Compas'2019 : Parallélisme/ Architecture / Système
LIUPPA - IUT de Bayonne, France, du 24 au 28 juin 2019

The Hardware Abstraction Layer of Nanvix for the Kalray MPPA-256 Lightweight Manycore Processor [*]

Pedro Henrique Penna[1,2], Davidson Francis[2] and João Vicente Souto[3]

[1]Université Grenoble Alpes (UGA), Grenoble, France
[2]Pontifícia Universidade Católica de Minas Gerais (PUC Minas), Belo Horizonte, Brazil
[3]Universidade Federal de Santa Catarina (UFSC), Florianópolis, Brazil

Abstract
Lightweight manycores stand out for their performance, but lack on programmability and software portability. While these challenges may be tackled at Operating System (OS) level, existing systems that are narrowed for this context require some redesign, due to architectural intricacies that they do not completely handle. In this scenario, we introduce a Hardware Abstraction Layer (HAL) for lightweight manycores that cope with key issues that are often encountered when designing an OS for these processors. We present the interface exposed by our HAL, as well as a discussion about its implementation for the Kalray MPPA-256 manycore.

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

1. Introduction

Lightweight manycores differ from other high core count platforms in several architectural points. This next-generation of processors (i) integrate in a single die up to thousands of low-power cores; (ii) are designed to cope with Multiple Instruction Multiple Data (MIMD) workloads; (iii) rely on a high-bandwidth Network-on-Chip (NoC) for fast and reliable message-passing communication; (iv) present constrained memory systems; and (v) oftentimes feature a heterogeneous configuration. Some industry-successful examples of lightweight manycores are the Kalray MPPA-256 [5]; the Adapteva Epiphany [12]; and the Sunway SW26010 [20].
While the aforementioned architectural particularities granted lightweight manycores further performance scalability and energy efficiency, they introduced significant challenges in software development. For instance, due to the presence of rich NoCs, engineers are frequently required to adopt a message-passing programming model [8]. Additionally, the usually-missing hardware support for cache coherency forces programmers to handle it explicitly in software level and frequently calls out for a redesign in their applications [6]. Furthermore, the frequent presence of multiple physical address spaces and small local memories require data tiling and prefetching to be handled by the software [3]. Finally, the heterogeneous configuration turned the actual deployment of applications in lightweight manycores in a complex task [1].
Indeed, this poor programmability support currently arises in application- level because lower-level software layers, which are the OS and runtime libraries, do not completely handle architectural particularities of lightweight manycores transparently [15]. Due to this hotspot, sev-

[*] This work was supervised by Márcio Castro, François Broquedis, Henrique Freitas and Jean-François Méhaut.

eral research efforts are currently focused on addressing this challenge [4, 7, 17]. For instance, to ease the portability of existing software, as well as to broaden the actual applicability of lightweight manycores, some investigations are narrowed to get fully featured OSs running on them [2, 9, 11, 16]. Likewise, we target this long-term goal and also support the former research frontier, but we believe that important barriers are yet to be overcome before this scenario turns into reality. Architectural intricacies of lightweight manycores prevent commodity OSs to be simply ported, without undergoing through a heavy and complex re-design [2, 9, 11, 16]. Furthermore, existing OSs that are narrowed to these emerging processors still do not account in their design for some architectural points, such as the constrained memory system [15].

Overall, to enhance programmability for lightweight manycores and cope with software portability to them, we argue that an OS for next-generation processors should be redesigned from scratch around all their tight architectural constraints. Therefore, aiming towards this long-term objective, in this work we focus on addressing first-order programmability challenges that arise. More precisely, our goal is to introduce a generic and flexible HAL for lightweight manycores that cope with key issues that are often encountered when designing an OS for these processors. With this HAL, the development and deployment of a fully-featured OS becomes easier not only to a particular lightweight manycore, but also enables the portability of an OS across multiple of these emerging processors. Additionally, this works carries out a discussion on: (i) the integration of this HAL with Nanvix [13, 14, 15], the research OS for lightweight manycores that we are designing; and (ii) the deployment of our HAL on the Kalray MPPA-256.

The remainder of this work is organized as follows. In Section 2, we present an architectural overview of the Kalray MPPA-256, as well as the programmability challenges that it features and we target. In Section 3, we discuss the implementation of the HAL that we propose for Kalray MPPA-256. In Section 4, we present related works and highlight our main contributions to them. In Section 5, we draw our conclusions and future our research roadmap.

