Inter-Kernel Communication Facility of a Distributed Operating System for NoC-Based Lightweight Manycores

Pedro Henrique Penna, João Vicente Souto, João Fellipe Uller, Márcio Castro, Henrique Freitas, Jean-François Méhaut

Inter-Kernel Communication Facility of a Distributed Operating System for NoC-Based Lightweight Manycores

Pedro Henrique Penna[a,b], João Vicente Souto[c], João Fellipe Uller[c], Márcio Castro[c], Henrique Freitas[a], Jean-François Méhaut[b]

[a]*Pontifícia Universidade Católica de Minas Gerais (PUC Minas), Belo Horizonte, Brazil*
[b]*Université Grenoble Alpes (UGA), Grenoble, France*
[c]*Universidade Federal de Santa Catarina (UFSC), Florianópolis, Brazil*

Abstract

Lightweight manycore processors deliver high performance and scalability by bundling in a single chip hundreds of low-power cores, a distributed memory architecture and Networks-on-Chip (NoCs). Operating Systems (OSes) for these processors feature a distributed design, in which a communication layer enables kernels to exchange information and interoperate. Currently, this communication infrastructure is based on *mailboxes*, which enable fixed-size message exchanges with low latency. However, this solution is suboptimal because it can neither fully exploit the NoC nor efficiently handle the diversity of OS communication protocols. We propose an Inter-Kernel Communication (IKC) facility that exposes two kernel-level communication abstractions in addition to *mailboxes*: *syncs*, for enabling a process to signal and unlock another process remotely, and *portals*, for handling dense data transfers with high bandwidth. We implemented the proposed facility in Nanvix, the only open-source distributed OS that runs on a baremetal lightweight manycore, and we evaluated our solution on a 288-core processor (Kalray MPPA-256). Our results showed that our IKC facility achieves up to 16.87× and 1.68× better performance than a *mailbox*-only solution, in synchronization and dense data transfers, respectively.

Keywords: Lightweight Manycore Processor, Network-On-Chip, Distributed Operating System, Message-Passing Communication
2020 MSC: 68M14, 68N25

1. Introduction

Lightweight manycore processors were introduced to address the ever-increasing scalability and low-power consumption demands of parallel applications [1]. They rely on specific architectural characteristics to achieve the former requirement, such as a distributed memory architecture and a rich Network-on-Chip (NoC) [2]. In addition, to improve the energy efficiency, they are built with simple low-power Multiple Instruction Multiple Data (MIMD) cores [3], they have a memory system based on Scratchpad Memories (SPMs) with no hardware coherency support [4] and they exploit heterogeneity by featuring cores with different capabilities [5]. Some industry-successful examples of these processors are the Kalray MPPA-256 [6], the Adapteva Epiphany [7] and the Sunway SW26010 [8], being the latter employed in Sunway TaihuLight [9], currently the world's fourth most powerful supercomputer according to TOP500[1].

Operating Systems (OSes) for lightweight manycores embrace a distributed design to cope with the high core count and the distributed memory architecture. This approach enables the system to scale up to thousands of cores [10, 11], while exposing richer abstractions and Application Programming Interfaces (APIs) to user-level software. Overall, subsystems of a distributed OS for a lightweight manycore (e.g., memory manager, process manager and file system manager) are deployed across the cores of the processor and they communicate with one another by exclusively exchanging data through the underlying NoC [12, 13].

The communication layer of a distributed OS for lightweight manycores resembles a simplified version of a traditional distributed OS for clusters of workstations. It handles multiplexing of the communication channel, security, routing, network congestion, message addressing, control flow, reordering and network packaging. In contrast to distributed OSes that target clusters of workstations, it does not handle communication errors, since the underlying inter-

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

connect is reliable. To enable subsystems to communicate with low-latency and reliably, distributed OSes for lightweight manycores rely on a *mailbox* abstraction [10, 11, 12, 13]. This specialized system-level structure exposes send/receive primitives and enable fixed-size data exchanges.

Although *mailboxes* enable system-level communications in lightweight manycores, they may be inefficient. Overall, we argue on this limitation based on two observations. First, the capabilities of the NoC are not fully exploited. For instance, these interconnects oftentimes feature special hardware units for efficiently supporting different types and granularities of communications, such as small/large data transfers and synchronization signals. Since the right use of each of these hardware resources depends on the semantics of the communication and this information is unavailable at the communication layer, the NoC capabilities are not fully exploited. Second, the existing diversity in communication protocols across the different subsystems calls out for multiple abstractions, otherwise the implementation of protocols is inefficient. For example, services that manipulate coarse-grain data (e.g., memory manager and file system manager) require control flow to avoid a single client process to hug all the bandwidth of the service. In contrast, subsystems that manipulate fine-grain data (e.g., process manager) do not usually need control flow because they exchange small fixed-size messages, so control flow is implicitly handled. Therefore, if the same abstraction (i.e., *mailboxes*) is used for implementing both protocols, overhead is unnecessarily imposed in the latter one.

To overcome this problem, we claim that the communication layer of distributed OSes for lightweight manycore processors should provide additional abstractions with richer semantics. Thus, not only the NoC capabilities may be better exploited but also the communication characteristics between different subsystems may be better addressed. In this context, this work proposes an Inter-Kernel Communication (IKC) facility that: (i) abstracts on-chip communication capabilities of lightweight manycores; (ii) provides virtualization and multiplexing of the NoC links; and (iii) enforces communication security at kernel-level. In summary, this work delivers the following contributions to the state-of-the-art in OS support for lightweight manycores:

- A richer kernel-level communication facility for lightweight manycores in a distributed OS. Overall, this facility exposes three communication abstractions: (i) *syncs*, for enabling a process to signal and unlock another process remotely; (ii) *mailboxes*, for sending fixed-size messages with low latency; and (iii) *portals*, for handling dense data transfers with high bandwidth.

- An implementation and integration of the proposed communication facility in Nanvix [14], the only open-source distributed OS to date that runs on a baremetal lightweight manycore.

We evaluated the proposed facility using two complementary approaches. First, we used micro-benchmarks to study the raw performance and scalability of our abstractions. Then, we relied on a benchmark suite that exercises important communication protocols that are applied in different subsystems of Nanvix. We carried out baremetal experiments on Kalray MPPA-256, a NoC-based lightweight manycore processor that features a distributed memory architecture and integrates 288 cores in a single chip. Overall, our results showed that our communication facility enables up to 16.87× and 1.68× better performance than a *mailbox*-only solution, in synchronization and dense data transfers, respectively.

The remainder of this work is organized as follows. In Section 2, we cover the background on lightweight manycore processors and distributed OSes. Then, we present our IKC facility (Section 3) and we detail how we implemented it in Nanvix (Section 4). In Section 5, we present our evaluation methodology, which is then applied in Section 6 to analyze the experimental results. Related works are presented and discussed in Section 7. Finally, Section 8 concludes this paper.

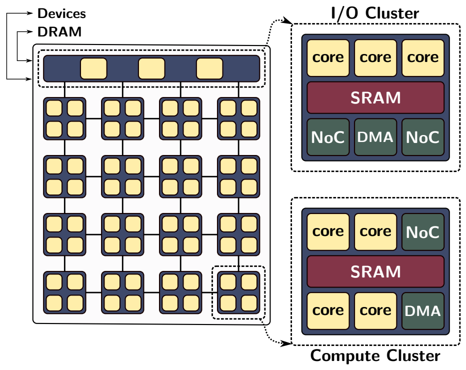


Figure 1: A conceptual lightweight manycore processor with 67 cores.

2. Background

In this section we cover the background of our work. First, we discuss about lightweight manycores. Then, we present an overview of distributed OSes that target these architectures.

2.1. Lightweight Manycore Processors

Lightweight manycore processors have an endeavor to deliver high performance with energy efficiency. To this end, they rely on the following features:

(i) they integrate up to thousands of low-power cores in a single die;

(ii) they are designed to cope with MIMD workloads;

(iii) they have their cores disposed in tightly-coupled groups;

(iv) they feature a constrained memory system, with a distributed memory architecture and small local memories;

(v) they rely on one or more high-bandwidth NoCs for fast and reliable message-passing communication; and

(vi) they have a heterogeneous configuration in terms of I/O and/or computing capabilities.

To better understand these key features, we present a detailed discussion on a simplified concept processor (Figure 1). Nevertheless, the following discussion equally applies to any other baremetal lightweight manycore available [6, 15, 9]. The concept processor presented in Figure 1 integrates 67 cores in a single chip, which in turn are disposed into 17 tightly-coupled groups (called *clusters*). Inside a cluster, cores share some local hardware resources, such as a local Static Random Access Memory (SRAM) and a NoC interface, and have uniform access latencies to these components. Clusters may have different characteristics, such as processing power, local memory sizes and communication capabilities, so that they are used to accomplish different goals. For instance, in this example, the I/O Cluster is specialized in communications with external memories and devices, while Compute Clusters are meant for processing user workloads.

Clusters have distinct address spaces, and they may communicate with one another by explicitly exchanging hardware-level messages through the NoC. To write to external devices and to the Dynamic Random Access Memory (DRAM) attached to the I/O Cluster, Compute Clusters must tile in software the output data into hardware-level messages and transfer them through the NoC to the I/O Cluster. To enable asynchronous communications and provide higher bandwidth for dense data transfers, lightweight manycores may feature built-in Direct Memory Access (DMA) engines in their NoC interfaces.

