UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

João Vicente Souto

A Task-based Execution Engine for Distributed Operating Systems Tailored to Lightweight Manycores with Limited On-Chip Memory

Florianópolis
25 de julho de 2022

João Vicente Souto

A Task-based Execution Engine for Distributed Operating Systems Tailored to Lightweight Manycores with Limited On-Chip Memory

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do título de Mestre em Ciência da Computação.
Orientador: Prof. Márcio Bastos Castro, Dr.

Florianópolis
25 de julho de 2022

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

Souto, João Vicente
A task-based execution engine for distributed operating systems tailored to lightweight manycores with limited on chip memory / João Vicente Souto ; orientador, Márcio Bastos Castro, .
72 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, .

Inclui referências.

1. Ciência da Computação. 2. Lightweight manycores. 3. Sistemas operacionais distribuídos. 4. Microkernel assimétrico. 5. Restrições de memória. I. Castro, Márcio Bastos. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

João Vicente Souto

A Task-based Execution Engine for Distributed Operating Systems Tailored to Lightweight Manycores with Limited On-Chip Memory

Florianópolis, 25 de julho de 2022.

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

ACKNOWLEDGEMENTS

This study was financed in part by *Fundação de Amparo à Pesquisa e Inovação do Estado de Santa Catarina* (Fapesc) and *Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil* (CNPq).

We can only see a short distance ahead,
but we can see plenty there that needs to be done.
(TURING, A., 1950)

RESUMO

Processadores *lightweight manycore* surgiram para conciliar os requisitos de desempenho, eficiência energética e escalabilidade um único *chip*. Sistemas Operacionais (SOs) para essa classe de processadores apresentam um *design* distribuído, onde instâncias isoladas do SO cooperam para mitigar problemas de programação e portabilidade provenientes de suas complexidades arquitetônicas. Atualmente, serviços do SO geralmente recorrem a abstrações de fluxo de execução tradicionais (processos ou *threads*) para implementar funcionalidades pequenas, periódicas ou assíncronas. Embora essas abstrações simplifiquem consideravelmente o projeto do sistema, elas têm um impacto inegociável nas limitadas memórias presentes no *chip*. Devido às restrições de memória, argumentamos que as abstrações no nível do SO podem ser reformuladas para reduzir o consumo de memória do SO, sem introduzir uma sobrecarga considerável. Neste contexto, propomos um motor de execução complementar ao nível do SO que suporta tarefas leves e cooperativas que compartilham uma única pilha de execução e possuem recursos de sincronização por meio de grafos de fluxo de controle e dependência. Essa solução é ortogonal ao suporte de execução subjacente e fornece uma quantidade significativa de fluxos de execução no nível do SO com consumo de memória reduzido. Implementamos nosso motor em um SO distribuído e executamos experimentos em um *lightweight manycore* real. Nossos resultados mostram que o motor proposto possui as seguintes vantagens quando comparada à abstração clássica de *thread*: (i) fornece $63,2$ vezes mais fluxos de execução por MB de memória; (ii) apresenta menor sobrecarga para gerenciar fluxos de execução e chamadas de sistema; (iii) melhora a utilização do núcleo dedicado ao SO; e (iv) apresenta resultados competitivos em aplicações do mundo real.

Palavras-chave: Lightweight Manycores. Sistemas Operacionais Distribuídos. Microkernel Assimétrico. Restrições de Memória.

RESUMO ESTENDIDO

Introdução

Processadores *lightweight manycore* destacam-se pelo seu alto grau de paralelismo e baixo consumo energético. Devido a características que os diferem dos demais *manycores*, essa classe de processadores consegue conciliar requisitos de desempenho, eficiência energética e escalabilidade em um único *chip*. Particularmente, *lightweight manycores* (i) integram milhares de núcleos de baixa potência com capacidade *Multiple Instruction Multiple Data* (MIMD); (ii) apresentam uma arquitetura de memória distribuída com pequenas memórias locais compartilhadas por grupos de núcleos (vulgo *clusters*); (iii) possuem Redes-em-Chip (NoCs) confiáveis e rápidas para troca de mensagens; e (iv) podem oferecer capacidades de processamento heterogêneas. Dentre os aspectos arquiteturais de um *lightweight manycore*, o sistema de memória distribuída desempenha um papel crítico. Especificamente, memórias pequenas, restritivas e fisicamente separadas compõem múltiplos espaços de endereçamento, forçando desenvolvedores a reprojetar suas aplicações para atingir um desempenho satisfatório. Apesar de tais aspectos possibilitarem a escalabilidade e eficiência energética aprimorada desses processadores, eles criam um ambiente suscetível a erros. Consequentemente, Sistemas Operacionais (SOs) Distribuídos para *lightweight manycore* foram recentemente propostos para lidar efetivamente com as complexidades arquitetônicas desses processadores, oferecendo um ambiente de execução mais robusto e produtivo. Dentre os SOs propostos, o *multikernel* destaca-se por mitigar problemas de programabilidade e portabilidade através de um ambiente distribuído composto por instâncias isoladas do SO que interagem com outras entidades do sistema por meio de uma abordagem cliente-servidor. Como a memória no *chip* é limitada, o objetivo mais importante em um projeto de SO é manter seu *kernel* pequeno enquanto preserva suas funcionalidades mais importantes. Sob essa perspectiva, o *design* de um *microkernel* assimétrico como instância do *multikernel* destaca-se por ser uma solução flexível e escalável ao mesmo tempo que apresenta baixo consumo de memória. Neste *design*, o SO é executado isoladamente em um núcleo do *cluster*, deixando os demais para uso geral. Desta forma, o SO apresenta menor interferência nas aplicações do usuário. No entanto, o *microkernel* não é suficiente para aliviar todos os problemas provenientes das restrições de memória existentes nos *lightweight manycores*. Por exemplo, serviços do SO geralmente recorrem a abstrações de fluxo de execução clássicas (como processos, corrotinas e *threads*) para implementar funcionalidades pequenas, periódicas ou assíncronas no nível do SO. Embora essas abstrações simplifiquem consideravelmente o projeto do sistema, elas têm um impacto inegociável nas limitadas memórias do *chip*. A redução de memória disponível afeta o desenvolvimento de software desde o nível do *kernel* até as aplicações do usuário. Nesse contexto, nós argumentamos que as abstrações de fluxo de execução no nível do SO podem ser remodeladas para reduzir o consumo de memória do *kernel* sem introduzir uma sobrecarga considerável ao sistema.

Objetivos

Para aliviar as restrições de memória presentes nos *lightweight manycores*, nós propomos um motor de execução complementar no nível do SO que suporta tarefas leves e cooperativas que compartilham uma pilha de execução e possuem recursos de sincronização por meio de grafos de fluxo de controle e dependência. Essa solução permite a execução de vários fluxos de execução no nível do SO com consumo de memória reduzido, uma alter-

nativa às custosas abstrações tradicionais. Neste contexto, o motor de execução baseado em tarefas proposto tem como principais objetivos: (i) reduzir o consumo de memória do SO sem introduzir sobrecarga considerável (leveza); (ii) ser independente do suporte de execução nativo subjacente, aproveitando o ambiente pré-existente (ortogonalidade); e (iii) coexistir com soluções tradicionais para suportar o reprojeto incremental das funcionalidades do SO (flexibilidade). De forma geral, o projeto de implementação do motor de execução apresenta contribuições significativas ao estado da arte em suporte de SOs para processadores *lightweight manycore*. Especificamente, nós suportamos uma maior quantidade de funcionalidades confiáveis no nível do SO com consumo de memória reduzido para *lightweight manycores* com memória interna limitada.

Metodologia
O motor de execução baseado em tarefas proposto é genérico suficiente para ser implementado em qualquer SO distribuído projetado para *lightweight manycores*. Como prova de conceito, nós implementamos a proposta desta dissertação no Nanvix, um *multikernel* distribuído de código aberto compatível com o padrão POSIX. Para explorar os benefícios almejados pelo motor proposto, nós reprojetamos módulos, funcionalidades, serviços e um ambiente MPI (LWMPI) existentes no Nanvix. Para avaliar de forma compreensível nossa solução, nós projetamos um conjunto de *benchmarks* para mensurar impactos na memória, tempo de execução e consumo energético no Kalray MPPA-256, um *lightweight manycore* comercial. O conjunto de *benchmarks* comparam nossa solução com a abstração de *thread* nativa sob três variantes do Nanvix. As variantes aumentam gradualmente a remodelagem do sistema, avaliando a versão original (`Baseline`), a introdução de comunicações usando tarefas (`Partial`) e a adição da remodelagem dos serviços baseado em tarefas (`Full`). Ao todo, nós coletamos métricas de 10 execuções de aplicação executando no ambiente MPI (aplicações realísticas) e 30 execuções de *benchmarks* sintéticos, garantindo resultados estaticamente relevantes. Todos os resultados representam valores médios e baseados em um intervalo de confiança de 95% (significância de 5%).

Resultados e Discussão
Em todos os experimentos, nós focamos na comparação dos resultados obtidos com nossas implementações baseadas em tarefas contra as soluções baseadas em *threads* nativas do Nanvix. Utilizando um experimento teórico que avalia o consumo de memória pelas soluções, demonstrou-se que o motor proposto fornece $63,2$ vezes mais fluxos de execução por MB de memória. Adicionalmente, um conjunto de experimentos sintéticos empregou aplicações artificiais para mensurar o tempo de resposta do SO, assim como potenciais sobrecargas introduzidas ao *microkernel*. Especificamente, os experimentos sintéticos mostram que o motor proposto (i) apresenta melhor escalabilidade e menor custo para iniciar funcionalidades do SO; (ii) o tempo de resposta de chamadas de sistemas remotas são $3,1\times$ mais rápidas; e (iii) o compartilhamento do núcleo dedicado ao SO entre a thread de *kernel* (*thread* mestre) e o *Dispatcher* (*thread* executora de tarefas) não adicionou sobrecarga significativa no tempo de resposta do SO. Para avaliar os impactos no nível do *multikernel*, dois conjuntos de *benchmarks* foram propostos para avaliar a performance dos serviços do SO e de aplicações do mundo real. A avaliação dos serviços do SO demonstraram que o motor proposto (i) melhor gerenciou comunicações pequenas, explorando o menor tempo de resposta das chamadas de sistema; (ii) promoveu maior isolamento do *kernel*, uma premissa violada pelo módulo de comunicação original; e (iii) os *daemons* de serviços existentes não apresentaram interferência significativa ao

sistema quando implementados com tarefas. Por fim, escolhemos três aplicações MPI de um pacote de *benchmarks* para *lightweight manycores* (CAP Bench) com o intuito de examinar condições mais realísticas. As aplicações executam sobre o LWMPI suportado pelo Nanvix, tornando trivial sua adaptação para as diferentes variantes consideradas. No geral, os resultados mostraram que apesar do motor serializar a execução de operações que eram originalmente paralelas, o mesmo apresenta resultados competitivos em todas as aplicações.

Considerações Finais
Processadores *lightweight manycore* alcançam alto desempenho e eficiência energética graças a seus aspectos arquitetônicos, como extremo paralelismo com uma arquitetura de memória distribuída e restritiva. SOs para essas arquiteturas adotam o modelo distribuído de um *multikernel* para prover escalabilidade enquanto expõem um ambiente mais robusto e produtivo ao usuário. Complementarmente, o *design* de um *microkernel* assimétrico é comumente adotado para lidar com as peculiaridades dos *lightweight manycores* devido ao seu consumo de memória reduzido e escalabilidade. No entanto, tais abordagens não contemplam todas as restrições provenientes da memória limitada desses processadores. Para aliviar as restrições de memória existentes, nós propomos um motor de execução baseado em tarefas como uma alternativa às custosas abstrações de fluxo de execução tradicionais, como processos ou *threads*. O principal aspecto do motor é a eliminação da necessidade de processos/*threads* dedicados para implementar funcionalidades leves no nível do SO. Desta forma, fluxos de execução como *daemons* de serviços do SO podem ser implementados com consumo de memória reduzido. Além disso, o motor é ortogonal ao suporte de execução do SO subjacente, exigindo poucas modificações ao *kernel*. A implementação do nosso motor em um SO distribuído que executa em um *lightweight manycore* real proporcionou uma avaliação compreensível dos impactos do motor em diferentes níveis do sistema. De forma geral, os resultados demonstraram que o motor proposto é uma alternativa viável e competitiva às clássicas e custosas abstrações de fluxo de execução. Esta dissertação concentrou-se em consolidar os fundamentos do motor de execução proposto e sua implementação em um contexto do mundo real. Contudo, trabalhos futuros devem investigar aspectos mais detalhados, tais como, (i) aumentar a responsividade do SO removendo a espera-ocupada existente no módulo de comunicação, (ii) implementar políticas de escalonamento mais sofisticados, (iii) substituir *threads* servidoras por *Dispatchers*, aumentando o número possível de serviços simultâneos, e (iv) desenvolver novos serviços baseados em tarefas.

Palavras-chave: Lightweight Manycores. Sistemas Operacionais Distribuídos. Microkernel Assimétrico. Restrições de Memória.

ABSTRACT

Lightweight manycore processors arose to reconcile performance, energy efficiency, and scalability requirements on a single chip. Operating Systems (OSes) for these processors feature a distributed design, where isolated OS instances cooperate to mitigate programmability and portability issues coming from their architectural intricacies. Currently, OS services often resort to traditional execution flow abstractions (processes or threads) to implement small, periodic, or asynchronous functionalities. Although these abstractions considerably simplify the system design, they have a non-negotiable impact on the limited on-chip memories. Due to the memory restrictions, we argue that OS-level abstractions can be reshaped to reduce the OS memory footprint without introducing considerable overhead. In this context, we propose a complementary OS-level execution engine that supports cooperative lightweight tasks that share a unique execution stack and features task synchronization via control flow and dependency graphs. This solution is orthogonal to the underlying execution support and provides numerous OS-level execution flows with reduced memory consumption. We implemented our engine in a distributed OS and executed experiments on a lightweight manycore. Our results show that it has the following advantages when compared to the classical thread abstraction: (i) it provides 63.2 times more execution flows per MB of memory; (ii) it features less overhead to manage execution flows and system calls; (iii) it improves core utilization; and (iv) it exhibits competitive results on real-world applications.

