On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores

Pedro Henrique Penna, João Souto, Davidson Lima, Márcio Castro, François Broquedis, Henrique Freitas, Jean-François Mehaut

To cite this version:

Pedro Henrique Penna, João Souto, Davidson Lima, Márcio Castro, François Broquedis, et al.. On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores. SBESC 2019 - IX Brazilian Symposium on Computing Systems Engineering, Nov 2019, Natal, Brazil. pp.1-31. hal-02297637v3

HAL Id: hal-02297637

https://hal.archives-ouvertes.fr/hal-02297637v3

Submitted on 30 Mar 2020

On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores

Pedro Henrique Penna, João Souto, Davidson Lima, Márcio Castro, François Broquedis, Henrique Freitas, Jean-Francois Mehaut

On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores

Pedro Henrique Penna
UGA, PUC Minas
Grenbole, France
pedro.penna@sga.pucminas.br

João Vicente Souto
UFSC
Florianópolis, Brazil
joao.vicente.souto@grad.ufsc.br

Davidson Francis Lima
PUC Minas
Belo Horizonte, Brazil
davidson.francis@sga.pucminas.br

Márcio Castro
UFSC
Florianópolis, Brazil
marcio.castro@ufsc.br

François Broquedis
Grenoble INP, INRIA
Grenbole, France
francois.broquedis@univ-grenoble-alpes.fr

Henrique Freitas
PUC Minas
Belo Horizonte, Brazil
cota@pucminas.br

Jean-François Méhaut
UGA, LIG, CNRS
Grenbole, France
jean-francois.mehaut@univ-grenoble-alpes

***Abstract*—Multikernel operating systems (OSs) were introduced to match the architectural characteristics of lightweight manycores. While several multikernel OS designs are possible, in this work we argue on one that is structured in asymmetric microkernel instances. We deliver an open-source implementation of an OS kernel with these characteristics, and we provide a comprehensive assessment using a representative benchmark suite. Our results show that an asymmetric microkernel design is scalable and introduces at most 0.9% of performance interference in an application execution. Also, our results unveil co-design aspects between an OS kernel and the architecture of lightweight manycore, concerning the memory system and core grouping.**

***Index Terms*—manycore, operating system, kernel, MPPA-256**

I. Introduction

Lightweight manycores were introduced to cope with the ever-increasing high performance and low-power consumption demands of applications [1]. To meet these requirements, this emerging class of processors relies on a selected set of architectural characteristics. On one hand, to enable scalability, lightweight manycores feature a distributed memory architecture and a rich on-chip interconnect [2]. On the other hand, to achieve energy efficiency, they are built with simple and low-power cores [3]; have a memory system based on Scratchpad Memories (SPMs) with no hardware cache coherency support [4]; and exploit heterogeneity by bundling cores with different capabilities [5]. Some examples of lightweight manycores are the Kalray MPPA-256 [6]; the Adapteva Epiphany [7]; and the Sunway SW26010 [8].

Due to an outstanding performance scalability and energy efficiency, lightweight manycores are currently employed in areas that have demand for such requirements, like in Embedded Computing and High-Performance Computing (HPC) [9]. Nonetheless, when comes to the applicability of these processors in other use-case scenarios that have additional needs, such as multi-application/user support, lightweight manycores are not yet that diffused. The rationale for this observation emerges from the system-level support for these processors. So far, several solutions that run at the runtime level were introduced to ease programmability of lightweight manycores, such as OpenMP, Partitioned Global Address Space (PGAS) and Message Passing Interface (MPI) [10], [11]. However, these solutions do no effectively enable resource sharing and multiplexing; and they lack on providing rich abstractions.

To expand the applicability of lightweight manycores, as well as to improve software portability and programmability to them, some research initiatives are focused on introducing a rich Operating System (OS) solution for these emerging processors [12]–[14]. Among these efforts, we highlight the multikernel design, which stands out due to its natural match to the architecture of lightweight manycores themselves. In this approach, the OS is structured as a distributed system [15], [16]: a set of independent OS kernels is deployed in the processors; these kernels communicate with one another via message-passing only; and they implement in a distributed fashion traditional OS subsystems.

Several designs for a multikernel OS are possible [17]–[19], but we claim that a design that is structured on top of asymmetric microkernel instances better matches the unique architectural features of a lightweight manycore. We argue in favor of this structure due to multiple reasons. First, the microkernel layout is inherently minimalist and hence leads to a small memory footprint. Second, a microkernel structure enables maximum flexibility. Finally, the asymmetric design grants the microkernel scalability [12].

In this context, the goal of this work is to provide a comprehensive assessment on the asymmetric microkernel design for lightweight manycores. To do so, we implemented a kernel with the aforementioned characteristics in the Nanvix research OS [20], [21], and we evaluated its performance scalability and isolation in the Kalray MPPA-256 processor, using a representative suite of benchmarks. In summary, this work delivers the following contributions to the state of the art on OS kernel construction for lightweight manycores:

We thank CNRS, CNPq, FAPESC, FAPEMIG for supporting this research. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

- Unveil that an asymmetric microkernel design matches the architectural features of a lightweight manycore, delivering performance and isolation to user software.
- Deliver an open-source implementation of an asymmetric microkernel.
- Uncover co-design aspects between an OS kernel and a lightweight manycore.

The remainder of this work is organized as follows. In Section II, we present a background on lightweight manycores and the multikernel OS design. In Section III, we discuss the structure of Nanvix. In Section IV, we detail the evaluation methodology that we adopted. In Section V, we analyze our experimental results. In Section VI, we present related works. Finally, we draw our conclusions in Section VII.

