SAFURE

D4.2
Analysis of run-time and software applications on multi-core

Project number:	644080
Project acronym:	**SAFURE**
Project title:	SAFety and secURity by dEsign for interconnected mixed-critical cyber-physical systems
Project Start Date:	1st February, 2015
Duration:	40 months
Programme:	H2020-ICT-2014-1
Deliverable Type:	Report
Reference Number:	ICT-644080-D4.2 / 1.0
Work Package:	WP 4
Due Date:	November 2017 - M34
Actual Submission Date:	28 November 2017
Responsible Organisation:	BSC
Editor:	Jaume Abella
Dissemination Level:	Public
Revision:	1.0
Abstract:	This document describes the methodologies integrated to characterize applications on different hardware setups relevant for the use cases. Results on benchmarks and an avionics prototype are provided along with the methodologies.
Keywords:	Multi-core, Hardware support, Timing integrity, Performance analysis, Performance monitoring

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644080.

This work is supported (also) by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0025. The opinions expressed and arguments employed herein do not necessarily reflect the official views of the Swiss Government.

Editor

Jaume Abella (BSC)

Contributors (ordered according to beneficiary numbers)

Sylvain Girbal (TRT)
Gabriel Fernandez, Jaume Abella, Francisco J. Cazorla (BSC)

Disclaimer

Executive Summary

The integration of time-critical tasks in multicore processors needs to take into account the impact of interferences due to accessing shared hardware resources. Those interferences impact the execution time of tasks in non-obvious ways and means are required to account for them in the Worst-Case Execution Time (WCET) estimates of tasks. This deliverable describes the methodologies integrated towards quantifying such impact in some specific hardware platforms relevant for the use cases as well as a toolset (METrICS) to analyze performance and understand the sources of interference. The effectiveness of those methodologies is assessed quantitatively with appropriate benchmarks and avionics prototypes, leaving their evaluation in use cases for WP6.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Increasing time-predictability needs of the consumer electronics market and increasing (guaranteed) performance needs of the critical real-time market push toward their convergence [16, 19]. In particular, consumer electronics market, which includes mobile phones and tablets, is expected to embed a growing number of critical functionalities related to monitoring and communication with other devices in the health, car, smart cities, Internet-of-Things domains. The critical real-time market, which includes avionics, automotive, robotic and healthcare applications, has also been shown to demand higher guaranteed performance to meet the needs of an increasing number of complex functions (e.g. autonomous vehicles) [3].

High-performance processors in the consumer electronics market are well known to pose difficulties to derive execution time bounds needed for critical real-time applications, while processors in critical real-time embedded systems offer (typically low) guaranteed performance. Therefore, while time-predictable high-performance processors have the potential to satisfy the needs of both markets, reconciling high-performance and predictability is a major challenge. Some issues have already been identified [8, 16], however, such convergence has not been explicitly studied for many popular processor architectures in the consumer electronics market. As an example, the ARM big.LITTLE architecture used in many tablets and mobile phones, as well as in driving safety support systems in commercial automotive solutions (e.g. Renesas R-Car H3 [37]), has not been fully analysed against the requirements of critical real-time applications. Analogously, processors intended for critical applications, such as the Infineon AURIX architecture, pose similar challenges (at a lower scale), which also need to be studied.

In this deliverable we address these challenges with two complementary directions:

- Providing means to upper-bound contention in shared resources of commercial off-the-shelf (COTS) processors (see Chapter 2).
- Delivering a toolset that allows assessing performance interference in COTS processors so that bottlenecks can be understood (see Chapter 3).

In particular, the former contribution, allows tightly upper-bounding potential contention in COTS processors, either allowing *any* task to be consolidated together with the task under analysis, or once a specific task consolidation has been performed and contender tasks are known, thus allowing deriving tighter contention upper-bounds. The latter contribution allows assessing whether specific consolidations of critical and non-critical tasks lead to high or low performance interference, and what the source of that interference is, so that the tasks can be integrated in a way that interference is minimized.

Chapter 2 Assessment and Modelling of the Relevant Hardware Platforms

The multicore contention models developed in WP3 are intended to be integrated in one of the use cases. In particular, the main target was the Telecom use case. Different hardware platforms have been considered for the integration of the use cases. Initially, the Qualcomm SnapDragon 810 was regarded as the best fit for the Telecom use case based on the (limited) information available about the different target platforms. It implements an ARM big.LITTLE architecture. Details on the analysis of its hardware support to manage timing interferences and how to interface it from the software layers is provided in Section 2.1.

Results with the SnapDragon board have shown that it cannot be easily mastered with the vendor support available. Therefore, a development board also implementing an ARM big.LITTLE architecture, the ARM Juno board, has been assessed instead. While it cannot be directly used in the Telecom use case[1], it allows providing a first assessment of whether ARM big.LITTLE architectures provide the timing characteristics needed. Of course, conclusions on the Juno board illustrate those aspects that need improvement in commercial platforms, such as the SnapDragon 810, for their appropriate use in real-time applications. In any case, while conclusions are positive, as shown in Section 2.2, this platform cannot be directly assessed against the Telecom use case. Instead, results are obtained for an avionics prototype that could be successfully integrated onto the Juno board.

Finally, we have further assessed the timing interferences occurring in the Infineon AURIX TC27x processor, which is the preferred platform for the Automotive multicore use case. While this effort was not initially planned, additional resources have been devoted to integrate multicore contention management technology in one of the use cases. Integration efforts and results are presented in Section 2.3. The evaluation on the Automotive multicore use case is done as part of WP6.

2.1 SnapDragon

2.1.1 Goal and Scenario

In this section we determine whether the SnapDragon 810 processor can be used in the context of critical real-time applications. We use the term critical real-time to refer to any hardware or software component with any time criticality need: either mission, business of safety related.

2.1.1.1 The Platform

The SnapDragon 810 processor is a Qualcomm implementation of the ARM big.LITTLE architecture used in several recent Sony Xperia mobile phones. In the SnapDragon 810, several hardware events can be tracked with PMCs. Therefore, conclusions obtained on this specific processor apply to several others in the consumer electronics market, especially those building upon ARM big.LITTLE architecture and those implemented by Qualcomm.

The architecture of the processor, shown in Figure 2.1, comprises 2 clusters (also referred to as processors according to ARM's nomenclature): an ARM Cortex-A57 cluster with 4 cores and an ARM

[1] The Juno board is a development board that has been regarded neither compatible nor appropriate for commercialization for the Telecom use case. This board has neither appropriate interfaces nor the form factor for embedded products.

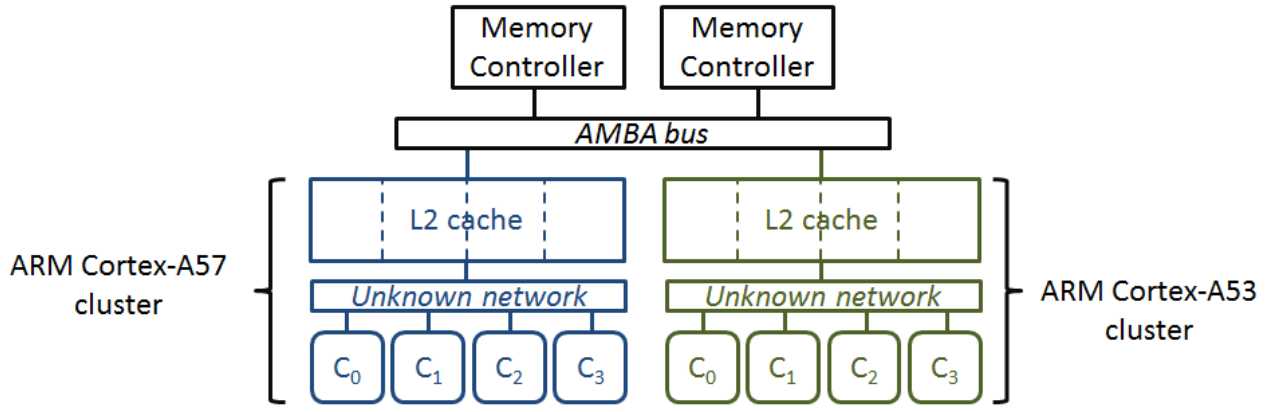


Figure 2.1: Schematic view of the elements of the SnapDragon 810 processor analysed in this work

Cortex-A53 cluster with 4 cores. A57 cores are high-performance cores with out-of-order execution, whereas A53 cores are low-power lower-performance in-order cores. Yet, the A53 cluster is already a relatively high-performance platform w.r.t. current microcontrollers in many critical real-time systems. Therefore, as a first step we analyse this cluster, whose cores are more amenable to current timing analysis technology, with the aim of extending our work to the A57 cluster at a later stage. Each A53 core is equipped with local first level instruction (IL1) and data (DL1) caches. Level 1 caches are connected to a shared L2 cache, local to the cluster. An Advanced Microcontroller Bus Architecture (AMBA) bus interface connects both clusters to two shared memory controllers to access DRAM. Peripherals and accelerators (present in this platform but not shown in the figure) are also connected to the AMBA bus. In this work we decrease their effect by keeping them either disabled or idle.

Both clusters (A57 and A53) and the AMBA bus have been developed by ARM, the Intellectual Property (IP) provider. Qualcomm, the chip manufacturer, integrates those components along with some others, which may or may not be provided by ARM. Moreover, in the integration process Qualcomm may introduce modifications in some IP components and/or their interfaces.

2.1.1.2 Tracing and Events

We focus on measurement-based timing analysis (MBTA), widely used in most real-time domains. For instance MBTA is used in avionics systems [33, 5], including those with DAL-A safety requirements [26] (though on top of much simpler single-core processors).

In the context of MBTA, in D3.2 we have shown that it is fundamental tracing those events that impact shared resource contention – e.g. cache misses – to derive bounds for a task factoring in a specific contention level rather than the worst contention possible [11]. It follows that MBTA techniques demand more and more advanced hardware tracing mechanisms.

For that purpose we build upon the existence of PMCs to derive the type and number of accesses each task does, since this is needed to account for the contention a task can experience from (or produce on) others (see D3.2, section 4.6). We also build upon stressing benchmarks, i.e. small user level applications, that are able to create very high contention for each access type to the target shared resource [12].

2.1.2 Qualitative Analysis: Specifications

The main source of information for the analysis of the SnapDragon 810 processor is the ARM Cortex-A53 processor technical reference manual [2]. As detailed in the manual, a number of A53 features are regarded as ‘implementation dependent’, thus meaning that the processor manufacturer has the flexibility to choose among different options available. For instance, this is the case of the DL1, IL1 and L2 cache sizes. From the information available in the A53 manual, we regard the following

information particularly relevant for contention analysis:

- The arrangement of the main components in the A53 cluster, including DL1, IL1 and L2 caches, as well as data prefetching features in DL1 and coherence support in L2.
- PMCs for events occurring in the cores (e.g. in DL1 and IL1 caches) and in the L2 cache.

However, some parameters are not available in the A53 manual, including the following: (1) Timing characteristics of the interconnect between DL1/IL1 and L2 caches; (2) Specific characteristics of the different cache memories such as, for instance, their sizes; and (3) PMCs for events spanning beyond the A53 cluster such as accesses to the bus connecting A53 and A57 clusters with memory, and PMCs for the memory controllers. From a detailed analysis of each of those missing parameters, we reached the following conclusions:

- The interconnect between DL1/IL1 and L2 caches, as the remaining in-cluster components, should be documented in ARM manuals. The lack of this information in the manuals makes us resort to software testing (e.g. stressing benchmarks) to bring some light on the characteristics of this interconnect.
- Some instructions exist to read the particular characteristics of the cache hierarchy so they can be directly retrieved from the platform itself.
- PMCs and events beyond the A53 cluster should be documented in the SnapDragon 810 manual, since the processor manufacturer integrates those components, and so has access to the appropriate information for each component.

By the time we performed this work, ARM manuals were available, so we could retrieve them[2]. However, SnapDragon 810 manuals are neither publicly available in Qualcomm's website, nor included in the documentation coming along with the Intrynsic DragonBoard (development board based on the SnapDragon 810 processor), nor obtained upon request. In particular, while we requested appropriate manuals through Qualcomm public services as well as through internal contacts, and NDAs are in place, we were unable to get access to them. Other partners in the consortium have experienced similar issues. Therefore, to the best of our knowledge, no information has been obtained on what PMCs/events exist beyond the A53 cluster and how they could be used. From our analysis of the available information, we have reached the following conclusions:

- The interconnect between DL1/IL1 and L2 can only be analysed empirically without specific guidance on its timing behaviour. The confidence on those measurements is limited due to the unknown specification of the interconnect.
- DL1, IL1 and L2 features can be directly obtained from the board via control instructions.
- Specific instructions exist to disable the data prefetcher. This is particularly relevant to discount uncontrolled (prefetcher) effects during operation. However, as explained later, the prefetcher could not be disabled.
- The L2 is inclusive with DL1 for coherence purposes. Thus, one core can create interferences on the DL1 of other cores by evicting their data from the L2 cache.
- The L2 cache cannot be partitioned across cores. This feature, together with L2 cache inclusivity, leads to potentially high inter-core interferences if not controlled by software means.
- PMCs up to the L2 cache exist and are abundant. However, no information is available about PMCs beyond the L2 cache.

[2]They have later become unavailable online and can only be retrieved upon request to ARM.

Overall, several cache features challenge the calculation of inter-core interference execution time bounds, and the lack of documentation for the DL1/IL1-L2 interconnect and the PMCs/events beyond L2 decrease the confidence that can be obtained on measurement-based bounds. However, some information about contention can still be retrieved empirically based on information available. In the next section we present the experiments performed for that purpose, and the results obtained.

2.1.3 Quantitative Analysis: Experimentation

The number of hardware events that can be monitored in the SnapDragon 810 processor is limited according to ARM's documentation. For instance, while cache and memory accesses can be counted, it cannot derived whether DL1/IL1 and L2 cache accesses turn out to be hits or misses. This complicates the development of our methodology to measure the impact of contention in the access to shared resources.

2.1.3.1 Stressing Benchmarks

In order to access PMCs we have developed a library with the an interface to read/write PMCs. The main functions of the library include resetting/setting PMCs, activating/stopping PMCs, read/write PMCs and start/stop the Performance Monitoring Unit. To quantify the impact of contention in the access to the different shared resources, we have developed several stressing benchmarks that stress each specific resource separately, in line with the method in [12]. This allows estimating the maximum delay that a request to a particular shared resource can suffer. Then this data is used to upper-bound contention impact. As starting point, we have developed stressing benchmarks to account for contention in the access to the shared L2 cache and to the shared memory controller. The basic structure of a stressing benchmark is given in Algorithm 1.

Algorithm 1 Structure of a stressing benchmark

```
1: procedure SB_BODY
2:     for (i = 0; i ≤ N; i++;) do
3:         reset PMCs;
4:         for (j = 0; j < M; j++) do
5:             R2 = Load [@A+R1]; R1 = R1+STRIDE;
6:             R2 = Load [@A+R1]; R1 = R1+STRIDE;
7:             ...
8:             R2 = Load [@A+R1]; R1 = R1+STRIDE;
9:         end for
10:        read PMCs;
11:    end for
12: end procedure
```

Since measurements can be polluted, e.g. by the OS running below, we collect several (N) measurements and remove outliers keeping only the statistical mode. The iterator M and the number of LOAD operations in the loop are set to values sufficiently high so that the overhead of the loop (i.e. the control instruction) and the overhead to fill the IL1 cache become negligible (e.g. $M = 1000$ and 16 LOAD operations). In each outer loop iteration, the particular PMCs/events read and reset depend on the contention that is to be measured in a particular experiment. Finally, STRIDE relates to the distance between memory objects accessed so as to make sure that they either hit in L1, miss in L1 and hit in L2, or miss in L1 and L2. Vector size is properly set also with the same goal.

