core). Dans ces clusters, la cohérence du cache matériel n'est pas prise en charge. En revanche, I/O Clusters ont 4 RM, 8 NoC interfaces et 4 MB de SRAM (1 MB par core). Deux de ces clusters sont connectés à un autre contrôleur DRAM, et les deux autres

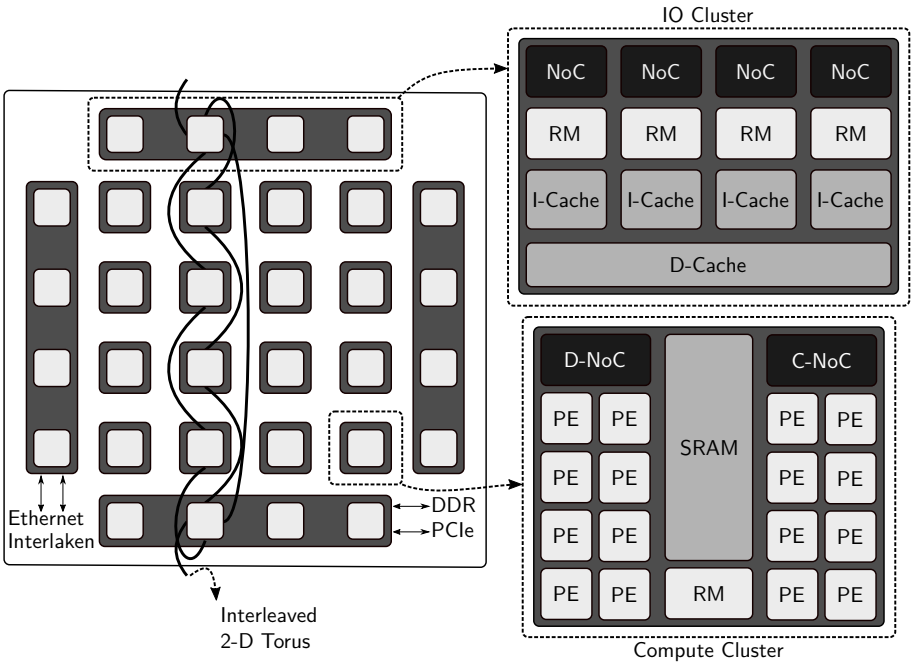


Figure 64 – Présentation architecturale du processeur MPPA-256 LW Manycore.

sont connectés à PCI et Ethernet contrôleurs. les clusters ne sont pas attachés à la mémoire globale et ils ont tous espaces d'adressage privés. Ainsi, les clusters doivent échanger du matériel messages par l'un des deux tores 2D entrelacés différents NoC à effectuer des communications : (i) un C-NoC à faible bande passante et est destiné aux petits transferts de données ; et (ii) le D-NoC qui présente une bande passante élevée et est dédié aux transferts de données denses.

Quel logiciel concerne le développement dans Kalray MPPA-256, ce processeur est livré avec une version corrigée de GCC 4.9.4 et Binutils 2.11.0. Aucun OS n'est fourni par le vendeur, et les ingénieurs logiciels doivent s'appuyer sur un environnement d'exécution non conforme pour écrire leur applications (DINECHIN et al., 2013), si vous n'utilisez pas Nanvix. Par ailleurs, concernant l'implémentation du noyau OS, les ingénieurs système doivent se fier sur un hyperviseur propriétaire de Kalray. Cet hyperviseur fonctionne sur le les couleurs du firmware du processeur et les intermédiaires de toutes les opérations de bas niveau. Il est à noter que l'hyperviseur Kalray ne peut pas être modifié ni configuré, et donc il impose des défis supplémentaires dans la construction de OS pour ce processeur LW Manycore.

