# P-SOCRATES: A Parallel Software Framework for Time-Critical Many-core Systems

Vincent Nélis[1], Patrick Meumeu Yomsi[1], Luís Miguel Pinho[1], Eduardo Quiñones[2], Marko Bertogna[3], Andrea Marongiu[4], Paolo Gai[5], and Claudio Scordino[5]

[1] CISTER/INESC-TEC Research Center, Porto, Portugal
[2] Barcelona Supercomputing Center, Spain
[3] University of Modena, Italy
[4] IIS - ETH Zürich, Switzerland
[5] Evidence Srl

**Abstract.** Current embedded systems are following a general trend: they constantly demand for more computational performance to process large amounts of data from multiple data sources, some of them with guaranteed processing response times. From the hardware standpoint, multi-core and many-core processors are, as in the general domain, good candidates to improve the overall performance of these systems. From the software point of view, parallel computing is no longer a niche in the high performance computing (HPC) field, but an essential ingredient in all domains of computer science. However, it is not straightforward how event-based embedded applications can be structured in order to take advantage of parallelization to achieve higher performance and energy-efficient computing. The P-SOCRATES project envisions the necessity to bring together next-generation many-core accelerators from the embedded computing domain with the programming models and techniques from the HPC computing domain, supporting this with real-time methodologies to provide timing predictability. This paper gives an overview of the framework proposed by the project to achieve these objectives.

## 1 Introduction

Traditionally, High Performance Computing (HPC) has been the focus of specialized industries and specific groups within academia as it demands analytics and simulations applications that require large amounts of data to be processed. Similarly, researchers and industry in the embedded computing (EC) domain have focused mainly on specific systems with specialized and fixed set of functionalities for which timing requirements prevailed over performance requirements. Today, both the HPC and EC domains are broadening their initial focus to other application areas due to the ever-increasing availability of more powerful processing platforms, but therefore they need affordable and scalable software solutions [7, 17].

The need for energy-efficiency (in the HPC domain) and flexibility (in the embedded computing domain), that come along with Moore's law greedy demand for performance and the advancements in the semiconductor technology, have progressively paved the way for the introduction of many-core systems — i.e., multi-core chips containing a high number of cores (tens to hundreds) — in both domains.

Today, many-core computing fabrics are being integrated together with general purpose multi-core processors to provide a heterogeneous architectural harness that eases the integration of previously hard-wired accelerators into more flexible software solutions. The HPC computing domain has seen the emergence of accelerated heterogeneous architectures, most notably multi-core processors integrated with General Purpose Graphic Processing Units (GPGPU) [19, 20]. Examples of many-core architectures in the HPC domain include the Intel MIC [8] and Intel Xeon Phi [9] (features 60 cores).

Similarly, the real-time embedded domain has seen the emergence of the STMicroelectronics P2012/STHORM [1] processor, which includes a dual-core ARM-A9 CPU coupled with a many-core processor (the STHORM fabric); and the Kalray MPPA (Multi-Purpose Processor Array) [10], which includes four quad-core CPUs coupled with a many-core processor. One can also cite the Parallela from Epiphany and the Keystone II from Texas Instrument. In most cases, the many-core fabric acts as a processing accelerator [18].

The introduction of such platforms has set up the basic environment that allowed for the deployment of new types of applications sharing objectives and requirements from both the EC and HPC domains. For such applications, the correctness of the result depends on both performance and real-time requirements, and the failure to meet those is critical to the functioning of the system. Real-time Complex Event Processing (CEP) systems[6] [11] are an example of such applications; they challenge the performance capabilities by crossing the boundaries between the two domains.

**This research.** This work presents an overview of the framework — i.e., the software stack and computation model — proposed by the P-SOCRATES project consortium to parallelize applications on a many-core architecture while providing guarantees on their response times. Although the framework will be ported to different hardware architectures, we focus here on the Kalray MPPA platform to illustrate the proposed methodology.

