OS-Level Task-Based Mechanism for Lightweight Manycore Processors

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

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

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

***Abstract*—Lightweight manycore processors stand out for their high level of parallelism and energy efficiency. However, its architectural characteristics introduce several challenges in software development. One of the central difficulties is the limited amount of local memory that the applications have available to work. This limitation needs to be considered by software solutions for lightweight manycores. In this context, this work proposes the definition of a task mechanism within an asymmetric microkernel designed for lightweight manycores. This mechanism makes it possible to define multiple execution streams at a low cost to the memory system through the implementation of a limited number of special threads. The results showed that the proposed mechanism achieved similar performance to that of a dedicated thread to each task.**

***Index Terms*—HAL, Distributed Operating System, Lightweight Manycore, Kalray MPPA-256**

I. Introduction

The class of the *Lightweight Manycore Processors* stands out for its high level of parallelism with low energy consumption [1]. Applications designed to run on these processors have available hundreds of cores distributed on a single chip. To address the high scalability and provide energy efficiency, lightweight manycores exhibit architectural characteristics that differ from other manycores. Specifically, they integrate: (i) thousands of low-power cores with Multiple Instruction Multiple Data (MIMD) capability [2]; (ii) distributed memory architecture and small local memories shared by tightly-coupled groups of cores (aka *clusters*) [3]; (iii) reliable and fast Network-on-Chips (NoCs) for message-passing [3]; and (iv) heterogeneous processing capabilities [4]. Some industry-successful examples of lightweight manycore processors are the Kalray MPPA-256 [5] and the Adapteva Epiphany [6].

These characteristics introduce several challenges from low- to high-level software development. For instance, (i) hybrid programming between shared-memory and message-passing models due to the nature of the lightweight manycores [7]; (ii) lack of hardware support for cache coherence [1]; (iii) distributed and restrictive memory system [8]; and (iv) programming of heterogeneous components [9]. In this context, different proposed solutions, at different abstraction levels, sought to alleviate the difficulties, from runtimes specialized in a programming paradigm [10], [11] to more robust and generic programming environments [12].

Among all the studies, one of the principal difficulties is how to deal with the reduced amount of local memory within a cluster. It is worth mentioning that this memory is often not exclusive to the user. The local memory needs to store code and data structures of the application and its dependencies, e.g. Operating System (OS) and/or libraries. One element on an OS that consumes a considerable amount of memory is the support for multiple execution streams. For instance, allocating memory pages to store the context and stack for executing a routine. In this context, the present work proposes an *OS-level Task-based Mechanism* to define a generic execution unit. Similar to a pool of threads, a system thread waits to receive these units to run. With this mechanism, we seek to achieve the following objectives and contributions:

- Design a kernel-level task mechanism for internal OS and client application to use;
- Decrease the memory allocation of the thread system by reusing the execution stack of a single thread;
- Improve the use of idle cores;
- Explore the locality of data in a single core cache;
- Enable asynchronous and/or periodic operations, e.g. asynchronous sending in lightweight manycores that do not feature a Direct Memory Access (DMA);
- Facilitate the modeling of internal kernel functionality.

This work is part of the development of *Nanvix*, a distributed operating system designed to address the characteristics of lightweight manycores. The scope of the work is limited to the improvement of an asymmetric microkernel that manages the local resources of a cluster.

The remainder of this work is organized as follows. In Section II, we discuss related work. In Section III we cover the problem definition. In Section IV, we present our proposal. In Section V, we detail the experimental environment. In Section VI, we discuss our experimental results. In Section II we discuss related works. In Section VII, we draw our conclusions.

II. Related Work

Several proposed solutions aim to take advantage of the high level of parallelism and low energy consumption of the lightweight manycores without introducing an unwanted overhead to the system. Specific solutions, which model the entire execution environment around a type of programming paradigm, seek to eliminate existing competition problems in multiprocessing and preemptive systems. For instance, task-based [10] or coroutine [11] execution runtimes exploit the task parallelism that works cooperatively. These runtimes support flexibility and fine-grained job control. It is also possible to map parallel programming patterns, such as OpenMP

directly into task/coroutine sets. Cooperation between the execution units is an essential benefit of these systems, where the context change is light and does not occur involuntarily, making the execution more predictive. However, these runtimes limit the class of applications that can be ported to lightweight manycores.

Other solutions explore to develop standards and mechanisms at the kernel-level [12] modeled specifically for lightweight manycores without losing compatibility with other systems. However, some OS services may come to require an excessive footprint of memory, forcing the client application to work with a limited amount of memory. We believe that many of these internal OS services can be modeled from a perspective of requested, periodic, and/or asynchronous tasks. This solution fits especially in asymmetric microkernels using the idle time of the master core and the sequential form that the syscalls are handled.