C.5.3 Conception expérimentale

Nous nous sommes appuyés sur des approches quantitatives et qualitatives pour répondre aux questions d'évaluation que nous nous sommes posées (Section 5.1). pour en évaluant la programmabilité et la portabilité, nous menons une discussion qui dévoile comment ces aspects sont activés par Nanvix. En revanche, pour évaluer la performances de Nanvix nous exécutons les programmes que nous avons détaillés précédemment (Section 5.2) sur le Kalray MPPA-256 LW Manycore processeur (Section 5.3). Il convient de noter, en discutant des résultats expérimentaux des utilitaires système et programmes scientifiques que nous montrons effectivement que Nanvix offre programmabilité (Q-1) et portabilité (Q-2) vers LW Manycore.

Dans l'évaluation expérimentale, nous avons rassemblé des mesures de temps à l'aide de matériel compteurs de performances pour permettre une surveillance avec un minimum d'interférences. Sur le d'autre part, pour récupérer les mesures de consommation d'énergie, nous nous sommes appuyés sur un périphérique qui est fixé à l'extérieur de la carte du processeur. cette dispositif mesure la dissipation de puissance sur la carte et comprend des statistiques pour toutes les couleurs, NoC et autres ressources sur puce. De plus, pour chaque expérience, nous avons effectué 10 essais pour éliminer les effets d'échauffement indésirables, puis exécuté 30 essais pour recueillir les résultats. Tous les résultats qui sont présentés dans ce travail sont basés sur un seuil d'intervalle de confiance de 95%.

C.6 Résultats expérimentaux

Dans ce chapitre, nous présentons les résultats de notre évaluation. nous discutons d'abord résultats de l'évaluation spécifique que nous avons effectuée pour les sous-systèmes de

Nanvix, puis nous passons notre analyse aux résultats d'une évaluation plus large qui englobe l'interaction entre les sous-systèmes, ainsi que l'utilisateur et bibliothèques système.

C.6.1 Programmabilité et portabilité

Nanvix expose trois primitives principales qui peuvent être utilisées pour le processus communication et synchronisation : (i) *syncs* permet aux processus d'attendre des signaux et ainsi de se synchroniser les uns avec les autres ; (ii) *mailboxes* permet aux processus d'échanger des messages de taille fixe (iii) *portal* permet aux processus d'échanger des quantités arbitraires de données. Globalement, ces primitives améliorent la programmabilité sur plusieurs points. D'abord, ils ne nécessitent pas que le programmeur sache dans quel cluster un processus donné est en cours d'exécution. Deuxièmement, ils gèrent le contrôle du flux de communication. troisièmement, ils gérer l'allocation des ressources matérielles de bas niveau. Enfin, ils permettent la multiplexage des interfaces NoC.

Extraits de code C.1 et C.2 illustre certains de ces améliorations en montrant comment l'échange de messages de taille fixe peut être réalisé dans Nanvix, contrairement aux bibliothèques de fournisseurs fournies avec Kalray MPPA-256. Lors de l'utilisation de cette dernière bibliothèque C.1, les éléments suivants des étapes sont impliquées. Tout d'abord, le chemin du point de terminaison distant doit être construit en conséquence (ligne 4). Ceci spécifie quel matériel tampon émetteur à utiliser (`rxbuf`), la taille des messages qui sera envoyé par et l'identifiant physique de la destination grappe. Une fois cette configuretion terminée, le canal de communication vers le point de terminaison est ouvert (ligne 5), le message est envoyé (ligne 6) et le le canal de communication final est fermé (ligne 7). Il est à noter que dans ce flux, l'ingénieur logiciel doit gérer des aspects du matériel, lors de la configuretion du point de terminaison. En revanche, lors de l'utilisation du système de communication de Nanvix (Code Snippet C.2) trois étapes plus simples sont obligatoire: (i) ouvre une boîte aux lettres au processus cible en fournissant son numéro d'identification (ligne 3); (ii) écrit le message via la boîte aux lettres (ligne 4); et (iii) ferme la boîte aux lettres (ligne 5).