Keywords: Lightweight Manycores. Distributed Operating Systems. Asymmetric Microkernel. Memory Constraints.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

BT	Basic Task.	39
C-NoC	Control NoC.	55
D-NoC	Data NoC.	55
DAG	Directed Acyclic Graph.	40, 41
DRAM	Dynamic Random Access Memory.	33, 55
ECU	Electric Control Unit.	39
ET	Extended Task.	39
FN	Friendly Numbers.	19, 58, 63, 64, 65
GF	Gaussian Filter.	19, 58, 63, 64, 65
HAL	Hardware Abstraction Layer.	34
IKC	Inter-Kernel Communication.	48, 53, 56
IPC	Inter-Process Communication.	48, 63
kB	Kilobyte.	33, 36, 50, 59, 62
KM	K-Means.	19, 58, 63, 64, 65
MB	Megabyte.	30, 33, 55, 57, 59, 67
MIMD	Multiple Instruction Multiple Data.	29, 33
MMU	Memory Management Unit.	40
MPI	Message Passing Interface.	19, 27, 30, 53, 55, 56, 58, 59, 63, 64
MTSS	Multitask Stack Sharing.	40, 41
NoC	Network-on-Chip.	29, 33, 48, 55, 56
NUMA	Non-Uniform Memory Access.	43
OS	Operating System.	17, 19, 29, 30, 33, 34, 35, 36, 39, 40, 41, 43, 45, 46, 47, 49, 50, 53, 55, 56, 57, 59, 60, 63, 67, 68
PE	Processing Element.	55
POSIX	Portable Operating System Interface.	30, 34, 49, 53

Q1	Question 1.	55, 57
Q2	Question 2.	55, 57
Q3	Question 3.	55, 57, 58
Q4	Question 4.	55, 58
RM	Resource Manager.	55
RTOS	Real-Time Operating System.	36, 37, 39
SET	Semi-Extended Task.	39, 40, 41
SMP	Symmetric Multiprocessing.	43
SRAM	Static Random Access Memory.	33, 55, 57
SRP	Stack Resource Policy.	37
TLB	Translation Lookaside Buffer.	33, 55
WCSC	Worst-Case Stack Consumption.	37

LIST OF SYMBOLS

$=$	Equality
$\approx$	Approximately
$+$	Sum
$\times$	Multiplication
$\{\}$	Set
$()$	Tuple
$\mid$	Such that
$\wedge$	And
$\in$	Set membership
$\subseteq$	Subset
$P(D)$	Digraph property

CONTENTS

1 INTRODUCTION

Lightweight manycores stand out for their high degree of parallelism with low energy consumption (FRANCESQUINI et al., 2015). To achieve high scalability and energy efficiency, lightweight manycores feature architectural characteristics that differ from other manycores: (i) they integrate thousands of low-power cores with Multiple Instruction Multiple Data (MIMD) capability (ROSSI et al., 2017); (ii) they feature a distributed memory architecture and small on-chip memories shared by tightly-coupled groups of cores (aka *clusters*) (BOHNENSTIEHL et al., 2017); (iii) they expose reliable and fast Networks-on-Chip (NoCs) for message-passing (BOHNENSTIEHL et al., 2017); and (iv) they may offer heterogeneous processing capabilities (DAVIDSON et al., 2018). Some industry-successful examples of lightweight manycores are the Kalray MPPA-256 (DINECHIN et al., 2013) and the Sunway SW26010 (ZHENG et al., 2015).

Distributed Operating Systems (OSes) have been recently proposed to effectively deal with architectural intricacies of these processors, offering a richer execution environment for developers (BOYD-WICKIZER et al., 2008; RHODEN et al., 2011; WISNIEWSKI et al., 2014). Among these proposed OSes, we highlight the multikernel design (WENTZLAFF; AGARWAL, 2009; BAUMANN et al., 2009), where isolated OS instances interact with other system entities through a client-server approach to mitigate programmability and portability issues. Since the on-chip memory is restricted, the most important design goal is to keep the OS kernel small while preserving its most important features.

From this point of view, the *asymmetric microkernel design* (PENNA et al., 2019) stands out as a flexible and scalable solution while presenting a small OS memory footprint. In this design, the OS kernel exclusively runs on one core of the cluster, leaving the remaining ones to general-purpose use, thus reducing the interference of the kernel in user-level software. However, this design is not enough to relieve all architectural issues coming from existing memory constraints.

Specifically, OS services often resort to classical execution flow abstractions, such as processes (LOMET, 1977), coroutines (PAULI; SOFFA, 1980) and threads (BIRRELL, 1989), to implement small, periodic, or asynchronous OS-level functionalities. Although these abstractions considerably simplify the system design, they have a non-negotiable impact on the limited on-chip memories, leaving a small amount of free memory to user applications and, consequently, affecting the software development from the kernel to the application levels. In this context, we argue that OS-level abstractions can be reshaped to reduce the OS memory footprint without introducing considerable overhead.

1.1 TARGET PROBLEM AND PROPOSED APPROACH

To alleviate the memory constraints of lightweight manycores, we propose a complementary OS-level execution engine that supports cooperative lightweight tasks that share a unique execution stack and features task synchronization via control flow and dependency graphs. Our proposal is inspired by solutions established in extremely restrictive environments, but which efficiently deal with memory limitations. However, our main goal is to increase the density of OS-level execution flows with reduced memory consumption, introducing an alternative solution to the expensive traditional abstractions.

1.2 GOALS AND CONTRIBUTIONS

We design the proposed task-based execution engine based on the following properties: (i) **density**: it should allow to increase significantly the number of concurrent flows in the OS; (ii) **lightness**: it should ensure that the OS memory footprint remains small without introducing considerable overhead (iii) **orthogonality**: it should be independent of the underlying native execution support, taking advantage of the pre-existing environment; and (iv) **flexibility**: the solution should co-exist with traditional solutions to support incremental redesign of OS functionalities.

In summary, the design, implementation and improvements of the proposed engine bring the following contributions to the state-of-the-art in OS support for lightweight manycores:

1. A complementary task-based execution engine for distributed OSes tailored to lightweight manycores with limited on-chip memory for implementing trustworthy OS-level functionalities with reduced memory consumption;
2. Implementation of the proposed engine in Nanvix, an open-source Portable Operating System Interface (POSIX)-compliant distributed OS that targets lightweight manycores;
3. Redesign and implementation of trustworthy modules, functionalities and daemons of existing Nanvix services using tasks; and
4. Demonstration of the impacts achieved by the engine through conceptual, synthetic benchmarks and real-world applications.

We compared our solution with the Nanvix native thread abstraction using a set of comprehensive benchmarks and evaluated the impacts on memory, execution time and energy consumption on a silicon lightweight manycore (Kalray MPPA-256). Our results showed that our solution achieves the following improvements: (i) it provides 63.2 times more execution flows per Megabytes (MBs) of memory; (ii) it features less overhead to manage execution flows and system calls; (iii) it improves core utilization; and (iv) it exhibits competitive results when running Message Passing Interface (MPI) applications.

This dissertation reuses text from a paper published by the same author (SOUTO; CASTRO; PENNA, 2021). However, this dissertation includes a more complete and detailed experimental validation section and analysis.

1.3 ORGANIZATION OF THE DISSERTATION

The remainder of this paper is organized as follows. In Chapter 2, we cover the background on lightweight manycore processors and on the asymmetric microkernel design. Subsequently, we discuss related work in Chapter 3. We detail our task-based execution engine and its implementation in Chapter 4. Then, we present our evaluation methodology in Chapter 5, which is then applied in Chapter 6 to evaluate the experimental results. Finally, we draw our conclusions in Chapter 7.

2 BACKGROUND AND MOTIVATION

In this section, we give a brief introduction to lightweight manycores and show how their architectural characteristics affect software development. Then, we present the foundations of state-of-the-art Distributed OSes and highlight how they fail to address memory challenges of lightweight manycores.

2.1 LIGHTWEIGHT MANYCORES AND THEIR INTRINSIC CHALLENGES

Lightweight manycore processors stand out for their high performance and energy efficiency. To this end, they rely on architectural characteristics such as hundreds of low-power cores and a distributed memory system. Some examples of lightweight manycores are Kalray MPPA-256 (DINECHIN et al., 2013) and Sunway SW26010 (FU et al., 2016).

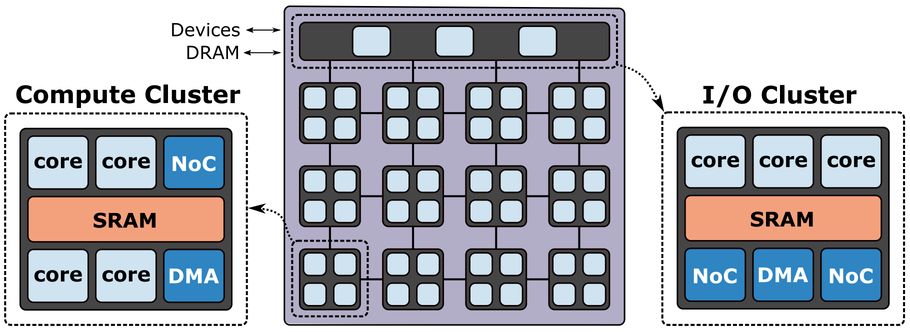
Figure 1 presents a conceptual lightweight manycore, which pictures the key characteristics of this class of processors. It integrates 51 MIMD low-power cores disposed into 13 tightly-coupled groups, named *clusters* (STERLING, 2011). Cores within a cluster share and have uniform access to local hardware resources, e.g., local Static Random Access Memory (SRAM) and NoC interfaces. However, clusters may have heterogeneous resources. For instance, *Compute Clusters* usually feature more complex cores with higher processing power to cope with high computing demands whereas *I/O Clusters* usually have more communication capabilities and connectivity to a Dynamic Random Access Memory (DRAM).

The distributed memory system of lightweight manycores is a crucial feature that allows these processors to scale. It is composed of a set of restricted SRAMs, each one physically located in a cluster. Overall, the distributed memory system: (i) features multiple address spaces; (ii) presents local memories ranging from hundreds of Kilobytes (kBs) to a few MB; (iii) does not support hardware-level cache coherence; and (iv) features a software-managed Translation Lookaside Buffer (TLB) for virtual address translation. Due to the distributed memory system, applications must use a hardware-level message-passing interface to make inter-cluster communications through the NoC.

The aforementioned characteristics enhance the scalability and energy efficiency of lightweight manycores. However, they also introduce several challenges in software development (FRANCESQUINI et al., 2015). Specifically concerning their memory system, we highlight the following issues:

- The *small amount of on-chip memory* obliges software developers to explicitly tile the working dataset into chunks, load/store these chunks from/to remote memory, and manipulate them locally. Additionally, the software must take care of data caching and replication to improve performance.
- The *absence of cache coherence support in hardware* forces software developers to

Figure 1 – A conceptual lightweight manycore.



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

handle data coherency explicitly in software and frequently calls out for application redesign.

- The *distributed memory architecture* requires software to handle data partitioning and remote data accesses across multiple physical address spaces.

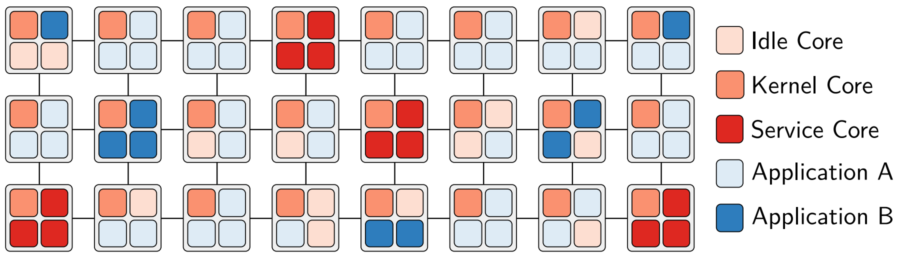
2.2 DISTRIBUTED OPERATING SYSTEMS FOR LIGHTWEIGHT MANYCORES

Lightweight manycores face several programmability and portability challenges, which are inherited from their architectural features. Recent research efforts to mitigate these challenges leverage Distributed OSes (KLUGE; GERDES; UNGERER, 2014; ASMUSSEN et al., 2016; PENNA et al., 2019) to provide a more robust and richer execution environment. In this design, the OS is factored in a set of services, each of which is deployed on a core of the parallel architecture. Cores that do not run OS services are made available to user-level applications, which request their assistance through a client-server interface.

Multiple architectures and implementations for a distributed OS are possible, each one targeting a specific set of design goals and constraints. However, a three-tier approach is commonly adopted by distributed OSes for lightweight manycores such as MOSSCA (KLUGE; GERDES; UNGERER, 2014), M^3 (ASMUSSEN et al., 2016) and Nanvix (PENNA et al., 2019). In the bottom layer, a generic and flexible Hardware Abstraction Layer (HAL) is provided to enable portability across different processor architectures. A *microkernel* lies in the middle layer and provides minimum system abstractions, handles local resource multiplexing and ensures security policies. Finally, in the top layer, runtime OS libraries expose a standard interface to user-level applications such as POSIX.

OS kernel instances may run in all processor cores or a subset of them (symmetric vs. asymmetric design) (PENNA et al., 2020). In this dissertation, we are interested in *asymmetric multikernel OSes* due to their outstanding performance isolation between kernel and user spaces (NIGHTINGALE et al., 2009). Figure 2 presents a snapshot of

Figure 2 – Hypothetical distributed OS on a lightweight manycore.



Source: Adapted from Penna et al. (2020).

an asymmetric distributed OS running in a lightweight manycore. Cores within the same cluster of the processor share an OS kernel instance. The OS kernel (*master thread*) runs on a dedicated core (*master core*) of the cluster. The remaining cores are available for OS services and user applications. In this design, kernel and user do not time-share core hardware structures, delivering performance isolation. Moreover, there is no contention in the structures of the OS kernel.

The disadvantage of the asymmetric design is that one core must be reserved to execute the OS kernel. This requirement decreases the number of cores available for user threads. Therefore, if the OS kernel is not heavily used, the overall performance of the system is reduced. This problem becomes even worse when cores are also reserved to execute OS services. A more detailed evaluation can be found in Penna et al. (2020).

2.3 PROBLEM DEFINITION

The memory subsystem of lightweight manycores imposes challenges on the design and implementation of OSes for these architectures. The asymmetric microkernel alleviates the problems of cache coherence and pollution, since user and kernel working datasets are well separated and thus cache utilization is improved (PENNA et al., 2019). However, this is not enough to relieve all architectural issues that concern the limited amount of on-chip memory available.

