RMem: An OS Service for Transparent Remote Memory Access in Lightweight Manycores

Pedro Henrique Penna, Matheus Souza, Emmanuel Junior, João Souto, Márcio Castro, Francois Broquedis, Henrique Cota de Freitas, Jean-François Mehaut

RMem: An OS Service for Transparent Remote Memory Access in Lightweight Manycores

Pedro Henrique Penna[1,2], Matheus Souza[2],
Emmanuel Podestá Junior[3], João Souto[3], Márcio Castro[3],
François Broquedis[4], Henrique Freitas[2], and Jean-François Méhaut[1]

[1] Université Grenoble Alpes (UGA), France
[2] Pontifícia Universidade Católica de Minas Gerais (PUC Minas), Brazil
[3] Universidade Federal de Santa Catarina (UFSC), Brazil
[4] Institut Polytechnique de Grenoble (Grenoble INP), France

Abstract. Lightweight manycores deliver high performance and scalability at low power consumption. However, architectural intricacies of these processors impose programmability challenges that keep them away from mass adoption. While several efforts aim at introducing parallel programming environments to lightweight manycores, few initiatives are concerned about how to design rich Operating Systems (OSs) to them. In this work, we focus on the open challenges that arise from constrained memory subsystems of lightweight manycores, such as the presence of multiple address spaces and limited on-chip memory. To cope with transparent data access in this scenario, we introduce an OS service, named RMem. This service provides a shared memory abstraction over multiple address spaces and exposes system calls that enable one-sided communication on top of this abstraction. We implemented a prototype of our service in the Nanvix research OS, and we deployed the system the Kalray MPPA-256 lightweight manycore. Our experimental results with a microbenchmark unveiled that, while exposing an easier-to-program interface, the RMem Service may deliver about 91% of the write performance and up to 2.4× better read performance than the primitives in the libraries of the experimental platform.

1 Introduction

During the past decade, performance improvements were mostly achieved by scaling up the number of processors in a system. However, the rapid growth in design complexity and power consumption pushed investigations towards lightweight manycores [7]. To address the ever-increasing performance demands of applications, research efforts focused on bundling in a chip as many simple low-power cores as possible, rather than integrating fewer complex and power-hungry cores. For instance, European researchers launched the Mont-Blanc Project with the goal of designing a scalable and power-efficient High-Performance Computing (HPC) platform based on low-power embedded technologies [30]; and, the

Sunway TaihuLight Supercomputer, the second most-powerful computer ranked in Top 500[5], was built using 10.6 million cores running at 1.45 GHz [12].

Lightweight manycores differ from large-scale multicores and other manycore platforms in several points, besides their operational frequency. Unlike Graphics Processing Units (GPUs), which are meant to be used as computing accelerators and primarily target Single Instruction Multiple Data (SIMD) workloads, lightweight manycores are designed to cope with Multiple Instruction Multiple Data (MIMD) workloads and thus may be employed in several other contexts. In contrast to large-scale multicore platforms, lightweight manycore processors feature a different architecture: (i) they integrate up to thousands of cores in a single chip; (ii) they rely on a high-bandwidth Network on Chip (NoC) for fast and reliable message-passing communication; and (iii) they present constrained memory subsystems. Some examples of such emerging class of processors are the Tilera TILE64 [6]; the Intel Single-Cloud Computer [15]; the Kalray MPPA-256 [10]; the Adapteva Epiphany [20]; and the Sunway SW26010 [31].

While the aforementioned architectural differences granted lightweight manycores better scalability and energy efficiency than their counterparts, they introduced challenges in software design. For instance, engineers are oftentimes required to adopt a message passing programming model [16], and the missing hardware cache coherency forces programmers to not only handle data coherency in software level but also calls out for a redesign in applications [29]. Furthermore, the heterogeneity trend in lightweight manycores turned the actual deployment of applications a daunting task [3]. Indeed, enhancing the programmability support for lightweight manycores consists of a hot research subject, in which multiple efforts are currently focused on. While some initiatives aim at proposing parallel programming models, environments and frameworks to these processors [8,13,28]; other investigations push towards the way in which Operating Systems (OSs) should be designed for lightweight manycores [5,17,19,25]. Researchers working in the latter frontier argue that a rich OS may broaden the applicability of lightweight manycores. We likewise support this claim, but we also acknowledge that multiple challenges should still be overcome before this scenario turns into reality. For instance, existing OSs that are narrowed to lightweight manycores do not account in their design for the presence of multiple address spaces nor the limited amount of on-chip memory, thereby requiring engineers to explicitly deal with data accessing, tiling and prefetching [11,29]. As a side effect, several other problems arise and remain unsolved, such as efficient process migration and thread placement, and effective multiprogramming.