2. Lightweight Manycores: Performance, Programmability and Portability

In this section, we present the architectural particularities of lightweight manycores that drove the design of the HAL that we propose. To guide our discussion, we take the Kalray MPPA-256 as example, present an architectural overview of this lightweight manycore, and then precisely state those points that motivated us. It should be noted that the following discussion also holds for other lightweight manycores that our HAL targets, like OpTiMSoC [18] and HERO [10].

2.1. The Kalray MPPA-256 Lightweight Manycore Processor

Figure 1 presents an architectural overview of the Kalray MPPA-256 [5] processor, codenamed Bostan. It features 256 general-purpose cores, named Processing Elements (PEs), and 32 cores dedicated for system use, referred to as Resource Managers (RMs). The processor is built with 28 nm CMOS technology, and all cores (i.e., PEs and RMs) run at 500 MHz. Both RMs and PEs implement a 64-bit capable proprietary instruction set and present a 5-issue Very Long Instruction Word (VLIW) architecture. Furthermore, these cores feature level-1 private instruction and data caches, and a Memory Management Unit (MMU) with software-managed Translation Lookaside Buffers (TLBs) for virtual memory support. 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 and a 2 MB of local SRAM. In these clusters, level-1 data cache coherence is not supported by the hardware. On the other hand, each I/O Cluster features 4 RMs level-2 private instruction cache, level-2 shared data cache, and 4 MB of

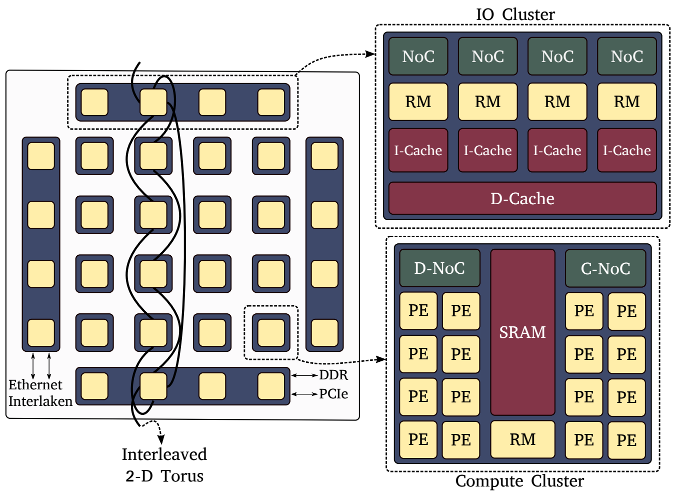

Figure 1: Architectural overview of the Kalray MPPA-256 processor.

local SRAM. In contrast to Compute Clusters, these clusters support level-1 data cache coherence, which may be disabled if required. The north and south I/O Clusters have a DDR3-1600 controller each, thereby enabling access to up to 64 GB of DRAM.
Clusters have a distinct physical address spaces, and they may communicate with one another by explicitly 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. Clusters are connected to C-NoC and D-NoC interfaces and have at their disposal DMA engines to transfer data between these interfaces and the local memories.

2.2. OS Development Challenges

Performance capabilities of Kalray MPPA-256 can be quickly drawn from the architectural overview that we presented previously. Therefore, in the paragraphs that follow, we turn our focus to further detail those architectural particularities that introduce challenges in OS implementation and portability for this lightweight manycore. In the next section, we discuss how these issues drove the design of the HAL that we propose.

VLIW Architecture PEs and RMs feature a VLIW pipeline. That is, they fetch and issue a bundle of instructions at once. Instructions in the same bundle are executed in parallel, thus they should not have neither data nor functional unit dependency among themselves. When writing assembly code, it is up to the kernel engineer to ensure this requirement.

Cache Coherency In Compute Clusters, hardware-level coherency for memory caches is not supported. However, VLIW cores feature special machine instructions to invalidate, flush and reload data caches, and thus enable coherency to be implemented in software-level.