We argue that OS-level abstractions can be reshaped to reduce the OS memory footprint and, consequently, make more room for user-level software (CAI; ZHANG; HUANG, 2020; Zhang et al., 2020; LIU et al., 2014). For instance, OS services often resort to classical execution flow abstractions, such as processes (LOMET, 1977) and threads (BIRRELL, 1989), to implement small, periodic, or asynchronous functionalities, even though they spend most of their lifetime blocked/sleeping. Although this simplifies the system design, it has a non-negotiable impact on memory consumption. The main reason is the significant waste of memory by the commonly used stack layout for multitasking systems, named *cactus stack* (SARDESAI; MCLAUGHLIN; DASGUPTA, 1998; PIZKA, 1999). In this layout, the kernel needs to book memory pages to accommodate separate stacks for each existing thread in the system, limiting the maximum density of

Table 1 – Hard limit on the number of threads per cluster.

Processor	**Memory size** (per cluster)	**# of Cores** (per cluster)	**# of threads** (per core)	**# of threads** (per cluster)
PULP (ROSSI et al., 2017)	32 kB	4	2	**8**
CoreVA (Ax et al., 2018)	256 kB	16	4	**64**
Sunway SW26010 (ZHENG et al., 2015)	2048 kB	64	8	**512**
Kalray MPPA-256 (DINECHIN et al., 2013)	2048 kB	16	32	**512**

Source: Developed by the Author.

flows. In a simple case, in which each thread takes exactly one page to keep its stack, the amount of memory reserved would be proportional to the number of threads.

Table 1 summarizes what would be the hard limit on the number of threads per Compute Cluster in different state-of-the-art lightweight manycores when considering a single page of 4 kB per stack frame. Note, however, that these limits were computed based on the strong conservative assumption that the whole memory in a cluster is available for keeping stacks of threads, which is rather unrealistic. In practice, we should account for kernel and user stack frames, other system data structures and user allocated memory. In this context, the development of OS-level software is restricted to a short portion of the limited amount of execution flows so as not to reduce the parallelism of applications.

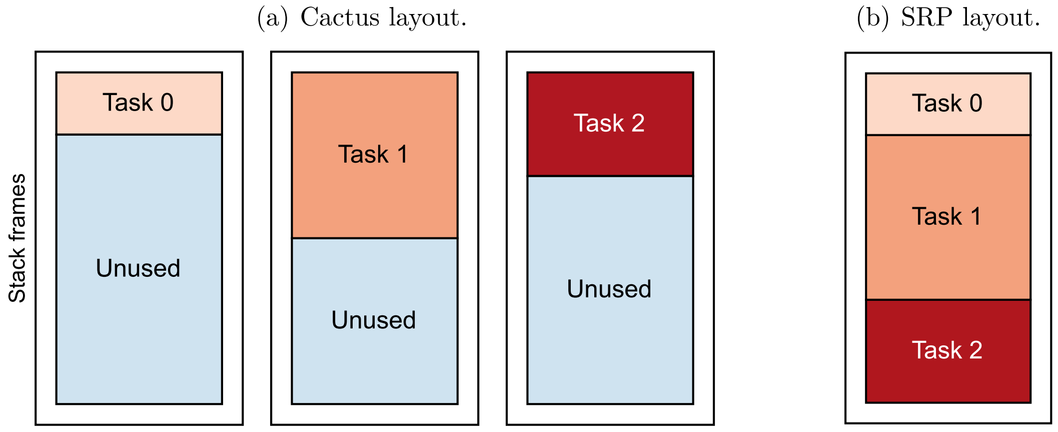
2.4 MOTIVATION

Over the years, the evolution of edging technology has brought a wide range of architectures that share resource-constraint outlines. Out of the data center and the desktop environment, these architectures arrive to provide efficient solutions over extremely resource-constrained conditions, such as real-time embedded systems. Thereby, we can analyze the clusters of a lightweight manycore from the same perspective of resource limitations, relating them to well-known proposed solutions. However, our intention is not to compete with these systems but to bring the discussion about resource scarcity to general-purpose environments.

Real-Time Operating System (RTOS) is an OS type commonly used to deal with the constraints of the embedded systems and presents a comprehensive number of studies on resource-optimization topics. Particularly, a RTOS runs applications with critical time constraints to process data or events. Furthermore, RTOSes should incur low processing overhead and memory usage for this type of architecture (PARK et al., 2011). The time constraints imposed on the environment require that critical tasks have deterministic response time and memory consumption to be possible to analyze and guarantee the strict response deadlines. Furthermore, decoupling applications into multiple tasks improves the development process, thereby increasing system density.

Several approaches have been proposed to deal with the processing, memory, and energy consumption constraints present in real-time embedded systems. Particularly,

Figure 3 – Common stack layouts.



Source: Developed by the Author.

established solutions that mitigate the memory-constraints impacts on OS and user levels motivated us to explore their benefits under a general-purpose OS running on lightweight manycores.

One of the simplest ways to keep the RTOS footprint small is to share the same stack frame to execute different tasks. In this way, the size of this region can be equal to the largest execution stack among the tasks. However, critical tasks must execute as soon as possible, turning impossible to serialize executions. In this context, Stack Resource Policy (SRP) (GAI; LIPARI; NATALE, 2002) allows the preemption of tasks, stacking their contexts in the same memory frame. In this way, the RTOS dispenses the cactus layout, efficiently using a single memory region to support the same number of tasks.

Figure 3 illustrates two stack layouts in an environment with three tasks. While the cactus layout (Figure 3(a)) requires three separate stack frames to keep the tasks' contexts, the SRP layout (Figure 3(b)) requires only one. The technique used to co-allocate different tasks/threads on a single stack at the same time is called Worst-Case Stack Consumption (WCSC) (DIETRICH; LOHMANN, 2018). Additionally to the SRP, Non-Preemption Groups (DAVIS; MERRIAM; TRACEY, 2000) considers different event priorities and introduces preemption levels to limit the number of contexts that can be stacked. This solution further reduces the stack frame, demanding only the sum of the largest contexts of each possible level.

In addition to stack sharing, the memory optimization of other system parts presents promising results. For example, the analysis and profiling of machine learning applications together with the execution environment can reduce the overall memory footprint (KATSARAGAKIS et al., 2020). This reduction is achieved by (i) replacing static memory management with dynamic, exploiting heap reuse; (ii) removal of unused static data; and (iii) reducing the code hierarchy, removing unused functions. However, such optimizations take into account a dedicated environment adapted to a finite set of applications.

In general, the way developers design and establish memory management policies influence the cost and behavior of the computational environment (BATENI et al., 2020). Furthermore, memory impositions are becoming even more evident as computer systems are increasingly parallel and distributed. This fact is known as the memory wall (WULF; MCKEE, 1995). The memory wall is one of the main motivations that led to the arrival of lightweight manycores. Consequently, it is mandatory to study and explore ideas to efficiently deal with these resource-constrained architectures in face of complex general-purpose OS functionalities.

3 RELATED WORK

Execution support has been an extensively studied topic over the past few years. From low-level execution support to high-level execution runtimes, these studies have contributed to increasing the performance and programmability of computer systems. In this chapter, we discuss significant established solutions at the OS level. Furthermore, we compare these solutions against our proposal and show how our engine can complement them.

Within the scope of general-purpose distributed OSes, Scheduler Activations (ANDERSON et al., 1992) provide means of forwarding kernel-level preemption or scheduling actions to the user-level scheduler in an efficient way through *upcalls*. Although this support has performance and control benefits, it still requires a stack frame per thread to keep the states of concurrent execution flows. In this context, our approach is easily switchable to work in conjunction with Scheduler Activations due to its orthogonality. In particular, we would benefit from the scheduling decisions being handled directly by our engine executor instead of using two levels of scheduling as in other environments.

Despite being in a very different scope, the memory limitation in real-time embedded systems also impacts how RTOSes define and design their execution flow abstractions. For instance, OSEK OS (PARK et al., 2011) is a standard RTOS for Electric Control Unit (ECU) software for vehicles originally designed for extremely resource-constrained environments. To deal with the limited memory on the embedded systems, OSEK OS specifies hybrid execution support to distinguish between Basic Task (BT) and Extended Task (ET). On the one hand, BTs are non-blocking tasks that share a single stack frame. On the other hand, ETs allocate a dedicated stack to allow blocking behaviors. Also, OSEK OS reduced the complexity and memory requirement of the scheduling queue considering aspects of different task classes.

Semi-Extended Tasks (SETs) (DIETRICH; LOHMANN, 2018) expand the expressiveness of the ETs in the OSEK OS model. SETs allow decreasing the execution flow granularity from task to function context. Thereby, SETs start with their private stack (such as ET) but switch to BT behavior whenever it is possible. Likewise, this optimization allows the execution of blocking function/system calls by switching context to non-conflicting functions. Similarly, HEROS OS (LIU et al., 2014) explores a hybrid scheduling model via a subfunction-granularity thread switch mechanism, distinguishing real-time from event-oriented tasks to reduce memory consumption. However, this mechanism uses low memory Java threads, imposing some of the overhead of the Java environment.

The solution proposed in this dissertation and the aforementioned stack-sharing solutions share a similar goal, the reduction of memory consumed by execution supports. However, RTOSes are much more restricted than general-purpose OSes, allowing a detailed analysis of memory usage and execution of real-time tasks. In this sense, despite the benefits of task preemption, it would be impractical to define task relationships in the context of a general-purpose OS. Nevertheless, our proposed engine complements these execution supports by introducing an explicit form of synchronization between tasks, opening up possibilities for optimization in scheduling decisions. Moreover, our engine can introduce hybrid scheduling for OSes that only implements a cactus stack layout.

Although previous solutions reduce the OS memory footprint, they require significant modifications. In contrast, some solutions rely on distinct hardware features to minimize kernel adaptation. For instance, Multitask Stack Sharing (MTSS) (MIDDHA; SIMPSON; BARUA, 2008) proposed a new paging system to share a common memory region to allocate all private tasks. This paging system does not reduce the number of stacks, but identifies stack overflow and allocates unused space in other stack frames. Similarly, StackMMU (MAURONER; BAUNACH, 2017) handles the stack growth, shrinkage, and access operations through the Memory Management Unit (MMU) functionality. Both solutions demand extra memory and a specific hardware component to work, but only the memory module holds the majority of the required modifications. In this regard, our proposal requires just a few modifications for its kernel integration. Moreover, our engine demands common hardware attributes, being able to profit from sophisticated hardware components.

On top of lower-level solutions, data-flow and continuation-based programming models proved to be adequate to express data dependency between execution flows and align with memory reduction goals. For instance, CEFOS (KUSAKABE et al., 2007) is an OS that defines continuation-based fine-grained threads to implement execution flows. This model can benefit from stack sharing because threads are non-preemptive and the data-flow dependencies guarantee the correctness of the application. Likewise, a Memory-Aware Directed Acyclic Graph (DAG) model (MARCHAL et al., 2018) defines computational workflows used for parallel scheduling of tasks under memory constraints. This DAG model reduces the maximum memory peak through memory dependencies. These dependencies prevent the scheduler from running out of memory during the execution.

Our solution presents the same benefits as data-flow and continuation-based solutions. However, our proposal differs on two points: (i) we seek to complement the underlying execution support, where only a portion of OS functionalities need to be remodeled, whereas the entire development environment revolves around the data flow design; and (ii) we provide security guarantees over the stack by restricting the use of the engine by trustworthy functionalities, avoiding unnecessary overhead for stack sharing protection.

Table 2 summarizes the comparison of the related work based on some important criteria. Overall, our solution tackles memory consumption issues in architectures with limited on-chip memory. Unlike OSEK OS, SETs and HEROS OS, we focus on reducing the number of stack frames rather than reducing the stack size. Although CEFOS and Memory-Aware DAG model would benefit from the same reduction achieved by our

Table 2 – Comparison between related work on dissertation-related criteria.

Related Work	General-Purpose OS	Memory-constrained Systems	Complementarity
Scheduler Activations (ANDERSON et al., 1992)	✓	✗	✗
OSEK OS (PARK et al., 2011)	✗	✓	✗
SETs (DIETRICH; LOHMANN, 2018)	✗	✓	✗
HEROS OS (LIU et al., 2014)	✗	✓	✗
MTSS (MIDDHA; SIMPSON; BARUA, 2008)	✗	✓	✓
StackMMU (MAURONER; BAUNACH, 2017)	✗	✓	✓
CEFOS (KUSAKABE et al., 2007)	✗	✓	✗
Memory-Aware DAG (MARCHAL et al., 2018)	✗	✓	✗
Proposed Engine	✓	✓	✓

Source: Developed by the Author.

solution, it would fall into security issues by not being restricted to trustworthy applications. Moreover, our solution addresses general-purpose OSes for lightweight manycores, which are much more complex than other solutions that are designed for specific domains (OSEK OS, SETs, HEROS OS, CEFOS, and Memory-Aware DAG) or demand specific hardware features (MTSS and StackMMU). Finally, to the best of our knowledge, our solution is the only one capable of co-existing with the default execution supports available for distributed OSes, thus avoiding the need for a complete redesign of OS-level code.

4 TASK-BASED EXECUTION ENGINE

Lightweight manycores present a simplified on-chip memory system compared to Symmetric Multiprocessing (SMP) and Non-Uniform Memory Access (NUMA) architectures. Despite the relatively small amount of memory per thread, the constrained memory system allows for better energy efficiency and scalability. However, lightweight manycores impose several programmability and portability issues to software design. As discussed in Section 2.2, existing distributed OSes based on asymmetric design stand out for their adherence to the distributed and restrictive nature of lightweight manycores but still fail to address their memory restrictions.

We believe that classical kernel data structures and abstractions can be reshaped to reduce OS memory footprint. In this dissertation, we look at the opportunity to reduce the memory consumption of trustworthy OS functionalities that execute small, periodic, or asynchronous operations, such as OS service protocols, daemons, kernel-level communications, and I/O interfaces. We revisit the concept of *execution flow* to propose a lightweight task-based execution engine that complements the underlying execution support for state-of-the-art distributed OSes tailored to lightweight manycores with limited on-chip memory. Overall, our solution aims at the following goals:

- Increase the density of concurrent OS-level execution flows within a lightweight manycore cluster;
- Ensure that the OS memory footprint remains significantly small;
- Improve small, periodic, or asynchronous trustworthy OS functionalities;
- Avoid significant overhead; and
- Make better use of the core that run the asymmetric microkernel.