We argue that the OS of a lightweight manycore should be designed from scratch around the tight architectural constraints of the memory subsystem. Therefore, towards the long-term goal of engineering an OS that additionally meets this requirement, in this work, we focus on addressing the first-order programmability challenge in this context. More precisely, our goal is to introduce a new facility that enables transparent data access at the OS level for lightweight manycores that features multiple address spaces and a limited amount of on-chip

[5] Available at: https://www.top500.org

memory. Overall, this work delivers the following contributions to the state-of-the-art programmability support for lightweight manycores:

- An OS-level facility that provides a flat shared memory abstraction over multiple address spaces. We called our solution Remote Memory (RMem), and it is provided by a new OS service that we named RMem Service. Our abstraction enables different applications to effectively communicate and share data in lightweight manycores with the targeted constraints. Furthermore, it also enables the OS to transparently migrate data around, to better exploit locality and thus mitigate NoC congestion.
- An interface that exposes one-sided communication primitives on top of RMem. By relying on these primitives, processes may transparently access data, apart from their location; as well as to manipulate large amounts of data, regardless of the limited on-chip memory. The provided interface features two system calls: `memread()`, which copies data from remote to local memory, and `memwrite()` which carries out the converse operation.
- The deployment of an RMem prototype in the Kalray MPPA-256 lightweight manycore. We implemented our prototype in the Nanvix research OS [22,23] and made it publicly available at `https://github.com/nanvix/multikernel`.

In addition to these contributions, in this work, we also present a performance evaluation of RMem using a microbenchmark. We ran multiple experimental configurations, and we compared the performance of RMem system calls with the primitives in the libraries that are shipped with Kalray MPPA-256.

The remainder of this work is organized as follows. In Section 2, we discuss the programmability challenges that we target in lightweight manycores. In Section 3, we present our OS service (RMem). In Section 4, we detail the evaluation methodology and, in Section 5, the experimental results. In Section 6, we present an overview of related works and highlight our main contributions to them. In Section 7, we present our conclusions and discuss future works that we aim.

2 Target Challenges on Lightweight Manycores

In this section, we precisely position the programmability challenges of lightweight manycores that we target. We do so by taking the Kalray MPPA-256 processor as an example, presenting a detailed view of its architecture, and then pointing out how those challenges arise from its constrained memory subsystem.

2.1 The Kalray MPPA-256 Processor

Figure 1 presents an architectural overview of Kalray MPPA-256 [10], codenamed Bostan. It features 256 general-purpose cores, named Processing Elements (PEs), and 32 cores dedicated for system use, referred as Resource Managers (RMs). All cores run at 400 MHz, and the processor is built using 28 nm CMOS technology. This manycore was introduced to target compute-intensive and time-critical applications of the embedded computing market share. Notwithstanding, due to its

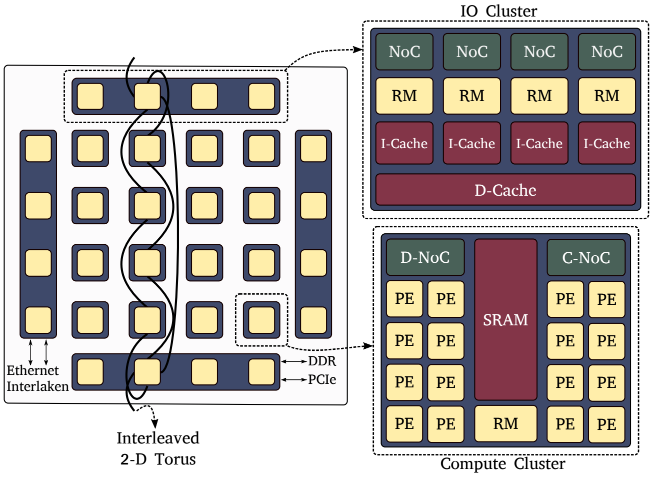


Fig. 1: Architectural overview of the Kalray MPPA-256 processor.

high energy efficiency [11], Kalray MPPA-256 also offers a promising alternative for other applications as well.

Both RMs and PEs implement a proprietary instruction set and present a pipelined 5-way Very Long Instruction Word (VLIW) architecture. Furthermore, these cores feature private 2-way associative instruction and data level-1 caches of 8 kB each and a Memory Management Unit (MMU) for software-managed virtual memory. PEs and RMs are grouped within 16 compute clusters and 4 I/O clusters, respectively. Each compute cluster features 16 PEs, 1 RM and a 2 MB of local Scratchpad Memory (SPM). Hardware cache coherence in compute clusters is not supported. On the other hand, each I/O cluster features 4 RMs with a shared level-2 coherent data cache, and a DDR controller, which enables the access to up to 64 GB of external DDR3-1600 at 8.5 GB/s.

Clusters have a distinct physical address spaces each, and they may communicate with one another by two different NoCs: a Control NoC (C-NoC) that features low bandwidth and it is intended for small data transfers; and a Data NoC (D-NoC) that presents high bandwidth and thus is dedicated to dense data transfers. Both NoCs present an interleaved 2-D torus topology with bidirectional links, wormhole switching and XY routing with injection rate control. Overall, D-NoC may deliver up to 1.6 GB/s per link and direction at 400 MHz.

2.2 Programmability Challenges from the Memory Subsystem

The Kalray MPPA-256 processor features a constrained memory subsystem. The memory itself is physically distributed; several physical address spaces co-exist and MMUs cannot be configured to provide a single address space at hardware level; the amount of memory in each compute cluster is very limited for nowadays applications that typically run in desktops and servers (2 MB); and caches of compute clusters are not coherent. In the next paragraphs, we detail some of the first-order programming challenges that arise in this scenario, and following we summary our motivation for this work.

