A dedicated micro-kernel to combine real-time and stream applications on embedded manycores

P. Dubrulle, E. Ohayon

Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 1634 – 1643

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

A Dedicated Micro-Kernel to Combine Real-Time and Stream Applications on Embedded Manycores

Paul Dubrulle[a,*], Emmanuel Ohayon[a]

[a]CEA, LIST, Real-Time Embedded Systems Laboratory, Nano-Innov / Saclay, 91191 Gif-sur-Yvette Cedex, France

Abstract

The latest generation of many-core processors offers more than ever the opportunity to pool different applications into a single embedded system. This opportunity however depends on the ability to provide safety guarantees, especially when it comes to embedded life- or mission-critical applications.

For that matter, we introduce a new multi-task preemptive micro-kernel for many-core architectures called *Psigma*. This micro-kernel is able to run simultaneously and safely tasks written with very different programming paradigm, and very different execution requirements: hard real-time applications and stream applications. This paper shortly presents both programming models, then focuses on the design and performances of the micro-kernel.

Keywords: many-core, multi-core, micro-kernel, embedded systems, stream programming, real-time, instrumentation & control

1. Introduction

Multi-core architectures progressively become a standard in industrial embedded systems – even safety-critical ones. Naturally following this trend, many processor manufacturers are now releasing *many-core* chips, that gather from dozens to hundreds of cores on a single device, with a high performance-to-power ratio. Within the next few years, a majority of embedded systems solutions will most likely rely on such processors. However, even more than with multi-processors, the issue of the programmability of such massively parallel architectures has become crucial. It may be addressed in several ways: by using adapted programming models or, under appropriate safety requirements, by mixing multiple independent legacy applications on the chip, possibly with different levels of criticality. The work exposed in this paper is actually an attempt to implement both approaches.

1.1. Motivations and objectives

CEA LIST has developed two programming languages for embedded platforms, with quite different purposes: the ΣC programming language [1], to implement massively parallel data flow applications (mainly targeting many-core chips); and the ΨC programming language [2], to implement multi-task hard real-time applications that require a high level of safety. Each of these languages comes with a source-to-source C compiler, runtime generation

*P. Dubrulle. Tel.: +33-1-69-08-00-61
E-mail address: firstname.lastname@cea.fr.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.331

tools, and a dedicated execution support. The underlying task models offer functional determinism at execution, as well as guarantees on task behavior that make cohabitation easier, as detailed in this paper.

This paper presents the design and implementation of a *single* micro-kernel, that is able to run simultaneously ΣC and ΨC applications (see fig.1.1). This new micro-kernel, called "Psigma-kernel", offers an elegant and efficient way to run mixed-criticality applications with a high level of safety, and yet make the most of a massively parallel architecture.

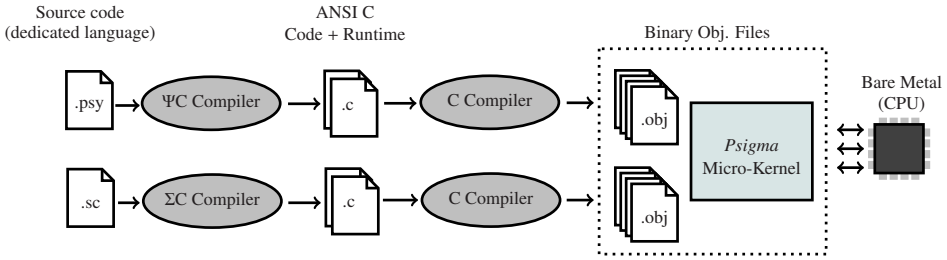


Fig. 1. The Psigma kernel in the Ψ and Σ compilation tool-chains

The motivations for developing a dedicated micro-kernel to support both execution models, rather than user-mode libraries relying on general-purpose operating systems, were twofold. The first is *safety*, as running hard real-time applications with a high level of safety requires a complete control of low-level mechanisms (such as interrupts and timers), especially to implement strict scheduling policies. The other is *performances*, since especially on embedded platforms, bare metal system programming is necessary to achieve a high level of both raw computing performances and energy efficiency.

1.2. Target architectures

The choice of the target architecture is of first importance regarding the design of a specialized micro-kernel. We choose to target multi-core systems with shared on-chip memory. This memory can either be a specialized local storage shared among several cores, or be a shared L2 or L3 on-chip cache. We also assume that the platform provides inter-core *interrupts*, and *events* synchronization instructions [1].

A specialized micro-kernel targeting such architectures is scalable to embedded many-core chips, most such architectures being clustered (hence one instance of the micro-kernel runs on each cluster, which typically is a small SMP - cf. Figure 2). When it is not so, the set of cores can be partitioned.