In this chapter, we cover the main aspects of our solution. First, we present the concept of *lightweight execution flow*. Then, we present our engine and show how it addresses the memory-related issues and complements the underlying execution support. Finally, we detail its implementation in an open-source distributed OS that runs on a silicon lightweight manycore and we illustrate how we implemented important OS functionalities on top of it.

4.1 COOPERATIVE LIGHTWEIGHT TASKS

The *execution flow* is the base element in any OS. Over the years, different granularities of flows have been proposed such as processes (LOMET, 1977), threads (BIRRELL, 1989), and coroutines (PAULI; SOFFA, 1980). Typically, an execution flow contains basic information about the execution of a program, e.g., instruction pointer, data stored on registers, working variables, and execution stacks. The stack, in particular, holds the call history of procedures and allows several flows to share the same core. However, the amount of memory needed to keep execution stacks is non-negotiable for systems with limited on-chip memory.

In order to provide more execution flows with reduced memory consumption, we propose an approach that reshapes the execution flow concept to reuse a unique stack. To do so, we decompose execution flows into *control flow and dependency graphs of cooperative lightweight tasks*. For the sake of simplicity, we will use the term “task” to refer to “cooperative lightweight task” unless explicitly stated otherwise.

A task is the basic unit of execution and represents a processing step of a flow. It encapsulates a subroutine or block of operations with deterministic response time (which can take input arguments and produce output values) and can be periodically rescheduled. Combining these concepts, we build the complete state of an execution flow by synchronizing a set of tasks.

Task synchronization via control flow and dependency graphs plays a fundamental role within the proposed engine. First, it implicitly preserves the execution history of the functionality, allowing the use of a single stack to interleave executions of non-dependent tasks. Second, functionalities that need to wait for events can block specific tasks to stop competing for shared resources. Finally, relationships between tasks explicitly define preemption points.

4.1.1 Control-Flow and Dependency Graphs

To structure the complete state of an execution flow, we use a multi-labeled directed graph. It is defined by a quintuple that specifies sets for vertices (V), arcs (A), and labels (T, L, and C). Specifically, the following equations define the digraph.

$$D = (V, A, T, L, C) \tag{4.1}$$

$$V(D) = \{v \text{ is a task} \mid v \text{ has deterministic response time}\} \tag{4.2}$$

$$A(D) = \{(u, v, t, l, c) \mid u, v \in V \wedge t \in T \wedge l \in L \wedge c \subseteq C\} \tag{4.3}$$

$$T(D) = \{t \in \{\text{flow}, \text{dependency}\} \mid t \text{ defines the relationship type}\} \tag{4.4}$$

$$L(D) = \{l \in \{\text{persistent}, \text{temporary}\} \mid l \text{ defines the lifetime of the arc}\} \tag{4.5}$$

$$\begin{aligned} C(D) = \{c \subseteq \{&\text{regular conclusions}, \text{rescheduling}, \text{error handling}\} \\ &\mid c \text{ are possible conditions to select the arc}\} \end{aligned} \tag{4.6}$$

The arc-related labels increase the expressiveness in the execution flow definition using tasks. Specifically, the *type* attribute (Equation 4.4) combines control flow and data dependency concepts. *Flow arcs* describe all possible paths of a flow, making it possible to create control flows such as conditional constructs (*ifs*) and *loops*. *Dependency arcs* add the concept of *task dependency* to the *flow arcs*, meaning that all dependencies of a task must be met to unblock it.

The *lifetime* attribute (Equation 4.5) provides two types of interactions between tasks. *Persistent arcs* define permanent relationships between tasks whereas *temporary arcs* specify brief relationships between tasks that are valid only once.

The *conditions* attribute (Equation 4.6) establishes control flow and management options based on the conclusion of a task. They describe the subset of paths (arcs) to be released, allowing control flow to exist (path choice). In addition, conditions indicate how the current task should behave once it is concluded. For instance, it is possible to determine if the current task shall be automatically rescheduled and/or if it shall release its successor tasks.

4.1.2 Execution Engine

The execution engine operates through a generic task executor called *Dispatcher*. It schedules tasks, one at a time, and executes them on the same reserved memory (i.e., the same stack). Thus, we avoid using a dedicated process or thread for each task, which reduces the OS memory footprint and makes their creation/destruction extremely fast.

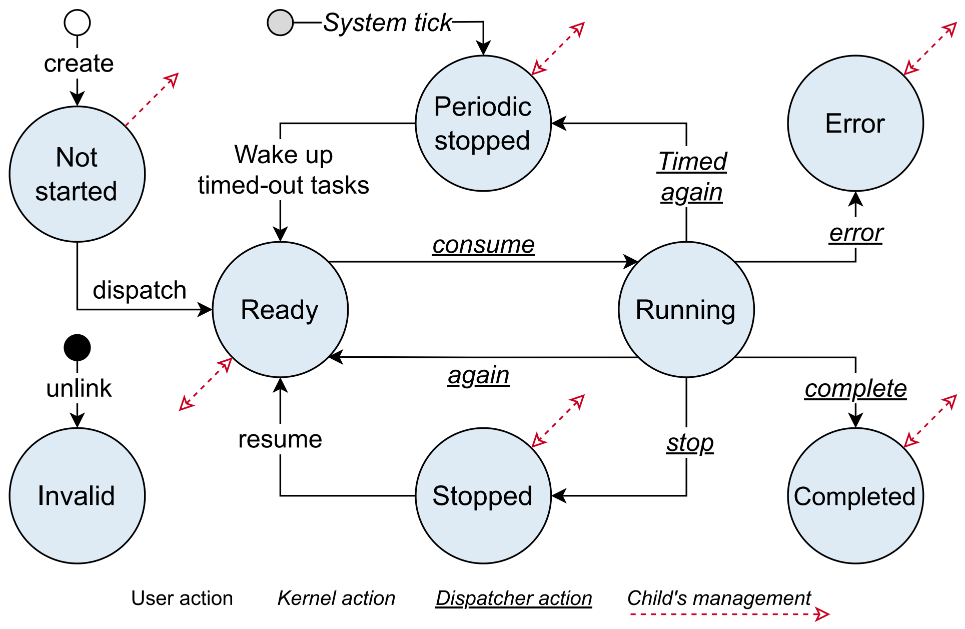
We highlight that our execution engine *complements* the native OS abstractions and is built on top of them. Because of that, it can provide more or less advanced features depending on the underlying execution support provided by the OS. For instance, the *Dispatcher* can benefit from finer scheduling control decisions if the OS allows such flexibility (ANDERSON et al., 1992; BAUMANN et al., 2009).

Figure 4 depicts the states of tasks from the execution engine perspective and the respective management interface. Tasks are configured/destroyed (*create*/*unlink*) individually and connected/disconnected (*connect*/*disconnect*) to/from a graph externally to the *Dispatcher* functionality. Tasks without predecessors should be explicitly dispatched to ensure that only configured tasks are treated (*dispatch*). The *Dispatcher* processes tasks through a producer-consumer approach (*consume*). Different scheduling strategies can be used to improve task throughput depending on the underlying OS execution support.

When a task completes, it forwards the exit conditions and successor parameters to the *Dispatcher*. Based on the conditions, the *Dispatcher* does one of the following actions: (i) complete the task correctly (*complete*); (ii) abort the task and propagate the error (*error*); (iii) reschedule the task immediately (*again*) or after a predefined period (*timed again* and *wake up*); or (iv) block the task (stopped) waiting for an external signal (*resume*).

In all task conclusions, the *Dispatcher* can change the state of successor tasks if they fulfill an exit condition and no longer have dependencies. However, the subsequent

Figure 4 – States of tasks form the execution engine perspective.



Source: Developed by the Author.

state of each successor task depends on its internal criterion, which is defined when the task was created.

4.2 IMPLEMENTATION DETAILS

The proposed task-based execution engine is generic enough to be implemented in any distributed OS that targets lightweight manycores. In this dissertation, we chose to implement it in Nanvix,[1] since it is an open-source distributed OS that runs on a silicon lightweight manycore. Similar to other distributed OSes for lightweight manycores, Nanvix is structured in three logic layers. Our implementation lies in the middle layer (within the asymmetric microkernel), which provides a fixed number of threads within a cluster.

4.2.1 Overview of the Task-based Engine

The execution engine consists of four elements: the task structure, management queues, arc-related labels and the *Dispatcher*. Table 3 shows the attributes of a task. The task is notably compact and occupies only 128 bytes. We introduced variables to manage and keep information about: (i) scheduling decisions; (ii) synchronization; and (iii) graph control variables.

The task management is composed of three linked queues. Each queue represents a possible waiting state (i.e., *ready*, *periodic stopped* or *stopped*). The *ready queue* is a

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

Table 3 – Attributes of a lightweight task.

Group	**Attributes**	**Size**
Internal Control	Flags, priority, identifier and state	19 B
Execution Parameters	Function pointer, arguments and return values	28 B
Schedule Decision	Period and next state's criterion	8 B
Waiting Control	Kernel semaphore and triggers	25 B
Graph Control	Predecessors/successors control and arc attributes	48 B
	Total	128 B

Source: Developed by the Author.

priority queue that contains all executable tasks. A *delta queue* manages periodic tasks and sorts them based on period decomposition. Finally, the *stopped queue* holds tasks that are waiting for external events to progress.

We implemented the arc-related labels defined in Equations 4.4, 4.5 and 4.6 as follows. The *type* and *lifetime* sets follow the same values specified in Equations 4.4 and 4.5. However, we only support one arc type at a time between directed pairs for simplicity (Ⓐ → Ⓑ and Ⓑ → Ⓐ are distinct pairs). The *conditions* set has the following possible values: (i) three types of *regular* paths to support generic control flow structures; (ii) three rescheduling types (*again*, *stop*, and *periodically stop*); and (iii) two error handling options (*throw*/*catch*). The error handling option describes the relationship between two tasks but both are selected on an error condition.

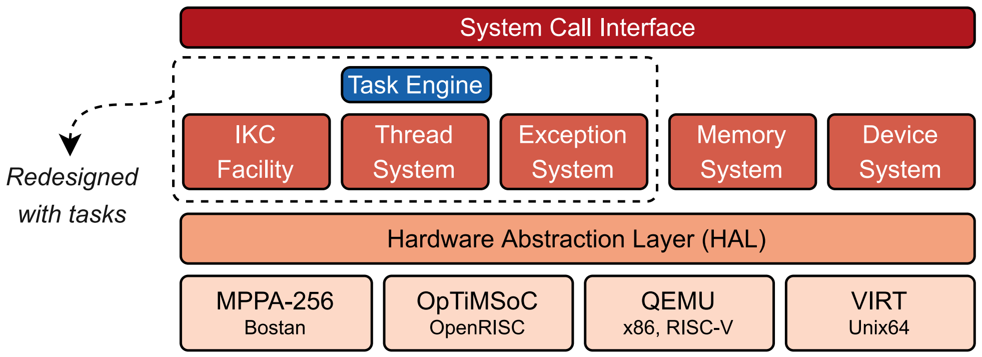
Since a master thread exclusively serves system calls isolated on a dedicated core, OS-level functionalities and user applications compete for the rest of the available cores. In this context, we chose to explore the idle time among system calls allocating the *Dispatcher* in the same OS-dedicated core. This decision seeks to minimize kernel interference and avoid reserving another core for the OS. As we will show in the results, this design choice allows us to optimize the use of the master core. To do so, we had to implement thread preemption in Nanvix from scratch.

4.2.2 Nanvix Microkernel Redesign

The Nanvix microkernel consists of modules to provide the basic OS abstractions. These functionalities are exported through system calls and complemented by runtime libraries. Figure 5 shows an overview of the Nanvix microkernel. Our engine, built from scratch, lies on top of the execution support provided by the thread system. Besides the execution support, our engine also supports *exception* and *communication* modules using tasks.

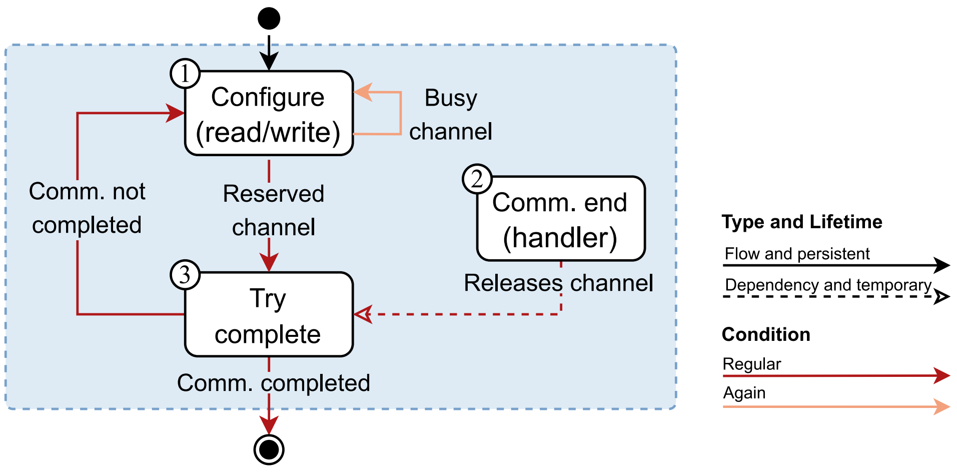
The *exception module* is responsible for providing a user interface to dispatch exceptions to other system entities. For instance, the page fault handler can delegate the work to an OS daemon that implements a shared and distributed memory system. The

Figure 5 – Conceptual overview of the Nanvix Microkernel.



Source: Developed by the Author.

Figure 6 – Generic communication flow with tasks.



Source: Developed by the Author.

original exception module blocks the handler on a semaphore and releases the daemon to handle the page fault. Our solution replaces the daemon thread with tasks that only exist during the exception handling. On completion, we release the user exception handler.

The *Inter-Kernel Communication (IKC) module* (PENNA et al., 2021), which is responsible for implementing the abstractions for Inter-Process Communication (IPC) in Nanvix, is more complex because it uses virtual channels to multiplex the limited NoC channels. Requesters must wait for arriving messages on *receives* or permissions to transmit on *sends*. When the operation completes, the kernel wakes up the blocked requester. Receivers consume only messages addressed to their virtual channel. If the physical channel is already reserved, the requester waits in a busy-wait fashion. If a message is larger than the supported size, the runtime breaks it into chunks and repeats the communication procedure. More details about the IKC module can be found in Penna et al. (2021).