Inter-Cluster Communication and Data Access Threads that run on different clusters necessarily work on different address spaces. If a thread A running on cluster 1 wants to communicate to a thread B running on cluster 2, it must either use explicit message-passing primitives or Direct Memory Access (DMA) transfers. Unfortunately, in both cases, thread A needs to know the physical location where thread B is running, at the bare minimum. Consequently, if thread B is deployed in a different location, communication fails.

Data Tiling Each compute cluster features 2 MB of local memory and no advanced virtual memory facility. As a consequence, if a thread wants to manipulate a dataset that is larger than 2 MB, it has to explicitly tile the dataset into chunks, load/store them from/to a remote memory, and locally manipulate these chunks one at a time.

Data Prefetching The actual way in which data accesses and tiling are carried out directly impacts on the time in which threads waste due to NoC latencies. Therefore, to achieve high-performance, an application should rely on asynchronous transfers and prefetching schemes, so that it overlaps communication with computation as much as possible.

Apart from runtime alternatives that target these challenges, OS-level solutions for them were not so far investigated. We argue that inter-cluster communication, explicit data accessing and data tiling impose important programmability barriers for lightweight manycores. If data cannot be accessed regardless where it is placed, nor arbitrarily large amounts of data can be transparently manipulated, not only software development becomes more challenging and non-portable, but also essential OS mechanisms, such as process and data migration, core multiplexing, and core partitioning, may not be addressed. In summary, these observations motivated our work.

3 The Remote Memory Service

In this section, we present the RMem Service in a top-down fashion. First, we introduce the system call interface that is exported by it; then, we detail the design of RMem; and finally, we unveil insights of the implementation of our prototype in the Kalray MPPA-256 lightweight manycore processor.

3.1 System Call Interface Overview

The RMem Service exposes two blocking system calls: (i) `memread()`, which copies data from a remote to the local memory; and (ii) `memwrite()`, which carries out the converse transfer. By relying on these primitives, it is possible to access data regardless of where it is physically placed, as well as manipulate large amounts of data without having to deal explicitly with tiling.

Code Snippet 1.1 depicts how these two primitives may be naively used to carry out a scalar-vector multiplication. In this example, `a` is a pointer to the

```
1 void scalar_vector_multiply(double *a, int n, double f)
2 {
3     double tmp;
4     for (int i = 0; i < n; i++) {
5         memread(&tmp, &a[i], sizeof(double));
6         tmp *= f;
7         memwrite(&tmp, &a[i], sizeof(double));
8     }
9 }
```

Snippet 1.1: Scalar-vector multiplication using RMem Service.

remote memory address where the target vector resides, `n` is the length of the array itself, and `f` is the scalar multiplying factor. First, the i-th element of the vector is loaded into a local variable `tmp`, using the `memread()` operation (line 5). Then, the actual computation is performed (line 6). Finally, the resulting element is written back to the remote memory, using the `memwrite()` operation (line 7). Both system calls handle underlying data transfers between the local and remote memories in a thread-safe fashion. Furthermore, they perform bounds and permissions checks to ensure access is granted only to allowed data.

At this point, it is essential to state the differences between the system call interface exposed by the RMem Service from existing solutions that lie in the runtime system level, such as message-passing libraries and Partitioned Global Address Space (PGAS) frameworks. First, the unified addressing scheme exposed by our service enables data to be transparently accessed. That is, threads do not need to know where the target data is placed. Second, our system call interface allows a thread to expand its address space and thereby transparently manipulate datasets that are effectively too large to fit in the local memory. Third, thanks to the shared memory abstraction provided by RMem, processes may effectively share data and operate on it, as they would do in a shared memory programming environment. Fourth, our service features bounds and permissions checks, thereby enabling two unrelated threads (i.e., threads of different applications) to communicate with each other safely. Finally, the RMem Service does not require communication peers to be paired at all. One thread may write data to the remote memory regardless if there is a reader thread in the other end.

3.2 Design Discussion

Figure 2 pictures an architectural overview of the RMem Service. Essentially, it features a distributed structure, so that it matches the distributed configuration that is inherent in lightweight manycores; and it is in conformance with the multikernel OS design approach, which is a trending OS architecture for manycores [5,17,19,25]. In our design, the RMem Service is implemented on top of multiple system engines and facilities: (i) Name Client; (ii) Name Server; (iii) IPC Connectors; (iv) RMem Client; and (v) RMem Server. To simply explain the internals of our service, as well as present how it actually works, let us consider an example, where a thread A writes some data to the remote memory.

When A invokes the `memwrite()` system call, the RMem Client intercepts and handles it as follows. First, the RMem Client parses the target remote memory address to check if the they lie within the bounds of the remote address space. This address space is disjoint from the local one, has the size of the available remote DRAM, and is not handled nor managed by the compiler. In Nanvix, the Memory Allocator service provides extra facilities to user applications, so that they can allocate/deallocate memory regions in the remote memory. Due to space limitations, details about the implementation of this second service are out of the scope of this work, but can be retrieved from the source code that we made available. Either way, in the case in which the remote addresses lie within the remote memory address space, the system call then determines which RMem server is in charge of bookkeeping and performing access to the target data. It does so by extracting the w most significant bits of the address and using them to index the array of known RMem Servers. The number of RMem Servers in the system (2^w) grows proportionally with the amount of remote DRAM banks that are available. In the Kalray MPPA-256 lightweight manycore, system configurations with up to two RMem Servers are possible.