TLB Management Hardware TLBs are software-managed and feature a hierarchical design. They are formed by the union of a small fully associative TLB, called Locked TLB (LTLB); and a large 2-way set associative TLB, named Join TLB (JTLB). Entries of both TLBs may encode pages of arbitrary sizes, ranging from 4 kB to 512 kB. However, LTLB entries may be configured when powering on the processor, and JTLB entries may be dynamically programmed through privileged machine instructions.

Direct Memory Access (DMA) Engines To transfer data between the local memories and the NoC interfaces, clusters must rely on DMA engines. Each of these specialized micro-cores has 256 inputs and 8 outputs buffers for the D-NoC, and 128 inputs and 1 output buffers for the C-NoC. DMA engines may be programmed to perform either synchronous or asynchronous transfers, and in the latter case, up to 8 operations may be on-going at the same time. The number of DMA engines varies from one cluster to another: Compute Clusters have a single of these units, whereas I/O Clusters have 4.

3. A Hardware Abstraction Layer for Lightweight Manycores

Figure 2 presents an overview of the HAL that we propose. Our implementation of it is open-source [2] and currently supports the Kalray MPPA-256 and OpTiMSoC processors. Overall, this HAL is structured in two major logic layers: one that abstracts the management of a single cluster, which is named *Cluster Abstraction Layer*; and another that encapsulates architectural features that spawn across multiple clusters, which is called *Processor Abstraction Layer*. In turn, each of these layers group several modules that abstract specific components of the underlying hardware: (i) the *Memory Management* module; (ii) the *Core Management* module; (iii) the *Inter-Cluster Communication* module; and (iv) *Debugging and Monitoring* module. Due to space limitations, in the paragraphs that follow we discuss the first three of the modules mentioned above, which indeed summary the contributions of our work.
The *Memory Management* abstraction module exposes a uniform view of TLBs and paging structures of cores in a cluster. Furthermore, it exports routines for: (i) flushing, reloading and invalidating caches; (ii) encoding, writing and reading TLB entries; and (iii) looking up and changing paging structures. We relied on two great decisions to design this interface, and thus enable a fully featured and coherent-capable memory management system to be written on top of it. First, by providing a standard view of the TLBs and paging structures, as well as routines to operate on these, the memory management system of an OS kernel running on top of our HAL does not need to worry about the actual layout of these hardware components. For instance, recall that in Kalray MPPA-256, the TLB is split into two smaller structures (i.e., JTLB and LTLB), with different lengths and associativity capabilities. If this complexity was handled to the overlying memory management system, then the kernel would be platform-dependent – which is not what we aim at. Second, we chose to expose a software-managed view of the memory hierarchy, even though it may be fully-managed by the hardware in some targets. In summary, our motivation for this decision is two-fold. In architectural families which these memory hardware components may be managed in software, an important performance improvement can be achieved when the actual management takes place in higher software levels. Whereas in architectural families which manage caches and TLBs at the hardware level, no performance drawback is observed if such interface is exposed, because indeed the underlying implementation is linked to dummy wrappers (i.e., do nothing). Overall, our motivation to expose a software-managed view of the memory hierarchy comes from the following observation. To make this point clear, notice that with such abstraction module, an OS running on Kalray MPPA-256 may choose which of its internal structures shall be kept coherent or not. In Annex A we further discuss the HAL interface for managing the TLB.
The *Core Management* abstraction module exposes routines for powering on/off, starting, stopping and suspending instruction execution in the cores of a cluster. Furthermore, it also provides a lightweight lock interface, and a uniform numbering scheme for hardware interrupts,

[2] Publicly available under the MIT Licence at: `https://github.com/nanvix/microkernel`.

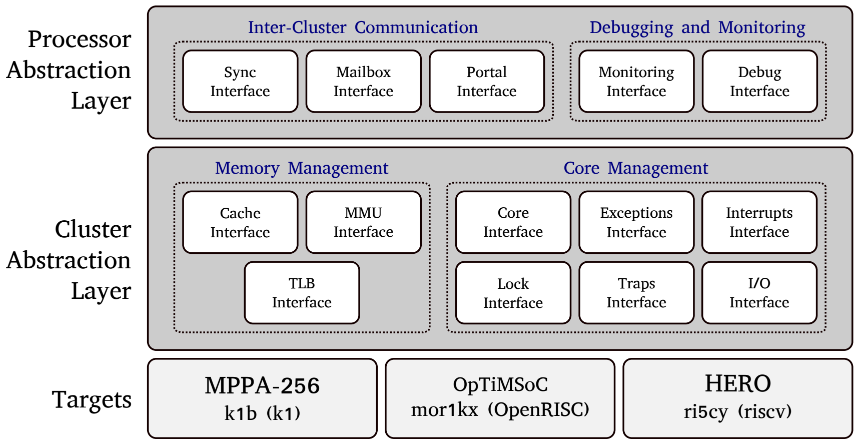

Figure 2: Structural overview of the proposed HAL.