The aforementioned set of architectural features highlight important distinctions between lightweight manycores and other well-known manycore processors:

- Manycore processors such as Intel Xeon Phi, Tilera TILE-Gx100 and Intel Single-Cloud Computer do not have a constrained memory system, with small local memories that are physically distributed across the clusters;
- Symmetric Multiprocessing (SMP) architectures based on Non-Uniform Memory Access (NUMA) design are built with multiple CPU packages that communicate with one another through a dedicated interconnect hardware outside of the processor chip (e.g., NUMAlink or NumaConnect); and

- Graphics Processing Units (GPUs) are not designed to cope with MIMD workloads.

Overall, these differences make lightweight manycores more scalable in terms of performance and energy efficiency. However, they also introduce challenges in software programmability, which affects both OS construction and user-level application development. Some of the challenges are described bellow:

High density circuit integration turns dark silicon [16] into reality. Since all cores of a lightweight manycore may not be powered on at the same time in certain situations, thermal-aware scheduling strategies are required to achieve high performance.

Distributed memory architecture requires software to be designed to handle data partitioning and remote data accesses 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].

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 [17]. Additionally, it is up to the software to take care of data caching and replication to boost performance.

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 [18]. The latter should be extensively considered in order to guarantee uniform communication latencies [19].

Missing cache coherence in hardware forces programmers to handle data coherency explicitly in software and frequently calls out for a redesign in their applications [1].

Heterogeneous configuration turns the actual deployment of system functionalities and applications into a complex task [20].

2.2. Distributed Operating Systems

Distributed OSes were initially introduced to cope with performance scalability problems of traditional single-chip OS designs [11, 10]. More recently, they have been used to address the challenges in software development and deployment in lightweight manycore processors [12, 13, 14].

Figure 2 pictures a distributed OS running on a parallel architecture. In this approach, 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. In this figure, groups of cores represent either *clusters* of a lightweight manycore or *processors* of an SMP architecture (e.g., NUMA). Multiple architectures and implementations for a distributed OS are possible, each one targeting a specific set of design goals and constraints. Nevertheless, we highlight here a three-tier approach that is commonly adopted by several distributed OSes such as Barrelfish [11], FOS [10], HeliOS [21], MOSSCA [12], M^3 [13] and Nanvix [14].

Hardware Abstraction Layer (HAL) In the bottom layer, a generic and flexible abstraction of the underlying hardware is provided, so that portability across different processor architectures is enabled. This is either implemented by baremetal supporting libraries or by an exokernel.

OS Kernel An OS kernel lies in the middle layer, which provides minimum

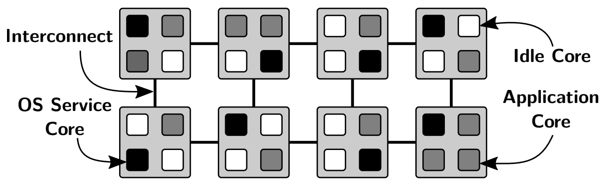


Figure 2: A distributed OS running on a parallel architecture. Groups of cores represent either *clusters* of a lightweight manycore or *processors* of an SMP architecture.

system abstractions (e.g., threads), handles local resource multiplexing and ensures security policies. Any kernel construction design may be applied at this level (e.g., nanokernel, microkernel or monolithic kernel).

Runtime OS Libraries In the top layer, runtime OS libraries expose a standard interface such as the Portable Operating System Interface (POSIX) for interacting with the OS. These libraries are linked with user-level applications and interoperate with the underlying kernel and OS services to provide a transparent programming environment. Overall, this layer aims at software portability.

In this work, we are specially interested in distributed OSes designed for lightweight manycores such as MOSSCA [12], M^3 [13] and Nanvix [14]. In these OSes, a microkernel-based design is employed. The rationale for this lies on the observation that on-chip resources in a lightweight manycore are scarce, and thus a full-weight kernel implementation (e.g., monolithic design) is either not possible or would yield to low resource availability to user-level applications. In contrast, the microkernel approach enables the implementation of system services in user-level, which may be distributed across the processor cores.

It is important to note that the other aforementioned distributed OSes (i.e., Barrelfish, FOS and HeliOS) are out of the scope of this work. These systems do not target lightweight manycores, so they do not address the architectural constraints of these processors discussed in Section 2.1. Since these OSes were not designed to cope with the limited amount of local memory of lightweight manycores, it is not possible to actually deploy them on lightweight manycores without a complete redesign and significant source code changes. Furthermore, architectures targeted by these OSes do not feature a rich on-chip interconnect (NoC). Therefore, communication challenges handled by the IKC facility proposed in this paper are not present, such as routing decisions, network congestion and control flow.

3. Kernel-Level Communication Facility

In this work, we propose an Inter-Kernel Communication (IKC) facility to better exploit the NoC capabilities of lightweight manycores and to address the communication characteristics between different subsystems of distributed OSes for these processors. First, we discuss the design goals of our IKC facility. Then, we present its three main abstractions: *syncs*, *mailboxes* and *portals*.

3.1. Design Goals

At hardware-level, lightweight manycores leverage a fast and reliable NoC to deal with asynchronous data transfers and to enable routing optimizations and quality of service. At system-level, on the other hand, subsystems of distributed OSes require concurrency and security in their communication, as well as they call out for primitives that efficiently support multiple communication granularities (i.e., fine- or coarse-grain) and purposes (i.e., data transfer or synchronization). Our IKC facility aims at the following design goals to cope with these three perspectives:

- *Flexibility*: different communication patterns should be supported;
- *Efficiency*: fine- and coarse-grain data transfers should be efficiently supported; and
- *Composability*: the communication facility should expose abstractions that may serve as building blocks for more complex protocols.

Furthermore, we guided our design and implementation to feature: (i) *a uniform addressing scheme*, so peers may rely on a logical addressing scheme and communication is processor-independent; (ii) *a transparent setup and management of the hardware*, so communications between subsystems become less complex; and (iii) *a low memory footprint*, since this resource is scarce in lightweight manycores. By accomplishing all these goals and features, we believe that our IKC facility makes a step further in the state of the art in kernel-level communication for distributed OSes that target lightweight manycores. Table 1

Abstraction	**Pattern**	**Designed For**
sync	`N:1`, `1:N`	Synchronization
mailbox	`N:1`	Small Message Exchange, Low-Latency
portal	`1:1`	Dense Data Transfers, High-Bandwidth

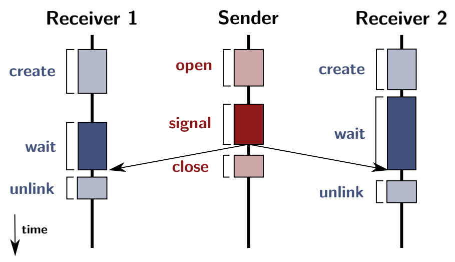
Table 1: Summary of communication abstractions proposed in this paper.

summarizes the three abstractions available in our IKC facility, and it gives insights on how they provide *flexibility*, *efficiency* and *composability*. A detailed discussion on this is presented later, in Section 3.5.

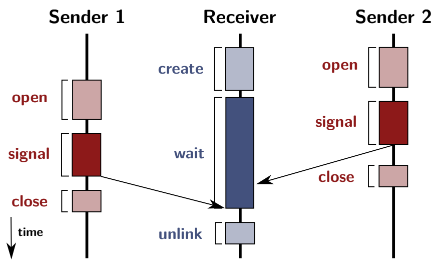
3.2. Syncs

The *sync* abstraction (shorthand for synchronization point) provides the basis for peer synchronization. It works by having on one side multiple peers (i.e., receivers) to block and wait for peers on the other side (i.e., senders) to issue a notification. The notification itself does not carry any information other than the required to wake up the receivers, thus this abstraction works with fine-grain data. At system-level, *syncs* are used at system startup to synchronize subsystems and when a distributed application is launched. Furthermore, *syncs* may be used to build more complex synchronization structures such as distributed mutexes, semaphores and barriers. The rationale for providing this abstraction in our facility is twofold. First, small amounts of data should be transferred around (i.e., tens of bytes) to synchronize peers. Thus, using a coarser-grain abstraction such as *mailboxes* to this purpose would be inefficient. Second, the on-chip interconnect of some lightweight manycores include special hardware to enable low-latency inter-cluster synchronization [9, 6]. If the synchronization semantic is explicit, our facility may be implemented so as to better exploit the capabilities of the underlying hardware.

Figure 3 outlines the semantics of the *sync* abstraction. We propose two operating modes for it: `1:N` and `N:1`, which in turn define senders and receivers. In `1:N` mode (Figure 3a), there is a single sender that issues wake up notifications to multiple receivers waiting for them. Conversely, in `N:1` mode (Figure 3b),



(a) 1:N mode.



(b) N:1 mode.

Figure 3: Execution flow of *sync* abstraction.

multiple senders issue notification signals to a single receiver. The set of operations available for each side of the communication is different. On the one hand, senders are allowed to `open`, `signal` and `close` a *sync*. On the other hand, receivers can `create`, `wait` and `unlink` a *sync*.