Once the RMem Client has determined the target RMem Server, it should find out the physical location of the server (i.e., in which cluster and core). To do so, the RMem Client relies on the Name Service, an OS daemon of the process management subsystem. This supporting service is implemented by the Name Client and Server, and it provides a naming functionality that enables user applications and system servers to link/unlink symbolic names to themselves, as well as a mechanism for resolving symbolic names into physical locations in the processor. The RMem Client invokes the name resolution interface exposed by the Name Service, and the request is intercepted by the Name Client. This latter client looks up into its cache for a valid entry that is linked with supplied name. If such entry exists, the physical location of the target RMem Server is directly retrieved from there and returned to the RMem Client. Otherwise, the Name Client queries the Name Server engine for this information, caches the response and returns it to the RMem Client facility.

With the physical location of the target RMem Server, the RMem Client proceeds with the data transfer in two steps, through the IPC Connectors. First, the RMem Client (i) sends the header of the remote write operation to the RMem Server using a mailbox, a communication primitive that is intended for small messages; (ii) setups the write operation; and (iii) and it waits for an

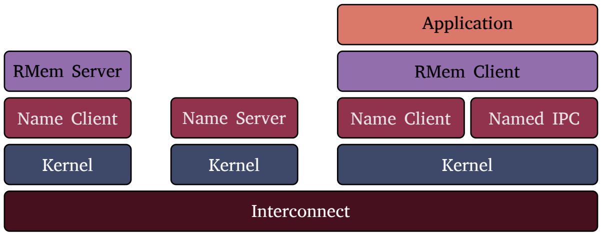


Fig. 2: RMem Service architectural overview.

acknowledgment from the RMem server to continue. The operation header contains meta-information about the request, such as the type of the operation, the remote memory address and the transfer size. When the RMem Server receives the operation request, it checks if the operation lies within the bounds of the remote memory that is addressable by the target RMem Server, and if the remote application has enough permissions to write to the target address range. If the operation passes both assertions, then the RMem Server sends a positive acknowledgment to the application and waits for the data; otherwise it returns an error code. As soon as the RMem Client receives a reply from the RMem Server, it resumes. Upon a negative acknowledgement, it aborts the execution of the thread A, else it sends the data through a portal, a communication primitive designed to dense data transfers.

We implemented a prototype of the RMem Service in Nanvix [22,23], a research OS that features a multikernel design and targets lightweight manycores. Our implementation is available at: https://github.com/nanvix/multikernel.

4 Evaluation Methodology

The performance of our RMem Service is quantified by the throughput that it sustains. Thus, to evaluate this performance, we relied on an in-house implementation of the microbenchmark kernel proposed in [14], which performs several accesses to RMem in blocks of fixed size and no computation. The kernel takes as input three parameters: (i) the number of blocks to read/write; (ii) the size of a block; and (iii) the number of peers that concurrently use the RMem Service.

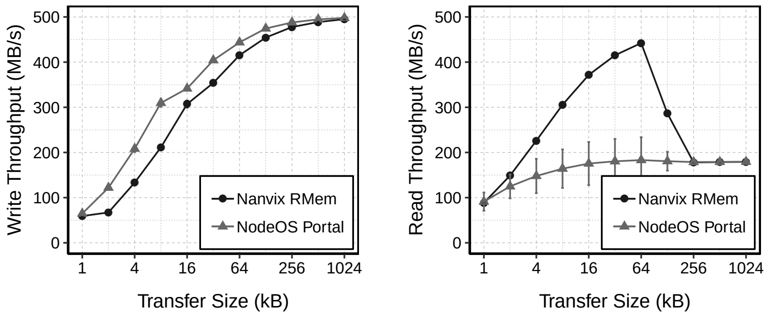
Overall, based on this synthetic kernel, we conducted two experiments, iteratively. In the first experiment, which we named *Transfer Size Scaling*, we fixed in 1, the number of peers that concurrently use the RMem Service; and we varied the block size of our service in a constant factor of 2, from 1 kB to 2^{10} kB. With this assessment, we aimed at identifying the block size that leads to the best performance. Once we identified this scenario, we then launched the second experiment, which we labeled *Peer Scaling*. In this latter experiment, we fixed the block size into the previous value that we noted; and we varied the number of peers in a constant factor of 2, from 1 to 16. Our goal was to analyze whether or not the performance of RMem was impacted when scaling up the stress on it.

In both experiments, we considered as our baseline the implementation of the same synthetic kernel using the native runtime system and libraries that are shipped with the Kalray MPPA-256 processor. The rationale for choosing such baseline lies on the fact that the actual implementation of our RMem Service prototype is based on the latter runtime system and libraries. Therefore, by taking this baseline, we may effectively observe the overheads (or improvements) when using a richer Application Programming Interface (API). Finally, it is important to note we carried out several experimental replications for each configuration to ensure 95% of confidence in our results. For each replica, the actual order in which individual runs were executed was randomly determined.