## 2 Overview of the Kalray MPPA-256 architecture

The Kalray MPPA many-core chip features a total of 288 identical Very Long Instruction Word (VLIW) cores on a single die. More precisely, it is composed of 256 user cores referred to as Processing Elements (PEs) and dedicated to the execution of the user applications and 32 system cores referred to as Resource Managers (RM) and dedicated to the management of the software and

[6] A real-time CEP system is a system in which the data coming from multiple event streams are correlated in order to extract and provide meaningful information.

processing resources. The cores are organized in 16 compute clusters and 4 I/O subsystems. In Figure 1, the 16 inner nodes (blue boxes) correspond to the 16 compute clusters holding 17 cores each: 16 PEs and 1 RM. Then, there are 4 I/O subsystems located at the periphery of the chip, each holding 4 RMs.

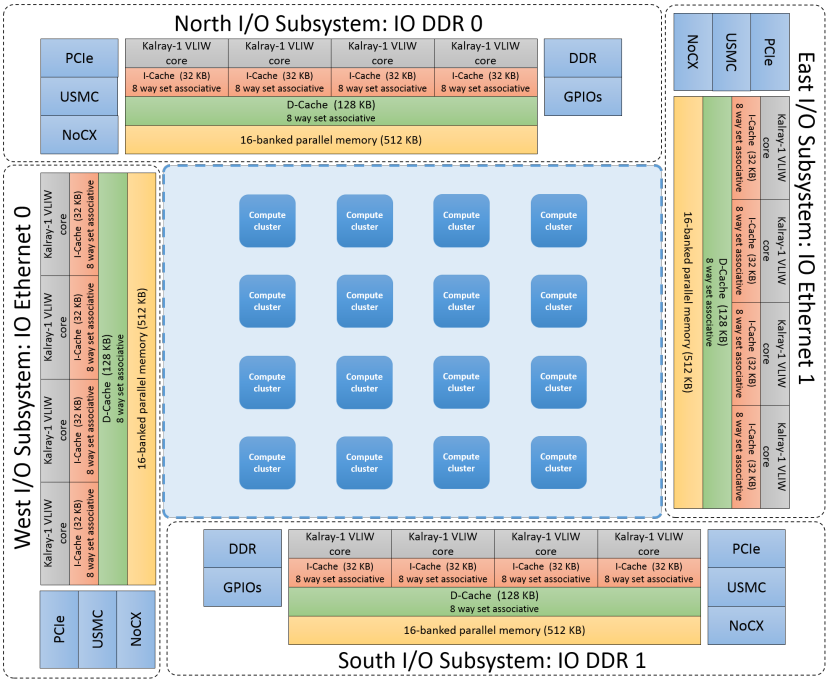


**Fig. 1.** Overview of the Kalray MPPA platform.

## 2.1 The I/O subsystems architecture

The 4 I/O subsystems (also denoted as IOS) are referenced as the *North*, *South*, *East*, and *West* IOS. They are responsible for all communications with elements outside the Kalray MPPA processor. Each IOS contains 4 RM cores, which operate as controllers for the MPPA clusters. Each of these quad-core (i) runs a RTEMS operating system, (ii) is connected to a shared 16-banked parallel memory of 512 KB, (iii) has its own private instruction cache of 32 KB and (iv) share a data cache of 128 KB, which ensures data coherency between the cores. Any program is started on the I/O cores, which are then responsible to properly offloading computation to the compute clusters via the NoC. Note that the IOS also implement various standard interfaces such as DDR3 channels; PCIe Gen3 X8; NoC eXpress interfaces (NoCX), etc.