II. Lightweight Manycores

In this section, we cover the background of our work. First, we carry out a discussion on the architecture of lightweight manycores. Then, we highlight how programmability challenges in these processors emerge. Finally, we present the multikernel OS design and argue why they match lightweight manycores. For a discussion on Nanvix, see Section III.

A. Architectural Blueprints

Lightweight manycores differ from other high core count platforms, such as Non-Uniform Memory Access (NUMA) processors and Graphics Processing Units (GPUs), in several points. In contrast to the latter architectures, lightweight manycores: (i) integrate in a single chip hundreds to thousands of low-power cores; (ii) are designed to target Multiple Instruction Multiple Data (MIMD) workloads; (iii) present a distributed memory architecture; (iv) feature a constrained memory system, often with no single address space, small local memories and no cache coherency; (v) rely on high-bandwidth and rich Networks-on-Chip (NoCs) to carry out fast and reliable message-passing communication; and (vi) may have a heterogeneous configuration, in terms of both I/O and computing capabilities. To better understand these key features of lightweight manycores, in the paragraphs that follow, we carry out a detailed discussion on a concept processor of this emerging class. Noteworthy, the following discussion equally applies to real-world lightweight manycores [6]–[8].

Figure 1 pictures an architectural overview of a concept lightweight manycore. Overall, it has 67 cores that are bundled into seventeen different tightly-coupled groups, called clusters (or tiles). Cores within the same cluster share some local hardware resources, such as SRAM and NoC interfaces; and have uniform access latencies to these local components. Note that clusters may differ in their structure, organization and computing capabilities and/or I/O connectivity. In the given example, two types of clusters co-exist:

i. sixteen Compute Clusters, each of which featuring four cores, a SRAM, one NoC interface, and one Direct Memory Access (DMA) engine; and
ii. one I/O Cluster that has three cores, a SRAM, two NoC interfaces, one DMA engine, and connectivity to DRAM and devices.

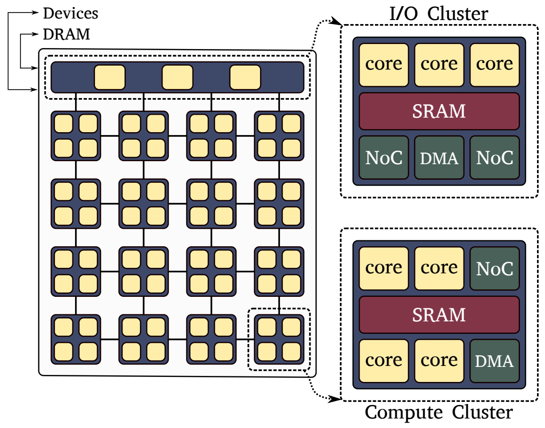

Fig. 1. Concept lightweight manycore processor with 67 cores.

Clusters have distinct address spaces, and they may communicate with one another by explicitly exchanging messages at software-level through a global mesh NoC. For instance, if a core in a given Compute Cluster A wants to communicate with a core lying in another Compute Cluster B, it can do so by sending messages to the core in cluster B. Conversely, to access external devices and DRAM attached to the I/O Cluster, Compute Cluster A must exchange messages with the I/O Cluster. Noteworthy, due to the explicit message-passing communication, some lightweight manycores feature built-in DMA controllers in their NoC interfaces to enable asynchronous communications.

B. Programmability Challenges

Performance capabilities of lightweight manycores can be drawn from their massive number of cores and rich on-chip interconnect. However, their architectural features also introduce challenges in software programmability, which affect both OS construction and user-level application development. First, the *High Density Circuit Integration* turns dark silicon [22] into reality. Since it is not possible to power on all the cores of a lightweight manycore at the same time, thermal-aware scheduling strategies are required to enable the maximum of them to be powered up safely. Second, the *Distributed Memory Architecture* requires software to be designed to handle data partitioning and accessing across multiple physical address spaces. Data should be explicitly fetched from remote memories to local ones, in order to be manipulated, which leads to non-trivial software design [1].

Third, the *Small Amount of On-Chip Memory* requires the software to explicitly tile the working data set into chunks, load/store them from/to a remote memory, and locally manipulate these chunks one at a time [20]. Additionally, it is up to the software to take care of data caching and replication to boost performance. Finally, the *Rich On-Chip Interconnect* exposes mechanisms for asynchronous programming and explicit routing on the chip. The former should be used in order to overlap communication with computation [23]. The latter should be extensively considered in order to guarantee uniform communication latencies [24].

C. The OS Multikernel Design

The multikernel OS design was introduced to address architectural characteristics of lightweight manycores [15]. In this approach, the OS is structured as a distributed system: a set of independent OS kernel instances is deployed in the processor; these kernels communicate with one another via message-passing only; and they implement in a distributed fashion traditional OS subsystems. When it comes to lightweight manycores, the rationale for this structure is three-fold [16]:

i. it relies on state replication instead of sharing, thereby enabling large-scale scalability;
ii. it makes the OS structure hardware-neutral, thus being inherently portable across diverse hardware; and
iii. it deals with inter-core communication explicitly, thus enabling the OS to make more efficient use of the on-chip interconnect by employing network optimizations.