2.1.3.2 Disabling the Data Prefetcher

We disabled the data prefetcher so that read and write operations occurring in the different cache memories are only triggered explicitly by the instructions executed in the stressing benchmarks, rather than being automatically generated by hardware. For that purpose, we have configured the `CPUACTLR`

register as described in the A53 manual [2]. Unfortunately, the execution of these commands led to a system crash.

In order to verify the source of the problem, we repeated the same experiment on a PINE A64 [35] board. The PINE A64 platform is built with the aim of being a low-cost open source platform. It implements the same Quad-Core A53 Processor as the low-power SnapDragon 810 cluster. Thus, its interface is expected to be the same. In the PINE A64 platform, the commands to disable the prefetcher worked properly and subsequent experiments revealed that the data prefetcher was effectively disabled on that board. However, such board is a low-cost and low-power general-purpose computer, so the board itself is not oriented to the industry in the mobile market. Instead, it is an open platform. Thus, mobile industry will unlikely use it since there is no a large enterprise that provides support in the long term.

Overall, we could not disable the prefetcher in the SnapDragon 810. This problem likely relates to potential modifications introduced by the processor manufacturer, from which we did not succeed in obtaining the information required about the SnapDragon 810.

As a confirmatory experiment, we run a stressing benchmark accessing 88KB of data, thus exceeding DL1 capacity (64KB) but fitting L2, with a 8B stride. Hence, every 8 accesses we have 1 DL1 miss and 7 DL1 hits due to spatial locality (DL1 line size is 64B). With the prefetcher disabled, we would expect that the number of L2 accesses was 1/8 those in DL1. We observed that the number of DL1 accesses matches quite well our expectations, but the number of L2 accesses is roughly 0, revealing that the prefetcher is active and fetches data into DL1 reducing L2 accesses (the PMC for prefetch requests confirms this hypothesis).

2.1.3.3 Assessing Stressing Benchmark Results

In order to assess the behaviour of the PMCs in the A53 cluster, we have run our stressing benchmark, which performs 11,000 load operations with a specific stride. This code is in a loop iterating 100 times, and we report average results across those 100 iterations to minimise the impact of cold misses in the first iteration and noise in the measurements.

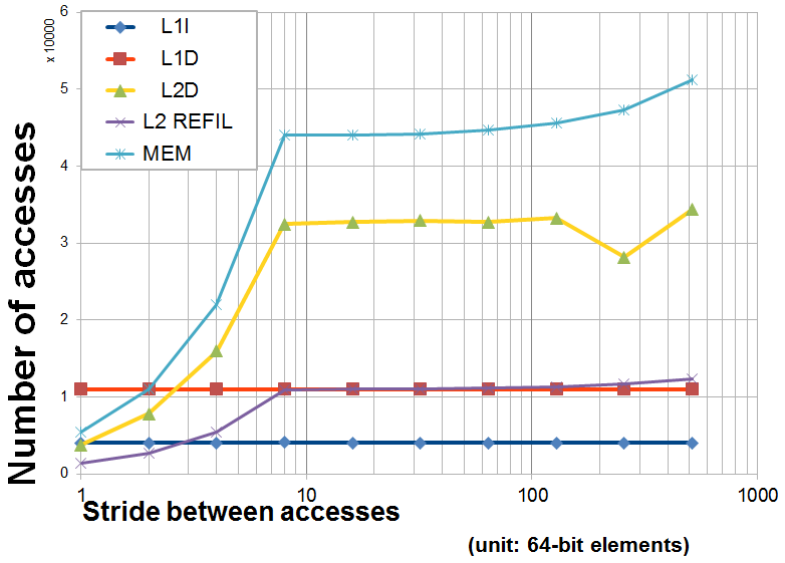


Figure 2.2: Avg. number of IL1 (L1I), DL1 (L1D), L2 (L2D) and memory (MEM) accesses, and L2 refills per loop iteration for different data strides.

We explore strides ranging between one 64-bit element (8 bytes) and 512 elements. With the smallest stride (8 bytes), we traverse a vector of $\approx 88KB$ (11,000 elements x 8 bytes), which does not fit in DL1, but fits in L2. Thus, the number of DL1 accesses expected is 11,000 approximately.

Each load is expected to miss in DL1 when 64-byte boundaries (DL1 cache line size) are crossed, and should hit in DL1 otherwise.

Overall, for a 1-element stride we expect 1,375 (11,000/8) L2 accesses per outer loop iteration. Then, since $\approx 88KB$ fit in L2, we expect roughly 0 memory accesses (13.75 on average in practice). When doubling the stride, we expect L2 accesses to double until reaching value 11,000 (at stride 8), and then flatten. Memory accesses should remain roughly 0 until L2 cache capacity (512KB) is exceeded, at stride 8 ($\approx 704KB$), when all accesses become also L2 misses so we have 11,000 DL1, L2 and memory accesses.

Figure 2.2 shows how DL1 accesses effectively match expectations while L2 data accesses (L2D in the plot) show much higher values. Interestingly, L2 refills (`L2 REFIL`), i.e. lines brought explicitly on a DL1 miss, match our expectation for L2 accesses. This reveals that, apart from the DL1 misses, we have another source of L2 accesses, which seems to be the prefetcher. When looking at the number of memory accesses (MEM), we observe that it matches quite accurately L2 accesses plus L2 refills, thus reflecting a number of accesses around 4x the expectations (44,000 memory accesses instead of 11,000). This also reveals interferences from the prefetcher since, even when data should fit in L2 (up to stride 8) and so memory accesses should be negligible, we have plenty of them. Overall, this experiment reveals that the prefetcher is active and produces severe interferences that defeat any intent to control contention in shared resources in the A53 cluster.

2.1.4 Summary of Lessons Learned

In this section we analysed the difficulties that entails using a popular microprocessor in consumer electronics, the SnapDragon 810, in the context of critical real-time applications. This microprocessor provides the level of performance needed by many critical real-time applications, but at the same time poses a number of challenges in its utilisation, which we summarise next.

Uncontrolled resource sharing. The use of a fully-shared L2 cache across several cores brings some difficulties to control or tightly upper-bound inter-core interferences. In particular, one task running in one core is allowed to evict any line in the L2 cache, thus affecting the performance of other cores in non-obvious ways if those other cores rely on the contents stored in L2. This issue may be potentially exacerbated by the fact that the L2 cache of this processor is inclusive with DL1 caches. Thus, a task may also get its data evicted from DL1 due to the inclusion property with L2.

The most promising approach to overcome this challenge builds upon cache partitioning. For instance, the Freescale P4080 processor, also representative of a high-performance processor of interest for real-time applications, allows configuring its shared L3 cache so that private regions are allocated to specific cores [6]. However, space partitioning may not be enough if buffers and queues are shared, which may still allow high contention across cores, thus leading to low performance guarantees [40]. However, as shown in this section, a popular processor such as the SnapDragon 810 does not provide such a support yet.

Need for documentation. For enabling MBTA based on PMCs, at least some documentation about components interfaces is mandatory. The information on hardware-to-hardware interfaces includes the way in which requests are managed (e.g. whether shared queues are used, what policies are used to serve requests). This allows reasoning about the theoretical worst-case scenarios so that stressing benchmarks can be developed to stress them and obtain timing information via measurements.

Regarding software-to-hardware interfaces, which include precise information on how to enable/disable some features (e.g. prefetchers) or how to monitor hardware events through PMCs available, information released is often limited. Again, this prevents appropriate configuration and monitoring of the processor, thus defeating the intent of obtaining tight WCET estimates on top of these platforms. The unavailability of this information often relates to IP protection and competition.

Both issues are exacerbated by the fact that many microprocessors incorporate IP from different suppliers, as in the case of the SnapDragon 810 processor, which includes at least IP from ARM and Qualcomm. In our view, detailed information will be made progressively available as market pressure

increases and releasing details becomes the only way to make sales grow. Still, this shift towards openness will occur slowly, so we have to resort to study a development platform, the ARM Juno board, to assess ARM big.LITTLE architecture at this stage.

2.2 Juno Board

In this section we follow a similar process to that for SnapDragon 810, but for the Juno SoC, which has only been developed by ARM, thus avoiding unmatching features with those documented, and also allowing a broader access to technical documentation.

2.2.1 The Platform

Next we provide the most relevant details of the ARM Juno R2 board for this work. The Juno board includes the Juno SoC, whose general purpose components are depicted in Figure 2.3. This ARM big.LITTLE design includes two computing clusters, being one of them equipped with 2 Cortex-A72 high-performance cores and another with 4 Cortex-A53 low-power cores. We refer to those clusters as *HPclus* and *LPclus* for short. Each cluster includes a local shared L2 cache. Both clusters are connected to a shared memory controller.

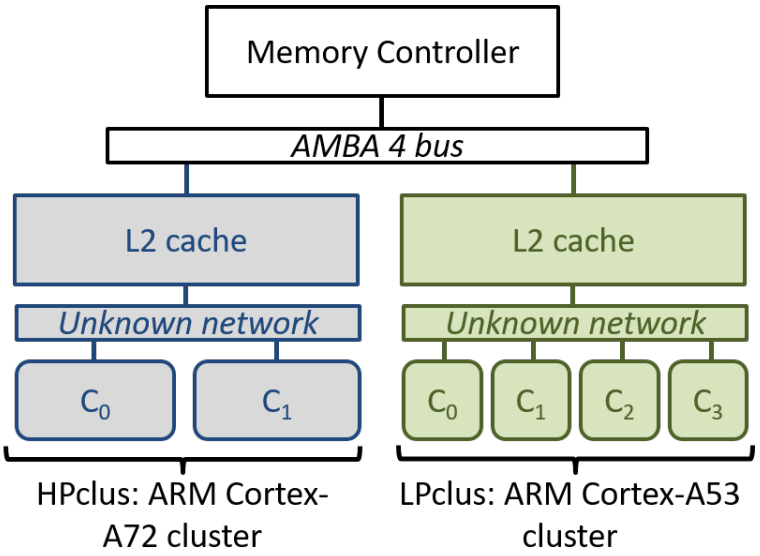


Figure 2.3: Schematic view of the Juno SoC components analyzed in this work.

A72 cores feature out-of-order execution, whereas A53 ones offer low-power in-order execution instead. For the sake of facilitating the interpretation of results, the assessment in this work will focus on A53 cores (LPclus), to discount the measurement noise that out-of-order execution could introduce. Yet, conclusions reached in this work, are not necessarily restricted to this cluster and, to some extent, apply to both clusters.

Each core includes a first level instruction (IL1) and data (DL1) cache. Also, each cluster includes a shared L2 cache. In order to assess the impact of shared resources on execution time we stress caches with increasing data sizes. The size IL1 caches do not impact results since benchmarks tested are designed to be tiny ($<$ 1KB) in comparison to the IL1 size (32KB for LPclus and 48KB for HPclus). Data cache sizes for the LPclus are 32KB 4-way 64B/line for DL1 and 1MB 16-way 64B/line for the L2.

The particular interconnect between cores and L2 caches is not described in the documentation. L2 caches are connected to the memory controller through an ARM AMBA 4 bus (ARM CoreLink CCI-400 Cache Coherent Interconnect). Other components, such as accelerators and peripherals

(not depicted in Figure 2.3), are also attached to this bus. Since they are not used in our analysis, we omit details and keep them disabled or idle during test campaings.

2.2.1.1 Measurement Collection Infrastructure

Measurement-Based Timing Analysis (MBTA) is the most common practice in industry for timing analysis [42]. Hence, our quantitative assessment builds upon empirical evidence obtained directly on the Juno SoC. For our analysis, we build upon the measurements collected with the Performance Monitoring Unit (PMU), which allows us to monitor the following events: execution time, instructions retired, DL1 accesses and IL1 accesses (per core), L2 accesses (per cluster), and memory accesses (global). Although some other events may also be monitored, we focus on the ones listed as they prove being enough to reach clear conclusions.

Note, however, that the number of Performance Monitoring Counters (PMCs) in the PMU is limited, so we cannot monitor all events in a single run. Instead, each experiment needed to be repeated twice with different event-to-PMC mappings to obtain all measurements. The particular experimental methology used is described later in Section 2.2.3.1.

2.2.2 Qualitative Assessment

We have conducted first a qualitative analysis on the adequacy of ARM big.LITTLE architectures for Critical Real-Time Embedded Systems (CRTES). The main sources of information for such analysis are the processor specifications such as the ARM Cortex-A53 processor technical reference manual [2]. Although several parameters are regarded as *implementation dependent* in the documentation, conducting our work on an ARM development board – the Juno board – grants us access to further details and means to poll the platform for implementation details. For instance, cache parameters can be obtained by executing specific instructions, which provide the user with the particular sizes, number of ways and line sizes of the different cache memories in the SoC.

Next we review the main shared resources in the SoC that may become sources of contention, i.e. timing interference, and to what extent they are expected to impact the execution time of critical real-time tasks due to contention.

2.2.2.1 L1-L2 Interconnect

The first shared resource that tasks use when exiting the core is the interconnect between L1 and L2 caches. Unfortunately, details on its design are not publicly documented, so its impact can only be assessed quantitatively (see Section 2.2.3). However, our expectation is that, given that the L2 may be accessed often from the different cores, the L1-L2 interconnect provides high bandwidth, thus, avoiding the complete serialization of accesses.

2.2.2.2 Shared L2 cache

It is unclear whether requests are processed in-order or in a round-robin fashion across contenders since this feature is not documented in the specification. Hence, the impact of this feature can only be sensibly assessed quantitatively, as done later in Section 2.2.3. However, the main issue related to the L2 cache for critical real-time tasks is the fact that it cannot be partitioned. Hence, unless software partitioning is implemented (e.g. controlling the memory regions allowed for the different tasks in each core [28, 31]), L2 cache space is fully shared across cores. Therefore, any task can evict data belonging to tasks running in other cores. Note that avoiding those interferences by means of software partitioning would have important side effects since it (1) increases the difficulty of task migration across cores, (2) makes data sharing more challenging (needed at least for communication purposes), (3) causes memory fragmentation, and (4) may be not even implemented for shared libraries or Operating System services.

2.2.2.3 Inclusive DL1 caches

DL1 caches are inclusive with L2 caches. This means that, on an L2 eviction, if the evicted line resides in any DL1 cache, it is also evicted from the DL1 cache. Hence, even if a task is not using L2 cache contents explicitly and reuses its DL1 contents, it may suffer evictions due to other tasks evicting its DL1 cache lines from L2.

2.2.2.4 L2-memory Interconnect

The AMBA bus connecting L2 caches and memory may serialize requests, thus making contention have an impact in memory latency. However, requests are already serialized per cluster in L2 caches. Hence, the additional serialization caused by this bus has a limited impact in memory latency, which is expected to be relatively low since memory latency is typically much larger than the latency of a shared bus.

2.2.2.5 Memory Access Contention

The fact that 6 cores share a memory controller and a single memory subsystem may make memory contention have an impact in memory latency. However, the Juno board uses DDR3 DRAM modules, which allows them to process memory requests quickly w.r.t. the SoC operating frequency. In particular, LPclus and HPclus operate at 950MHz and 1200MHz respectively, the L2-memory AMBA bus at 400MHz, and DDR3 DRAM at 1600MHz.