### 2.2 The Network-on-chip (NoC)

The NoC holds a key role in the performance of the Kalray MPPA processor, especially when different clusters need to exchange messages at run-time. The 16 compute clusters and the 4 IOS are connected by two explicitly addressed NoC — the data NoC (D-NoC)[7] and the control NoC (C-NoC)[8] — with bi-directional links providing a full duplex bandwidth between two adjacent nodes. The two NoC are identical with respect to the nodes, their 2D-wrapped-around torus topology, and the wormhole route encoding. However, they differ at their device interfaces by the amount of packet buffering in routers and by the flow regulation at the source available on the D-NoC.

### 2.3 The compute clusters architecture

Each compute cluster and IOS owns a private address space, while communication and synchronization between them is ensured by the D-NoC and the C-NoC. We recall that each cluster contains 16 PE and one RM core.

**The cores.** Every core is equipped with private instruction and data L1 caches; runs nodeOS and communicates with other cores in the cluster through the shared memory. The RM core is in charge of (i) scheduling the threads on the PEs; (ii) managing the communication between the clusters and (iii) managing the communication between the clusters and the main memory.

**The shared memory.** The shared memory (SMEM) has a total capacity of 2 MB and comprises 16-banked independent memory of 128 kB per bank, enabling low latency access. A direct memory access (DMA) engine is responsible for transferring data between the shared memory and the NoC or within the shared memory. A Debug Support Unit (DSU) is also available.

## 3 Application architecture

In the P-SOCRATES view, the *application* comprises all the software parts of the systems that operate at the user-level and that have been explicitly defined by the user. The application is the software implementation (i.e., the code) of the functionality that the system must deliver to the end-user. It is organized as a collection of *real-time tasks*.

A *real-time (RT) task* is a recurrent activity that is a part of the overall system functionality to be delivered to the end-user. Every RT task is implemented and rendered parallelizable using OpenMP 4.0, the de facto standard parallel programming model used in shared memory-based architectures such as the Kalray MPPA. OpenMP version 4.0 has evolved from previous versions to

[7] The D-NoC is optimized for bulk data transfers.
[8] The C-NoC is optimized for small messages at low latency.

consider very sophisticated types of dynamic, fine-grained and irregular parallelism.

An RT task is characterized by a software procedure that must carry out a specific operation such as processing data, computing a specific value, sampling a sensor, etc. It is also characterized by a few (user-defined or computed) parameters related to its timing behavior such as its worst-case execution time, the frequency of its activation (aka period), the time frame in which it must complete (aka its deadline), etc. In P-SOCRATES, every RT task comprises a collection of *task regions* whose inter-dependencies are captured and modeled by a graph called the extended task dependency graph (eTDG).

A *task region* is defined at run-time by the syntactic boundaries of an openMP task construct. For example:

```
#pragma omp task
{
  // The brackets identify the boundaries of the task region
}
```

Since the task regions are defined in the code through the openMP task constructs, we will henceforth refer to them as *openMP tasks*.

An *openMP task part* (or simply, a *task part*) is a non-pre-emptible (at least from the OpenMP view of the world) portion of an openMP task. Specifically, consecutive task scheduling points (TSP) such as the beginning/end of a task construct, the synchronization directives, etc., identify the boundaries of an openMP task part. In the plain OpenMP task scheduler a running openMP task can be suspended at each TSP (not between any two TSPs), and the thread previously running that openMP task can be re-scheduled to a different openMP task (subject to the task scheduling constraints).

## 4 Overview of the P-SOCRATES software stack

Figure 2 gives an overview of the P-SOCRATES runtime methodology. The following explanation is organized as a list of bullet-points that traces the execution of a real-time task (using Figure 2 as reference), from its initial partial execution on the IOS to its offload onto the accelerator, explaining along the way the cluster assignment, the openMP task dependency checks, the mapping to the OS threads and the scheduling of the threads on the cores.

**① On the IOS side**

As illustrated in the box in the top-left corner of Figure 2, all the real-time tasks start their execution on the IOS to which they have been assigned and are scheduled on that quad-core by a partitioned or global scheduling algorithm. The RT tasks do not migrate from one IOS to another at run-time. In this example we have depicted four real-time tasks RT1, RT2, RT3 and RT4, all running on the same IOS. Note that each IOS runs on Linux as we envision a fully open-source software stack.