Note also that shared memory architectures allow for a global and dynamic scheduling of the tasks, which is essential to combine the task models that we chose, as we will see in sections 3 and 4.

2. Related works

Our implementation relies on a single, efficient micro-kernel able to run two specific execution models: one in hard real-time, the other in best-effort. This micro-kernel implements a sparing scheduling policy, that fully uses the "idle" CPU time left by real-time tasks to run the stream tasks. To the best of our knowledge, this is an original approach, and the first one to target many-core embedded chips. Our work is essentially inspired from separate research topics, from the Real-Time and from the Embedded Operating Systems communities.

The idea of allocating a subset of the processors in a SMP system in order to increase responsiveness was proposed by Brosky & Rotolo in [3]; they called this technique "shielding", and implemented it in a Linux kernel to reduce the latency for soft real-time applications. This solution is wasting CPU time though, when the real-time tasks are not fully consuming their allocated execution resources, which is almost always true – especially with

[1] Events are sleep-state hardware synchronization barriers. This energy-efficient mechanism is especially adapted to embedded multi-core platform. If it is not provided by the hardware though, it may easily be implemented with busy-waiting locks, or even with inter-processors interrupts

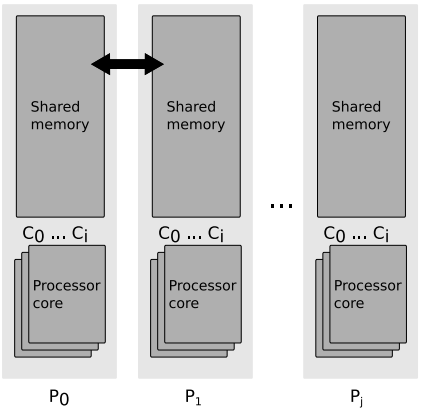


Fig. 2. A multi-core micro-kernel is scalable to many-core by partitioning the many-core and executing an instance of the micro-kernel per partition; each multi-core partition P_0 to P_j hosts an instance of the micro-kernel, the double arrow represents the need of communication between partitions (e.g. partitions could be clusters and the double arrow a NoC for a hierarchical many-core target).

"pessimistic" WCETs[2]. Adding an efficient load-balancing policy while keeping guarantees of low latencies for the real-time tasks helps to partly compensate this waste, as it is done with ARTiS [4]. This last approach has some similarities with ours, but is less flexible. Notably, at most one hard real-time task can run on a shielded core, without migration nor preemption, and the scheduling policy is restricted to fixed-priority. Besides, the programming model is based on usual POSIX and Linux APIs, which clearly shows advantages regarding legacy softwares, but also brings an inherent unpredictability.

Vestal [5] was one of the first to propose a formal approach for using spare CPU time when running tasks with mixed-criticalities, regardless of the underlying platform. For that purpose, compositional or hierarchical scheduling are now active research topics in the Real-Time community [6, 7, 8], and aim at providing safe policies for running heterogeneous tasks on a single system, e.g. hard real-time, soft real-time, and best effort. Note that these work usually attempt to comply with the safety requirements of avionics systems, and most of them come with an implementation prototype as a Linux kernel module or patch, such as LITMUS^{RT} [9]. These works however differ from ours both in their approach and their implementation. Our approach is different because the scheduling policy and the kernel are tightly bounded to dedicated programming models and languages. As for our prototype implementation, we believe that the seek of performances and restrained latency requires a simple, micro-kernel design, whereas the complex monolithic architecture of a Linux kernel makes it unsuitable for hard real-time applications.

A more radical (but non exclusive) way to conceive mixed criticalities is through virtualization: a hypervisor manages the hardware resources, and provides real-time guarantees to one or several of its Virtual Machines (VMs). Real-time extensions were brought to the major hypervisors of the Linux world, namely KVM [10, 11] and Xen [12], but they only support soft real-time tasks. In [13] however, Lee et al. use compositional scheduling in RT-Xen to run some classes of hard real-time tasks. Outside the Linux world, [14] proposes a dedicated real-time micro-kernel with strict resource sharing policies, able to run a para-virtualized Linux host next to hard real time tasks. Also, several proprietary solutions exist such as the real-time hypervisors of National Instruments or of WindRiver; unfortunately only "commercial" documentation is available. Both allow running para- or fully-virtualized general purposes OSes such as Windows or Linux, next to hard real-time bare applications or RTOSes. However the core shielding appears to be strict: no best-effort VM may run on RT-cores. In a general way, virtualization is more flexible but clearly less efficient than our micro-kernel solution, as it supposes managing multiple OSes on a single system. However, the fast evolution of mobile processors may make virtualization viable for embedded real-time systems in the future – especially if dedicated instruction sets like Intel's VT-x or AMD-V are developed.

