Accounting for interferences in the design of Time-Triggered Applications

Antoine FERLIN, IRT Saint-Exupéry, <u>antoine.ferlin@irt-saintexupery.com</u> Eric Jenn, Thales AVS and IRT Saint-Exupéry, <u>eric.jenn@irt-saintexupery.com</u> Marc Kaufmann, Safran Electronics and Defense, <u>marc.kaufmann@safrangroup.com</u>

Abstract: Multicore processors are making their way into safety critical embedded systems. In order to ensure compliance with temporal requirements, interferences between cores induced by the shared platform resources must be taken into account and controlled. This paper proposes an approach to account for SDRAM interferences in the context of a time-triggered software architecture. The approach is applied on a simple robotic application. **TOPIC**: Embedded computing platforms and networked system/Multi-core/Many-core platforms **Keywords**: Multi-core, Interference, SDRAM, LET

1 Introduction

Multicore processors are rapidly making their way into safety critical embedded systems, pushed by an ever increasing need for more computing power in less space and with less energy. If computing power and energy were the only drivers, we would simply use – or get inspiration from – the numerous computing platforms used in the general market for desktop computing or mobile communication or gaming. However, our embedded systems must be dependable, and a prerequisite to dependability is the capability to demonstrate the compliance with all requirements, and in particular those directly or indirectly related to time.

Demonstrating compliance with temporal requirements requires *at least* an accurate – and possibly a precise – estimation of software execution times, and more specifically, of their *worst-case values* (WCET).

On the hardware side, it is well known that obtaining these estimations on a modern high-end processor is extremely difficult due to the presence of the numerous, complex, and sometimes not so well documented mechanisms aimed at reducing the mean execution times. Some of those mechanisms are part of the processing units, such as caches, branch prediction and speculative execution units, out-of-order execution pipeline, etc. [1]–[5]. Some others may be located out of the processing units, such as interconnects, communication accelerators (e.g., NXP's Datapath Acceleration Architecture (DPAA)), or Synchronous Dynamic RAMs (SDRAM). All these mechanisms make WCET estimation complex for the very reason that they are often shared resources between applications running concurrently on different cores. In addition, their behaviours are often complex (e.g., branch prediction and speculative execution) and their documentation is sometimes incomplete or not available. In any case, the effects of these hardware mechanisms on execution times have to be estimated and, possibly, bounded, using analytical or empirical means. In this paper, focus is placed on interferences caused by the SDRAM.

On the software side, compliance with temporal requirements may leverage on appropriate programming abstractions, such as ZET (Zero Execution Time) or LET (Logical Execution Time) synchronous programming models [6], [7]. The key feature of the synchronous model is to provide an abstract, ideal, and mathematically tractable view of time. That way, the programmer does not have to consider the physical time at which computations occur, but the logical time at which they occur. With an appropriate choice of this logical time, programming, compilation, debugging, and – last but not least – verification, are greatly simplified. In the ZET abstraction, the execution platform of a program is *conceptually* considered to run infinitely fast so that each job (task instance) executes instantaneously, at specific *ticks*. In the LET model, inputs and outputs are performed at *tick* boundaries too, but the job itself may take place at any time between two successive ticks. Note that, even though this model of computation strongly constrains inter-task communications, it does not prevent interferences at SDRAM level since accesses to SDRAM may indeed occur at any time during the execution of a job.

In this paper, we consider a time-triggered implementation of the LET model [8] in which events are triggered by realtime. Giotto, is an example of an implementation of such model [9].

2 Our contribution

This paper proposes to combine two engineering techniques in order to improve the temporal guarantees given to an application developer: (i) the Krono-Safe's Asterios technology¹ which implements the time-triggered LET model, and (ii) an approach to estimate the delays due to interferences caused by concurrent accesses to SDRAM.