Figure 6 illustrates how we modeled a generic *communication flow* in Nanvix with tasks. The *communication flow* consists of two persistent tasks (Task ① and ③) and a temporary task (Task ②). We isolate the allocation/configuration of the physical channel (Task ①) and consumption/release of used resources (Task ③) from the waiting operation (Task ②). Two flow arcs connect Task ① and ③, allowing channel multiplexing and large transfers to be carried out.

Task ② is a global task exclusively related to a physical channel. On the physical channel reservation, Task ① connects ② to ③ on demand. This temporary arc represents the dependency between Task ③ and the conclusion of the physical communication. On the communication conclusion, a handler dispatches Task ②, which, in turn, releases Task ③. The physical channel reservation and the temporary connection guarantee that only one execution flow is notified of the conclusion.

To support the communication execution with tasks, we only modified the communication module to abstract how the handler notifies the waiting entity (thread or task). Although we still do a *busy-wait* to reserve the channel, we prioritize releasing resources by setting Task ② and ③ as high priority tasks.

4.2.3 Nanvix Asymmetric Multikernel Redesign

Nanvix implements an asymmetrical multikernel environment composed of services and abstractions compatible with the POSIX standard. This design decision seeks to balance programmability, portability, and performance goals to lightweight manycores and support existing applications. Currently, Nanvix consists of three services that use dedicated execution flows beyond the abstractions provided by the modules:

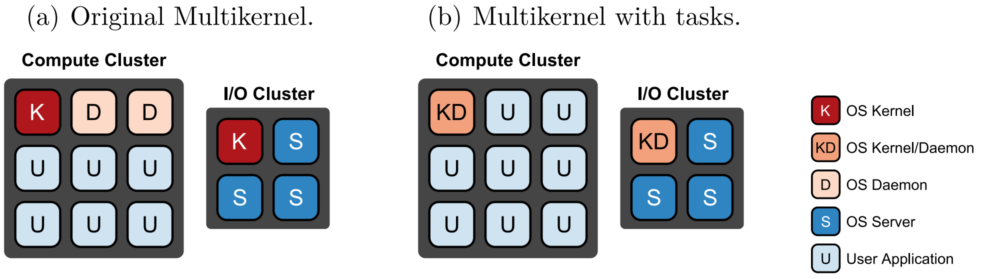
NameServer enables a transparent placement of processes and OS services on the lightweight manycore. It maintains a table of aliases that maps a unique logical identifier of a process to the cluster where this process resides and can resolve an alias into the location of the process.

RMem provides a shared memory abstraction over multiple address spaces. It supports transparent data access at the OS level through system calls that enable *one-sided communication* on top of this abstraction (PENNA et al., 2019).

SHMem maintains the shared page mappings in the system and coordinates the access to shared pages. Built on top of the RMem, it keeps track of ownership and access permissions in shared memory regions. Whenever a change occurs in a shared page, the service broadcasts a *page invalidation* signal to all processes that share the corresponding page. This service is part of the *SysV* interface.

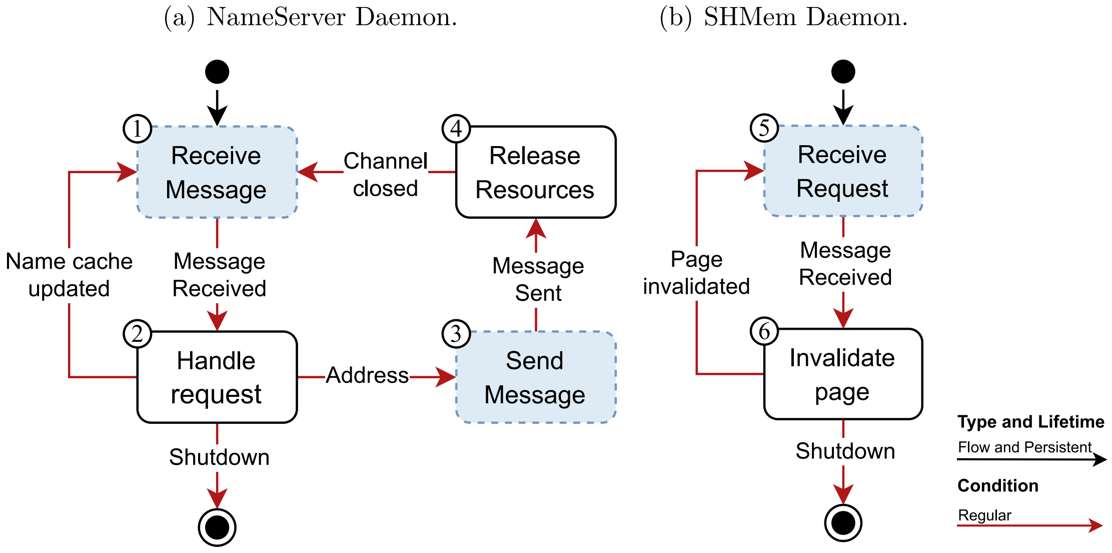
Figure 7 compares the original Nanvix multikernel with its new task-based implementation. In both solutions, the server threads are isolated on I/O Clusters and fulfill requests from other system entities. Compute Clusters, in contrast, are more memory-restrictive and need to share resources between the kernel and the user application. In its original version (Figure 7(a)), a dedicated core is reserved to each OS daemon. Our task-based solution (Figure 7(b)) uses a single core of the Compute Cluster, making more

Figure 7 – Conceptual overview of the Nanvix Multikernel environment.



Source: Developed by the Author.

Figure 8 – Definition of the NameServer and SHMem Daemons with tasks.



Source: Developed by the Author.

cores available to the user applications. The memory consumption is also reduced, since each user and kernel stacks of each OS daemon occupies 4 kB in Nanvix.

We now detail how we implemented three Nanvix daemons (NameServer, RMem and SHMem) with tasks. For the sake of simplicity, we abstract the communication flows (blue box) as a unique box that follows the description given in Section 4.2.2.

4.2.3.1 ***NameServer Daemon***

Algorithm 1 details the behavior of NameServer. This daemon receives (line 3) two types of lookup requests (*local* and *remote*) in an infinite loop. *Local lookups* handle name cache misses as follows: it redirects the request to the NameServer (lines 6 - 10); then, it updates the name cache with the lookup response; and, finally, it releases the user thread (line 5). *Remote lookups* are similar to local ones but they target another name daemon to extract extra information about a process (lines 6 - 10).

Figure 8(a) shows the graph representation of NameServer. Task ① and Task ③ comprise the communication logic presented in Figure 6. Task ② encompasses all de-

Algorithm 1 – Abstract behavior of the NameServer Daemon.

```
Require: $I \leftarrow$ Configured input channel
1: procedure NameDaemon
2:     while True do                                          ▷ Steps
3:         $M \leftarrow$ ReceiveMessage($I$)                    ▷ 1
4:         if $M_{operation} = (Success$ or $Fail)$ then          ▷ 2
5:             UpdateNameCache($M$)                            ▷ 2
6:         if $M_{operation} = Address$ then
7:             $R \leftarrow$ BuildMessage($M_{type}$)            ▷ 2
8:             $O \leftarrow$ OpenChannel($M_{target}$)           ▷ 2
9:             SendMessage($O, R$)                              ▷ 3
10:            CloseChannel($O$)                                ▷ 4
11:        if $M_{operation} = Shutdown$ then                   ▷ 2
12:            $Break$                                          ▷ 2
```

Source: Developed by the Author.

Algorithm 2 – Abstract behavior of the SHMem Daemon.

```
1: procedure ShmDaemon
2:     while True do                                          ▷ Steps
3:         $R \leftarrow$ ReceiveRequest($I$)                    ▷ 5
4:         if $R_{operation} = Shutdown$ then                   ▷ 6
5:             $Break$                                          ▷ 6
6:         ShmInvalidatePage($Ri_{addr}$)                       ▷ 6
```

Source: Developed by the Author.

terministic operations between two communications except closing the required opened channel, which is done by Task ④ to complete the scope of the request.

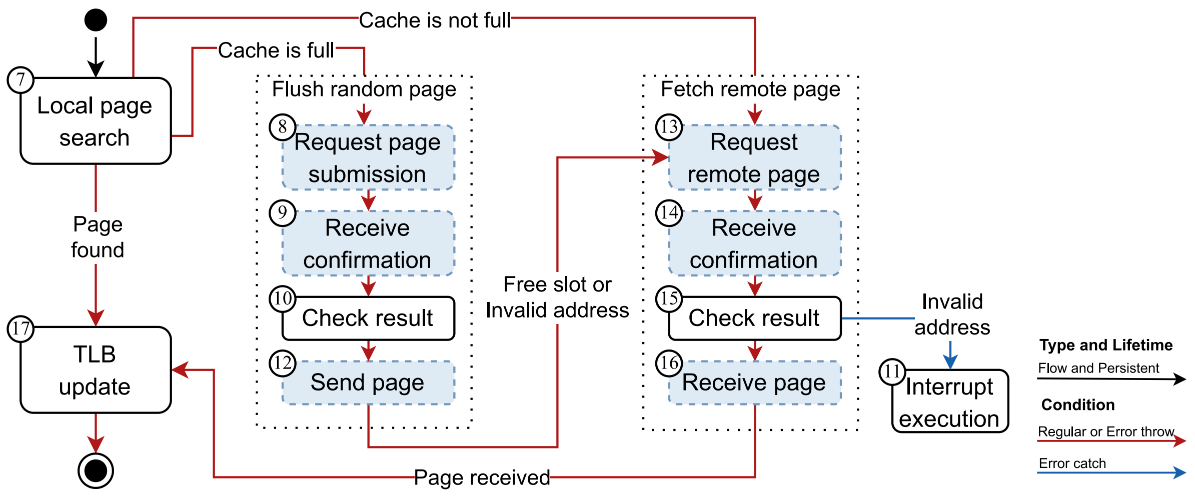
4.2.3.2 ***SHMem Daemon***

Algorithm 2 and Figure 8(b) presents the behavior and the graph representation of SHMem, respectively. This daemon waits for page invalidation requests (line 3) and invalidates the corresponding pages (line 6). It is composed of a communication flow (Task ⑤) and a deterministic task to invoke the invalidation protocol (Task ⑥). Similar to the previous flow, no dependency arcs than those contained in the communication flow are required.

4.2.3.3 ***RMem Daemon***

Algorithm 3 outlines the procedures of the RMem. This daemon uses the exception module to handle page faults, intermediating the communication with the RMem server, and updates the memory structures. It searches the relative page on the local cache (line 3). If the page is not found and the cache is full, it flushes one page to the remote server (lines 4 - 11). With a free slot in the cache, it fetches the required page

Figure 9 – Definition of the RMem Daemon with tasks.



Source: Developed by the Author.

Algorithm 3 – Abstract behavior of the RMem Daemon (page fault handler).

```
 1: procedure RMEMHANDLER                                          ▷ Steps
 2:     $Ex \leftarrow$ GETPAGEFAULTEXCEPTION                               ▷ 7
 3:     if SEARCHPAGECACHE($Ex_{addr}$) == NotFound then                   ▷ 7
 4:         if PAGECACHEISFULL then                                        ▷ 7
 5:             $p_i \leftarrow$ RELEASEONEPAGE                   ▷ Flush one page : 7
 6:             $Ru \leftarrow$ BUILDFLUSHREQUEST($p_i$)                        ▷ 7
 7:             SENDREQUEST($Ru$)                                          ▷ 8
 8:             $Ru \leftarrow$ RECEIVECONFIRMATION                             ▷ 9
 9:             if $Ru_{result} = InvalidPage$ then                        ▷ 10
10:                 DOPANIC                                                ▷ 11
11:             SENDPAGE($p_i$)                                            ▷ 12
12:         $Re \leftarrow$ BUILDFETCHREQUEST($Ex_{addr}$)       ▷ Fetch remote page : 7
13:         SENDREQUEST($Re$)                                              ▷ 13
14:         $Re \leftarrow$ RECEIVECONFIRMATION                                 ▷ 14
15:         if $Re_{result} = InvalidPage$ then                            ▷ 15
16:             DOPANIC                                                    ▷ 11
17:         RECEIVEPAGE($Ex_{addr}$)                                       ▷ 16
18:     UPDATEMEMORYSYSTEM($Ex_{addr}$)                                    ▷ 17
```

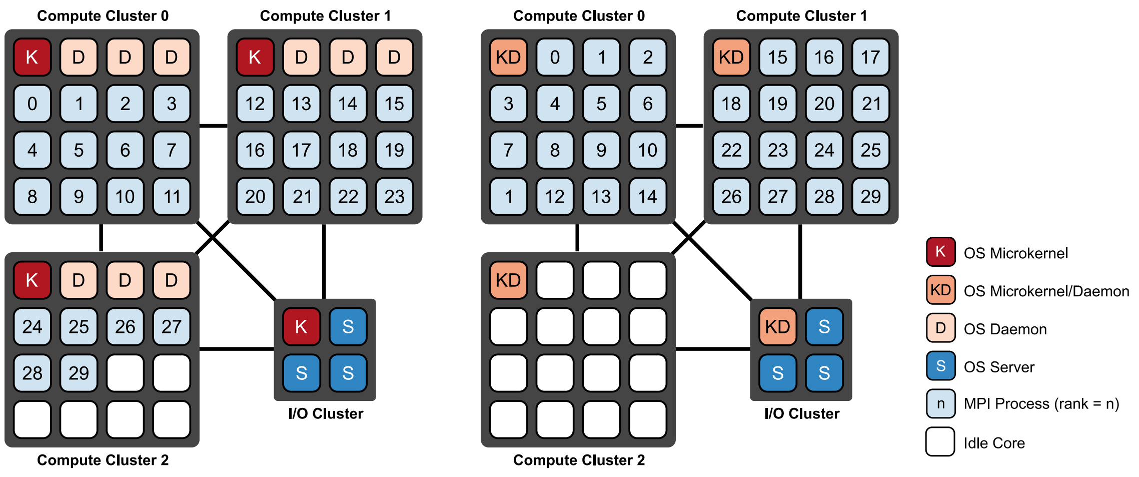
Source: Developed by the Author.

(lines 12 - 17). If the address is invalid, the daemon interrupts the application execution (line 10 and line 16). On the flow conclusion, the user handler is released (line 18.

Figure 9 illustrates the graph representation of RMem. Due to its complexity, this flow requires a considerable number of tasks (23 tasks in total). To deal with the invalid address detection and the execution interruption, we defined a task that is only scheduled when a request fails (Task (11)). All arcs in flush and fetch operations (dashed boxes) can propagate errors in addition to the regular paths. When the error propagation reaches Task (11), the *Dispatcher* schedules Task (11) due to a catch condition.