Figure 2 presents a snapshot of a multikernel OS running in a 40-core lightweight manycore. Cores of the underlying processor are either idle (black cores), running an OS Kernel Instance (gray cores), or executing a user application (white cores). OS Kernel Instances are spawned during the system startup and are spatially distributed across the processor. Each OS Kernel Instance may provide a different set of functionalities, and they all rely on agreement protocols to coordinate their work. User applications are dynamically launched on idle cores and rely on Remote Producere Call (RPC) libraries to send requests to OS Kernel Instance.

At this point, it is interesting to highlight how flexible is the multikernel design. At the deployment level, the set of OS Kernel Instances that are started up with the system, as well as the stub RPC libraries that are provided to applications, may be narrowed to a specific use-case domain. For instance, to improve fault tolerance, multiple OS Kernel Instances that provide file system operations may be launched at system start up. Conversely, if the network stack is not required, OS Kernel Instances that implement this functionality do not need to be brought up at system startup. At the implementation level, the actual architecture and capabilities of an OS Kernel Instance may be further tailored to better meet a specific set of target platforms. For instance, OS Kernel Instances may be implemented as a specialized monolithic kernel [19], to address heterogeneous Instruction Set Architecture (ISA); or as an exokernels [15], to enable maximum control over the hardware, by user-level software.

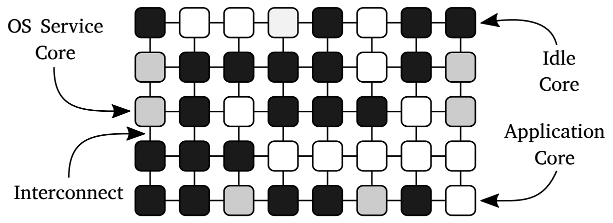

Fig. 2. Snapshot of a multikernel OS running.

In this wide-spectrum of design possibilities for a Multikernel OS, we argue on an implementation based on asymmetric microkernel instances, in order to meet the architectural features of lightweight manycores. The rationale is four-fold:

i. a microkernel layout is inherently minimalist and hence leads to a small memory footprint, which is an important constraint for lightweight manycores;
ii. a microkernel structure enables maximum flexibility, since subsystems may be easily adapted and deployed so as to better meet the requirements of a use-scenario (i.e. no need to change the kernel);
iii. an asymmetric design delivers better isolation to user applications [18]; and
iv. an asymmetric design enables co-design optimizations between the architecture and the OS kernel [8].

III. The Nanvix Operating System

Nanvix[1] is an open-source research OS that targets lightweight manycores and aims at POSIX compliance. It is under constant development in joint collaboration between PUC Minas (Brazil), UFSC (Brazil) and UGA (France); and it currently supports multiple architectures, including Kalray MPPA-256 [6], OpTiMSoC [25] and PULP [3]. Our OS features a multikernel design that is structured on a set of asymmetric microkernel instances. In this section, we cover the main design aspects of Nanvix, which we argue to better meet the architectural characteristics of lightweight manycores. First, we present an overview of the system structure, and then we uncover the internals of the Nanvix microkernel.

A. System Structure Overview

Nanvix is organized in three layers. On the bottom layer, one microkernel instance is deployed on each cluster. Our microkernel features an asymmetric design, which means that it exclusively runs in one core of the cluster, and it leaves the remaining cores to general-purpose use. Overall, the microkernel provides resource sharing and security, as well as bare minimum OS abstractions and primitives, at cluster level.

On the mid-layer, system servers are launched in some cores that were not used by the kernel. These servers implement abstractions, primitives and services commonly found in an OS, such as process scheduling, memory mappings, and files table. Subsystems such as process management, memory management and file system are thus implemented by a set of these system servers that work together, in a distributed fashion. Noteworthy, the internal organization and the number of system servers that compose each subsystem may vary, from one subsystem to another, as well as each underlying platform.

Finally, on the top layer, there are the runtime libraries. These libraries are linked to the user applications and provide a standard interface for user-level software to interact with the OS. For instance, to create a process, a user application would rely on a RPC library call of the process management subsystem, which would be forwarded to the right system server.

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

B. Microkernel

Figure 3 presents an overview of the Nanvix microkernel. It features an asymmetric design, which means that it exclusively runs in one core of the underlying cluster. The Nanvix microkernel is structured into three layers, from bottom to top: the *Hardware Abstraction Layer (HAL)*, the *Modules Layer* and the *Kernel Call Layer*. The *HAL* enables the portability of the kernel itself, by abstracting the hardware and exposing a uniform interface to higher levels of the microkernel across multiple lightweight manycores. For a detailed discussion on the Nanvix HAL, refer to our previous work [21].

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

The *Communication Facility* provides three inter-cluster communication primitives: (i) *syncs* which enable mutual exclusion; (ii) *mailboxes* which are meant to exchange small and fixed-size messages; and (iii) *portals* which enable one-sided dense data transfers. All these primitives have a synchronous behavior. However, for *mailboxes* and *portals*, asynchronous mode of operation is also supported in lightweight manycores that feature DMA engines. Finally, the *Memory Management Module* provides a virtual memory extension to cluster-level. It is based in a two-level paging scheme, supports pages of heterogeneous sizes and uses a capabilities system [16] to keep track of permissions on pages. Furthermore, to enable address space management in user space, this module exposes routines for inspecting and updating paging structures. The *Kernel Call Layer* lies on top of the *Modules Layer* and effectively exposes the functionalities of each module to user space. This topmost layer performs security checking, controls execution flow, and, more importantly, it handles the actual asymmetric characteristic of our microkernel design.