The basic principle is the following. The application is designed and a time budget is allocated to each of its concurrent components called *"agents"*. A time-triggered schedule is generated using Krono-Safe's toolset. This schedule and a model of the SDRAM is used to estimate the delays induced by SDRAM accesses. This estimation is then used to point out the *"hot spots"* of the software architecture, i.e., the time intervals that show numerous conflicts and for which the budgets of the interfering tasks need to be checked or possibly corrected. This information can also be used to modify the design so as minimize interferences, and reduce delays.

The rest of this paper is organized as follows: Section 3 presents the time-triggered implementation model; Section 4 gives an overview of the related works; Sections 5 and 6 presents our analysis method and its application; finally, Section 7 concludes the paper.

3 The time-triggered model

The time-triggered implementation of the LET model is one of the models selected by the CAPHCA research project² to ensure the dependable programming and deployment of parallel applications on multicore platforms [10]. In practice, we use the implementation of this model provided by Krono-Safe's Asterios environment [11].

Two concepts are worth being introduced: Agent and Repetitive Sequence of Frame.

- An **agent** is an autonomous unit of processing programmed in Krono-Safe's PsyC language. It is composed of a specification of the agent's communication interface and a description of the task to be executed. The task is described using a combination of C processing statements (actions) and statements expressing temporal constraints (*advance* statement). Each agent is cadenced by a logical clock: between two consecutive ticks of this clock, an instance of the described task, called job, is executed. All logical clocks of the program are derived from one unique physical clock provided by the Asterios environment or by another external source.
- A **Repetitive Sequence of Frame** (RSF) is a time-triggered sequence of jobs that is repeated during the program execution. The RSF is built from the description of the agents and from the time budgets allocated to them. Hence, the RSF is valid only if the defined time budgets are valid. Note that in case of budget violation, the error is confined to the agent without any impact on the other agents. In addition, a spatial partitioning mechanism ensures that each agent only accesses its own memory space defined at configuration time.

An example of a RSF is given on Figure 1. It shows the jobs of 14 agents executed on the two cores of the execution platform. Each agent is executed every 1ms except for the Ethernet ones which are executed every 2ms. The length of the RSF in this example is 2ms.



Figure 1 Example of Repetitive Sequence of Frame (RSF)

¹ http://www.krono-safe.com/

² CAPHCA is carried out at the Institut de Recherche Technologique Saint-Exupéry of Toulouse.

4 Related Works

The multi-core processors used for critical embedded system requires to deal with interferences problems. Interferences directly increase the WCET of the tasks, then the WCET optimization could be a means to handle interference problems. A lot of works explore the way to compute the WCET of a multi-core program [1], [2], [4], [12]. The major problem is the computation of an accurate over-approximation of real WCET of a multi-core application. In addition, sources of latency are multiple : SDRAM [13], interconnect [14], ...

A memory interference is an access conflict between two agents at SDRAM level. In addition, occurrences of interferences cannot be precisely predicted within the time-line of the program. [15] bypasses this problem by defining a probabilistic WCET, built using a set of experimental measurements on executed tasks. Based on measurement during executions of the application, a bias can be inconveniently introduced because of lacks of test coverage. Another important idea of this method is the use of the scheduling to compute the WCET. [16] defines the notion of superblock, a non-pre-emptable task instance with a defined memory model. It considers then three memory models dedicated to the analytical computation of the WCET and shows the importance of a design that separates computations from accesses to SDRAM. Combination of different static analysis means such as abstract interpretation and model checking has also be explored by [17], to estimate L1 cache load, and then SDRAM-and-bus loads. Transition to industrial scale may be jeopardised by the requirement of annotations at statement level with cache hit/miss classification.

Another research field consists in merging several techniques in order to build a program with a limited number of memory interferences by construction. [18] looks at schedulability from a different perspective of our works. Instead of analysing interferences to ensure schedulability, they analyse programs to detect interference in case of non-schedulability. Results are based on several empirical and formal analyses. [19] and [20] focused on paralleling program written with synchronous languages as LUSTRE. Interferences are taken into account by computing delays due to read and write memory accesses. Once integrated into the WCET computed in isolation, a scheduling is then computed, and the obtained program is correct-by construction. Use of WCET enhanced by interference delays instead of time budget is the main difference between these works and our approach. In our opinion, interference delays are always pessimistic, which has a high impact on the computed schedule.

