An Iterative Benchmark Configuration Method for Quantifying Multi-Core Interference

Sébastien Levieux, Frank Singhoff, Stéphane Rubini
Lab-STICC UMR CNRS 6285, University of Brest, 29200 Brest, France; email: firstname.lastname@univ-brest.fr

Philippe Plasson, Pierre-Vincent Gouel, Lee-Roy Malac-Allain
LESIA, Observatoire de Paris, Université PSL, CNRS, Sorbonne Université, Université Paris Cité, 5 place Jules Janssen, 92195 Meudon, France; email: firstname.lastname@obspm.fr

Lucas Miné, Gabriel Brusq
Centre National d'Etudes Spatiales (C.N.E.S.), 18 av. Edouard Belin, 31401 Toulouse, France; email: firstname.lastname@cnes.fr

Abstract

Interference within a multi-core architecture may have several origins. Understanding where interference comes from is mandatory for verification and certification purposes. Unfortunately, the complexity of current architectures makes it difficult to quantify such interference. In this article, a new approach is introduced that enables benchmark configurations to isolate and quantify interference. An experiment with DMA interference is presented and shows a WCET overhead of up to 0.26% at 25 Mbit/s. This experiment was also able to discover and identify interference related to DMA, such as interruptive flow overhead, around 3% for 25 Mbit/s, or packet transmission memory access overhead, around 9% for 25 Mbit/s.

1 Introduction

Predicting the temporal behavior of tasks running in a multi-core system is hard due to the number of various interference that tasks may suffer [1]. Many approaches have attempted to quantify interference. Several are model-based [2], which involves modeling the system and, for example, simulating its execution. Other methods involve measurement on the system itself [3] by running it under different scenarios to obtain the interference and its impact.

Problem Statement The context of this article is flight software for space missions. Such mission-critical software undergoes a cycle of reviews during which schedulability analysis must be performed to justify the designed real-time architectures. For example, PLATO [4] flight software architecture was justified using an AADL model and simulations with Cheddar [5] as early as the Preliminary Definition Review (PDR).

When schedulability is investigated in such a context, and when a multi-core architecture is used, it may be difficult to model all interference that could occur because of the hardware mechanisms and their interactions.

Contribution In this article, a new approach is introduced to understand the interference that an application may suffer when running on a multi-core architecture. The proposed approach enables the production of a set of configurations, which are combinations of different viewpoints of the target platform and the application. These configurations can then be used to associate metrics (collected at execution time) and interference (produced by one or more components).

The rest of the article is organized as follows. Section 2 presents background. Section 3 introduces the method proposed to quantify interference, following in section 4 by an example and preliminary results obtained from it. Finally, related works and the conclusion complete the article, respectively in sections 5 and 6.

2 Background

This section introduces the notions and terms required to understand the proposed benchmarking method.

2.1 Multi-core Architecture and Interference

In this article, interference is a delay in the task execution time caused by the action of another component in the system.

A multi-core architecture is composed at least of two cores and, generally, of one or further cache units. In addition, each architecture provides a set of hardware mechanisms to ensure robustness, reliability, performance...

With this type of architecture, the interference suffered by a task may come from different sources [1]. From a hardware point of view, DMA transfers, for example, generate hardware interrupts for each packet of sent data. Another example is Compare and Swap (CASA) instructions that lock the memory bus for execution. When two tasks use the same cache unit, the miss rate may increase, and accesses to the main memory may cause additional delays due to such interference. From a software point of view, the activity of other tasks in the system may be another source of interference. The use of a memory bus may be concurrent with the activity of tasks producing memory accesses on other cores. Shared