[2]WCET = Worst Case Execution Time

3. Task Models

This section gives only the basis of the Σ and Ψ programming and execution models, necessary to understand the design of the Psigma kernel. Communications between tasks are handled by wait-free, fully preemptive service libraries, outside of the micro-kernel; therefore they will not be addressed in this paper. Note however that these libraries respectively participate in key properties for both models, which are determinism and reproducibility. For further details, please refer to [2, 1].

3.1. The Σ model

To test the implementation of our micro-kernel, we used the ΣC [1] programing language for the expression of the stream application. In this subsection, we introduce the stream programming paradigm, its relevance to our approach, and the existing execution support we used to build our micro-kernel.

3.1.1. Stream programming

Stream programming relies on Kahn Process Networks (KPN [15]), more precisely on their special derivation, Data Process Networks [16], as well as their more restrictive variants such as Cyclo-Static Data Flows (CSDF [17]). KPN and CSDF are deterministic and the possibility to run a CSDF in bounded memory is a decidable problem [18]. The advantages of stream programming rely for a part in their theoretical bases which make them amenable to formal verification of important application properties like dead-lock freeness, execution within limited memory bounds, or correctness of parallel applications including functional determinism, or absence of race conditions [16]. Even though stream programing is not adapted to all application domains, it is a very good approach for signal and image processing, which are predominant in the embedded applications.

Stream programming is based on the following two elements.

First, a set of *filters* which are computing units that take values as entries on specified read-only channels, use these values for processing and output computation result values on predeclared write-mostly channels. Reading on input channels of filters is blocking. Output channels are theoretically not limited in size, but of course a desired property of the system is that it is amenable to run in finite memory.

Second, a *communication graph* which links either output channels or sources to either input channels or sinks. The communication graph can be quite complex, holding expression of data access patterns including but not limited to permutations, with possible duplication or decimation (without any change to the transfered data).

One possible restriction for stream programs is to conform to the CSDF model, which is sufficient to express complex multimedia implementations [19]. In CSDF, a filter f has in general several input channels and several output channels, and a cyclic execution sequence of N_f functions $[f(0), \ldots, f(N_f - 1)]$. The number of data tokens produced (resp. consumed) for each channel is set per function in the execution cycle. For example, if c is an input channel of f, its intake cycle is defined by a suite of N_f positive integers $[f_0^c, \ldots, f_{N_f-1}^c]$ such that the i^{th} time f executes, it calls the function $f(k)$ and consumes f_k^c tokens on c, with $k = i \mod N_f$.

ΣC defines a superset of CSDF which remains decidable though allowing data dependent control to a certain extent. As the focus of this paper is not set on the programming aspects we will consider that it is limited to CSDF, without loss of generality.

3.1.2. Compilation and execution support

The streaming application is compiled to transform the communication through channels into efficient communication through shared memory-mapped circular buffers. The filters expressed in the language are transformed into tasks that will actually run on a given platform. The shared buffers are the only way to exchange data between tasks (this is formally enforced by an adequate link edition process). The compilation tools are used to ensure that the application liveness property is fulfilled provided that task scheduling is correct.

The correct execution of the system relies on a correct partial order of execution of the compiled tasks so that the circular buffers become a simulation of the channels in the communication graph. The compilation process ensures that tasks preserve the cyclic behavior of the compiled filter or set of filters. For a compiled task, an *activation* is an event in the task's lifetime corresponding to calling one of the CSDF functions $f(i)$.

It is possible from this partial order to generate, per task, a *vector clock* and a *set of vector increments* that can be used at runtime to determine if a task activation can start or must wait [20]. Each task has a *dependency*

counter updated when a task activation ends at runtime. A task activation can start if this counter reaches zero after the update.

This execution model was implemented with a micro-kernel architecture, as detailed in [20]. The only system call is issued at the end of the task activations, for rescheduling on the core executing the calling user context (cf. section 4).

3.2. The Ψ Model

This subsection briefly introduces the ΨC real-time programming model, and the services it requires from the micro-kernel.

3.2.1. Anatomy of a Ψ Application

A Ψ application is a static set of parallel real-time communicating tasks (typically a dozen for an Instrumentation & Control application). A task is a non-terminating chronological succession of *jobs*. A job is a portion of user code that requires to be executed within a temporal window, i.e. no sooner than a *release time* and no later than a *deadline*. The deadline of a job always matches the start time of the succeeding job; besides, the Ψ programming model does not require the windows of all jobs to be of equal lengths. For the readers convenience, Figure 3 gives a minimal example of a ΨC application source code, and its corresponding timeline.