Because of difficulties to compute a precise WCET to handle the interference problem, our work focuses on the definition of a metric to quantify interference instead. [12] proposes an analysis of the dependence of tasks using direct acyclic diagrams as task model. Interferences are not studied at hardware level but at software level. In our works, time-triggered abstraction allows the limitation of interference due to task dependences. [13] precisely describes architecture of a SDRAM component and provides a methodology to compute interference delays according to the memory characteristics. But, resulting delays are computed at core level and not at task level. This granularity is not sufficient to validate time budget of each agent. [21] similarly computes interference delays, but because it does not precisely define the delay per memory access with parameters of the SDRAM datasheet, it is not as precise as [13]. In addition, the scheduling is not precisely taken into account, for the both methods.

Our work aims at defining an interference delay, handled as a cost function to be optimized. Hence, we use [13] to compute a delay per memory access, but we have to define a precise method which takes the RSF into account.

5 Accounting for interferences

5.1 Overview

Our interference analysis covers steps 3 and 4 of the five-step process depicted on Figure 2:

- The first step consists in developing the high level architecture of both software and hardware using AADL. The second step consists in translating the AADL model to generate an initial, architectural, PsyC model that will be completed manually. The bridge is based on the OSD metamodel developed by Krono-Safe within the S3P collaborative project³, using model engineering techniques. Additional C libraries referenced by the AADL model have to be developed separately.
- The **third step** concerns the computation of the *RSF* by Asterios. A schedulable program is followed by its compilation, otherwise the process has to restart from the first step.
- The **fourth step** consists in extracting the information from the RSF required to compute the interferences. The output of this step is an evaluation of the delay caused by the interferences.

³ Smart, Safe and Secure Platform : http://www.s3p-alliance.fr/

• Finally, the AADL model is enriched using the output of the previous step, in order to reduce the interference level. The process can be repeated until obtaining an acceptable level of interferences.

The objective of step 3 and 4 is to point out underestimated time budgets (or correct those budgets) considering the estimation of the interferences that may occur during the execution of the application's agents. This process, represented by the grey arrow on the figure, is iterative. Starting from an estimation of the budgets, an initial RSF is generated. This RSF allows a temporal profile of memory accesses to be computed, from which interference delays are estimated. As we will see later, the estimated interferences may either be used to update the software architecture in order to remove the interference "hot-spots", or to update the budgets. In both cases, the process is iterative since a modification of the budget or a modification of the architecture may both lead to a new RSF, so to a new interference pattern. This point will be discussed later, in section 6.



Figure 2 Overview of the approach

5.2 Interference analysis

Any resources shared by the cores of a multi-core processor is potentially a source of interferences. We focuses our analysis on the SDRAM component which is an important contributor to interferences. Even if memory caches partially mitigate SDRAM interference, we will consider these ones inhibited. This point will be also discussed in section 6.

[13] gives a method to compute an upper bound of the delay that suffer a core of a processor on a given period. However, the result is not sufficiently precise to optimize interference delays. Indeed, a global delay on a given core lead to proceed by trial and error to understand its origin.

Therefore, we provide an *agent-driven* approach, to compute the delay for each agent. This delay is computed for the *hyper-period* of the agent set, i.e., the lowest common multiple of the agent periods. This new approach is expected to be more precise than the *job-driven* one proposed in [13] since conflicts between agents can be localized more precisely, thanks to the RSF. The *agent-driven* approach requires the *request-driven* approach of [13] and can be seen as a fork of the *job-driven* approach. Hence, the maximum number of memory accesses per job, the SDRAM characteristics and the application RSF are required as input.

The computed delay is a cost functions which has to be optimized to limit interferences. Instead of working at core level as in [13], we aim at being able to select the agent to be optimized. Finally, delay computation first requires the formalization of a memory model.