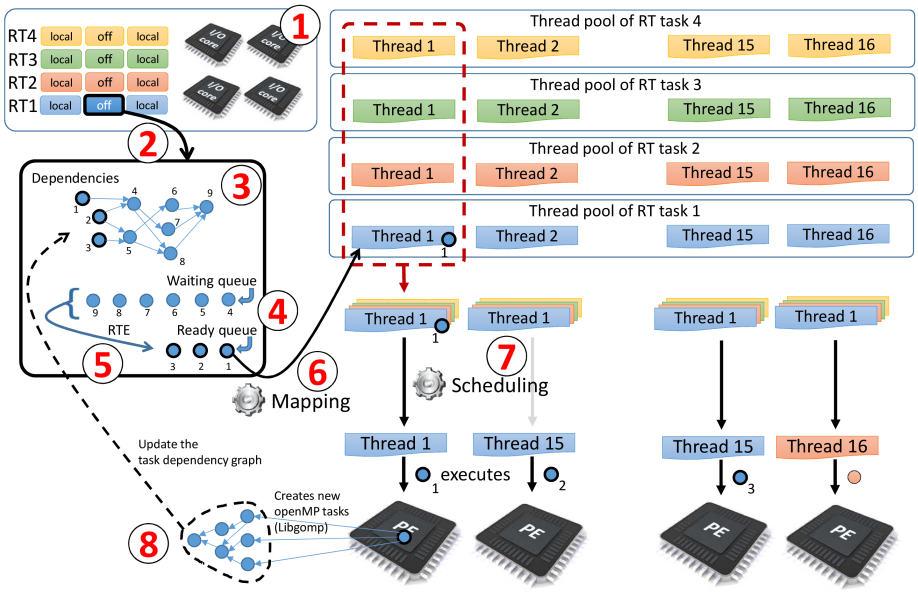


**Fig. 2.** Illustration of the software stack. Note that all the components depicted outside of the box in the top-left corner are part of a compute cluster.

As mentioned in the previous section, each RT task is modeled as a graph of openMP tasks. Some of these openMP tasks will be executed "locally" on the IOS to which their RT task has been assigned while others will be offloaded onto the accelerator, i.e. the many-core fabric. It will be written explicitly in the RT task's code (through the mppa_spawn() routine) which parts of the code, i.e. which of its openMP tasks, will be executed on the IOS and which ones will be sent to the accelerator. Therefore, each RT task can be seen as a collection of logical *segments*, where each segment is a collection of openMP tasks that execute either locally or on the accelerator. Although we have drawn only tasks composed of three consecutive segments (one local – one to be offloaded – one local), RT tasks can actually comprise an arbitrary number of segment, each containing an arbitrary number of openMP tasks.

### ② Offloading openMP tasks to the accelerator

Each time an openMP task is sent to the accelerator, a scheduler must select the cluster on which the openMP task will execute. This assignment will first be done via a simple bin-packing strategy such as next-fit, first-fit, etc., but later in the project, more elaborated techniques will be investigated in order to optimize, for example, the memory traffic between clusters and between the clusters and the main memory. As a first step, we will also impose that all the openMP tasks issued from the same RT task can only be offloaded to the same cluster in order to avoid the need for inter-cluster synchronization mechanism and potentially

reduce the communication traffic between clusters.

**③ The task dependency graph**

In Figure 2, consider that all the openMP tasks of the offloaded segment of RT1 have been sent to cluster 1. When a segment is offloaded to a cluster of the many-core, the essential openMP task dependency information is captured within a streamlined data structure hosted in the on-cluster shared memory. This data structure is a graph of openMP tasks called the *task dependency graph* (TDG), whose edges represent the inter-dependencies between them.