En matière de portabilité, les primitives exposées permettent de constriction de POSIX synchronisation/abstractions de communication comme suit :

```
1  void send_message(int dest, int rxbuf, char (*msg)[128])
2  {
3    char path[64];
4    sprintf(path, "/mppa/channel/128:%d/%d:%d", rxbuf, dest, rxbuf);
5    int ch = mppa_open(path, O_WRONLY);
6    mppa_write(ch, msg, 128);
7    mmppa_close(ch)
8  }
```

Snippet C.1 – Envoi de messages de taille fixe en MPPA à l'aide des bibliothèques baremetal de Kalray.

```
1  void send(pid_t dest, char *msg, size_t n)
2  {
3    int outbox = kmailbox_open(dest);
4    kmailbox_write(outbox, msg, n);
5    kmailbox_close(outbox)
6  }
```

Snippet C.2 – Envoi de messages de taille fixe en MPPA à l'aide de Nanvix.

- Les signaux sont implémentés au dessus des *mailboxes*. Dans ce contexte, les messages portent des informations sur un signal concerné, telles que sa numéro.
- Les sémaphores sont implémentés au-dessus des *mailboxes* et des *syncs*. Le premier est utilisé pour transmettre des valeurs d'incréments/décréments, tandis que ce dernier est utilisé pour envoyer des notifications de réveil.
- Les files d'attente de messages sont implémentées au-dessus des *mailboxes*. Dans ce cas, les messages encapsulent la charge utile qui est transmise, ainsi que la priorité d'un message.
- Les canaux sont implémentés en utilisant *portals* et *sync*. Le premier est utilisé pour transférer des données, alors que ce dernier est utilisé pour envoyer/attendre réveillez les notifications.

Dans cette section, nous évaluons le système de mémoire de Nanvix. d'abord, nous discutons comment il améliore la programmabilité et la portabilité dans LW Manycore processeurs via le DPS (Section 6.2.1). Ensuite, nous découvrons les performances des accès mémoire, ainsi que nous analysons la latence pour l'invalidation et la récupération des pages (Section 6.2.2).

Rappelez-vous que les processeurs LW Manycore ont des contraintes concernant leur système de mémoire. Premièrement, ils comportent un architecture de mémoire distribuée : les modules de mémoire sont physiquement répartis sur le processeur, l'espace d'adressage n'est pas unique, et la mémoire n'est pas globalement accessible par toutes les couleurs. Accès à la mémoire à distance n'est possible qu'avec la communication de transmission de messages via le réseau sur puce. Deuxièmement, les temps d'accès aux modules de mémoire ne sont pas uniformes. Il est plus rapide d'accéder à une banque de mémoire locale que de accéder aux données (via le NoC) qui sont stockées sur un module de mémoire distant. Enfin, la quantité de mémoire sur puce est très limitée. pour exemple, dans le processeur Kalray MPPA-256,

```
1  size_t n = (1 << 30);          /* 1 GB                          */
2  char *mem = malloc(n);         /* Allocate a memory area.       */
3
4  for (size_t i = 0; i < n; i++) /* Manipulate the memory area. */
5    mem[i] = 0xff;
6
7  free(mem);                     /* Free the memory area.         */
```

Snippet C.3 – Utiliser le cas pour l'extension d'adresse.

```
1  /* Allocate some shared memory. */
2  int fd = shm_open("shared-memory", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
3  ftruncate(fd, 4096);
4  char *shmem = mmap(NULL, n, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)
5
6  /* Write to shared memory. */
7  for (size_t i = 0; i < 4096; i++)
8    shmem[i] = 0xff;
9
10 munmap(shm, size);
11 shm_unlink(fd);
```

Snippet C.4 – Utiliser le cas pour le partage d'espace d'adressage.

les mémoires sur puce sont de 2 à 4 Mo gros. En résumé, ces caractéristiques architecturales imposent défis à la fois en termes de programmabilité et de portabilité des logiciels. nous Discutez plus en détail de la façon dont notre système de mémoire surmonte ces problèmes dans le prochain paragraphes en examinant deux exemples de cas d'utilisation représentatifs.