5.2.1 Formalization of the memory model

An application is composed of a set of agents \mathbb{T} executed on a predefined core. *Jobs*, are executed during the RSF hyperperiod of duration *HP*. The period of an agent τ is $P(\tau)$. A job can be split into several job parts because of agent preemption. Interference delay will be computed over the hyper-period: all following elements will be defined within this duration. $J(\tau)$ is the set of jobs executed, related to the agent τ . In Figure 3 for instance, \mathbb{T} is composed of the agents in blue, orange, yellow, green, grey and dark-blue. Grey jobs are pre-empted by dark blue jobs.



Figure 3 Example of RSF elements and memory profiles

We aim at computing the Worst Case of Memory Access Delay of a given agent ($WCMAD(\tau)$), which includes the number of jobs executed during the hyper-period:

$$WCMAD(\tau) = \max_{j \in J(\tau)} (delay(j)) \times \frac{HP}{P(\tau)}$$

The delay generated by a job j depends on the maximum latency of one memory access (denoted by RD) and on the WCMA - Worst Case number of Memory Accesses - required by concurrent jobs. [13] already provides a maximum delay (RD) of one memory access depending on the characteristics of the SDRAM and on the bank organization. Indeed, a bank shared by several cores deteriorate the SDRAM delays. Consequently, RD depends on the core. Delay is expressed by the following formula:

$$delay(j) = \sum_{j' \in \text{conflict}(j)} RD_{core(j')} \times WCMA(j')$$

In order to evaluate delay(j), we need to estimate:

- the Worst Case number of Memory Accesses (WCMA(j')),
- the set of conflicts between jobs (conflict(*j*)).

5.2.2 Evaluation of the number of memory accesses

WCMA(j') can be evaluated using the same means as for WCET estimation: analytically on a model, or empirically using a probe on the actual hardware or on a simulation model. In the first case, WCMAs could be a by-product of a static WCET analysis used to estimate the time budgets. In the second case, WCMAs could be estimated using the Measurement Based Probabilistic Analysis technique (MBPTA) based on the Extreme Value Theory [22]. In our experiments, we rely on the classic High Water Mark approach.

5.2.3 Conflict between two jobs

A conflict between two jobs is conservatively considered to happen when both jobs are executed at the same time. If $t_{iob}(x)$ is the interval of execution of job x, then we have:

$$conflict(j) = \{j' \in \mathfrak{J}, j' \neq j, t_{job}(j') \cap t_{job}(j) \neq \emptyset\}$$

In Figure 3, the red areas depict the conflicts between the blue job and the grey job.

In the absence of a precise memory access profile, all memory accesses are conservatively considered to occur in sequence at any time during the execution of the job (Figure 3, sample memory profile). Hence, WCMA(j') will matches with the sum of all memory accesses occurring during the job execution.

A more precise memory model would require defining the exact time at which memory accesses occur during the job execution (Figure 3, precise memory profile). If t_{mp} denotes the set of all time intervals where memory accesses occur (assuming that the intersection of two sets of time interval is the set of non-empty intersections of their elements two by two), then we have:

$$conflict(j) = \{ j' \in \mathfrak{J}, j' \neq j, t_{mp}(j') \cap t_{mp}(j) \neq \emptyset \}$$

In that case, WCMA(j') will correspond only to the sum of concurrent memory accesses. Finally, the definition of a thin memory models only depends on the ability to measure the number of memory accesses with their time of occurrence during the execution of the job.

After computing the interference delay for each agent, we must select the conflicts to deal with. We propose to help developers by automatically identifying the set agents requiring a particular attention, i.e., the set of agents for which the delay is greater or equal than a threshold T and which budget must be analysed carefully.

The set is defined as follows: instead of using delay of an agent, we deal with delay of its jobs. A matrix M stores the conflicts such as M[i][j] represents the delay generated on job i by job j (M[i][i] = 0). If M[i][j] and M[j][i] are both upper than the threshold value, then the conflict between the both jobs must be avoided. Note that if one of both values is upper than the threshold value, then the conflict should be avoided "if possible".