Figure 10 – Overview of the MPI process management in LWMPI (original vs. tasks).

(a) Original LWMPI. (b) LWMPI with tasks.



Source: Developed by the Author.

4.2.4 Impacts on Nanvix User Libraries

Although the engine primarily brings advantages to the design of OS-level functionalities, it has a positive impact on the support of user-level libraries. In general, the main benefits for user-level software are the following: (i) more cores available to the user application, improving user density and parallelism; (ii) lower competition and interference over user resources; and (iii) more memory available, since OS services implemented with tasks consume less memory. Some examples of user-level libraries in Nanvix are SysV, POSIX, and LWMPI.

In this dissertation, we focus on LWMPI (ULLER et al., 2021), which is an open-source MPI library featured by Nanvix that transparently profits from the advantages of the task-based environment. LWMPI follows the MPI specification v3.1 and implements a subset of the MPI functionalities. LWMPI relies on two important OS functionalities that were redesigned to work with tasks: NameServer and IKC. The former is used in LWMPI to individually address all MPI processes whereas the latter is used to implement the MPI communication protocols and to carry out fine- and coarse-grained communications.

Figure 10 presents a graphical representation of a scenario with 30 MPI processes running on a subset of Compute Clusters of the Kalray MPPA-256. Thanks to the microkernel and multikernel redesigns, it is possible to run MPI processes on all cores of the Compute Cluster except the master core. Therefore, the task-based version of the LWMPI delivers increased processing power within the cluster and better explores shared-memory communications.

5 EVALUATION METHODOLOGY

To carry out a comprehensive assessment of the proposed engine, we designed a set of benchmarks to assess the impacts on the memory, execution time, and energy consumption on a baremetal lightweight manycore. The following questions guided our evaluation methodology:

(Q1) *How much memory is preserved by our solution?*
(Q2) *What is the cost of dealing with tasks instead of threads?*
(Q3) *How much interference (overhead) does our solution introduce?*
(Q4) *What is the performance achieved by our solution when subjected to MPI applications?*

In this chapter, we first give a brief description of the lightweight manycore employed in all experiments. Then, we discuss the experimental design that endeavors to answer the aforementioned research questions.

5.1 LIGHTWEIGHT MANYCORE PROCESSOR

Nanvix supports a distinct variety of lightweight manycore architectures with limited on-chip memory. In this dissertation, we chose the Kalray MPPA-256, a silicon lightweight manycore that features all peculiarities discussed in Section 2.1.

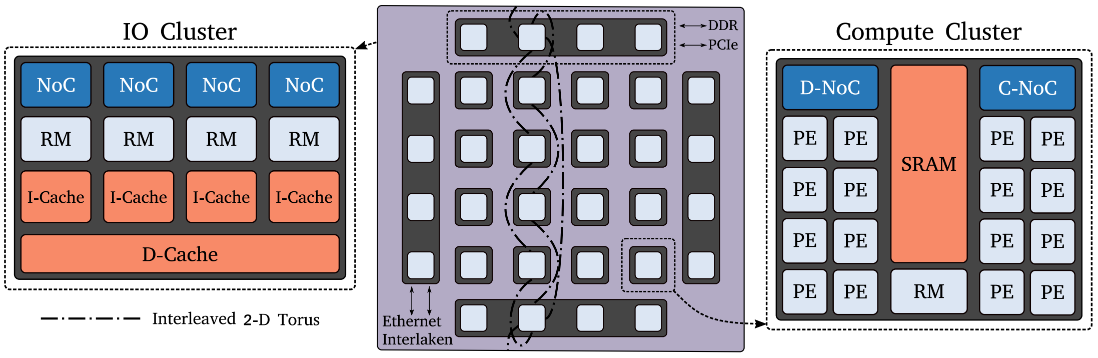
Figure 11 presents an architectural overview of the Kalray MPPA-256 processor. Kalray MPPA-256 integrates 288 general-purpose cores, grouped into 16 Compute Clusters and 4 I/O Clusters. Each Compute Cluster features 16 Processing Elements (PEs), 1 Resource Manager (RM), 2 MB of local SRAM, two NoC interfaces, a software-managed TLB, and does not have hardware support for cache coherence. Each I/O Cluster features 4 RMs, 4 MB of SRAM, 8 NoC interfaces, and hardware support for cache coherence. Two I/O Clusters have access to 4 GB of DRAM, and the other two have access to an Ethernet interface.

Two distinct 2-D Torus NoCs handle inter-cluster communication: a Control NoC (C-NoC) allows the exchanging of small control messages and a Data NoC (D-NoC) supports transfers of arbitrary amounts of data.

5.2 EXPERIMENTAL DESIGN

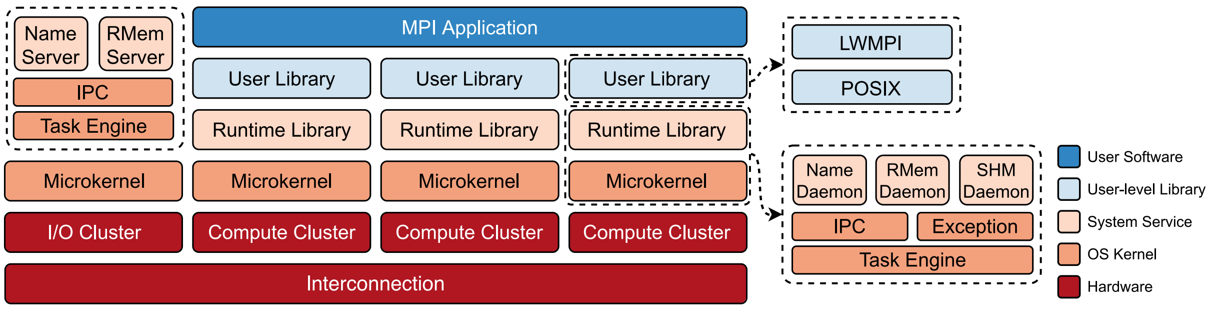
The experiments seek to evaluate the performance of the proposed engine and its impacts on MPI applications. We considered both OS- (kernel and runtime libraries) and user-level (user libraries and applications) software, thus allowing a detailed analysis of the whole system. Figure 12 presents an overview of the software stack of Nanvix.

Figure 11 – Architectural overview of the Kalray MPPA-256 processor.



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

Figure 12 – Nanvix software stack.



Source: Adapted from Uller et al. (2021).

We gathered time measurements using hardware performance counters to enable monitoring with minimum interference. We collected energy measurements from sensors available on the Kalray MPPA-256 board, which provide the power consumption of the entire processor (including the on-chip memories and NoC). All measures started after the boot period of Nanvix.

We carried out 10 runs of each MPI application and 30 runs of each synthetic and OS service benchmark in order to guarantee statistical relevant results. All results represent average values and are based on a confidence interval threshold of 95% (significance of 5%). All experiments and the software stack are publicly available.[1]

5.2.1 Nanvix Variants

We considered 3 variants of Nanvix in the experiments:

Baseline (`Baseline`). This is the standard version of Nanvix without any changes or additional features introduced in this dissertation.

Partial task (`Partial`). This version replaces the original Inter-Kernel Communication module (IKC) with a task-based one. This means that all communications are carried out with tasks.

[1] Available at: https://github.com/nanvix/benchmarks, commits c57d45b and dd02d38.

Table 4 – Parameters used by the experiments.

Type	Name	Parameters
Theoretical	OS footprint	2 MB SRAM
Synthetic	Thread vs Task	N flows, $N \in [1, 21]$
	Kernel Latencies	N tasks vs M threads, $N \in [0, 1], M \in [1, 15]$
	Core Usage	One heartbeat protocol execution by one process
OS Services	Pginval	N invalidations, $N = 100$
	Pgfetch	N transfers, $N = 100$
User Library	FN	Numbers $\in [1000001, 1000001 + N)$, $N = 1536 * nclusters$
	GF	N images of 256×256 pixels, 7×7 mask, $N = 1200 * nclusters$
	KM	N 2D points, 128 centroids, $N = 13440 * nclusters$

Source: Developed by the Author.

Full task (`Full`). This version has full support of tasks throughout the system, including communications, OS daemons and OS modules.

5.2.2 Benchmarks and Applications

In all experiments, we focused on comparing the results obtained with our task-based implementations against Nanvix's native thread-based solutions. We classified the experiments in 4 classes, which are presented next: (i) **theoretical**: evaluates the memory consumption in different layers of the OS; (ii) **synthetic**: employs synthetic applications to measure the OS response time; (iii) **OS services**: evaluates the performance of OS protocols; and (iv) **user library**: evaluates the performance of user applications.

A brief description of each experiment as well as its correlation with the research questions are given below. Table 4 summarizes the configuration of each experiment.

OS Footprint (Q1). This theoretical analysis compares the maximum number of concurrent execution flows and the memory footprint of the thread- and task-based versions of Nanvix.

Thread vs. Task (Q2). It compares the execution time needed to create/destroy threads vs. dispatch/wait for tasks. Threads/tasks execute the same dummy function to measure the basic overheads. This experiment is limited to 21 flows because it is the maximum number of user threads that Nanvix currently supports in a Compute Cluster.

Kernel Latencies (Q2, Q3). It measures the response time of remote system calls requested by traditional threads vs. by the *Dispatcher*. This experiment observes the interference introduced by our engine in kernel response and how it affects kernel scalability.

Core Usage (Q2, Q3). It details the execution time spent by all entities involved in the *heartbeat* protocol of the NameServer service. This protocol periodically informs the process manager that a process is alive.

Pginval (Q3). It measures the latency for invalidating remote page cache entries. It launches multiple processes that share and iteratively request the invalidation of a memory region. This program relies on N processes requesting $1 : N-1$ broadcast communication protocol using the SHMem service.

Pgfetch (Q3). It assesses the time for transferring a page from the memory server to a process. It launches multiple processes that fetch several pages from the remote server by iteratively allocating some memory, reading from it, and releasing it. This program relies on a $1 : 1$ ping-pong communication protocol using the RMem service.

User Library (Q4). It evaluates the performance of user applications when running on the original and task-based versions of Nanvix. To evaluate MPI applications, we chose three applications extracted from *CAP Bench* (SOUZA et al., 2017), a benchmark suite designed to evaluate the performance of lightweight manycores. These applications run on top of LWMPI (ULLER et al., 2021). Although the proposed engine allows assigning more MPI processes per Compute Cluster, we ran all CAP Bench applications with 12 MPI processes so as to make a fair comparison between the original and task-based versions of Nanvix. A brief description of each application is given below:

- **Friendly Numbers (FN)** is an application that 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 with regular loads. The problem is predominantly CPU-bound.
- **Gaussian Filter (GF)** is a filter that reduces the noise of an image by applying a matrix convolution operation with a special two-dimensional Gaussian mask to the image pixels. GF follows the *stencil* parallel pattern and features a CPU-intensive workload with medium communication intensity.
- **K-Means (KM)** is a clustering technique employed in data analysis. Given a set of n points in real d-dimensional space, KM randomly split them into k partitions. Then, it applies the *Map* parallel pattern to distribute points and replicate data centroids between the Compute Clusters. This application has the highest communication intensity, since data centroids are updated at every iteration.

6 EXPERIMENTAL RESULTS

In this chapter, we present and discuss our experimental results. First, we show a simple qualitative analysis of the memory consumption of our solution. Then, we examine its raw performance and scalability with synthetic benchmarks. Finally, we evaluate its performance when running OS services and MPI applications.

6.1 MEMORY CONSUMPTION

Table 5 presents an extrapolation of the number of concurrent execution flows that can coexist in different reserved memory sizes. For simplicity, the assessment does not consider the memory required for the OS and user data. Furthermore, we assume that the entire memory of a Compute Cluster (2 MB in Kalray MPPA-256) is available to keep task data and stacks. We consider 128 bytes and 4 kB for the task structure and memory page, respectively. An execution stack is made up of two memory pages (one for user mode and the other for kernel mode).

Overall, we observed a linear increase in the maximum number of execution flows as the memory reserved is increased. However, the maximum number of threads increases with a factor of 0.125× whereas this factor is about 7.9× for the task approach. This extrapolation unveils that our solution can significantly reduce the OS memory footprint. The reuse of the execution stack and the small control structures of tasks provide 63.2× more execution flows per MB of memory than traditional threads.

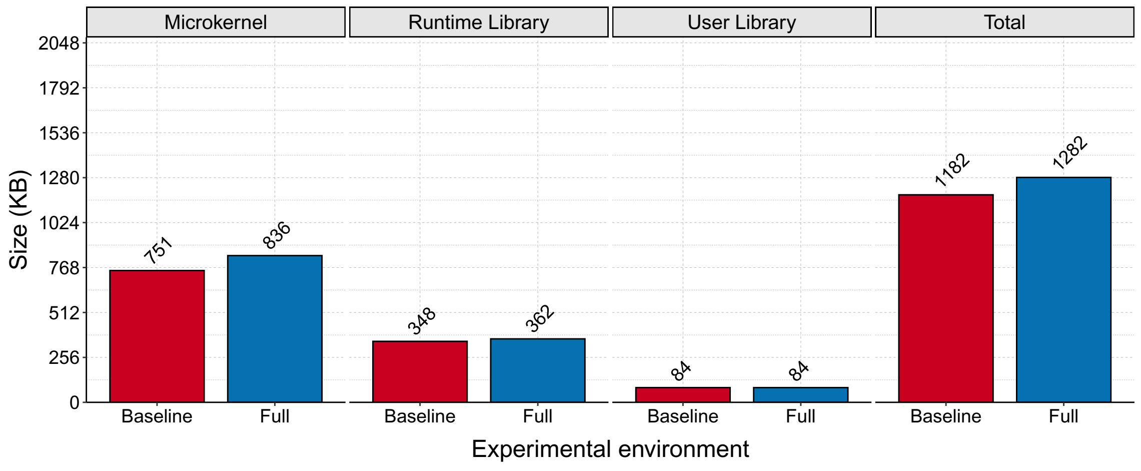
Figure 13 shows an approximation of the memory consumption of `Baseline` and `Full` variants. Results show that the memory footprint of the task-based microkernel (`Full`) is about 12% bigger than its original version (`Baseline`). This difference represents the entire implementation of the task engine as well as kernel components. However, `Full` allows the inclusion of new OS daemons without any significant impact on the memory consumption, since it replaces OS daemons (threads) by a single *Dispatcher* thread that executes lightweight tasks.

