CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Journal Paper

P-SOCRATES: A parallel software framework for time-critical many-core systems

Luis Miguel Pinho*
Vincent Nélis*
Patrick Meumeu Yomsi*
Eduardo Quiñones
Marko Bertogna
Paolo Burgio
Andrea Marongiu
Claudio Scordino
Paolo Gai
Michele Ramponi
Michal Mardiak

*CISTER Research Center

CISTER-TR-151002

2015/11

P-SOCRATES: A parallel software framework for time-critical many-core systems

Luis Miguel Pinho*, Vincent Nélis*, Patrick Meumeu Yomsi*, Eduardo Quiñones, Marko Bertogna, Paolo Burgio, Andrea Marongiu, Claudio Scordino, Paolo Gai, Michele Ramponi, Michal Mardiak

*CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: lmp@isep.ipp.pt, nelis@isep.ipp.pt, pamyo@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Current generation of computing platforms is embracing multi-core and many-core processors to improve the overall performance of the system, meeting at the same time the stringent energy budgets requested by the market. Parallel programming languages are nowadays paramount to extracting the tremendous potential offered by these platforms: parallel computing is no longer a niche in the high performance computing (HPC) field, but an essential ingredient in all domains of computer science. The advent of next-generation many-core embedded platforms has the chance of intercepting a converging need for predictable high-performance coming from both the High-Performance Computing (HPC) and Embedded Computing (EC) domains. On one side, new kinds of HPC applications are being required by markets needing huge amounts of information to be processed within a bounded amount of time. On the other side, EC systems are increasingly concerned with providing higher performance in real-time, challenging the performance capabilities of current architectures. This converging demand raises the problem about how to guarantee timing requirements in presence of parallel execution.

The paper presents how the time-criticality and parallelisation challenges are addressed by merging techniques coming from both HPC and EC domains, and provides an overview of the proposed framework to achieve these objectives.

Microprocessors and Microsystems 39 (2015) 1190–1203

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

P-SOCRATES: A parallel software framework for time-critical many-core systems ☆

Luís Miguel Pinho [a], Vincent Nélis [a], Patrick Meumeu Yomsi [a], Eduardo Quiñones [b], Marko Bertogna [c,*], Paolo Burgio [c], Andrea Marongiu [d], Claudio Scordino [e], Paolo Gai [e], Michele Ramponi [f], Michal Mardiak [g]

[a] ISEP, Porto, Portugal
[b] Barcelona Supercomputing Center, Spain
[c] University of Modena, Italy
[d] ETH, Zurich, Switzerland
[e] Evidence Srl, Italy
[f] Active Technologies Srl, Ferrara, Italy
[g] ATOS, Spain

ARTICLE INFO

Article history:
Available online 24 June 2015

Keywords:
Many-core systems
Real-time systems
Embedded systems
WCET analysis
Real-time scheduling
Parallel programming models

ABSTRACT

Current generation of computing platforms is embracing multi-core and many-core processors to improve the overall performance of the system, meeting at the same time the stringent energy budgets requested by the market. Parallel programming languages are nowadays paramount to extracting the tremendous potential offered by these platforms: parallel computing is no longer a niche in the high performance computing (HPC) field, but an essential ingredient in all domains of computer science. The advent of next-generation many-core embedded platforms has the chance of intercepting a converging need for *predictable high-performance* coming from both the High-Performance Computing (HPC) and Embedded Computing (EC) domains. On one side, new kinds of HPC applications are being required by markets needing huge amounts of information to be processed within a bounded amount of time. On the other side, EC systems are increasingly concerned with providing higher performance in real-time, challenging the performance capabilities of current architectures. This converging demand raises the problem about how to guarantee timing requirements in presence of parallel execution.

The paper presents how the time-criticality and parallelisation challenges are addressed by merging techniques coming from both HPC and EC domains, and provides an overview of the proposed framework to achieve these objectives.

1. Introduction

High-performance computing (HPC) and embedded computing (EC) systems have been traditionally running in opposite directions. On one side, HPC systems are typically designed to make the common case as fast as possible *in the average case*, without concerning themselves for the timing behavior (in terms of execution time) of the *not-so-often cases*. As a result, they are based on complex hardware and software structures that make any reliable timing bound almost impossible to derive. On the other side, real-time embedded systems implement energy-efficient and predictable solutions, without heavy performance requirements. Instead of *fast* response times, they aim at having *deterministically bounded response times*, in order to guarantee that deadlines are met. For this reason, EC systems are typically based on simpler architectures, using fixed-function hardware accelerators that are strongly coupled with the application domain.

In the last years, multi-core processors hit both high-performance and embedded computing markets [55]. The huge computational necessities to satisfy the performance requirements of HPC systems and the related exponential increments of power requirements (typically referred to as the *power-wall*) exceeded the technological limits of classic single-core architectures. For

☆ This work has been financially supported by the European Commission through the P-SOCRATES project (FP7-ICT-2013-10).

* Corresponding author.
E-mail addresses: lmp@isep.ipp.pt (L.M. Pinho), nelis@isep.ipp.pt (V. Nélis), pamyo@isep.ipp.pt (P.M. Yomsi), eduardo.quinones@bsc.es (E. Quiñones), marko.bertogna@unimore.it (M. Bertogna), paolo.burgio@unimore.it (P. Burgio), a.marongiu@iis.ee.ethz.ch (A. Marongiu), claudio@evidence.eu.com (C. Scordino), pj@evidence.eu.com (P. Gai), ramponi@activetechnologies.it (M. Ramponi), michal.mardiak@atos.net (M. Mardiak).

http://dx.doi.org/10.1016/j.micpro.2015.06.004
0141-9331/

these reasons, the main hardware manufacturers are offering an increasing number of computing platforms integrating multiple cores within a chip, contributing to an unprecedented phenomenon sometimes referred to as *the multi-core revolution*. Multi-core processors provide a better energy efficiency and performance-per-cost ratio, and performance scaling is achieved *via* thread level parallelism (TLP). Applications are split into multiple tasks that run in parallel on different cores, extending to multi-core system level an important challenge already faced by HPC designers at multi-processor system level: *the extraction of parallelism from applications*.

In the embedded systems domain, the deed for more flexible and powerful systems (e.g., from fixed function phones to smart phones and tablets) have pushed the embedded market in the same direction. Also in this market, multi-cores are increasingly considered as the solution to cope with performance and cost requirements [18]: the capability of scheduling multiple application services on the same processor, maximizes the utilization of hardware resources while reducing cost, size, weight and power requirements. However, embedded applications with time-critical (real-time) requirements are still executed on simple architectures that guarantee a predictable execution pattern while avoiding the appearance of timing anomalies [34]. For this reason, real-time embedded platforms still rely on single-core or simple multi-core CPUs, sometimes coupled to fixed-function hardware accelerators into the same System-on-Chip (SoC).

The convergent need for energy-efficiency (in HPC) and for flexibility (in EC), coming 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.

Examples of many-core architectures include the Tilera Tile CPUs [59] (shipping versions feature 64 cores) in the embedded domain and the Intel MIC [27] and Intel Xeon Phi [28] (features 60 cores) in the HPC domain. 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. In recent years, 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), because GPGPUs are a flexible and programmable accelerator for data parallel computations [54,62]. Similarly, in the real-time embedded domain, STMicroelectronics P2012/STHORM [8] processor, which includes a dual-core ARM-A9 CPU coupled with a many-core processor (the STHORM fabric); the Kalray MPPA (Multi-Purpose Processor Array) [29], which includes four quad-core CPUs coupled with a many-core processor; and the Parallella board from Adapteva[3], featuring a dual-core ARM-A9 coupled with the Epiphany many-core system. In all cases, the many-core fabric acts as a processing accelerator. A similar heterogeneous computing system is given by the Keystone II from Texas Instrument [56], featuring a host ARM15 quad-core coupled with eight DSP Very-Large Instruction Word (VLIW) cores. The VLIW capabilities of the DSPs allow simultaneously processing 64 instructions per cycle, as in larger many-core systems.

Many-core platforms offer a tremendous potential in terms of parallelism and energy efficiency (Gops/Watt), but the task of turning it into actual performance is demanded at the software level, and at programmers' skills. To this aim, several languages/extensions were proposed, that provide constructs (such as keywords or code annotations) to enable parallel code development at a high level of abstraction [47,48,39,26], that is, to let programmer *explicitly creating parallelism* in the code. The expressiveness of these programming frontends is a good starting point for designing an accurate and transparent methodology for timing analysis of highly parallel application, based on the information captured directly from within their source code.

The introduction of many-core systems has set up an interesting trend wherein both the HPC and the real-time embedded domain converge towards similar objectives and requirements. New types of applications are challenging the performance capabilities of hardware platforms, and the demand for increased computational performance is even more challenging when large amounts of data from multiple data sources must be processed and aggregated with *guaranteed processing response times*. This is the case of real-time complex event processing (CEP) systems [33], a new area for HPC in which the data coming from multiple event streams is correlated in order to extract and provide meaningful information within a bounded amount of time. Examples include cyber-physical systems (CPS), ranging from automotive and aircrafts to smart grids and traffic management, and banking/financial computing systems, where large amounts of real-time information needs to be processed for detecting time-dependent patterns, automatically triggering operations in a very specific and tight time-frame [58].

