Deterministic scheduling reconciles cache with preemption for WCET estimation

M. Destelle[1], J.-L. Dufour[1]

1: Sagem, Avionics Division - Avenue du gros chêne, 95610 Eragny, France

Abstract: Inertial Reference Systems are highly critical in the avionics context. As a result, their software components are submitted to a rigorous demonstration of the Worst-Case Execution Time (“WCET”), together with a demonstration of “robust” partitioning. With a processor using cache memory, if task scheduling is preemptive, these demonstrations become a real challenge. The difficulty comes in adequately bounding the cache refill cost due to context switches. This paper presents the solution we have adopted on our new generation of Inertial Reference Systems: it consists in a particular scheduling policy, which allows in turn a particular cache management policy. This cache policy is a key point for performance, because modern processors rely radically on cache to achieve the promised MIPS: the result is – to our knowledge – an unrivalled use of cache on a DAL A system. Lastly, the new scheduling policy puts the final touch to our lock-free inter-task communication protocol.

Keywords: preemptive scheduling, WCET, cache memory, robust partitioning, lock-free communication.

1. Introduction

Inertial Reference Systems are highly critical avionics systems, because – among other things – they deliver pitch and roll angles directly involved in the stability of the aircraft. When the DO-178B/ED-12B recommendation is applicable, the majority of the software is “level A” : this implies a rigorous demonstration of the impossibility of a CPU-time overload. Moreover, “robust” partitioning may be requested between pure inertial computations, barometric computations and hybrid inertial/GPS computations: this implies that “*A software partition should be allowed to consume shared processor resources only during its period of execution*” ([DO-248B/ED-94B] §4.14.5), and of course CPU time can be considered as one of the shared processor resources. In the context of a real project certification, we decided to not open Pandora’s box and to strictly follow this recommendation, but in the conclusion we will take the liberty to technically challenge it.

To deal with these issues, we use traditionally on our Inertial Navigation Systems a very simple multi-tasking architecture: a purely periodic preemptive scheduling (no aperiodic task: asynchronous events are polled, with hardware time-tagging of events when needed), with a fixed-priority rate-monotonic policy (no priority inversion) and harmonic periods (see the annex). With this architecture, the CPU-time overload issue breaks down immediately into task level issues: each task must have a WCET less than its allocated “deadline” (defined here as a limit duration, in contrast to a limit date). To enforce these deadlines, at each task launch the scheduler arms a watchdog with the corresponding duration, and if the watchdog goes off, the faulty task is no more scheduled.

On the simple processors we used until now (pipelined, but without cache), context switches had mainly a deterministic cost (the saving and restoration of the registers), because the disruptions of the pipeline had a “random” but negligible cost. So, on a given application with an average of let’s say 3 task switches every ms, with 30µs per switch, the CPU-load of the scheduler was valued at 3*30/1000 = 9%, and the application had to fulfill:

$$\sum_i D_i/T_i < 91\% \quad [1]$$

(D_i is the deadline and T_i the period of the task number i)
and of course $WCET_i < D_i$ with moreover a security margin.
To summarize, the cost of preemptions was not taken into account in the application tasks, but via an extra “scheduler task”.

On our new generation, this architecture has been confirmed, but a “new mature” processor has been introduced: a PowerPC 8270 (Freescale), containing a 500MHz “G2” core, fed by a 100MHz external memory. This frequency gap is not (on average) a problem, because the G2 core contains two 16Kb caches: one for instructions, and one for datas. They allow (with the help of the pipeline) one instruction fetch and data access per cycle, to be compared to the 15 cycles (30ns) needed by an instruction or data external read: the global performance ratio can reach 30. The next section details the subtle problems the cache can create when it is associated with a preemptive scheduler.

