Predictable Flight Management System Implementation on a Multicore Processor

Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, W. Puffitsch

Predictable Flight Management System Implementation on a Multicore Processor

Guy Durrieu* Madeleine Faugère† Sylvain Girbal† Daniel Gracia Pérez† Claire Pagetti* Wolfgang Puffitsch‡
*ONERA - Toulouse, France, †Thales TRT - Palaiseau, France ‡DTU Copenhagen, Denmark

***Abstract*—This paper presents an approach for hosting a representative avionic function on a distributed-memory multicore COTS architecture. This approach was developed in collaboration by Thales and ONERA, in order to improve the performance of the function while enforcing its predictability. Once the target avionic function and the multicore architecture have been introduced, the execution model and the needed basic services are described and evaluated.**

I. Introduction

Multicore processors are becoming the only solutions available for the development of embedded safety critical applications with increasing performance requirements. These architectures are challenging for safety critical applications because they are in general not predictable, in the sense that (1) evaluating the worst case execution time (WCET) of an application is almost impossible [22] and (2) synchronizing the different cores precisely is sometimes hardly achievable [11]. The difficulties increase when the multicore hosts several applications and in particular mixed critical applications [6]. A solution to enforce predictability relies on the use of an appropriate *execution model* [2], [4]. Such a model is a set of rules to be followed by the designer in order to avoid, or at least reduce, unpredictable behaviors.

The purpose of the paper is twofold: (1) to define an improved execution model that fits the requirements of complex safety critical applications, and (2) to describe the executive services implementing the constrained behaviors imposed by the execution model on a real target. The execution model and its implementation will be evaluated using a *Flight Management System* as a representative complex safety critical application from the avionic domain.

A. Input of the project

The application studied in this work is a simplified but representative version of a *Flight Management System (FMS)* developed at Thales. In modern avionics, the FMS provides the crew with centralized control for the aircraft navigation sensors, computer based flight planning, fuel management, radio navigation management, and geographical situation information. Pilot and co-pilot have a strong control on the application and therefore the implementation comprises both periodic and aperiodic tasks.

The target COTS architecture is a Texas Instruments TMS320C6678 (TMS in short) high-throughput multicore processor [7], comprising 8 DSP cores. This architecture offers several advantages with regard to real-time: it is possible to configure the local caches as local SRAM, making the platform a distributed memory architecture with explicit accesses to the shared bus and memories. However, this target suffers from a lack of synchronization: the cores are started independently and the hardware local clocks, even though they do not drift, have unpredictable offsets.

B. Context and related work

Executing the FMS safely on several cores of the TMS requires a robust real-time implementation ensuring predictability regardless of executing conditions (e.g. starting order of the cores, variation in the execution times). While there is active research on time-predictable computer architectures [21], [16] these architectures cannot (yet) compete with COTS processors in terms of availability or cost. Therefore, the work presented in this paper targets a hardware platform that was not designed with predictability in mind. The sources of unpredictability on COTS multicore processors are numerous (e.g. concurrent accesses to the shared resources), and are identified nowadays. Several approaches have highlighted different ways to reduce the unpredictability by acting on:

- The application design. For instance, program development can follow the MISRA coding rules [5].
- The access to the shared resources. The designer must rely as much as possible on predictable mechanisms offered by the hardware, such as TDMA arbitration [17]. However such mechanisms are rare, thus constraining the concurrent accesses is mainly reached by forcing some determinism. For instance, preventing a core from accessing the RAM can be realized by coding tasks that are small enough to be fully stored in the local caches [9], or limiting the number of conflicts can be obtained by restraining the number of accesses during an interval [23].
- The scheduling of tasks. Partitioned non-preemptive offline schedules are much more predictable than global preemptive schedules.
- The hardware mechanisms. For instance, deactivating the cache coherence [3] increases the predictability at the cost of managing the coherent vision by software (cost of CPU time and additional code).

C. Contribution

As pointed out in the previous section, reducing the unpredictability with an execution model requires to take into

account both the application's specifics and the target capabilities. Existing work in the literature cannot handle a safe execution of the FMS on the TMS, because these approaches are often partial (e.g. no management of aperiodic tasks) and/or too restrictive (e.g. no solution to handle large databases, no approach to minimize the performance degradation when several cores run in parallel). The FMS is full of irregular real-time behaviors and of complex communication patterns between the tasks, requiring for instance to manage validity of data. We therefore promote an execution model which combines and extends existing results. It is not a new model but rather an improvement and adaptation of existing approaches in order to implement a representative application on a multicore COTS architecture.