Dans Snippet C.3, nous présentons du code C qui alloue dynamiquement à une grande zone de mémoire (1 Go) en utilisant la norme l'allocateur de mémoire de la bibliothèque C (ligne 2), manipulez cette mémoire zone (lignes 4 et 5) et la libère (ligne 7). Cet exemple fonctionnerait dans la plupart des systèmes modernes qui ont suffisamment de mémoire virtuelle. Cependant, OSes pour LW Manycore manque actuellement de mémoire riche au niveau du système gestion et ni ces processeurs n'ont assez de physique sur puce mémoire pour accomplir cette opération. Par conséquent, seuls les programmes qui avoir des empreintes mémoire suffisamment petites pour tenir dans les mémoires physiques d'un LW Manycore cluster peut être porté sans introduire modifications importantes de leur code source. Nanvix permet cela exemple de cas d'utilisation à exécuter sans aucune modification dans Kalray MPPA-256.

Dans Snippet 6.4, nous présentons un fragment d'un C programme qui décrit les bases du partage d'espace d'adressage entre deux processus. Cela fonctionne comme suit. Tout d'abord, il alloue à 4 kB de mémoire partagée région, qui est tapée comme lisible et inscriptible par les programmes du même utilisateur (ligne 2). Ensuite, il attache cette région de mémoire partagée à l'espace d'adressage du programme appelant (ligne 4). Ensuite, il écrit dans cette mémoire partagée région (lignes 7 et 8). Enfin, il libère cette région de mémoire partagée (lignes 10 et 11). Malheureusement, les OSes actuels pour LW Manycore ne prennent pas en charge ce flux d'exécution. La seule façon dont les programmes ont communiquer passe par la transmission de messages. Ceci grandement modifie la conception des programmes et compromet ainsi la programmabilité. Nanvix prend en charge les régions de mémoire partagée et permet aux processus de les utiliser pour communiquer.

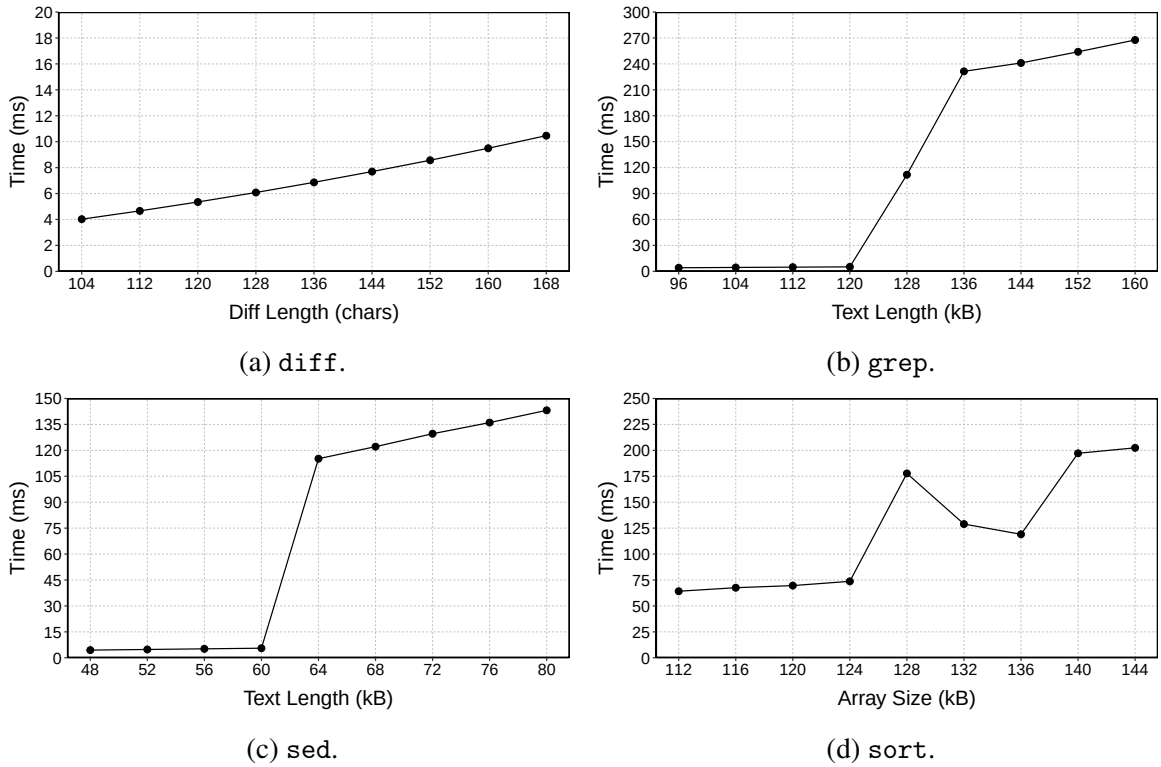
C.6.2 Performances des utilitaires système