III. Problem Definition

The internal memory management of a cluster influences all abstraction levels in the context of lightweight manycores. For instance, OSs need to be small and lightweight to make the most memory available to the application. On the other hand, concurrent and parallel applications need to manually manage the consistency of the manipulated data.

To alleviate the cache coherence problems existing in these architectures, Penna ETAL proposed an asymmetric microkernel to concentrate the management and manipulation of the OS structures in a single core. Figure X illustrates the execution of a system call within Nanvix. When a slave core wants to perform a privileged operation, it sends a request to the master core, which in turn, performs and returns the result to the requester. The problems explored in this work are contained within the scope of this level of the Nanvix.

In addition to the difficulties imposed by the memory system, our proposal seeks to address some of the disadvantages of the microkernel and the communication system. The following descriptions summarize the main problems that our solution aims to address:

Memory Utilization For each new thread, the memory system must reserve two memory spaces, for user and kernel stacks. Generally, each stack is the size of a memory page. Therefore, rapidly increasing the amount of memory needed to create new threads in the system.

Data Locality Because of the lack of hardware support for cache coherence, competition for a shared memory region is costly. For instance, a thread must invalidate their memory cache to access the local memory bank before entering a critical region. When leaving it, the thread must force the written of the modified data to keep the coherence.

Core Utilization Due to the need to have a core reserved for the master thread, the asymmetric microkernel loses processing power when it leaves the master core idle between syscall requests.

Asynchronous Operations For power consumption reduction, lightweight manycores may not feature a dedicated DMA to perform asynchronous communication. Thus, a specific thread must to manually send the data through the NoC. All system calls within the microkernel are also blocking, even though a thread can perform independent operations in the meantime.

Periodic Operations To allow a cluster to be monitored or receive external commands, a specific thread must request communication checks constantly. The existence of these operations is essential to develop more complex services, e.g. process management and migration, invalidation of shared and distributed memory, and execution of remote procedures.

IV. OS-Level Task-Based Mechanism for Lightweight Manycore Processors

The proposed *task* abstraction can be seen as a special case of coroutines. It encapsulates a subroutine that can be performed regardless of who created them. However, unlike coroutines, the proposed task definition is disconnected from the need to have a dedicated thread per task. At this point, we introduce a special thread, named *dispatcher*, which is a generic task executor. Following the producer-consumer model, the dispatcher consumes tasks from a global task queue, where requesters atomically insert tasks to be performed.

A *task* is a standardized structure that holds semantic and control information. On the one hand, the semantic variables store the function to be performed, its arguments, and a slot to store the return value. On the other hand, control variables, referring to the semantics of the dispatcher, are composed of the state of the task, a list of dependent tasks, the number of active parent tasks, and a synchronization control with the requester. This allows the creation of dependency graphs that introduce more sophisticated managements and asynchronous behaviors. These dependencies guarantee that a task will only be ready to execute when all the parent tasks have completed their executions.

Pseudocode 1 summarizes the behavior of a dispatcher. Blocked at a semaphore, the dispatcher will wait for new task requests to appear. When consuming a task, the dispatcher

```
while Not shutdown do
    Waits for a task;
    Execute task function;
    switch Task return do
        case TASK_RET_SUCCESS do
            Complete the task and schedule children;
        end
        case TASK_RET_AGAIN do
            Reschedule the task;
        end
        case TASK_RET_STOP do
            Insert the task into a waiting queue;
        end
        case TASK_RET_ERROR do
            Propagate the error and release all tasks;
        end
    end
end
```

Algorithm 1: How to write algorithms

updates its state and begins executing the task scope. At this point, it is possible to notice the reuse of a stack of the dispatcher for numerous independent tasks. Specifically, when it returns from the task scope, it will be in its initial state and able to start another task. This algorithm supports the spawn of more than one dispatcher that will cooperate to handle the tasks. It will increase the parallelism and decrease the waiting time but required more memory pages. The task can also signal the completion status to the dispatcher between three values. Specifically:

TASK_RET_SUCCESS the dispatcher will complete the task, signaling its completion to child tasks and releasing the requester.

TASK_RET_AGAIN The task returned with an error but is recoverable and the task will be rescheduled.

TASK_RET_STOP The task returned with an error, but despite being recoverable, it needs to wait for another operation to resume.