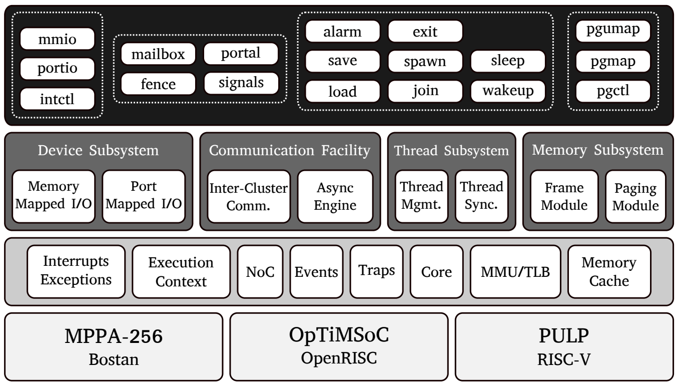

Fig. 3. Structural overview of the Nanvix microkernel.

In our asymmetric microkernel design, one core of the underlying cluster, called the master core, executes the complex routines of the *Module Layers*, and simple routines of the kernel are executed in the other cores, called slave cores. The controller module of the *kernel Call Layer* decides whether or not a given kernel call should be executed locally (i.e. in the slave core), or remotely (i.e. in the master core) based on a two-fold condition. If the kernel call changes structures of the requesting core, or if it accesses only read-only data structures of the kernel, then this kernel call executes locally, otherwise it executes remotely. For serving remote kernel calls, the master core processes them in a FIFO fashion, and it relies on a kernel-land semaphore to control the requests coming from user-threads. With this design, remote kernel calls are expected to have a higher latency than local ones. Still, in Section V, we show that this approach leads to scalability and isolation.

IV. Evaluation Methodology

To deliver a comprehensive assessment of an asymmetric microkernel design for lightweight manycores, we studied the execution of several representative benchmarks in Nanvix running on the Kalray MPPA-256 processor. In this section, we present our evaluation methodology, and in the next one, we unveil our experimental results and outcomes. First, we detail the experimental benchmarks that we employed. Then, we give further information on Kalray MPPA-256. Finally, we discuss about our experimental design.

A. Experimental Benchmarks

We employed five benchmarks in our assessment. These programs were selected so as to exercise important design aspects of an asymmetric microkernel: (i) the *performance* of core mechanisms of the kernel itself; (ii) the *interference* of the kernel in the execution of user-level applications; and (iii) the *scalability* of the kernel to serve user-level requests (i.e. kernel calls). Overall, we partitioned these five benchmarks into two groups, to ease our discussion: µ*Benchmarks* and *Synthetic Benchmarks*. In the first group, we intended to evaluate the upper bound *performance* of fundamental mechanisms of an asymmetric microkernel design. In contrast, in the *Synthetic Benchmarks*, we aimed at assessing the *scalability* and *isolation* of the kernel. In the paragraphs that follow, we present a short description of each of these benchmarks. Noteworthy, we made this application suite available[2] for a detailed study.

Local Kernel Call (L-Kcall) This µ*benchmark* assesses the performance of a local kernel call, which is one that executes in the same core that the requesting user thread is running. We used the `kthread_self()` kernel call

[2]Publicly available at: https://github.com/nanvix/microkernel-benchmarks.

of Nanvix in this benchmark, which retrieves the identifier of the underlying kernel thread.

Remote Kernel Call (R-Kcall) This µ*benchmark* assesses the performance of a remote kernel call. Remote kernel calls execute in a dedicated core (i.e. master core) of the underlying cluster, and they consist in the key feature of asymmetric OS kernels. In this benchmark, we considered a void remote kernel call of Nanvix.

Fork-Join This *synthetic benchmark* assesses the overhead for creating and terminating kernel threads. The cost for supporting these operations is important, if the OS aims at efficient support for multithread frameworks, such as Java Concurrency and POSIX Threads.

Buffer This *synthetic benchmark* launches multiple thread pairs that perform buffered transfers. The actual transfer of data between each thread pair relies on: (i) two semaphores for flow control; (ii) a ring buffer to hold temporary data; and (iii) a mutex for mutual exclusion to the ring buffer. Essentially, this use-case scenario benchmarks contention in the kernel to serve kernel calls. Since operations in semaphores and mutexes in Nanvix execute in the master core, this benchmark directly evaluates the scalability of the asymmetric kernel design.

Kernel Noise (KNoise) This synthetic benchmark evaluates the interference introduced by the kernel in the execution of a user application, in the possible worst-case scenario. Let n be the number of cores available in the underlying cluster. Therefore, this benchmark launches x compute-intensive threads in x of these cores, and in the $n - x$ remainder cores it launches kernel-intensive tasks (i.e. threads that issue remote kernel calls in a tight loop).

B. Experimental Platform and Design

In this work, we benchmarked the Nanvix Microkernel in the Kalray MPPA-256 processor [6]. This is a commercial lightweight manycore that features the major characteristics that we discussed in Section II-B. We chose this platforms over the others that are supported by Nanvix, because it would enable us to assess our asymmetric microkernel design in an existing industrial solution and thus uncover some valuable co-design aspects between the hardware and OS kernel.