- Periodic tasks execute on dedicated cores. Scheduling is partitioned non-preemptive off-line. All instructions and data are stored in the distributed local SRAM memories. This is a direct adaptation of [11] and [4]. A bare-metal executive layer has been developed specifically for the TMS. A bare-metal approach is a low-level programming method that allows to bypass the BIOS or operating system interface to achieve high-performance and real-time computing with minimal footprint.
- Aperiodic tasks execute on dedicated cores provided with the Texas Instruments SYS/BIOS operating system. Scheduling is global deadline monotonic (DM). This part is an on-going work.
- Communication between the cores is done by directly writing to the distributed on-chip SRAM. A memory area, named MPB (Message Passing Buffer), is reserved for that purpose in this SRAM on all cores. The notion of MPBs is inspired by the architectural design of the Intel Single-chip Cloud Computer (SCC) [8].

The paper is organized as follows: Section II describes the real-time design of the FMS application; Section III depicts the features of the TMS target that can be used to enforce predictability; Section IV details the execution model, the current implementation and the next steps to be realized by the end of the year.

This study was realized using the TMS320C678 as the hardware target. The solutions presented in this paper however are not restricted to this particular architecture, can could easily be applied to any distributed-memory multicore.

II. Description of the FMS

The Flight Management System aims at performing in-flight guidance of aircrafts. Such guidance is based on the use of *flight plans* selected before departure, either by the pilot or a dispatcher for airliners. A flight plan includes basic information such as departure and arrival points, in-flight waypoints, estimated time en route, alternate landing airports, expected weather conditions, and so on. During the flight, the FMS is then in charge of (1) determining the plane localization and (2) computing the trajectory in order to follow the flight plan and the pilot directives.

A. General overview

The Thales FMS220 [19] is used both on regional airliners (ATR-72, ATR-42) and on some helicopters (Sikorsky S-76D). The original version was sequential and the code has been rewritten in a multi-threaded way.

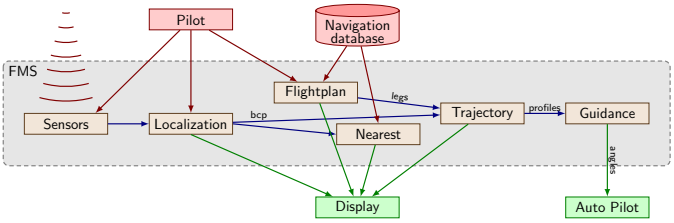

Fig. 1. Functional overview of the FMS

Figure 1 gives a functional overview of the Flight Management System. For the sake of simplicity, the system is described as a set of functional groups, which do not necessarily exist at implementation level.

- The *sensor group* collects information from various sensors such as the Anemo-barometric sensor, the Pure Inertia Reference System (IRS) sensor, the Global Positioning System (GPS) or the Doppler sensor.
- The *localization group* performs some sensor merging on the localization and speed information provided by the sensor group to compute a *Best Computed Position (BCP)* corresponding to the most probable position of the plane.
- The *flight plan group* deals with in-flight flight plan modifications, extracting new flight routes from the navigation database.
- The *trajectory group* uses the BCP position to compute some physically possible trajectories (profiles) to follow the selected flight plans. This computation involves very complex physical models and constraints, such as respecting arrival time windows and optimizing the fuel consumption.
- The *guidance group* translates the profiles computed by the trajectory group to angle corrections values that are transmitted to the autopilot.
- The *nearest group* is regularly computing the list of the nearest airports. This information is not used to guide the plane, but will help the pilot in case of an emergency landing.
- The *display* is not directly part of the FMS but every transmitted data could be sent to the display such as the BCP, the selected flight plan, or the list of nearest airports.

Figure 2 zooms on the content of localization group. LOC_{C1}, LOC_{C2}, LOC_{C3}, and LOC_{C4} are periodic tasks respectively in charge of sensor merging, flight phase computation, magnitude variation correction and navigation performance computation. LOC_{A1}, LOC_{A2}, and LOC_{A3} are aperiodic tasks corresponding to manual selection of active sensors, manual reinitialization of the magnetic variation, and manual configuration of the required navigation performance.