2. The subtle interaction cache / preemption

The impact of caches on performance is well known [Agarwal89][Mogul91], but not so well dealt with by tools [Wilhelm07]:

- For a single non-interrupted task, the average execution time can be significantly shorter (see the former ratio), but the variation can be significantly larger (the data cache can even increase the WCET; for the instruction cache, this is reserved to pathological programs [Sebek02]). This variation is inherent in the memory accesses: a cache hit takes one cycle, a cache miss takes 60 cycles (120ns) if the cache line to be replaced needs to be written back before it can be filled (a read or write burst takes 60ns). This variability is not a difficult issue, in particular it must not be confused with "non-determinism" (the G2 behavior is deterministic): it just puts a greater emphasis on the exhaustive determination of the WCET path(s) and on the security margins applied on the measurements.
- For an interrupted task, the problem is different, and we can really speak of "non-determinism", because the preemption points are really "non-deterministic":
    - When a task is interrupted, the proportion of "dirty" (defined in section 4.1) cache lines is random, so the next task – even if it starts from the beginning and makes always the same accesses - will have to evict a random number of cache lines: the standard solution in real-time systems is to ensure a "clean" cache at each context switch.
    - When a task is restarted, the miss ratio will be increased (compared to the same execution of the task, but non-interrupted) in a hard-to-predict way: we have a cold-start in the middle of the execution, because useful lines have been lost due to the former preempting tasks (Linux/Windows case) or due to the scheduler (real-time case). The key notion of "useful cache block" has been defined hardly 15 years ago [Lee96], and since it is still an active research area, not yet operational in the commercial WCET tools (to our knowledge).

This cache refill cost after a preemption is both hard-to-predict and not-easily-diminished:

- Software-based cache partitioning [Mueller95] is very restrictive for the compilation chain and very memory-inefficient with our constraints of spatial partitioning.
- Hardware-based cache partitioning [Liedtke97] is not easy in the G2 core, because the 4 cache ways are not individually lockable, and a locked way doesn't survive to an invalidation.
- The explicit choice of preemption points [Simonson95] is not compatible with the constant evolution of the software (the sub-tasks limits move at each new version).

With the performance ratio of 30 in mind, even if it is sometimes still practiced [Rodríguez03], the approach of disabling the caches to ensure an easy WCET estimation is completely unrealistic in our context. So we will have to assume that at each preemption, we have to refill completely the cache. This cost has to be multiplied by the maximum number of preemptions: this number is also a current research topic [Burguière09], but for this, we can do a simple thing: make the scheduling deterministic, in such a way to make the number of preemptions constant and statically computable. This is the topic of the next paragraph.

3. Deterministic scheduling

3.1 Foreword: What is a "periodic task" ?

Before describing the new deterministic scheduling, we will shortly describe a key point of our scheduling policies, to make easier the understanding of the next sections.

Generally (and fortunately), real-time schedulers don't change the design patterns used in non-real-time computing, they just adapt them. This is typically true for communication (semaphores, …), this is also true for the very basic notion of "periodic task". For example with an [ARINC-653] scheduler, a periodic task looks like:

```
entry:  while(1) {
                applicative_state_update();
                PERIODIC_WAIT();
        }
```

where the last call is a system call meaning "suspend me until the next period". The scheduler calls the entry point only once in the very first period, and then its an endless repetition of suspend/resume at the level of the system call. In particular it means that at this place the context is saved and then restored, at each period.

Our notion of "periodic task" follows another pattern:

entry: applicative_state_update()

and the repetition (the former while loop) is done by the scheduler, which calls the entry point at each period. We say that the task is "started" at each period, or that each period contains a "run" of the task. In particular no context is saved nor restored between successive runs. On a high frequency task (1ms period), the CPU saving reaches 2%.

3.2 Principle

[ARINC-653] has been the initial source of inspiration: it can be summarized as the avionics "multi-user time-sharing system", where "user" means "equipment supplier". Each supplier has a statically specified periodic set of time-slots (a time-slot is a time interval defined by its offset from the start of the period and its duration) called a partition, and an ARINC-653 scheduler is able to ensure robust partitioning between the partitions. Inside a partition, multi-tasking is performed with a priority-based preemptive scheduler. Inter-partition communication must use messages (addressed to partition: a specific task cannot be referred to). Intra-partition (task to task) can also use classical shared areas with mutual exclusion dynamically guaranteed by semaphores, together with task priority changes to avoid task interlocks.