5 Experimental Results

Figure 3a and Figure 3b depict the write and read throughput on the remote memory, respectively, when varying the transfer size and one communication peer. When analyzing the write throughput to the remote memory, we spotted a three-phase performance behavior: (i) first the write throughput little increases for small block sizes, i.e., up to 2 kB; (ii) then it scales up linearly for medium-sized blocks, i.e., from 2 kB to 64 kB; and finally, (iii) it shows up little performance increases for large block sizes, i.e., larger than 64 kB. Indeed, we found out that the rationale for this behavior in the time results for the experiment itself. In the first phase, the overhead imposed by system libraries is greater than the network time. However, in the second phase, not only this scenario reverses, but also when the block size increases no important impact in the network time was observed. Finally, in the third phase, when increasing the block size, network time significantly lengthens, thus causing throughput to scale poorly. Overall, we observed that our service may achieve similar write throughput to NodeOS (default library in Kalray MPPA-256). We noted that 128 kB block sizes offer the best trade-off between the performance achieved by RMem and the overhead that it imposes. In this scenario, our service achieves about 91% of write throughput that is delivered by NodeOS.

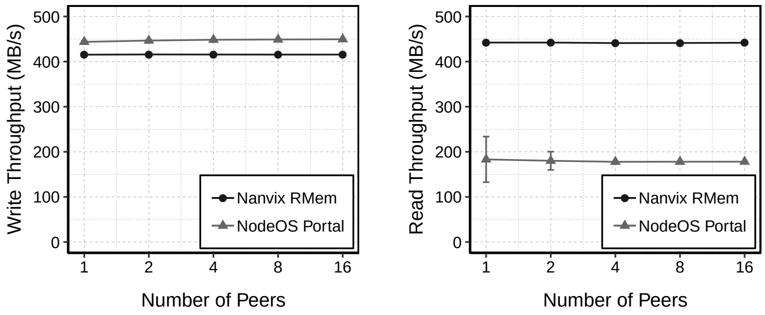
When studying the read throughput from the remote memory, we observed a different behavior. Performance significantly increases when using block sizes of up to 64 kB, and then it sharply decreases to the performance plateau of NodeOS. We found out that rationale for this behavior is two-fold. First, during the read operation itself, the RMem server and its remote peer partially overlap their execution in time, thanks to the two-step protocol that we employ. In this scheme, as soon as the remote peer sends the read request header to the RMem server, it may set up the read operation in its side. In contrast, when using the native interface in a naive way (i.e., single-step protocol), the remote peer blocks until the other remote is ready to send the data. Second, we observed that



(a) Remote write.

(b) Remote read.

Fig. 3: Experimental results for *Transfer Size Scaling* experiment.



(a) Remote write.

(b) Remote read.

Fig. 4: Experimental results for *Peer Scaling* experiment.

the actual reduction in the synchronization time due to the two-step protocol accounts for an important amount of time in the transfer for block sizes of up to 64 kB. As a consequence, the read throughput is significantly increased. However, for larger block sizes, we observed the network time dominated the total transfer time, thus causing read throughput to drop sharply. In comparison with NodeOS, we observed that RMem may achieve up to 2.4× higher read throughput.

After analyzing write and read throughputs in the remote memory, we observed that our service achieves the best performance for blocks of 64 kB. Recall that this result was observed when a single remote peer is communicating with the RMem Server. Therefore, in order to conclusively determine whether or not this performance is impacted by the number of peers that concurrently write/read to/from the RMem, we conducted a *Peer Scaling* experiment in the peak-performance point that we identified. That is, we benchmarked the throughput of remote writes and reads when fixing the block size of the RMem Service in 64 kB and varying the number of peers. Figure 4a and Figure 4b outline the write and read throughput on the remote memory in this second experiment, respectively. Overall, we observed that for both operations, throughputs do not significantly change when varying the number of remote peers. Putting it differently, nor the RMem server nor the underlying on-chip interconnect impose significant overheads in 64 kB remote reads and remote writes, thereby enabling a constant performance to be sustained regardless the number of remote peers that are involved. We also noted that the performance gap between RMem NodeOS remained nearly constant, for both operations, at the same levels that we observed before, in the *Transfer Size Scaling* experiment. This outcome further supports the conclusions that we drew previously.

As a final remark, it is important to note that regardless the size of the blocks and number of peers employed in our experiments, the theoretical 1.6 GB/s throughput for the D-NoC link (i.e., the hardware peak performance) was not achieved by either facilities, which are the NodeOS runtime and the Nanvix RMem Service. Indeed, after investigating this behavior further, we found out that the native development libraries, which are available in the experimental platform and used by both facilities, are not optimized. Therefore, for NoC-intensive workloads peak performance cannot be achieved.

6 Related Work

The distributed memory configuration and tight memory footprint constraints of low-power manycores introduce several challenges that still should be addressed, such as one-sided inter-process communication support and transparent data accessing and tiling. Aiming at an utterly transparent solution, some research efforts looked for hardware level alternatives to implement a coherent global memory system on top of the bare metal distributed configuration. For instance, a fully-programmable processor to be integrated into the network interface of a NoC was proposed [24]. That interface, called Typhoon, transparently handles low-level communication and exposes to user-level a cache coherent shared memory system that may be fine-tuned to cope with the application characteristics. However, the hardware-level implementation introduces additional complexity, and thus increases power consumption and chip area utilization. Applications that do not much benefit from this feature end up wasting resources [1,2].