La figure 65 présente le temps d'exécution des quatre unix outils pris en compte, lors de la variation de la taille de l'entrée fonctionnant ensemble. Il est à noter que chaque utilitaire a

été exécuté exclusivement dans Nanvix. Dans en d'autres termes, à part les services OS, il n'y avait pas d'autre niveau d'utilisateur application en cours d'exécution. Par conséquent, il n'y a pas d'interférence externe, et ainsi, les résultats rapportés reflètent la performance de la limite supérieure.

Lors de l'observation des résultats de l'utilitaire `diff` (Figure 65a), nous avons noté une augmentation linéaire de temps d'exécution lors de l'augmentation de la taille de l'ensemble de travail. la raison car ce comportement s'explique par les caractéristiques de l'application. L'algorithme de sous-séquence le moins courant traite la table de mémorisation en avant et chaque nouvel accès à cette structure de données entraîne une page la faute. par conséquent, plus l'ensemble de données est grand, plus le nombre des défauts de page, ce qui entraîne des temps d'exécution plus longs. depuis le temps de la récupération des pages est constante, le temps d'exécution linéaire suit.

Figure 65b et Figure 65c présents les temps d'exécution des programmes `grep` et `sed`, respectivement. À la fois cas, nous avons observé un comportement de performance échelonné qui conduit à complétions complémentaires pour `diff`. Ces deux utilitaires (i.e., grep et `sed`) proposent des modèles d'accès qui profitent grandement de l'espace localité. Pour cette raison, lorsque l'ensemble de travail est suffisamment petit pour tenir dans le cache des pages de notre système mémoire, nous avons observé des temps d'exécution réduits. Cependant, dès que l'ensemble de travail ne rentre pas entièrement dans le cache, nous observé que le temps d'exécution augmente considérablement. Il est à noter, lorsque cette dernière situation se produit,



(a) `diff`. (b) `grep`.

(c) `sed`. (d) `sort`.

Figure 65 – Performance pour les utilitaires système.

les augmentations du temps d'exécution sont linéaires, car nous observé pour `diff`. Enfin, il est important de souligner que les résultats de ces expériences montrent à quel point le cache de pages est important pour notre système. Merci à celui-ci, les applications peuvent atteindre jusqu'à 10× de meilleures performances que accéder directement aux mémoires distantes.

Enfin, nous présentons dans la Figure 65d les résultats obtenus avec l'utilitaire `sort`. Dans cette expérience, nous avons remarqué un comportement plus irrégulier avec trois phases principales. Premièrement, lorsque la taille de l'ensemble de travail est petite assez pour tenir dans le cache de la page (¸est-à-dire jusqu'à 128 kB), temps d'exécution augmente légèrement. Ensuite, lorsque la taille de l'ensemble de travail varie de 100 % à environ 105 % de la taille du cache de pages, les performances présentent une grande variation. Troisièmement, lorsque l'ensemble de travail est suffisamment grand, nous observons augmentations linéaires du temps d'exécution à partir d'une ligne de base plus élevée. Les résultats pour les première et troisième phases sont alignées sur les comportements notés dans le autres utilitaires. En revanche, le raisonnement du comportement observé dans la deuxième phase provient du modèle d'accès aléatoire du programme lui-même. Dans ce scénario, la stratégie de remplacement de page fonctionne au plus aussi mal qu'un modèle pathologique (pic dans 128 kB) ou aussi bon que lorsque la localité est exploité (creux dans 136 kB).