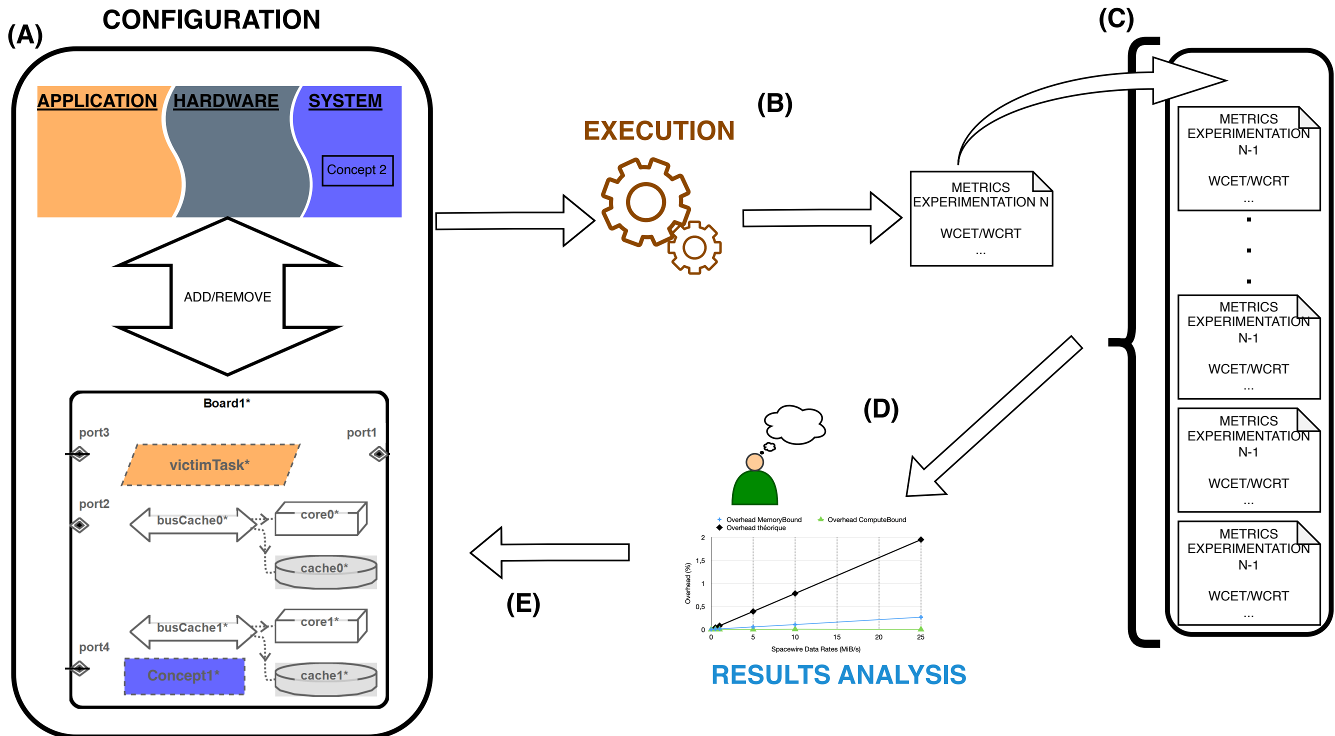

Figure 1: Interference Analysis Process

memory locked by tasks, or any preemption are also sources of interference.

In this work, we consider all sources of interference.

2.2 Benchmark

A benchmark is an application designed to be run on a target platform to evaluate its performance. For example, the Mälardalen benchmark suite [6] aims to evaluate WCET (Worst Case Execution Time). PapaBench [7] provides a set of applications useful for scheduling analysis. From the point of view of interference in a multi-core architecture, Rodinia [8] offers a representative set of multi-core applications. Splash-2 [9], and its extension Splash-4 [10], provide a set of applications characterized in particular by their memory traffic. The latter allows us to show interference related to memory bus.

2.3 GERICOS

GERICOS [11] is a C++ framework developed by LESIA for space applications. This framework offers an integrated solution for rapidly developing multi-core applications using an AMP (Asymmetric MultiProcessing) approach. The framework allows the developer to define application architectures and the assignment of each component to each core. The C++ classes implementing the application are defined independently of the core on which they will run. GERICOS also provides services to measure task WCET and WCRT (Worst Case Response Time). However, in GERICOS, there is no means to quantify the delays for each interference that contributes to WCET.

3 Proposed Approach

In a multi-core architecture, it may be difficult to obtain the accurate delay resulting of a specific interference. The WCET, even though it is supposed to be measured in isolation, may suffer interference from the hardware resources that interact with the system in the background.

Our work aims to quantify all interference through metrics, but in this article we illustrate with an example concerning only hardware resources. In this article, two metrics are considered: the WCET and WCRT. A method is proposed to configure a benchmark, and, with a set of experiments, deduce the delay caused by a hardware resource. First, we explain what we mean by benchmark configuration. Second, we present the proposed analysis process. Finally, we introduce examples of benchmark configurations that will be used in the next section of this article.