Orthogonal to the idea of solving memory-related problems on low-power manycores at the hardware level, other research efforts sought for providing a coherent memory system in software [21]. Supporters of this alternative approach argued that by providing a shared memory abstraction on software, coherency traffic and power-consumption might be decreased, as well as the memory system may be dynamically reconfigured to better meet the characteristics of applications [32]. The main idea consisted in providing a transparent shared memory abstraction at page granularity in the OS [4,18]. For instance, in Shasta [27] remote loads and stores were intercepted and transparently handled. However, the downside arises when memory-bound applications come into play, with a high number of remote fetches incurring on performance degradation.

To overcome this problem, alternative software solutions tackled memory-related challenges of distributed architectures at the runtime system level employing high-performance parallel programming models. Among the alternatives, PGAS has stood out as a promising and ever-growing solution [9]. It consists of a runtime system that provides a globally shared address space abstraction to processes of a distributed application, and it exposes one-sided communication primitives to operations on this address space. Indeed, due to its success, PGAS is already available on the Intel Single-Cloud Computer and Adapteva Epiphany low-power manycores, and it proved to achieve reasonable performance and energy efficiency [13,26]. Nevertheless, despite its advantages, PGAS is not a fully-featured solution, once it lies on the runtime system, and thus it may not address the communication of different applications, nor provide means for an application to operate on large amounts of data.

Our proposal (RMem) differs from the previously discussed solutions in several points. In contrast to hardware solutions, the application may or may not use the service itself, but applications that do not choose RMem, do not have any extra cost. When compared to existing solutions in the OS level, the RMem stands out as a different abstraction, which in turn may be used to build higher-level features, such as process migration. Finally, different from runtime system level solutions, our service enables a fully featured alternative, in which processes from different applications may communicate; data can be transparently accessed and migrated, and data may be securely shared.

7 Conclusions

Lightweight manycores deliver performance and scalability while featuring low-power requirements and high energy efficiency. Unfortunately however, due to their architectural intricacies and programmability challenges that follow, these processors face several barriers that keep them away from mass adoption.

Aiming at bridging the programmability gap in lightweight manycores, in this work, we focus on addressing open challenges that arise from their constrained memory subsystems, such as the presence of multiple address spaces and limited on-chip memory. To cope with transparent data access in this scenario, we introduce a new facility at the OS level, which we named RMem. Our solution consists in an OS service that provides a shared memory abstraction over multiple address spaces and exposes system calls that enable one-sided communication on top of this abstraction. By relying on the RMem Service, (i) processes may manipulate large amounts of data, regardless the limited on-chip memory; (ii) different applications may effectively communicate and share data; and (iii) the OS itself is able to transparently migrate data around, to better exploit locality and thus mitigate NoC congestion. We implemented a prototype of our service in the Nanvix research OS, and we deployed it in the Kalray MPPA-256 processor. Furthermore, we evaluated the performance of RMem using a synthetic kernel and contrasted its performance with the communication primitives that are available in the libraries which are shipped with the experimental platform. Our results unveiled that while exposing a simpler and easier-to-program API, RMem may deliver about 91% of the write performance that is achieved by the baseline primitives, and 2.4× better read performance. Overall, these results strongly encourage us to a high-performance implementation of our prototype.

In future work, we intend to investigate how to integrate data caching, prefetching and tiling into RMem. Also, we intend to rely on our service to introduce high-level OS abstractions, such as shared memory segments, core multiplexing and process migration, to lightweight manycores that feature a constrained memory subsystem.

Acknowledgements

We thank CNRS, CNPq, FAPEMIG and FAPESC 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. Alvarez, L., Vilanova, L., Gonzalez, M., Martorell, X., Navarro, N., Ayguade, E.: Hardware-Software Coherence Protocol for the Coexistence of Caches and Local Memories. IEEE Transactions on Computers (TC) **64**(1), 152–165 (Oct 2013). https://doi.org/10.1109/TC.2013.194, http://ieeexplore.ieee.org/document/6616543/
2. Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M., Marwedel, P.: Scratchpad Memory: A Design Alternative for Cache On-Chip Memory in Embedded Systems. In: 2002 IEEE 10th International Symposium on Hardware/Software Codesign. pp. 73–78. CODES '02, IEEE, Estes Park, Colorado, USA, (May 2002). https://doi.org/10.1145/774789.774805, https://ieeexplore.ieee.org/document/1003604
3. Barbalace, A., Sadini, M., Ansary, S., Jelesnianski, C., Ravichandran, A., Kendir, C., Murray, A., Ravindran, B.: Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms. In: Proceedings of the 10th European Conference on Computer Systems. pp. 1–16. EuroSys '15, ACM, Bordeaux, France (Apr 2015). https://doi.org/10.1145/2741948.2741962, http://dl.acm.org/citation.cfm?doid=2741948.2741962
4. Bathen, L.A., Dutt, N.: Software Controlled Memories for Scalable Many-Core Architectures. In: 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. pp. 1–10. RTCSA '12, IEEE, Seoul, South Korea (Sep 2012). https://doi.org/10.1109/RTCSA.2012.60, http://ieeexplore.ieee.org/document/6301551/
5. Baumann, A., Barham, P., Dagand, P.E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The Multikernel: A New OS Architecture for Scalable Multicore Systems. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles. pp. 29–44. SOSP '09, ACM, Big Sky, Montana, USA (Oct 2009). https://doi.org/10.1145/1629575.1629579, https://dl.acm.org/citation.cfm?doid=1629575.1629579
6. Berezecki, M., Frachtenberg, E., Paleczny, M., Steele, K.: Many-Core Key-Value Store. In: 2011 International Green Computing Conference and Workshops. pp. 1–8. IGCC '11, IEEE, Orlando, Florida, USA (Sep 2011). https://doi.org/10.1109/IGCC.2011.6008565, https://ieeexplore.ieee.org/document/6008565
7. Borkar, S., Shekhar: Thousand Core Chips. In: Proceedings of the 44th annual Design Automation Conference. pp. 746–749. DAC '07, ACM, San Diego, California, USA (Jun 2007). https://doi.org/10.1145/1278480.1278667, http://portal.acm.org/citation.cfm?doid=1278480.1278667
8. Christgau, S., Schnor, B.: Exploring One-Sided Communication and Synchronization on a Non-Cache-Coherent Many-Core Architecture. Concurrency and Computation: Practice and Experience (CCPE) **29**(15), e4113 (Mar 2017). https://doi.org/10.1002/cpe.4113, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4113
9. De Wael, M., Marr, S., De Fraine, B., Van Cutsem, T., De Meuter, W.: Partitioned Global Address Space Languages. ACM Computing Surveys (CSUR) **47**(4), 1–27 (May 2015). https://doi.org/10.1145/2716320, http://dl.acm.org/citation.cfm?doid=2775083.2716320
10. de Dinechin, B.D., de Massas, P.G., Lager, G., Léger, C., Orgogozo, B., Reybert, J., Strudel, T.: A Distributed Run-Time Environment for the Kalray