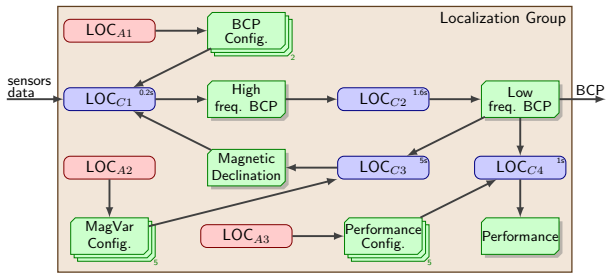

Fig. 2. Detailed view of the Localization group

The communication of this data between tasks is done via dedicated *blackboard* or *mailbox* structures appearing as shadowed box or stacks in the figure. For instance, the variable *High freq BCP* is stored in a *blackboard*, produced by LOC$_{C1}$ and consumed by LOC$_{C2}$. A *blackboard* is a single-writer / multiple-reader double-buffer allowing concurrent reads and write. A *mailbox* is a single-writer / single-reader FIFO usually used to enqueue pilot and co-pilot requests in case of manual intervention (after an aperiodic task). These requests are then taken into account on the next activation of the consumer task.

B. Real-time constraints

The scheduling of the system is done independently of the communication between tasks. In particular, when a manual configuration is done by the pilot, the associated aperiodic task does not trigger the periodic task that consumes the *mailbox*. As a consequence, tasks in the system may work with value that does not correspond to the very latest input value. Figure 3 shows the asynchronous communication between aperiodic and periodic tasks.

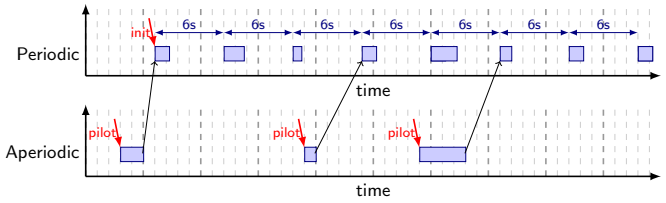

Fig. 3. Consumption of pilot requests

The Flight Management System contains a specific type of tasks named *restartable tasks*. Such a long-running task is managed by another task that can either suspend or restart it (see Figure 4). This is for instance the case of trajectory computation tasks, that could be suspended either on flight plan change or when the BCP value is considered too old.

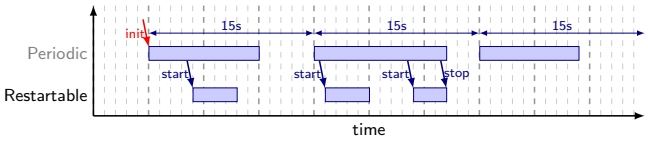

Fig. 4. Restartable task activation patterns

Globally, the FMS is composed of 10 *periodic* tasks, 15 *aperiodic* tasks and 3 *restartable* tasks. All the above-mentioned tasks have hard real-time constraints. The Table I gives the real-time constraints associated to the tasks of the localization group. Periods are varying from 200ms to 5s, while the number of asynchronous task activations is bounded per period, due to the usage of an HMI.

task	period
LOC$_{C1}$	200ms
LOC$_{C2}$	1.6s
LOC$_{C3}$	5s
LOC$_{C4}$	1s

task	max activation
LOC$_{A1}$	2 every 200ms
LOC$_{A2}$	5 every 5s
LOC$_{A3}$	5 every 1s

TABLE I
REAL-TIME CONSTRAINTS OF THE LOCALIZATION GROUP

C. Memory footprint

For distributed memory systems, it is critical to describe the memory footprint of the application, as its deployment is constrained by the distributed memory sizes. The overall memory footprint of each task can be decomposed into: (1) the *code size* that corresponds to the part of the *text segment* from the binary that is related to the task; (2) the *instance size* that corresponds to the task object including all the task interval variables; (3) and finally the *output data size* that corresponds to the size of the target output blackboard of mailbox buffers.

task	code size	instance size	output size
LOC$_{C1}$	6236	2744	1232
LOC$_{C2}$	3072	1360	1232
LOC$_{C3}$	3892	1104	272
LOC$_{C4}$	2528	1104	208

task task	code size	instance size	output size
LOC$_{A1}$	9012	200	136
LOC$_{A2}$	9216	224	328
LOC$_{A3}$	9524	224	328

TABLE II
MEMORY CONSTRAINTS (IN BYTES) OF THE LOCALIZATION GROUP

Table II shows the memory footprint of the Localization group. In addition to the code size of the tasks to be deployed on each core, some additional code is added at compile time: 60KB to deal with the Flight Management data types, and 105KB for the platform related level libraries.