3.2.2. Micro-Kernel Features & Services

Three system services are provided to Ψ tasks. The first service is *Job Releasing (time-triggered)*, executed on timer interrupts, which are programmed by the micro-kernel to occur at the corresponding release date. The micro-kernel then makes the job ready for execution[3]. The second service is *Job Termination (system call):* it is triggered by a job to signal the end of its processing. The last service is *Deadline Postponement (system call)*, triggered by a job to postpone its current deadline[4]. At last, the micro-kernel is in charge of *monitoring* CPU budget and deadlines of each jobs, by setting appropriate watchdogs.

The scheduling policy basically assigns a priority to each job, which may change with time (*dynamic* scheduling). Once priorities are set, the micro-kernel allocates the available cores to the jobs with the highest priorities. The Ψ programming model makes no assumption on the scheduling policy; however, some policies such as *Earliest Deadline First (EDF)* [21] are preemptive: *the micro-kernel must therefore be able to interrupt and restore a job at any time*. The Ψ micro-kernel usually implements an EDF policy, or its multi-processor variants[5]. For this work, we chose to implement a *Global-EDF* algorithm [22]: on a *n*-core processor, the *n* jobs with the highest priority are executed. See section 4 for more details. Figure 3(b) gives the EDF-scheduling of two tasks on a single core.

4. Overview of the Microkernel

The micro-kernel is designed to take advantage of the target parallelism while preserving determinism, and to oversee task execution in abstraction of their execution models. Therefore in the following we will generically refer to "tasks" (respectively Ψ or Σ if the distinction is necessary), as execution entities manipulated by the micro-kernel.

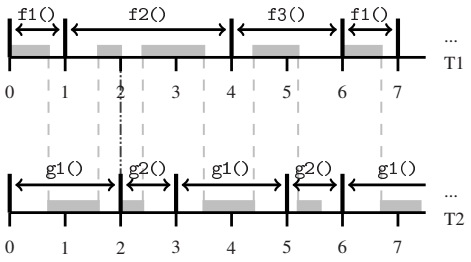
[3]The Ψ programming model is exclusively *time-triggered*, meaning that unlike a CSDF filter, a Ψ job may not blocked waiting for another job to produce an output. A job is eligible for execution iff. its release time has passed and its deadline has not.

[4]Both system calls are not made directly by the ΨC programmer: they are automatically generated by the compiler.

[5]The scheduling policy itself, i.e. the routine that assigns a priority to a job could be implemented by a service external to the micro-kernel: its single function could be to run the jobs with respect to the assigned priorities. In practice, we will see that the scheduling policy is kept within the micro-kernel, mainly for safety and simplicity reasons.

```
1  task T1
2  {
3    f1(); advance(1);
4    f2(); advance(3);
5    f3(); advance(2); /* loop */
6  }
7  task T2
8  {
9    g1(); advance(2);
10   g2(); advance(1); /* loop */
11 }
```

(a) ΨC source code (simplified syntax). The `advance` keyword gives the width of the temporal window.



(b) Corresponding timelines. Double arrows are temporal windows specified by the source code, and grey boxes are actual job execution.

Fig. 3. A simple Ψ application made of 2 tasks (or *agents*), and its execution on a mono-processor with EDF scheduling. Note in (b) how at $t = 2$, the job of `T1` is preempted by the one of `T2`, whose deadline comes earlier.

4.1. Processor Partitioning and Task Mapping

The cores of the target are divided in two categories: 1) Control Core (CC), which is in charge of the main part of task scheduling, and supervises the other cores; 2) Processing Core (PC), which is in charge of user computation, inter-process communication and of a minor part of task scheduling.

There is always one CC, and at least one PC. This repartition takes benefit from the parallelism of target architectures to optimize the scheduling operations, by minimizing the scheduling overhead on the PC (they only perform small operations that do not imply race conditions on the different PCs). Using mostly lock-free and wait-free algorithms [23], the CC can perform the remainder of the scheduling operations while the PC keeps running ready tasks (if any left). The only locked synchronization occurs when tasks are inserted in the list of ready tasks sorted by priority, which could be improved by using a more complex data structure that can be concurrently accessed with lock-free or even wait-free algorithms (*e.g.* as proposed in [24]). The number of PCs must be adequate to avoid making the CC a bottleneck. This is not a problem when partitioning the available cores into logical or physical computation clusters (cf. 1.2).