exceptions and traps, as well as routines for registering handlers for them. Overall, this abstraction module was designed so that the following OS features could be directly built on its top, without requiring additional architecture-dependent code to be written: (i) a fully featured thread management and synchronization system and (ii) rich interrupt/exception handling, and (iii) a system call interface. Note that we decided to not expose routines specifically for saving and restoring the execution context of a core, but we rather chose to encapsulate these the functionalities in the routines for stopping and (re)starting instruction execution. We were motivated to do so because we found out that if we decoupled this functionality, a thread system built on top of our HAL would need to implement some assembly routines. While this may enable performance to be slightly tuned, we argue that: (i) it would go against the conceptual idea of our HAL, which is to enable the overlying kernel to be platform-independent; and (ii) the actual development of efficient assembly code may be a complex task in some platforms. For instance, in a VLIW architecture, such as Kalray MPPA-256, recall that it is up to the kernel engineer to ensure that instruction bundles are well composed. In Annex B we further detail the HAL interface for managing cores.

The *Inter-Cluster Communication* module exports three major abstraction modules and operations on them, which all together enable clusters to exchange data: *sync*, *mailbox*, and *portal*. A *sync* enables a set of clusters to wait for a notification from a cluster, and thus provide the bare bones for inter-cluster synchronization. It is an analogous abstraction to POSIX signals, with the sole difference that a *sync* notification carries no other information but a *wakeup* one. The *mailbox* abstraction enables clusters to exchange fixed-size messages with one another. A message is intended to encode small operations and system control signals, and can have one or many recipients. Finally, the *portal* abstraction enables dense data transfers between clusters, either synchronously or asynchronously (if supported by hardware). A *portal* can be opened just between a pair of clusters and it features a built-in flow control semantic, likewise POSIX pipes. Notice that in this design we decoupled small from large data transfers by exporting two abstractions (i.e., *mailboxes* and *portals*). We were motivated to do so because in this way we could expose some control over Quality of Service (QoS) to the overlying OS kernel. For instance, in lightweight manycores that feature multiple NoCs, with possibly different bandwidths, such as Kalray MPPA-256, one NoC may be exclusively used for *syncs* and *mailboxes* and another one for *portals*. In Annex C, we depict how an OS service for remote memory access may be implemented on top of this interface.

4. Related Work

Several OS kernels were so far proposed to address programmability and portability challenges in lightweight manycores at system level [1, 2, 9, 19]. Overall, these solutions focus on delivering to user-level applications high-level programming abstractions, such as threads and files, so that software development experience is improved. In contrast to these works, our HAL lies one level below, and focus on providing a uniform architectural of several lightweight manycores, so that the portability and implementation of an OS kernel becomes easier. Indeed, one can say that a HAL is the lowest-level layer of an OS kernel, and thus existing kernels narrowed for lightweight manycores may be contrasted to our solution. While this claim holds at first-instance, we argue that these kernels overlook to some architectural features that are a trend in the next-generation of lightweight manycores [15], such as VLIW cores, no support for cache coherency at the hardware level, complex hardware TLB structures and DMA engines for NoC-based communication. Nevertheless, it is worth noting that the design of the *Core Management* of our HAL borrowed different ideas from the HAL of Barrelfish [2]. Furthermore, for the design of the *Inter-Cluster Communication* interface, we relied on ideas proposed along with the NodeOS distributed runtime system [5]. However, in contrast to this runtime environment, our HAL exposes flow control operations for NoC connectors, so that QoS and data transfers may be fully controlled by the overlying OS kernel.