The underlying commonality of the real-time systems described above is that they are time-critical (whether business-critical or mission-critical) and with high-performance requirements. In other words, for such systems, the correctness of the result is dependent on both performance and timing requirements, and the failure to meet those is critical to the functioning of the system. In this context, it is essential to guarantee the timing predictability of the performed computations, meaning that arguments and analysis are needed to be able to make arguments of correctness – e.g., performing the required computations within well-specified bounds.

In this current trend, challenges that were previously specific to each computing domain, start to be common to both domains (including energy-efficiency, parallelisation, compilation, software programming) and are magnified by the ubiquity of many-cores and heterogeneity across the whole computing spectrum. In that context, cross-fertilization of expertise from both computing domains is mandatory. Although some research in the embedded computing domain has started investigating the use of parallel execution models (by using customized hardware designs and manually tuning applications by using specialized software parallel patterns [49]), a real cross-fertilization of expertise between HPC and embedded computing domains is still missing. FP7 project P-SOCRATES [57] is bringing together the required expertise from the HPC and EC domains to jointly address the challenge of providing timing criticality guarantees to systems with huge performance requirements. As a result, P-SOCRATES will enable the adoption of next-generation many-core embedded platforms in both the HPC and the embedded computing domains. Fig. 1 summarizes the ambitious target of the project.

The paper presents how the time-criticality and parallelization challenges can be addressed with a holistic approach, from the programming model to the underlying software stack, complemented with analysis tools for timing predictability. The overall approach and system stack is described, discussing how it allows tackling the predictable performance challenge.

The paper is structured as follows. The next section presents the main problems towards achieving time-predictable systems integrating techniques from high-performance and embedded computing domains. Section 3 shows the P-SOCRATES approach, the programming model adopted, and how the predictability problem is tackled. Section 4 describes how an application is structured according to the P-SOCRATES framework and the chosen programming model. It presents the vision and challenges of the proposed

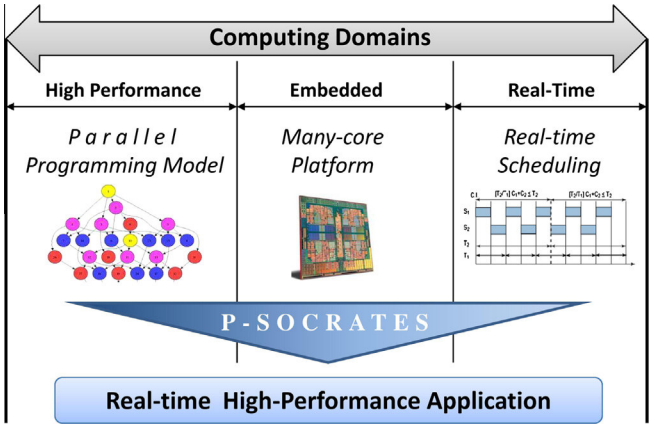


Fig. 1. P-SOCRATES approach integrating HPC parallel programming models, high-end embedded many-core platforms and real-time systems technology.

real-time programming model, which builds upon HPC programming models, augmented with dependencies and timing information. Section 5 describes in details the target hardware platform, and the specifications for P-SOCRATES software stack and tool chain. Section 6 then provides a brief summary of related work, while Section 7 summarizes the paper.

2. The predictability challenge

We believe that the next step towards the integration of high-performance and embedded computing domains will be the use of many-core embedded processors, which will provide the required performance level, still being energy-efficient and time predictable. An example towards this integration is provided by Mont-Blanc and Mont-Blanc2 FP7 projects [40], which are developing a new hybrid supercomputer based on energy-efficient embedded ARM CPUs coupled with high-performance NVIDIA GPU many-core processors. However, there is still one fundamental requirement that has not yet been considered: *time predictability as a mean to address the time criticality challenge when computation is parallelised to increase the performance*.

Industries with both high-performance and real-time requirements are eager to benefit from the immense computing capabilities offered by new many-core embedded designs. However, these industries are also highly unprepared for shifting their earlier system designs to cope with this new technology, mainly because such a shift requires adapting the applications, operating systems, and programming models in order to exploit the capabilities of many-core embedded computing systems. Real-time methods to determine the timing behavior of an embedded system are not prepared to be directly applied to the HPC domain and many-core platforms, leading to a number of significant challenges. Although customized processor designs could better fit real-time requirements [49], the design of specialized processors for each real-time system domain is unaffordable.

On one side, different parallel programming models and multiprocessor operating systems have been proposed and are increasingly being adopted in today's HPC computing systems. In recent years, the emergence of accelerated heterogeneous architectures such as GPGPUs, have introduced parallel programming models such as OpenCL [46], the currently dominant open standard for parallel programming of heterogeneous systems, or CUDA [44], the dominant proprietary framework of NVIDIA. Unfortunately, they are not easily applicable to systems with real-time requirements, since, by nature, many-core architectures are designed to integrate as many functionalities as possible into a single chip, harnessing predictability and the capability of providing timing guarantees.

On the other side, modern embedded platforms include plenty of application-specific accelerators in their designs, and applications are manually tuned to ensure predictable performance. This limits the flexibility of the whole system, and complicates the software development process. The fact that, in the near future, COTS many-core components are likely to dominate the embedded computing market, further exacerbates these problems. As a consequence, migrating real-time applications to many-core execution models with predictable performance guarantees will require a complete redesign of current software architectures and design approaches.

The main problem in applying classic real-time techniques to many-core systems is related to the difficulties in deriving reliable and tight upper bounds on the worst-case execution time (WCET) and response time (WCRT) of real-time tasks. The WCET represents the maximum time needed to execute a real-time task when it runs in isolation, e.g., without any interference from other tasks or devices. Instead, the WCRT also accounts for the worst-case interference that the task may suffer from other executing tasks. Although different timing and schedulability analysis techniques are available in the real-time literature to derive tight WCET and WCRT bounds in single-core systems, such techniques cannot be easily extended to many-core systems.

In a schedulability analysis, the objective is to check analytically, at design or run time, whether all the timing requirements of the system will be met. In its simplest form, a schedulability test is just a mathematical proof that the system is deemed schedulable, i.e., all the deadlines of concurrent tasks will always be met at run-time. Unfortunately, most of the current state-of-the-art techniques for analysis and scheduling assume that the system activities (tasks) are functionally independent and most of their parameters are exactly known at design time. For example, most of the schedulability tests proposed so far assume that the worst-case execution time of an activity is known at design time and invariant. However, when running on a real hardware platform, tasks that are co-scheduled on different cores share hardware resources, such as caches, communication buses and main memory. This introduces implicit functional dependencies among tasks, affecting their timing behavior. This effect is magnified when scaling to a many-core. As a consequence, current analysis and scheduling techniques cannot be applied to many-cores, but they need to be augmented to factor in all the sources of contention due to shared resources. Preliminary results toward this direction have already been presented (e.g. [38,14]).

Fig. 2 shows a typical distribution of the execution times of a real-time task running in isolation on a single core system [1]. As shown, the range of possible execution times is typically large, with a tail in the distribution that may be rather far from the average-case behavior: since it is often quite difficult to exactly compute the WCET of a task, currently we typically overestimate a safe upper bound for it, to ensure real-time guarantees. This large variability in the spectrum of execution times, along with the uncertainties in determining tasks WCET, causes a significant waste of computing resources. Real-time systems are often (over) dimensioned to deal with worst, pathological "corner" cases, and a considerable amount of computing resources is used to correctly deal with situations that are very unlikely to happen (at the rightmost side of the spectrum in the figure). In many-core systems, this problem is even more magnified by the additional interferences due to the simultaneous execution of multiple cores, who share the memory, I/O and communication resources. Therefore, smarter analysis and scheduling algorithms must be devised in order to efficiently and optimally using the resources available in the system while meeting timing constraints.

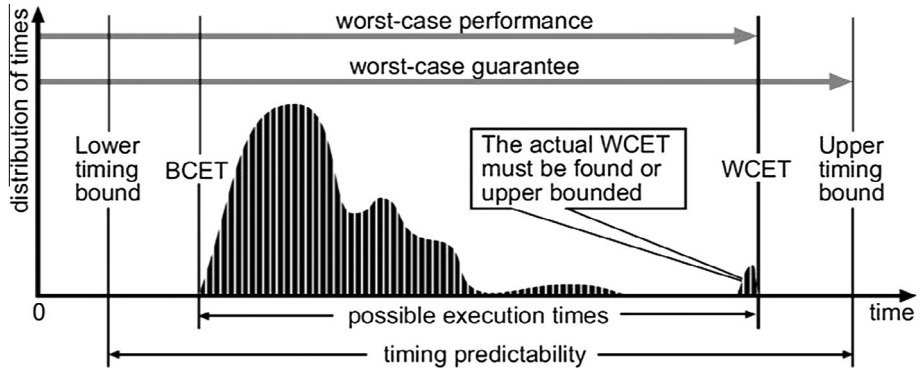


Fig. 2. Typical distribution of execution times of a real-time task.