The complete memory footprint of the FMS application consists of 295KB of code and 25KB of data, to be completed with a few megabytes for the navigation database stored in the DDR memory.

D. Communication requirements

The FMS application relies on the use of *semaphores* or *mutexes* to protect communication structures allowing concurrent accesses to the mailboxes and the blackboards. Additionally, some specific data structures, such as the flight plan, also involve some mutual exclusion sections to avoid concurrent modifications by the pilot and the copilot.

Moreover, the FMS application requires some *output communication channels* to send the output data to the autopilot and the display. Such a traffic is characterized by a very low throughput.

III. Description of the TMS

To implement any real-time application on the TMS, we need to identify the available hardware features to enforce predictable behavior.

A. General overview

The Texas Instruments TMS320C6678 (see Figure 5) is a multicore platform, comprising 8 TMS320C66x DSP processors that are clocked at 1 GHz. The TMS320C66x cores implement a VLIW instruction set architecture and can issue up to 8 instructions in the same clock cycle. The cores are connected via the TeraNet on-chip network, which also provides access to several auxiliary hardware modules (e.g., I/O interfaces and DMA engines). Each core contains 32 KB level 1 program memory (L1P), 32 KB data memory (L1D), and 512 KB level 2 memory (L2) that may contain instructions and data. The level 3 memory (MSMC) provides 4 MB of on-chip SRAM that is shared between cores. The external DDR3 memory is shared between cores as well. While this memory has a long latency compared to the on-chip memories, it is also considerably larger (512 MB on our evaluation board).

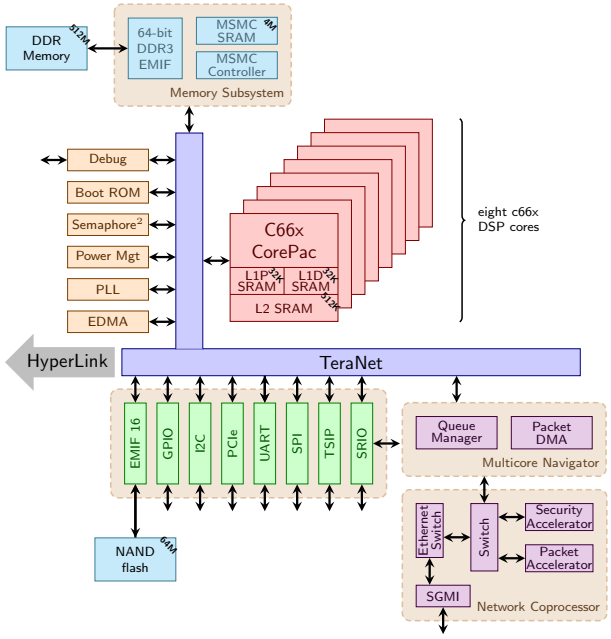

Fig. 5. TMS320C6678 architecture

B. Direct Communication through On-Chip Memories

The L1P, L1D, and L2 memories can be configured as cache, SRAM, or a mix of both. When part of a local cache is configured as SRAM, all other cores can have direct distant access to that memory area thanks to global memory addressing. The access to the local memories can be used to implement direct communication between cores without crossing shared L3 or DDR3 memory. A core can access its local memories through two different addresses: *global address* (accessible by all cores) or *local address* (only accessible by the associated processor through aliased address). For example, core 3 can access its L2 SRAM memory through local address `0x00800000` or through global address `0x13800000`. The local addresses are exactly the same for all cores and should thus be used for common code to be run unmodified on multiple cores.

C. Timing management

On each core there is a local time-stamp counter that runs at the core's frequency of 1 GHz. This local clock is obtained by reading two registers: *TCSL* and *TCSH*. We have to be careful with those registers for two reasons:

1) The cores start independently and the counters on each core are launched at different moments. This entails that even though they do not drift, they have unpredictable offsets.
2) When executing on the evaluation board, the debugger may interrupt the cores and therefore stall those registers. In particular, the use of *printf* causes an interaction between the core and the debugger that can stall the processor.