3.1 Benchmark Configuration

A benchmark configuration is produced from 3 viewpoints represented by (A) in figure 1. A configuration is the assignment of the various components of the 3 viewpoints on the target platform. In addition, one or more victim tasks are also defined in each benchmark configuration. Victim tasks are tasks from which the metrics are retrieved after benchmark execution to measure delays related to interference.

The first viewpoint is related to the kind of application the benchmark will run. The proposal is to design and run a set of applications that stress the system in a particular way. For example, in this article, as [12], two types of applications are investigated: *MemoryBound* applications that are composed of a task making memory accesses during its execution, and *ComputeBound* which runs a task taking CPU time only, i.e. without any memory access.

The second viewpoint models any hardware entities and mechanisms that may raise a potential interference. For example,

	APPLICATION					HARDWARE					SYSTEM												
Configuration Name	Compute Bound	Memory Bound	SpaceWire Sender	SpaceWire Receiver	Interrupt Bound	Core	Instruction Cache	Data Cache	Interrupt (IRQ)	DMA	Spinlock	Circular Buffer	Single Buffer	InterCore Manager	Interrupt Handler	Shared Object	Single Object	Synchronized Object	Task	Timer	OS	Scheduling Policy	Allocation Resource Policy
Configuration 1.1	1	0	1	1	0	2	*True*	*True*	*True*	*True*	0	4	4	0	2	0	1	0	3	3	RTEMS	HPF	PIP
Configuration 1.2	0	1	1	1	0	2	*True*	*True*	*True*	*True*	0	4	4	0	2	0	1	0	3	3	RTEMS	HPF	PIP
Configuration 2.1	1	0	1	1	0	2	*True*	*True*	*True*	*True*	0	4	4	0	2	0	1	0	3	3	RTEMS	HPF	PIP
Configuration 2.2	0	1	1	1	0	2	*True*	*True*	*True*	*True*	0	4	4	0	2	0	1	0	3	3	RTEMS	HPF	PIP
Configuration 3.1	1	0	1	1	0	3	*True*	*True*	*True*	*True*	0	4	4	0	2	0	1	0	2	2	RTEMS	HPF	PIP
Configuration 3.2	0	1	1	1	0	3	*True*	*True*	*True*	*True*	0	4	4	0	2	0	1	0	2	2	RTEMS	HPF	PIP

Figure 2: The 3 viewpoints for the GR712RC board

most of the current multi-core architectures have different levels of cache or hardware interruption types. Each of these entities may be modeled since they may cause interference.

The last viewpoint models system artifacts that may change the scheduling or the synchronization of the task composing the benchmark. For example, GERICOS provides the concept of `GscSharedResource` that enforces critical section on shared data with spin locks and mutexes. In this viewpoint are also specified the scheduling policy, the operating system and the allocation resource policies.

Figure 2 is a 3-viewpoint model of the benchmark for the GR712RC board used in the section 4. Each line represents a configuration. Each column stores either a quantity or a boolean indicating whether the element is activated or not. For example, configuration 1.1 uses 2 cores and the instruction cache is enabled. In the Application view, *MemoryBound* is a task that constantly performs memory accesses, and *ComputeBound* which only takes CPU time without memory any accesses. `SpwSender` and `SpwReceiver` are tasks that respectively send and receive data packets on the SpaceWire ports of the board. Finally, *InterruptBound* is a task that generates hardware interrupts during its execution.

At hardware viewpoint, we have to specify, the number of cores used during execution, data and instruction cache units, IRQ interrupts and DMA transfers from SpaceWire ports (which can be enabled or not).

The last viewpoint, i.e. the system viewpoint, contains GERICOS concepts, such as the `InterCoreManager`, a task in charge of the communications between cores. Notice that several concepts in this viewpoint, such as spinlock or the operating system, may be not specific to GERICOS.

3.2 Analysis Process

We now describe how benchmark configurations are expected to be used iteratively to understand how interference occurs.

The analysis process is shown in figure 1. The process consists of iteratively running several benchmark configurations.

In the benchmark configuration phase (A), two benchmark configurations are produced at least. The first, named the victim configuration, suffers desired interference. The second configuration, named reference configuration, does not suffer any interference. The reference configuration execution is compared to the victim configuration execution to discover interference.