**④ The runtime queues**

The cluster also defines and maintains a ready-queue and a waiting-queue for that real-time task RT1. As seen in Figure 2, the offloaded segment of RT1 comprises 9 openMP tasks whose dependencies are captured by the TDG stored in the shared memory of the cluster. Among these 9 openMP tasks, three are ready to execute as they have no predecessor in their dependency graph while the remaining six must wait not to violate their precedence constraints.

**⑤ The openMP runtime library/environment**

At runtime, upon reaching an openMP *task scheduling point* like a task creation/completion/synchronization point, an OpenMP runtime library/environment (RTE) is responsible for updating the dependency graph and flagging the next openMP task[s] that become now ready to execute as a result of this update, i.e. all their predecessor nodes in the graph have finished their execution. Those openMP tasks that became ready are moved from the waiting queue to the ready queue, thereby indicating to the system that they are ready to be mapped to the OS threads.

There are multiple run-time libraries for the openMP programming model and in P-SOCRATES we have evaluated two of them, namely Nanos++ (that comes with OmpSs) and Libgomp (that comes with GCC). Nanos++ only supports OpenMP 4.0 while Libgomp has versions for OpenMP 4.0 and 3.0. The difference between the versions 3.0 and 4.0 of openMP is that the 4.0 introduces and provides support to handle dependencies between the openMP tasks. As the Kalray MPPA only supports Libgomp 3.0 and so cannot handle the dependencies, we will extend this Kalray-supported Libgomp 3.0 run-time library to incorporate mechanisms that handle dependencies.

**⑥ The openMP tasks to OS threads mapping**

In each cluster to which an openMP task of an RT task is assigned, there is for that RT task and thus for all its openMP tasks, a pool of 16 OS threads (one thread per PE) dedicated to their execution. The OpenMP run-time environment decides which ready openMP task is executed by which thread respecting the run-after dependencies among openMP tasks as defined in the TDG. This mapping between the OpenMP tasks and the threads will consider the poten-

tial inter-thread conflicts when accessing the on-cluster shared memory. Such an effect will be incorporated latter in the overall timing analysis of the RT tasks.

Our initial idea was to run the mapper on a dedicated PE but we are currently considering running it in a more distributed fashion. Note that the scheduler (see next bullet-point) could also benefit from running on a dedicated core as it would allow a higher runtime complexity and thus a higher precision when taking the scheduling decisions. This is because scheduling decisions may be based on heavy computations if for example, the objective is to minimize the traffic between the cores and the memory at runtime.

The OpenMP runtime environment internally holds the required data structures to dispatch the openMP tasks to the available "*workers*" and to properly synchronize. A *worker* is an *openMP thread* and those are the entities that are actually mapped to the OS threads, which we simply refer to as threads. For simplicity, we have overlooked in Figure 2 this additional conceptual layer of openMP threads and pretended that the openMP tasks are the entities to be mapped directly to the OS threads by the RTE.

We have struggled to keep the implementation of such infrastructure as lightweight as possible, to reduce to a minimum the library overhead and its final impact on parallelization effectiveness. In particular, data dependence checking is known to be among the principal contributors to this overhead. We are thus designing a lightweight lookup mechanism to support this feature at a low cost. Parallel updates to a simple look-up table are synchronized among multiple openMP threads. The look-up table placement leverages the multi-banked nature of the on-cluster shared memory to minimize the probability of conflicts.

**⑦ The scheduling of the OS threads on the PEs**

While the OpenMP tasks-to-threads mapping is entirely managed within the OpenMP RTE, as we mentioned in ① multiple RT tasks can be assigned to the same cluster and thus their 16 assigned OS threads may compete for the same cores. The scheduling of these threads is managed within the RTOS, Erika. To minimize the overheads for OpenMP to RTOS interaction, we have also designed a minimal support layer for fork-join parallelism, which tightly integrates OS threads and OpenMP threads.