5. Conclusions

Lightweight manycores are well known for their performance, but currently lack on programmability and software portability. While these challenges are inherently introduced by architectural features of these processors, several efforts look at bridging them at OS level [1, 2, 9, 19]. Likewise, we support this research frontier and target the same long-term goal, but we argue that important barriers are yet to be bridged before this scenario turns into reality. For instance, existing OSs do not account in their design for constrained memory systems of lightweight manycores nor their heterogeneous configuration [15]. Indeed, we believe that in order to cope with these issues, an OS that addresses this next-generation of processors should be re-designed from scratch around all their tight architectural constraints. Therefore, targeting this long-term goal, in this work we aim at introducing a HAL for lightweight manycores that copes with key issues that emerge when designing OSs for these processors. With this HAL, the development and deployment of a fully-featured OS becomes easier not only to a particular lightweight manycore, but also enables the portability of its kernel across multiple of these processors.
This work is actually inserted in the context of a larger joint-research project that aims at improving programmability and software portability in lightweight manycores by means of a fully-featured POSIX-compliant OS. This system, which we named Nanvix, features a multikernel design [2] and it is being designed from scratch around architectural issues of lightweight manycores. In [15] we presented a prototype of a memory service of Nanvix, which is indeed a core service in our system. We deployed this prototype on the Kalray MPPA-256 lightweight manycore and our experimental results motivated a bare metal implementation of our OS. The HAL that we presented in this work consists of the first building block towards this implementation. On top of it, we are currently designing and implementing a microkernel that shall run in each cluster of a lightweight manycore and provide bare bones system abstractions, such as thread management, thread synchronization, and virtual memory support. In future works, we intend to present a discussion and evaluation of our microkernel.

Acknowledgements

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

References

1. Barbalace (A.), Sadini (M.), Ansary (S.), Jelesnianski (C.), Ravichandran (A.), Kendir (C.), Murray (A.) et Ravindran (B.). – Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms. – In *Proceedings of the 10th European Conference on Computer Systems, EuroSys '15*, EuroSys '15, pp. 1–16, Bordeaux, France, avril 2015. ACM.
2. Baumann (A.), Barham (P.), Dagand (P.-E.), Harris (T.), Isaacs (R.), Peter (S.), Roscoe (T.), Schüpbach (A.) et Singhania (A.). – The Multikernel: A New OS Architecture for Scalable Multicore Systems. – In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP '09*, SOSP '09, pp. 29–44, Big Sky, Montana, USA, octobre 2009. ACM.
3. Castro (M.), Francesquini (E.), Dupros (F.), Aochi (H.), Navaux (P. O.) et Méhaut (J.-F.). – Seismic wave propagation simulations on low-power and performance-centric manycores. *Parallel Computing*, vol. 54, 2016, pp. 108—-120.
4. Christgau (S.) et Schnor (B.). – Exploring One-Sided Communication and Synchronization on a Non-Cache-Coherent Many-Core Architecture. *Concurrency and Computation: Practice and Experience (CCPE)*, vol. 29, n15, mars 2017, p. e4113.
5. de Dinechin (B. D.), de Massas (P. G.), Lager (G.), Léger (C.), Orgogozo (B.), Reybert (J.) et Strudel (T.). – A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. – In *Procedia Computer Science, ICCS '13*, volume 18, pp. 1654–1663, Barcelona, Spain, juin 2013. Elsevier.
6. Francesquini (E.), Castro (M.), Penna (P. H.), Dupros (F.), Freitas (H.), Navaux (P.) et Méhaut (J.-F.). – On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms. *Journal of Parallel and Distributed Computing (JPDC)*, vol. 76, nC, février 2015, pp. 32–48.
7. Gamell (M.), Rodero (I.), Parashar (M.) et Muralidhar (R.). – Exploring Cross-Layer Power Management for PGAS Applications on the SCC Platform. – In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing, HPDC '12*, HPDC '12, pp. 235–246, Delft, The Netherlands, juin 2012. ACM.
8. Kelly (B.), Gardner (W.) et Kyo (S.). – AutoPilot: Message Passing Parallel Programming for a Cache Incoherent Embedded Manycore Processor. – In *Proceedings of the 1st International Workshop on Many-core Embedded Systems, MES '13*, MES '13, pp. 62–65, Tel-Aviv, Israel, juin 2013. ACM.
9. Kluge (F.), Gerdes (M.) et Ungerer (T.). – An Operating System for Safety-Critical Applications on Manycore Processors. – In *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC '14*, ISORC '14, pp. 238–245, Reno, Nevada, USA, juin 2014. IEEE.
10. Kurth (A.), Vogel (P.), Capotondi (A.), Marongiu (A.) et Benini (L.). – Hero: Heterogeneous embedded research platform for exploring risc-v manycore accelerators on fpga. – In *Proceedings of Computer Architecture Research with RISC-V Workshop (CARRV' 17)*, 2017-10. – First Workshop on Computer Architecture Research with RISC-V (CARRV 2017); Conference Location: Boston, MA, USA; Conference Date: October 14, 2017.
11. Nightingale (E.), Hodson (O.), McIlroy (R.), Hawblitzel (C.) et Hunt (G.). – Helios: Het-