Benchmark configurations are executed on the platform in (B), and from the victim tasks, a set of metrics is retrieved (i.e. WCET and WCRT).

In (C) and (D), metrics retrieved from configuration execution helps the user to understand whether components contribute or not to interference on the victim tasks. At those steps, two outcomes are possible: either the number of executed configurations is sufficient to understand interference, or interference is not identified and the analysis process is repeated.

To sum up the method, we derive benchmark configurations to progressively eliminate undesired interference until the interference created by a specific hardware resource is isolated, or at least quantified. The interest of this method is to be able to characterize, using a configuration cycle, interference caused by a specific hardware resource.

In this article, we focus on WCET and WCRT to quantify interference. Notice that other metrics could be mandatory to understand the system behavior. For example, *CPU load* or DMA transfers can be the indicator of a background activity. Furthermore, *Cache information* such as the L1 or L2 hit/miss rates may reveal precious information on memory usage and can explain interference origin. Hardware counters would be used also for such a purpose.

3.3 Example

We now illustrate the proposed method with a use case in which we expect to quantify interference due to bus contention during DMA transfers.

The hardware viewpoint in figure 4 models two GR712RC boards. Each board is a Dual-Core LEON3FT SPARC V8 processor with a 32KiB L1 cache for each core and 4 SpaceWire ports. The GR712RC block diagram is represented in figure 3.

We model this system as 6 configurations. Each of them represents different steps to isolate interference and will be used to run the benchmark. To investigate the impact of DMA transfers on bus contention different measurements are done with different transfer rates. All the results are discussed in the next section.

The first configuration, named 1.1, is the reference configuration. In this configuration, a task named `SpwEmitter`, which periodically sends data packets on the SpaceWire network, is assigned and run to *core0*. On the other core, called *core1*, runs the task `SpwReceiver` which is receiving packets emitted from the *core0*. Connection (A) in the figure 4

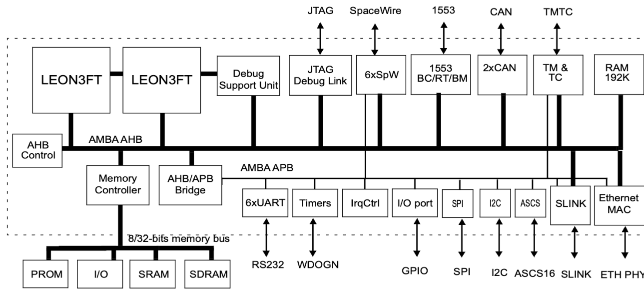

Figure 3: GR712RC Block Diagram

models such communication. The victim task, *ComputeBound*, is placed on *core1*. The victim task, by its nature, cannot be subject to interference from memory accesses and is therefore used as a reference. The second configuration, named 1.2, is the 1.1 configuration when *ComputeBound* is replaced by *MemoryBound*.

The third configuration is the reference configuration, noted 2.1, is similar to the first configuration but `SpwReceiver` is moved on the *core0*. The objective is to isolate the victim task on the *core1*, i.e. not suffer interference from the presence of `SpwReceiver` on the same core. The fourth configuration, named 2.2, is the 2.1 configuration when *ComputeBound* is replaced by *MemoryBound*.

The fifth configuration is the reference configuration, noted 3.1, `SpwEmitter` is assigned on *core0* of *Board2*. `SpwRmapConfigurator` is run on *core1* of *Board1*. Communications between the two boards are modeled by the connection (B) of figure 4. The task `SpwReceiver` is replaced by `SpwRmapConfigurator` in this benchmark configuration. The victim task is assigned on the *core0* of *Board1* to fully isolate it of all perturbations, except the bus contention created by the SpaceWire network during DMA operations. The sixth configuration, named 3.2, is the 3.1 configuration when *ComputeBound* is replaced by *MemoryBound*.

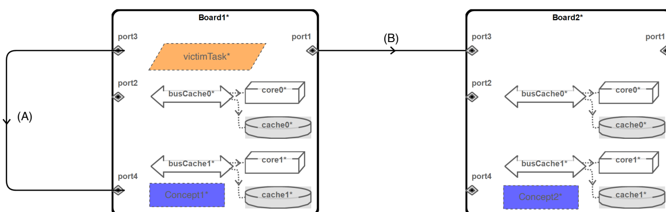

Figure 4: Example of a benchmark configuration

The next section shows the results when running the benchmark configurations above.