La figure 66 représente les performances de Nanvix lorsqu'il est soumis à de lourdes charges. Dans cette expérience, nous avons déployé plusieurs instances des quatre utilitaires Unix en même temps. Les images système ont été généré aléatoirement, variant le nombre de processus dans le système. plus important, cette expérience montre que Nanvix active les ressources multiplexage, isolation et partage dans LW Manycore processeurs.

Dans l'ensemble, nous avons noté que le temps d'exécution présente une certaine irrégularité, variant de 12 ms jusqu'à 30 ms. Cependant, ce comportement était attendu car les images système étaient générées aléatoirement et le les caractéristiques de chaque service public varient de l'un à l'autre. néanmoins, le le temps d'exécution n'augmente pas de manière significative lors du passage d'un scénario léger, dans lequel un seul utilitaire est déployé, à un un poids lourd, dans lequel seize services publics fonctionnent simultanément dans Nanvix. Les performances

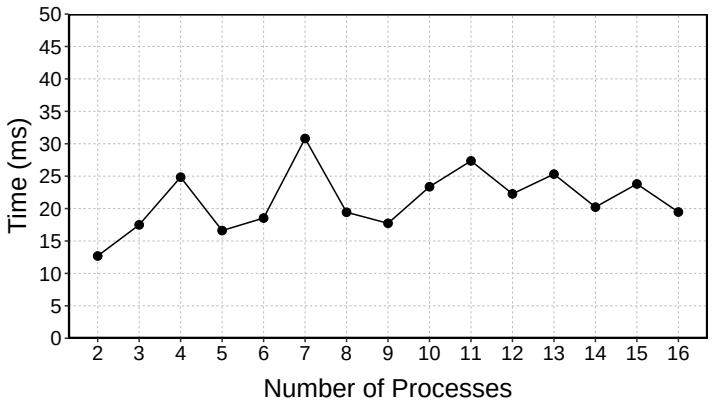


Figure 66 – Performance de Nanvix sous forte charge.

diminuent de manière sublinéaire avec la charge dans le système : les performances globales ne se dégradent que de 38 % en augmentant charge du système par un facteur de 16×.

C.7 Conclusion

LW Manycore ont été introduits pour offrir une évolutivité des performances avec une faible consommation consommation. Pour répondre au premier aspect, ces processeurs s'appuient sur des caractéristiques architecturales, telles qu'une architecture de mémoire distribuée et un riche NoC. Pour atteindre une faible consommation d'énergie, ces manycores sont construits avec couleurs MIMD simples à faible consommation, ils ont un système de mémoire basé sur SPM et ils exploiter l'hétérogénéité en présentant des couleurs avec des capacités différentes. additionner des exemples de succès dans l'industrie de ces processeurs sont le Kalray MPPA-256, le PULP et le Sunway SW26010.

Bien que cet ensemble unique de caractéristiques architecturales confère à LW Manycore des performances l'évolutivité et l'efficacité énergétique, ils présentent également de multiples défis dans développement et déploiement de logiciels. Premièrement, le *intégration de circuit haute densité* transforme le silicium noir en réalité. Deuxièmement, la *architecture de mémoire distribuée* exige que les données soient explicitement récupéré/déchargé des mémoires distantes vers les mémoires locales. Troisièmement, le *petite quantité de mémoire sur puce* force le logiciel partitionner son ensemble de données de travail en morceaux et décider lequel d'entre eux doit être conservés en local et qui doivent être déchargés dans la mémoire distante. Quatrièmement, le *on-chip interconnect* invite les ingénieurs logiciels à adopter une modèle de programmation de passage de messages. Enfin, la *hétérogénéité sur puce* rend le déploiement de applications complexes.

Pour résoudre les problèmes susmentionnés, il existe deux approches principales : runtime systems et OSes. Le premier vise à exposer un environnement de programmation à répondre aux besoins d'une classe spécifique d'applications (HO et al., 2015; Mohr; Tradowsky, 2017). En revanche, la seconde approche aspire à combler les complexités du matériel dans un sens plus large, en exposant des abstractions riches et des interfaces de programmation, comme ainsi que la gestion de l'allocation, du partage et du multiplexage des ressources.