This method only provides an approach to decide the conflicts which have to be resolved first. Note that for a critical software, all conflicts must be analysed carefully. Finally, intuitively, two agents of a program with a high level of memory access should not be executed simultaneously.

6 Experiments and Results

6.1 The TwIRTee Case Study

Our approach has been applied and evaluated on the software of a 3-wheel robot – TwIRTee – currently under development at IRT. TwIRTee's role is to guide visitors from the IRT building entrance desk to a designated meeting room or office. It is used as a demonstrator for various research activities on dependable embedded systems, including Design Space Exploration [23], formal verification [24] etc. TwIRTee is composed of two redundant channels configured as MASTER / SLAVE. The application handling the master/slave management has been retargeted for the Asterios and T1040 platform.





An overview of TwIRTee functional model is given on Figure 4. The complete application is composed of the following functions:

- Redundancy manager selects the role (master or slave) of each channel.
- *Mission manager* is used to compute the missions of the robot, i.e., the sequence of waypoints to pass-by to move from a source point to a destination point ;
- Localization locates the robot using odometry, in-door UWB positioning system sensor data, hybridized using a Kalman filter;
- Tracking computes the longitudinal and rotational speed setpoints to follow the programmed track;
- Speed selector selects the robot's speed according to the applicable constraints;
- *Simulator* (GPS, compass, odometer) is used to simulate TwIRTee sensors / actuators. Basically, it computes the future position according to the current position and the selected speed.
- *Ethernet* manages the communication between the redundancy managers of both channels.

As a remark, Figure 1 is the RSF of TwIRTee application without enhancement to limit interferences. The next section precisely describes the different steps of one iteration. Results and convergence of our method also will be discussed.

6.2 WCMA estimation

Instead of using a probe on the hardware to estimate the WCMA of each agent, we have used the VLab [23] virtual platform to run the unmodified binaries of the embedded software and to count the memory accesses (fetch, write, read) using VLAB's advanced debugging and tracing features. Table 1 provides WCMA results for each agent.

Agent	Fetch	Read	Write
Redundancy Manager	99 891	39 118	27 333
Mission Manager	1 841 841	732 311	556 781
Localization	290 389	113 262	58 411
Tracking	175 948	72 214	56 161
Speed Selection	71 935	27 964	19 483
Simulator	85 181	26 671	16 842
Ethernet	11 645	4 514	4 329

Table 1 Number of the memory accesses according to the studied agent.

As a remark, the Mission Manager seems to require a lot of memory accesses. Additional investigation shows that the peak number of memory accesses occurs during the initialization phase, when the map is consulted to compute the path followed by TwIRTee using the Dijkstra algorithm. In all other phases, the number of memory accesses is much lower (around 200000). Note that the number of accesses for each task is known, we can now compute the maximum delay encountered by each agent on the hyper-period.

6.3 Delay computation

To compute the delay, we have developed a tool – BRaMI – to compute the interference delay. This tool represents around 6400 lines of Python code. BRaMI takes as inputs the memory profile of agents, each profile being characterized by a unique time interval and a number of memory accesses (fetch, load or store) for that interval. The more precise model proposed in the previous section (i.e., a set of (time interval, number of accesses) is not used yet).

Agent	Fetch (ms)	Read (ms)	Write (ms)
Ethernet	87,17	34,22	24,23
Mission Manager C1	51,98	20,67	15,71
Mission Manager C2	103,95	41,33	31,43
Redundancy manager C1	5,64	2,21	1,54
Redundancy manager C2	5,64	2,21	1,54
Localization C2	0,66	0,26	0,24
Simulator C2	0,66	0,26	0,24
Speed selector C2	0,66	0,26	0,24
Tracking C2	0.66	0.26	0.24

Table 2 Delay due to interference memories in ms

In our case, results have been computed using the characteristics of the T1040 SDRAM⁴, the RSF provided by Figure 1 and the memory profile provided by Table 1. Table 2 regroups interference delays computed bin BRaMI.