As RTOS for the many-core fabric we have chosen Erika Enterprise [3–5], a free and open-source RTOS certified for the automotive market. This RTOS has a very small footprint (2KB) and already implements several scheduling algorithms known in the real-time literature. Thus, it is a very good candidate for our software stack. Within P-SOCRATES, we will port the RTOS onto the Kalray architecture.

As a first step, the scheduling algorithm that we will implement will be simple: the 16 threads of every RT tasks are indexed from 1 to 16 and thread number $k$ will be executed on PE number $k$. That is, we will implement partitioned scheduling where a thread, say $k$, cannot migrate from one PE to another at run-time. In fact, it cannot even execute on a PE $j \neq k$. Note that this partitioned scheduling paradigm may lead to an important waste of processing

resources as it is possible for a PE to be idle while other threads await their respective PEs to finish their current workload. Later in the project, we intend to extend the scheduling to global in order to overcome this limitation.

The scheduler is priority-based. Every RT task is assigned a constant priority level, which is passed on to the openMP tasks and ultimately to the threads executing those tasks. We are considering implementing different scheduling policies, starting with a fixed priority algorithm and then extend it to dynamic priorities such as EDF. Note that Rate Monotonic (RM) and other priority assignments that have good performance on a single core system may not be suitable for the case under consideration for different reasons. First, a well-known constraining factor on the achievable utilization of RM and EDF is given by Dhall's effect[2]. Therefore hybrid schedulers and priority assignments which are not uniquely based on the rate (or the deadline) of each task may achieve better performance than classic solutions. Second, the overhead related to pre-emptions and migrations need to be properly considered before adopting a preemptive scheduler, and/or when enforcing a particular schedule. To this end, we will investigate more refined models such as "limited-preemptive" scheduling solutions which reduce the cache-related overhead without affecting the overall schedulability [14, 13] as well as more dynamic techniques which try to balance those same effects against the load (e.g. work-stealing approaches [12, 6, 15]).

On an orthogonal dimension, the accesses to main memory for delivering fresh data to the cluster needs to be taken into account whenever a new task is scheduled. Access to main memory represents a significant bottleneck for data-intensive applications that perform a limited number of operations to large sets of data. This has a significant impact on the schedulability, so that properly scheduling memory and communication bandwidth could result in a greater increase in the systems schedulability than overly focusing on the scheduling of the processing elements, as long as the applications programming model is amenable to this. We will therefore also investigate and design memory-aware schedulers that jointly consider the allocation of processing and memory bandwidth inside each cluster. One of the options to evaluate will be using a predictable execution model (PREM) [16] that divides the execution of each task between a memory phase, when all data and instructions are fetched from shared memory to the local memory of each PE, and an execution phase, where each PE executes without conflicts on the shared bus. We will investigate in the tradeoffs obtained against the requirements imposed to the program code.

As a final remark, note that we do not make any restrictive assumption on the semantics of the supported OpenMP programs. Thus, it is allowed to dynamically create new tasks, possibly within conditional execution patterns, as shown by ⑧ in Figure 2. These newly created openMP task are directly handled by the openMP RTE (the TDG and the queues are updated accordingly).

## 5 Conclusion

This paper aimed at presenting an overview of the framework that is being developed within the EU project P-SOCRATES. More precisely, we introduced the software stack that we envision and that will enable applications to run in parallel, by using a parallel programming model from the HPC world, on a many-core architecture coming from the embedded computing world. We consider the unification of these two worlds as a necessity as modern applications have started sharing requirements from both. Many applications today have high-performance requirements as they have to deliver results requiring huge amount of data to be processed, and real-time requirements as they must carry out such computations in pre-defined time bounds. So far, the project has always received very positive and constructive feedback and the outcome of the first round of reviews was very positive as well. Many technical questions are still open today, and we foresee to open many more as we progress in the development/adaptation of the P-SOCRATES techniques and tools.