Dans le contexte des processeurs LW Manycore, malheureusement, pas aujourd'hui, aucun des deux les approches précédentes fournissent une solution complète pour aborder la programmabilité et défis de portabilité. D'une part, runtime systems manque intrinsèquement de couvrir tous les problèmes que nous indiqué précédemment. Par exemple, certains runtime systems fournissent de riches les abstractions s'attaquent à la communication sur puce, mais négligent les défis qui concernent l'architecture de la mémoire distribuée. D'un autre côté, les fichiers existants ne sont pas conçus pour faire face à tous les contraintes architecturales des processeurs LW Manycore. En résumé, il n'est pas possible de déployer ces OSes dans LW Manycore sans refonte complète et/ou significative modifications de leur code source. Dans l'ensemble, nous avons identifié le problème sans réponse suivant dans le des systèmes

d'exploitation de pointe, que nous visons à aborder dans cette thèse : Comment concevoir un système d'exploitation de manière à gérer la programmabilité et des problèmes de portabilité dans les processeurs multicœurs légers ?

LW Manycore ont été introduits pour offrir une évolution des performances avec une faible consommation consommation. Pour répondre au premier aspect, ces processeurs s'écrivent sur des caractéristiques architecturales, telles qu'une architecture de mémoire distribuée et un riche NoC. Pour atteindre une faible consommation d'énergie, ces manycores sont construits avec couleurs MIMD simple à faible consommation, ils ont un système de mémoire basé sur SPM et ils exploiter l'hétérogénéité en présentant des couleurs avec des capacités différentes. additionneur exemples de succès dans l'industrie de ces processeurs sont le Kalray MPPA-256, le PULP et le Sunway SW26010.

Bien que cet ensemble unique de caractéristiques architecturales confère à LW Manycore de performances l'évolutivite et l'efficacité énergétique, ils présentent également de multiples défis dans développement et déploiement de logiciels. Premier, le *integration of circuit haute densité* transform le silicium noir en réalité. Deuxièmement, la *architecture de mémoire distribuée* exige que les données soient récupéré/déchargé des mémoires distantes versus mémoires locales. Troisièmement, le *petite quantité de mémoire sur puce* force le logiciel partitionner son ensemble of donnees de travail en morceaux et décider fanl d'entre eux doit être conservé en local et qui doit être chargé dans la mémoire distante. Quatrièmement, le *on-chip interconnect* invite les ingénieurs logiciels à adopter une modèle de programmation de passage de message. Enfin, la *hétérogénéité sur puce* louer le déploiement de candidatures complexes.

Pour résoudre les problèmes proposés, il existe deux approches principales : runtime systems et OSes. Le premier vise à exposer un environnement de programmation à répond aux besoins d'une classe spécifique d'applications (HO et al., 2015; Mohr; Tradowsky, 2017). En revanche, la seconde approche aspire à combler les complexités du matériel dans un sens plus large, en exposant des abstractions riches et des interfaces de programmation, comme ainsi que la gestion de l'allocation, du partage et du multiplexage des ressources.

Dans le contexte des processeurs LW Manycore, malheureusement, pas aujourd'hui, aucun des deux les approches précédentes fournissent une solution complète pour la frontière de la programmabilité et déficits de portabilité. D'une part, runtime systems manque fondamentalement à couvrir tous les problèmes que nous ne connaissons pas Indiquer précédemment. Par exemple, certains runtime systems fournissent de richesses Les abstractions s'attaquent à la communication sur puce, plus négligent les déficits qui concernent l'architecture de la mémoire distribuée. D'un autre coté, les fichiers existants ne sont pas conçus pour faire face à tous contracting architecturales des processeurs LW Manycore. En résumé, il n'est pas possible de visible ces OSes dans LW Manycore sans refonte complete et/ou significative modifications du code source lu. Dans l'ensemble, nous avons identifié le problème sans réponse suivant dans le des systèmes d'exploitation de pointe, that nous visons à border dans cette thèse : Commentaire pour concevoir un système d'exploitation de manière à générer la programmabilité et des problèmes

de portabilité dans les processeurs multicœurs légers ?