We don't have the multi-supplier constraint, so to avoid a needless complexity (coexistence of partition scheduling and task scheduling, priority inversion) and to maintain the efficiency of our classic scheduling (task-level communication between partitions, based on shared areas statically protected [section 5]), we had the idea to keep only the static philosophy of the partitions and to apply it to the tasks: the only difference between our classic rate-monotonic scheduler and the new one is that tasks execute now in a constant time, and formally, like ARINC-653 partitions, tasks are now static periodic sets of time-slots. More precisely, a task of period T can be defined as follows in any interval [nT, (n+1)T[:

- A non-interrupted task is a single time-slot,
- A task which is pre-empted k times is a sequence of (k+1) time-slots.

The notion of partition still exist: it is reduced to a set of tasks such that, as soon as one of them fails, the other ones must also be declared faulty.

The implementation is quite simple : instead of giving back the control to the scheduler at the end of the applicative computations of a task, tasks always terminate in an empty endless loop, the "IDLE" loop, and it is the watchdog timer programmed with the deadline of the task which transfers the control to the scheduler. This way, tasks have a constant duration, equal to their deadline. In other words, the deadline watchdog is promoted from a safety status to a scheduling status. The negative aspect is the lost time in the IDLE loop, but first it can be used for "background" jobs like memory BIST, and second in a DAL A certification context we considered it was a very "attractive lost".

This new scheduler is named "DMS" for Deterministic Monotonic Scheduler. A good summary of its specificity is:
"***At any time, we know who is running***".
Together with the cache policy (next section), it is the key property for obtaining:

- a constant number of preemptions for tasks, which permits a safe but fine task-WCET estimation,
- a compositional estimation of the global CPU-load, because the task-WCETs can be estimated independently.

It is the result of:

- our strict rate-monotonic scheduling policy (no aperiodic task, no priority change),
- tasks of constant duration,
- scheduler system calls of "nearly constant" duration (next subsections).

3.3 Implementation

Like all preemptive schedulers in the real-time domain, our standard scheduler is based on the concept of "tick scheduling": preemptions are initiated by a periodic InTerrupt ("IT"), triggered by a Real-Time Clock ("RTC", "HTR" in French), with a fixed period typically equal to 1ms or 2ms on our systems.

The new scheduler is based on an additional IT, the Deadline IT (previously the "watchdog" used to enforce deadlines). We call it also the DECrementer ("DEC") IT, because it is trigged by an internal G2 timer called the "decrementer", programmed by the scheduler at each task switch.

The job of the RTC IT is to "preempt and schedule", i.e.:

- a) save the context (registers, among them the decrementer) of the pre-empted task,
- b) declare "eligible" (i.e. ready to be scheduled) the tasks for which the current time is a multiple of their period (2ms tasks are re-elected every 2ms, …),
- c) choose the next task to (re-)start: it is the eligible task which has highest priority,
- d) configure the MMU for this task,
- e) launch the task; it be either an initial launch, and in this case no context restoration has to be done, or a re-launch if

the task was interrupted (pre-empted), and in this case the context has to be restored.

The job of the DEC IT is to “terminate and schedule”, i.e.:

- a) declare the finished task as “no more eligible”, and if applicative computations were not terminated, declare a fault at the task and partition level,
- b) perform the c/d/e actions of the RTC IT.

A task of period T is in one of the following states:

STATE NAME	STATE DEFINITION
INIT	Initial state before real-time
ELIGIBLE	at the beginning of a new period T
RUNNING	when the task is running on CPU board. At any time, only one task is in EXECUTE state
INTERRUPTED	when the task is pre-empted by the RTC IT
WAITING	when the task is terminated by the DEC IT, and its application has finished in time
FAILED	when the task is terminated by the DEC IT, and its application has not finished (CPU overhead error)

Table 1 : task states

The state transitions are as follows:

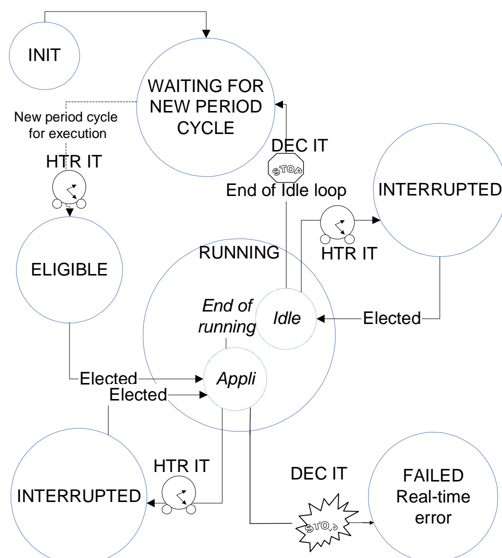

Figure 1 : task state automaton

We illustrate the scheduling with the following configuration:

- the RTC has a 1ms period,
- task 1 has a 1ms period and 480us deadline,
- task 2 has a 10ms period and 750us deadline.

The DMS “phantom task” is the background task that is launched when no more applicative task is eligible.

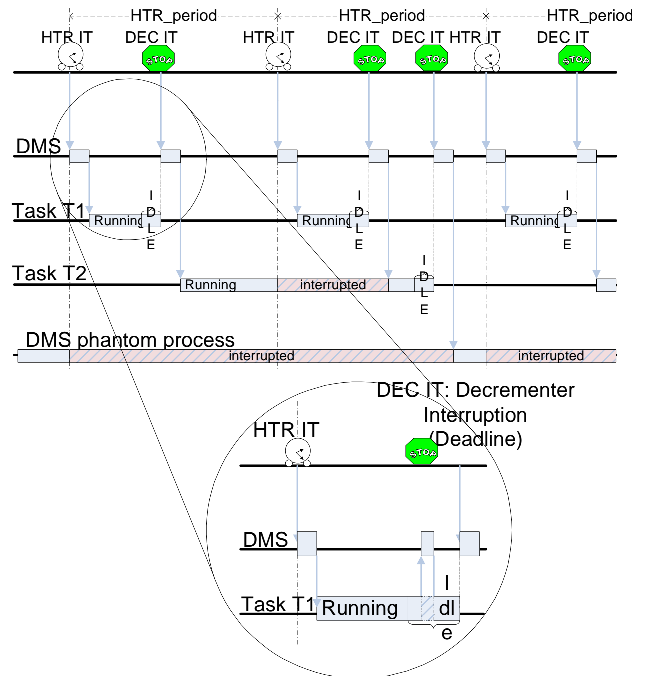

Figure 2 : scheduling example

It is visible in the previous figure that the two DMS handler have a non-negligible duration (a few tens µs), which must be taken into account to determine correctly the time-slots, and in particular the number of preemptions.

3.4 What means “deterministic” ?

This section clarifies the claim “at any time, we know who is running”, explaining that, strictly speaking, it is false, but also how it can be made practically all the time true, and in any case sufficiently true to be able to demonstrate a constant number of preemptions.

The first problem comes from the non-interruptibility of the two IT handlers: if the RTC IT occurs while the DEC handler is running, the RTC handler has to wait the end of the DEC handler. We say that the RTC handler has been “pushed”.

In fact it is not a real problem, because we know when Deadlines occur, so we can schedule in such a way to avoid this case.

The second problem comes from the variability of the duration of the two IT handlers. This variability does not come from their design, because we have followed “WCET-oriented” principles first advocated

in [Puschner02]. It comes from the architecture of our PowerPC: the internal bus is not exclusively hold by the G2 core, but is shared with a communication core. With a high-speed network like AFDX, it can steal a few percents of the internal bus bandwidth available for the G2 core, in a completely asynchronous way. A solution could have been to terminate the handlers with a timer-polling loop to enforce a constant duration, but it has not been necessary: variations are small enough to be manageable.
This handler variability generates a variability on the Deadline instants (not on the task durations), and this variability increases with the length of the task (more precisely, with the number of preemptions) and inversely with the priority of the task (a task is launched after the tasks with higher priority).