3. P-SOCRATES approach

In order to tackle the predictable parallelization challenge, P-SOCRATES specified a complete and coherent software system stack, able to bridge the gap between application design and hardware many-core platform. For this purpose, the project devised a new programming framework to combine real-time embedded mapping and scheduling techniques with high-performance parallel programming models and associated tools, able to express parallelisation of applications. The programming model is being extended to support real-time properties and timing information. The software stack designed in the project (shown in Fig. 3) extracts a task dependency graph from the application, statically mapping these tasks to the threads of the operating system, which are then dynamically scheduled on the many-core platform.

Enhanced parallel programming models are being investigated, incorporating new annotations and compiler techniques to automatically generate an extended task dependency graph containing not only the data dependencies among tasks, but also relevant information to derive the impact on execution time due to sharing resources when tasks communicate. This information is then used by the mapping and scheduling algorithms to properly select the most suitable resource allocation strategies. The mapping algorithm we are currently designing, statically builds the required run-time configuration, efficiently assigning tasks-to-threads in order to guarantee timing requirements without performance degradation. Then, the underlying scheduling algorithm, implemented within the operating system, dynamically interprets the task-to-thread mapping into an efficient thread-to-core allocation, selecting which thread to execute on each core, and arbitrating the access to other shared resources.

The proposed techniques will be implemented on open-source real-time operating systems ported on the selected many-core platform. A timing and schedulability analysis (including a schedulability analysis integrated with an interference analysis) and a module that the mapping will use to check response time of the allocation, guarantee that the real-time requirements of the application are met using the selected mapping and scheduling algorithms. Finally, a general method to express the COTS many-core processor design characteristics is being defined to drive the allocation of tasks-to-threads and threads-to-cores, along with the associated timing and schedulability analysis.

The approach will be benchmarked using a set of existing applications: a complex event processing system (CEP) engine, part of an Intelligent Transport Application; an application from the aerospace domain which pre-processes samples of infra-red H2RG detectors; and an online service for smart advertisement.

These applications will be used to assess the validity and efficiency of the P-SOCRATES framework, using the following metrics: (i) reduction of average and worst-case response time, (ii) percentage of tardy instances, and (iii) memory and code footprint.

3.1. Real-time parallel programming model

In this section we briefly describe OpenMP, which is the target programming models chosen for P-SOCRATES.

Programming parallel architectures is a non-trivial task: in 2010, Blake et al. [9] highlighted that most of the existing desktop applications are not able to exploit more than a few cores at the same time. In the last 30 years, a number of programming model were proposed to support parallel programming: most of them provide abstractions for thread-based parallelism under shared memory assumption. For instance, the PTHREADS API [41] is a successful attempt of supporting and standardizing parallel programming, by directly exposing the concept of *thread* to the high-level source code. OpenMP [48] also follows a thread-centric programming style, and evolved through the years to include the concepts of *task* and *processing device* in the programming model (in 2008 and 2013 respectively). Due to its ease-of-use and its lightweight pragma-based programming interface, OpenMP eventually became one of the most known and widely adopted parallel programming models, representing the *de facto* standard for shared-memory systems.

In the last decade, GPUs became popular also outside the gaming and image processing domain, and they were adopted to build the most powerful and energy efficient HPC systems.[1] As a consequence, these platforms and the related programming models evolved: the Compute Unified Device Architecture (CUDA [44]) is now the reference model for programming NVIDIA architectures; on another side, standardization efforts led to the OpenCL specifications, and to domain-specific programming extensions, such as the OpenVX API (for image-processing and computer vision).

One of the goals of P-SOCRATES is to merge the tasking model "traditionally" used in the real-time domain and the programming models coming from the HPC and EC domains, and to provide a unique execution model that fits both needs and that is as much as possible platform independent. The OpenMP specifications already incorporate the concepts of *threads* and (regular and irregular) *tasks* in a shared memory model, hence being the perfect candidate for our purposes. Among the different programming styles provided by OpenMP, we found *tasking* perfectly suits our needs, for its flexibility and expressiveness.

OpenMP [48] was developed at the end of the 90s to support loop-based, regular applications. The standard evolved during the years to support a more irregular and multi-level parallelism [47], and, recently, also architectural heterogeneity [48]. Thanks

[1] see, e.g., the Green500 list [13].

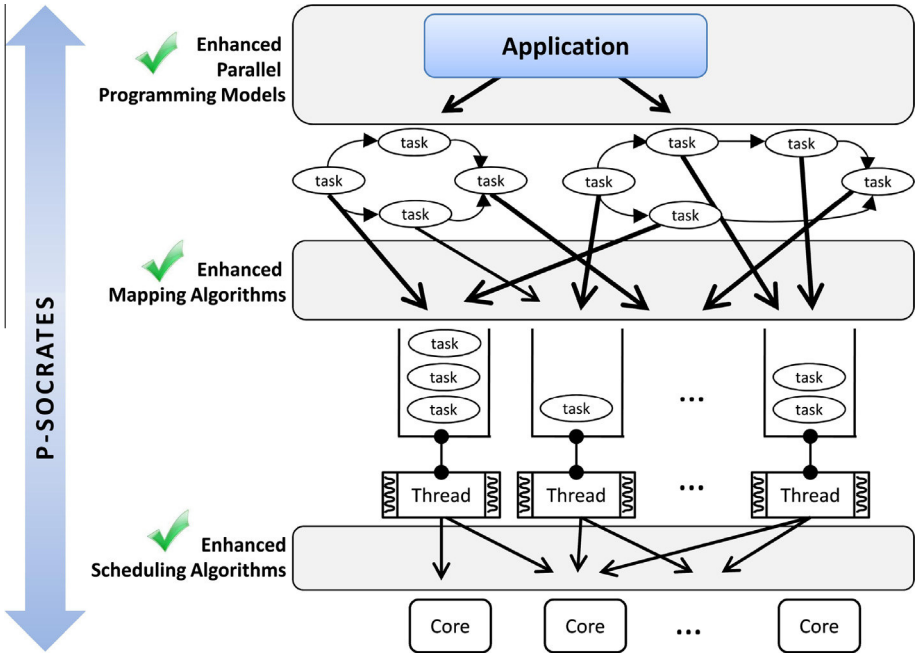


Fig. 3. P-SOCRATES approach integrating HPC parallel programming models, high-end embedded many-core platforms and real-time systems technology.

to this, it has also gained attention in the embedded domain, due to its ease-of-use and straightforward pragma-based programming interface.

OpenMP defines *task* annotations [47] to represent independent units of work that can run concurrently. Recently, its specifications evolved [48] with new directives, *in*, *out* and *inout*, that allow introducing asynchronous parallelism by defining dependencies among task. Fig. 4 shows a source code example using the dependency annotations. Each instance of task *foo* depends on data generated in previous loop iterations – i.e., *inout*($A[i-1]$). Similarly, the task *bar* depends on *foo* outcome – i.e., *out*($B[i]$).

This extension increases the freedom of task scheduling: tasks are scheduled for execution as soon as all depend tasks finished and there are available processor resources. In [60], we performed a detailed analysis on how information given by the OpenMP task dependencies can be used to support application timing analysis, proving the applicability of OpenMP also in a real-time domain.

OpenMP 4.0 specifications also introduce the so-called *offloading extensions*, with a `target` pragma for annotating portions of code to run on a specific accelerator device.

Current implementations of OpenMP include the GNU GCC, which incorporates a run-time, Libgomp [20], compatible with OpenMP 4.0.

In P-SOCRATES we employ OmpSs [17], a parallel programming framework developed by Barcelona Supercomputing Center (BSC) whose effectiveness has been widely demonstrated in the HPC domain [12]. In OmpSs, the data-dependencies annotations are interpreted by a compiler, Mercurium, that emits calls to the run-time system Nanos++. Nanos++ is a parallel run-time system that dynamically generates the *task dependency graph* (TDG) at run-time. Each time a new task is created its *in* and *out* dependencies are matched against those of existing tasks. If a dependency (*read-after-write*, *write-after-write* or *write-after-read*) is found, the task becomes a *successor* of the corresponding tasks. Tasks are scheduled for execution as soon as all their predecessor in the graph have finished and there are available processor resources. Fig. 5 shows the complete system stack of OmpSs. For this reason, we choose OmpSs as the official OpenMP support for this project.

3.2. Tackling the predictability challenge

Current parallel frameworks base scheduling decisions on information available at run-time – i.e., the task dependency graph and processor resources availability (see Fig. 5) – which makes it difficult to provide real-time guarantees. The reason is that the way tasks use shared processor resources determines the interferences that different tasks will suffer when accessing them, affecting the overall execution time of the application. A different usage of processor resources will result in a different execution.

In order to provide real-time guarantees without suffering any performance degradation, it is required to statically identify at design time which run-time configuration is needed, so the usage of shared processor resources is fixed and time guarantees can be provided. Therefore, it is of paramount importance to recover, at design time, relevant information to fix the usage of processor resources and so provide timing guarantees.