2.2.2.6 Recap

In summary, shared resources may lead to contention interferences. From this qualitative assessment we expect the impact of contention in the interconnects and DRAM memory to be limited, whereas the impact of shared L2 cache space and inclusive DL1 caches may be potentially high.

2.2.3 Quantitative Assessment

This section presents the quantitative assessment of the timing behavior of the Juno SoC with regard to its adequacy for CRTES. First, we present the experimental setup. Then, we provide results identifying pros and cons of the ARM big.LITTLE architecture for its use in CRTES.

2.2.3.1 Experimental Setup

To perform measurements, we use stressing benchmarks (SB) that traverse a data vector, in line with the proposals in the literature [12] and analogously as for the SnapDragon 810 processor. The size of the vector is set to be 2^NKB and accesses are performed with a stride of 64B, thus matching cache line size. PMCs are reset before each traversal and read right after. Each experiment is performed twice reading different events so that all events of interest can be monitored.

Collecting measurements on a real board poses some challenges, such as the difficulties brought by the monitoring software to interface the counters and the noise of the interrupts. These factors may also interfere with the measurements. Hence, some measurements can be abnormally high or low. To mitigate the impact of noise, traversals are repeated 1,000 times (per set of events read). The pseudo-code of the stressing benchmarks is shown in Algorithm 1, where the main loop is unrolled 16 times to reduce the relative overhead of the loop management instructions. Such an approach produces 1,000 measurements for each event. The first one is intended to warm up caches. Hence, its results can be regarded as irrelevant for our study. Then, to discount outliers we keep the median, which is still subject to some noise. However, filtering outliers automatically with this approach allows identifying trends as needed for this work.

Two main experimental setups are considered. The first one (see Figure 2.4 (a)) runs the monitored SB (SB_{mon}) in one of the cores of the LPclus. All the remaining cores remain idle. The only exception

is one core of the HPclus, which runs the Real-Time Operating System (RTOS), marked with an asterisk. Such activity has been placed in a different cluster to minimize unwanted interference. The second setup is identical to the first one, but all contender cores run SB also (see Figure 2.4 (b)). To simplify the discussion, we focus only on the results of experiments conducted where N is identical across all SB in all cores.

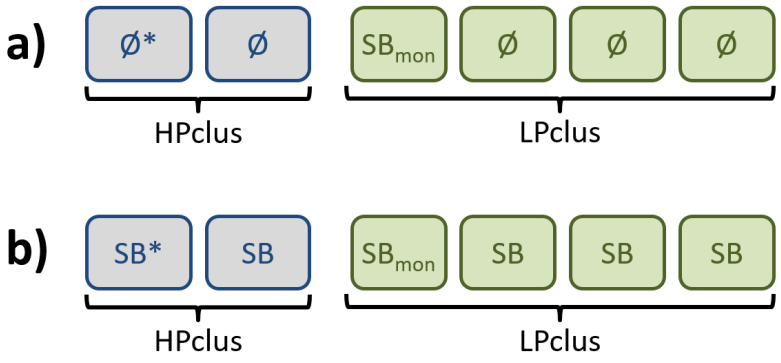


Figure 2.4: Experimental setup (a) in isolation and (b) with contention.

Vector sizes per SB vary between 1KB and 2MB (so $10 \leq N \leq 21$), hence the SB_{mon} may either fit in DL1, exceed DL1 and fit in L2, or exceed L2 cache space.

2.2.3.2 Assessing Stressing Benchmark Results

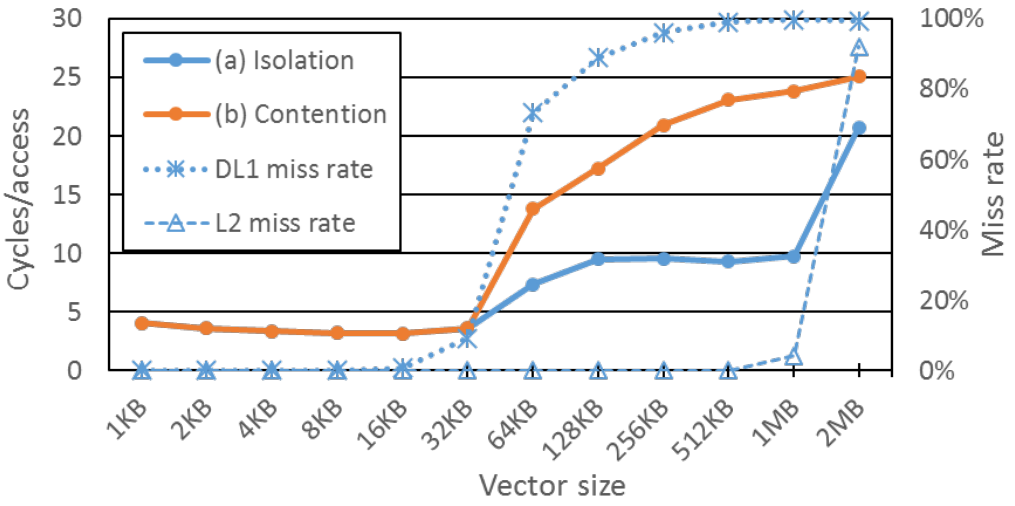


Figure 2.5: Cycles per access for the two setups when varying vector size.

Figure 2.5 presents the results of the experiments in both setups in the form of cycles per (memory) access, or CPA for short (straight lines). Such a metric allows comparing all measurements regardless of the size of the vector. The plot also includes DL1 and L2 miss rates w.r.t. the total number of accesses for the setup in isolation. While this is the usual miss rate for DL1, since all accesses are directed to DL1, it is not for L2. However, we report the number of L2 misses w.r.t. the total number of accesses so that, if the total number of L2 misses is very low, this metric is also very low. Otherwise, we could have a very high L2 miss rate (e.g. 50%) with very few L2 misses (e.g. 1,000,000 DL1 accesses, causing only 4 L2 accesses, out of which 2 are L2 misses). If the absolute number of L2 misses is very low w.r.t. the total number of memory accesses, then the L2 miss rate (L2 misses divided by L2 accesses) is irrelevant in practice.

Results in isolation show that the CPA is slightly above 3 cycles when the vector size does not exceed 32KB. Such a vector size fits in DL1 and hence, vector accesses are expected to hit, as reflected in the DL1 miss rate. Note that, since each memory access comes along an arithmetic operation to increase the index, 3 cycles is expected to be the latency to execute both, the memory access and the arithmetic operation. We observe that the CPA slowly decreases when moving from 1KB to 16KB. This occurs because the code inside the loop has some prologue and epilogue to set up and read the PMCs. For a larger vector size, the relative impact of such code decreases. We also note that the lowest CPA value is 3.15 for a vector size of 16KB, still above 3 cycles. This occurs because every 16 memory accesses, there are few arithmetic instructions to check the loop condition. We also note that whenever the vector size matches DL1 size exactly (32KB), the CPA increases to 3.62. This occurs because prologue and epilogue fetch few cache lines that cause some DL1 evictions in each iteration and hence, some additional L2 cache accesses. Hence, DL1 miss rate grows from $<$1% to 9%.

When moving to vector sizes in the range 64KB-1MB, the CPA reaches values slightly above 9 cycles, with the exception of the case for 64KB. In all these cases data does not fit in DL1, but fits in L2, as reflected in DL1 and L2 miss rates. As for the case when data fit in DL1, we observe that larger vector sizes slightly decrease the CPA until we reach the exact L2 size (1MB), when CPA slightly increases. The CPA for 64KB is abnormally low. The source of this unexpected value is still under investigation, although it seems to relate, to some extent, to the DL1 miss rate, which is around 73% when we would expect it to be close to 100%. However, we are pessimistic on whether the cause can be identified given the limited documentation available, which omits details on, for instance, whether some form of buffering exists between DL1 and L2, or whether translation lookaside buffers (TLBs) could create further delays.

Finally, for a 2MB vector size, L2 cache space is exceeded and virtually all vector accesses reach memory, thus producing a CPA slightly above 20 cycles. This information is also reflected in the L2 miss rate.

In the case of the setup with contention (setup (b) in Figure 2.4), results for vector sizes between 1KB and 32KB match exactly those for setup (a) in isolation. This is expected since all cores hit in their respective DL1 caches and hence, no contention occurs in shared resources.

For vector sizes between 64KB and 256KB, the vectors of all cores in LPclus still fit in L2 (i.e. $256KB \times 4 \leq 1MB$). Hence, there is no meaningful contention for L2 space, since only some residual contention due to loop prologue and epilogue code is expected when vector sizes are 256KB. Therefore, the CPA increase w.r.t. the isolation setup can be attributed to contention in the L1-L2 interconnect and serialization of the accesses in L2. As the size of the vector increases, the number of consecutive memory accesses increases and hence, there are fewer non-memory instructions per access (due to loop condition check plus prologue/epilogue), and thus, the degree of contention per access increases. Performing an exhaustive assessment of all potential conditions to discover the maximum CPA is left for future evaluation, although the methodology needed to discover such a value has already been described in [12]. Note, however, that the degree of contention in the access to L2 is so high that, despite data fitting in L2 for the 256KB vector size, CPA is 20.9, slightly higher than the CPA of experiments in isolation when L2 cache space is exceeded (20.7).

For vector sizes in the range 512KB-1MB, L2 cache space is exceeded in LPclus, so the CPA becomes 23.1 cycles for 512KB and 23.8 for 1MB. This indicates that moving from a scenario with high contention in the accesses to L2 to a scenario where L2 cache space is exceeded can only cause a modest CPA increase.

Finally, for a vector size of 2MB, the SB in the HPclus also exceed their L2 cache space, which is 2MB. Hence, the SB_{mon} experiences additional contention in its memory accesses, thus having a CPA slightly above 25 cycles. While such an increase can be noticed, it is also rather modest since DRAM memory is very fast in comparison with the operating frequency of the Juno SoC, and contention in the access to L2 proves to be the main performance bottleneck.

For the sake of completeness, we have considered the setup with contention, but placing an in-

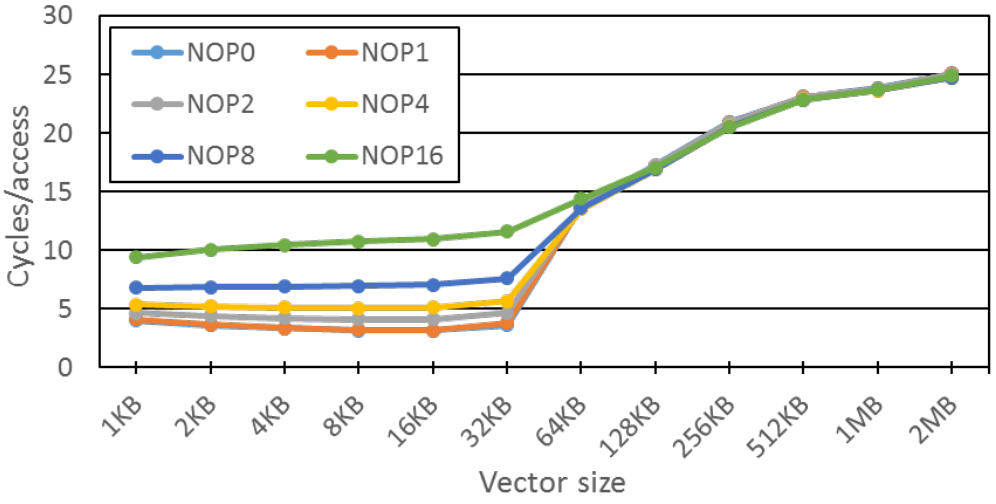


Figure 2.6: CPA with contention varying the number of NOPs between accesses.

creasing number of no-operations (NOP) between memory accesses. Results for NOP counts between 0 (the default case) and 16 are depicted in Figure 2.6. As shown, an increasing number of NOPs increases the CPA when data fits in DL1, since NOP latency can be hardly overlapped with DL1 access latency, thus impacting execution time. However, as soon as the vector size exceeds DL1 size (from 64KB onwards), execution time is completely dominated by the contention in the access to shared resources, so even 16 NOPs can be executed between two consecutive memory accesses without further increasing execution time, so that CPA remains constant. This is reflected in the fact that the CPA is roughly the same regardless the number of NOPs for any vector size equal or higher than 64KB.

2.2.4 Assessing an Avionics Prototype

For the sake of completeness, we evaluate our methodology on an avionics prototype. As avionics prototype, we use a critical real-time application as described in [13]. In particular, this application is an experimental version of a Flight Management System (*FMS*), based on an operational *FMS* architecture from Thales. This test application aims at performing in-flight guidance of aircrafts following a set of predefined flight plans. During the flight, the *FMS* is in charge of (1) determining the plane localization and (2) computing the trajectory in order to follow these flight plans. The experimental setup is similar to that used for stressing benchmark evaluation, as described in Section 2.2.3.1, but replacing SB_{mon} by the avionics application, *FMS*, and keeping the HPclus always without contention. As explained later, contention coming from the HPclus is irrelevant. Hence, the avionics application runs either in isolation or with contention against SB in the other cores, which create sustained contention. To further increase the amount of quantitative information obtained, we consider 3 setups with contention, varying the number of cores running SB from 1 to 3. Hence, *FMS* is evaluated in 4 setups: in isolation, against one SB, against two SB and against three SB (see Figure 2.7).

Since the duration of the *FMS* application is larger than that of the SB, SB are run repeatedly so that *FMS* experiences sustained contention during its complete execution.

Results. First, we analyze the execution time to assess the impact of contention. Figure 2.8 shows the slowdown for all setups when varying the vector size of the SB from 2KB to 4MB for each setup. Results are shown w.r.t. the fastest case. As shown, the slowdown is very low in all cases. The reason for this behavior is the fact that *FMS* runs for around 4.5 million cycles in isolation and performs only ≈ 400 accesses to the L2 cache. Hence, the *FMS* is highly insensitive to contention in shared resources. It keeps its working set in DL1 so the potential slowdown that it can suffer is roughly negligible.

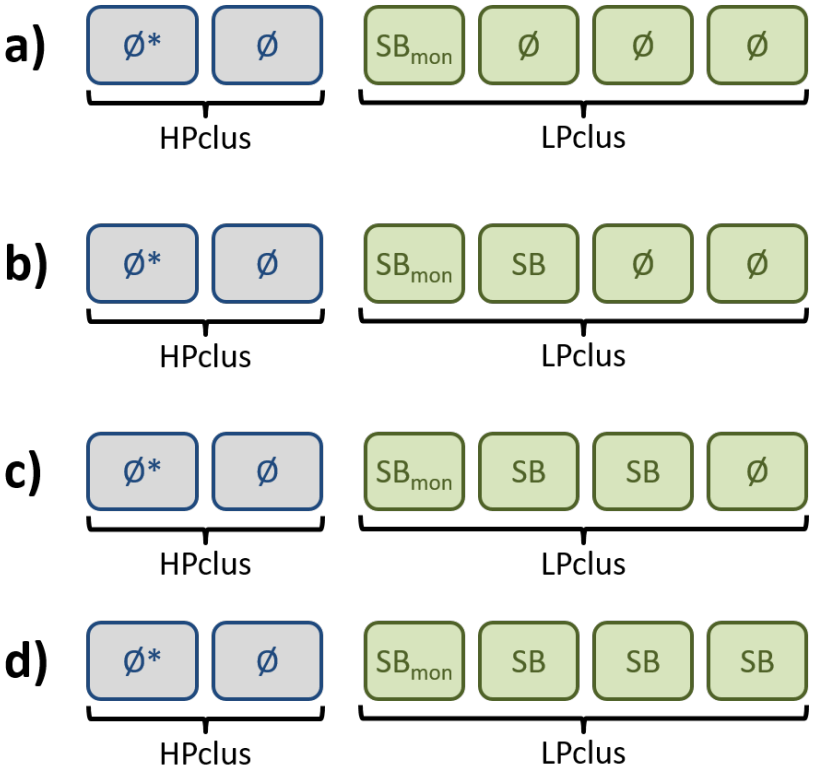


Figure 2.7: Experimental setup (a) in isolation, (b) with contention in 1 core, (c) with contention in 2 cores and (d) with contention in 3 cores.