The hardware of the debugging board provides also a global time-stamp counter that runs at 250 MHz. This clock cannot be used for implementing the execution model but can be used during the debugging phase for verification.

D. Hardware support for communication

The TMS platform provides full support for various means of communication, including Ethernet, serial RAPID-IO, serial UART, GPIO and system calls translated to the local host through the hardware debug probe.

Some of these features however cannot be easily used in a safety critical context. Indeed, communication through the hardware debug probe involves inserting software breakpoints during the execution of distant I/O accesses breaking the hard real-time concept. Relying on the Ethernet controller for communication on the other side would require to either develop or reuse an heavyweight full IP stack, that will both result in a high resource usage and a lack of predictability. Finally some interfaces such as UART could be very lightweight, but only offer small data bandwidth.

The TMS platform also provides some hardware semaphore support thanks to the *semaphore*^{2} hardware module, that provides up to 64 independent semaphores accessible across all the cores to implement shared-resource protection. Note also that the TMS hardware does not provide any hardware features to directly implement barriers.

IV. Porting the FMS to the TMS

The certification of safety critical systems relies on the compliance with industry-level certification standards. Avionic application in particular must conform to the DO-178C [14] (for software development), DO-254 [12] (for hardware design) and DO-297 [13] (for mixed-critical system) standards.

The implementation of the Flight Management System must fulfill the *timing determinism* requirement from those standards. As stated in the introduction, multi-core platforms

introduce multi-clock domains and rely on shared hardware resources which access is handled by dedicated non predictable arbitration mechanisms.

In this section, we explain the execution model that permits us to implement a predictable FMS on the target. We then detail the executive layer specifically developed to execute tasks according to the model's rules and restrictions.

A. Execution model

The execution model provides a set of rules that will enforce predictable behaviors. A first straightforward execution model to ensure predictability is to avoid any interference. Strictly applying time segregation principles as proposed in ARINC 653 on a multi-core system implies to define system-wide time slots where only one core can execute and use the shared system resources, as depicted in Figure 6.

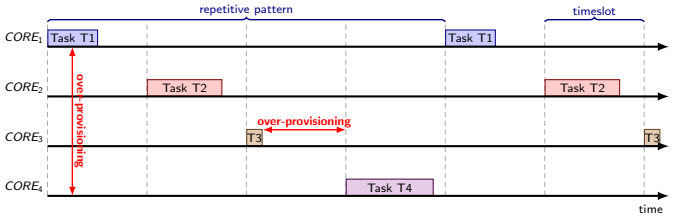

Fig. 6. Scheduling with system-wide time slots

In such a scheme, each task that executes during its dedicated time slot is guaranteed to have full access on all the shared resources (interconnect, devices, memories, ...), eliminating all unpredictable competition on shared hardware resources. The main drawback is that the multi-core platform is underused, and therefore there will be no performance benefits over a single-core. Both sources of resource over-provisioning responsible for the lack of performance are depicted in Figure 6: over-margin of shortest tasks, and unused cores during a time slot.

This is the reason why we promote the execution model illustrated Figure 7.

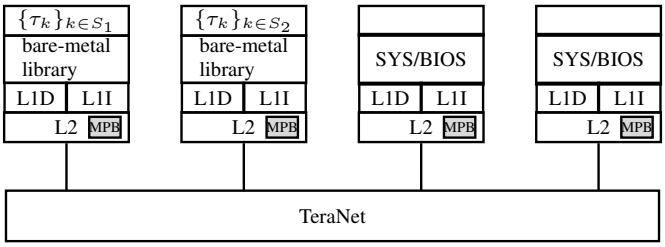

Fig. 7. Execution model at a glance

Periodic tasks execute on dedicated cores. These tasks are the most constrained ones in terms of deadlines. Therefore, we choose to optimize their real-time behavior as much as possible and reuse the most predictable execution model ideas. The following situations should be avoided:

1) interruption by another task due to preemption,
2) stalling by implicit hardware mechanisms such as cache coherence mechanisms,
3) delays during accesses to the TeraNet due to unknown concurrent accesses.

To avoid situation 1), we impose partitioned non-preemptive off-line scheduling. Since there are no precedence constraints among the tasks, standard techniques apply. To compute such a schedule, we have to evaluate the WCET on the TMS of each task.