This challenge is addressed by *extending parallelism annotations*, which are extracted by the compiler, to identify portions of the application (tasks) that can run in parallel as well as *relevant information to derive the impact on execution time due to sharing resources when tasks communicate*. The compiler generates an *extended task dependency graph* (eTDG) containing the information required by the mapping and scheduling tool and the timing analysis method to allocate tasks to the different processor resources, guaranteeing that the real-time constraints of the application are accomplished. In other words, in order to provide timing guarantees, there is a necessity to fix the usage of shared processor resources.

Fig. 6 shows the envisioned real-time parallel programming framework in which relevant information for task scheduling and timing analysis is recovered at compile- and design-time to fix the usage of processor resources.

The P-SOCRATES approach presents multiple research challenges – both at compile-time and at design-time – that involve timing analysis and scheduling for predictability. This section summarizes the most important ones, and how we tackled each of them.

```
void compute (int *A, int *B, int N) {
    for(int i=0; i<N; i++) {
#pragma omp task depend(in:A[i-1], inout:A[i], out:B[i])
        foo(&A[i-1],&A[i],&B[i]);

#pragma omp task depend(in:B[i-1], inout:B[i])
        bar(&B[i-1],&B[i]);
    }
}
```

Fig. 4. OpenMP 4.0 code sample showing the data-flow dependencies among tasks.

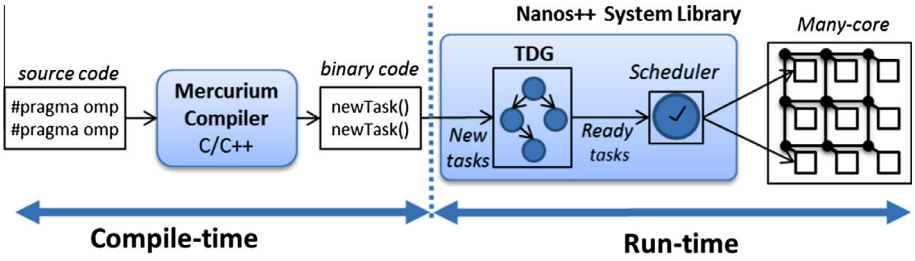


Fig. 5. OmpSs parallel programming framework.

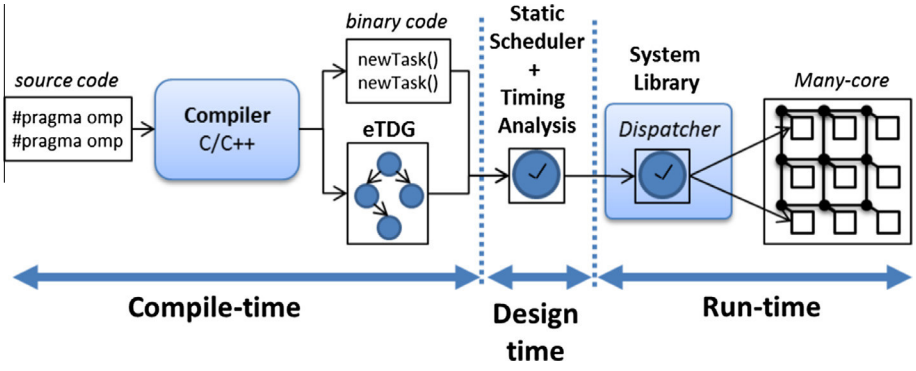


Fig. 6. Envisioned real-time parallel programming framework to provide timing guarantees.

3.2.1. Timing analysis

When performing the timing analysis of an application, tight execution bounds may be provided only if the details of all tasks are completely known. Unfortunately, some of this information is available only at run-time. This is the case, for instance, of data dependencies based on pointers, variable values or loop boundaries. In Fig. 4, if the number of iterations (N) is not known, we cannot determine how many task instances of *foo* and *bar* will be executed and so we cannot generate a complete eTDG. Similarly, in Fig. 7, if i and j values are not known at compile-time, it is not possible to determine if a data-dependency among tasks *produce* and *consume* exists.

In case the data-dependency cannot be solved, or loop boundaries are not known at compile-time, it is required to consider *conservative approaches* in order to guarantee the functional correctness of the program. Thus, if there is an unknown data-dependency, the construction of the eTDG must assume that this data-dependency exists. Similarly, if a loop boundary is unknown, it is required to determine an upper bound of the maximum number of loop iterations [2]. Needless to say that following a conservative approach will affect the average performance of the application. That is, false data-dependencies in the eTDG will force tasks to be executed sequentially. Similarly, assuming loop boundaries with higher number of iterations will make the eTDG to contain a higher number of task instances than the ones actually created, over-dimensioning the system due to tasks that are never executed.

```
void compute (int *A, int i, int j) {
#pragma omp task depend(inout:A[i])
    produce(&A[i]);

#pragma omp task depend(inout:A[j])
    consume(&A[j]);
}
```

Fig. 7. The value of i and j must be known to determine the dependency among tasks *producer* and *consumer*.

Fig. 8 shows the expected trends in the average and guaranteed performance when following conservative approaches. X-axis represents different levels of data recovered at compile-time, so the usage of processor resources can be fixed. As more information is extracted at compile-time, a more precise eTDG can be built and so higher guaranteed performance can be provided (light blue curve on the bottom). However, due to the conservative approaches, the eTDG can differ from the TDG created at run-time, so that the average performance can be degraded (dark

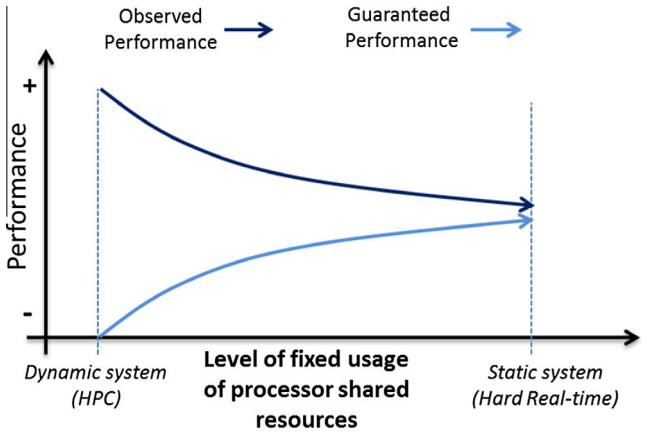


Fig. 8. On the left side, highly dynamic systems (e.g., HPC) can achieve a higher observed peak performance than static hard-real time systems (on the right), but lower guaranteed/worst-case performance.

blue curve on the top). Although counter-intuitive, it may happen that, in order to increase the guaranteed performance, a core is kept idle even when there is some pending workload to execute.

3.2.2. Scheduling

At *design-time*, it is necessary to provide the system with appropriate means to map the task dependency graphs to the underlying operating system threads (mapping), and dynamically schedule these threads to achieve both predictability and high-performance (scheduling). Although previous works [51,24] have shown that run-time characterization and management of locality has more potential than static locality analysis, dynamic information usage is a stopper to provide the timing guarantees for parallel applications on a many-core. As a consequence, we are performing research about how to allocate tasks to processor resources, which accounts for inter-task interference when accessing shared resources. To that end, the programming model must be extended so that the responsibility for managing locality is shared among the programmer and the mapping tool. This allows providing timing guarantees to application customers, and maximum performance. Data annotations with *in*/*out* clauses provide a reasonable balance between the programmer and the system in managing locality [17], but further research is needed to minimize the interferences when accessing shared resources.

3.2.3. Sources of interference and the memory problem

There are multiple sources of interference that may affect the timing behavior of the system activities, including contention for the cores (by higher priority workloads/tasks), contention on the buses and interconnection networks, concurrent accesses to cache memory, main memory bottleneck, etc. The P-SOCRATES project investigates the impact of such interference on the timing behavior of the tasks in the targeted many-core platforms. To characterize this impact new timing analysis techniques and tools are being devised. These tools are essentially based on runtime measurements and have for objective to assign a maximum execution time to every node of the activity dependency graphs. These maximum execution times are calculated to factor in all possible run-time interferences between the nodes (activities) and the hardware architecture. That is, they provide for all possible contention delays due to concurrent accesses to all the shared resources, except for the cores. The interference between the activities at the core-level is analyzed and upper-bounded at a later stage, during the schedulability analysis.

The challenge of characterizing the system timing behavior must be tackled in a holistic, integrated perspective, identifying the main scheduling bottlenecks of the considered hardware architectures. Our proposal is to construct the extended task dependency graph (eTDG) in synergy with the mapping and scheduling algorithms, with feedback from the timing and schedulability analysis. Smart scheduling solutions are being devised, which consider not only the cores, but also other kind of resources, such as memories, communication and synchronization mechanisms, to limit the variability in the task execution times when running on many-core systems. The proposed scheduling solutions lend themselves to a tight timing and schedulability analysis for extracting worst-case upper bounds that are sufficiently close to the exact response times of each task. The strategy cannot be to search for all possible combinations in the whole design space. A guided process needs to be introduced, which is able to reason on the best mapping for a particular result.