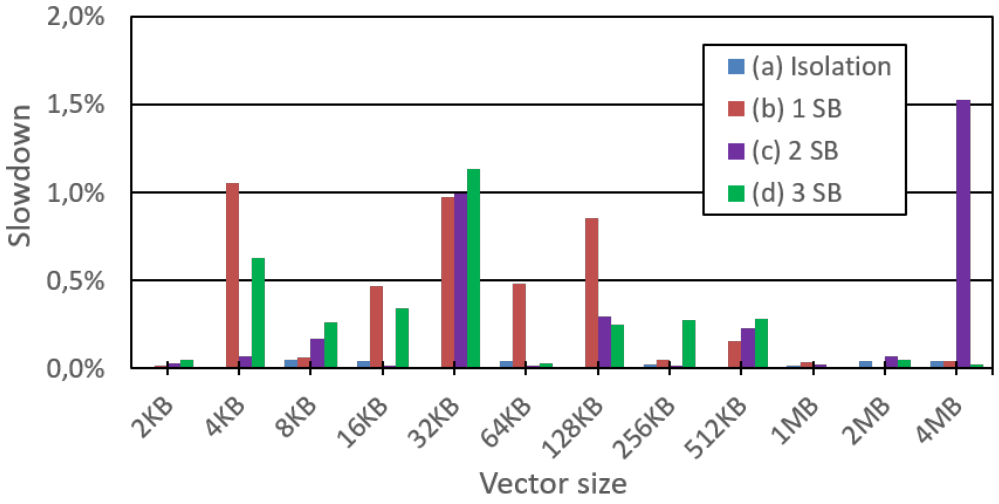


Figure 2.8: Slowdown w.r.t. fastest case when varying vector size.

We note that in some specific cases the slowdown is between 0.5% and 1.5%, thus reflecting some impact due to contention. Our first analysis indicates that 400 bus accesses, even if they experience the highest contention shown with benchmarks (22 cycles per access, from 3 to 25 cycles), contention would be only 8,800 cycles, which is below 0.2%. Hence, only contention in the access to shared resources cannot explain this behavior. Hence, we have analyzed the number of accesses to the L2 cache, which are shown in Figure 2.9. As we can see, those cases where slowdown is relatively high match those cases where the number of bus accesses grows from 400 to above 4,500. The reason for this increase in the number of L2 accesses is the fact that DL1 is inclusive with L2. Hence, the

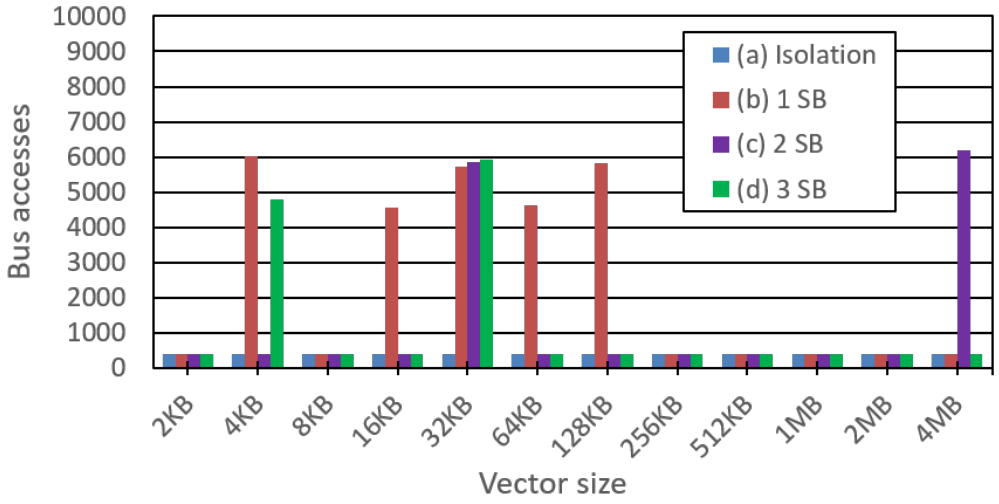


Figure 2.9: Bus accesses when varying vector size.

FMS keeps its working set in DL1, but the SB may evict *FMS* data from L2 in some cases, thus causing DL1 evictions and thus, an increased number of L2 accesses. This explains the slowdown. In particular, if we consider the case with the highest number of L2 accesses, 6,172, we realize that the worst potential slowdown that could ever occur is 3%. Obviously, contention experienced is not the highest one since *FMS* accesses may incur lower contention due to the time alignment of events. For instance, an access may arrive right when an access from another core has been served and thus, it finds fewer requests in front that can delay the access.

Overall, we realize that the *FMS* is a good candidate application for this architecture since, despite it may suffer some undesired contention due to inclusivity evictions in DL1, those may cause low slowdown in practice.

2.2.5 Summary of Lessons Learned

Our qualitative and quantitative assessment of the timing behavior of ARM big.LITTLE architectures through the ARM Juno board provide valuable lessons:

- The variation in terms of latency between L2 hits and L2 misses is large. In our results in isolation they are 9 and 20 cycles respectively. Hence, a task hitting L2 cache often is highly vulnerable to L2 cache space interference, which may increase execution time by a factor above 2x.
- Contention in the access to L2 due to access serialization can be as significant as the impact of transforming L2 hits into L2 misses. In fact, L2 latency without and with contention is also 9 and 20 cycles respectively, hence a factor above 2x.
- Access latency of DRAM (≈ 25 cycles) is relatively low in comparison to access latency of L2 (≈ 20 cycles) in the presence of contention, hence a factor around 1.25x.
- Tasks hitting in their local DL1 are highly insensitive to contention. Eventually they might suffer some DL1 evictions due to inclusivity, but they are only expected to be significant if the degree of L2 thrashing caused by contenders is very high. Unless very unfortunate cache placement occurs where the task under analysis and its contenders compete for very few L2 cache sets, one might expect that contenders need to thrash the complete L2 cache to evict all DL1 data of the task under analysis. Hence, the smaller the data set, the less frequently DL1 misses due to inclusivity evictions will occur.

Based on these observations, we can raise the following recommendations for the use of ARM big.LITTLE architectures in the context of CRTES:

- Critical real-time tasks must avoid hitting in L2. This implies either hitting in DL1 (in the smallest number of lines possible) or largely exceeding L2 capabilities. Examples of such tasks would be control tasks with limited working sets and streaming tasks. In fact, these types of tasks could easily coexist without jeopardizing their execution times due to contention.
- Exploiting L2 cache space must only be allowed for non-real-time tasks (or non-critical real-time tasks) since they are highly vulnerable to interference due to L2 cache access serialization and L2 evictions caused due to contention.
- Documentation needs to be sufficient and accurate. This is the case for the Juno SoC, but it is not for the SnapDragon 810 processor.

2.3 AURIX TC27x

In this section we integrate the concept of signatures and templates described in WP3 on an Infineon AURIX platform. In particular, this section builds upon the concepts presented in Section 4.6 of deliverable D3.2, whose undertstanding is required for the right interpretation of the details in this section. This hardware platform has a number of features different from general purpose processors, whose structure is quite close to that of the SnapDragon 810 and the Juno SoC. Instead, AURIX processors are specifically designed to be time-predictable. Still, contention in the access to shared resources can occur and impact execution time significantly. Hence, we model multicore contention as described next.

2.3.1 Preliminaries

2.3.1.1 Reference Platform

We address an AURIX™ TC277 board [20] with three different TriCore™ processors, sharing a common ISA but differently characterized w.r.t. computational and energy efficiency: a low-power core with a simple microarchitecture (16E) and two higher-performance cores (16P), equipped with high-performance features such as more complex pipelines, dynamic branch predictors and larger caches.

As summarized in Figure 2.10, all processors are equipped with separated core-local memories (scratchpads and caches) for instructions and data[3]. The 1.6P processor includes large scratchpads, respectively 32 KB and 128 KB for Program ScratchPad RAM (PSPR) and Data ScratchPad RAM (DSPR). Caches are relatively smaller, with 16 and 8 KB respectively for Program Cache (PCACHE) and Data Cache (DCACHE). Processors are connected to a shared 'memory system' through the Shared Resource Interface (SRI). The shared memory system comprises a SRAM device, accessed via the Local Memory Unit (LMU or lmu) and a FLASH device, accessed via the Program Memory Unit (PrMU) through three independent interfaces, two for code (i.e. program) and one for data.

LMU and PrMU memory areas can be accessed both in cacheable or uncacheable mode, depending on the address segment used. System software statically defines where the different elements in the application (e.g., stack, functions, and data) are mapped and the cacheability options.

In the AURIX platform, contention happens when parallel requests (from different *master* modules such as the cores) are directed to the same slave interface (e.g. LMU, PrMU). Requests to target modules (slaves) in the SRI are arbitrated according to a priority-driven policy, where masters (cores in our case) are mapped by configuration to a priority class. When a priority class accommodates more than one master (e.g. several cores running with the same priority), then a round-robin mechanism is used.

[3]The 1.6E processor implements a dedicated buffer in place of a data cache.

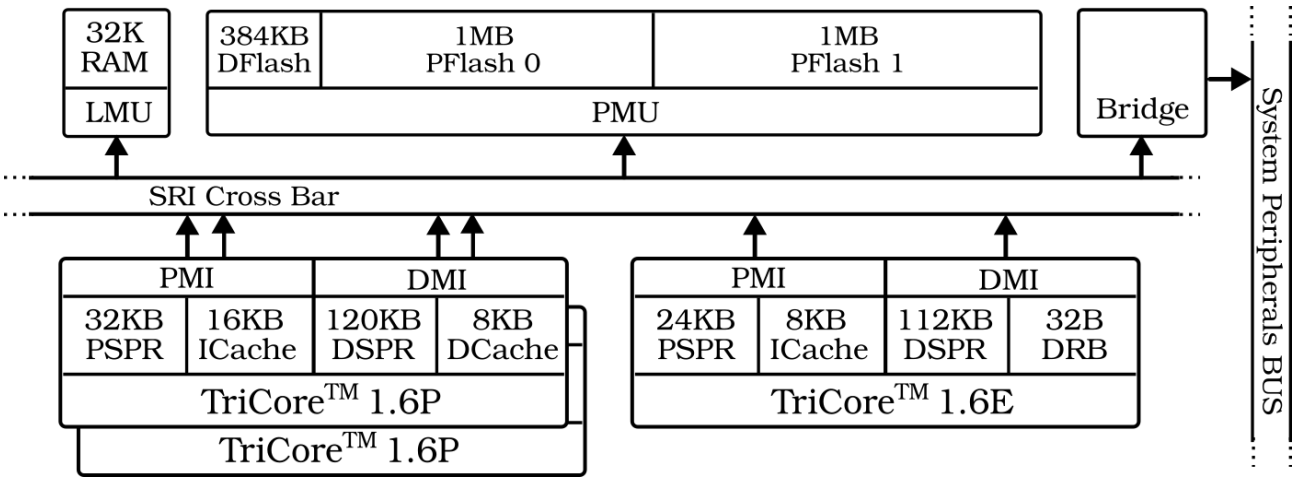


Figure 2.10: Block Diagram of the AURIX™ TC-27x

2.3.1.2 Basic Notation and Assumptions

We consider one task under analysis (*tua* or τ_a), a time-critical task for which a WCET estimate is to be derived, and a contender task (τ_b). In the general case, τ_a's requests can map to any priority class in the SRI, depending on the priority assigned to the core where τ_a is executed. However, we restrain our focus to those cases where τ_a's requests can potentially incur a relevant amount of interference. Since τ_a is time-critical, we can exclude scenarios where the requests of contenders are mapped to higher priority classes. Instead, we assume τ_a requests are mapped to the same (shared) priority class of the contender's ones, i.e. the typical case when contenders are also time-critical. In this scenario, stalls can happen whenever τ_a issues a request to a target and the request arrives right after the arbitration turn has selected another request by a contender task on the same target[4]. Hence, contention is determined on a per-target basis and depends on the round-robin arbitration policy. It is worth mentioning that the SRI also implements an anti-starvation mechanism to provide a minimum service to lower priority requests. However, since we assume all requests belong to the same priority class, the anti-starvation mechanism can never be triggered. How to account for scenarios with mixed priority classes is left as future work and, in the rest of this section, we build on the assumption that all requests from all tasks belong to the highest priority class.

In general, the exact contention a given task under analysis τ_a will suffer from another contender task τ_b cannot be effectively predicted as it depends on how inter-core requests interleave in the interconnect. Moreover, analysis is often forced to build on information obtained on the task under analysis in isolation as information from joint execution cannot be derived until late design phases, when applications are integrated together. Therefore, contention models cannot determine *exactly* how tasks will interleave in reality once integrated. Contention models have no option other than being conservative and assume that contenders' requests align in the worst possible way with τ_a's requests [34, 24].

In the TC-27x, contention is also determined by the number of requests of τ_a and the number and type of requests of τ_b, i.e. it does not depend on the type of requests sent by τ_a. As a result, a conservative contention model is forced to assume τ_a is delayed by each request of its contender by the duration of that request, which in turn depends on the target resource (t) and operation type (o). Restricting the scope of our analysis to the Electronic Control Unit (ECU)[5], the relevant target resources accessible through the interconnect in the AURIX platform comprises LMU and PrMU. The latter is further broken down in different Flash memory interfaces for data and program (code), which

[4] In fact the same effect is obtained in general when τ_a requests lose the arbitration round because of higher-priority requests, with the only difference that the anti-starvation mechanism could intervene.

[5] We do not consider external peripherals and special communication devices.

Table 2.1: Definitions used in this section.

Acronym	Description
Target Resources and Operation Types	
$\mathcal{T}$	Target resources in the SRI
$\mathcal{O}$	Types of operations on a SRI target
$\overrightarrow{\mathcal{O}} = \{o', o'', ...\}$	Operations in descending latency order
Access Counts	
n_a,	Total access count of τ_a
n_a^{co}, n_a^{da}	Data and code access count of τ_a
$\hat{n}_a$, $\hat{n}_a^{co}$, $\hat{n}_a^{da}$	Upperbounds to previous values
$n_a^{t,o}$	τ_a's accesses of type o to resource t
$nr_a^{t,o}$	τ_a's remaing o-type (or shorter) accesses
Latencies	
$l^{t,o}$	Access latency of o-type requests to t
$cs^{t,o}$	Stall cycles when accessing t with an access o
cs_a^{co}, cs_a^{da}	τ_a's code and data stall cycles in isolation
Δcs_a^{co}, Δcs_a^{da}	τ_a's increment in stall cycles due to contention

we denote dfl, and $pf0$ and $pf1$ respectively.