erogeneous Multiprocessing with Satellite Kernels. – In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09*, SOSP '09, pp. 221–234, Big Sky, Montana, USA, octobre 2009. ACM.

12. Olofsson (A.), Nordstrom (T.) et Ul-Abdin (Z.). – Kickstarting High-Performance Energy-Efficient Manycore Architectures with Epiphany. – In *2014 48th Asilomar Conference on Signals, Systems and Computers, ACSSC '14*, ACSSC '14, pp. 1719–1726, Pacific Grove, California, USA, novembre 2014. IEEE.
13. Penna (P. H.), Castro (M.), Freitas (H.), Méhaut (J.-F.) et Caram (J.). – Using the Nanvix Operating System in Undergraduate Operating System Courses. – In *2017 VII Brazilian Symposium on Computing Systems Engineering, SBESC '17*, SBESC '17, pp. 193–198, Curitiba, Brazil, novembre 2017. IEEE.
14. Penna (P. H.), Castro (M.), Freitas (H.), Méhaut (J.-F.) et Caram (J.). – Using the Nanvix Operating System in Undergraduate Operating System Courses. – In *2017 VII Brazilian Symposium on Computing Systems Engineering, SBESC '17*, SBESC '17, pp. 193–198, Curitiba, Brazil, novembre 2017. IEEE.
15. Penna (P. H.), Souza (M.), Jr. (E. P.), Souto (J.), Castro (M.), Broquedis (F.), Freitas (H.) et Méhaut (J.-F.). – Asynchronous One-Sided Communications and Synchronizations for a Clustered Manycore Processor. – In *Twelfth International Workshop on Programmability and Architectures for Heterogeneous Multicores, MultiProg 2019*, MultiProg 2019, pp. 51–60, Valencia, Spain, jan 2019.
16. Rhoden (B.), Klues (K.), Zhu (D.) et Brewer (E.). – Improving Per-Node Efficiency in the Datacenter with New OS Abstractions. – In *Proceedings of the 2nd ACM Symposium on Cloud Computing, SoCC '11*, SoCC '11, pp. 1–8, Cascais, Portugal, octobre 2011. ACM.
17. Serres (O.), Anbar (A.), Merchant (S.) et El-Ghazawi (T.). – Experiences with UPC on TILE-64 Processor. – In *Aerospace Conference, AERO '11*, AERO '11, pp. 1–9, Big Sky, Montana, USA, mars 2011. IEEE.
18. Wallentowitz (S.), Wagner (P.), Tempelmeier (M.), Wild (T.) et Herkersdorf (A.). – *Open Tiled Manycore System-on-Chip*. – Rapport technique narXiv:1304.5081, ArXiV, avril 2013.
19. Wisniewski (R.), Inglett (T.), Keppel (P.), Murty (R.) et Riesen (R.). – mOS: An Architecture for Extreme-Scale Operating Systems. – In *ROSS '14 Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS '14*, ROSS '14, pp. 1–8, Munich, Germany, juin 2014. ACM.
20. Zheng (F.), Li (H.-L.), Lv (H.), Guo (F.), Xu (X.-H.) et Xie (X.-H.). – Cooperative Computing Techniques for a Deeply Fused and Heterogeneous Many-Core Processor Architecture. *Journal of Computer Science and Technology (JCST)*, vol. 30, n1, janvier 2015, pp. 145–162.