## References

1. Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 983–987, 2012.
2. S. K. Dhall and C. L. Liu. On a real-time scheduling problem. In *Operations Research*, volume 26(1), pages 127–140, 1978.
3. Evidence Srl. *Erika Enterprise*. Available at `http://erika.tuxfamily.org/drupal/`.
4. P. Gai, E. Bini, G. Lipari, M. Di Natale, and Abeni L. Architecture for a portable open source real-time kernal environment. In *2nd Real-time Linux workshop*, 2000.
5. P. Gai, F. Esposito, R. Schiavi, M. Di Natale, C. Diglio, M. Pagano, C. Camicia, and L. Carmignani. Towards an open source framework for small engine controls development. In *SAE/JSAE 2014 Small Engine Technology Conference & Exhibition*, 2014.
6. Ricardo Garibay-Martínez, Luis Lino Ferreira, Cláudio Maia, and Luis Miguel Pinho. Towards transparent parallel/distributed support for real-time embedded applications. In *8th IEEE International Symposium on Industrial Embedded Systems*, 2013.
7. S. Girbal, M. Moretó, A. Grasset, J. Abella, E. Quiñones, F.J. Cazorla, and S. Yehia. The next convergence: High-performance and mission-critical markets. In *1st Workshop on High-performance and Real-time Embedded Systems (HiRES)*, 2013.
8. Intel Corporation. *Intel Many Integrated Core (MIC) Architecture*, last access Aug 2014. Available at `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integratedcore/intel-many-integrated-core-architecture.html`.
9. Intel Corporation. *Intel Xeon Phi*, last access Aug 2014. Available at `http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`.

10. Kalray Corporation. *Kalray MPPA-256*, last access Aug 2014. Available at `http://www.kalray.eu/products/mppa-manycore/`.
11. D.C. Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. In *Addison-Wesley Longman Publishing Co. Inc.*, 2001.
12. Cláudio Maia, Luis Miguel Nogueira, and Luis Miguel Pinho. Scheduling parallel real-time tasks using a fixed-priority work-stealing algorithm on multiprocessors. In *8th IEEE International Symposium on Industrial Embedded Systems*, 2013.
13. José Marinho, Vincent Nélis, Stefan M. Petters, Marko Bertogna, and Robert Davis. Limited pre-emptive global fixed task priority. In *34th IEEE Real-Time Systems Symposium*, 2013.
14. José Marinho, Vincent Nélis, Stefan M. Petters, and Isabelle Puaut. Preemption delay analysis for floating non-preemptive region scheduling. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 497–502, 2012.
15. Luis Miguel Nogueira, Luis Miguel Pinho, José Fonseca, and Cláudio Maia. On the use of work-stealing strategies in real-time systems. In *High-performance and Real-time Embedded Systems (HiRES)*, 2013.
16. R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011.
17. L.M. Pinho, E. Quiñones, M. Bertogna, A. Marongiu, J. Pereira-Carlos, C. Scordino, and M. Ramponi. P-socrates: A parallel software framework for time-critical many-core systems. In *Proceedings of the 17th Euromicro Conference on Digital System Design (DSD)*, 2014.
18. L.M. Pinho, E. Quiñones, M. Bertogna, A. Marongiu, J. Pereira-Carlos, C. Scordino, and M. Ramponi. Time criticality challenge in the presence of parallelised execution. In *2nd Workshop on High-performance and Real-time Embedded Systems (HiRES)*, 2014.
19. L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers*, volume 27(3), pages 1–15, 2008.
20. H. Wong, A. Bracy, E. Schuchman, T.M. Aamodt, J.D. Collins, P.H. Wang, G. Chinya, A. Khandelwal-Groen, H. Jiang, and H. Wang. Pangaea: A tightly-coupled IA32 heterogeneous chip multiprocessor. In *Proceedings of the 17th ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 52–61, 2008.