Table 5 – Extrapolation of the maximum number of flows per Compute Cluster.

Execution Flow	**Memory Required**	**No. of Flows per Reserved Memory (kB)**					
		≈8	**16**	**64**	**256**	**1024**	**2048**
Thread	8 kB per thread	1	2	8	32	128	256
Task	128 B per task + 8 kB	1	64	448	1984	8128	16320

Source: Developed by the Author.

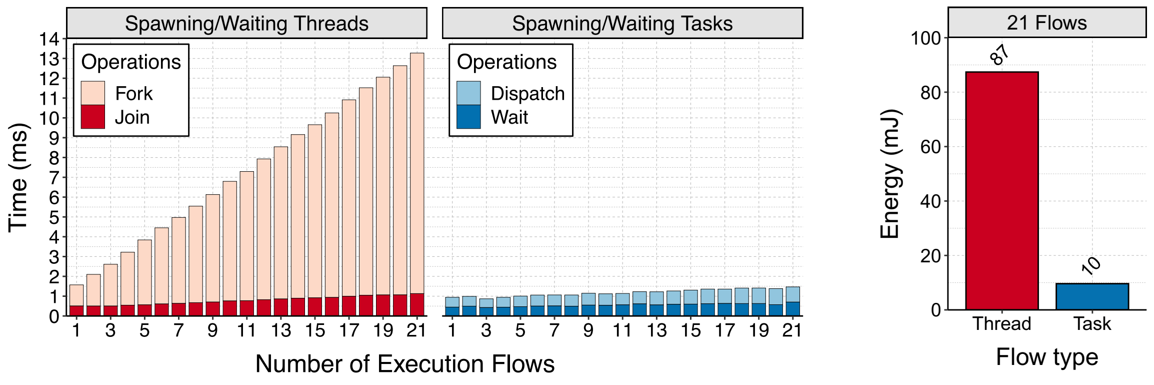
Figure 13 – Approximation of the memory footprint of the `Baseline` and `Full` Nanvix variants.



Source: Developed by the Author.

Figure 14 – Latency and energy consumption involved in spawning/waiting threads/tasks.

(a) Latency. (b) Energy with 21 flows.



Source: Developed by the Author.

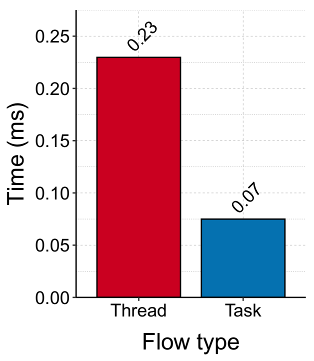
6.2 PERFORMANCE, SCALABILITY AND OVERHEAD

Figure 14 shows the latency and energy consumption of spawning (fork/dispatch) and waiting (join/wait) execution flows (threads vs. tasks). Overall, we observed that the latency of the join is slightly higher than the wait operation, being the former up to 1.87× higher than the latter (Figure 14(a)). However, we noticed that the overhead involved in forking a new thread increases significantly compared with creating tasks. This overhead is at least 2× and 15.3× higher with 1 and 21 threads, respectively. These results unveil that our task-based solution presents better scalability and lower costs to spawn OS functionalities. Figure 14(b) shows the energy consumption of the experiment with 21 threads/tasks. The task-based solution consumes 6.6× less energy than traditional threads, reflecting the lower latencies of the dispatching/waiting of tasks.

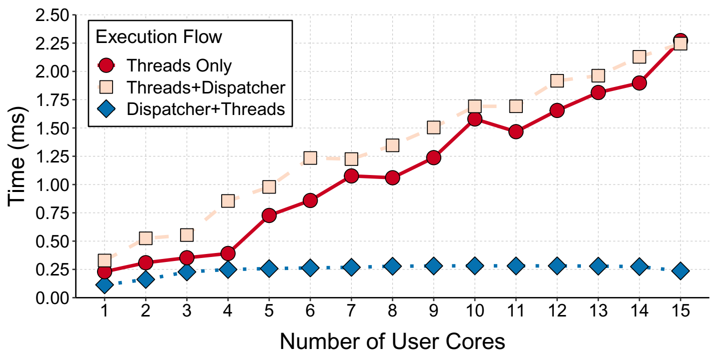
Figure 15 presents the response time of remote system call requests. In Figure 15(a), we analyze the base response time when a single thread or a single task requests

Figure 15 – Response times of remote system calls.

(a) Single thread/task.



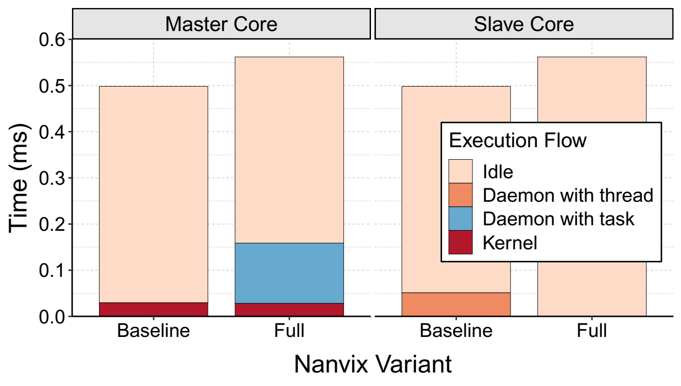
(b) Single task, multiple threads.



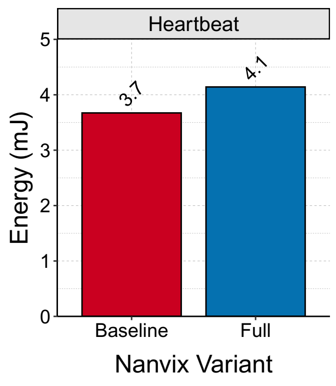
Source: Developed by the Author.

Figure 16 – Detailed execution of the *heartbeat* protocol of the NameServer Service.

(a) Core Usage.



(b) Energy.



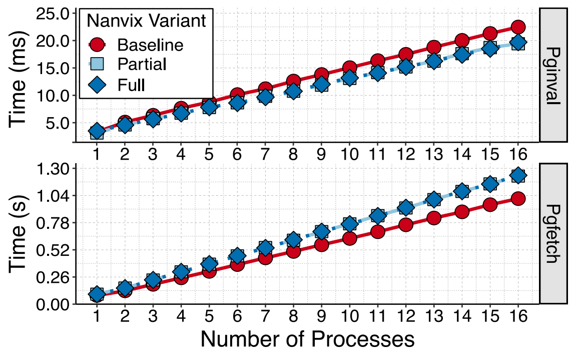
Source: Developed by the Author.

a remote system call. As it can be noticed, the response time is 3.1× shorter for the task. The reason behind this result is that the *Dispatcher* shares the same core of the master thread. Figure 15(b) shows the response times when system calls are requested by 1 task along with a variable number of threads. In this plot, there are three different curves depicted, each of which featuring a distinct behavior: (i) the response time where only threads make system calls (*Threads-only*); (ii) the response time of the *Dispatcher* when competing with user threads (*Dispatcher+Threads*); and (iii) the average response time of a thread when competing with other threads and the *Dispatcher* (*Threads+Dispatcher*). In general, the *Dispatcher+Threads* presented good scalability with an overhead smaller than the *Threads-only*. Conversely, the *Threads+Dispatcher* was 1.22× slower compared to the *Threads-only* scenario. Overall, these results unveil that the *Dispatcher* did not add significant overhead to the response time of the kernel.

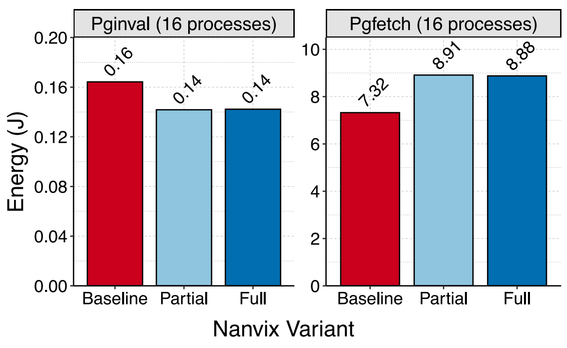
Figure 16 shows the breakdown of execution times of the *heartbeat* protocol of the NameServer service on `Baseline` and `Full` variants. The results exhibit three relevant findings: (i) the *Dispatcher* takes advantage of the idle time of the master core (the

Figure 17 – Overall response times when running the Pginval protocol.

(a) Latency.



(b) Energy.



Source: Developed by the Author.

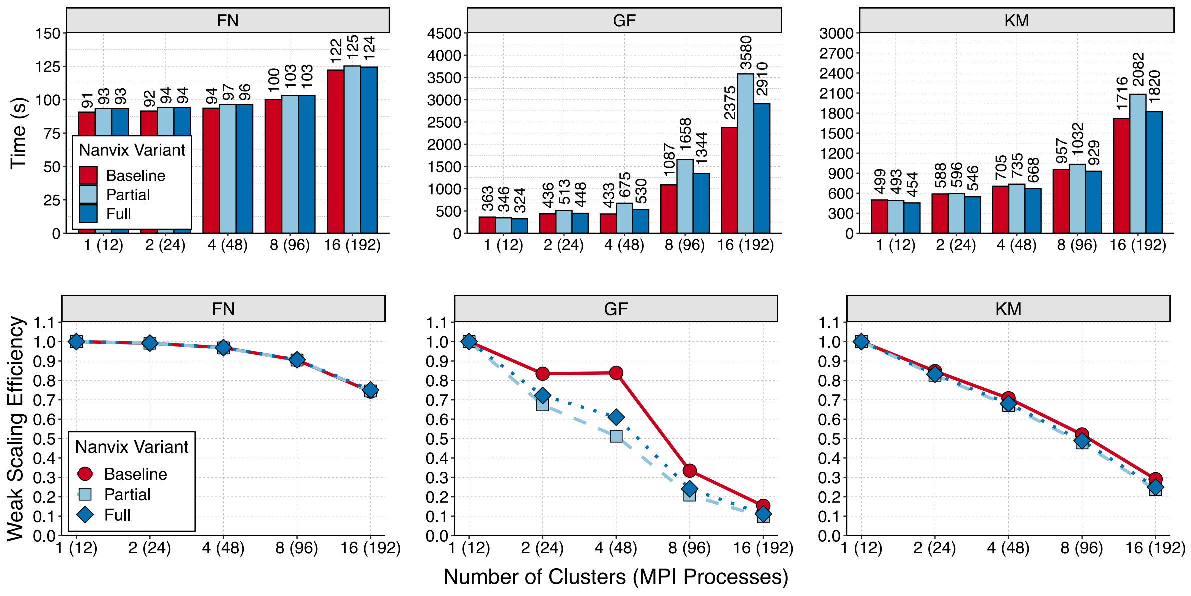
one that executes the master thread) to execute tasks; (ii) the *busy-wait* aspect of the communication protocol increases the core utilization of *Dispatcher*, although it does not significantly interfere with the execution of the kernel (master thread); and (iii) the *extra work* and the *isolation* of the communication on the master core were the foremost factors for increasing the latency and energy consumed by the service (e.g., requesting more context switching between the *Dispatcher* and the master thread).

6.3 PERFORMANCE ANALYSIS OF OS SERVICES

Figure 17 pictures the latency and energy consumption for invalidating page cache entries using the *SHMem service* (Pginval benchmark). In this benchmark, N processes compete to request page invalidations to the SysV server that broadcasts small and fixed-size messages to $N-1$ SHMem daemons. We evaluated the `Baseline`, `Partial`, and `Full` variants with different numbers of processes (from 1 to 16 processes). Results obtained with Pginval show that `Partial` and `Full` variants achieved 13.5% and 15% better performance and energy efficiency than `Baseline`, respectively. Results with tasks were faster because: (i) the *Dispatcher* better manages fine-grained communications, exploring the lower syscall latencies during communications on the master core; (ii) the invalidation protocol relies mainly on the SysV server and the SHMem daemon does not affect the requester side. Although `Partial` and `Full` had similar performance, the `Full` variant has a smaller memory footprint.

Figure 17 also shows the latency and energy consumption for fetching remote pages using the RMem service (Pgfetch benchmark). In this benchmark, each process manipulates unrelated 32 memory pages (4 kB each). To fetch a remote page from the RMem server, the protocol employs both small and large messages, which correspond to control data and page data transfers, respectively. Our results unveil a different behavior in Pgfetch, where `Baseline` approximately achieved 22% and 21% better performance and energy efficiency than the task-based variants, respectively. In the original version of

Figure 18 – Weak scaling: execution times (top) and efficiencies (bottom) for FN, GF and KM.



Source: Developed by the Author.

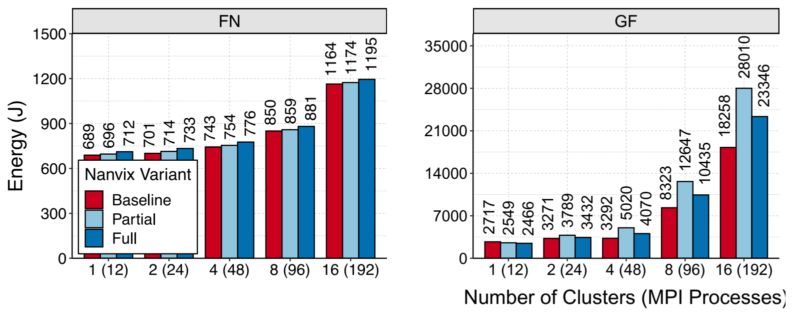
Nanvix (`Baseline`), data copy and deallocation of communication buffers are delegated to user threads. This solution improves the system performance, since user threads run in parallel with the kernel, but it breaks the premise of *kernel isolation* of the asymmetric microkernel model. Since we isolate communications on the master core in `Partial` and `Full` variants, all these operations are executed by the *Dispatcher*, resulting in such overhead. Lastly, OS daemons with tasks did not present any interference on the executions comparing the `Partial` and `Full` results.

6.4 PERFORMANCE ANALYSIS OF MPI APPLICATIONS

We now evaluate the performance of user-level parallel applications implemented with MPI. As mentioned before, these applications use the LWMPI library, which in turn, leverages Nanvix IPC to carry out communications between MPI processes. Since Nanvix IPC is implemented with tasks in `Partial` and `Full` variants, we expect performance impacts on communication-bound applications. Figure 18 pictures the weak scaling results for FN, GF, and KM applications based on two metrics: execution time and weak scaling efficiency.