The shift towards multi- and many-core architectures gave birth to a huge number of platforms and architectural templates, where hundreds of cores are connected and communicate each other. In these platforms, communication is often implemented *via* shared memory banks, under NUMA[2] or non-NUMA scheme. We believe that the growth of computing units, together with the increasing performance gap between memories and cores (known as *memory-wall*) will soon bring us to a scenario where *the memory, and not the computing cores, will be the scarce resource in the system, hence the resource to schedule to provide timing guarantees*. This is in contrast with "traditional" approaches, where one or few processing cores must be shared – and scheduled – among multiple threads. In the EC and HPC world there are several techniques were already proposed to speedup performance by scheduling wisely the memory transfers, for instance, to enable *data prefetching* or to increase data locality. Strangely, not much work was performed in this direction by the real-time community (Pellizzoni et al. [50] are the first one moving in this direction), and we are also exploring this research path, and to propose a *memory-centric scheduling approach* to real-time many-core systems.

4. 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 [48], which supports 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, its period, and its deadline. 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, as shown in Fig. 7.

[2] *Non-Uniform Memory Access*: the time spent by a core for accessing a specific memory item depends on the locality of the corresponding memory bank to the core itself.

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 (as defined by the OpenMP tasking execution model) 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).

5. Overview of the P-SOCRATES stack

This section provides the execution model and software stack of the P-SOCRATES framework. The model itself is independent of a particular platform architecture, nevertheless its instantiation needs to take in consideration the specific structure of the underlying many-core. Therefore, the section will first provide a brief description of the reference architecture being used, before detailing the software stack.

5.1. Overview of the many-core architecture

In the scope of the project, the Kalray MPPA-255 [29] is the reference architecture, since it is a representative state-of-the-art many-core platform, tailored for several application domains, varying from HPC calculus to embedded image processing, computer vision and so on. The software ecosystems is extremely rich, and several programming models are supported. The software development kit (SDK) and runtime libraries shipped with the board provide native support for programming OpenMP tasks on the computing clusters.

Recently, Kalray performed a study on the possible applicability of the board in a real-time scenario [15], showing a great potential and a promising research path, but still many real-time challenges need to be addressed, and there is still the need for a complete real-time framework for this platform.

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 processing resources. The cores are organized in 16 compute clusters and 4 I/O subsystems. In Fig. 9, 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.

The 4 I/O subsystems (also denoted as IOS) are referenced as the *North*, *South*, *East*, and *West* IOS. They are responsible for communications with elements outside the Kalray MPPA processor. Each IOS contains 4 RM cores, which operate as controllers for the MPPA clusters. Each RM 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. Programs written for the Kalray MPPA start their execution on the I/O cores, which in turn offloads computation to the compute clusters via the NoC. Communication to and from the "external word" is supported via a number of standard interfaces such as DDR3 channels; PCIe Gen3 X8; and NoC eXpress interfaces (NoCX).

The on-chip 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 NoCs – the data NoC (D-NoC) and the control NoC (C-NoC)[3] – 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.

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.

Every core is equipped with private instruction and data L1 caches. It runs a lightwight micro-kernel called NodeOS, and communicates with other cores in the cluster through shared memory. The RM core is in charge of scheduling the threads on the PEs, and of managing the communication between the clusters, and between the clusters and the main 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.

5.2. Software stack

Fig. 10 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 Fig. 10 as reference), from its initial partial execution on the I/O subsystem (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. As previously introduced, the IOS is composed of four clusters located at the boundaries of the chip, each of which embeds four processing cores.

① On the IOS side

As illustrated in the box in the top-left corner of Fig. 10, 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.

As mentioned in Section 4, 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. In a system with full OpenMP 4 support, the standard mechanism for doing that are the so-called *offloading extensions*, more specifically using `target` pragma for specifying the code to accelerate. The software ecosystem of MPPA currently supports OpenMP specifications till 3.1, and only in the cluster subsystem, so there are no the offloading extensions in the IOS. We are currently implementing OpenMP 4.0 offloading for IOS clusters, and we will build the necessary runtime support on top of the native

[3] The D-NoC is optimized for bulk data transfers, while the C-NoC is optimized for small messages at low latency.

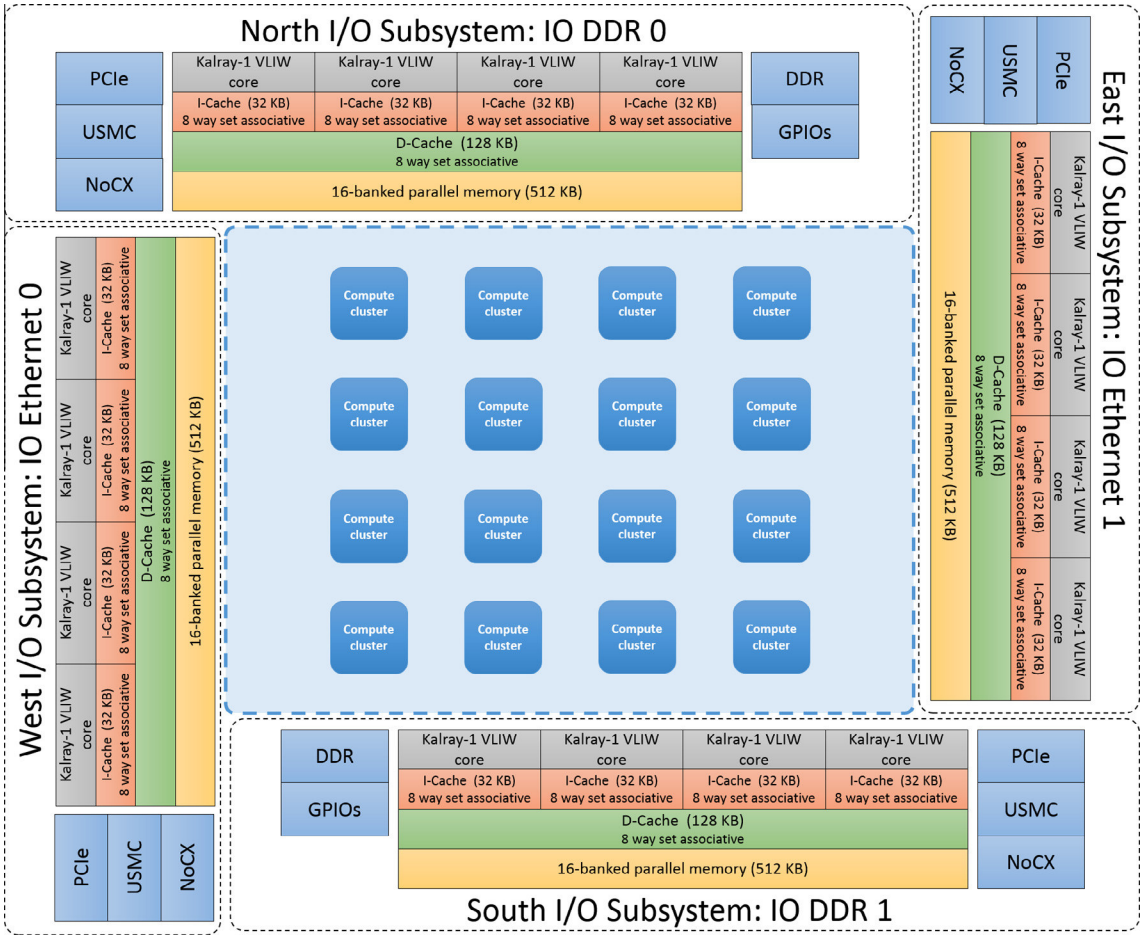


Fig. 9. Overview of the Kalray MPPA platform.

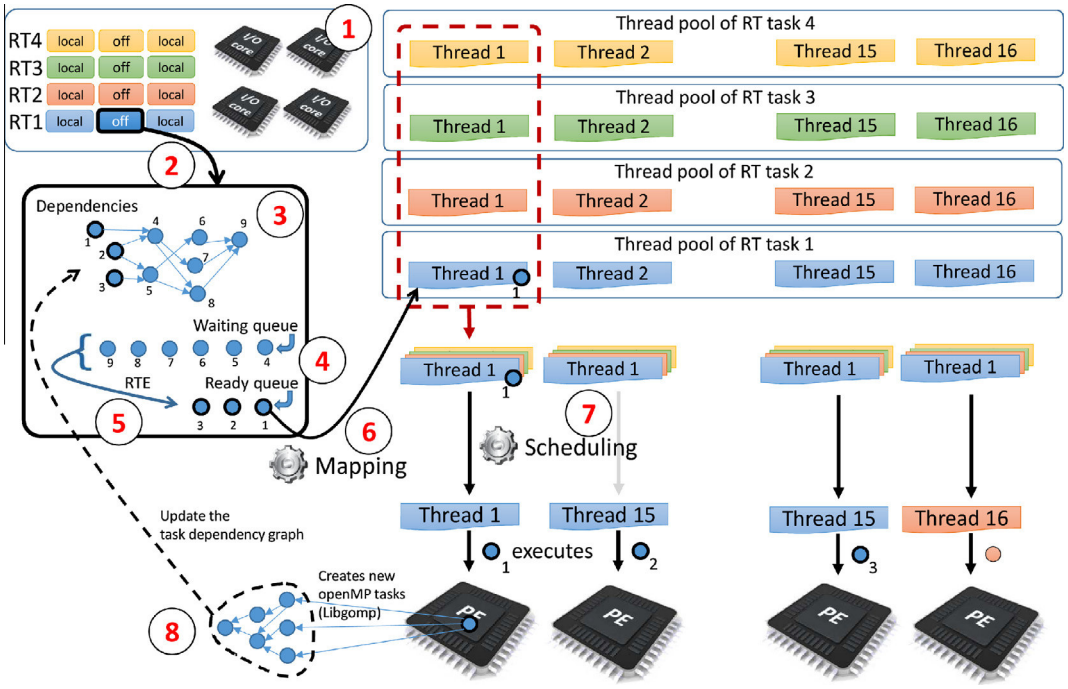


Fig. 10. Illustration of the software stack. Note that all the components depicted outside of the box in the top-left corner are part of a compute cluster.