Table 2.1 summarizes the main terms we use. We consider the set of target resources $\mathcal{T} = \{dfl, pf0, pf1, lmu\}$. While each target might exhibit different latencies depending on the type of request (operation) processed (i.e., code/data reads or data writes), for the time being we differentiate only among code (*co*) and data (*da*) requests for all targets in $\mathcal{O} = \{co, da\}$, $\forall\, t \in \mathcal{T}$. Code accesses can target the $pf0$, $pf1$ and lmu, while the data accesses can target any resource: dfl, $pf0$, $pf1$ and lmu as presented in Figure 2.11.

2.3.1.3 Hardware Profiling

We develop our contention model on top of the AURIX™ Debug Support Unit (DSU) interface. The TC-27x comes with a set of multi-purpose PMCs that can be configured to collect data on both core-local and inter-core events. As baseline support to derive the latencies of all operations on the platform, we exploit the on-chip cycle counter (`CCNT`). The most relevant information related to inter-core contention has been identified in the `PMEM_STALL` and `DMEM_STALL` counters: these PMCs count the number of cycles the pipeline has been stalled when accessing the Program/Data memory interface respectively. Finally, as a complementary source information, we also exploit the PMCs related to cache performances and specifically to cache misses (i.e., `PCACHE_MISS`, `DCACHE_MISS_CLEAN` and `DCACHE_MISS_DIRTY`). The relevance of cache misses information comes from the fact that SRI accesses may be triggered in response to a miss event.

Owing to configuration constraints, only fixed subsets of PMCs can be collected over a single execution. Three executions were required to collect all PMCs we deemed relevant to build the contention model. We followed a strict measurement pattern where measurement noise is completely removed by a smart enabling and disabling of PMCs.

Maximum observed end-to-end latencies for SRI transactions in isolation are reported in Table 2.2. Note that the reported latency is the maximum between read and write operations per SRI target as we are only interested in discriminating between code and data operations. Table 2.2 also report $cs^{t,o}$ as the amount of stall cycles incurred in the *best-case* for single accesses in isolation to each SRI target. Best-case stall counts should take into account the effects of prefetching, pipelining in the SRI, etc. so that they can be used to compute an overapproximation of the number of SRI accesses of a given application or task.

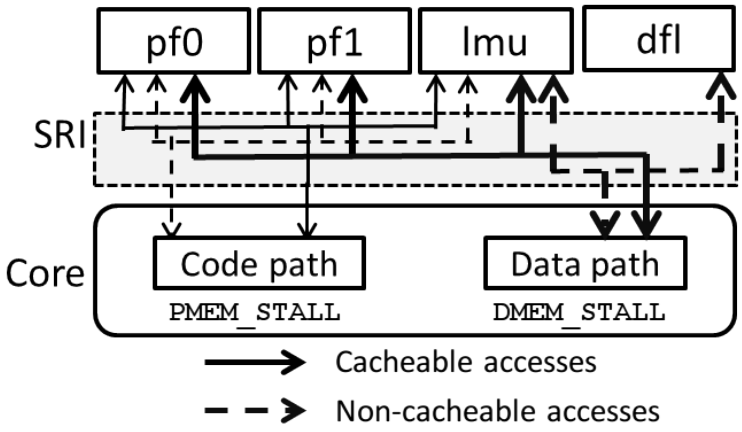


Figure 2.11: Code and data access paths to the SRI

Table 2.2: Maximum latency and minimum stall cycles

	Target (t)		
	lmu	pf	dfl
l^{max}	11(21)	16	43
l^{min}	11	12	43
$cs^{t,co}$	11	6	-
$cs^{t,da}$	10	11	42

2.3.2 Contention Models

Our first model assumes all relevant information on SRI access is available. Then, we show how we can cope with the lack of information and proceed with a step-by-step definition of a realistic and tight ILP-based contention model.

2.3.2.1 Ideal contention model for the AURIX

We explain the ideal model by analyzing a simplified scenario, in which τ_a performs more accesses than τ_b to every single resource, i.e. $n_a^t > n_b^t \ \forall t$. Under the worst-overlap described above, the model simply assumes that each request of τ_b delays τ_a by its duration. The worst-case contention τ_b can cause on τ_a, i.e. $\Delta_{b\to a}^{cont}$, is computed according to Equation 2.1, where $n_b^{t,o}$ is the number of τ_b requests of type o to target resource t and $l^{t,o}$ is the latency of that request.

$$\Delta_{b\to a}^{cont} = \sum_{t\in\mathcal{T}} \sum_{o\in\mathcal{O}} n_b^{t,o} \times l^{t,o} \tag{2.1}$$

In general, τ_a and τ_b may have an arbitrary number of requests, which can be trivially captured by the model.

2.3.2.2 Coping with limited information

The above ideal model builds on detailed information on (i) the latency of each operation for each target resource; (ii) the total access count per resource of the task under analysis; and (iii) total access count and operation type per target resource of the contender tasks. However, such information is not always (sufficiently) available, due to the limited hardware support in typical DSU for deriving $n_x^{t,o}$ for an arbitrary task τ_x. Furthermore, focusing on maximum $l^{t,o}$ for each resource and operation type inherently introduces pessimism by possibly discarding effects of prefetching on the SRI targets. We had to cope with these concerns in the TC-27x.

2.3.2.2.1 Latencies

We empirically derived the longest latency incurred by each target resource when processing a code or data request, as reported in Table 2.2. To measure the maximum latency to each target resource we considered the latency incurred by single accesses to a target (slave) resource in the SRI as measured by the on-chip cycle counter (`CCNT`). Note that dirty data misses latency on the LMU are reported within parentheses as they apply only on limited scenarios.

2.3.2.2.2 Access counts of τ_a

In the case we have a lack of specialized PMCs on SRI accesses on per-resource basis, we resorted to a secondary metric to derive upper bounds to the total (data and code) SRI requests of a task. To that end we used the existing stall cycle counters and in particular the `PMEM_STALL` and `DMEM_STALL` PMCs in our target AURIX platform. An upperbound to the number of SRI requests can be derived, separately for code and data, by dividing the total amount of stall cycles by the minimum amount of stall cycles per single request. We derived the latter by analyzing the `PMEM_STALL` and `DMEM_STALL` PMCs under a specific set of *stressing benchmarks* comprising a known number of requests of a given type to a desired target resource. This allowed us deriving a lower bound to the stall cycles a task suffers while completing a code and data request to a given target, $cs^{t,co}$ and $cs^{t,da}$. Since this processor does not have access counters for the slaves, we have to infer them from the stall cycles. Hence, we are interested in lower bounds to the stall cycles per request in order to upperbound the number of possible accesses to the slave (i.e. assuming that requests experience the lowest stall possible so that we maximize the number of possible accesses). The second factor for the computation of (an over-approximation of) the SRI traffic of a given task consists in the total amount of stall cycles suffered in isolation because of stalls in the memory interface. For a given task τ_x, these values can be obtained for code and data request separately (cs_x^{co} and cs_x^{da}) by running the task in isolation and collecting cumulative end-to-end values of `PMEM_STALL` and the `DMEM_STALL` counters.

From the information on stall cycles we can derive an upperbound to the number of code and data requests assuming that the entire stall delay has been caused by the requests of the shortest duration, i.e. $cs^{o}_{min} = min(\{cs^{t,o}\}_{\forall t \in \mathcal{T} \wedge \forall o \in \mathcal{O}})$. As depicted in Figure 2.11, the lowest possible stall cycles incurred for code and data requests in the AURIX™ platform can be derived by taking into account the architectural constraints on where code and data can be deployed:

$$cs^{co}_{min} = min\left(cs^{pf0,co}, cs^{pf1,co}, cs^{lmu,co}\right) \tag{2.2}$$

$$cs^{da}_{min} = min\left(cs^{pf0,da}, cs^{pf1,da}, cs^{lmu,da}, cs^{dfl,da}\right) \tag{2.3}$$

An upperbound to code and access counts of task τ_a can be derived by assuming that all requests are of the type incurring the lowest number of stalls (hence more requests are needed to cause cs_a^{co} and cs_a^{da}) by dividing the stall cycles by the duration of the shortest request.

$$\hat{n}_a^{co} = \left\lceil \frac{cs_a^{co}}{cs^{co}_{min}} \right\rceil \text{ and } \hat{n}_a^{da} = \left\lceil \frac{cs_a^{da}}{cs^{da}_{min}} \right\rceil \tag{2.4}$$

2.3.2.2.3 Per Target Access Counts (PTAC) of τ_b

Cumulative SRI access counts for code and data do not suffice for deriving a reasonably tight contention model. Since the SRI mechanism allows handling requests to different slaves in parallel (and each slave does incur different latencies), a good approximation of inter-core contention cannot be had without considering Per-Target Access Counts (PTAC). As shown in Figure 2.11, code and data accesses of a given task can go to different targets. With the current AURIX™ DSU support, despite having upperbounds for cs_b^{co} and cs_b^{da}, we cannot break them down in per-target components

(rightmost part in Equation 2.5).

$$n_b = n_b^{co} + n_b^{da} = \sum_{t \in \mathcal{T}} n_b^{t,co} + \sum_{t \in \mathcal{T}} n_b^{t,da} \tag{2.5}$$

In the next subsections we present three distinct models relying on different approximations of PTAC information.

2.3.2.3 fTC model for PTAC

A first simple model disregards per-target information altogether, using only cumulative information at code/data access level. In terms of access counts, we derive access data and code counts for τ_a and τ_b as described by Equation 2.4. In terms of delay, instead, the model exploits the maximum delay a code/data request from τ_a can suffer from τ_b, based on the type of requests that can go to each resource.

Code accesses can potentially address the *pf0*, *pf1* or *lmu* interfaces, hence the longest delay a code access from τ_a can suffer is defined by the longest latency it can suffer owing to τ_b accessing the same interfaces for code ***and*** data, as shown in Equation 2.6. Likewise, the maximum delay a data access can suffer is defined by Eq. 2.7 that matches the previous one with the exception that it factors in dflash (data) accesses from τ_b. Hence, the contention delay τ_a can suffer (Eq. 2.8) is defined as the number of code and data accesses of τ_a times the longest latency each request can suffer.

$$l_{max}^{co} = max(l^{pf0,co}, l^{pf0,da}, l^{pf1,co}, l^{pf1,da}, l^{lmu,co}, l^{lmu,da}) \tag{2.6}$$

$$l_{max}^{da} = max(l_{max}^{co}, l^{dfl,da}) \tag{2.7}$$

$$\Delta_{b \to a}^{cont} = \hat{n}_a^{co} \times l_{max}^{co} + \hat{n}_a^{da} \times l_{max}^{da} \tag{2.8}$$

This contention model is fully time-composable as it assumes that all τ_a requests always suffer the longest possible contention from its contenders. The model however is inherently pessimistic since to derive the maximum code/data access counts we assume requests are of the type incurring the shortest contention delay, and to derive the actual contention we assume each of them is of the longest latency.

2.3.2.4 Code-Data Based PTAC Model (CD-PTAC)

An immediate improvement to this model can be obtained by considering the number of τ_b code and data requests. Hence, only the minimum among τ_a and τ_b requests are affected by contention, see Equation 2.9. Code and data requests are collectively considered as they may address the same target.

$$\Delta_{b \to a}^{cont} = min(\hat{n}_a^{co} + \hat{n}_a^{da}, \hat{n}_b^{co} + \hat{n}_b^{da}) \times max(l_{max}^{co}, l_{max}^{da}) \tag{2.9}$$

The model is not fully time composable since contention bounds only hold under (up to) specific SRI access profiles from τ_b. However, this partially time composable variant is capable of deriving tighter contention bounds than the fTC one, under the condition that τ_a triggers fewer accesses than its contenders and/or the difference between l_{max}^{co} and l_{max}^{da} is non-negligible.

2.3.2.5 ILP-Based PTAC Model (ILP-PTAC)

More tight bounds can be obtained by considering upper bounds to per-target access counts of the contender task τ_b ($n_b^{t,co}$ and $n_b^{t,da}$). To that end we formulate the model as an Integer Linear Programming problem where we are interested in finding the per-target mapping of τ_a's and τ_b's requests that maximizes the contention suffered by τ_a.

2.3.2.5.1 Objective function

Our objective function maximizes the SRI stall cycles incurred by τ_a because of contention in code and data accesses (Δcs_a^{co}, Δcs_a^{da}). This is modelled in Equation 2.10 where $n_{b\to a}^{t,o}$ stands for the number of requests from contender τ_b targeting interface t for accesses of type o that are assumed to **interfere** with τ_a. Note that we break down interference between data and code accesses.

$$\begin{aligned}\Delta_{b\to a}^{cont} &= [\Delta cs_a^{co}] + [\Delta cs_a^{da}] = \\ &\Big[n_{b\to a}^{pf0,co} \times l^{pf0,co} + n_{b\to a}^{pf1,co} \times l^{pf1,co} + \\ &n_{b\to a}^{lmu,co} \times l^{lmu,co}\Big] + \\ &\Big[n_{b\to a}^{dfl,da} \times l^{dfl,da} + n_{b\to a}^{pf0,da} \times l^{pf0,da} + \\ &n_{b\to a}^{pf1,da} \times l^{pf1,da} + n_{b\to a}^{lmu,da} \times l^{lmu,da}\Big]\end{aligned} \tag{2.10}$$

As explained before, we assume that each interfering request of τ_b aligns in the worst manner with τ_a requests. Hence, each interfering request delays τ_a by $l^{t,o}$.

2.3.2.5.2 Constraints

Constraints in the ILP formulation are defined on the number of requests per target resource as follows.

Equation 2.11 captures that the number of data requests from τ_b that can contend with τ_a on the dfl is bounded by the maximum number of requests that τ_a and τ_b make to the dfl.

The next set of constraints, Equations 2.12, 2.13 and 2.14, cover the case of the $pf0$ that is a bit more complex, since unlike the dfl, it can receive both data and code requests. The constraint in Equation 2.12 captures that the maximum number of *inflictive* code requests from τ_b onto $pf0$ that interfere with both τ_a's code and data requests is bounded by the minimum between τ_b's code requests and all τ_a requests (still to $pf0$). Similarly, Equation 2.13 states that the number of inflictive data requests from τ_b onto $pf0$ is smaller than τ_b's data requests and τ_a's data and code requests to $pf0$. Finally, Equation 2.14 states a cumulative constraint on the total number of conflicts τ_a can suffer because of τ_b accesses to $pfl0$, which is bounded by the total number of τ_a code and data accesses to $pf0$.