To avoid situations 2) and 3), we reuse ideas from [9] and [4]: all executions are performed without any shared resource access. This means that all the instructions and data are locally stored in the local memories. Since the size of the periodic tasks is small enough to fit in the local memories, the L2 caches are configured as SRAM. This configuration prevents the cores from accessing implicitly the DDR, L3 memory and TeraNet. Evaluating the WCET without any non-predictable behavior (thanks to the locality of the data and code within the caches) on a uniprocessor, is a well-known problem.

The counterpart is to manage the communication with tasks on distant cores explicitly. To avoid congestion on the DDR and to be more efficient, we rely on the direct communication provided by the TMS. A shared portion of L2 SRAM is reserved for passing messages between cores. To ensure the predictability during the access to the mailboxes and blackboards, we use an *AER task model*. The idea is the following: the remaining interferences only occur during the communication phases between cores, devices and memories. By decoupling execution and communication phases as proposed in [18], [10], [9], [2], it allows us to safely exploit parallelism between execution phases, while still guaranteeing an interference-free system by forcing the communication phases to be sequential.

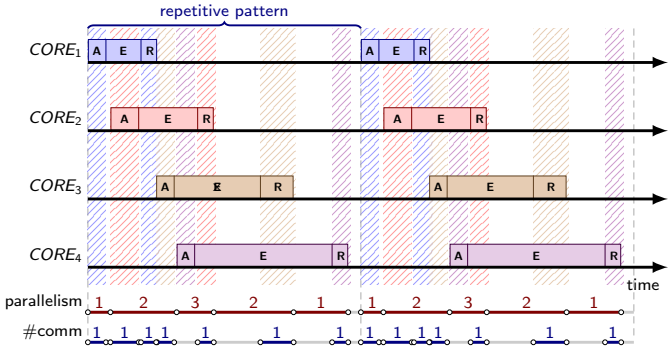

Fig. 8. Scheduling with strict AER task model

Decoupling execution from communication has to be performed at task-level within the application, by decomposing each task into three successive phases: acquisition, execution, and restitution (AER task model). Communication between the distributed memories and devices are only allowed during acquisition and restitution phases, while the execution phase is restricted to only use the local private memory. Such a model allows to run execution phases in parallel ensuring that only

one acquisition or restitution phase runs at a given time as depicted by Figure 8.

The extra design cost of splitting each task into separate acquisition, execution and restitution phases was factored in with the design cost of the spatial partitioning that already forces us to rely on local copies at task level.

MPB. Inter-core communication is done using direct accesses to the on-chip SRAM. To avoid overwriting other data, we reserve some space as *message passing buffer* (MPB). Stack, code and data sections (`.stack`, `.data`, `.text`,...) are allocated in the remaining L2 SRAM. Furthermore, DDR3 memory is partitioned such that each core has a private area. The L3 SRAM remains shared, and can be used to pass data that does not fit into the MPBs. Note that explicit message passing via the MBP has a cost in terms of WCET since a core is blocked as long as the writing to a distant MPB is not finished. We have made several benchmarks on the TMS to evaluate the worst-case latencies when there are concurrent accesses. No unbounded behavior in these latencies has been detected and a write takes at most a bounded number of cycles.

Aperiodic and restartable tasks use large amounts of data (in particular those retrieved from database). Therefore, the approach followed for periodic tasks which consists in storing everything in the L2 SRAM is not applicable. Similarly, static off-line scheduling does not apply either. Fortunately, these tasks are less constrained (large deadlines) and are more tolerant or robust with regard to unpredictability. We can therefore relax the execution model and we choose to execute them on separate cores with the SYS/BIOS operating system. Scheduling is global deadline monotonic (DM). We did not yet start this part, but we will probably rely on non-preemptive scheduling. The configuration of L2 memory has to be discussed because the smallest SRAM portion is 50% of the L2 size. Communication with periodic tasks will be done through the MBPs (in case of configuration 50%) or through L3 memory otherwise. Communication with other aperiodic/restartable tasks will go through the shared DDR memory.

B. Bare-metal library for periodic tasks

We have developed a run-time environment on the TMS based on a bare-metal library. This library is an adaptation of bare-metal libraries[1] developed at ONERA for the Intel SCC [15], [11].

Synchronization of the local time-stamps. As explained section III, the local clocks of the TMS320C6678 are synchronous (i.e. no clock drift between the local clocks) but they are not perfectly synchronized because they do not boot at the same time. The offsets between the cores are not handled by the hardware and it is up to the user to manage a synchronization if needed. We have encountered the same problem on the Intel Single-chip Cloud Computer (SCC) and the TILERA TILEMPOWERGX-36 [20].