3.3. Mailboxes

The *mailbox* abstraction enables peers to exchange fixed-size messages with each other. The size of a message is designed to be small (i.e., hundreds of bytes), so that communication latency is reduced. This abstraction features an `N:1` semantic and works as follows. On one endpoint, a receiver owns a *mailbox* from which it reads messages. On the other endpoint, multiple senders may write messages to this *mailbox*. At system-level, *mailboxes* are used for exchanging control messages, which either encode simple operations or encapsulate meta-information of more complex tasks. For instance, the memory management subsystem may use a single *mailbox* message to request a remote peer to invalidate its page cache. On the other hand, one peer of the file system manager may rely on a message to pack information concerning a file read/write operation (i.e., name of the file, offset and read/write size). The dense data transfer is then carried out with a *portal* abstraction (see Section 3.4). Overall, we included this fixed-size *mailbox* abstraction in our facility to decouple small message exchanges from dense data transfers. In this way, we enable low-latency communication among the peers of a distributed OS for lightweight manycores.

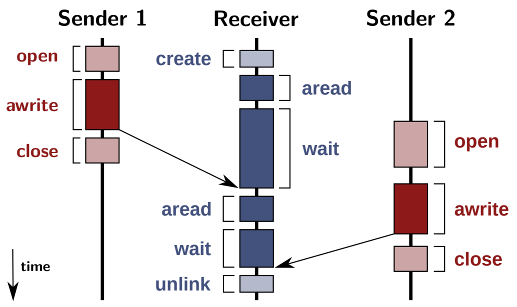


Figure 4: Execution flow of *mailbox* abstraction (`N:1`).

Figure 4 details the semantics of a *mailbox*. On the receiver side, four operations are available: (i) `create`, for creating a *mailbox*; (ii) `aread`, for asynchronously reading incoming messages; (iii) `wait`, for waiting for any income message; and (iv) `unlink`, for destroying a *mailbox*. Conversely, the sender features three operations: (i) `open`, for establishing a connection with a remote *mailbox*; (ii) `awrite`, for asynchronously posting a message to a remote *mailbox*; and (iii) `close`, for terminating the connection with a remote *mailbox*. Noteworthy, the *mailbox* abstraction features an asynchronous read/write semantic. Nevertheless, synchronous reads may be achieved by issuing an `aread` followed by a `wait` operation. On the other hand, synchronous writes may be accomplished by having each side owning a *mailbox* and a send-acknowledge protocol implemented.

3.4. Portals

The *portal* abstraction allows two peers to exchange arbitrarily large amounts of data (i.e., thousands of bytes) with each other, with built-in support for receiver-side control flow. This abstraction presents an `1:1` semantic and works as follows. On one endpoint, a receiver owns a *portal*, from which it reads incoming data. On the other endpoint, a sender may write data to this *portal*, once a connection with the remote receiver is established. This connection is explicitly established by the receiver itself by allowing a write on its *portal* from the sender. At system-level, *portals* may be used in a wide range of scenar-

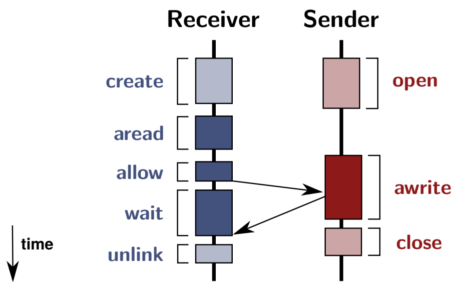


Figure 5: Execution flow of *portal* abstraction (`1:1`).

ios, specially when dense data transfers are needed. For instance, the process management may rely on this abstraction to deploy a binary file in a remote cluster or the file system manager may use *portals* for transferring file system blocks around. We included this abstraction in our communication facility to semantically support efficient dense data transfers. In this way, if the NoC of the underlying lightweight manycore features special hardware to take care of this, the overall system performance may be improved.

Figure 5 outlines the semantics of a *portal*. On the receiver side, five operations are available: (i) `create`, for creating a *portal*; (ii) `allow`, for authorizing a remote sender to write to the *portal*; (iii) `aread`, for asynchronously reading data; (iv) `wait`, for waiting incoming data; and (v) `unlink`, for destroying a *portal*. On the other hand, the sender features three operations: (i) `open`, for establishing a connection with a remote *portal*; (ii) `awrite`, for asynchronously writing data to a remote *portal*; and (iii) `close`, for terminating the connection with a remote *portal*.

3.5. Discussion

In this section we discuss how our IKC facility addresses the design goals outlined in Section 3.1. First, different communication patterns are supported (i.e., *flexibility*) by construction. The *sync* abstraction features `N:1` and `1:N` built-in operating modes; whereas *mailbox* and *portal* portal support `N:1` and `1:1` communication patterns, respectively. Second, efficient fine- and coarse-grain data transfers are supported (i.e., *efficiency*) because we explicitly provide abstractions for targeting different communication granularities. On the one hand, the *mailbox* abstraction is designed for exchanging small and fixed-size messages with low-latency. On the other hand, the *portal* abstraction enables dense data transfers with high bandwidth. Finally, our abstractions may be composed to construct more complex communication patterns and/or distributed OS protocols (i.e., *composability*). For instance, we can easily provide an `N:N` variant of the *mailbox* abstraction by creating a single built-in *mailbox* in each peer. Also, we can provide an `N:N` variant of the *portal* abstraction by opening `N` *portals* to each peer. Specifically concerning *composability* for OS protocols, the built-in *sync* abstraction may be used to build a master-slave barrier primitive [22], which in turn is required to synchronize peers at system startup as well as in application deployment. The memory manager may rely on a built-in *mailbox* to request a remote process to invalidate an entry of its page cache. Finally, a client of the file system service may use a built-in *portal* to write back dirty buffers in the underlying disk block device.

4. Implementation

In this section, we present the implementation details of our IKC facility in Nanvix: a POSIX-compliant open-source research OS[2] that targets lightweight manycores [14]. Nanvix currently supports multiple baremetal architectures, including Kalray MPPA-256 [6] and OpTiMSoC [23]. In addition, Nanvix features a built-in lightweight manycore simulator that enables OS development and debugging on top of Linux. Thus, other research groups may prototype and experiment new ideas in Nanvix even if they do not have access to a baremetal lightweight manycore.

To the best of our knowledge, Nanvix is currently the only open-source distributed OS that runs on commercially available baremetal lightweight manycores. In contrast, MOSSCA and M^3 run on an in-house simulator and a pro-

[2]Available at: `https://github.com/nanvix`.

cessor prototype implemented in a Field Programmable Gate Array (FPGA), respectively. Due to this reason, we chose to implement our facility in Nanvix rather than in the other OSes. Nevertheless, it is worth noting that the proposed solution may be also implemented in MOSSCA and M^3.

Our IKC facility was implemented in two layers of Nanvix: HAL [24] and microkernel [14]. Overall, we designed our solution using this two-tier approach in order to fulfill the following requirements:

- *Multiplexing*: NoC links should be used by several peers at the same time;
- *Virtualization*: virtual communication channels should be exposed, so that the number of communicating peers is not limited by the hardware; and
- *Security*: third party processes should not overthrow, disable or sniff the communication of peers.

In the HAL, we narrowed our implementation for the Kalray MPPA-256 processor and added a module for managing NoC routers. This module is responsible for setting up and configuring underlying hardware resources, such as TX/RX buffers and routing tables, as well as to catch and handle interruptions. Overall, this communication module exposes a uniform interface that features: (i) a logic cluster numbering scheme; (ii) primitives for sending/receiving synchronization signals; (iii) primitives for sending/receiving fine-grain fixed-size messages; and (iv) primitives for sending/receiving coarse-grain fixed-size data blocks. We intentionally provide three primitives to match the semantics of the overlying abstractions (i.e., *sync*, *mailbox* and *portal*) while exploiting the best possible the underlying hardware. The details of these primitives are provided below:

- Primitives for synchronization signals are entirely provided on top of a dedicated and low-latency Control NoC (C-NoC);
- Primitives for fine-grain messages use the general purpose Data NoC (D-NoC) to exchange data and the C-NoC for communication setup. They

operate with fixed-size buffers that are optimized for low-latency communication; and

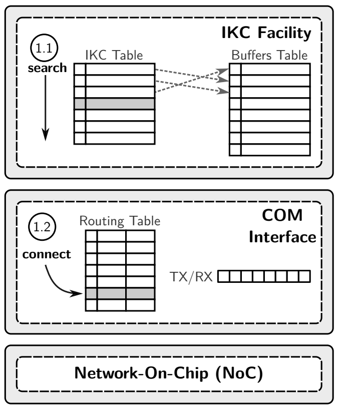
- Primitives for coarse-grain data transfers use the D-NoC to exchange data and the C-NoC for communication setup. They operate with fixed-size buffers that are tuned for high-bandwidth communication.

Noteworthy, both primitives that operate with fine- and coarse-grain data (i.e., *mailboxes* and *portals*) are backed up by fixed-size buffers. The reason for this stands for the way in which the Kalray MPPA-256 processor setups communication through the NoC. It requires both ends to know the size of a transfer beforehand.

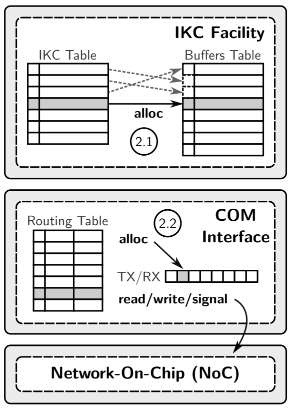
In the microkernel, we introduced an IKC facility that implements the *sync*, *mailbox* and *portal* abstractions (Section 3). In this level, we aimed at providing the communication semantics of each abstraction as well as exposing a multiplexed and virtualized interface of them. These two characteristics are built on a buffering scheme combined with a port based addressing strategy. At creation, each instance of an abstraction reserves a port in the related physical connection. Then, each virtual instance is referenced uniquely by its connection identifier combined with its port number, representing an address for the communication facility.