Kalray MPPA-256 features 272 general-purpose cores, named Processing Elements (PEs), and 16 firmware cores, called Resource Managers (RMs). All cores (i.e. PEs and RMs) run at fixed frequency of 400 MHz, implement a 64-bit proprietary instruction set, present a 5-issue Very Long Instruction Word (VLIW) pipeline, 8 kB instruction and data caches, and feature a software-managed Memory Management Unit (MMU). Overall, the 288 cores of the processor are grouped within 16 Compute Clusters, which are intended for computation, and 4 I/O Clusters, which are designed to provide connectivity to peripherals. Each Compute Cluster bundles 16 PEs, 1 RM, a 2 MB of local SRAM, one DMA engine, and two NoC interfaces. In these clusters, cache coherence is not supported by the hardware. On the other hand, each I/O Cluster has 4 PEs, 4 MB of SRAM, four DMA engines, and eight NoC interfaces. Two of these clusters are connected to a DDR controller each, and the other two are attached to PCI and Ethernet controllers.

TABLE I
BENCHMARK PARAMETERS FOR EXPERIMENTS.

Benchmark	Group	Threads	Workload
L-Kcall	µBenchmarks	1	-
R-Kcall		1	-
Fork-Join	Synthetic	1 to 14	-
Buffer	Benchmarks	2, 6, 8, 10, 14	1 to 8 kB objects
Knoise		1 to 14	1000 FOPs

Clusters have distinct physical address spaces, and they may communicate with one another only by exchanging messages via either one of two different interleaved 2-D torus NoCs: (i) a Control NoC (C-NoC) that features low bandwidth and is intended for small data transfers; and (ii) a Data NoC (D-NoC) that presents high bandwidth and is dedicated to dense data transfers. In our experiments, we considered the execution of the Nanvix microkernel in Compute Clusters.

What concerns software development in Kalray MPPA-256, the processor is shipped with a patched version of GCC 4.9.4 and Binutils 2.11.0. Furthermore, specifically regarding OS kernel implementation, OS engineers are required to rely on a proprietary hypervisor from Kalray. This hypervisor runs on the firmware cores of the processor and provides some low-level routines for dealing with the hardware. Noteworthy, Kalray Hypervisor imposes additional challenges in OS kernel construction, because it was designed to host runtime libraries.

Table I details the parameters used for each benchmark. In both groups of experiments, we used performance counters to monitor hardware events with minimum overhead. We conducted experiments in the Nanvix Microkernel commit 0a0088b, built with level three optimizations. With this build, the memory footprint of the Nanvix Microkernel was 128 kB, plus 320 kB of the Kalray Hypervisor. For each experimental configuration, we carried out several replications to ensure 95% of confidence in our results.

V. Experimental Results

In this section, we present our experimental results. First, we analyze the outcomes for *Microbenchmarks*, and next we move to a discussion on the results for the *Synthetic Benchmarks*.

A. Microbenchmarks

Figure 4 pictures the breakthrough of execution cycles for the *L-Kcall Benchmark*. Before proceeding with our analysis, it is important to note that hardware events that are depicted are not exclusive with each other. For example, an *I-Cache Stall* may incur in a *Register Stall*, thus the bar labeled *Total Cycles* is not the aggregation of the other bars. Noteworthy, this observation applies to the other plots that follow.

Overall, we observed that the Kalray Hypervisor accounts for an important amount of execution statistics, which suggests a hot spot for improvement. Besides this, we noted that the cost for local kernel calls is about 766 cycles, from which

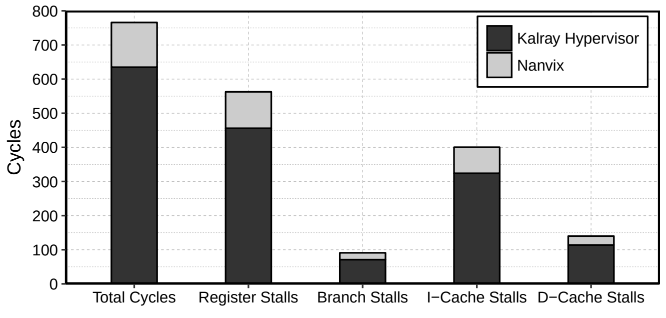

Fig. 4. Execution breakthrough for the *L-Kcall Benchmark*.

164 cycles are required for saving and restoring machine registers upon mode switches. Execution stalls caused by the branch unit and d-cache account for 11.88% and 18.28% of total execution cycles, respectively. Together, these two metrics suggest that the actual overhead for issuing local kernel calls is low. The complex execution flow does not cause the branch unit to perform badly; and the mode switch does not introduce enough stalls in the d-cache so as it becomes the execution bottleneck. On the other hand, stalls caused by the register file and the i-cache account for 73.50% and 52.22% of execution cycles, respectively. This behavior unveils that local kernel calls impose substantial pressure in the i-cache, which incurs in a great number of misses. We analyzed the code of this benchmark, and we found out that it is small enough to fit in the i-cache of the underlying core. Therefore, this behavior of the i-cache is due to either: (i) its small associativity (2-way); or (ii) bad performance of the prefetching buffer unit.

Figure 5 presents the breakthrough of execution cycles for the *R-Kcall Benchmark*. Again, we observed that stalls on register file accounted for 68.4% of the execution cycles. Nevertheless, we found out that this behavior was caused by stalls on both i-cache and d-cache, which added up to 40.51% and 38.12% of execution cycles, respectively. For the i-cache behavior, we carried out the same code analysis that we did for the *L-Kcall Benchmark*, and we reached out the same conclusions. However, for stalls caused by the data cache we found out a different rationale. By design, a remote kernel call requires inter-core communication and synchronization, and from the perspective of the d-cache, some coherence traffic arises. In this scenario, the performance issue thus emerges from the fact that hardware cache coherence is not supported in Kalray MPPA-256. To deal with this problem in software level, Kalray MPPA-256 features instructions to flush, discard and wait for changes to the d-cache of the underlying core, which Nanvix makes use of. Notwithstanding, due to a lack of context information, this is not enough for the kernel to either run a fine-grain data prefetch algorithm or a coherency protocol. Therefore, in order to ensure correct execution semantics, the kernel issues a full cache shoot-down, just before starting a kernel call and before returning from it, thereby causing performance drops.