4 Evaluation

In this section, we illustrate the proposed analysis process with an experiment. This section presents the goal, the details of the experiment process, and the results obtained.

The aim of this experiment is to characterize the interference of bus contention in a DMA transfer context. To quantify this interference, a communication is generated on a SpaceWire network which creates DMA transfers and then generates contention on the memory bus.

The system is evaluated with various data rates on the SpaceWire network: 0.5 Mbit/s, 1 Mbit/s, 5 Mbit/s, 10 Mbit/s and 25 Mbit/s. In the configurations, in order to ensure packet transmission, the tasks responsible for sending and receiving data have higher priority levels than the victim tasks. The size of a packet is constant. The period of the sending task defines the data rate. The victim tasks are executed 10 times, with an execution time of 1 second and a period of 10 seconds. Statistics related to tasks, such as WCET, are measured through the GscMethodReport class of GERICOS. GscMethodReport provides a set of functions that will be called by the task itself to measure and record data.

We apply the process proposed in section 3 by configuring and running the benchmark with 6 configurations. The results, shown below, compare the overhead due to interference on the WCET of the victim tasks. Each graph shows the results of two comparable configurations for a different type of victim task.

First, we configure the benchmark according to the configurations 1.1 and 1.2 described in section 3, run it, and get results of figure 5. In this figure, the overhead of *ComputeBound* and *MemoryBound* are closer, respectively of 11,41% and 11,33% for 25 Mbit/s, which means suffered interference is the same. However, *ComputeBound* does not use the memory bus, so interference should not be found for this task with these configurations. In fact, the first configuration creates interference than even a task that does not access memory suffer. It is caused by `SpwReceiver`, which is on the same core, and has a higher priority than the victim task. Such results lead us to investigate the system performance with the configurations 2.1 and 2.2.

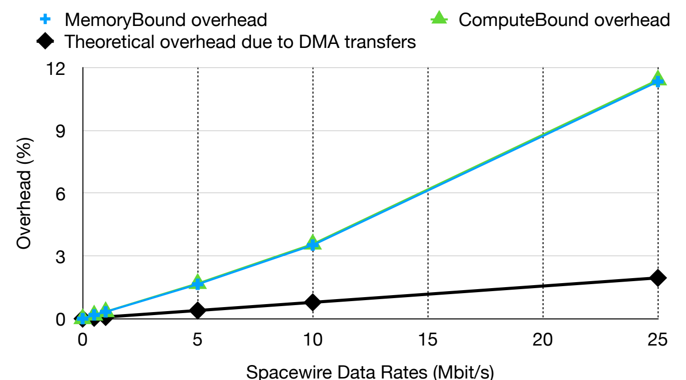

Figure 5: Throughput for configurations 1.1 and 1.2

Figure 6 shows the results for the configuration 2.1 and 2.2, described in section 3. In this figure, we show the *ComputeBound* overhead curve rises more slowly than *MemoryBound* overhead curve, reaching a gap of 0.86% at 25 Mbit/s. This indicates a difference in suffered interference which cannot

be related to the contention on the memory bus, as 89.69% of the interference perceived by *MemoryBound* is also perceived by *ComputeBound*. This may be due to the number of interruptions caused by DMA transfers, which are performed on the same board as the task being analyzed. To investigate this assumption, we run the next configurations.

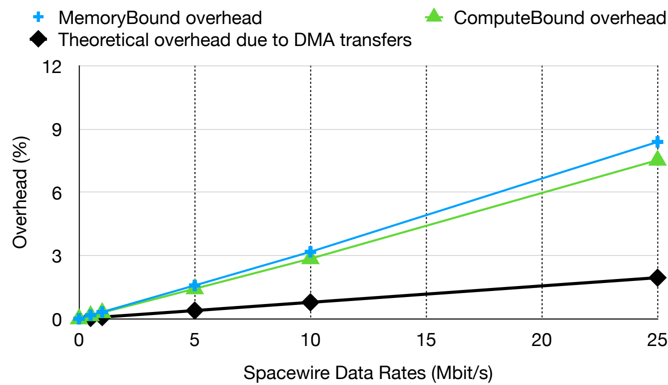

Figure 6: Throughput for configurations 2.1 and 2.2

To complete the analysis, we run the benchmark with the configurations 3.1 and 3.2, and get the results of the figure 7. Now, *ComputeBound* suffers no interference with an overhead of 0.1 ms between 10 Mbit/s and 25 Mbit/s; whereas *MemoryBound* suffers interference of up to 0.26% overhead at 25 Mbit/s. Assuming that the performance gap between *ComputeBound* and *MemoryBound* is due to memory accesses, it means that we succeeded to isolate interference created by DMA accesses on the memory bus with this last experiment.