Figure 6 pictures a generic overview of our implementation. Overall, the kernel has two fixed-size tables for each abstraction: one for keeping track of the active connections (i.e., IKC table); and another for buffering either signals, messages or data blocks that are yet to be sent/received (i.e., buffer table). Two major operations take place whenever a communication abstraction is opened/created (Figure 6a). First, an entry is allocated in the proper table of active connections and is initialized with meta-information concerning the connection to be established (step 1.1), such as involved peers. Then, the HAL interface is invoked to setup and establish the connection itself (step 1.2).

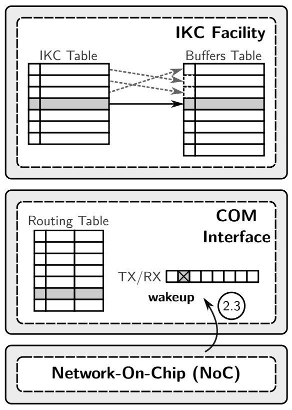
As soon as a communication operation is issued (i.e., `signal`, `wait`, `allow`, `read` or `write`), the table of active connections is traversed to determined if a



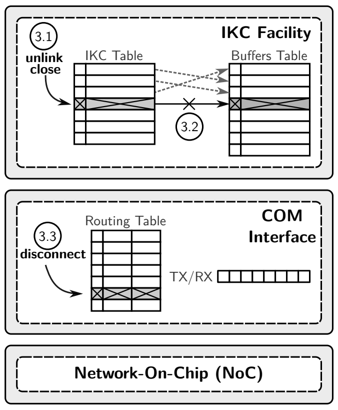
(a) Create/open.



(b) Read/write.



(c) Read/write completion.



(d) Close/unlink.

Figure 6: Execution flow internals in the proposed IKC facility.

connection was previously established (Figure 6b). If so, the operation continues as follows. For `signal`, `read` and `write` operations, an entry in the table of buffers is booked and filled in with the information concerning the operation itself (step 2.1). Next, the request is scheduled in a queue (step 2.2) and eventually is dispatched by the HAL (TX/RX registers). For the `wait` operation, however, the table of buffers is searched to determine whether or not the target on-going operation is completed. If so, data is copied back to the requesting peer and the respective entry in the table of buffers is released. Otherwise (6c), the calling thread of the requesting peer is blocked until the target operation is completed. At this time, the thread is then awaken by the interrupt handler of the NoC (step 2.3). Finally (Figure 6d), once communication is over, involved peers call either `close` or `unlink` (step 3.1) to hang up. In either way, the respective entry in the table of active connections is released (step 3.2) and the HAL interface is invoked to terminate the underlying connection (step 3.3).

5. Evaluation Methodology

In this section, we present our evaluation methodology, which was conceived so as to answer the following main questions:

- What is the upper-bound performance and scalability of our IKC facility?
- What is the performance delivered by our IKC facility in realistic OS use case scenarios?

This discussion is organized as follows. First, we present the experimental programs conceived to analyze the performance and scalability of our IKC facility. Next, we detail the lightweight manycore processor considered in this paper. Finally, we discuss our experimental design.

5.1. Experimental Programs

We considered two sets of experimental programs to evaluate our IKC facility. The first set consists in a suite of micro-benchmarks that make raw use of the communication abstractions of our IKC facility (i.e., *sync*, *mailbox*, and *portal*). The second set is a collection of programs that exercise different communication protocols in Nanvix, and thus illustrate the use of our IKC facility in use case scenarios. Table 2 summarizes the most important characteristics of each experimental program. All these programs are publicly available[3]. The micro-benchmark suite is composed of the following programs:

[3]Available at: `https://github.com/nanvix/benchmarks`.

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

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

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

The benchmark of communication protocols is composed of the following programs:

`spawn` It benchmarks the latency for spawning system services. It spawns N system services and waits them to boot up with a master-slave barrier primitive [22] that was implemented on top of our *sync* abstraction.

`lookup` It benchmarks the latency for resolving the physical location of a process (i.e., the cluster of lightweight manycore processor in which the process is running). It launches a process that dispatches several name lookup requests to the name server. This program relies on a `1:1` ping-pong communication protocol using a *mailbox*.

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

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

Type	Subsystem	Program	Abstraction(s)	Pattern(s)	Transfer Size
Micro-benchmark	Communication	`fence`	*sync*	`1:1`, `1:N`, `N:1`	Tens of Bytes
		`mail`	*mailbox*	`1:1`, `1:N`, `N:1`	Hundreds of Bytes
		`cargo`	*portal*	`1:1`, `1:N`, `N:1`	Tens of Kilobytes
Protocol benchmark	Process management	`spawn`	*sync*	`N:N`	Tens of Bytes
		`lookup`	*mailbox*	`1:1`	Hundreds of Bytes
		`heartbeat`	*mailbox*	`N:1`	Hundreds of Bytes
	Memory management	`pginval`	*mailbox*	`1:N`	Hundreds of Bytes
		`pgfetch`	*mailbox* + *portal*	`1:1`	Tens of Kilobytes

Table 2: Benchmarks to evaluate the proposed communication facility.

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

5.2. Experimental Platform

Among the architectures supported by Nanvix, we chose Kalray MPPA-256 as the experimental platform in this work. This is a single-chip commercial lightweight manycore processor that features most of the characteristics discussed in Section 2. Kalray MPPA-256 features 272 general-purpose cores and 16 firmware-cores, called Processing Elements (PEs) and Resource Managers (RMs), respectively. The processor is built with 28 nm CMOS technology and it runs at 400 MHz. All cores implement a 64-bit proprietary instruction set, present a 5-issue Very Long Instruction Word (VLIW) pipeline, 8 Kilobytes (kB) instruction and data caches, and feature a software-managed Memory Management Unit (MMU).

The 288 cores of Kalray MPPA-256 are grouped into 16 Compute Clusters, which are intended for computation, and 4 I/O Clusters, which are designed to provide connectivity to peripherals. Each Compute Cluster bundles 16 PEs, one

RM, two NoC interfaces and a 2 MB of local SRAM. In these clusters, hardware cache coherence is not supported. In contrast, I/O Clusters have 4 RMs, 8 NoC interfaces and 4 MB of SRAM. Two of these clusters are connected to a different DDR controller, and the other two are attached to PCI and Ethernet controllers. Compute Clusters are not attached to a global memory and they all have private address spaces. Thus, Compute Clusters have to exchange hardware messages by one of two different interleaved 2-D torus NoCs to carry out communications: (i) a C-NoC that features low bandwidth and is intended for small data transfers; and (ii) a D-NoC that presents high bandwidth and is dedicated to dense data transfers.

What concerns software development in Kalray MPPA-256, this processor is shipped with a patched version of GCC 4.9.4 and Binutils 2.11.0. No OS is provided by the vendor, and software engineers should rely on a proprietary and non-conformant runtime environment to write their applications [25], if not using Nanvix. Furthermore, regarding OS kernel implementation, system engineers are required to rely on a proprietary hypervisor from Kalray. This hypervisor runs on the firmware cores of the processor and intermediates all low-level operations. Noteworthy, Kalray Hypervisor cannot be changed nor configured, and thus it imposes additional challenges in OS kernel construction for this lightweight manycore processor.

5.3. Experimental Design

We conducted three sets of experiments to assess our IKC facility and answer the questions stated earlier in this section. First, we employed micro-benchmarks to evaluate the upper-bound performance of our communication facility. To this end, we considered standalone communications using each one of the abstractions, while fixing the number of peers involved in communications and varying the transfer sizes. Overall, we varied the size of transfers from 64 Bytes (B) to 1024 B when using *mailboxes* and from 64 B to 16384 B when using *portals*. Since *sync* does not carry any information other than a wake up signal, we have not evaluated the impact of communication granularity in this abstraction. Second, we aimed at studying the upper-bound scalability of our abstractions. We did so by also launching the micro-benchmark suite in standalone mode, but we fixed the communication granularity of each abstraction and varied the number of communicating peers. Noteworthy, we set the communication granularity to the best case scenario that we identified in the first set of experiments. Finally, we relied on the suite of OS service protocols to benchmark the overall performance of our abstractions in real-world use case scenarios. We also carried out all the aforementioned experiments using a *mailbox*-based primitive. We consider this to be the baseline for our results, once related works rely only in this abstraction to carry out all kinds of on-chip communication.

In all experiments, we gathered execution statistics concerning communication time and amount of data transferred. All time measurements were performed using hardware performance counters to enable monitoring with minimum interference. On the other hand, to retrieve energy consumption statistics, we relied on a device that is externally attached to the board of the processor. This device measures power dissipation on the board and comprises statistics for all cores, NoCs and other on-chip resources. With this tool we proceeded as follows:

1. We launched a specific experiment to understand the power dissipation during Nanvix's startup. Overall, we concluded that while clusters are being turned on at boot time, power dissipation increases linearly. Then, it remains steady at around 6.7 W while Nanvix services are running.

2. We re-executed all the protocol benchmarks (`heartbeat`, `pginval`, `lookup`, `spawn` and `pgfetch`) and collected the power dissipation statistics using the external device.

3. Based on the Nanvix's startup time described before, we sliced the power dissipation measurements in each experiment so as to obtain only the power dissipation when the benchmark was running.