The conflicts between jobs are reported on Table 3. A threshold value of 2ms is used to discriminate critical conflicts. The number in brackets matches with the id of the job. For instance, Mission Manager C1 and Mission Manager C2 conflict must be avoided in the second part of the RSF. This conflict does not exist in the first part (Figure 1).

Critical conflicts (delay > 2ms)	Other conflicts
Redundancy Manager (0) C1 vs Redundancy Manager (0) C2	Ethernet(0) vs mission Manager C2 (0)
Redundancy Manager (1) C1 vs Redundancy Manager(1) C2	Ethernet(0) vs localization C2(0)
Mission Manager (1) C1 vs Mission Manager (1) C2	Ethernet(0) vs tracking C2(0)
	Ethernet(0) vs speed selection C2(0)

⁴ SDRAM Reference: MTA18ASF1G72AZ-8GB: information can be found in the SDRAM datasheet. Required information to compute delays are described in [13].

Ethernet(0) vs simulator C2(0)
Ethernet(1) vs localization C2(1)
Ethernet(1) vs tracking C2(1)
Ethernet(1) vs speed selection C2(1)
Ethernet(1) vs simulator C2(1)

Table 3 List of conflicts

6.4 Interference reduction

Once the interferences have been detected, the developer can either increase the budgets of agents involved in critical conflicts, or try to reduce the level of interferences.

To achieve this reduction, we have studied several strategies. The main issue is the extreme sensibility of the RSF to a modification of the budgets or a modification of the temporal constraints expressed in the PsyC program. More often than not, a modification completely changes the RSF, which make difficult to solve the memory interference issues. However, two main strategies have been successfully tried: separating accesses in space, or separating accesses in time.

The first strategy simply consists to dedicate each bank of the SDRAM to a single core. As it is possible to precisely allocate a memory region to a given agent, this strategy can be easily applied.

The second strategy consists to separate in time the executions of the conflicting agents.

Asterios provides a mechanism to limit latency as explained in the guide reference [25]. Normally, in a LET application an agent A must wait for the output time of an agent B as described by Figure 5. The zero latency communication mechanism is implemented using *fractional ticks*. A fractional tick attached to a given clock is a virtual tick which always occurs between two consecutive ticks of this clock. The precise position of this tick is only known at the RSF generation phase. This object is then used to position the execution of an agent before or after this fractional tick. In addition, to fully use the zero latency mechanism, temporal variable or stream variable has to be defined using this fractional tick.

A fractional clock basically expresses a temporal constraint (before the tick or after the tick). So it can be used to impose the relative position of the execution of two agents, without modifying the behaviour of the communication means. As a remark, several fractional ticks can be defined for a given clock and then, it is possible to position an arbitrary list of agents if the defined constraints are not inconsistent. In addition, agents using the same fractional tick as reference can be executed by different cores.

Some other strategies can be used in specific cases. One strategy could be to allocate memory-intensive agents to the same core in order to serialize all memory accesses, and suppress interferences. This simple strategy is obviously not always applicable due to the limited processing capability of a core, or due to possible core collocation or segregation constraints among agents. Another strategy was to define a "phantom agent" P which would perform no actual processing but would be simply used to occupy a slot in time and, consequently, prevent any other agent to be executed at the same time on the same core, hence preventing memory conflict with any other agent on another core. In practice, the difficulty was to position precisely the execution of P (without using fractional clocks). In addition, the time allocated to P was definitively lost for any useful computation.



Figure 5 Fractional tick used for zero latency communication

6.5 Discussion

The first import point concerns the fact that we have obtained a delay for an agent greater than the period of the agent itself. This seems to be an inconsistent result, but it is justified by the fact that, in our model, the cache of processor is

considered disabled. Therefore, each memory access goes right to the SDRAM, leading to a very high number of memory accesses and, consequently, a very pessimistic delay.

However, in our case, we are not interested by the absolute value of the delays, since they are simply used to indicate interference hotspots and, later, to reduce them. In addition, we currently assume (but this needs to be justified) that an action on the RSF that reduces memory interferences with the cache disabled, will also reduce interferences with the cache enabled.

