A New Configurable and Parallel Embedded Real-time Micro-Kernel for Multi-core platforms

Antonio Paolillo, Olivier Desenfans, Vladimir Svoboda, Joël Goossens, Ben Rodriguez

PARTS Research Center, Université Libre de Bruxelles, Mangogem S.A. and HIPPEROS S.A.
Corresponding author: antonio.paolillo@ulb.ac.be

***Abstract*—One of the main on-going initiatives of the *PARTS Research Center* together with *HIPPEROS S.A.* is the creation of a new Real-Time Operating Systems family called *HIPPEROS*. This paper focuses on the design and the implementation of its new real-time multi-core micro-kernel. It aims to address the challenge of efficient management of computing resources for competing real-time workloads on modern MPSoC platforms while maintaining the level of assurance and reliability of existing production systems. The objective of this paper is to present an overview of its inner architecture.**

I. Introduction

For the past twenty years, real-time theory has widely explored the possibility to use multi-core and many-core platforms for embedded systems. However, while this topic seems to be very mature in the literature, the safety-critical software industry still relies on uni-core techniques for operating system implementations. The industry state-of-the-art regarding multi-core platforms is to fully separate the processing resources in time and space isolated partitions with very few possible communication channels between the different partitions and therefore to consider that each partition operates as an independent uni-core platform. Examples of de facto standards for these techniques are ARINC653 [1] and AUTOSAR [2].

While from a conservative point of view these are the most reliable and predictable solutions, these are not the best available options in the real-time research w.r.t. efficiency, resource utilisation and cost. Moreover, the rising demand of computational power per silicon area happening in every domain — including safety-critical systems — puts a pressure on the low-level, middleware and kernel developers to implement efficient policies for real-time process management.

In recent years, efforts in the research community have been made to adapt existing general-purpose kernels such as Linux in order to provide a real-time execution environment suitable for the evaluation of efficient multi-core real-time scheduling and resource allocation policies [3], [4]. The main advantage of this approach is to reuse the existing kernel code base which has been tested and validated by millions of users worldwide. However this approach cannot be directly exploited in production systems. Indeed Linux is not originally intended nor designed to support neither *hard* real-time constraints nor safety-critical applications. Moreover it is not conforming to the highest demanding certification standards of this industry such as DO-178-B (level A, B) or ISO26262. As a consequence, the latest real-time multi-core algorithms have not been tested in a strict and realistic hard real-time environment yet. As stated by Brandenburg in [5]:

> Ideally, [...], worst-case kernel overheads [...] should be determined analytically. However, for the foreseeable future, this will likely not be possible in complex kernels such as Linux. Instead, it would be beneficial to develop (or extend existing) μ-kernels of much simpler design with LITMUS^{RT}-like functionality.

This kind of kernel would have to be built from the ground up with hard real-time and multi-core constraints integrated as parts of its base design principles. This would allow for simpler, finer-grained measurements of the overheads introduced by different implementations of the various solutions the literature has to offer. Moreover the architecture of this kernel must scale with an increasing number of cores to allow execution on many-core platforms.

In order to address the challenge of providing efficient multi-core kernel implementations while still providing the same level of assurance and reliability of the existing industry quality standards, the *PARTS Research Center*, together with the company *MangoGem S.A.*, launched the *HIPPEROS* project in 2010. The development of the *HIPPEROS* kernel started in June 2013. *HIPPEROS* aims to provide a family of RTOS solutions, each adapted specifically to the different needs of the real-time system designer and including the implementation of the latest results of the research community. It stands for *HIgh Performance Parallel Embedded Real-time Operating Systems*.

II. System overview

We started the project by developing a new kernel from scratch running as a bare-metal system on ARM and x86 systems. The objective is to have a fully configurable kernel, running transparently on different architectures and platforms with an arbitrary number of cores, that will be the seed of the different RTOS solutions mentioned above. With such a flexible design it would be possible to deeply explore the practicability of real-time theory solutions. To reach this the kernel has the following design characteristics:

- for scalability reasons, it has a distributed asymmetric micro-kernel architecture, meaning that each core can execute a local part of the kernel (the lightweight and very local operations like simple system calls or process context switching), while a dedicated core executes the heavy parts of the kernel (complex system calls, scheduling decisions, shared resources handling, etc), allowing to execute several parts of the kernel *in parallel*; to the

best of our knowledge, this kernel design approach is very rare for real-time systems although it is already used in high performance computing and scalable non real-time kernels [6]–[8];
- it is configurable at build-time to efficiently suit the different needs of the system designer or application developer; e.g. the scheduling policy or the resource allocation protocol for real-time processes can be chosen at build-time; notice that only the chosen policies will be embedded in the production executable binary image of the kernel (mainly for code size reasons);
- to manage hard real-time workloads, it implements the popular process model used in the real-time scheduling research literature: the concept of periodic and sporadic tasks generating jobs to schedule with a finite time budget and deadline.