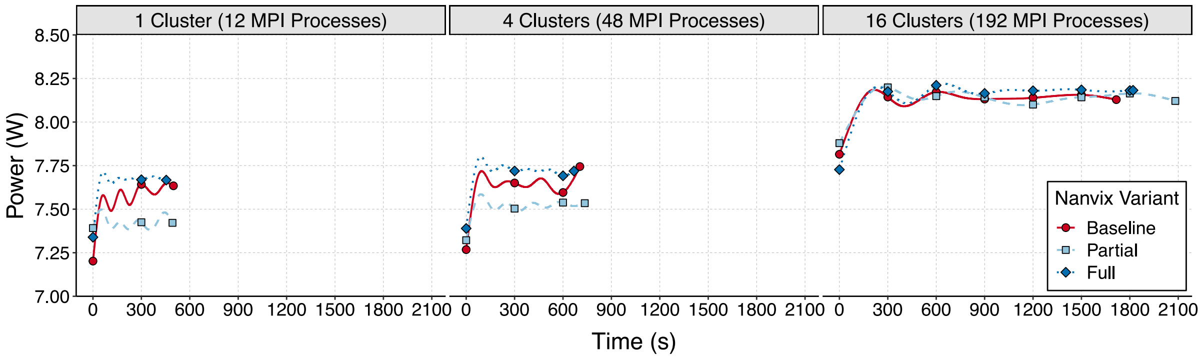
As expected, all Nanvix variants showed similar performance results for FN, since this application is CPU-bound and has low communication demand. This result also indicates that running the *Dispatcher* along with the master thread on the master core does not affect significantly the overall performance of the application. On average, the task-based execution engine introduced 2% of overhead on the FN execution time compared with the `Baseline`. The weak scaling efficiency of the FN did not present any

Figure 19 – Weak scaling: energy consumption of FN, GF, and KM.



Source: Developed by the Author.

Figure 20 – Weak scaling: power consumption of KM with 12, 48, and 192 MPI processes.



Source: Developed by the Author.

significant difference between the Nanvix variants.

The GF application exhibited significant differences between the Nanvix variants. This behavior comes from the time spent in coarse-grained inter-cluster communications. On average, the performance of GF with `Partial` and `Full` was 34% and 12% worse than the one achieved with `Baseline`, respectively. This performance degradation is due to the same reason explained in Section 6.3: the *Dispatcher* spends more time executing data copies and deallocations, preventing the application to continue its execution until these operations are completed. We noticed, however, that `Full` executed 19% faster than the `Partial`, on average. After a careful analysis of this experiment, we concluded that this is due to the better name resolution management on `Full`. The NameServer daemon with tasks removes the intermediary kernel thread and combines small communications with the daemon logic.

Finally, we observed a different behavior with KM. In this case, `Full` presented the best results of all Nanvix variants, except for the experiment with 192 MPI processes. Despite the high communication intensity aspect, KM relies on less coarse-grained communications than the GF. When executed on `Full`, KM was 1.04× and 1.10× faster than on `Baseline` and `Partial`, respectively. All Nanvix variants showed similar weak scaling efficiency with a slightly better efficiency achieved by `Baseline` when compared to the other task-based variants.

We now focus on the power and energy consumption of the applications when running with `Baseline`, `Partial` and `Full` variants. Figure 19 pictures the overall energy consumption of FN, GF, and KM. As it can be noticed, the energy consumption of each scenario follows the same trends observed in the execution times, meaning that the execution time is the most important factor for the energy consumption. Figure 20 exemplifies this observation, which shows the power consumption during the execution of KM. We noticed a slightly higher power consumption with `Full` due to a better exploitation of the master core (its idle time is used to carry out communications and daemons' logics). However, the increase in power consumption is not enough to have an important impact on the energy consumption. We observed the same behavior for the other applications.

7 CONCLUSION

Lightweight manycore processors achieve both high performance and energy efficiency thanks to a set of architectural features such as extreme parallelism with a distributed and restrictive memory architecture. OSes for lightweight manycores embrace a distributed structure to achieve scalability while exposing richer abstractions to user-level software. The asymmetric microkernel design is commonly adopted to deal with peculiarities of lightweight manycores due to its reduced memory footprint and high scalability. However, this design does not address all restrictions arrived from the limited on-chip memory.

To alleviate the memory constraints of lightweight manycores, we proposed a task-based execution engine as an alternative to traditional execution flow abstractions (e.g., processes and threads). The engine supports cooperative lightweight tasks that share a unique execution stack and features task synchronization via control flow and dependency graphs, eliminating the need of dedicated processes/threads to implement OS-level functionalities. It allows the execution of numerous OS-level execution flows with reduced memory consumption and is orthogonal to the underlying OS execution support.

We implemented our engine in a distributed OS and showed how important modules of the OS (microkernel and multikernel) could be redesigned to work with tasks. We carried out several experiments on Kalray MPPA-256, a lightweight manycore processor that features 288 cores in a single chip. Our results showed that the proposed solution has the following advantages when compared to the use of threads to implement OS-level functionalities: (i) it provides 63.2× more execution flows per MB of memory; (ii) it features less overhead to manage execution flows and system calls; (iii) it improves the master core utilization; and (iv) it exhibits competitive results with real-world applications.

7.1 FUTURE WORK

Due to limited time, we restricted the scope of the dissertation to consolidate the fundamentals of the proposed execution engine and its implementation in a real-world context. This restriction led us not to investigate some research aspects. In this context, future work can take into account the following facts:

Remove busy-wait from communication The communication module can be changed to avoid the busy-wait approach in the communication channel reservation using a flow chaining strategy, increasing engine responsiveness. This strategy could be implemented through the already supported on-demand dependency functionality.

Scheduling policies This dissertation limited the scheduling mechanism to a naive approach. Scheduling policies that consider control flow and dependency graph properties could identify critical flows to provide runtime optimizations.

Replacement of server threads We focused to study Compute Clusters in this dissertation because they are more restrictive and impose severe constraints on the number of system threads concurrently running. However, I/O Clusters present the same restrictions where the number of servers is limited to the number of threads available. Thus, it would be possible to replace server threads by a set of Dispatchers and this solution could deliver benefits in memory consumption, increasing the maximum number of services available in the system.

New services Currently, Nanvix requires only three OS daemons. As the system evolves, the benefits of the proposed engine can be studied in other OS service contexts.

7.2 PUBLICATIONS

The work presented in this dissertation was partially reported in the proceedings of the IEEE International Symposium on Computer Architecture and High Performance Computing (SOUTO; CASTRO; PENNA, 2021). The author also contributed to the publication of an article in the International Journal of Parallel and Distributed Computing (PENNA et al., 2021), an article to the Concurrency and Computation: Practice and Experience (ULLER et al., 2021) and of a paper in the proceedings of the Simpósio em Sistemas Computacionais de Alto Desempenho (ULLER et al., 2020). More information about these papers can be found below:

- SOUTO, J.; CASTRO, M.; PENNA, P. A task-based execution engine for distributed operating systems tailored to lightweight manycores with limited on-chip memory. In: **2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. Los Alamitos, CA, USA: IEEE Computer Society, 2021. p. 74–83. Disponível em: https://doi.ieeecomputersociety.org/10.1109/SBAC-PAD53543.2021.00019.
- 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 (JPDC)**, 2021. Disponível em: https://doi.org/10.1016/j.jpdc.2021.04.002.
- ULLER, J. F. et al. LWMPI: An MPI Library for NoC-Based Lightweight Manycore Processors with On-Chip Memory Constraints. **Concurrency and Computation: Practice and Experience**, 2021. Disponível em: https://doi.org/10.1002/cpe.6693.
- 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.

BIBLIOGRAPHY

ANDERSON, T. E. et al. Scheduler activations: Effective kernel support for the user-level management of parallelism. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 10, n. 1, p. 53–79, fev. 1992. ISSN 0734-2071.

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.

Ax, J. et al. Coreva-mpsoc: A many-core architecture with tightly coupled shared and local data memories. **IEEE Transactions on Parallel and Distributed Systems**, v. 29, n. 5, p. 1030–1043, 2018.

BATENI, S. et al. Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform. In: **2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)**. [S.l.: s.n.], 2020. p. 310–323.

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.

BIRRELL, A. D. **An introduction to programming with threads**. [S.l.], 1989.

BOHNENSTIEHL, B. et al. Kilocore: A 32-nm 1000-processor computational array. **IEEE Journal of Solid-State Circuits (JSSC)**, v. 52, n. 4, p. 891–902, 2017. ISSN 00189200.

BOYD-WICKIZER, S. et al. Corey: An operating system for many cores. In: **OSDI ’08**. USENIX, 2008. The 8th USENIX Symposium on Operating Systems Design and Implementation. Disponível em: https://www.microsoft.com/en-us/research/publication/corey-an-operating-system-for-many-cores/.

CAI, M.; ZHANG, D.; HUANG, H. A scalable virtual memory system based on decentralization for many-cores. **Journal of Systems Architecture**, v. 107, p. 101803, 2020. ISSN 1383-7621.

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.

DAVIS, R.; MERRIAM, N.; TRACEY, N. How embedded applications using an rtos can stay within on-chip memory limits. In: **In Proceedings of the Work in Progress and Industrial Experience Session, Euromicro Conference on RealTime Systems**. [S.l.: s.n.], 2000. p. 43–50.

DIETRICH, C.; LOHMANN, D. Semi-extended tasks: Efficient stack sharing among blocking threads. In: **2018 IEEE Real-Time Systems Symposium (RTSS)**. [S.l.: s.n.], 2018. p. 338–349.

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.

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.

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.

GAI, P.; LIPARI, G.; NATALE, M. D. Stack size minimization for embedded real-time systems-on-a-chip. **Des. Autom. Embedded Syst.**, Kluwer Academic Publishers, USA, v. 7, n. 1–2, p. 53–87, set. 2002. ISSN 0929-5585.

KATSARAGAKIS, M. et al. Memory footprint optimization techniques for machine learning applications in embedded systems. In: **2020 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.: s.n.], 2020. p. 1–4.

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.

KUSAKABE, S. et al. Os mechanism for continuation-based fine-grained threads on dedicated and commodity processors. In: **2007 IEEE International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2007. p. 1–10.

LIU, X. et al. Memory optimization techniques for multithreaded operating system on wireless sensor nodes. In: **2014 IEEE International Conference on Progress in Informatics and Computing**. [S.l.: s.n.], 2014. p. 503–508.

LOMET, D. B. Process structuring, synchronization, and recovery using atomic actions. **Operating Systems Review**, v. 12, p. 128–137, 1977.

MARCHAL, L. et al. Parallel scheduling of dags under memory constraints. In: **2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.: s.n.], 2018. p. 204–213.

MAURONER, F.; BAUNACH, M. Stackmmu: Dynamic stack sharing for embedded systems. In: **2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)**. [S.l.: s.n.], 2017. p. 1–9.

MIDDHA, B.; SIMPSON, M.; BARUA, R. Mtss: Multitask stack sharing for embedded systems. **ACM Trans. Embed. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 7, n. 4, aug 2008. ISSN 1539-9087. Disponível em: https://doi.org/10.1145/1376804.1376814.

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.

PARK, D. et al. Reducing the memory footprint of osek-based systems via stack sharing and light-weight ready queues. **International Journal of Automotive Technology**, v. 12, p. 451–460, 06 2011.

PAULI, W.; SOFFA, M. L. Coroutine behaviour and implementation. **Software: Practice and Experience**, Wiley, v. 10, n. 3, p. 189–204, mar 1980. Disponível em: https://doi.org/10.1002%2Fspe.4380100305.

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**, p. 1–5, 2020. ISSN 1943-0671.

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.

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 (JPDC)**, 2021. Disponível em: https://doi.org/10.1016/j.jpdc.2021.04.002.

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.

PIZKA, M. Thread segment stacks. In: **PDPTA**. [S.l.: s.n.], 1999.

RHODEN, B. et al. Improving per-node efficiency in the datacenter with new os abstractions. In: **Proceedings of the 2nd ACM Symposium on Cloud Computing**. New York, NY, USA: Association for Computing Machinery, 2011. (SOCC '11). ISBN 9781450309769. Disponível em: https://doi.org/10.1145/2038916.2038941.

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.

SARDESAI, S.; MCLAUGHLIN, D.; DASGUPTA, P. Distributed cactus stacks: Runtime stack-sharing support for distributed parallel programs. In: CITESEER. **Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications**. [S.l.], 1998. p. 57–65.

SOUTO, J.; CASTRO, M.; PENNA, P. A task-based execution engine for distributed operating systems tailored to lightweight manycores with limited on-chip memory. In: **2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. Los Alamitos, CA, USA: IEEE Computer Society, 2021. p. 74–83. Disponível em: https://doi.ieeecomputersociety.org/10.1109/SBAC-PAD53543.2021.00019.

SOUZA, M. A. et al. CAP Bench: A Benchmark Suite for Performance and Energy Evaluation of Low-power Many-core Processors. **Concurrency and Computation: Practice and Experience**, v. 29, n. 4, p. e3892, 2017. ISSN 1532-0634.

STERLING, T. L. Clusters. In: _____. **Encyclopedia of Parallel Computing**. Boston, MA: Springer US, 2011. p. 289–297. ISBN 978-0-387-09766-4. Disponível em: https://doi.org/10.1007/978-0-387-09766-4_18.

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. LWMPI: An MPI Library for NoC-Based Lightweight Manycore Processors with On-Chip Memory Constraints. **Concurrency and Computation: Practice and Experience**, 2021. Disponível em: https://doi.org/10.1002/cpe.6693.

WENTZLAFF, D.; AGARWAL, A. Factored operating systems (fos): The case for a scalable operating system for multicores. **SIGOPS Oper. Syst. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 43, n. 2, p. 76–85, apr 2009. ISSN 0163-5980. Disponível em: https://doi.org/10.1145/1531793.1531805.

WISNIEWSKI, R. W. et al. mos: an architecture for extreme-scale operating systems. In: **International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)**. Munich, Germany: ACM Press, 2014. p. 1–8. ISBN 9781450329507.

WULF, W. A.; MCKEE, S. A. Hitting the memory wall: Implications of the obvious. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 23, n. 1, p. 20–24, mar 1995. ISSN 0163-5964. Disponível em: https://doi.org/10.1145/216585.216588.

Zhang, R. et al. Loffs: A low-overhead file system for large flash memory on embedded devices. In: **2020 57th ACM/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2020. p. 1–6.

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.