TASK_RET_ERROR The task returned an error and is unrecoverable, the dispatcher will signal the error to the requester and propagate the error to all child tasks.

The proposed task-based mechanism at the OS-level addresses the problems described in the previous section as follows:

Memory Utilization The definition of multiple execution streams in isolated tasks, or in a dependency graph, reduces the use of memory pages for threads based on the number of existing dispatchers. If the kernel and the application are able to isolate simple behaviors without the need to create a dedicated thread, more memory will be available to store useful data.

Data Locality Configuring a dispatcher to always be in only the same core, it can perform tasks that share the same data structures by exploring the location of the data cache.

Core Utilization This a significant point to define the dispatcher at the OS-level that makes it possible to move it to the master core. Therefore, it will share the execution time with the master thread. At one extreme, we could even define the master thread itself has a dedicated dispatcher.

Asynchronous Operations Modeling tasks to move the responsibility to send the data to the dispatcher, introduce this notion of asynchronous send/read. The same idea applies to syscalls, where non-critical operations can be performed asynchronously.

Periodic Operations Tasks can be modeled to have a period to which they are unblocked and operate. In this way, we eliminate the need to have a dedicated thread that is waiting for some condition to carry out specific operations. For example, reading a communication from the NoC that will create a new task to execute a remote procedure.

Figure 1 illustrates the interactions within the kernel after implementing the task mechanism. Periodic tasks created by the kernel allowed for a richer intra- and inter-cluster interaction. As the master thread responsible for handling one request at a time, moving that responsibility to the dispatcher will not introduce a bottleneck in the system beyond what already exists. Finally, tasks requested by ordinary users can

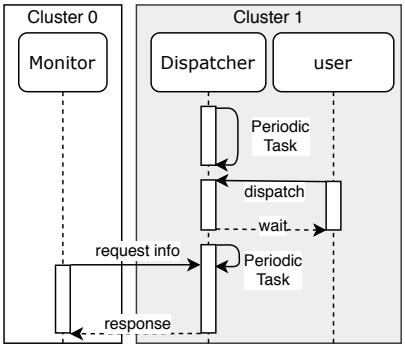

Fig. 1. Clusters interations using tasks.

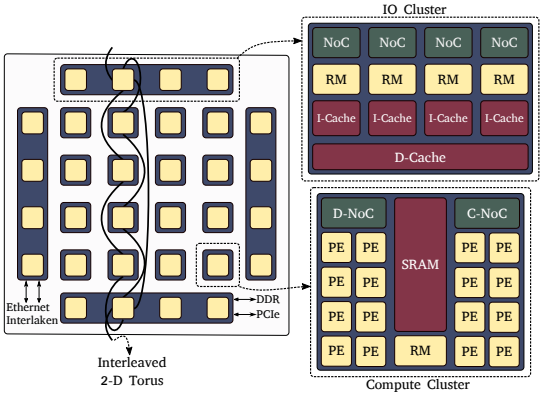

Fig. 2. Kalray MPPA-256 Architecture Overview.

and should be interleaved, which leads to the need for finer-grain management than a simple insertion in a queue.

V. Experimental Environment

Kalray MPPA-256 processor is one of the architectures supported by Nanvix and will be used as a case study in this work. As Figure 2 pictures, Kalray MPPA-256 integrates 288 general-purpose cores, grouped into 16 Compute Clusters, intended for useful computing, and 4 I/O Clusters responsible for communicating with peripherals. On the one hand, each Compute Cluster integrates 16 Processing Elements (PEs), 1 Resource Manager (RM), 2 MB of local SRAM, two NoC interfaces, and does not have hardware support for cache coherence. On the other hand, each I/O Cluster features 4RMs, 4 MB of SRAM, and 8 NoC interfaces. Two of these I/O Clusters have access to a 4 GB of DRAM, and the other two are connected to Ethernet controllers. Two distinct 2-D Torus NoCs handle inter-cluster communication. Specifically, Control NoC (C-NoC) allows the exchanging of small control messages, and Data NoC (D-NoC) supports the transferring of arbitrary amounts of data.

VI. Results

To assess the costs of creating and waiting for a task, we created a synthetic benchmark that write each byte of 16 memory page, e.g., 64 KB. To guarantee 95% confidence, 50 replications were performed, the first 10 being discarded to ignore the warm-up period.

TABLE I
BENCHMARK PARAMETERS FOR EXPERIMENTS.