As in [4], we partition the set of available PCs in two categories: Real-Time (RT) and Non-Real-Time (NRT). The set of RT processors can run either real-time or stream tasks, with an appropriate scheduling policy. The set of NRT processors is dedicated to the stream tasks. Unlike the fore-mentioned approach, we do not bind one single real-time task per RT core (multiple Ψ tasks can run on the same core, they can be preempted, and even freely migrate to other RT-cores), and any stream task can run on any RT-core, as long as no real-time task is waiting for execution; this allows a far more efficient use of all available cores.

4.2. General Execution Model

The execution of both the real-time and stream tasks is supervised by their respective execution model. Our approach in order to have a cohabitation between those tasks is to extract common concepts from both models to make a global and common one; some specific concepts of each model remain, but are considered (and implemented) as refinements of the global model.

We first define a task *instance* as a portion of its execution for which a beginning and an end can be clearly identified. For the stream tasks, as detailed in 3.1, it corresponds to the execution of one of its CSDF functions. For the real-time tasks, as detailed in 3.2, it corresponds to one job. A task instance is always preemptible.

A task's *status* is one of the following and evolves as shown in Figure 4:

1. *ready eligible*: the task is ready in its specialized execution model and waiting for a PC to load it – this state can be an initial state, the only possible transition is when the task is loaded on a PC (to state *running*);
2. *ready not eligible*: the task is still ready in its execution model, but has been preempted, or stopped by itself, and awaits processing by the CC before being eligible again;
3. *running*: the task is loaded on a PC, and executing its current instance – the possible transitions are to *ready not eligible* or *ended instance*;
4. *ended instance*: the task has finished its current instance, it was unloaded from a PC and is waiting for the CC to update its status, possible transitions are to *blocked* or to *ready not eligible*;

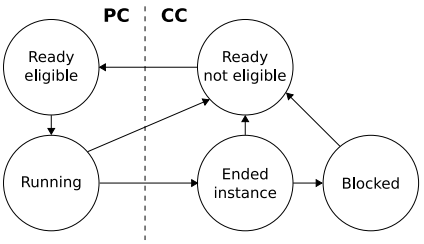


Fig. 4. *The automaton describing a task's cyclic behavior in the system*; the dashed boundary separates the states where either PC or CC is responsible for triggering transitions to another state.

5. *blocked*: the task is not ready in its specialized execution model – this state can be an initial state, the only possible transition is to *ready not eligible*.

A task t which is ready eligible for its current instance has a priority number $p_t \in \mathbb{N}$, with $\forall u \neq t, p_t \neq p_u$ and instance of task t has a higher priority than the instance of task u if and only if $p_t < p_u$. A stream task always has a priority $p_\Sigma = \infty$. The priority of the real-time tasks is computed using an appropriate multi-core scheduling algorithm, like G-EDF [22].

4.3. PC Micro-kernel Routines

There are two micro-kernel routines for the PC, both executed in supervisor mode. They are triggered either by a system call from user mode, or by a preemption interrupt sent from the CC. Note that only RT-PCs may receive a preemption interrupt: preemption may only be necessary to execute Ψ tasks, since Σ tasks have an infinite priority.

Depending on the system call type, the routine puts the current task descriptor either in a "Ended Instance" list or in a "Ready-Not-Eligible" list, both of which are shared with the CC. An event is then sent to the CC to indicate that tasks are awaiting processing; then the PC looks for new available tasks in a "Ψ-ready" list, or if the latter is empty, in a "Σ-ready" list.

In the preemption interrupt subroutine, the PC looks for a task in the "Ψ-ready" list with a higher priority than the instance it currently executes. If it finds a match, the current instance is preempted, put in the "Ready-Not-Eligible" list, and the higher priority task is executed. An event is also sent to the CC to indicate that a task is awaiting processing.

4.4. CC Processing

Because the CC is used exclusively for micro-kernel processing, its execution flow is quite simple: a single routine is repeatedly executed, using synchronization instructions (recall 1.2) to reduce the average load. The main routine basically processes the events triggered by the PCs and by the real-time clock subroutine, and runs the global scheduling in order to ultimately feed the "Ψ-ready" list and the "Σ-ready" list with instances of tasks to execute. Part of the scheduling includes a "core election" procedure, that chooses and notifies the PCs that should run a new task instance; this algorithm is critical to prevent task priority inversions. Note that because the CC is the most obvious potential bottleneck of our system, the main CC routine is optimized to ensure the lowest latency for real-time tasks processing; in particular, the latency induced by the execution of Σ tasks is bounded, and independent from the number of Σ tasks in the system.