The next two sets of constraints, Equations 2.15-2.17 and Equations 2.18-2.20 are the counterparts of Equations 2.12-2.14 but applied to the $pf1$ and the lmu respectively.

$$n_{b\to a}^{dfl,da} = min(n_a^{dfl,da}, n_b^{dfl,da}) \tag{2.11}$$

$$n_{b\to a}^{pf0,co} \leq min(n_a^{pf0,co} + n_a^{pf0,da}, n_b^{pf0,co}) \tag{2.12}$$
$$n_{b\to a}^{pf0,da} \leq min(n_a^{pf0,co} + n_a^{pf0,da}, n_b^{pf0,da}) \tag{2.13}$$
$$n_{b\to a}^{pf0,co} + n_{b\to a}^{pf0,da} \leq n_a^{pf0,co} + n_a^{pf0,da} \tag{2.14}$$

$$n_{b\to a}^{pf1,co} \leq min(n_a^{pf1,co} + n_a^{pf1,da}, n_b^{pf1,co}) \tag{2.15}$$
$$n_{b\to da}^{pf1,co} \leq min(n_a^{pf1,co} + n_a^{pf1,da}, n_b^{pf1,da}) \tag{2.16}$$
$$n_{b\to a}^{pf1,co} + n_{b\to da}^{pf1,co} \leq n_a^{pf1,co} + n_a^{pf1,da} \tag{2.17}$$

$$n_{b\to a}^{lmu,co} \leq min(n_a^{lmu,co} + n_a^{lmu,da}, n_b^{lmu,co}) \tag{2.18}$$
$$n_{b\to a}^{lmu,da} \leq min(n_a^{lmu,co} + n_a^{lmu,da}, n_b^{lmu,da}) \tag{2.19}$$
$$n_{b\to a}^{lmu,co} + n_{b\to a}^{lmu,da} \leq n_a^{lmu,co} + n_a^{lmu,da} \tag{2.20}$$

$$cs_a^{co} = n_a^{pf0,co} \times cs_a^{pf0,co} + n_a^{pf1,co} \times cs_a^{pf1,co} + n_a^{lmu,co} \times cs_a^{lmu,co} \tag{2.21}$$
$$cs_a^{da} = n_a^{pf0,da} \times cs_a^{pf0,da} + n_a^{pf1,da} \times cs_a^{pf1,da} + n_a^{lmu,da} \times cs_a^{lmu,da} + n_a^{dfl,da} \times cs_a^{dfl,da} \tag{2.22}$$

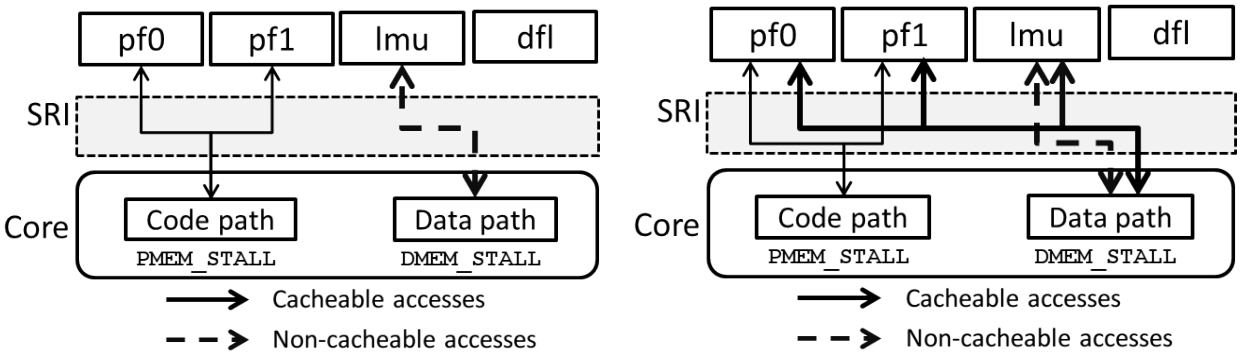
$$cs_b^{co} = n_b^{pf0,co} \times cs_b^{pf0,co} + n_b^{pf1,co} \times cs_b^{pf1,co} + n_b^{lmu,co} \times cs_b^{lmu,co} \tag{2.23}$$
$$cs_b^{da} = n_b^{pf0,da} \times cs_b^{pf0,da} + n_b^{pf1,da} \times cs_b^{pf1,da} + n_b^{lmu,da} \times cs_b^{lmu,da} + n_b^{dfl,da} \times cs_b^{dfl,da} \tag{2.24}$$

The following pairs of constraints wrap up the problem variables for the objective function. Equations 2.21 and 2.22 represent the SRI access profile (for code and data separately) from the single core execution: they reflect that τ_a makes $n_x^{t,co}$ and $n_x^{t,da}$ accesses to the different resources, which result in cs_x^{co} and cs_x^{da} stall cycles respectively. The latter values are exactly those obtained by reading `PMEM_STALL` and `DMEM_STALL` when running τ_a in isolation. Equations 2.23 and 2.24 are the equivalent constraints on τ_b execution in isolation. Note that discarding these latter constraints on τ_b would make the ILP model to be fully time-composable.

It is worth recalling that, while PMCs provide unique values for cs_τ^{co} and cs_τ^{da}, there are no unique stall values for each single $cs_b^{t,o}$ as the actual stall cycles are not constant and depend on pipelining and prefetching effects. As a conservative assumption, we consider the minimum observed stall cycles per request, with the inherent drawback of potentially accounting for more requests than those actually performed by the application.

2.3.2.6 Use of Scratchpads

The TC-27x includes relatively large code and data scratchpads for predictable and short latencies and are naturally used to accommodate part of a task code and data. The task working set exceeding the scratchpad size is mapped to extra-core memory areas via the SRI. Typically, some (possibly cacheable) code is retrieved from *pf0/pf1* and some data is shared among cores in the *lmu*. While technically possible, we do not consider SRI traffic caused by code and data requests targeting external PSPR and DSPR respectively. Admitting those types of request would introduce stalls in the memory interface even when accessing core-local memories. This not only complicates the charac-



(a) Scenario 1

(b) Scenario 2

Figure 2.12: Scenarios deployed in this section

terization of the SRI traffic but invalidates any attempt to approximate the number of SRI accesses through the PMEM_STALL and DMEM_STALL counters.

2.3.3 Evaluation

The AURIX™ TC-27x supports a variety of deployment configurations with different choices for code and data placement and cacheability. In this section we report on the assessment of the different contention models on selected deployment scenarios. The evaluation of these contention models on the automotive multicore use case will be reported conveniently in WP6.

2.3.3.1 Deployment scenarios and model tailoring

Interestingly, architectural constraints limit the range of possible deployment configurations. For our models, which do not discriminate between reads and writes, architectural constrains are summarized in Table 2.3.

Table 2.3: Constraint on code/data wrt SRI slaves.

	$pf0$	$pf1$	dfl	lmu
Code $	✓	✓	×	✓
Code n$	✓	✓	×	✓
Data $	✓(const)	✓(const)	×	✓
Data n$	×	×	✓	✓

Despite these constraints, the number of feasible placements for code and data is still large enough to allow for reasonable system-level flexibility. Knowledge on the application code and data layout can be injected into the contention model to obtain tighter results. Our generic ILP model can be easily tailored to capture any scenario by adding some constraints on target and access type. As a common deployment strategy, part of the application code and data is always deployed into the local scratchpads[6].

As a matter of fact, with respect to the rest of the application, some configurations, though admissible, are rarely used in practice. In the evaluation of our contention models, we focus on the two deployment scenarios in Figure 2.12, particularly representative of real-world deployment configurations. Without loss of generality, we assume deployment configurations equally apply to task under analysis and contenders. Next, we describe the scenarios and the model tailoring we applied. The PMCs used by the model and respective notation for τ_a and τ_b are reported in Table 2.4.

[6]We are not interested in this part of the application as it does not generate traffic on the SRI.

Table 2.4: PMC information available.

PMC	Task a	Task b
PMEM_STALL	$\texttt{PS}_a$	$\texttt{PS}_b$
DMEM_STALL	$\texttt{DS}_a$	$\texttt{DS}_b$
PCACHE_MISS	$\texttt{PM}_a$	$\texttt{PM}_b$
DCACHE_MISS_CLEAN	$\texttt{DMC}_a$	$\texttt{DMC}_b$
DCACHE_MISS_DIRTY	$\texttt{DMD}_a$	$\texttt{DMD}_b$

Scenario 1 (see Figure 2.12-a): part of the code and data fit on local scratchpads, some (cacheable) code is fetched from *pf0/pf1*, and some (non cacheable) data is shared among cores in the *lmu*. In this specific case, we can exploit the fact that the PCACHE_MISS counter, will hold the exact number of code requests from a task on the SRI (as all code requests through the SRI are performed in cacheable mode): $n_a^{co} = \texttt{PM}_a$ and $n_b^{co} = \texttt{PM}_b$. Nothing, instead, can be argued on data requests. The left column in Table 2.5 shows the instantiation of the three models (fTC, CD-PTAC, ILP-PTAC) to this scenario. Latencies used in the formulas are retrieved from Table 2.2.

Table 2.5: Tailoring of fTC, CD-PTAC and ILP-PTAC models.

Scenario 1	**Scenario 2**
fTC Model	
$\Delta_{cont}^{b\to a} \leq \texttt{PM}_a \times 16 + \lceil \frac{\texttt{DS}_a}{10} \rceil \times 11$	$\Delta_{cont}^{b\to a} \leq \texttt{PM}_a \times 21 + \left\lceil \frac{\texttt{DS}_a^{n\$}}{10} \right\rceil \times 11 + \lceil \frac{\texttt{DMC}_a + \texttt{DMD}_a}{6} \rceil \times 21$
CD-PTAC Model	
$\Delta_{cont}^{b\to a} \leq min(\texttt{PM}_a, \texttt{PM}_b) \times 16 + min\left(\lceil \frac{\texttt{DS}_a}{10} \rceil, \lceil \frac{\texttt{DS}_b}{10} \rceil \right) \times 11$	$Req(\Delta_{cont}^{b\to a}) = min\Big(\texttt{PM}_a + \texttt{DMC}_a + \texttt{DMD}_a + \left\lceil \frac{\texttt{DS}_a^{n\$}}{10} \right\rceil, \texttt{PM}_b + \texttt{DMC}_b + \texttt{DMD}_b + \left\lceil \frac{\texttt{DS}_b^{n\$}}{10} \right\rceil \Big)$ $\Delta_{cont}^{b\to a} \leq min\Big(Req(\Delta_{cont}^{b\to a}), \texttt{DMD}_b\Big) \times 21 + \Big(Req(\Delta_{cont}^{b\to a}) - \texttt{DMD}_b\Big) \times 16$
ILP-PTAC Model	
$n_a^{dfl,da}=0,\ n_a^{lmu,co}=0$ $n_a^{pf0,da}=0,\ n_a^{pf1,da}=0$ $n_a^{pf0,co} + n_a^{pf1,co} = \texttt{PM}_a$	$n_a^{dfl,da}=0,\ n_a^{lmu,co}=0$ $n_a^{pf0,da} + n_a^{pf1,da} + n_a^{lmu,da} \geq \texttt{DMC}_a + \texttt{DMD}_a$ $n_a^{pf0,co} + n_a^{pf1,co} = \texttt{PM}_a$

Scenario 2 (see Figure 2.12-b): part of the code and data fit on local scratchpads, some code is fetched from *pf0/pf1* (cacheable), some data is deployed to *lmu* (cacheable and non-cacheable), and finally constant data is found in *pf0/pf1* (cacheable). For the fTC model, we are forced to assume all cacheable accesses to any target can incur the latency of a dirty miss (see right column in Table 2.5).

To improve on the fTC model, we can benefit from the information from cacheable code as the PCACHE_MISS counter gives us the exact number of code requests on *pf0/pf1*. We cannot do the same for data since the sum of DCACHE_MISS_CLEAN and DCACHE_MISS_DIRTY provides the cumulative count of cacheable data requests but does not discriminate between the target of each access, which can equally be the *pf0/pf1* or the *lmu* (also accessed in non-cacheable mode).

This reasoning is reported in the pTC formulation in Table 2.5, where we assume we have the same deployment configuration for τ_a and τ_b. $Req(\Delta_{cont}^{b\to a})$ is used as a placeholder to reduce the length of the contention formula. Finally, Table 2.5 shows the tailoring of the ILP model, obtained by introducing few additional constraints. Again, the definition of the same constraints on τ_b may give a different degree of composability.

2.3.3.2 Assessment

Table 2.6: PMC readings for Scenarios 1 and 2.

		PM	DMC	DMD	PS	DS
Sc1	Core1	236544	0	0	3421242	8345056
	Core2	120594	0	0	1744167	4251811
Sc2	Core1	458394	200	0	2753995	86371
	Core2	233694	200	0	1404145	42826

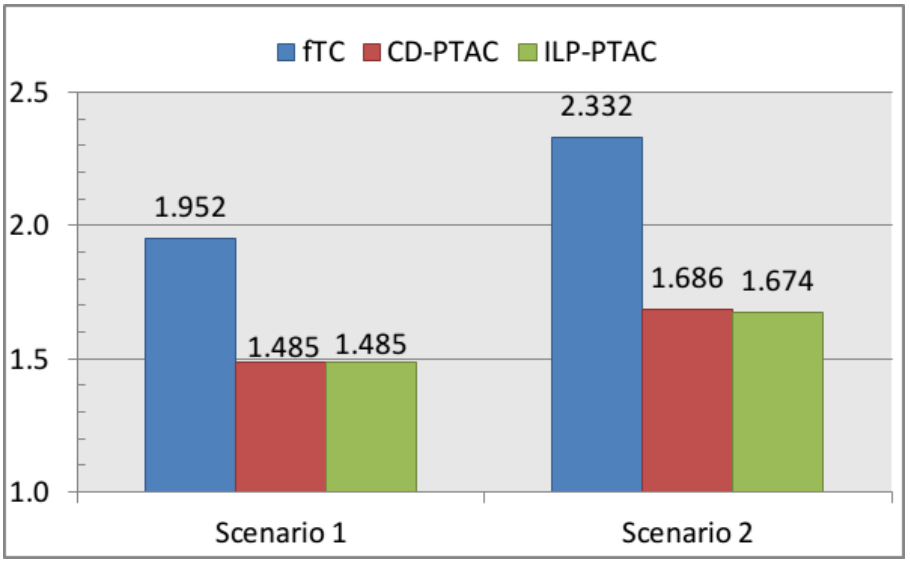


Figure 2.13: Model predictions w.r.t. execution in isolation.

We evaluated the different contention models on synthetic applications conforming to the identified scenarios. We first executed the applications (target tasks and contenders) in isolation to collect PMC readings. We then used the PMC information to feed the models and compare the so-obtained predictions against empirically observed contention-related delays.

In all scenarios, Core 1 and Core 2 (TC-1.6P) host the application under analysis and a contender respectively. The benchmark we used in both scenarios is a moderately data-intensive application fetching part of the code from *pf0/pf1* and performing data reads and writes in different memory segments, according to scenario and segment constrains. We considered as contender a reduced version of the same benchmark, performing approximately half of the accesses.

Under Scenario 1, contention only happens on *pf0/pf1* and lmu. The benchmarks are fetching part of the code from the PFlash and performing read and writes on the lmu. Scenario 2, instead, requires data to be deployed to the lmu (in both cacheable and non-cacheable mode) and to *pf0/pf1* (constant and cacheable). Contention is suffered on the same slave because of different types of accesses (code and data).

Table 2.6 reports the values observed for the PMC of interest under the two reference scenarios, for cores 1 and 2, running the target application and the contender respectively. The fact that dirty data cache misses are zeroed under both scenarios is not surprising as cacheable data accesses are typically performed to address constant data.

Figure 2.13 compares the predictions of the different models on the multicore execution of the target application against the timing results after applying our model and ILP approach. Data is normalized against the maximum observed execution time when the target application runs in isolation.

Results from both scenarios clearly indicate that the usefulness of fully time-composable bounds may end up being poorly useful in consideration of the pessimism they incur. Partially time-composable

models, instead, exploit the information from the PMCs to derive realistic bounds on contention under specific deployment configuration. In both cases, contention cycles are halved in comparison to fTC bounds. When comparing the code-data based with the ILP model, results depend on how close a specific scenario is to the conservative assumption of the CD-PTAC model, when bounding the number of SRI requests. In our experiments the inherent pessimism in the model is limited by the the fact that all code-related transactions are exactly intercepted by the program cache PMC. In the first scenario, the deployment constraints are already fully captured by the CD-PTAC model. Instead, the ILP formulation provides slightly better results in scenario 2 ($\sim$2% fewer contention cycles). It is worth noting that the ILP model also allows for configurations where contenders are only partially characterized, which leads to contention bounds with larger degree of time composability.