By combining the available configuration options, the *HIPPEROS* build system is able to generate a large variety of RTOS solutions, ranging from a low-overhead statically linked run-time executive implementing the simple rate monotonic scheduling policy [9] to a full fledged micro-kernel based operating system supporting several independent ELF applications with memory isolation between processes, inter-core message passing IPC and optimal scheduling policies.

This distributed and highly configurable kernel supporting real-time workloads aims to provides both a productive system to industry application designers and an experimental software platform to real-time researchers. The goal is to test, validate and run into production low-overhead energy efficient hard real-time systems running on modern embedded multi-core platforms with different instruction set architectures.

III. Process model

To derive straightforward implementations of state-of-the-art algorithms, we chose to faithfully interpret the task model w.r.t. real-time scheduling theory. We map the popular task model of real-time literature [9] to the internal HIPPEROS process abstraction. More specifically, we implemented *constrained deadline sporadic and periodic tasks*.

A set of tasks is statically registered to the kernel. Each task is configurable by providing the following information: an executable program and timing information (sporadic/periodic, offset, deadline, period and worst-case execution time). Time unit for these values is the *number of kernel ticks*, a configurable atomic time period. At kernel initialisation time, the process manager module registers one process for each of these tasks and configures it according to the task parameters.

The *scheduler API* is preemptive and priority-based: each time a process changes state, the scheduler module is called to decide if some process context switches must occur according to their priority. If a real-time process overruns its associated task's WCET or misses its deadline, a configurable policy is applied. It could be that the process is killed (the reason being the non-respect of its contract with the kernel), the event ignored or the priority of the process changed.

These simple mechanisms allow to easily implement and evaluate theoretical multi-core scheduling algorithms (like RUN [10], U-EDF [11] or power- and thermally-aware algorithms) and the associated resource allocation protocols. The model could be easily extended in the future to support mixed-criticality tasks: it would require vectorial timing information rather than scalars.

IV. Asymmetric kernel architecture

A recurring problem in kernel design for multi-core platforms is how to distribute the privileged work amongst the different processing resources. Usual implementations like Linux use a symmetric design, where each core goes through the same kernel code and protect data structures with fine-grained lock mechanisms. However, this approach can lead to *kernel serialisation*, meaning that each kernel thread is actually executed sequentially (each waiting for the completion of one other) and has been proven not to scale with an increasing number of cores [6], [12]. Furthermore, in [12], Cerqueira *et al* suggest an asymmetric distribution of the work, where one core has the responsibility to execute the scheduler and dispatches the processes to the other cores through message passing.

We adopted a similar solution in the *HIPPEROS* kernel design: a designated core called the *master core* is responsible for managing the global resources, keeping a coherent state of the system and calling the scheduler to decide which process has to be *preempted* or *dispatched*. We went further than [12] by implementing this design not only for the scheduler but also for system calls and process message passing mechanisms. It allows the kernel to be executed in *parallel*.

The principle is the following: each time a scheduling decision has to be made (e.g. a process changes state), the *master core* must be woken up. When the *master core* has to notify another core (called *slave core*) that it has to execute a context switch (process preempted or dispatched), the *master* sends a software-generated inter-processor interrupt (IPI) to the slave core to notify it of the changes. When a process executing on a slave core calls a system call that may impact scheduling, the *remote system call mechanism* is used. The slave part of the kernel serialises the system call arguments, triggers an IPI to the master and goes back to user mode to execute a busy loop waiting for the response of the master part of the kernel. Notice that this busy loop is process-specific, executed in user mode and can be interrupted by a context-switch request of the *master core*.

In opposition to the symmetric approach, this master/slave kernel architecture requires *almost* no locking mechanism as the system's global state must not be shared and is only visible by the *master core*.

To correctly implement the system calls and the context switches, some small data structures are shared between the *master* and each slave. These data structures are currently protected with mutexes, and wait-free data structures are considered to be integrated for the foreseeable future. Notice that as the contention on these data structures is limited by the process-to-kernel interactions, several slave cores require distinct mutexes. Therefore, the peak contention of the concurrency mechanisms is low. In the long term, our goal is to be able to predictably bound this contention. As the shared data structures between *master* and *slaves* are limited to what is necessary for system calls and context switches and the rest of the system state (e.g. scheduler data structures) is

maintained only by the *master*, we also expect to have limited performance-degrading cache-line bouncing.

The inter-process communication (IPC) scheme is built on top of this master-slave RPC mechanism. We support two different API for IPC: the *Copy buffer IPC* (CB-IPC), where the message is copied from the sender buffer to the receiver buffer and the *Zero copy IPC* (ZC-IPC), where a page is shared between the sender and the receiver (no copy is then performed when passing the message). When a process calls the *send* or *receive* system calls, the *master core* is warned through an IPI to update the process states accordingly. However, in case of CB-IPC, the message is copied locally by the slave to avoid overloading the master with memory operations.