A. HAL TLB Interface

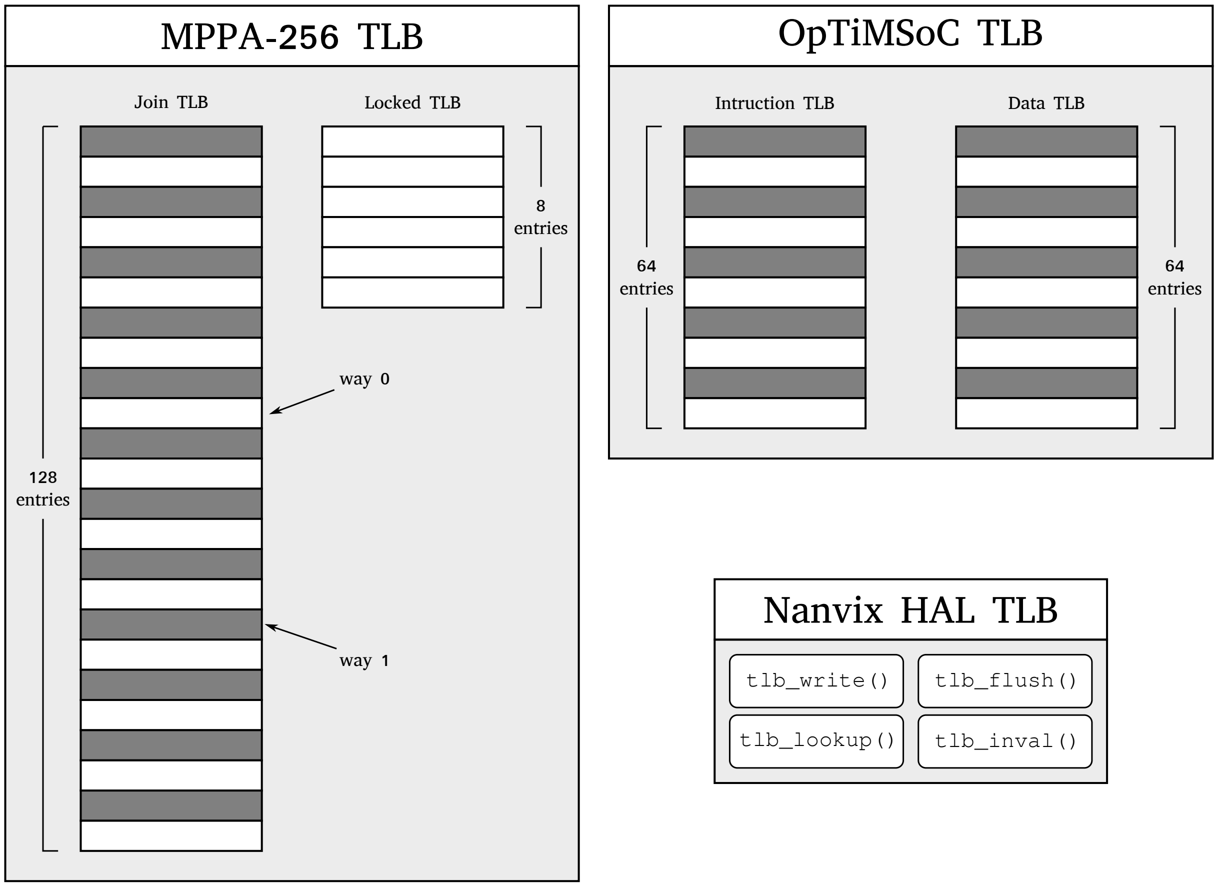

Figure 3: TLBs of Kalray MPPA-256 and OpTiMSoC, and the interface exposed by our HAL.

In Figure 3 we depict the architectural TLBs of the Kalray MPPA-256 (left) and OpTiMSoC (upper right) lightweight manycore processors, as well as the interface exposed by our HAL for managing hardware TLBs (bottom right). It is important to remark how much the hardware TLBs of these two processors differ, and how our HAL bridges this heterogeneity. Overall, TLBs of both processors are software-managed, but feature a different design. In Kalray MPPA-256, TLBs are formed by the union of a small fully associative (LTLB); and a large 2-way set associative TLB (JTLB). Entries of both, LTLB and JTLB, may hold address for data and instruction pages. In contrast, in OpTiMSoC, the TLB is split in a Instruction TLB (ITLB) and a Data TLB (DTLB). Both, ITLB and DTLB, feature a 2-way set associativity and have the same length, but the former may encode only pages that hold code (i.e., executable and readable, but not writable), and the latter pages that hold data (readable and/or executable, but not writable). An OS kernel running on both of these processors, would need to deal with the different organization, geometry and associativity, if not using our HAL. In contrast, our interface exposes to the overlying kernel a virtual TLB which features full associativity and each of its entries may encode either instruction or data pages.