The main routine may only be preempted by the Real-Time Clock interrupt, that updates the global current date, used by the Ψ tasks. The RTC interrupt is programmed to trigger periodically: this time period is in practice the smallest time quantum measurable by the micro-kernel. It must therefore be chosen wisely, as a trade-off between the real-time requirements of the application, and the CC overhead induced by the interrupt subroutine. Incrementing and reading of the current time value is made using Lamport's shared clocks algorithm[25], to allow wait-free accesses. Note that the interrupt subroutine runs in constant time.

4.5. On Weak Synchronization between the CC and the PCs

Although the CC is in charge of the global scheduling policy, there are paradoxically very few guarantees on the way cores are mapped to ready tasks. For instance, when the CC sends a `WAKEUP` event to idle PCs because some Σ tasks are standing in the "Σ-ready" list, there is no way to say exactly which core will execute which task; first because there is no guarantee on the order in which the PCs will actually wake up [6], and more importantly because it is possible for another PC, that was not allocated for this task in the first place, to "steal" the task before the other PCs. A PC may therefore receive a `WAKEUP` event "for nothing", i.e. find the "Σ-ready" list empty and fall back asleep. For similar reasons, a RT-PC may also receive a *preemption interrupt* for nothing, because the high-priority task the preemption interrupt was sent for in the first place has already been handled by another RT-PC.

This weakly synchronized architecture ensures that priorities are respected, and that tasks are executed as fast possible, i.e. as soon as an appropriate PC is free – even if the CC is not aware of it yet. Note however that this could be modified to fulfill other requirements, such as binding a set of tasks to a specific core. But we intended to show with this implementation that CPU allocation and task migration could be handled *online*, safely and efficiently, by the system software.

5. Early Benchmarking

We present in this section the first performance evaluations of the Psigma micro-kernel, in order to validate its main architectural choices. Although the code is still weakly optimized, we show that our prototype provides pretty good real-time performances, while running simultaneously a compute-intensive Σ application.

5.1. Platform & Applications Description

Our test platform is a dual processor Intel Xeon 2.53 GHz quad-core, with HyperThreading deactivated[7]: therefore 8 SMP cores are available. It is not exactly an embedded platform to say the least, but it was clearly more comfortable to use a widely spread and tried processor for our first implementations, debugging and benchmarking. Most of our measures were made using two representative multi-task Ψ and Σ applications:

PID Autopilot (Ψ) : 6 communicating real-time tasks, 4 being dedicated to a 5ms-periodic Instrumentation & Control (I&C) loop that runs a UAV autopilot following GPS waypoints. The other 2 tasks are used for I/O (VGA display and manual command input, with respectively 10ms and 50ms period). The autopilot core code is a slightly modified version of an actual scale model UAV.

Edge Detection (Σ) : 16 stream tasks continuously running a standard zero-crossing Laplacian algorithm on an input image, as fast as possible. On an actual embedded system, this application could process a live video feed, e.g. from an embedded infrared camera.

In the following, all the available cores are defined as RT-PCs, meaning they are all authorized to run Ψ or Σ tasks; this allows us to stress the scheduling capabilities of the micro-kernel for mixed-criticality execution[8].

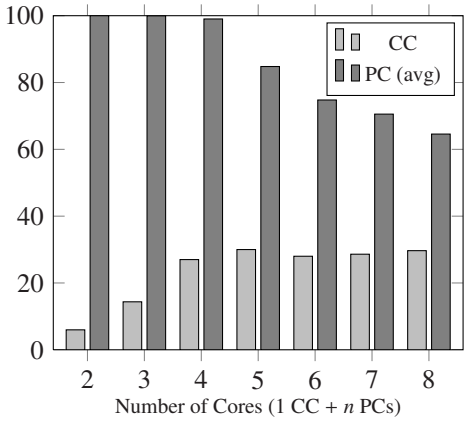
5.2. Cores Load & Real-Time Latency

Figure 5(a) gives the average CPU loads when enabling from 1 to 7 PCs. Beyond 3 PCs (i.e. 4 cores), the system appears to be oversized for the application, as the load of the PCs falls beneath 90%. For comparison, when the real-time application runs alone on 7 PCs (i.e. with Σ tasks disabled) the average PC load is 7.49% while the CC load is barely 0.1%. Besides, when running the Σ application alone (i.e. with Ψ tasks disabled), the average PC load varies from 97.40% with 3 PCs turned on, to 59.78% with 7 PCs turned on. Thus when running both

[6] Unless otherwise provided by the hardware

[7] Hyperthreading (HT – Intel implementation of SMT) has shown undesirable side effects during our tests: in some cases, logical cores may drastically slow each other execution.