4. We computed the energy consumption for each experiment by integrating the sliced power dissipation measurement.

Noteworthy, we had to apply the above methodology because the Kalray MPPA-256 processor lacks more precise utilities for measuring power dissipation during a specific portion of execution. Therefore, the presented results unveil an upper-bound energy consumption of our solution. Overall, we carried out 10 trials of each experimental configuration to eliminate undesired warm-up effects, and then we executed 30 trials to collect results. All comparisons of these metrics are based on a confidence interval threshold of 95% (significance of 5%).

6. Experimental Results

In this section, we present and discuss our experimental results. First, we analyze the upper-bound performance of our IKC facility. Then, we examine the scalability of our abstractions. Finally, we discuss the performance of our IKC facility in realistic use case scenarios. All results are compared with a solution based on fixed-size *mailboxes*, since this is the communication abstraction adopted by related works. Noteworthy, we set the size of *mailbox* messages to 128 B, which we found to be the optimum for Kalray MPPA-256. A more detailed discussion on this conclusion is presented in Section 6.1.

6.1. Raw Performance Analysis

Figure 7 presents the latency for reading/writing to a *mailbox* obtained with the `mail` micro-benchmark. Overall, we observed two behaviors: (i) latency proportionally increases with the size of the message payload, for both operations; and (ii) latency grows slower for reads than for writes due to explicit copying to underlying TX buffers. These results uncover an important design aspect: the message size should be kept small to minimize the communication latency of both operations. For 64 B payloads, this yields to 26 µs reads and 27 µs writes. In contrast, for 128 B payloads, hits yields to 29 µs reads and 31 µs writes.

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

Figure 8 pictures the throughput for reading n bytes when carrying out this operation using a fixed-size *mailbox* of 128 B (baseline) and when using our *portal* abstraction with buffers of the same size of the data transfer. These results were obtained with the `cargo` micro-benchmark and show up the peak bandwidth of our *portal* abstraction, and were used to dimension the size of buffers for this abstraction.

Overall, we observed that the throughput delivered by *mailbox* is constant, whereas the throughput achieved by *portal* increases with the transfer size, to some extent. A three-phase behavior can be noticed in the plot: (i) for transfer

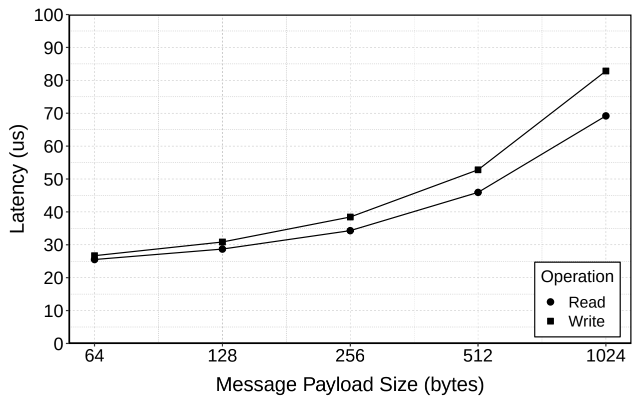


Figure 7: Mailbox latency when varying message size (`mail` micro-benchmark).

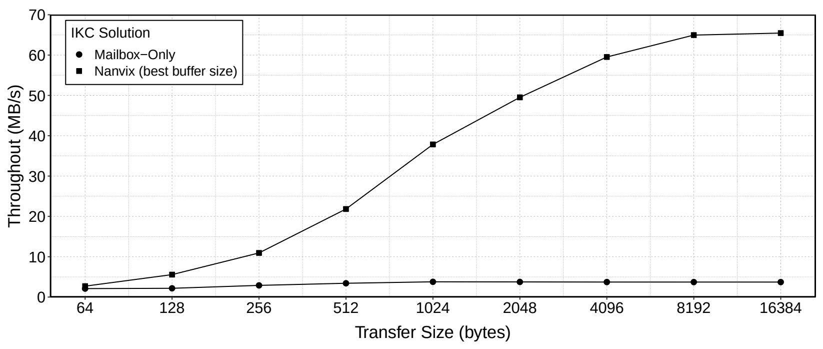


Figure 8: Mailbox and portal throughput when varying transfer size (`cargo` micro-benchmark).

sizes ranging from 64 B to 1024 B, a linear increase in throughput is observed; (ii) for transfer sizes ranging from 1024 B to 8192 B, a sub-linear increase in throughput is observed; and (iii) for transfer sizes bigger than 8192 B, a constant throughput is achieved. The rationale for this lies on the fact that transfer sizes of up to 1024 B are efficiently handled by the NoC, which becomes saturated beyond that point.

We relied on these conclusions to dimension the default buffer size configuration of our portal abstraction in Kalray MPPA-256. Recall that this abstraction is meant to enable efficient coarse-grain communication between the different subsystems of the OS. In this context, we highlight two typical use cases: 4 kB page transfers involving the memory management subsystem; and 1 kB file block transfers involving the file management subsystem. Therefore, we set the default size of portal buffers to 4 kB to achieve a better performance for the former subsystem. From now onward, all results with *portal* will be based on a 4 kB configuration, unless otherwise stated.

6.2. Raw Scalability Analysis

We now analyze the scalability of *mailbox*, *sync* and *portal* abstractions.

Fine-grain Data Transfers with Mailboxes. Figure 9 presents the throughput for reading and writing fixed-size messages from/to a *mailbox*, when increasing the number of communicating processes. These results were obtained with the `mail` micro-benchmark. We noted that both operations deliver linear throughput scalability, with a capacity for reading and writing about 3.2 k and 3.8 k messages per second, respectively. These results show the inherent scalability of rich NoCs, in contrast to traditional interconnects such as buses and crossbar switches. Notwithstanding, we found that the throughput gap between the two operations is justified by technical limitations of the hypervisor that runs on the Kalray MPPA-256 processor. It is not an open-source software, and it is not shipped with enough documentation/information to enable us a full utilization of DMA engines. As a consequence, our implementation relied on polling to provide this operation, resulting in a slightly worse performance. This observation uncovers an important point for improvement in the Kalray MPPA-256 hypervisor.

Process Synchronization with Synchronization Points. Figure 10 shows the latency for sending and receiving synchronization signals between multiple processes using either *mailbox* (baseline) or *sync*. These results were obtained with the `fence` micro-benchmark. On the one hand, we observed an important difference between *mailbox* and *sync*: our *sync* abstraction handles these signals 64× faster than the baseline solution. The rationale for this is three-fold: (i) synchronization signals require a few bytes of data to be transferred around; (ii) *mailboxes* work with a coarser data-transfer granularity (128 B) in contrast

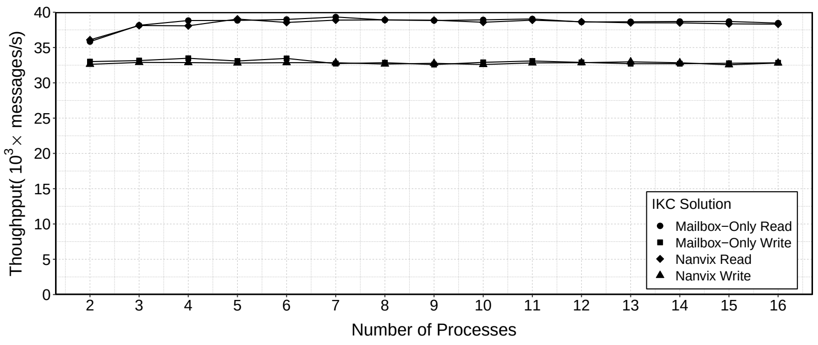


Figure 9: Mailbox throughput for fixed-size messages (`mail` micro-benchmark).

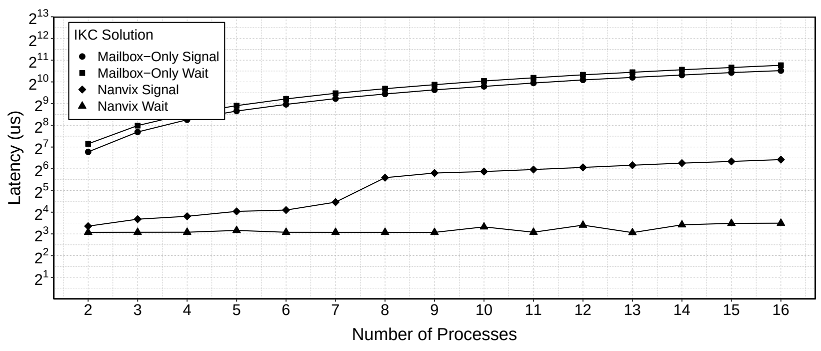


Figure 10: Sync latency scalability for synchronization signals (`fence` micro-benchmark).

to *sync* (4 B); and (iii) our abstraction relies on the C-NoC to transfer data, which delivers lower latencies than the D-NoC, which is narrowed for bandwidth and is used by *mailboxes*. On the other hand, when we contrasted the latency of the two operations when increasing the number of processes involved, we observed that the *wait* latency stays constant at 8 µs, whereas the *signal* latency increases from 10 µs to 64 µs. The rationale for this comes from the fact that *syncs* are backed up by the C-NoC of Kalray MPPA-256 and this NoC can receive multiple signals in parallel through the same interface, but it cannot do the same when sending signals. Finally, concerning the step behavior for `signal` operation from seven to eight processes, we found the rationale in the NoC topology. If we change the deployment layout of processes in this benchmark, this step behavior also shows up, but not necessarily when moving from seven to eight processes.