In fact, a quick analysis shows that the property “constant number of preemptions” is implied by “no RTC handler pushed”. So, to ensure a “deterministic” scheduling, a sufficient criterion is to determine the time intervals where the Deadline handler may be running, and to check that they don’t contain any RTC IT.

4. Cache management policy

4.1 Foreword: Cache configuration w.r.t. certification

Each writable data area declared to the MMU must be configured in one of the two following modes:

- “write-through”, means that every write is “simultaneously” done in cache and in memory,
- “copy-back”, means that writes are done only in the cache, and memory updating is performed later (possibly while the execution of another task). A data in cache but not yet in memory is said “dirty”, and a cache containing at least one such data is said “dirty”.

The action of updating memory with all the dirty data is called a “flush” of the cache. It takes a non negligible time essentially proportional to the amount of dirty data (between 1 and 32µs on our hardware configuration). It must not be confused with an “invalidation”, which is an instantaneous reset of the cache: any dirty data is lost.

The delayed memory update in copy-back mode is a violation of time partitioning (not of spatial partitioning, as is sometimes found). That’s why a clean cache is recommended at each scheduling. Moreover, if some data or code have survived between two successive invocations of a task, the second invocation may be quicker, and this can lead to optimistic measurements. This is also a violation of time partitioning, so it is recommended on a task launch that the cache doesn’t contain any task data or code. When a cache satisfies this last property and is clean, we will say that it is “neutral”.

4.2 Copy-back at the end of the tasks

The standard and easy way to ensure a neutral cache is to invalidate it, which implies that the cache must already be “clean” at the schedule point. So:

- if the task is not interrupted, the application can work on copy-back areas, provided that it explicitly flushes the cache at the end;
- if the task is pre-empted, the application must run in write-through mode only: the impact on CPU time can be significant.

But with a deterministic scheduling, the scheduler knows if the task it currently launches will be pre-empted or will terminate before the next RTC IT. With this information, we can launch tasks either in write-through mode (on an intermediate time-slot) or in copy-back mode (on the final time-slot), and in this last case an explicit flush of the cache is performed at the end of the task (the flush time is counted in the execution time of the task).
It is a generalization of the fact that non-interrupted tasks can easily run in copy-back mode (with an explicit flush at the end).

4.3 Copy-back unlimited

The remaining problem is to ensure a clean cache before the invalidation in the RTC handler. The obvious solution is to ask the RTC handler to flush the cache. But a cache flush takes a very variable time, and this will destroy the deterministic aspect of the scheduling.

The solution is to perform the flush in a constant time: like tasks, it is followed by an idle loop, not infinite but waiting for a defined duration (32µs).

On a 1ms RTC period, it represents 3% of CPU-time: much less than the potential gain of the copy-back mode.

5. Lock-free communication

5.1 Synchronous communication

The constraint of purely periodic tasks simplifies drastically the mutual exclusion problem. Schedulers permitting aperiodic tasks (like ARINC-653) are forced to dynamically protect the shared memory accesses (more generally the critical sections), typically with semaphores. This dynamic aspect makes task interlocks unpredictable, which makes necessary priority inversion mechanisms.
With periodic tasks (harmonic periods and rate-monotonic scheduling help also), potentially critical

sections become predictable. Moreover, our systems use the simplifying paradigm “one writer / many readers”.

When a writer is writing into shared memory, it must not be preempted by its readers because they could obtain a mix of old and new values. Symmetrically, when a reader is reading into shared memory, it must not be preempted by its writer because the reader could again obtain a mix of old and new values. With fixed priority rate-monotonic scheduling, we don’t know exactly when all preemptions occur, but we know when they don’t occur: a task can never be preempted by lower-priority tasks, so it can never be preempted by slower tasks. The consequence is that the following protocol is safe:

Between a fast task T1 and a slow task T2,

- T2 can always communicate (read and write their shared variables) without taking any precaution,
- T1 communicates only in its First run in a T2 period, and T1 knows that it is in such a particular run by evaluating the system predicate “First(T2)”.