`mppa_spawn()` routine. In the current version of P-SOCRATES stack, we offload (RT-)tasks to the many-core accelerators directly invoking the `mppa_spawn()`, and specifying which parts of RT tasks (i.e., the subset of OpenMP tasks) will be executed on the IOS, and which ones will be sent to the accelerator.

As a consequence, 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 in the figure 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 a logical segment is sent to the accelerator, a scheduler must select the cluster on which its OpenMP tasks will execute. This assignment will first be done via a simple bin-packing strategy such as next-fit and first-fit, 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. The restriction could be relaxed, i.e., we can let RT tasks migrate from one cluster to another (or from IOS to another), but this would increasing the complexity of the timing analysis due to inter-cluster communication. For this reason, we are currently consider a simple model where RT tasks are confined within a cluster/IOS.

③ The task dependency graph

In Fig. 10, 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. Note that the size of TDG exponentially grows with the number of nodes and edges, hence its storage becomes a "practical" problem. For this reason, we are implementing a lightweight OpenMP run-time library (see also ⑤); preliminary results indicate a significant reducing of the memory consumption compared to current implementations.

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 Fig. 10, 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 [17]) and Libgomp (that comes with GCC [20]). 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 a pool of 16 OS threads (one thread per PE) dedicated to the execution of the set of OpenMP tasks belonging to that RT task. The OpenMP run-time environment schedules every OpenMP task which is ready for execution on a specific thread, and enforces the run-after dependencies among OpenMP tasks which are defined in the TDG. This mapping between the OpenMP tasks and the threads considers all the potential inter-thread conflicts when accessing the on-cluster shared memory. Also the timing analysis of the RT tasks takes this into account.

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 among them. 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 Fig. 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 dependency 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 [19,21,22], a free and open-source RTOS certified for the automotive market. This RTOS has a very small footprint (2 KB) and already implements several scheduling algorithms

known in the real-time literature. Thus, it is a very good candidate for our software stack onto the Kalray architecture, as the OS kernel for the computing clusters subsystem that comes with MPPA package does not provide real-time capabilities. Note that a real-time operating system (RTEMS) already runs on the IOS, so there is no need to port Erika also on it. Nevertheless, we are currently investigating the use of real-time variants of Linux in the IOS.

The first version of scheduling algorithm is 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, it implements a 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, and every RT task is assigned a constant priority level. Since the execution model of OpenMP has no notion of "task priority", the runtime propagates this information directly to the worker threads at the OS layer, at the point where the specifications allow execution to be pre-empted (e.g., at task scheduling points – TSPs). 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 [16]. 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 are investigating more refined models such as "limited-preemptive" scheduling solutions which reduce the cache-related overhead without affecting the overall schedulability [37,36] as well as more dynamic techniques which try to balance those same effects against the load (e.g. work-stealing approaches [35,23,43]).

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 are therefore also investigating and design memory-aware schedulers that jointly consider the allocation of processing and memory bandwidth inside each cluster. One of the possible options we are currently exploring is to use the (previously introduced) predictable execution model (PREM) [50] 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 are investigating 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 Fig. 10. These newly created OpenMP task are directly handled by the OpenMP RTE (the TDG and the queues are updated accordingly).

6. Related work

The end of Dennard scaling enabled a shift to **massively parallel architectures**, both in the HPC and EC domains. And a promising design choice for next-generation computing systems is to couple a powerful, general-purpose host processor to massively parallel accelerators featuring tens-to-hundreds of simple and energy efficient cores [29,8,3]. Kalray MPPA [29], the platform targeted in P-SOCRATES, is an example of how such a technology can be successfully adopted in the EC domain, while Intel MIC [27] and Intel Xeon Phi [28] are well-known examples of platforms for HPC. To enable architectural scalability, many-cores accelerators are organized as a fabric of tightly coupled clusters, with complex hierarchical memory systems. To meet the low energy budget typical of embedded systems, data caches are partially replaced with more energy-efficient scratchpad memories. This is for instance the case of STMicroelectronics' STHORM/P2012 [8]. General purpose computation on graphics processing units (GPGPU) has also received a lot of attention, as it delivers high performance computing at a rather low cost. This approach is however efficient only for data parallel computations, programmed in frameworks such as OpenCL [46] or CUDA [44], and it does not allow supporting time-predictability.

In what concerns **programming models**, the HPC world has seen a plethora of proposals for data or task parallelism (e.g. [5,61,25]). Furthermore, approaches such as [51] or [48,17] also allow expressing dependencies among tasks, being the run-time system responsible of the dependencies to be satisfied before spawning dependent tasks. Task-based models can be dynamically managed by mapping the tasks to threads in a thread pool, e.g., using the popular Work-Stealing algorithm [10]. Yet, sources of non-determinism at run-time cause timing divergences among threads. Dynamic schedulers try to compensate by detecting them at run-time [11,45] and either (i) "re-molding" into more threads on-the-fly; (ii) boosting relative priorities, or (iii) adapting the mapping and number of allocated processing units. Since performance is the major goal, mapping strategies are mostly dynamic in nature, and, although being able to provide better average behavior, they may allow for unpredictable unbounded delays.

OpenMP has been already considered as a convenient interface to describe real-time applications [30,4]. However, the easy-to-use and well-known OpenMP directives have been used as a mere programming frontend to describe a taskgraph. Traditional timing-analysis and scheduling techniques are then applied to this graph, neglecting the semantics of the OpenMP execution model and bypassing the functionality of the runtime system. Moreover, the focus has typically been on the most traditional loop-centric OpenMP specification v2.5, which is limited to a standard fork-join type of parallelism and does not take advantage of the way more expressive *tasking* interface [47].

Agrawal et al. [4] proposed the RT-OpenMP framework as a means to provide timing guarantees within the OpenMP programming model [47], introducing a scheduler of (RT-) tasks for a generic multi-core system. P-SOCRATES has a more "general" approach to timing analyzability and schedulability, taking a holistic view of the whole system, analyzing the contribution of each system component, either hardware (cores, memory) or software (task, RT-task) and composing them together to build *a fully integrated technological stack* to provide timing guarantees in a *generic* many-core system.

In the real-time community, **scheduling techniques** have been the subject of extensive research. Traditional techniques have been extended for the multiprocessor case and more recently for parallel execution (e.g. [31,53,6,42]). After the majority of the works considering 1-to-1 mappings, where each parallel execution is mapped to a thread, new models are appearing with more complex mapping approaches. Different mappings of parallel tasks to threads can be done, mostly statically [53,6] but also dynamically [42], in order to increase system utilization while maintaining predictability. These strategies need however to be extended for exploiting the dependency graphs from compiler generated parallel task graphs.

Static approaches for timing analysis typically infer timing properties from mathematical models and logical abstractions (e.g., [2]), while measurement-based techniques exploit the results of extensive simulations to derive worst-case estimations. Hybrid techniques combine features from both static and measurement-based approaches while avoiding (as much as possible) their respective pitfalls (e.g. [52]). In what concerns schedulability analysis, different tests have been proposed for various kinds of workload and platform models, and for different scheduling algorithms. In its simplest form, a schedulability test is just a mathematical condition such that, if the condition is satisfied, then the system is deemed schedulable, i.e., all the deadlines will always be met at run-time. Unfortunately, there are still many open problems and NP-hard issues in the schedulability analysis of multi-core systems [7]. Furthermore, timing and schedulability analysis cannot be taken in isolation from the mapping approach, since the mapping of tasks to particular cores clearly impacts the timing analysis.