In Table 4, we provide a comparison between the *job-driven* approach of [13] and our *agent-driven* approach, on the Twirtee use case. The results of *agent-driven* approach are deduced by summing the agent results.

	Job-driven (in ms)		Agent-driven (in ms) / (job-driven as reference)			
	Fetch	Read	Write	Fetch	Read	Write
Core 0	217,17	85,64	62,23	144,78 (67%)	57,09 (67%)	41,48 (67%)
Core 1	218,48	86,15	62,72	112,22 (51%)	44,56 (52%)	33,94 (54%)

Table 4 Comparison of job-driven approach and agent-driven approach

Because our method is based on a fixed scheduling, we can provide a precise information for each agent and not only for a given core. A comparison of both approaches shows that the order of magnitude is similar. Differences can be explained by three facts: (i) we use a fixed scheduling, (ii) the Ap(t) factor (max number of memory requests generated by the core p during the time interval t) of the job-driven approach is over-approximated, and (iii) the request-driven approach consider that write accesses and read/fetch accesses alternately occurs, even if it is not possible.

The second important point concerns convergence of our iterative method. The main challenge encountered during experiments was to find a predictable modification strategy of the RSF. Indeed, adding a constraint on some agents would completely modify the RSF. As a consequence, our process is currently not always convergent for complex use cases. Extending the toolchain with the capability to temporally segregate or aggregate the execution of set of agents would be useful to support a convergent process. Otherwise, the only solution (not necessarily feasible in practice) consists in exploring all possible combinations of segregation constraints on conflicting agents, without guarantee of success.

7 Conclusion

We have proposed a combination of two engineering techniques: the use of the LET programming model implemented by Krono-Safe's Asterios, and a methodology to estimate the delay due to SDRAM memory interferences. Delays are computed thanks to an adaptation of an existing work, originally with SDRAM characteristics as input and extended to account for the RSF scheduling. Through this combination, we aim at improving the temporal guarantees given to an application developer, in an industrial constrained process.

This method is then included in a global iterative industrial process. Experiments have been performed on a research software representative of real industrial cases, TwIRTee. Results have been compared with the original works and show an accuracy enhancement of computed delay.

But, a current identified block depends on the Time-Triggered LET technology used. Indeed, simplifying the interference limitation step requires a mechanism to specifically exclude or favour concurrent executions to limit the iterations. Hijack existing features not dedicated to interference limitation could be an option for basic cases, but it is not a general solution if it would lead to manually rewrite the program scheduling. Another required feature is the ability to define a memory mapping to avoid SDRAM banks shared by the same core of a CPU, which strongly degrades interference delays.

As a future work, we plan to improve our memory profile, a high source of over-approximation of the memory delays. We consider the use of static analysis to compute a precise profile, instead of using measurement techniques, as in [5]. Indeed, with the worst case and best case of execution time of statement combined with its WCMA, it will be possible to build a precise memory profile. Another important track is the definition of a model including a CPU cache. Indeed, an accurate interference delay cannot be computed without a model which take caches into account. Additional experiments are necessary, especially to use more complex memory profiles. Finally, we hope to compare our interference delay with other existing tools, such as Platform Architect for instance.

Acknowledgements

This work is funded by the CAPHCA project. CAPHCA is supported by the French Research National Agency (ANR).