The following figure illustrates this with a fast task running a the RTC period and a slow task running 3 times slower. Vertical arrows indicate reads and writes to a shared memory.

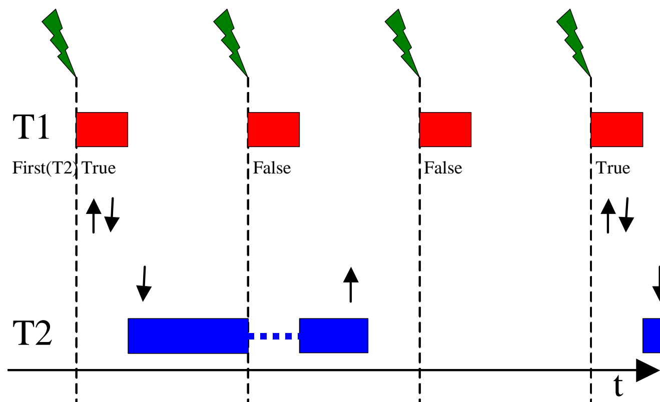

Figure 3 : the “First” predicate

This is very similar to the way data-flows on different clocks communicate in synchronous languages like LUSTRE (see the operators “when” and “current”, [Caspi87]).

5.2 The « Last » primitive becomes static

To minimize latencies, it is sometimes useful to a fast task T1 to be able to communicate with a slow task T2 not in its first run, but in its last run in a T2 period. To know that it is in this run, T1 can evaluate the “Last(T2)” system predicate. Of course, the underlying assumption is that T2 will have finished its execution when the Last run of T1 begins. This is a fundamental difference with First, where the absence of preemption was ensured by the very principle of rate-monotonicity. This assumption had to be demonstrated with timing measurements.

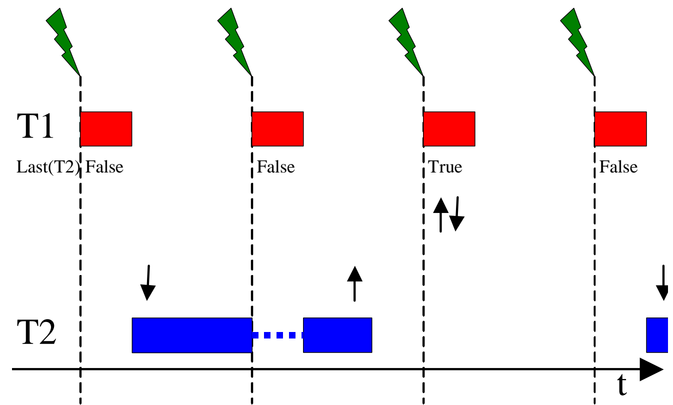

Figure 4 : the “Last” predicate

Deterministic scheduling changes that: we know when preemptions occur, so we are able to statically check that T2 will not be preempted by the last run of T1.

6. Conclusion

The static determination of the number of preemptions allows to secure the WCET estimations (obtained by test or static analysis) with a justified margin.
Nevertheless, the cost of a complete cache refill is very expensive and clearly over-estimates the real cache-related preemption cost: even with an exact determination of the number of preemptions, the price to pay is very high, and confirms the urgent need of having a good estimation of the preemption costs in WCET tools.

This analysis has brought us to wonder about the validity of the partitioning recommendation, more precisely: what are exactly the feared events it must block ? Our impression is that it must address events with unpredictable effect (like a CPU overload of a task), and that the expulsion of dirty lines of a cache could be considered out of the scope, because the impact is well-known: maximum 32µs. So, to ensure safety without any cache invalidation or flush is possible: we simply have to add the good margins on the WCET estimations. This is a good subject for the current decade.

7. Acknowledgements

The authors acknowledge Frédéric Titeux for technical discussions around the fact that a strict “determinism” is not the only answer to the DO178B

(but of course the authors think that it is the best), and François de Virel for having patented with lightning speed, allowing us to have a substantial paper.