MPPA-256 Integrated Manycore Processor. In: Procedia Computer Science. ICCS '13, vol. 18, pp. 1654–1663. Elsevier, Barcelona, Spain (Jun 2013). https://doi.org/10.1016/j.procs.2013.05.333, http://linkinghub.elsevier.com/retrieve/pii/S1877050913004766

11. Francesquini, E., Castro, M., Penna, P.H., Dupros, F., Freitas, H., Navaux, P., 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) **76**(C), 32–48 (Feb 2015). https://doi.org/10.1016/j.jpdc.2014.11.002, http://linkinghub.elsevier.com/retrieve/pii/S0743731514002093
12. Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., Yang, G.: The Sunway TaihuLight Supercomputer: System and Applications. Science China Information Sciences **59**(7), 1–16 (Jun 2016). https://doi.org/10.1007/s11432-016-5588-7, http://link.springer.com/10.1007/s11432-016-5588-7
13. Gamell, M., Rodero, I., Parashar, M., 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. pp. 235–246. HPDC '12, ACM, Delft, The Netherlands (Jun 2012). https://doi.org/10.1145/2287076.2287113, http://dl.acm.org/citation.cfm?doid=2287076.2287113
14. Hascoët, J., de Dinechin, B., de Massas, P., Ho, M.Q.: Asynchronous One-Sided Communications and Synchronizations for a Clustered Manycore Processor. In: Proceedings of the 15th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia. pp. 51–60. ESTIMedia '17, ACM, Seoul, Republic of Korea (Oct 2017). https://doi.org/10.1145/3139315.3139318, https://dl.acm.org/citation.cfm?id=3139318
15. Howard, J., Dighe, S., Vangal, S., Ruhl, G., Borkar, N., Jain, S., Erraguntla, V., Konow, M., Riepen, M., Gries, M., Droege, G., Lund-Larsen, T., Steibl, S., Borkar, S., De, V., Van Der Wijngaart, R.: A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling. IEEE Journal of Solid-State Circuits (JSSC) **46**(1), 173–183 (Jan 2011). https://doi.org/10.1109/JSSC.2010.2079450, http://ieeexplore.ieee.org/document/5621843/
16. Kelly, B., Gardner, W., 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. pp. 62–65. MES '13, ACM, Tel-Aviv, Israel (Jun 2013). https://doi.org/10.1145/2489068.2491624, http://dl.acm.org/citation.cfm?doid=2489068.2491624
17. Kluge, F., Gerdes, M., 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. pp. 238–245. ISORC '14, IEEE, Reno, Nevada, USA (Jun 2014). https://doi.org/10.1109/ISORC.2014.30, http://ieeexplore.ieee.org/document/6899155/
18. Lankes, S., Reble, P., Sinnen, O., Clauss, C.: Revisiting Shared Virtual Memory Systems for Non-Coherent Memory-Coupled Cores. In: Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores. pp. 45–54. PMAM '12, ACM, New Orleans, Louisiana, USA

(2012). https://doi.org/10.1145/2141702.2141708, http://dl.acm.org/citation.cfm?doid=2141702.2141708