2.3.4 Summary

In this section, we presented three analytical contention models for the AURIX TC-27x platform, building on the existing PMC support. While fTC bounds are confirmed to be overly pessimistic, partially time-composable models provide realistic bounds that are valid for a wide range of contention scenarios. Further, formulating the contention as an ILP problem, guarantees better adaptability of the model to different configuration scenarios.

Chapter 3 METrICS: a Measurement Environment for Multi-Core Time Critical Systems

In this Chapter, we will present **METrICS**, a toolsuite dedicated to perform fine-grain time and resource access measurements in safety critical systems, allowing us to actually measure timing interference and search for the causes of this interference.

For the sake of completeness, we first recap in Section 3.1, the main challenges with regards to timing integrity and multi-core as already presented in Deliverable D3.1.

We then provide further details in Section 3.2 on the specific challenges of accurate and non-intrusive time-profiling for Real Time Operating Systems, followed in Section 3.3 by a presentation of METrICS, the profiling toolsuite we developed in the context of the SAFURE project.

Section 3.4 proposes an evaluation of METrICS in terms of accuracy, time intrusiveness and source code intrusiveness. Section 3.5 provides an example of METrICS usage on one of the application composing the SAFUFE WP4 prototype.

Section 3.6 introduces **METrICS server**, the host-side development allowing us to perform full scale automatized characterization campaigns involving days of evaluation on the target board, and hundreds of gigabytes of performance data collection. Finally, Section 3.7 presents **xTRACT visualizer**, a GUI that helps an expert data-mine and analyze the collected results.

3.1 Time Integrity Challenges for using Multi-Core COTS in Safety-Critical Systems

For the last decades, industries from the safety-critical domain have been using Commercial Off-The-Shelf (COTS) processors despite their inherent runtime variability. To guarantee hard real-time constraints in such systems, designers massively relied on resource over-provisioning, time and memory space partitioning, and disabling the features responsible for runtime variability.

The demand for cheaper equipment and more stringent SWaP (Size Weight and Power) constraints [4] makes the shift from from single-core to multi-core COTS processor for safety critical products appealing. But, as a consequence, the industry is facing an even larger trade-off in terms of performance versus predictability [25, 30].

On a multi-core processor, different pieces of software will be executed on different cores at the same time. Such software will, even if they are completely independent, compete electronically to use the shared hardware resources of the processor architecture, causing concurrent accesses to the same hardware as shown in Figure 3.1.

In the figure, several co-running tasks are executed on different cores and are trying to concurrently access the same main memory shared hardware resource. From a functional point of view, this behavior is not an issue and all the memory accesses will be successfully served.

From a hardware point of view, concurrent accesses to shared hardware resources are arbitrated, introducing inter-task or inter-application jitter defined as **timing interference** [15]. These timing interference are breaking the timing isolation principles required by the safety standards [22, 23, 36] of time-critical software.

The literature has proposed various solutions [14] to deal with timing interference but a quantitative comparison of such solutions is missing. Also, most of these solutions, presented in Deliverable

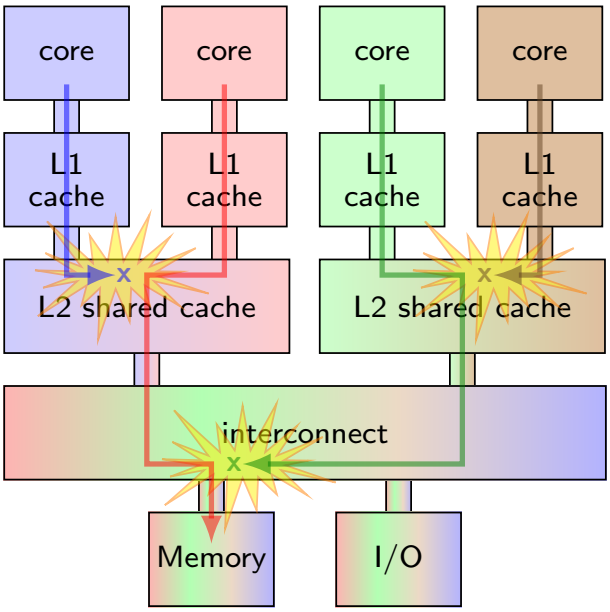


Figure 3.1: The challenge of timing-interference in multi-core systems

D3.1, require accurate measurement of either task runtime or some particular hardware resource loads, and therefore require some infrastructure to perform profiling.

3.2 The Challenge of Profiling with Real Time Operating Systems

Performance monitoring and profiling tools have existed for a long time to help the programmers with debugging their systems, optimizing their applications, or identifying bottlenecks. A wide variety of generic tools exists for non-RTOS systems [43] such as gprof [10], valgrind [32], or atom [9]. These tools rely on either OS features such as multi-threading, interrupts or timers, or either on pseudo-automatic code instrumentation to collect the required timing information.

In a real-time operating system, such features are either not available (with enforced static scheduling), restricted or prohibited due to their impacts on time determinism (such as the impact of interrupts on WCET). This is especially true for safety critical software that is constrained by drastic limitations due to the safety standards [22, 23, 36].

Beyond this limitation, even if collecting timing information is enough to observe timing interference, it is not sufficient to regulate the shared resource usage that causes interference due to resource contention. As a consequence collecting resource usage information is as critical as collecting timing information.

Generic tools such as oprofile [27] specialize in collecting such information by gathering the Performance Monitor Counters that are usually only available in privileged mode. The claim is that oprofile is low-overhead and non-obtrusive, and it is true from a non-RTOS point of view: Both the monitored application and the kernel remain untouched thanks to a dedicated kernel module. Also, the overhead mainly depends on the interrupt-based sampling frequency.

In RTOS systems, features like modular kernels do not exist, and using interrupt-based sampling is not an option for systems based on static scheduling. Such systems are relying on micro-kernels and modularity is even prohibited for safety and security reasons. Also ”low-overhead” does not have the same meaning for large scale systems running minutes to hour-long applications where a cost of tens of milliseconds is negligible and for periodic safety critical systems that are likely to have task deadlines in the order of 10 millisecond or less.

Furthermore, dealing with timing interference forces us to perform measurement at function-call or system-call level, where even a cost of tens of microseconds might not be acceptable.

Also, resource contentions (the main sources for timing interference) only occur at specific moments in time, during the cycles when an arbitration occurs. As a consequence, measurements and overheads have to be evaluated at cycle level.

Finally, even though sampling techniques are very efficient for best effort applications, such techniques can be very troublesome for safety critical applications that focus on how the worst case should behave. The sampling just acts as a filter that could filter out the worst case.

3.3 The METrICS toolsuite

METrICS consists of several components appearing in Figure 3.2, already presented in Delivrable D3.3. The brown parts in the figure correspond to, from bottom to the top, the selected embedded computer architecture and the PikeOS operating system above the platform support package (PSP) corresponding to the board. The blue parts correspond to the running applications we wish to monitor, each in their own partition.

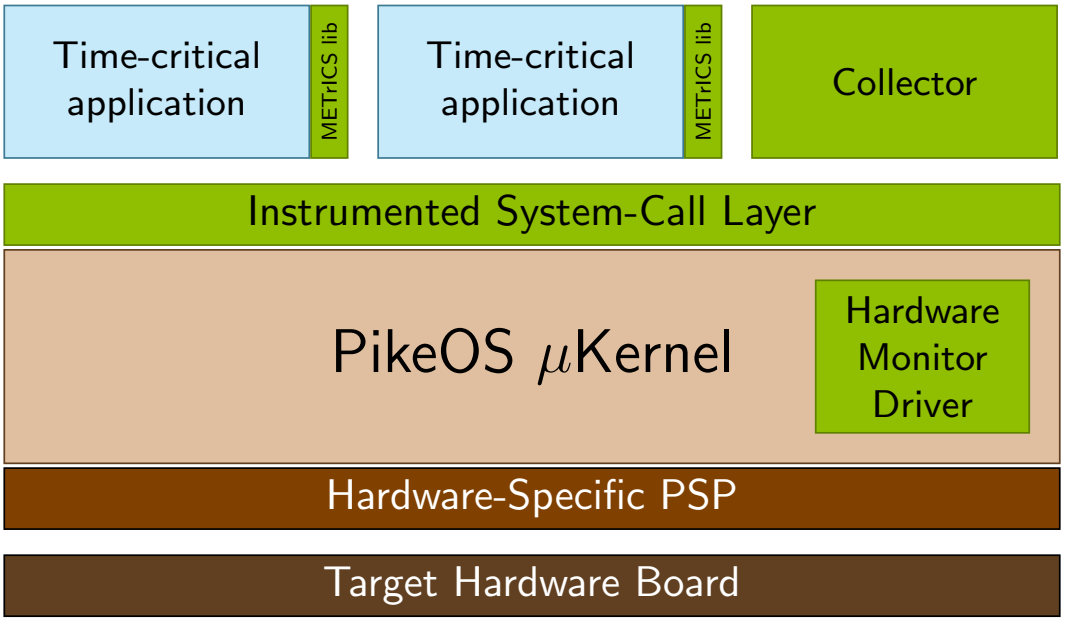


Figure 3.2: Architecture of the METrICS measurement tool

Finally the green parts in Figure 3.2 are the core components of the METrICS environment and are described below:

3.3.1 The METrICS Library

The **METrICS library** is meant to be linked with the running applications to provide them with an access to the measurement probes API, allowing the collection of time and resource access information. The library contains: 1) the instrumented system call layer; 2) the application instrumentation interface; 3) the user-level interface to the instrumentation kernel driver; and 4) the user-level interface to the collector.

Syscall instrumentation layer: The instrumentation of system calls of some PikeOS personalities (e.g. the APEX system calls in case of ARINC-653 applications) automatically inserts measurement probes before and after every system call. It especially allows us to determine communication times for the intra-partition and inter-partition communications that rely on such system calls, and that can be a significant part of the application's running time.

Application instrumentation: besides the syscall-level instrumentation, we provide the ability to manually insert measurement probes directly in applications. This is achieved by adding a pair of

functions (`metrics_probe_begin()` and `metrics_probe_end()`) around the section of the code to be monitored. Upon execution, these functions collect the highest precision time-base counter available (usually corresponding to the number of elapsed CPU cycles since booting) and the PMC registers of the current core. The latter function is also responsible for sending the monitored data to the collector.

3.3.2 The Hardware Monitor Kernel Driver

On most hardware architectures, the access to hardware performance monitor counters (PMC) requires supervisor-level privileges. Also on all supported hardware targets, the configuration of these registers to select the events that should be counted does require these privileges. As PikeOS legitimately prevents applications from getting such privileges, it was necessary to develop a **kernel driver** (a new feature since PikeOS 4.0) allowing us to select the PMC we wish to collect on the target.

The services provided by our driver are: 1) selecting the hardware events monitored by the hardware performance counters of the local processor core; 2) starting the counters; 3) stopping the counters.

The user interface for these services, implemented in the library, uses the regular ioctl calls provided by PikeOS for drivers. For each core of the ARMv8 Juno board, about 50 events can be selected for 6 different counters.

3.3.3 The Collector

The collector is implemented as a native PikeOS partition whose role is: 1) to define a shared memory space where each instrumented application will save its collected measurements; 2) to configure specific measurement scenarios (like selection of events via the driver); 3) to launch the measurement campaign (relying on PikeOS scheduling schemes); 4) to transfer the content of the shared memory to the host computer, either at the end of the measurement campaign (preferred to keep time intrusiveness level low) or during the run (to allow huge data collection).

In case of a failure (e.g. a deadline miss, or an unexpected timeout that is causing the application to stop), the collector also transfers the current memory content, allowing us to perform post-mortem analysis and debugging.

3.3.4 Internal Operations

For more details on METrICS internal operation, refer to Delivrable D3.3, where we explain how we provide the user with **accurate** runtime and resource usage measurements while minimizing both the **intrusiveness** and the **adherence** to the hardware.

3.3.5 Intrusiveness Trade-off

A major challenge in performance monitoring tools is its intrusiveness in the system it monitors. We distinguish **execution time intrusiveness** and **code intrusiveness**. The former limits the accuracy of the measurement due to the monitoring overhead, whereas the latter requires an effort from the developer to instrument the code of either the application or the RTOS.

Automated instrumentation tools commonly suffer from a trade-off between measurement granularity (from process level down to instruction level), time intrusiveness, and code intrusiveness. Enhancing one usually has a detrimental impact on the other two. In METrICS we chose to focus on **minimizing timing intrusiveness**, due to the fact that 1) we focus on time-critical and safety-critical applications where time determinism is of prime concern and 2) our major objective is the ability to characterize timing interference, and correlate them to shared hardware usage.

To minimize time intrusiveness, we used several development techniques that make METrICS as light as possible during the execution of the application under test.

Firstly, all initialization and external communications were implemented in the Collector and run outside of the operational scheduling. We configured the time-base and hardware performance counter registers to be used in user-space, avoiding time-consuming context switches associated with protected mode. Time-base and PMC accesses are performed in inline assembly code to minimize latency. Finally, the shared memory containing the sample collection is mapped into the memory of the processes to avoid accesses through system calls.

3.4 Evaluating METrICS Accuracy and Intrusiveness

The METrICS environment allows us to collect various measurements during the execution of safety critical applications, including execution time distribution and shared hardware resource access information. Rather than only extracting minimum, average and maximum values, the METrICS tool suite extracts the whole distribution of each measured data, allowing us to study the correlations between runtime and hardware resource usage.

This Section will evaluate METrICS in terms of accuracy, precision and intrusiveness. Next sections will present an example of METrICS usage and focus more on the ability to correlate timings and resource usage, as well as its ability to perform timing interference characterization.

3.4.1 Selection of time measurement mediums

To be able to perform fine-grain timing measurements, we need to rely on some kind of time measuring instrument. This time measurement medium could be either external, provided by any of the layers of the operating system or directly provided by the target processor as part of the instruction set.

The kind of events we wish to accurately measure includes complex functional chains (up to several seconds in avionics), runtime of individual tasks (with deadlines typically in the order of a few hundreds of milliseconds) or time spent in system calls (typically in the order of microseconds).

Each measuring medium relies on a software or a hardware mechanism that itself has a working period, thus limiting the obtainable **precision** to no less than this period. Also, each of these mediums actually consumes time to perform a measurement, making it impossible to accurately measure time below this measurement time **overhead**.

Additionally, for multi-core processors, clocks are not necessarily synchronous between all cores, introducing the concept of inter-core **clock offset**. If this offset does not remain constant (which could be the case if not connected to the same quartz oscillator (or clock PLL) or if the core is subject to dynamic frequency scaling), then it additionally introduces the problem of **clock drift**.

3.4.2 Portfolio of time measurement mediums

Using the APEX (avionic) personality of PikeOS, the operating system provides two system calls allowing us to measure time: `p4_get_time()` is provided directly by the PikeOS kernel, and returns the system time since boot in nanoseconds. `GET_TIME()` is provided by the APEX personality, and returns the system clock time, that is common to all processors.

However, being system calls, these time measurement mediums involve at least a context switch from the task to the operating system, and may involve switch(es) to privilege mode(s), depending on how the OS is handling system calls. The expected overhead for such switches is more than 1000 CPU cycles, and relying on such calls for time measurement will simply prevent us to measure short events such as context switches and system calls themselves.