C.7.1 Contributions scientifiques

L'apport principal de cette thèse – *un roman distribué OS qui fait progresser la gestion des ressources dans LW Manycore processeurs* – peut se dérouler dans les contributions scientifiques spécifiques suivantes :

- Un HAL complet pour LW Manycore qui comble les différences architecturales entre plusieurs architectures, et fait face aux problèmes clés qui sont souvent rencontrés lors de la conception d'un OSes pour ces processeurs. Avec cette couche d'abstraction, le développement et le déploiement d'un OS complet devient plus facile non seulement pour un LW Manycore particulier, mais aussi permet la portabilité d'un OS spécifique sur plusieurs de ces processeurs émergents.

- Une approche de gestion de mémoire riche basée sur DPS. c'est un nouvelle solution que nous avons conçue pour gérer la mémoire d'un processeur LW Manycore et cela fonctionne comme suit. Les mémoires locales d'un processeur LW Manycore sont considérés comme des caches de pages pouvant stocker les données de n'importe quel processus. Globalement, cette solution permet un accès transparent aux données et la manipulation, ainsi que le partage et la cohérence des données dans LW Manycore ;

- Une installation de communication légère qui gère le interconnecter et exposer les primitives avec le multiplexage des canaux.

C.7.2 Production Technique

Du point de vue technique, cette thèse introduit également une importante contribution, qui est Nanvix. Ceci est une implémentation concrète d'un OS pour LW Manycore processeur qui comporte les éléments susmentionnés avancées scientifiques. Nanvix prend en charge plusieurs architectures (i.e., x86, OpenRISC, ARMv8 et RISC-V), fonctionne sur silicium LW Manycore processeurs, expose des abstractions riches et des interfaces de programmation de haut niveau.

Nanvix présente une conception multikernel qui est structuré sur plusieurs instances supérieures d'un microkernel. nous avons choisi cette organisation parce que nous pensons qu'elle répond le mieux aux caractéristiques de LW Manycore. D'une part, le ł'approche multinoyau s'adapte à l'architecture de mémoire distribuée de ces processeurs. D'autre part, la structure basée sur microkernel permet une mise en œuvre légère et modulaire des services système, ce qui est une exigence fondamentale compte tenu de la rareté du matériel Ressources. Il a une disposition à quatre niveaux que nous présentons ensuite.

Le HAL expose une abstraction complète du sous-jacent matériel (PENNA; FRANCIS; SOUTO, 2019). Il est implémenté comme une bibliothèque baremetal, et il embarque des routines et des structures de données communes pour manipuler les structures matérielles, telles

que les couleurs, MMU et NoC. En résumé, le HAL permet la portabilité de ou OS sur plusieurs processeurs LW Manycore, tels que Kalray MPPA-256, OpTiMSoC et PULP.

Le microkernel fournit le partage de ressources, le multiplexage, l'isolation et protection, au sein d'un cluster d'un LW Manycore (PENNA et al., 2019). Il court dans mode privilégié et expose les abstractions et primitives OS minimales, tels que les threads, l'espace d'adressage virtuel et la communication inter-processus mécanismes. Notre microkernel présente une conception asymétrique, ce qui signifie que il s'exécute exclusivement dans un cœur du cluster et laisse le reste couleurs à usage général. Cette conception atténue les interférences du noyau dans les logiciels de niveau utilisateur, et améliore ainsi les performances.

Les serveurs système sont déployés au-dessus du microkernel et implémentent services couramment trouvés dans un OS, tels que la planification de processus, la mémoire mappages et allocation de fichiers. À leur tour, les sous-systèmes sont mis en œuvre en ayant un ensemble de ces serveurs système qui fonctionnent ensemble, dans un mode. Enfin, en plus des serveurs système, les bibliothèques d'exécution sont en cours. ces les bibliothèques sont liées aux applications utilisateur et fournissent une interface standard pour que le logiciel de niveau utilisateur interagisse avec le OS.