19. Nightingale, E., Hodson, O., McIlroy, R., Hawblitzel, C., Hunt, G.: Helios: Heterogeneous Multiprocessing with Satellite Kernels. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 221–234. SOSP '09, ACM, Big Sky, Montana, USA (Oct 2009). https://doi.org/10.1145/1629575.1629597, http://doi.acm.org/10.1145/1629575.1629597
20. Olofsson, A., Nordstrom, T., Ul-Abdin, Z.: Kickstarting High-Performance Energy-Efficient Manycore Architectures with Epiphany. In: 2014 48th Asilomar Conference on Signals, Systems and Computers. pp. 1719–1726. ACSSC '14, IEEE, Pacific Grove, California, USA (Nov 2014). https://doi.org/10.1109/ACSSC.2014.7094761, http://ieeexplore.ieee.org/document/7094761/
21. Park, J., Jang, C., Lee, J.: A Software-Managed Coherent Memory Architecture for Manycores. In: 2011 International Conference on Parallel Architectures and Compilation Techniques. pp. 213–213. PACT '11, IEEE, Galveston, Texas, USA (Dec 2011). https://doi.org/10.1109/PACT.2011.46, http://ieeexplore.ieee.org/document/6113823/
22. Penna, P.H.: The Nanvix Operating System. Tech. rep., HAL, Belo Horizonte (Mar 2017). https://doi.org/10.13140/RG.2.2.20036.22407, https://hal.archives-ouvertes.fr/hal-01495741
23. Penna, P.H., Castro, M.B., Freitas, H., Méhaut, J.F., Caram, J.: Using the Nanvix Operating System in Undergraduate Operating System Courses. In: 2017 VII Brazilian Symposium on Computing Systems Engineering. pp. 193–198. SBESC '17, IEEE, Curitiba, Brazil (Nov 2017). https://doi.org/10.1109/SBESC.2017.33, http://ieeexplore.ieee.org/document/8116579/
24. Reinhardt, S., Larus, J., Wood, D.: Tempest and Typhoon: User-level Shared Memory. In: Proceedings of the 21st Annual International Symposium on Computer Architecture. pp. 325–336. ISCA '94, IEEE, Chicago, Illinois, USA (Apr 1994). https://doi.org/10.1145/191995.192062, https://dl.acm.org/citation.cfm?id=192062
25. Rhoden, B., Klues, K., Zhu, D., Brewer, E.: Improving Per-Node Efficiency in the Datacenter with New OS Abstractions. In: Proceedings of the 2nd ACM Symposium on Cloud Computing. pp. 1–8. SoCC '11, ACM, Cascais, Portugal (Oct 2011). https://doi.org/10.1145/2038916.2038941, https://dl.acm.org/citation.cfm?id=2038941
26. Ross, J., Richie, D.: Implementing OpenSHMEM for the Adapteva Epiphany RISC Array Processor. In: Procedia Computer Science. ICCS '16, vol. 80, pp. 2353–2356. Elsevier, San Diego, California, USA (Jun 2016). https://doi.org/10.1016/J.PROCS.2016.05.439, http://www.sciencedirect.com/science/article/pii/S1877050916309206
27. Scales, D., Gharachorloo, K., Thekkath, C.: Shasta: A Low Overhead, Software-only Approach for Supporting Fine-grain Shared Memory. In: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS '96, vol. 30, pp. 174–185. ACM, Cambridge, Massachusetts, USA (Oct 1996). https://doi.org/10.1145/248208.237179, https://dl.acm.org/citation.cfm?id=248208.237179
28. Serres, O., Anbar, A., Merchant, S., El-Ghazawi, T.: Experiences with UPC on TILE-64 Processor. In: Aerospace Conference. pp. 1–9. AERO '11, IEEE, Big Sky,

Montana, USA (Mar 2011). https://doi.org/10.1109/AERO.2011.5747452, http://ieeexplore.ieee.org/document/5747452/

29. Souza, M., Penna, P.H., Queiroz, M., Pereira, A., Góes, L.F., Freitas, H., Castro, M., Navaux, P., Méhaut, J.F.: 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), 1–18 (Jun 2016). https://doi.org/10.1002/cpe.3892, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3892
30. West, J., Rico, A., Mantovani, F., Ruiz, D., Vilarrubi, J.O., Gomez, C., Backes, L., Nieto, D., Servat, H., Martorell, X., Labarta, J., Ayguade, E., Adeniyi-Jones, C., Derradji, S., Gloaguen, H., Lanucara, P., Sanna, N., Méhaut, J.F., Pouget, K., Videau, B., Boyer, E., Allalen, M., Auweter, A., Brayford, D., Tafani, D., Weinberg, V., Brömmel, D., Halver, R., Meinke, J.H., Beivide, R., Benito, M., Vallejo, E., Valero, M., Ramirez, A.: The Mont-Blanc Prototype: an Alternative Approach for HPC Systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. SC '16, IEEE, Salt Lake City, Utah, USA (Nov 2016). https://doi.org/10.1109/SC.2016.37, https://ieeexplore.ieee.org/document/7877116
31. Zheng, F., Li, H.L., Lv, H., Guo, F., Xu, X.H., Xie, X.H.: Cooperative Computing Techniques for a Deeply Fused and Heterogeneous Many-Core Processor Architecture. Journal of Computer Science and Technology (JCST) **30**(1), 145–162 (Jan 2015). https://doi.org/10.1007/s11390-015-1510-9, https://link.springer.com/article/10.1007%2Fs11390-015-1510-9
32. Zhou, X., Chen, H., Luo, S., Gao, Y., Yan, S., Liu, W., Lewis, B., Saha, B.: A Case for Software Managed Coherence in Many-core Processors. In: Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism. pp. 1–6. HotPar '10, USENIX, Berkeley, California, USA (Jun 2010), http://static.usenix.org/event/hotpar10/tech/full_papers/Zhou.pdf