[8] When NRT-PCs are available, the core allocation algorithm tends to favor them over RT-PCs to run Σ tasks; this policy minimizes the number of preemptions on RT-PCs.



(a) Average CPU Loads (in %) measured on a 5min period. For each case, the standard deviation of the PCs loads is less than 5%, showing a fair dynamic load-balancing.

Number of cores	2	3	4	5	6	7	8
Min. latency	1.8	1.8	1.4	1.4	1.4	1.5	1.4
Avg. latency	1.8	1.9	2.0	2.3	2.4	2.5	2.6
Max. latency	4.0	4.7	4.9	5.3	5.8	6.0	6.9

(b) Real-Time Latency on the CC: time in μs between the timer interrupt and the release of the corresponding task in the "Ψ-ready" list – 10.000 samples

Inter-Processor Interrupt Between the sending instruction and the first instruction of the Interrupt subroutine executed by the receiver	0.9 μs
Full Context Switch Switch from one task hardware context to another: stack, general purpose registers – no FPU registers.	1.1 μs
Real-Time Latency – Wakeup Between the timer interrupt on the CC and the first user instruction executed on the PC, initially idle.	3.2 μs
Real-Time Latency – Preemption Between the timer interrupt on the CC and the first user instruction executed on the PC, initially running another task.	4.0 μs
Instance Termination Syscall Between the syscall instruction and the end of context unload – includes picking up of the next task to be executed, if any	1.9 μs

(c) Miscellaneous Micro-Kernel Operations Latencies. Each value is an average on 10.000 samples during nominal execution, with 7RT-PCs + 1 CC.

Fig. 5. Performance Evaluation of the Psigma micro-kernel – Running simultaneously the Autopilot & Laplacian Applications

applications, the Σ application clearly takes advantage of the unused CPU time, and the cohabitation overhead is negligible.

Note on figure 5(a) that the load of the CC rises with the number of PCs, but reaches a limit around 30%. This limit depends on the global number of tasks, especially Σ: a high rate of instance release & termination induces a higher load for the CC. Table 5(b) gives the evolution of real-time processing latency on the CC with the number of PCs. This latency increases linearly because of the core allocation algorithm, but remains beneath 2.6 μs on average. Table 5(c) gives some time references for basic micro-kernel operations; note that quite naturally, the worst latency for a real-time instance release corresponds to the case where a PC has to be preempted: 4 μs, including 0.9 μs for the interrupt signal to be issued and received. There is no difference however between preempting a Σ or a Ψ task, thanks to our unified task execution model. At last, basic PC micro-kernel operations such as task termination and context switching remain below 2 μs, which is very encouraging for a first prototype implementation.

6. Acknowledgements

The Psigma kernel was developed during the CHAPI project [26], which is financed by the French cluster "Minalogic". The authors wish to thank all the partners of the project, especially the engineering teams at Kalray working on the embedded many-core chip "MPPA".

7. Conclusion and future works

We proposed a common execution model for the cohabitation of time-triggered real-time tasks and compute intensive tasks, with a micro-kernel implementation for embedded multi-cores. The current implementation runs on an Intel SMP Xeon platform, and on a Kalray MPPA cluster. This prototype evaluation on an Intel platform showed that latencies of less than 4 μs can be achieved for the real-time tasks without a waste of processing power for the other tasks. We intend to make a more complete evaluation on the Kalray MPPA cluster, in order to

benchmark the micro-kernel on an actual embedded platform. Besides, WCET analysis on some critical routines could also be used to infer general latency properties in the micro-kernel.

As short term future work, we plan on implementing automatized memory protection for each task, as it has been done for Ψ tasks in other implementations [27]: it is obviously a key element to ensure safety by isolation, but also determinism on a multi-processor platform. Later on, we will improve our general task model with communication mechanisms that allow the tasks from different specialized models to exchange data. The main challenge here will be to preserve real-time guarantees while providing a tractable programing model. Another improvement will be to actually extend this approach to embedded many-cores, by handling the communication and synchronization between several instances of the micro-kernel on physical or logical clusters of multi-cores.

References

[1] T. Goubier, R. Sirdey, S. Louise, V. David, ΣC: a programming model and language for embedded manycores, in: Proceedings of the 11th international conference on Algorithms and architectures for parallel processing - Volume Part I, ICA3PP'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 385–394.

[2] C. Aussaguès, V. David, A method and a technique to model and ensure timeliness in safety critical real-time systems, in: Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on, IEEE, 1998, pp. 2–12.