8. Annex: harmonic periods

A set of periods is said “harmonic” when for any periods T1 and T2, either T1 divides T2 or the opposite.
This a fundamental property which is often overlooked, but which permits to check very simply the schedulability of a task set:

$$\sum_i D_i/T_i < 1 \qquad [2]$$

(D_i is the deadline and T_i the period of the task number i).
This was demonstrated for the first time in [Lehoczky91] (theorem 3.4), but we can find traces of this in one of the first papers of the discipline: [Liu73] (the remark on the “utilization factor” between theorems 3 and 4).
The demonstration of [Lehoczky91] is complex, because it starts from a more general case with non-harmonic periods. The engineer is often surprised by this complexity, because in the harmonic case, the expression $\sum_i D_i/T_i$ is simply the definition of the CPU load.

Among the practical interests of this constraint, we can mention the absence of beats: informations transmitted between tasks of different periods have always the same freshness.

9. References

[Agarwal89] A. Agarwal, M. Horowitz, J. Hennessy. *An analytical cache model*. ACM Trans. On Computer Systems 7 (2), May 1989.

[ARINC-653] ARINC: “Avionics application software standard interface”, first ed. Oct. 1996, current ed. March 2006.

[Burguière09] C. Burguière, J. Reineke, S. Altmeyer. *Cache-related preemption delay computation for set-associative caches – Pitfalls and solutions.* WCET’09, June 2009.

[Caspi87] P. Caspi, D. Pilaud, N. Halbwachs, J.A. Plaice: “*LUSTRE: a declarative language for programming synchronous systems*”, 14th ACM Symposium on Principles of Programming Languages, 1987.

[DO-248B/ED-94B] RTCA, EUROCAE: “*Final report for clarification of DO-178B/ED-12B*”, Oct. 2001.

[Lee96] C. Lee, J. Hahn, Y. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, C. S. Kim. *Analysis of cache-related preemption delay in fixed-priority preemptive scheduling*. 17th RTSS, Dec. 1996.

[Lehoczky91] J.P. Lehoczky, L. Sha, J.K. Strosnider, H. Tokuda: “*Fixed priority scheduling theory for hard real-time systems*”, in “*Foundations of real-time computing*”, A.M. van Tilborg, G.M. Koob eds., Kluwer Academic Publishers, 1991.

[Liu73] C.L. Liu, J.W. Layland: “*Scheduling algorithms for multiprogramming in hard realtime environment.*“, Journal of the ACM, 20(1), February 73.

[Liedtke97] J. Liedtke, H. Härtig, M. Hohmuth. *OS-controlled cache predictability for real-time systems.* 3rd RTAS, June 1997.

[Mogul91] J. Mogul, A. Borg. *The effect of context switches on cache performance*. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, April 1991.

[Mueller95] F. Mueller. *Compiler support for software-based cache partitioning*. ACM SIGPLAN workshop on Languages, Compilers and Tools for Real-Time Systems, June 1995.

[Puschner02] P. Puschner, A. Burns: “*Writing Temporally Predictable Code*“, 7th IEEE Int. workshop on Object-Oriented Real-Time dependable systems, Jan. 2002.

[Rodríguez03] M. Rodríguez , N. Silva , J. Esteves , L. Henriques , D. Costa , N. Holsti , K. Hjortnaes. *Challenges in Calculating the WCET of a Complex On-board Satellite Application.* WCET’03, 2003.

[Sebek02] F. Sebek. *Instruction cache memory issues in real-time systems.* Licentiate thesis of Mälardalen University, October 2002.

[Simonson95] J. Simonson, J.H. Patel. *Use of preferred preemption points in cache-based real-time systems.* IPDS'95, 1995.

[Wilhelm07] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström. *The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools.* ACM Transactions on Embedded Computing Systems (TECS)*, 2007*

10. Glossary

AFDX: Avionics Full DupleX (“deterministic” and redundant 100Mbps Ethernet)

BIST: Built-In Self-Test

DAL: Design Assurance Level

DEC: DECrementer

DMS: Deterministic Monotonic Scheduler

HTR: Horloge Temps Réel

IT: InTerrupt

MMU: Memory Management Unit

RTC: Real-Time Clock

WCET: Worst Case Execution Time