Dense Data Transfers with Portals. Figure 11 presents the read and write bandwidth when using either *mailbox* (baseline) or *portal*, while varying the number of communicating processes. These results were obtained with the `cargo` benchmark. In general, we observed that both abstractions provide linear bandwidth scalability for both operations, with reads being more efficient than writes, according to this metric. This observation is aligned with the previous conclusions on latency scalability for passing around fixed-sized messages with *mailboxes*.

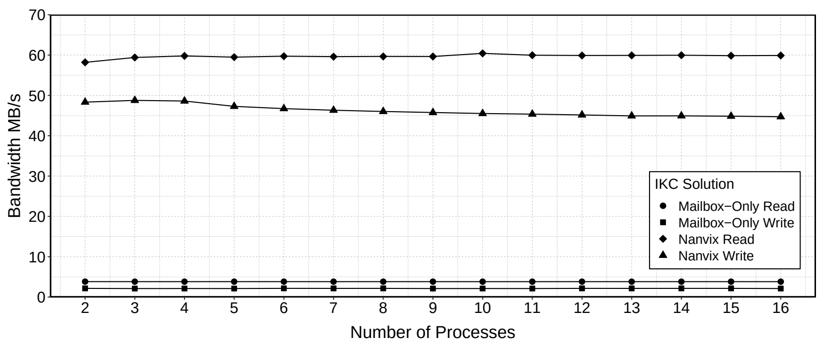


Figure 11: Portal bandwidth scalability for dense data transfers (`cargo` micro-benchmark).

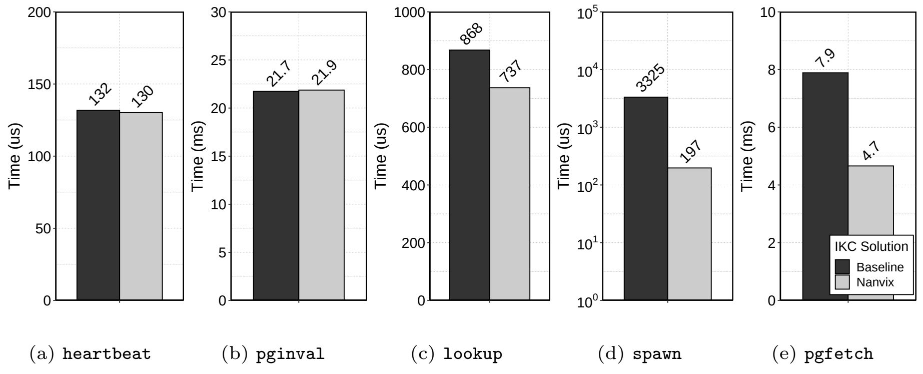
Again, the hypervisor of the Kalray MPPA-256 processor is not an open-source software, and it is not shipped with enough documentation/information to enable us a full utilization of DMA engines. Therefore, our *portal* implementation rely on polling to provide this operation. When contrasting the throughput performance of the two abstractions for working with dense data transfers, however, we spotted an important difference between them: *portal* achieved roughly 60 MB/s and 40 MB/S for read and write operations, respectively, as opposed to 8 MB/s and 5 MB/s achieved by *mailbox*. The rationale for this lies on the fact that *portals* are backed up by larger transfer buffers than *mailboxes*. Thanks to this, in dense data transfers, fewer chunks are needed to carry out the transfer, and thus the overall communication overhead that is required for flow control is smoothed out. As a consequence, higher bandwidth is achieved.

6.3. Realistic Use Case Experiments

So far, we analyzed the raw performance and scalability of our IKC facility. In this section, we turn our focus to a more realistic assessment that exercises important services and protocols in Nanvix. In these experiments, however, the communication performance is subjected to interference of other OS components of the system that cannot be completely isolated, such as concurrency for resources. Therefore, conclusions drew previously do not necessarily hold under these circumstances. Again, our baseline is based solely on the *mailbox* abstraction to implement synchronization signals as well as fine- and coarse-grain data transfers.

Heartbeat. Figure 12a presents the time spent in communication in the `heartbeat` benchmark. In this program, a `N:1` and fine-grain communication pattern is exercised. We observed that both abstractions yield to about 130 µs communication times, thus showing that a richer IKC facility does not negatively impact the performance. This conclusion is further reinforced by energy consumption results (Figure 13a).

Page Cache Invalidation. Figure 12b presents the time spent in communication in the `pginval` benchmark. In this program, a `1:N` and fine-grain communication pattern is benchmarked. As in `heartbeat`, we observed that both abstractions deliver similar performance for this communication pattern, thereby strengthening our previous conclusion. However, the communication time in this protocol is about 10 orders of magnitude higher than in the other evaluated protocols. Notwithstanding, this observation shows that communication may have significant impacts on the quantitative performance of the OS. Finally, energy consumption results show that there is no significant different between the two solutions (Figure 13b).



(a) `heartbeat` (b) `pginval` (c) `lookup` (d) `spawn` (e) `pgfetch`

Figure 12: Communication time spent in several OS services of Nanvix.

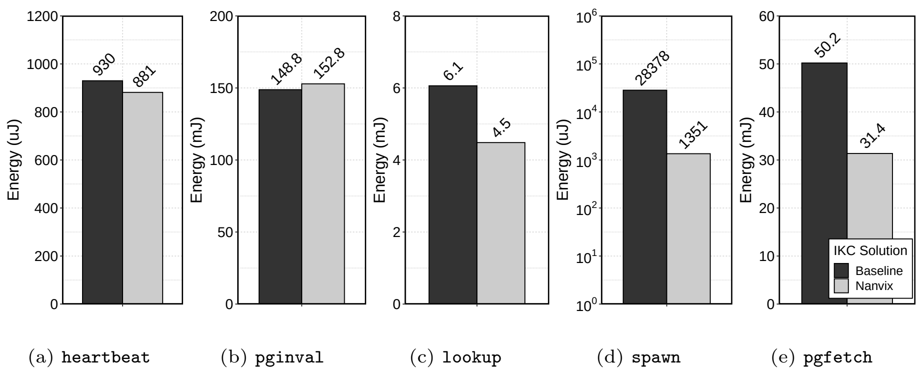


Figure 13: Energy spent during communication in several OS services of Nanvix.

Name Lookup. Figure 12c presents the communication time in the `lookup` benchmark. In this program, `1:1` fine-grain communication pattern is exercised. In contrast to the other two benchmarks that also rely on a *mailbox* (i.e., `heartbeat` and `pginval`), in this experiment we noted a performance difference from our IKC solution and the baseline. Although it may be surprisingly at first, the rationale for this is two-fold: (i) in this experiment there exists a strong competition for the communication infrastructure, due to the concurrent services that execute along with the name lookup; and (ii) our IKC facility enables better concurrency support due to a more appropriate hardware resources usage. Recall that in the baseline solution, all types of communications are multiplexed on top of *mailboxes*. As a consequence, not all TX/RX buffers that are available in the hardware are indeed used and thus communication contention happens if too many processes and/or OS services concurrently use the communication infrastructure. On the other hand, our IKC facility not only makes plain use of these buffers but also relies on both NoCs that are available to carry out communication. In the end, due to this better concurrency support, communication performance of this service is improved by 24.91%, while energy consumption drops by 26.22% (Figure 13c).

Spawn. Figure 12d presents the communication time in the `spawn` benchmark, in which a `N:N` fine-grain communication pattern is exercised. Overall, we observed an important performance gap between the baseline and our IKC facility, roughly by a factor of $16\times$ in time and $21\times$ in energy consumption (Figure 13d). Indeed, this OS service is backed up by synchronization signals and this observation is aligned to our previous performance scalability analysis of our *sync* abstraction. Thus, this result sustains another important hypothesis of our research:

> *If we provide an additional communication abstraction that exposes synchronization primitives to other OS subsystems, we may exploit hardware support for these operations, and thus improve communication performance.*

Page Fetch. Figure 12e presents the communication time in the `pgfetch` benchmark, in which a `1:1` dense communication pattern is assessed. Overall, our IKC facility achieved about $1.65\times$ shorter times and $1.59\times$ smaller energy consumption (Figure 13d) than the baseline. While the baseline solution relies only on *mailboxes* to carryout the same transfer, our IKC facility is backed up by a hybrid approach that uses *mailbox* and *portal*. The former is used to exchange operation headers in the protocol using hardware RX/TX registers, whereas the latter is used to carry out the page transfer itself from the memory manager to the faulting process by programming the DMA engine. Thus, this result sustains another important hypothesis of our research:

> *If we expose an additional communication abstraction that provides enhancement support to dense data transfers, we may significantly improve the performance of OS subsystems that feature such requirement.*

7. Related Work

In this section, we present existing communication mechanisms that are available for manycore platforms, contrasting them with our IKC facility. First, we discuss about application-level communication libraries for manycores. Then, we consider communication subsystems for OSes that target large SMP platforms. Finally, we turn our focus to kernel-level communication abstractions that are provided by OSes of lightweight manycores.