B. HAL Core Interface

```
/*
 * Powers on a core.
 */
extern void core_startup(void);

/*
 * Shutdowns the underlying core.
 *   - status Shutdown status.
 */
extern void core_shutdown(int status);

/*
 * Starts a core.
 *   - coreid ID of the target core.
 *   - start  Starting routine to execute.
 */
extern void core_start(int coreid, void (*start)(void));

/*
 * Stops and resets instruction execution in the underlying core.
 */
extern void core_reset(void);

/*
 * Suspends instruction execution in the underling core.
 */
extern void core_sleep(void);

/*
 * Resumes instruction execution on a sleeping core.
 */
extern void core_sleep(void);
```

Snippet 1: API overview of the core interface of our HAL.

In Snippet 1, we present the programming interface exposed by our HAL to manage instruction execution in the cores a lightweight manycore. The `core_startup()` routine is the first one called when the core powers on, and it actually setups and initializes all architectural structures of underlying hardware, such as the interrupt vector tables, TLBs and page tables. In contrast, the `core_shutdown()` function carries the analogue operating, by de-initializing hardware structures that are needed, and then powering off the underlying core. The next three functions, which are `core_start()`, `core_reset()`, `core_sleep()` and `core_wakeup()` may be used to start, stop, suspend and resume instruction execution in a core, respectively; and thus they provide the bare bones infra-structure for implementing a fully-featured thread management system For instance, a high-level routine for creating a thread would invoke `core_start()`, supply as a parameter the identifier of the core in which instruction execution should start. Conversely, whenever the thread finishes its execution, it would call `core_reset()`. Finally, thread switching as well as sleep/wakeup synchronization primitives may be implemented on top of `core_sleep()` and `core_wakeup()`.

C. HAL Inter-Cluster Communication Interface

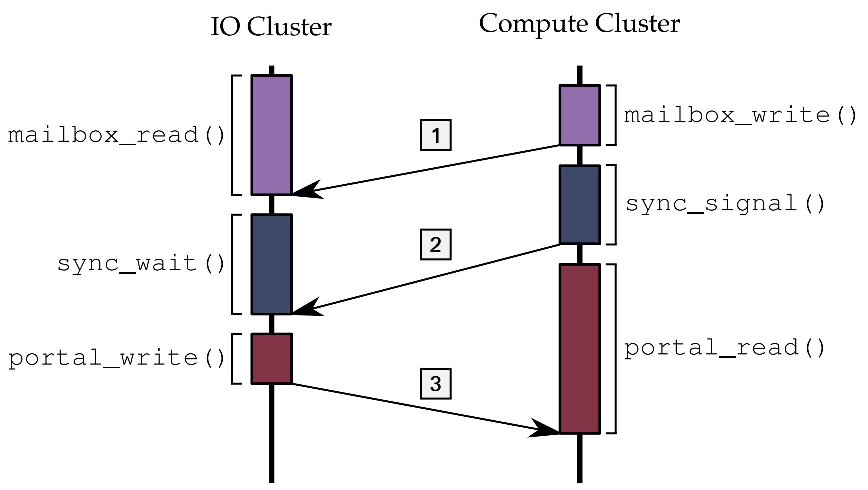

Figure 4: An OS protocol for remote memory accesses on top of the proposed HAL.

In Figure 4 we present how the inter-cluster communication interface exposed by our HAL may be used to implement a protocol for accessing remote memory in a lightweight manycore [15]. In this protocol, the IO Cluster is the server of remote memory, and the Compute Cluster is the client of this service. Aside from the details of the protocol itself, it is important to note how the *mailbox*, *portal* and *sync* connectors of our interface were used. Initially, the, the server is blocked, awaiting for a request to pop up in its input *mailbox*. As soon the client sends a request via an output *mailbox* (1), the server unblocks, processes the request, and blocks in a *sync*, until the remote client informs back that it is ready to receive data. As soon as data is ready to be received, the client unblocks the remote server (2), which sends back to him the requested data through an output *portal* (3).