[1]http://sites.onera.fr/schedmcore/

The algorithm, specifically developed for the TMS320C6678, is based on the writing of flags in the MPB. The synchronization algorithm works as follows:

- N Boolean variables are stored on the MPB of core 0;
- 1 Boolean variable is stored on each core's MPB;
- when core i starts, it sets the N variables to *false*. Then, the core makes an active wait: as long as it did not receive any value on its own variable, it continuously sets to *true* the i-th variable of core 0;
- core 0 works differently. It sets once all its MPB Boolean variables to *false* and waits actively until all cores are awaken. When this occurs, it sets the variables hosted by the other cores to *true*;
- when a core detects that its local variable is set to *true*, it starts a waiting of 1s using its local clock. After this second, it reads the current time. This value becomes its local offset;
- then the shared global time = local time - local offset.

The principles are drawn in Figure 9.

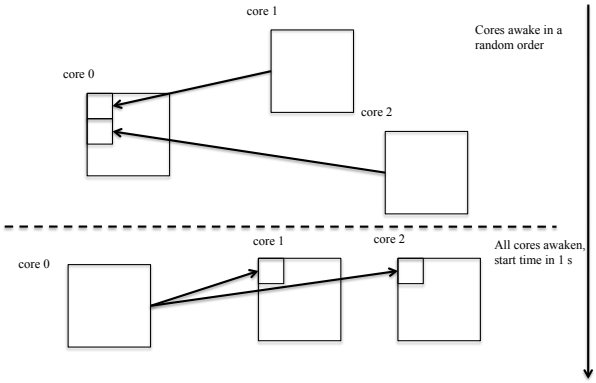

Fig. 9. Synchronization principles

This synchronization algorithm leads to a precision of 40 cycles, that is 40 ns. This is much better from what we obtained on the SCC (4 μs) and the TILERA (0.5 μs in the worst case, 50ns in most of the cases).

Message passing and dispatcher. All the mailboxes and blackboards are hard-coded as illustrated below:

```
Sensor_data = (blackboard *)(MPB(0)+64);
```

The local scheduling consists first in mapping a set of periodic tasks on each core. In order to respect the execution model, the size of the tasks must fit in the local SRAM. At compilation and execution times, if the sizes of the stack and the heap exceed those imposed in the configuration file, an error is raised. A hard coded dispatcher is then defined. For instance, for geometric tasks,

```
for(i=0;i<ITER;i++) {
t1 += PERIOD;
task1(); .... ;taskn();
     // sequence of task with smallest period
if(i%r1==0){
```

```
  task1_r1(); .... ;taskp1_r1();
     // sequence of task with
     // period = smallest period / r1
}

if(i%r2==0){
   task1_r2(); .... ;taskp2_r2();
     // sequence of task with
     //period = smallest period / r2
}
....

active_wait_until(t1);
}
```

C. Application Deployment

We have ported the FMS on the TMS320C6678 according to the execution model described previously. To start with, the integrator must first assess the WCET of each task. No static WCET analysis tool, such as ABSINT [22] or OTAWA [1], is available for the TMS320C6678 platform. Therefore, we use a measure-based approach, which is not safe in general but we can hardly do better at this stage. To measure the execution times, we run the task in isolation and use the local clock of each core that counts the number of cycles. From the experimental observation, there is no variation in the execution times which comfort us on the fact that our implementation is predictable. The values for the tasks of the localization group are given in Table III.

TABLE III
WCET

Task	WCET	Task	WCET
LOC_{C1}	7 ms	LOC_{C2}	6 ms
LOC_{C3}	5 ms	LOC_{C4}	5 ms

The second step consists in choosing a mapping and a scheduling of the tasks. As stated in the introduction, partitioned non-preemptive off-line scheduling best fits the predictability requirements. Partitioning also permits to uses a MIMD ("multiple instruction, multiple data") approach where the created binaries are specific to particular cores.

We tried several mappings for the periodic tasks. We observed no variation on the execution times of the tasks and a perfect reproducibility of the functional execution.

V. Conclusion and next steps

We have defined an execution model to implement safely the Flight Management System on the Texas Instruments TMS320C6678 multicore. At this stage, we are still on the validation and verification phases for the periodic tasks but the first results are promising. The development is not complete since the case of aperiodic and restartable tasks has not been treated.