B. Synthetic Benchmarks

So far, we studied the performance of two fundamental mechanisms of an asymmetric kernel design: local and remote kernel calls. In this section, we move our discussion to the scalability analysis of our approach as well as the interference of the kernel in the execution of the user application.

Figure 6 presents experimental results for the *Buffer Benchmark*. In the plot, we picture the memory bandwidth speedup when increasing the number of simultaneous buffered transfers. In addition, note that, for purpose of comparison, we also plot optimum increase results for this benchmark. This benchmark consists in a representative use-case of the kernel (i.e. buffered transfers) and enables us to assess the scalability of our design, since for each buffered transfer we rely on kernel calls (i.e. semaphores and mutexes) to synchronize pairs. If an asymmetric structure for the kernel is scalable, then the achieved memory bandwidth increase should be close from the optimal one. An analysis of our results unveiled precisely the performance that we sought in our design. When up to 4 simultaneous transfers take place, linear scalability is achieved, and beyond this point, we observe a 95% optimum performance (7 simultaneous buffered transfers). At this point, we highlight one co-design aspect that we are currently aiming. Recall that clusters of the underlying experimental platform feature a moderate granularity (i.e. 16 cores per cluster). Even though we do achieve near-optimum increase when using all the cores, we are interested in investigating co-design aspects that would offer a good compromise between the dimensions of a cluster (i.e. number of cores, local memory) and the achievable performance in representative use-case scenarios.

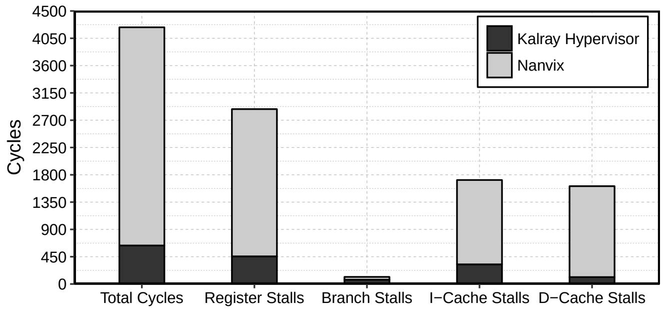

Fig. 5. Execution breakthrough for the *R-Kcall Benchmark*.

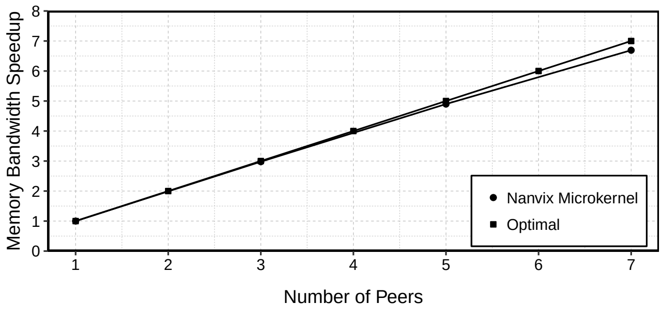

Fig. 6. Memory bandwidth increase for the *Buffer* Benchmark.

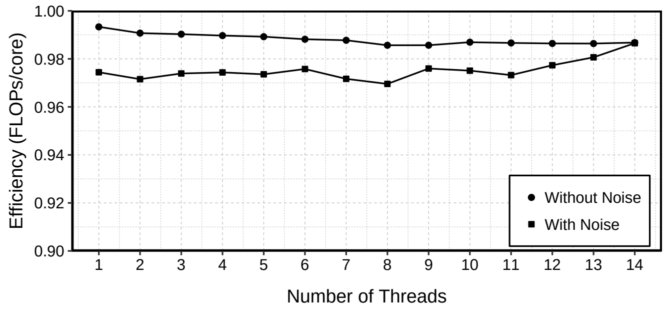

Fig. 7. Execution efficiency for the *KNoise Benchmark*.

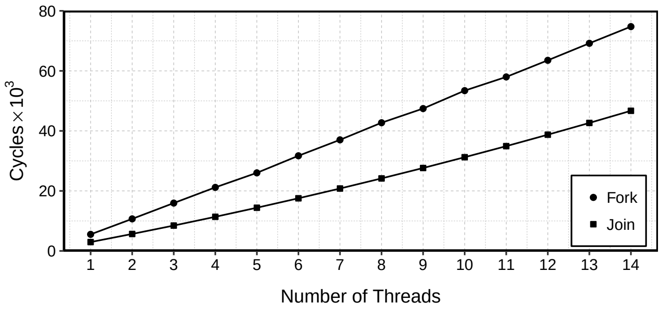

Fig. 8. Performance scalability for the *Fork-Join* Benchmark.

Figure 7 depicts results for the *KNoise Benchmark*. In the plot, we present the efficiency sustained per thread when increasing the number of threads that run a compute-intensive task. Note that, for purpose of comparison, we also plot results when running this benchmark without concurrently launching kernel-intensive tasks in idle threads (*without noise* line in the plot). Remind that this benchmark provides the worst-case evaluation for the execution interference in our asymmetric microkernel design: when kernel threads run the compute-intensive workload, all the remainder kernel threads execute the kernel-intensive one. Overall, our results uncover the major property that we claimed about an asymmetric kernel: low interference in the execution of user-level threads. When a single thread runs a compute-intensive workload, and the remainder 13 threads a kernel-intensive one, the efficiency achieved is 0.9% below from the one that we could achieve with no kernel-intensive tasks. Furthermore, we observed that when the number of compute-intensive threads increases, the performance interference gradually drops down to zero.