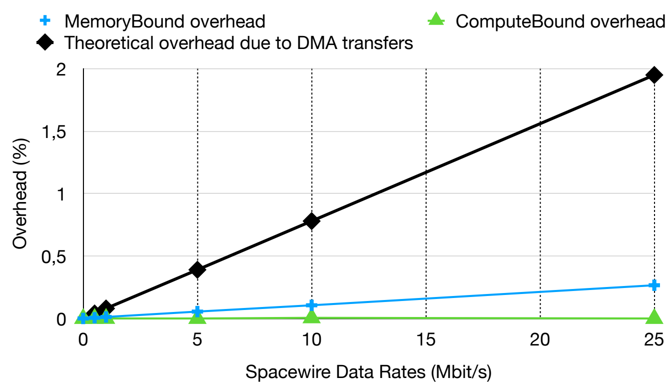

Figure 7: Throughput for configurations 3.1 and 3.2

In the 3 figures presenting the results, a curve called theoretical overhead is displayed. In the case of the PDR of PLATO project, interference of DMA transfers was estimated by a theoretical computation. Our experiments showed that this theoretical computation predicted an interference overhead of 7.33 times higher than those measured for 25 Mbit/s with configuration 3.2. This is explained because theoretical computation was carried out with the assumption that all DMA accesses are concurrent and have the highest priority, which is a pessimistic case that does not occur with configurations 3.1 and 3.2. In figure 5 and 6, the measurements show higher values than the theoretical curve because the measures are composed of different interference origins and not only from DMA memory bus contention, this motivated configurations 3.1 and 3.2 to better isolate DMA interference.

Furthermore, DMA transfers using SpaceWire also cause interrupts on reception, and memory accesses on transmission. Figures 5, 6 and 7 show the overhead caused by these two mechanisms. Switching the `SpwReceiver` to *core1* in configurations 2.1 and 2.2, shows that interrupts related to reception no longer interfere with the victim task. This leads to a loss of about 3% overhead for 25 Mbit/s between the curves in figure 5 and 6. Similarly, for configurations 3.1 and 3.2, the packet emission is no longer on the same board as the victim task, so it no longer competes for access to the memory bus. This reduces overhead by about 9% for 25 Mbit/s between the curves on the figure 6 and 7.

In conclusion, by applying the proposed method, the set of configurations that was produced enabled us to characterize the interference of bus contention in a DMA context. In addition, within the PLATO context, the measurements demonstrated the pessimism of the theoretical evaluation. Finally, all the curves enable us to make assumptions about the overhead caused by other mechanisms resulting from DMA transfers.

5 Related Works

A lot of works deals with interference management in multi-core architectures. Interference can be either predicted, measured or mitigated by proposing software and/or hardware designs that reduce them [13].

In [12], the author introduces a method to quantify interference by measurements. He focuses on several hardware components such as shared L1 and L2 caches by experimentations with memory bound and compute bound benchmarks. Applied benchmarks are similar to the one we have used in our configuration process.

In contrary, the multi-phases task model [14] avoids interference with the PREM model [15]. Tasks are designed as set of different phases that limit memory bus interference. The main drawback of this approach is to not support legacy programs. Supporting legacy programs is mandatory for systems targeted by the configuration process we propose.

Several benchmarks provide means to understand interference. Rodinia [8] and Splash-2 [9] are examples of them. Interference analysis with these benchmarks focuses on memory interference and does not consider many others such as interruption, DMA or operating system interference. Our approach expects to quantify any type of interference.