Benchmark	#Tasks	#Threads	Memory
Single Dispatcher	1 to 29	1	8 KB
Multiple Dispatchers	1 to 29	14	112 KB
Threads	1 to 29	1 to 29	[8, 232] KB

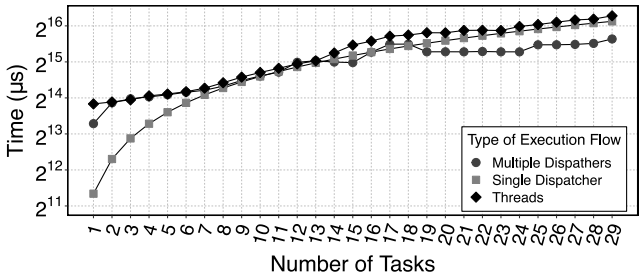

Fig. 3. Runtime Results.

Table I shows the application of this benchmark replicated in three different scenarios: (i) A single dispatcher handle all tasks sequentially; (ii) Each free slave core contains a different dispatcher serving tasks in parallel, resulting in 14 dispatchers; (iii) One thread for each task, ranging from 1 to 29 threads. The number of tasks was limited to the possible number of simultaneous threads within the microkernel not replicate the behavior of tasks in user threads. The amount of tasks and threads involved defines into the amount of memory required to execute each scenario, where each thread uses 8 KB for the stacks. Based on this, we can discern that much no extra effort, the user can use task abstraction with dispatchers and keep the amount of memory used controlled.

Figure 3 shows the execution time in μ for each scenario. The times collected are made up of the dispatch/create and wait/join period of a task/thread. Comparing the execution of a single task, the use of only one dispatcher showed superior performance to the other scenarios because there is no competition for the consumption of tasks, exploring the location of the data, and do not require the cost of the creation of a new thread. Between 7 to 13 tasks, all scenarios behaved similarly performance because the overhead of thread creation and concurrency were balanced by the workload of the single dispatcher. After 14 tasks, it is possible to notice an increase in the scenario of threads because from that moment, there are more threads than cores and they start to compete for execution time. In contrast, the scenario with 14 dispatchers showed a lower execution time because it needs less time to dispatch and consume a task than to create a thread. The single dispatcher scenario has steadily increased because the tasks are of regular size. Considering that the tasks are of regular size, share memory pages, and are executed in dedicated cores, the proposed mechanism showed performance similar to the scenario that uses specific threads per task.

VII. CONCLUSIONS

Lightweight manycore processors are promising candidates for modern computer systems to reach the Exascale era. However, its distinctive architectural features require efforts to redesign and propose new solutions that extract maximum performance from these processors. The purpose of this work is to explore the reduction in the use of kernel memory and provide a service to improve and assist concurrent and/or asynchronous services within the OS. The results showed that the proposed mechanism achieved similar performance to that of a dedicated thread to each task. As future work, we intend to model the entire Nanvix communication system using tasks and apply well-known task ordering scheduling algorithms on the proposed mechanism.

REFERENCES

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

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

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

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

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

[6] A. Olofsson, "Epiphany-v: A 1024 processor 64-bit risc system-on-chip," *ArXiv*, vol. 1610.01832, pp. 1–15, 2016. [Online]. Available: https://arxiv.org/abs/1610.01832

[7] B. Kelly, W. Gardner, and S. Kyo, "AutoPilot: Message Passing Parallel Programming for a Cache Incoherent Embedded Manycore Processor," in *Proceedings of the 1st International Workshop on Many-core Embedded Systems*, ser. MES '13. Tel-Aviv, Israel: ACM, Jun. 2013, pp. 62–65. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2489068.2491624

[8] M. Castro, E. Francesquini, F. Dupros, H. Aochi, P. O. Navaux, and J.-F. Méhaut, "Seismic wave propagation simulations on low-power and performance-centric manycores," *Parallel Computing*, vol. 54, pp. 108–120, 2016.

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

[10] Y. Zhou, J. Wu, Y. Zhang, Y. Yin, and S. Li, "Design and implementation of coroutine scheduling system on sw26010," in *Proceedings of the 2020 5th International Conference on Big Data and Computing*, ser. ICBDC 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 114–120. [Online]. Available: https://doi.org/10.1145/3404687.3404700

[11] D. Cesarini, A. Marongiu, and L. Benini, "An optimized task-based runtime system for resource-constrained parallel accelerators," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, pp. 1261–1266.

[12] P. H. Penna, J. Souto, D. F. Lima, M. Castro, F. Broquedis, H. H. Freitas, and J.-F. Mehaut, "On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores," in *SBESC 2019 - IX Brazilian Symposium on Computing Systems Engineering*, Natal, Brazil, Nov. 2019, pp. 1–31. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02297637