We expect this approach to scale up to 8 cores of the embedded platform. More cores contacting the *master* would eventually overload it, resulting in a situation where some running processes have to wait for the execution of all the system calls of the processes executing on the other cores. For more cores (e.g. for execution of HIPPEROS on a many-core platform), we foresee the usage of techniques like clustering, where several independent micro-kernel instances would be executed in parallel, like the Helios Satellite Kernels [8]. Each parallel kernel would be responsible of a subset of the platform processing cores with independent scheduling, memory management and process message passing. Processes on different clusters that want to communicate would use a dedicated inter-kernel communication channel. This mechanism still needs to be implemented and evaluated.

V. Kernel configurability

As *HIPPEROS* targets deeply embedded systems, the majority of options is configured at build-time to suit the specific requirements of the embedded software. Policies and components must then be chosen at build-time.

One of the goals of the kernel is to be portable across a large variety of architectures and platforms. The kernel currently supports ARM and x86 architectures. There is a wide variety of hardware platforms implementing these architectures and these targets can have very different levels of complexity and features. For example, a Memory Management Unit (MMU) can be present or not on a given platform. Therefore, the kernel must be configurable to the point of presenting several memory models, according to the presence or absence of a MMU. It is necessary to provide a MMU-free memory model as some of the critical embedded platforms used in production today are still MMU- and cache-free.

The scheduling policy (Partitioned-RM, Global-EDF, etc.) in place is also a modular component that is chosen at kernel build-time. For energy efficiency reasons, the number of cores of the target platform actually used can be configured too: the user could decide to only use a subset of the resources available on the target platform. Moreover, the set of cores could be shared between several operating systems (several *HIPPEROS* instances as mentionned in section IV or other OSes). Therefore, decide which cores are used or not will allow *HIPPEROS* to be suited for mixed-criticality environment with space partitioning: the execution of several OSes with various levels of criticality on top of hypervisor software.

VI. Conclusion

In this paper, we introduced a new configurable kernel designed for embedded multi-core platforms. In opposition to the traditional research approaches, our kernel is written from scratch and explores new ways of distributing privileged work among the different cores of the platform by relying on its asymmetric architecture. System reliability is enforced by design using a micro-kernel architecture. By implementing scalable policies inside the kernel, it will be adapted to modern and future multi-/many-core platforms.

To enable straightforward implementation of existing real-time scheduling strategies, we faithfully implemented the literature task model: periodic and sporadic jobs with a limited execution budget and a deadline.

Thanks to the high level of configurability and modularity built in the kernel by design, we expect to provide a new benchmarking platform to the research community.

Future developments will involve the integration of the *HIPPEROS* RTOS in mixed-criticality environments where a RTOS running highly critical workloads can be executed in parallel with a general purpose OS like Linux to make an effective usage of the modern MPSoC platforms.

A free academic license of the product will be available for distribution. The RTOS is now being validated for various use cases within the ARTEMIS CRAFTERS project by the *PARTS Research Center* and *MangoGem S.A.*. This work is supported by the Innoviris grant RBC/12 EUART 2a. The kernel is used in industrial Proof of Concept projects by *HIPPEROS S.A.*, which further develops it into a full-blown certifiable RTOS. *HIPPEROS* is a registered trademark of *HIPPEROS S.A.*.

References

[1] *Avionics Application Software Standard Interface*, Airlines Electronic Engineering Committee, Aeronautical Radio INC, June 2013.

[2] *Guide to Multi-Core Systems*, AUTOSAR, March 2014.

[3] J. M. Calandrino, H. Leontyev, A. Block, U. Devi, and J. H. Anderson, "LitmusRT: A testbed for empirically comparing real-time multiprocessor schedulers," in *27th IEEE Int. Real-Time Systems Symposium*, 2006.

[4] D. Faggioli, M. Trimarchi, F. Checconi, M. Bertogna, and A. Mancina, "An implementation of the earliest deadline first algorithm in Linux," in *24th Annual ACM symposium on Applied Computing*, 2009.

[5] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina, 2011.

[6] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): The case for a scalable operating system for multicores." SOSP, 2009.

[7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems." SOSP, 2009.

[8] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: Heterogeneous multiprocessing with satellite kernels." SOSP, 2009.

[9] J. W. S. Liu, *Real-Time Systems*. Prentice Hall, 2000.

[10] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *IEEE 32nd Real-Time Systems Symposium*, Nov. 2011.

[11] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic, "U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," in *ECRTS*, 2012.

[12] F. Cerqueira, M. Vanga, and B. Brandenburg, "Scaling global scheduling with message passing," in *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2014.