[3] S. Brosky, S. Rotolo, Shielded processors: Guaranteeing sub-millisecond response in standard Linux, in: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IPDPS '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 120.1–.

[4] E. Piel, P. Marquet, J. Soula, J.-L. Dekeyser, Asymmetric scheduling and load balancing for real-time on Linux SMP, in: Proceedings of the 6th international conference on Parallel Processing and Applied Mathematics, PPAM'05, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 896–903.

[5] S. Vestal, Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance, in: Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International, IEEE, 2007, pp. 239–243.

[6] S. Banachowski, T. Bisson, S. Brandt, Integrating best-effort scheduling into a real-time system, in: Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International, IEEE, 2004, pp. 139–150.

[7] S. Brandt, S. Banachowski, C. Lin, T. Bisson, Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes, in: Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE, IEEE, 2003, pp. 396–407.

[8] J. Herman, C. Kenna, M. Mollison, J. Anderson, D. Johnson, RTOS support for multicore mixed-criticality systems, in: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th, IEEE, 2012, pp. 197–208.

[9] J. Calandrino, H. Leontyev, A. Block, U. Devi, J. Anderson, Litmusˆ rt: A testbed for empirically comparing real-time multiprocessor schedulers, in: Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International, IEEE, 2006, pp. 111–126.

[10] J. Kiszka, Towards Linux as a real-time hypervisor, in: 11th Real-Time Linux Workshop, Dresden, 2009, pp. 28–30.

[11] J. Zhang, K. Chen, B. Zuo, R. Ma, Y. Dong, H. Guan, Performance analysis towards a kvm-based embedded real-time virtualization architecture, in: Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on, IEEE, 2010, pp. 421–426.

[12] M. Lee, A. Krishnakumar, P. Krishnan, N. Singh, S. Yajnik, Supporting soft real-time tasks in the Xen hypervisor, in: ACM Sigplan Notices, Vol. 45, ACM, 2010, pp. 97–108.

[13] J. Lee, S. Xi, S. Chen, L. Phan, C. Gill, I. Lee, C. Lu, O. Sokolsky, Realizing Compositional Scheduling through Virtualization , in: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th, IEEE, 2012, pp. 13–22.

[14] V. Legout, M. Lemerre, et al., Paravirtualizing linux in a real-time hypervisor, ACM SIGBED Review 9 (2) (2012) 33–37.

[15] G. Kahn, The semantics of a simple language for parallel programming, in: J. L. Rosenfeld (Ed.), Information processing, North Holland, Amsterdam, Stockholm, Sweden, 1974, pp. 471–475.

[16] E. A. Lee, T. Parks, Dataflow process networks, in: Proceedings of the IEEE, 1995, pp. 773–799.

[17] R. L. G. Bilsen, M. Engels, J. A. Peperstraete, Cyclo-static data flow, IEEE Transactions on Signal Processing 44 (2) (1996) 397–408.

[18] J. T. Buck, E. A. Lee, Scheduling dynamic dataflow graphs with bounded memory using the token flow model, Tech. rep. (1993).

[19] K. Denolf, M. Bekooij, J. Cockx, D. Verkest, H. Corporaal, Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations, EURASIP Journal on Advances in Signal Processing 2007 (1) (2007) 084078.

[20] P. Dubrulle, S. Louise, R. Sirdey, V. David, A low-overhead dedicated execution support for stream applications on shared-memory CMP, in: Proceedings of the tenth ACM international conference on Embedded software, EMSOFT '12, ACM, New York, NY, USA, 2012, pp. 143–152.

[21] C. Liu, J. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Journal of the ACM (JACM) 20 (1) (1973) 46–61.

[22] U. Devi, J. Anderson, Tardiness bounds under global EDF scheduling on a multiprocessor, Real-Time Systems 38 (2) (2008) 133–189.

[23] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, Burlington, MA, USA, 2008.

[24] A. Natarajan, L. Savoie, N. Mittal, Brief announcement: Concurrent wait-free red-black trees, in: M. Aguilera (Ed.), Distributed Computing, Vol. 7611 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 421–422.

[25] L. Lamport, Concurrent reading and writing of clocks, ACM Transactions on Computer Systems (TOCS) 8 (4) (1990) 305–310.

[26] CHAPI Project - Embedded High Performance Computing for Industrial Applications, http://www.minalogic.com/.

[27] M. Lemerre, E. Ohayon, D. Chabrol, M. Jan, M. Jacques, Method and tools for mixed-criticality real-time applications within pharos, in: Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on, IEEE, 2011, pp. 41–48.