Figure 8 presents scalability results for the *Fork-Join Benchmark*. In the plot, we depict the costs for creating (i.e. *Spawn*) and terminating (i.e. *Join*) kernel threads, in means of cycles. Recall that this benchmark provides a representative use-case assessment of our microkernel to support multithreaded frameworks. If efficiency is aimed, overheads should be constant. In general, our results unveiled precisely the desired performance behavior: when increasing the number of kernel threads that we spawn and join, costs grow linearly with the number of threads. In average, for spawning and terminating a thread, we observed an overhead of 5.14 k cycles (12.86 µs) and 3.42 k cycles (8.57 µs), respectively. With these results, we also uncovered three other observations that are worthy to point out. First, the latency for spawning and terminating a thread, which is 21.43 µs, gives insights on the cost for supporting frameworks based on the thread pop-up model, such as web servers. Second, we observed a performance gap of about $1.5\times$ between the two operations. We found out that the reason for this is the synchronous acquisition of resources in spawn, in contrast to the asynchronous release of resources in join. To decrease this performance gap, we intend to introduce thread recycling in Nanvix. Finally, the linear spawn/join scalability further shows that our asymmetric design is scalable, since both operations are implemented as remote kernel calls.

VI. Related Work

The design of Nanvix was greatly influenced by other multikernel OSs, but still our approach features differences that we briefly discuss next. The Barrelfish [16] multikernel OS is structured on top of highly optimized kernels, and in Nanvix we follow a similar approach. Our kernel is built upon a rich HAL that is designed to expose unique features of lightweight manycores [21]. Nevertheless, unlike Barrelfish, our multikernel OS does not maintain a single coherent state across multiple kernel instances. Instead, Nanvix delivers further flexibility and handles the application the possibility to choose, at runtime level, between a scalable distributed view or a simpler, but slower, single and coherent view. In this point, Nanvix thus shares ideas with FOS [15], in which OS functionalities are implemented by system servers. However, in contrast to FOS, which targets cloud computing scenarios, Nanvix aims lightweight manycores and is structured on top of asymmetric microkernels. With our structure: (i) we keep the memory footprint lower, since kernel code and data is not replicated to all cores; and (ii) we decrease the kernel interference, because caches are more efficiently used.

Like in MOSSCA [17] and Tessellation [26], Nanvix relies on the clustered organization of the target processors to provide core partitioning and application isolation. However, Nanvix aims lightweight manycores with a distributed memory architecture, which introduces an extra challenge on the resource partitioning algorithms. Similar to Popcorn Linux [19] and Helios [18], Nanvix seeks for addressing heterogeneity in a processor and exposing a standard interface to applications. Nevertheless, different from the former OS, our solution does not rely on compile-time analysis nor is restricted to ISA heterogeneity. On the other hand, in contrast to Helios, Nanvix features a distributed multikernel design, instead of a hierarchical one. Hence, Nanvix is inherently more scalable and fault-tolerant; once there is no master kernel instance which could be the bottleneck or point of failure.

VII. Conclusions

Resource management in lightweight manycores is required to enable their applicability in multi-user/application contexts. On this goal, several research initiatives are focused [10]–[14]. Nonetheless, in this work, we highlight the multikernel OS layout, due to its match to the aforementioned architectures.

Several structures for a multikernel OS are possible [17]–[19], but we argue on one that is structured on top of asymmetric microkernel instances. We deliver an open-source implementation of an OS kernel with the aforementioned characteristics and we provide a comprehensive assessment of such design in the Kalray MPPA-256 processor, using a representative benchmark suite. In our experiments, we assess the performance and isolation of our kernel. Our results unveiled that the benchmarked design achieves linear scalability to serve kernel calls, and it introduces at most 0.9% of performance interference in the execution of an application. Furthermore, our experiments unveiled three co-design investigations that may be pushed, between the OS kernel and lightweight manycores. First, the investigation of a better associativity and/or a smarter pre-fetching algorithm for the instruction cache, to reduce the stalls in the instruction cache in kernel calls. Second, hardware support for selective remote cache shoot-down should be studied, so as to reduce the stalls in the data cache when serving remote kernel calls. Finally, investigations towards the dimensioning of a cluster should be pushed, to find a compromise between the number of cores, the amount of local memory and achievable performance.

In future work, we intend to carry out the aforementioned co-design investigations with Nanvix and the academic lightweight manycores that it supports, which are OpTiMSoC and PULP. Furthermore, we intend to progress on our multikernel OS construction that is based in the asymmetric microkernel design, which we present and benchmark in this work.

References

[1] E. Francesquini, M. Castro, P. H. Penna, F. Dupros, H. Freitas, P. Navaux, and J.-F. Méhaut, "On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms," *Journal of Parallel and Distributed Computing*, vol. 76, no. C, pp. 32–48, Feb. 2015.

[2] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "KiloCore: A 32-nm 1000-Processor Computational Array," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 891–902, 2017.