References

- [1] T. Kelter and P. Marwedel, 'Parallelism analysis: Precise WCET values for complex multi-core systems', *Sci. Comput. Program.*, vol. 133, pp. 175–193, 2017.
- [2] S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, 'A Unified WCET Analysis Framework for Multi-core Platforms', in *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, Washington, DC, USA, 2012, pp. 99–108.
- [3] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, 'Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement', in 2014 26th Euromicro Conference on Real-Time Systems, 2014, pp. 109–118.
- [4] P. Bieber *et al.*, 'A model-based certification approach for multi/many-core embedded systems', in *ERTS 2018*, Toulouse, France, 2018.
- [5] W.-T. Sun, E. Jenn, and H. Cassé, 'Validating static WCET analysis: a method and its application', Stuttgart, Germany, 2019.
- [6] C. M. Kirsch and A. Sokolova, 'The logical execution time paradigm', in *Advances in Real-Time Systems*, Springer, 2012, pp. 103–120.
- [7] S. H. Son, I. Lee, and J. Y.-T. Leung, Eds., *Handbook of Real-Time and Embedded Systems*. Chapman and Hall/CRC, 2007.
- [8] D. Chabrol *et al.*, 'Time-and angle-triggered real-time kernel', in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, 2013, pp. 1060–1062.
- [9] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, 'Giotto: a time-triggered language for embedded programming', *Proc. IEEE*, vol. 91, no. 1, pp. 84–99, Jan. 2003.
- [10] E. Jenn, 'Towards a Dependable Parallelism for Real Time Systems', presented at the DASIA 2019, 2019.
- [11] D. CHABROL, V. DAVID, P. OUDIN, G. ZEPPA, and M. JAN, 'Freedom from interference among time-triggered and angle-triggered tasks: a powertrain case study', presented at the ERTS 2014, Toulouse, 2014.
- [12] F. Guet, L. Santinelli, J. Morio, G. Phavorin, and E. Jenn, 'Toward Contention Analysis for Parallel Executing Real-Time Tasks', 2018.
- [13] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, 'Bounding and reducing memory interference in COTS-based multi-core systems', *Real-Time Syst.*, vol. 52, no. 3, pp. 356–395, May 2016.
- [14] B. Andersson, A. Easwaran, and J. Lee, 'Finding an upper bound on the increase in execution time due to contention on the memory bus in COTS-based multicore systems', *SIGBED Rev.*, vol. 7, p. 4, 2010.
- [15] L. Cucu-Grosjean *et al.*, 'Measurement-Based Probabilistic Timing Analysis for Multi-path Programs', in *Proceedings Euromicro Conference on Real-Time Systems*, 2012, pp. 91–101.
- [16] A. Schranzhofer, J.-J. Chen, and L. Thiele, 'Timing Analysis for TDMA Arbitration in Resource Sharing Systems', in *Real-Time Technology and Applications Proceedings*, 2010, pp. 215–224.
- [17] M. Lv, W. Yi, N. Guan, and G. Yu, 'Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software', in 2010 31st IEEE Real-Time Systems Symposium, 2010, pp. 339–349.
- [18] J. Boudjadar and S. Nadjm-Tehrani, 'Schedulability and Memory Interference Analysis of Multicore Preemptive Real-time Systems', in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, New York, NY, USA, 2017, pp. 263–274.
- [19] K. Didier *et al.*, 'Correct-by-Construction Parallelization of Hard Real-Time Avionics Applications on Off-the-Shelf Predictable Hardware', *ACM Trans. Archit. Code Optim.*, vol. 16, no. 3, pp. 1–27, Jul. 2019.
- [20] J. Souyris *et al.*, 'Automatic Parallelization from Lustre Models in Avionics', presented at the Embeeded Real-Time Systems and Software (ERTSS), Toulouse, 2018.
- [21] J. Jalle *et al.*, 'Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP)', in 8th *European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, 2016.
- [22] L. Santinelli, F. Guet, and J. Morio, 'Revising Measurement-Based Probabilistic Timing Analysis', in 2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017, pp. 199–208.
- [23] P. Cuenot, E. Jenn, E. Faure, N. Broueilh, and E. Rouland, 'An Experiment on Exploiting Virtual Platforms for the Development of Embedded Equipments', in 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016), TOULOUSE, France, 2016.
- [24] M. Clabaut, N. Ge, N. Breton, E. Jenn, R. Delmas, and Y. Fonteneau, 'Industrial Grade Model Checking Use Cases, Constraints, Tools and Applications', in 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016), TOULOUSE, France, 2016.
- [25] 'PsyC reference guide'. Kronosafe, Jun-2017.