6 Conclusion

In this article, a method to isolate and evaluate interference in multi-core architectures is proposed. The approach consists of running several configurations of a benchmark and measuring task metrics to understand how interference contributes to the metrics. An experiment has shown that the proposed method can characterize, from a set of configurations, interference of a shared hardware resource through the WCET. For the PLATO mission, the theoretical computations for such interference, for 25 Mbit/s, were 3.25 times higher than our measurements which estimated it to 0.26% of the WCET. This demonstrates

the pessimism of the theoretical results and the accuracy of the benchmarking method we proposed. Finally, the experiment was also able to identify interference related to DMA, such as interruptive flow overhead, around 3% or packet transmission memory access overhead, around 9%.

In the future, we expect to evaluate the method on more complex architectures such as GR740RC, a naked quad-core with different cache levels and interference reduction mechanisms. Furthermore, this article has presented a method that gives users the ability to design benchmark configurations according to their understanding of the system behavior, which can be difficult to achieve. In the next steps, we expect to help users by defining configuration design patterns for specific hardware resource patterns. Finally, the hardware architecture of the GR712RC does not provide statistics on system interactions. More and more architectures are introducing performance counters [16] [17] [18] [19], notably the GR740RC. Performance counters may contribute to quantify/identify interference. We plan to integrate these counters in our benchmark configuration process.

References

[1] T. Lugo, S. Lozano, J. Fernández, and J. Carretero, "A survey of techniques for reducing interference in real-time applications on multicore platforms," *IEEE Access*, vol. 10, pp. 21853–21882, 2022.

[2] V. A. Nguyen, E. Jenn, W. Serwe, F. Lang, and R. Mateescu, "Using model checking to identify timing interferences on multicore processors," in *ERTS 2020-10th European Congress on Embedded Real Time Software and Systems*, pp. 1–10, 2020.

[3] J. Stärner and L. Asplund, "Measuring the cache interference cost in preemptive real-time systems," in *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pp. 146–154, 2004.

[4] P. Plasson, G. Brusq, F. Singhoff, H. N. Tran, S. Rubini, and P. Dissaux, "Plato n-dpu on-board software: an ideal candidate for multicore scheduling analysis," in *10th European Congress ERTSS Embedded Real Time Software and System*, 2022.

[5] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a flexible real time scheduling framework," in *Proceedings of the 2004 annual ACM SIGAda international conference on Ada: The engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies*, pp. 1–8, 2004.

[6] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks: Past, present and future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[7] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel, "Papabench: a free real-time benchmark," in *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*, pp. 44–54, Ieee, 2009.

[9] J. M. Arnold, D. A. Buell, and E. G. Davis, "Splash 2," in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pp. 316–322, 1992.

[10] E. J. Gómez-Hernández, J. M. Cebrian, S. Kaxiras, and A. Ros, "Splash-4: A modern benchmark suite with lock-free constructs," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 51–64, IEEE, 2022.

[11] P. Plasson, C. Cuomo, G. Gabriel, N. Gauthier, L. Gueguen, and L. Malac-Allain, "Gericos: A generic framework for the development of on-board software," *DASIA 2016-Data Systems In Aerospace*, vol. 736, p. 39, 2016.

[12] T. Beck, *Evaluation and analysis of Linux applications on multi-core processors in a space environment*. PhD thesis, Toulouse, ISAE, 2023.

[13] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–38, 2019.

[14] C. Maia, L. Nogueira, L. M. Pinho, and D. G. Pérez, "A closer look into the aer model," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, IEEE, 2016.

[15] 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 *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 269–279, IEEE, 2011.

[16] G. Cabo, S. Alcaide, C. Hernández, P. Benedicte, F. Bas, F. Mazzocchetti, and J. Abella, "Safesu-2: a safe statistics unit for space mpsocs," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1085–1086, IEEE, 2022.

[17] T. Scheipel, F. Mauroner, and M. Baunach, "System-aware performance monitoring unit for risc-v architectures," in *2017 Euromicro Conference on Digital System Design (DSD)*, pp. 86–93, IEEE, 2017.

[18] F. Cosimi, F. Tronci, S. Saponara, and P. Gai, "Analysis, hardware specification and design of a programmable performance monitoring unit (ppmu) for risc-v ecus," in *2022 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 213–218, IEEE, 2022.

[19] M. Lei, T.-Y. Yin, Y.-C. Zhou, and J. Han, "Highly reconfigurable performance monitoring unit on risc-v," in *2020 IEEE 15th International Conference on Solid-State & Integrated Circuit Technology (ICSICT)*, pp. 1–3, IEEE, 2020.