[3] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gurkaynak, A. Teman, J. Constantin, A. Burg, I. Miro-Panades, E. Beigne, F. Clermidy, P. Flatresse, and L. Benini, "Energy-efficient near-threshold parallel computing: The pulpv2 cluster," *IEEE Micro*, vol. 37, no. 5, pp. 20–31, sep 2017.

[4] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a Many-Core Computing Accelerator for Embedded SoCs," in *Design Automation Conf.*, New York, USA, jun 2012, p. 1137.

[5] S. Davidson, S. Xie, C. Torng, K. Al-Hawai, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor, "The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips," *IEEE Micro*, vol. 38, no. 2, pp. 30–41, mar 2018.

[6] B. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. de Massas, F. Jacquet, S. Jones, N. Chaisemartin, F. Riss, and T. Strudel, "A clustered manycore processor architecture for embedded and accelerated applications," in *Int. Conf. on High Performance Extreme Computing*, Waltham, USA, 2013, pp. 1–6.

[7] A. Olofsson, T. Nordstrom, and Z. Ul-Abdin, "Kickstarting high-performance energy-efficient manycore architectures with epiphany," in *Asilomar Conf. on Signals, Systems and Computers*, november 2014, pp. 1719–1726.

[8] F. Zheng, H.-L. Li, H. Lv, F. Guo, X.-H. Xu, and X.-H. Xie, "Cooperative Computing Techniques for a Deeply Fused and Heterogeneous Many-Core Processor Architecture," *Journal of Computer Science and Technology*, vol. 30, no. 1, pp. 145–162, Jan. 2015.

[9] H. Fu, W. Yin, G. Yang, X. Chen, C. He, B. Chen, Z. Yin, Z. Zhang, W. Zhang, T. Zhang, W. Xue, and W. Liu, "18.9-Pflops Nonlinear Earthquake Simulation on Sunway TaihuLight: Enabling Depiction of 18-Hz and 8-Meter Scenarios," in *Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, Denver, Colorado, nov 2017, pp. 1–12.

[10] J. Ross and D. Richie, "Implementing OpenSHMEM for the Adapteva Epiphany RISC Array Processor," *Procedia Computer Science*, vol. 80, no. C, pp. 2353–2356, jan 2016.

[11] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel, "A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor," *Procedia Computer Science*, vol. 18, no. 2013 Int. Conf. on Computational Science, pp. 1654–1663, jan 2013.

[12] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An Operating System for Many Cores," in *USENIX Conf. on Operating Systems Design and Implementation*, San Diego, USA, dec 2008, pp. 43–57.

[13] B. Rhoden, K. Klues, D. Zhu, and E. Brewer, "Improving Per-Node Efficiency in the Datacenter with New OS Abstractions," in *ACM Symp. on Cloud Computing*, Cascais, Portugal, Oct. 2011, pp. 1–8.

[14] R. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen, "mOS: An Architecture for Extreme-Scale Operating Systems," in *Int. Workshop on Runtime and Operating Systems for Supercomputers*, Munich, Germany, Jun. 2014, pp. 1–8.

[15] D. Wentzlaff and A. Agarwal, "Factored Operating Systems (FOS): The Case for a Scalable Operating System for Multicores," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 76–85, Apr. 2009.

[16] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *ACM Symp. on Operating Systems Principles*, Big Sky, USA, Oct. 2009, pp. 29–44.

[17] F. Kluge, M. Gerdes, and T. Ungerer, "An Operating System for Safety-Critical Applications on Manycore Processors," in *Int. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing*, Reno, USA, Jun. 2014, pp. 238–245.

[18] E. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: Heterogeneous Multiprocessing with Satellite Kernels," in *ACM Symp. on Operating Systems Principles*, Oct. 2009, pp. 221–234.

[19] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms," in *European Conf. on Computer Systems*, Bordeaux, France, Apr. 2015, pp. 1–16.

[20] P. H. Penna, M. Souza, E. Podestá Jr, J. Souto, M. Castro, F. Broquedis, H. Freitas, and J.-F. Mehaut, "Rmem: An os service for transparent remote memory access in lightweight manycores," in *Int. Workshop on Programmability and Architectures for Heterogeneous Multicores*, Valencia, Spain, jan 2019, pp. 1–16.

[21] P. H. Penna, D. Francis, and J. Souto, "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, jun 2019, pp. 1–11.

[22] M.-H. Haghbayan, A. Miele, A. M. Rahmani, P. Liljeberg, and H. Tenhunen, "Performance/Reliability-Aware Resource Management for Many-Cores in Dark Silicon Era," *IEEE Transactions on Computers*, vol. 66, no. 9, pp. 1599–1612, sep 2017.

[23] J. Hascoët, B. D. de Dinechin, P. G. de Massas, and M. Q. Ho, "Asynchronous One-Sided Communications and Synchronizations for a Clustered Manycore Processor," in *Symp. on Embedded Systems for Real-Time Multimedia*, Seoul, oct 2017, pp. 51–60.

[24] B. D. de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti, "Guaranteed Services of the NoC of a Manycore Processor," in *Int. Workshop on Network on Chip Architectures*, Cambridge, 2014, pp. 11–16.

[25] S. Wallentowitz, P. Wagner, M. Tempelmeier, T. Wild, and A. Herkersdorf, "Open Tiled Manycore System-on-Chip," ArXiV, Tech. Rep. arXiv:1304.5081, Apr. 2013.

[26] J. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf, K. Asanović, and J. Kubiatowicz, "Resource Management in the Tessellation Manycore OS," in *USENIX Conference on Hot Topics in Parallelism*. Berkeley, USA: USENIX Association, jun 2010.