The idea of holistically orchestrating **memory transfers** in a multi- or many-core platform is not completely new in the domains targeted by P-SOCSATES. However, traditionally, memory transfers are used to improved average performance of the system, rather than improving its worst-case behavior and making it more predictable. These *data prefetching* techniques are well known both in HPC and embedded computing domains, but also in the general purpose world. Data caches are a nice example where prefetching is automatically driven in hardware: a full set of data (a cache line) is loaded as a single data item contained in it is explicitly accessed. A very interesting recent study [32] tries to draw some "guiding lines" on the usage of prefetching to achieve an actual performance gain. However, no timing guarantees are provided, nor timing predictability is among the goals of the most advanced prefetching techniques. Some works by Pellizzoni et al.[50] are – to the best of our knowledge – the unique trying to schedule real-time tasks in a multi-core environment using a "memory-centric approach". Up to now, they targeted cache-based systems, with a single or limited set of general purpose cores, in some cases including *ad hoc* hardware controllers for controlling the accesses to memory banks.

7. Conclusions

There is currently an increasing interest in the convergence of high-performance and embedded computing domains. Not only new high-performance applications are being required by markets needing huge amounts of information to be processed within a bounded amount of time, but also embedded systems are increasingly concerned with providing higher performance in real-time, challenging the performance capabilities of current architectures. Meeting this dual challenge can only be provided by next-generation many-core embedded platforms, guaranteeing that real-time high-performance applications can be executed on efficient and powerful heterogeneous architectures integrating general-purpose processors with many-core computing fabrics.

This paper proposes a novel approach to address time-criticality and parallelisation by an integrated framework for executing workload-intensive applications with real-time requirements on top of next-generation COTS platforms based on many-core accelerated architectures. The framework devised in the P-SOCRATES FP7 project [57], addresses the problem from both the HPC and real-time computing domains, integrating the extraction of task dependency graphs with timing information from the applications code, with real-time mapping and scheduling algorithms, along with the associated timing and schedulability analysis. The main outcome of P-SOCRATES is a software stack 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.

References

[1] J. Engblom, A. Ermedahl, Execution time analysis for embedded real-time systems, Handb. Real-Time Embed. Syst. (January) (2007).
[2] AbsInt Corp, aiT WCET Analyser <http://www.absint.com/ait/>.
[3] Adapteva, Inc., Epiphany-IV 64-core 28 nm Microprocessor, 2013.
[4] Kunal Agrawal, Christopher Gill, Jing Li, Mahesh Mahadevan, David Ferry, Chenyang Lu, A real-time scheduling service for parallel tasks, in: Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), RTAS '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 261–272.
[5] Cédric Augonnet, Samuel Thibault, Raymond Namyst, Pierre-André Wacrenier, Starpu: a unified platform for task scheduling on heterogeneous multicore architectures, Concurr. Comput.: Pract. Exper. 23 (2) (2011) 187–198.
[6] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, A. Wiese, A generalized parallel task model for recurrent real-time processes, in: 2012 IEEE 33rd Real-Time Systems Symposium (RTSS), 2012, pp. 63–72.
[7] Sanjoy Baruah, Kirk Pruhs, Open problems in real-time scheduling, J. Schedul. 13 (6) (2010) 577–582.
[8] Luca Benini, Eric Flamand, Didier Fuin, Diego Melpignano, P2012: building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator, in: EDA Consortium on the Proceedings of the Conference on Design, Automation and Test in Europe, DATE '12, San Jose, CA, USA, 2012, pages 983–987.
[9] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, Krisztián Flautner, Evolution of thread-level parallelism in desktop applications, in: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10, ACM, New York, NY, USA, 2010, pp. 302–313.
[10] Robert D. Blumofe, Charles E. Leiserson, Scheduling multithreaded computations by work stealing, J. ACM 46 (5) (1999) 720–748.
[11] Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla, Mateo Valero, A dynamic scheduler for balancing HPC applications, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 41:1–41:12.
[12] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, Jesús Labarta, Productive cluster programming with ompss, in: Proceedings of the 17th International Conference on Parallel Processing – Volume Part I, Euro-Par'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 555–566.
[13] CompuGreen, LLC, The Green500 List.
[14] Dakshina Dasari, Bjorn Andersson, Vincent Nelis, Stefan M. Petters, Arvind Easwaran, Jinkyu Lee, Response time analysis of cots-based multicores considering the contention on the shared memory bus, in: Proceedings of the 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, TRUSTCOM '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 1068–1075.
[15] B.D. de Dinechin, D. van Amstel, M. Poulhies, G. Lager, Time-critical computing on a single-chip massively parallel processor, in: Design, Automation and Test in Europe Conference and Exhibition (DATE), March 2014, 2014, pp. 1–6.
[16] S.K. Dhall, C.L. Liu, On a real-time scheduling problem, Operations Res. 26 (1) (1978) 127–140.
[17] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, Judit Planas, Ompss: a proposal for programming heterogeneous multi-core architectures, Parall. Process. Lett. 21 (2011) 173–193 (2011-03-01).
[18] T. Ungerer, et al., Merasa: Multi-core execution of hard real-time applications supporting analysability, IEEE Micro 2010, Spec. Iss. Euro. Multi. Process. Proj. 30(5) (2010).
[19] Evidence Srl, Erika Enterprise.
[20] FSF – The GNU Project. GOMP – An OpenMP implementation for GCC.
[21] P. Gai, E. Bini, G. Lipari, M. Di Natale, L. Abeni, Architecture for a portable open source real-time kernal environment, in: 2nd Real-Time Linux Workshop, 2000.

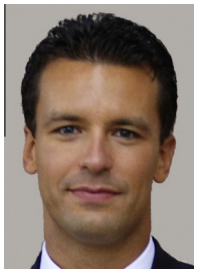
[22] P. Gai, F. Esposito, R. Schiavi, M. Di Natale, C. Diglio, M. Pagano, C. Camicia, L. Carmignani, Towards an open source framework for small engine controls development, in: SAE/JSAE 2014 Small Engine Technology Conference & Exhibition, 2014.
[23] Ricardo Garibay-Martínez, Luis Lino Ferreira, Cláudio Maia, Luis Miguel Pinho, Towards transparent parallel/distributed support for real-time embedded applications, in: 8th IEEE International Symposium on Industrial Embedded Systems, 2013.
[24] Mike Houston, Ji-Young Park, Manman Ren, Timothy Knight, Kayvon Fatahalian, Alex Aiken, William Dally, Pat Hanrahan, A portable runtime interface for multi-level memory hierarchies, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, PPoPP '08, ACM, New York, NY, USA, 2008, pp. 143–152.
[25] Intel Corp, Array Building Blocks <http://software.intel.com/en-us/articles/intel-array-buildingblocks>.
[26] Intel Corporation, Threading Building Blocks, 2006.
[27] Intel Corporation, Intel Many Integrated Core (MIC) Architecture <http://www.intel.com/content/www/us/en/architecture-and-technology/many-integratedcore/intel-many-integrated-core-architecture.html> (access November 2013).
[28] Intel Corporation, Intel Xeon Phi <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html> (access November 2013).
[29] Kalray Corporation, Kalray MPPA 256 <http://www.kalray.eu/products/mppa-manycore/> (access November 2013).
[30] K. Lakshmanan, S. Kato, R. Rajkumar, Scheduling parallel real-time tasks on multi-core processors, in: 2010 IEEE 31st Real-Time Systems Symposium (RTSS), November 2010, pp. 259–268.
[31] K. Lakshmanan, S. Kato, R. Rajkumar, Scheduling parallel real-time tasks on multi-core processors, in: 2010 IEEE 31st Real-Time Systems Symposium (RTSS), 2010, pp. 259–268.
[32] Jaekyu Lee, Hyesoon Kim, Richard Vuduc, When prefetching works, when it doesn't, and why, ACM Trans. Archit. Code Optim. 9 (1) (2012) 2:1–2:29.
[33] David C. Luckham, The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 2001.
[34] T. Lundqvist, P. Stenstrom, Timing anomalies in dynamically scheduled microprocessors, in: Proceedings of the 20th IEEE Real-Time Systems Symposium, 1999, pp. 12–21.
[35] Cláudio Maia, Luis Miguel Nogueira, 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.
[36] José Marinho, Vincent Nélis, Stefan M. Petters, Marko Bertogna, Robert Davis, Limited pre-emptive global fixed task priority, in: 34th IEEE Real-Time Systems Symposium, 2013.
[37] José Marinho, Vincent Nélis, Stefan M. Petters, Isabelle Puaut, Preemption delay analysis for floating non-preemptive region scheduling, in: Design, Automation and Test in Europe Conference and Exhibition, 2012, pp. 497–502.
[38] José Manuel Marinho, Stefan M. Petters, Marko Bertogna, Extending fixed task-priority schedulability by interference limitation, in: Proceedings of the 20th International Conference on Real-Time and Network Systems, RTNS '12, ACM, New York, NY, USA, 2012, pp. 191–200.
[39] Massachusets Institute of Technology, The Cilk Project, 1998.
[40] Mont-Blanc FP7 European Project, grant agreement 288777, 2011–2014 <http://www.montblanc-project.eu/>.
[41] Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell, Pthreads Programming, O'Reilly & Associates Inc., Sebastopol, CA, USA, 1996.
[42] L. Nogueira, J.C. Fonseca, C. Maia, L.M. Pinho, Dynamic global scheduling of parallel real-time tasks, in: 2012 IEEE 15th International Conference on Computational Science and Engineering (CSE), 2012, pp. 500–507.
[43] Luis Miguel Nogueira, Luis Miguel Pinho, José Fonseca, Cláudio Maia, On the use of work-stealing strategies in real-time systems, in: High-Performance and Real-Time Embedded Systems (HiRES), 2013.
[44] NVIDIA Corporation, NVIDIA CUDA Compute Unified Device Architecture, Version 2.0, 2008.
[45] Stephen L. Olivier, Allan K. Porterfield, Kyle B. Wheeler, Michael Spiegel, Jan F. Prins, Openmp task scheduling strategies for multicore numa systems, Int. J. High Perform. Comput. Appl. 26 (2) (2012) 110–124.
[46] OpenCL, The Open Standard for Parallel Programming of Heterogeneous Systems, 2013 <http://www.khronos.org/opencl/>.
[47] OpenMP Architecture Review Board (ARB), OpenMP Architectural Review Board, OpenMP Application Program Interface, Version 3.1, July 2011 <http://www.openmp.org>.
[48] OpenMP Architecture Review Board (ARB), OpenMP Architectural Review Board, OpenMP Application Program Interface, Version 4.0, July 2013 <http://www.openmp.org>.
[49] parMERASA FP7 European Project – grant agreement 287519. Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability, 2011–2014 <http://www.parmerasa.eu>.
[50] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, 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.
[51] J.M. Perez, P. Bellens, R.M. Badia, J. Labarta, Cellss: Making it easier to program the cell broadband engine processor, IBM J. Res. Develop. 51 (5) (2007) 593–604.
[52] Rapita Systems Corp, RapiTime <http://www.rapitasystems.com>.
[53] A. Saifullah, K. Agrawal, Chenyang Lu, C. Gill, Multi-core real-time scheduling for generalized parallel task models, in: 2011 IEEE 32nd Real-Time Systems Symposium (RTSS), 2011, pp. 217–226.
[54] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Pat Hanrahan, Larrabee: a many-core x86 architecture for visual computing, ACM Trans. Graph. 27 (3) (2008) 18:1–18:15.
[55] Herb Sutter, Welcome to the Jungle <http://herbsutter.com/welcome-to-the-jungle/>.
[56] Texas Instrument Inc., The Keystone Processor.
[57] The P-SOCRATES Consortium, P-SOCRATES (Parallel Software Framework for Time-Critical Many-core Systems) <http://p-socrates.eu>.
[58] R. Tieman, Algo Trading: The Dog That Bit its Master, Financial Times, 2008. March.
[59] Tilera Corporation, Tile Processor, User Architecture Manual, release 2.4, DOC.NO. UG101, May 2011.
[60] R. Vargas, E. Quiñones, A. Marongiu, OpenMP and timing predictability: a possible union? in: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE '15, pp. 617–620.
[61] Sandra Wienke, Paul Springer, Christian Terboven, Dieter an Mey, Openacc: first experiences with real-world applications, in: Proceedings of the 18th International Conference on Parallel Processing, Euro-Par'12, 2012, pp. 859–870.
[62] Henry Wong, Anne Bracy, Ethan Schuchman, Tor M. Aamodt, Jamison D. Collins, Perry H. Wang, Gautham Chinya, Ankur Khandelwal Groen, Hong Jiang, Hong Wang, Pangaea: a tightly-coupled ia32 heterogeneous chip multiprocessor, in: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT '08, 2008, pp. 52–61.