When comes to communication in manycore processors, several works focus on providing application-level solutions. On the one hand, there are vendor-specific baremetal libraries that rely on particular features of the underlying hardware to achieve high performance. For instance, Wijngaart [26] and Clauss [27] provide communication interfaces for the Intel Single-Cloud Computer processor. Similarly, Kalray MPPA-256 features a communication library that shares some similarity with POSIX [25] and a specific interface for one-sided communications [18]. Finally, a particular communication API is provided to developers that target the Adapteva Epiphany architecture [28]. On the other hand, there exist solutions based on well-known distributed programming interfaces, such as the Unified Parallel C (UPC) port for the Intel Single-Cloud Computer [29] and Tilera TILE64 [30] processors, the OpenSHMEM implementation for the Adapteva Epiphany processor [31], and an Message Passing Interface (MPI) port for Kalray MPPA-256 [32] and Adapteva Epiphany [33]. In contrast to both application-level approaches, our IKC facility focus on providing kernel-level abstractions for communication. This way, we are able to provide communication multiplexing among several applications in a secure and fair way. We see application-level communication libraries as a complement of our work.

Distributed OSes that target large SMP architectures, such as Barrelfish [11], FOS [10] and HeliOS [21], ultimately rely on shared memory to enable communication of subsystems. To this end, OS kernels book some range of the underlying shared memory to setup a communication infrastructure. For instance, Barrelfish provides a channel abstraction on top of a shared memory region that enables point-to-point cache-line-sized messaging between a single writer and reader cores (i.e., `1:1` communication). In contrast, FOS uses shared memory to expose a mailbox abstraction that supports transferring of larger fixed-size messages between multiple writers and one reader core (i.e., `N:1`). Finally, HeliOS provides over shared memory a First-in First-Out (FIFO) queue abstraction that enables arbitrarily large `1:1` message passing. Overall, while the communication infrastructure of these OSes provide efficient solutions to SMP manycore processors, they are not designed to cope with challenges and features of lightweight manycores. More precisely, these solutions do not take into account the distributed memory architecture as well as the capabilities of rich NoCs to deal with multiple communication granularities (i.e., small, medium and large transfers) and communication purposes (i.e., data transfers and synchronization). As a consequence, they would bring sub-optimal system performance to lightweight manycores.

On the other hand, distributed OSes for lightweight manycores are inherently designed to cope with a distributed memory architecture and rich NoCs. Thus, they feature communication abstractions that are closer to ours. For instance, MOSSCA [12] provides unidirectional channels (i.e., `1:1`) to enable communication between clusters. Channels have policies that guarantee properties and restrictions on the participating peers, such as flow control and fault tolerance. Furthermore, these channels may be used as building blocks to implement more sophisticated and robust communication patterns. Conversely, inter-cluster communication is enabled in M^3 [13] through either one of two abstractions: (i) message gates for exchanging fixed-size messages; or (ii) for transferring arbitrary large amounts of data through a ring-buffer in the DRAM. Overall, our work contrasts with these two others as follows. In respect to the communication module of MOSSCA, our IKC facility exposes abstractions for transferring fixed-size messages (i.e., *mailboxes*) and arbitrarily-large data (i.e., *portals*), thereby enabling the underlying rich NoC to be better exploited. In contrast to M^3, our communication facility additionally provides a synchronization abstraction, thus enabling synchronization and data transfer to be decoupled and hence communication performance improved. Furthermore, our *portal* abstraction (analogue to memory gates in M^3) enables direct communication between any pair of peers in the lightweight manycore, without the need of temporarily storing data in a ring buffer in the DRAM. Finally, our facility is designed to efficiently support different communication characteristics of distributed OS (i.e., synchronization, as well as fine- and coarse-grain transfers).

Porting our IKC facility to other OSes requires a large amount of work because each OS has its own particular aspects. We opted to implement it in Nanvix, since we aimed at evaluating its performance in a commercially available baremetal lightweight manycore, and, to the best of our knowledge, Nanvix is currently the only open-source distributed OS that runs on one of these processors. In contrast, MOSSCA and M^3 run on an in-house simulator and a processor prototype implemented in a FPGA, respectively.

8. Conclusions

Lightweight manycore processors achieve high performance and energy efficiency thanks to a selected set of architectural features, such as high count of low-power cores, distributed memory architecture and rich NoCs [6, 8, 9]. Conversely, to address such hardware design, OSes for this emerging class of processors embrace a distributed structure to achieve scalability while exposing richer abstractions and APIs to user-level software [10, 11]. In this approach, subsystems of the OS are factored in a set of services that: (i) are deployed across the cores of the processor; and (ii) collaboratively work with one another to implement system functionalities.

To effectively enable this distributed structure, the OS features a communication layer that provides communication primitives on top of the underlying NoC. To this end, current distributed OSes for lightweight manycores encapsulate these primitives in a *mailbox* structure: an abstraction that enables fixed-size messages to be transferred [12, 13].

Despite the fact that *mailboxes* enable a distributed OS design, we argue they are not enough for enabling efficient communication in these architectures. First, the architectural features of the NoC may not be fully exploited with a single abstraction. The rationale for this lies on the fact that right use of each of these hardware resources depends on the semantics of the communication and this information is unavailable at the communication layer. Second, the diversity in communication protocols across the different subsystems calls out for supporting multiple abstractions, otherwise the implementation of protocols is inefficient. For instance, some services require flow control while others do not. Therefore, if the same abstraction is used for implementing both protocols, overhead is imposed in the latter one even though it does not use this feature.

Therefore, to overcome this problem, in this work we proposed a richer IKC facility that exposes three communication abstractions to other subsystems of the OS: (i) *syncs*, for enabling a process to signal and unlock another process remotely; (ii) *mailboxes*, for sending fixed-size messages with low latency; and (iii) *portals*, for handling dense data transfers with high bandwidth. These abstractions enable the NoC capabilities to be better exploited, as well as the communication characteristics between different subsystems to be better addressed.

We implemented the proposed IKC facility in Nanvix, an open-source distributed OS that targets lightweight manycores [14]. Furthermore, we evaluated our solution using two sets of experimental benchmarks: (i) a micro-benchmark suite to study the raw performance and scalability of our abstractions; and (ii) a benchmark suite that exercises important communication protocols that are applied in different subsystems of Nanvix. We carried out baremetal experiments on Kalray MPPA-256, a NoC-based lightweight manycore processor that features a distributed memory architecture and integrates 288 cores in a single chip. Overall, our results showed that our communication facility enables up to 16.87× and 1.68× better performance than a *mailbox*-only solution, in synchronization and dense data transfers, respectively. Additionally, Concerning energy consumption our solution showed up to be up to 21× more efficient than the baseline.

This work is inserted into a larger scope, the joint research initiative between PUC Minas, UGA and UFSC that aims the design and implementation of a POSIX-compliant OS for lightweight manycore processors. In this context, Nanvix is an important research asset that effectively enables prototyping and benchmarking new ideas that concern OS construction for these emerging architectures. As future works, we intend to:

(i) optimize communication protocols of existing OS services to better exploit the semantics of the new abstractions that we introduced, and thus improve the overall performance of the system;

(ii) design and implement a POSIX Inter-Process Communication (IPC) service on top of the IKC facility to expose standard OS communicators, such as pipes, named semaphores, message queues and sockets;

(iii) work on a MPI port on top of our IKC facility, to enable applications that rely on this framework to be ported to Nanvix, and consequently to all lightweight manycores; and

(iv) evaluate our IKC facility running on a FPGA supported by Nanvix, to spot potential codesign aspects and improve even further communication between the subsystems of a distributed OS for lightweight manycores.

Finally, concerning the current implementation of our IKC facility on Kalray MPPA-256, we highlight that there is room for improvement as well. Due to the lack of appropriate documentation, our implementation does not make use of DMA engines that are available in the clusters of this processor, resulting in sub-optimal performance for dense data transfers. However, we emphasize that this limitation is not related to the abstractions proposed in this work, which were made to be platform-independent. If the required information on how to effectively use DMA engines is available, one could easily enhance the implementation by changing a small portion of its source code.

Acknowledgments

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.

References

[1] E. Francesquini, M. Castro, P. H. Penna, F. Dupros, H. Freitas, P. Navaux, 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 (JPDC) 76 (C) (2015) 32–48. doi:10.1016/j.jpdc.2014.11.002.
URL https://doi.org/10.1016/j.jpdc.2014.11.002

[2] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, B. Baas, KiloCore: A 32-nm 1000-Processor Computational Array , IEEE Journal of Solid-State Circuits (JSSC) 52 (4) (2017) 891–902. doi:10.1109/JSSC.2016.2638459.