An other on-going work, besides completing the FMS porting, consists in developing a designer kit. Indeed, for the first experiments, everything is hard coded which lacks of genericity. This kit will automatically generate (1) the source code for a given mapping of tasks to cores and communication buffers to MPBs, and (2) an off-line dispatcher.

Future work will prepare for a wider integration with other applications sharing the same target. The implementation of a mixed critical system must fulfill two main requirements from the avionic standards:

1) *spatial partitioning* to ensure that no running task can corrupt the memory space of another task,
2) *time partitioning* to ensure that no task can delay any other tasks.

Spatial partitioning can be obtained by extended uni-processor techniques while time partitioning remains a complex issue. The conflicts resolution introduces delays that break the time partitioning principle. The purpose of the execution model is therefore to enforce time partitioning. The MULCORS [6] project has given some rules and concepts for implementing mixed-critical systems on multi-core platforms but no solution currently exists.

Acknowledgments

We would like to acknowledge the the European Union Seventh Framework Programme for its partial funding through the Certainty project, grant No. 288175.

References

[1] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *8th IFIP WG 10.2 International Workshop Software Technologies for Embedded and Ubiquitous Systems (SEUS 2010)*, pages 35–46, 2010.

[2] E. Betti, S. Bak, R. Pellizzoni, M. Caccamo, and L. Sha. Real-time i/o management system with cots peripherals. *IEEE Trans. Computers*, 62(1):45–58, 2013.

[3] J. Bin, A. Grasset, D. Gracia Pérez, S. Girbal, P. Bonnot, and A. Merigot. Controlling execution time variability using cots in for safety critical systems. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES'12)*, Fiuggi, Italy, 2012.

[4] F. Boniol, H. Cassé, E. Noulard, and C. Pagetti. Deterministic execution model on cots hardware. In *25th International Conference Architecture of Computing Systems (ARCS'12)*, volume 7179 of *Lecture Notes in Computer Science*, pages 98–110. Springer, 2012.

[5] T. M. Consortium. MISRA-C:2004: Guidelines for the use of the c language in critical systems. Technical Report ISBN 0 9524156 2 3, MISRA, 2008, Edition 2.

[6] M. Gatti. Development and certification of avionics platforms on multi-core processors. In *Tutorial 4 - Mixed-Criticality Systems: Design and Certification Challenges, Embedded Systems Week*, Montreal, Canada, 2013.

[7] T. Instruments. TMS320c6678 Multicore fixed and floating-point digital signal processor. Technical Report SPRS691D, Texas Instruments Incorporated, 2013.

[8] Intel Labs. SCC external architecture specification (EAS). Technical report, Intel Corporation, May 2010.

[9] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011)*, 2011.

[10] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Design, Automation and Test in Europe (DATE 2010)*, pages 741–746, 2010.

[11] W. Puffitsch, E. Noulard, and C. Pagetti. Mapping a multi-rate synchronous language to a many-core processor. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*, Philadelphia, Pennsylvania, USA, 2013.

[12] Radio Technical Commission for Aeronautics (RTCA) and EURopean Organisation for Civil Aviation Equipment (EUROCAE). DO-254: Design assurance guidance for airborne electronic hardware.

[13] Radio Technical Commission for Aeronautics (RTCA) and EURopean Organisation for Civil Aviation Equipment (EUROCAE). DO-297: Software, electronic, integrated modular avionics (ima) development guidance and certification considerations.

[14] Radio Technical Commission for Aeronautics (RTCA) and EURopean Organisation for Civil Aviation Equipment (EUROCAE). DO-178B: Software considerations in airborne systems and equipment certification, 1992.

[15] J. Scheller. Real-time operating systems for many-core platforms. Master's thesis, ISAE/ONERA, Toulouse, France, 2012.

[16] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[17] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*, Stockholm, Sweden, 2010.

[18] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 332–337, 2010.

[19] THALES. FMS 220 software requirement specification (SRS). Technical report, THALES, Nov 2010.

[20] Tilera Corporation. Tile processor architecture - Overview for the TILEPro Series. Technical Report UG120, 2013.

[21] T. Ungerer, F. J. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, J. Wolf, H. Cassé, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.

[23] H. Yun, Y. Gang, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*, Philadelphia, Pennsylvania, USA, 2013.