Luís Miguel Pinho, Ph.D. (2001), is Professor at the Polytechnic Institute of Porto, and Vice-Director and Research Associate at the Research Centre in Real-Time and Embedded Computing Systems (CISTER), where he currently leads activities in, among others, real-time parallel models. He has participated in more than 15 R&D projects, being Coordinator of the FP7 R&D European Project and CISTER coordinator and work package leader in several other European projects. He has published more than 100 papers in international venues and journals in the area. He was Keynote Speaker at RTCSA 2010, Conference Chair and Program Co-Chair of Ada-Europe 2006, Program Co-Chair of Ada-Europe 2012 and General Co-Chair of ARCS 2015.

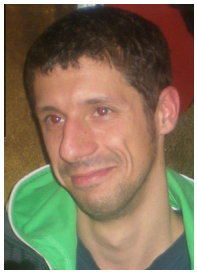
Vincent Nélis received his Ph.D. degree in Computer Science at the University of Brussels (ULB) in 2010. Since then, he has been working as a Research Associate at CISTER-ISEP Research Unit in Porto, Portugal. His research is centered around the real-time scheduling theory, with a focus on multiprocessor/multicore systems, and in execution time and interference analysis.

Patrick Meumeu Yomsi received his Ph.D. in 2009 from the Paris-Sud University, Orsay in France. Prior to joining the CISTER Research Center as a Research Associate in 2012 in Porto, Portugal, he was successively a member of the AOSTE research unit at the French National Institute in Computer Science and Control (INRIA) in Paris Rocquencourt, France, the PARTS Research Unit at the University of Brussels (ULB) in Brussels, Belgium, and finally a member of the TRIO Research Unit at INRIA in Nancy, France. His research interests include real-time scheduling theory, real-time communication and real-time operating systems.

Eduardo Quiñones is a senior researcher in the Department of Computer Science at the Barcelona Supercomputing Center (BSC). He received his Ph.D. in computer science from the Universitat Politècnica de Catalunya (UPC) in Barcelona in 2008. His research interests are strongly tied to next generation industry requirements for critical real-time systems spanning future many-core processor architecture, operating system and compiler designs.



Marko Bertogna is Associate at the University of Modena, Italy. He previously was Assistant Professor at the Scuola Superiore Sant'Anna of Pisa, Italy, where he also received (cum laude) a Ph.D. in Computer Engineering.
He has authored over 60 papers in international conferences and journals in the field of real-time and multiprocessor systems, receiving six Best Paper Awards and one Best Dissertation Award. He served in the Program Committees of the major international conferences on real-time systems.



Paolo Burgio got a M.S. degree in Computer Engineering from the University of Bologna in 2007, and a Ph.D. in Electronics Engineering jointly between the University of Bologna and the University of Southern-Brittany, in 2013. He now holds a post-doc position at University of Modena and Reggio Emilia. His main research interests are embedded many-core architectures and programming models, heterogeneous architectures, HLS, virtual platforms, and real-time systems.
He published several papers in top-level conferences and journals. He took part at the development of the VirtualSoC many-core simulator, and was involved in FP7 projects PREDATOR, Pro3D, vIrtical, and (currently) P-SOCRATES. He is currently performing research on predictable many-core architectures for next-generation real-time systems.

Andrea Marongiu received the Ph.D. degree in electronic engineering from the University of Bologna, Italy, in 2010. He currently is a postdoc researcher at ETHZ, Zurich. He also holds a postdoc position at University of Bologna. His research interests concern parallel programming model and architecture design in the single-chip multiprocessors domain, with special emphasis on compilation for heterogeneous architectures, efficient usage of on-chip memory hierarchies and SoC virtualization.

Claudio Scordino obtained Master Degree and Ph.D. in Computer Engineering from the University of Pisa in 2003. In 2007 he obtained the Ph.D. from the same university, with a thesis about power-aware real-time scheduling. His research activities include operating systems, real-time scheduling, energy saving and embedded devices. He collaborates with the Linux kernel community since 2008, having several patches integrated in the official Linux kernel. He currently works as Project Manager at Evidence Srl.

Dr. Paolo Gai, CEO, graduated (cum laude) in Computer Engineering at University of Pisa in 2000 with a graduation thesis developed at the ReTiS Laboratory of the Scuola Superiore Sant'Anna on the development of the modular real-time kernel SHaRK. He obtained the Ph.D. from Scuola Superiore Sant'Anna in 2004. Since 2000, he founded the ERIKA Enterprise project, an open-source RTOS which recently reached the OSEK/VDX certification, and which is currently used by various industries and universities. Since 2002 he is CEO and founder of Evidence Srl, a SME working on operating systems and code generation for Linux- and ERIKA- based industrial products in the automotive and white goods market. His research interests include development of hard real-time architectures for embedded control systems, multi-processor systems, object-oriented programming, real-time operating systems, scheduling algorithms and multimedia applications.

Michele Ramponi did his master degree (Ing.) in 2000 at Universita Degli Studi di Ferrara – Italy, Faculty of Electronic Engineering. In 2003 he founded Active Technologies in cooperation with other two business partners and the company was focused in since the beginning in high performance test and measurement development, signing contracts with the most important players on the market.
In Active Technologies Michele has been involved in several FP7 R&D European Project in semiconductor domain as well as high performance parallel computing hardware development.

Michal Mardiak did his master degree (Ing.) in 2009 and Ph.D. in 2012 in Slovak University of Technology in Bratislava, Faculty of Electrical Engineering and Information Technology. He has joined the ATOS SA in 2013 after several years working in mobile netwrok research focusing on LTE. In ATOS SA he has been involved in several FP7 R&D European Project in IoT domain as well as parallel programming area.