[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, L. Benini, Energy-Efficient Near-Threshold Parallel Computing: The PULPv2 Cluster , IEEE Micro 37 (5) (2017) 20–31. doi:10.1109/MM.2017.3711645.
URL http://ieeexplore.ieee.org/document/8065010/

[4] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, D. Dutoit, Platform 2012, a many-core computing accelerator for embedded socs, in: Design Automation Conference, DAC '12, ACM Press, New York, USA, 2012, pp. 1137–1142. doi:10.1145/2228360.2228568.
URL http://dl.acm.org/citation.cfm?doid=2228360.2228568

[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, M. B. Taylor, The celerity open-source 511-core risc-v tiered accelerator fabric: Fast architectures and design methodologies for fast chips, IEEE Micro 38 (2) (2018) 30–41. doi:

10.1109/MM.2018.022071133.
URL https://ieeexplore.ieee.org/document/8344478/

[6] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, T. Strudel, A clustered manycore processor architecture for embedded and accelerated applications, in: IEEE High Performance Extreme Computing Conference, HPEC '13, IEEE, Waltham, USA, 2013, pp. 1–6. doi:10.1109/HPEC.2013.6670342.
URL http://ieeexplore.ieee.org/document/6670342/

[7] A. Olofsson, Epiphany-v: A 1024 processor 64-bit risc system-on-chip, ArXiv 1610.01832 (2016) 1–15.
URL https://arxiv.org/abs/1610.01832

[8] F. Zheng, H.-L. Li, H. Lv, F. Guo, X.-H. Xu, X.-H. Xie, Cooperative computing techniques for a deeply fused and heterogeneous many-core processor architecture, Journal of Computer Science and Technology 30 (1) (2015) 145–162. doi:10.1007/s11390-015-1510-9.
URL https://link.springer.com/article/10.1007/s11390-015-1510-9

[9] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, G. Yang, The sunway taihulight supercomputer: System and applications, Science China Information Sciences 59 (7) (2016) 072001–0720016. doi:10.1007/s11432-016-5588-7.
URL http://link.springer.com/10.1007/s11432-016-5588-7

[10] D. Wentzlaff, A. Agarwal, Factored operating systems (fos): The case for a scalable operating system for multicores, ACM SIGOPS Operating Systems Review 43 (2) (2009) 76–85. doi:10.1145/1531793.1531805.
URL http://portal.acm.org/citation.cfm?doid=1531793.1531805

[11] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, A. Singhania, The multikernel: A new os architecture for scalable multicore systems, in: ACM SIGOPS Symposium on Operating Systems Principles, SOSP '09, ACM, Big Sky, Montana, 2009, pp. 29–44. `doi:10.1145/1629575.1629579`.
URL `http://portal.acm.org/citation.cfm?doid=1629575.1629579`

[12] F. Kluge, M. Gerdes, T. Ungerer, An operating system for safety-critical applications on manycore processors, in: International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC '14, IEEE, Reno, Nevada, 2014, pp. 238–245. `doi:10.1109/ISORC.2014.30`.
URL `http://ieeexplore.ieee.org/document/6899155/`

[13] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, G. Fettweis, M3: A hardware/operating-system co-design to tame heterogeneous manycores, in: International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, ACM, Atlanta, Georgia, 2016, pp. 189–203. `doi:10.1145/2872362.2872371`.
URL `http://dl.acm.org/citation.cfm?doid=2954680.2872371`

[14] P. H. Penna, J. V. Souto, D. F. Lima, M. Castro, F. Broquedis, H. Freitas, J.-F. Méhaut, On the performance and isolation of asymmetric microkernel design for lightweight manycores, in: Brazilian Symposium on Computing Systems Engineering, SBESC '19, IEEE, Natal, Brazil, 2019, pp. 1–8. `doi:10.1109/SBESC49506.2019.9046080`.
URL `https://ieeexplore.ieee.org/document/9046080`

[15] A. Olofsson, T. Nordstrom, Z. Ul-Abdin, Kickstarting high-performance energy-efficient manycore architectures with epiphany, in: Asilomar Conference on Signals, Systems and Computers, Asilomar '14, IEEE, Pacific Grove, USA, 2014, pp. 1719–1726. `arXiv:1412.5538`, `doi:10.1109/`

ACSSC.2014.7094761.
URL http://ieeexplore.ieee.org/document/7094761/

[16] M.-H. Haghbayan, A. Miele, A. M. Rahmani, P. Liljeberg, H. Tenhunen, Performance/reliability-aware resource management for many-cores in dark silicon era, IEEE Transactions on Computers (TC) 66 (9) (2017) 1599–1612. doi:10.1109/TC.2017.2691009.
URL http://ieeexplore.ieee.org/document/7892847/

[17] M. Castro, E. Francesquini, F. Dupros, H. Aochi, P. O. Navaux, J.-F. Méhaut, Seismic wave propagation simulations on low-power and performance-centric manycores, Parallel Computing (PARCO) 54 (2016) 108–120. doi:10.1016/j.parco.2016.01.011.
URL https://linkinghub.elsevier.com/retrieve/pii/S0167819116000417

[18] J. Hasco ët, B. D. de Dinechin, P. G. de Massas, M. Q. Ho, Asynchronous one-sided communications and synchronizations for a clustered manycore processor, in: Symposium on Embedded Systems for Real-Time Multimedia, ESTIMedia '17, ACM Press, Seoul, 2017, pp. 51–60. doi:10.1145/3139315.3139318.
URL http://dl.acm.org/citation.cfm?doid=3139315.3139318

[19] B. D. de Dinechin, Y. Durand, D. van Amstel, A. Ghiti, Guaranteed services of the noc of a manycore processor, in: International Workshop on Network on Chip Architectures, NoCArc '14, ACM Press, Cambridge, 2014, pp. 11–16. doi:10.1145/2685342.2685344.
URL http://dl.acm.org/citation.cfm?doid=2685342.2685344

[20] M. Souza, P. H. Penna, M. Queiroz, A. Pereira, L. F. Góes, H. Freitas, M. Castro, P. Navaux, J.-F. Méhaut, Cap bench: A benchmark suite for performance and energy evaluation of low-power many-core processors, Concurrency and Computation: Practice and Experience (CCPE) 29 (4) (2017) 1–18. doi:10.1002/cpe.3892.

[21] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, G. Hunt, Helios: Heterogeneous multiprocessing with satellite kernels, in: ACM SIGOPS Symposium on Operating Systems Principles, SOSP '09, ACM Press, Big Sky, Montana, 2009, pp. 221–234. `doi:10.1145/1629575.1629597`.
URL `http://portal.acm.org/citation.cfm?doid=1629575.1629597`

[22] O. Villa, G. Palermo, C. Silvano, Efficiency and scalability of barrier synchronization on noc based many-core architectures, in: International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '08, ACM Press, Atlanta, USA, 2008, pp. 81–89. `doi:10.1145/1450095.1450110`.
URL `http://portal.acm.org/citation.cfm?doid=1450095.1450110`

[23] S. Wallentowitz, A. Lankes, A. Zaib, T. Wild, A. Herkersdorf, A framework for open tiled manycore system-on-chip, in: International Conference on Field Programmable Logic and Applications, FPL '2012, IEEE, Oslo, 2012, pp. 535–538. `doi:10.1109/FPL.2012.6339273`.
URL `http://ieeexplore.ieee.org/document/6339273/`

[24] P. H. Penna, D. Francis, 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, 2019, pp. 1–11.

[25] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, T. Strudel, A distributed run-time environment for the kalray mppa-256 integrated manycore processor, Procedia Computer Science 18 (2013 International Conference on Computational Science) (2013) 1654–1663. `doi:10.1016/j.procs.2013.05.333`.
URL `https://www.sciencedirect.com/science/article/pii/S1877050913004766?via%3Dihub`

[26] R. F. van der Wijngaart, T. G. Mattson, W. Haas, Light-weight communications on intel's single-chip cloud computer processor, SIGOPS Operat-

ing Systems Review (OSR) 45 (1) (2011) 73–83. doi:10.1145/1945023.1945033.
URL https://doi.org/10.1145/1945023.1945033

[27] C. Clauss, S. Lankes, P. Reble, T. Bemmerl, Evaluation and improvements of programming models for the Intel SCC many-core processor, in: International Conference on High Performance Computing & Simulation (HPCS), IEEE, 2011, pp. 525–532. doi:10.1109/HPCSim.2011.5999870.
URL http://ieeexplore.ieee.org/document/5999870/

[28] A. Varghese, B. Edwards, G. Mitra, A. P. Rendell, Programming the adapteva epiphany 64-core network-on-chip coprocessor, in: International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IPDPSW '14, IEEE, Phoenix, USA, 2014, pp. 984–992. doi:10.1109/IPDPSW.2014.112.
URL http://ieeexplore.ieee.org/document/6969488/

[29] M. Gamell, I. Rodero, M. Parashar, R. Muralidhar, Exploring cross-layer power management for PGAS applications on the SCC platform, in: International Symposium on High-Performance Parallel and Distributed Computing (HPDC), ACM Press, New York, USA, 2012, p. 235. doi:10.1145/2287076.2287113.
URL http://dl.acm.org/citation.cfm?doid=2287076.2287113

[30] O. Serres, A. Anbar, S. Merchant, T. El-Ghazawi, Experiences with UPC on TILE-64 processor, in: Aerospace Conference, IEEE, 2011, pp. 1–9. doi:10.1109/AERO.2011.5747452.
URL http://ieeexplore.ieee.org/document/5747452/

[31] J. Ross, D. Richie, Implementing openshmem for the adapteva epiphany risc array processor, Procedia Computer Science 80 (C) (2016) 2353–2356. arXiv:arXiv:1604.04205, doi:10.1016/J.PROCS.2016.05.439.
URL http://www.sciencedirect.com/science/article/pii/S1877050916309206

[32] M. Q. Ho, B. Tourancheau, C. Obrecht, B. D. de Dinechin, J. Reybert, MPI communication on MPPA many-core NoC: Design, modeling and performance issues, in: International Conference on Parallel Computing, Vol. 27 of ParCo '15, IOS Press, Edinburgh, UK, 2015, pp. 113–122. doi:10.3233/978-1-61499-621-7-113.
URL https://doi.org/10.3233/978-1-61499-621-7-113

[33] D. Richie, J. Ross, J. Infantolino, A Distributed Shared Memory Model and C++ Templated Meta-Programming Interface for the Epiphany RISC Array Processor, Procedia Computer Science 108 (2017) 1093–1102. doi:10.1016/J.PROCS.2017.05.221.
URL http://www.sciencedirect.com/science/article/pii/S1877050917308293