On the other hand, both the ARM-v8 and PowerPC ISA provide low-level time measurement mediums. For instance, the e500mc/e6500 PowerPC provides two special registers that can be read with the `mfspr` assembly instruction: The `time base` register is a 64-bit register, set to 0 at board reset and incremented at the Platform Clock frequency, which is provided by a different PLL than the core clock frequency. The `time base` is thus 16 to 64 times slower than the Core clock frequency, 48 being

a common prescaler ratio. An advantage of the `time base` is that it corresponds to a global system clock, synchronized on all cores. The `alternate time base` register is a 64-bit register, also set at 0 upon reset, that increments at every core clock cycle. No specific guarantees are provided in the documentation about it being synchronous in all the cores.

3.4.3 Evaluation of time measurement mediums

The resolution of time measurement mediums are provided in their respective documentation. To evaluate the overhead of the above-mentioned mediums, we set up experiments using each medium twice in a row. The time offset between the two measurements is an upper bound of the time overhead. Each measurement pair was performed 180000 times to ensure that each overhead is not subject to variability. The results evaluated on a 1.8 GHz e6500-based NXP T2080 with PikeOS 4.1 are summarized in Table 3.1.

medium	layer	period	frequency	overhead
p4_get_time()	kernel	1 ns	n/a	240 ns
GET_TIME()	APEX	10 ms	n/a	10 ms
time base	register	48 cycles	37.5 MHz	1.67 ns
alt. time base	register	1 cycle	1.8 GHz	1.67 ns

Table 3.1: Resolution (period) and overhead

As expected, system-call-based mediums have a much higher overhead than special-register-based mediums. The APEX version is clocked with the Time Partition tick, used to define application time windows. Such a low resolution medium has a huge impact on overhead, making it impractical for fine grain timing.

Both special register mediums exhibit a 3 cycles (1.67ns) overhead, the `alternate time base` version providing a much better precision. For this reason, we chose this time measurement medium as the preferred method of measurement for METrICS.

With regards to timing offset between cores, only the `alternate time base` does not provide a null offset guarantee. We measured this offset against the synchronized `time base` and evaluated it being below 200ns for core 0 with respect to the other cores, and within the measurement precision between cores other than core 0. If a very high precision for inter-core measurements is necessary, the method we used for this evaluation can also be used for calibration at boot. Finally, none of the mediums showed a measurable drift among the 180000 runs.

3.4.4 Evaluation of a complete METrICS probe

A **METrICS probe** involves: 1) retrieving the timing information thanks to the core-dedicated special registers; 2) retrieving the performance monitor counters, again through direct register access; 3) retrieving thread-specific information from the OS; and 4) storing the collected information into the shared memory.

The intrusiveness of a METrICS probe in the source code is quite low, just adding a function call at the begin and the end of the code sequence to be monitored. Also, all the APEX system calls are automatically instrumented, requiring no further code modification. To perform this instrumentation, we overloaded the APEX function definitions with identical functions surrounded with our probes.

We also measured the intrusiveness in terms of timing of a complete METrICS probe by performing successive calls to METrICS probe the same way we did in previous section. Figure 3.3 presents the completion time results of such a probe, sorted over 180000 runs.

The probing time varies from 85ns up to 392ns. For 97% of the runs the overhead is below 110ns. For 2.998% of the runs it is between 110ns and 191ns. And for 0.002% of the cases, it is above 191ns and up to 392ns.

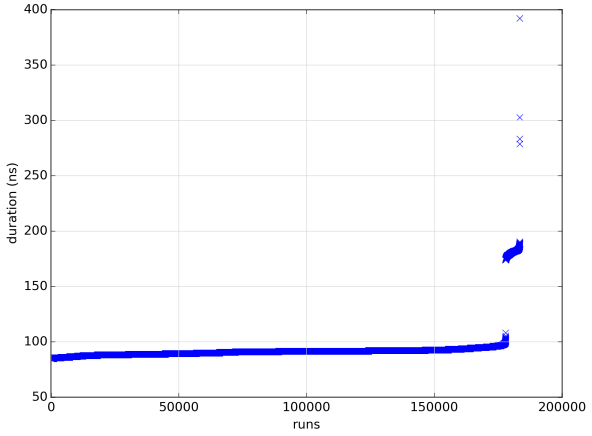


Figure 3.3: Completion time of a METrICS probe over 180000 runs

More precisely, the first three steps of a METrICS probe are quite stable with ∼25ns to retrieve both the timing and the performance monitor counters, and ∼20ns to retrieve the thread-specific information. The dominant part corresponds to the storage in the shared memory, varying from 40ns to 347ns, probably because of cache effects.

As a consequence of these results, the proposed measurement mechanisms presents a very limited overhead, and the highest achievable precision. Indeed the latency added by the full measurement probe remains in most cases shorter than the sole time measurement provided by the RTOS, and we used the fastest clock available in the target processor.

The next section will provide an example of usage of these **METrICS probes** for one of the application composing the WP4 avionic prototype.

3.5 Example of METrICS usage

In this section, we provide an example of METrICS usage to evaluate how different multi-core deployments of a simple avionic-like application react with regards to timing interference. The goal would be to evaluate which of these deployments is less sensitive to timing interference.

The experiments conducted in this section were performed with METrICS running on top of PikeOS 4.1. The selected application, part of the WP4 avionic prototype, is running with the PikeOS ARINC-653 personality commonly used in the context of avionic applications.

3.5.1 Evaluated Application

We evaluated an in-house application: eDRON (embedded Directed Rotodrone Operated Network), that is guiding a fleet composed of four quadricopter drones along a preset flight route. The purpose of this application is to mimic a representative behaviour of an avionic application, while exercising classical ARINC-653 communication mediums and proposing several multi-core deployment options. The software architecture of eDRON is presented in Figure 3.4. More details on the applicaton and the full avionic prototype will be provided in Deliverable D4.3.

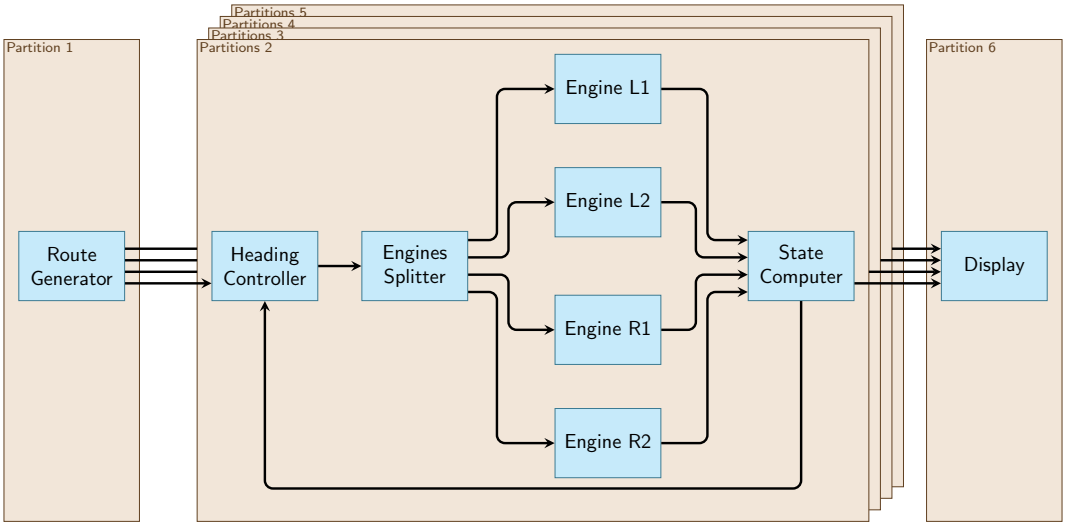


Figure 3.4: Software architecture of the eDRON application

This application is composed of six ARINC-653 partitions. The first one sets up the preset flight route for all the drones, the last one displays all the drone positions, and each of the four remaining partitions is dedicated to pilot a particular drone. These later partitions are composed of 7 tasks, with most of the computation being performed in the four engine control tasks, each being dedicated to control the velocity for one of the four engines of a drone, so that it follows the preset route.

When mapping such an application on a multi-core processor, we first need to decide what will run in parallel. The application offers two obvious parallelization schemes: **inter-partition** parallelism where each core will deal with a single drone, running the velocity control tasks sequentially for each drone; and the **intra-partition** parallelism where each core will focus on one particular engine, dealing with each drone sequentially. Some other parallelization options are available: for example parallelizing along the pipeline, or performing loop-level parallelization of the tasks, but those are beyond the scope of this project as they require deeper modifications of the application.

These two deployments have advantages and drawbacks: The Amdahl's law [1] may limit the performance of the intra-partition version, while the inter-partition version will benefit from the Gustafson's law [18], running independent applications in parallel. However, with regards to timing interference, the intra-partition version offers a white-box context where the partition scheduling can limit the level of interference between known tasks. The inter-partition parallelism on the other hand corresponds

to a black-box context where no easy control is possible to limit the interference level of another independent application.

3.5.2 Deployment evaluation

To evaluate the eDRON application with METrICS, we instrumented each task appearing in Figure 3.4 by adding a pair of begin / end probes around each task and partition. We then measured task runtimes while executing the application on one of the cores (single-core version), and later compared the results with both parallel deployments.

Note that while the full application is expected to be faster on a multi-core deployment, at the level of each task / partition, the runtime is only expected to increase due to possible timing interference. As a consequence, we expect to observe a slowdown at task level for the parallel versions.

Table 3.2 compares the runtime variability of a single Drone partition for three different deployments: a sequential / single-core deployment, and the deployments presented above with inter-partition or intra-partition parallelism.

parallelism	runtime (ms)				
	min	25%	median	75%	max
none	16.75	16.76	16.76	16.76	19.56
inter-partition	16.91	16.98	16.99	17.02	33.92
intra-partition	55.17	55.18	55.18	55.19	55.31

Table 3.2: Evaluating deployment impact on runtime and variability of one Drone partition

As expected, the single-core version is the deployment exhibiting the less variability with runtimes between 16 and 20ms, with a total execution time of 80ms to sequentially run the 4 drones. The multi-core deployment with inter-partition parallelism has similar lower bound and quartiles, but a much larger (x1.7) upper bound around 34ms. The deployment with intra-partition parallelism exhibits close to no variability, but with much larger runtimes of 55ms (x2.8).

We furthermore studied this latter version as the increased minimum runtime was suspicious. We figured out that the extra runtime was spent during the system calls performing inter- task communications.

3.5.3 Communication evaluation

Within the eDRON application, data communication is performed through the ARINC-653 system call layer. This API, corresponding to the PikeOS APEX personality, allows us to perform both intra-partition communication and inter-partition communication using either buffer-based or fifo-based communication services as illustrated in Table 3.3.

level	type	write / read
intra	buffer	DISPLAY_BLACKBOARD()
		READ_BLACKBOARD()
intra	fifo	SEND_BUFFER()
		RECEIVE_BUFFER()
inter	buffer	WRITE_SAMPLING_MESSAGE()
		READ_SAMPLING_MESSAGE()
inter	fifo	SEND_QUEUING_MESSAGE()
		RECEIVE_QUEUING_MESSAGE()

Table 3.3: ARINC-653 communication services

METrICS allowed us, thanks to the instrumented system call layer to automatically collect runtime information relative to these communication mediums during the experiments we ran to build Table

3.2.

The results corresponding to the deployment with inter-partition parallelism, that will serve as a reference, are presented as boxplots [39] in Figure 3.5 for both inter and intra-partition communication results. The results corresponding to the deployment with intra-partition parallelism will later appear in Figure 3.6.

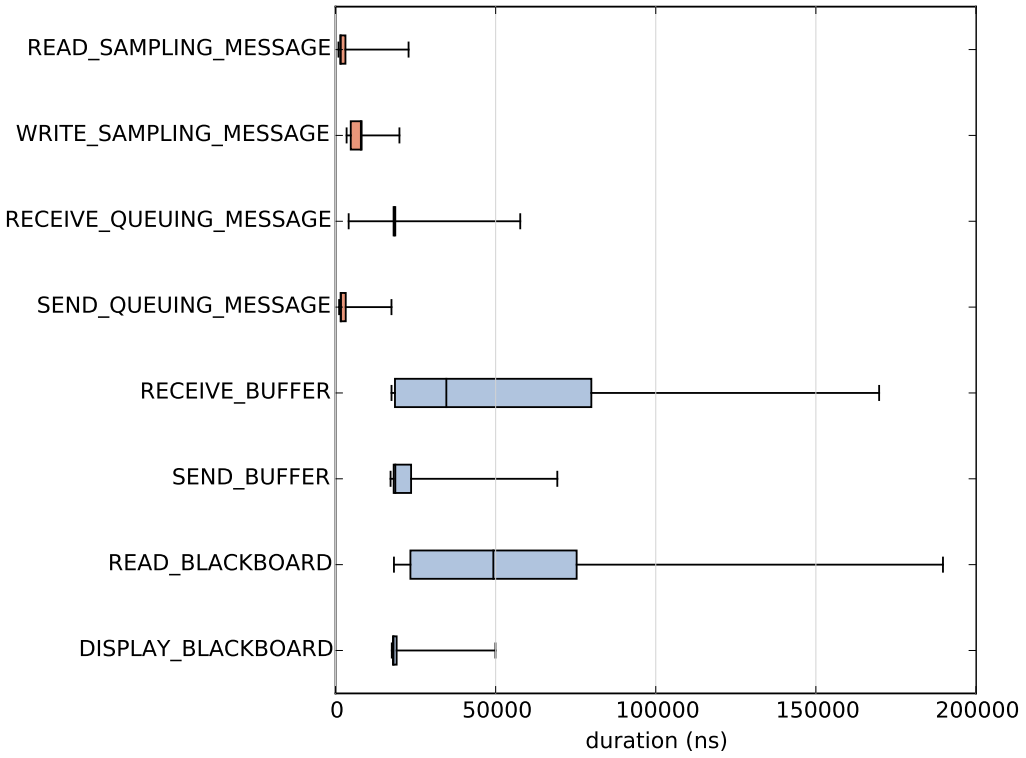


Figure 3.5: Measurements of inter-partition communication (top) and intra-partition communication services (bottom) for the deployment with inter-partition parallelism

Intra-partition communications are much more costly than inter-partition communication for the deployment with inter-partition parallelism, especially for the receiving functions. This is expected as running tasks sequentially introduce waiting time for communication receivers.

Figure 3.6 presents the communication time for the deployment with intra-partition parallelism. However, to be able to represent intra-partition communication times, we had to distinguish some outlier results. We therefore added a specific inset for both RECEIVER_BUFFER and READ_BLACKBOARD outliers with communication time above 36.9ms (36 965 107 ns).

Such high communication times occurs in 25% of the runs for READ_BLACKBOARD and in 49% of the runs for RECEIVE_BUFFER. With such a high occurrence rate, they are clearly responsible for the low performance of the deployment with intra-partition parallelism.

Further studying these strange timing behaviors, we figured out than the runtimes of these system calls were mostly equal to their timeout value (The ARINC-653 layer specifies communication with timeouts). Strangely, the communications that reached their timeout were successful as well. Increasing the timeout value or reducing the data size did not decrease the phenomenon, but setting infinite timeout fixed the issue.

We reported this strange behaviour to SYSGO and this issue is now fixed in the latest 4.2 version of the RTOS.

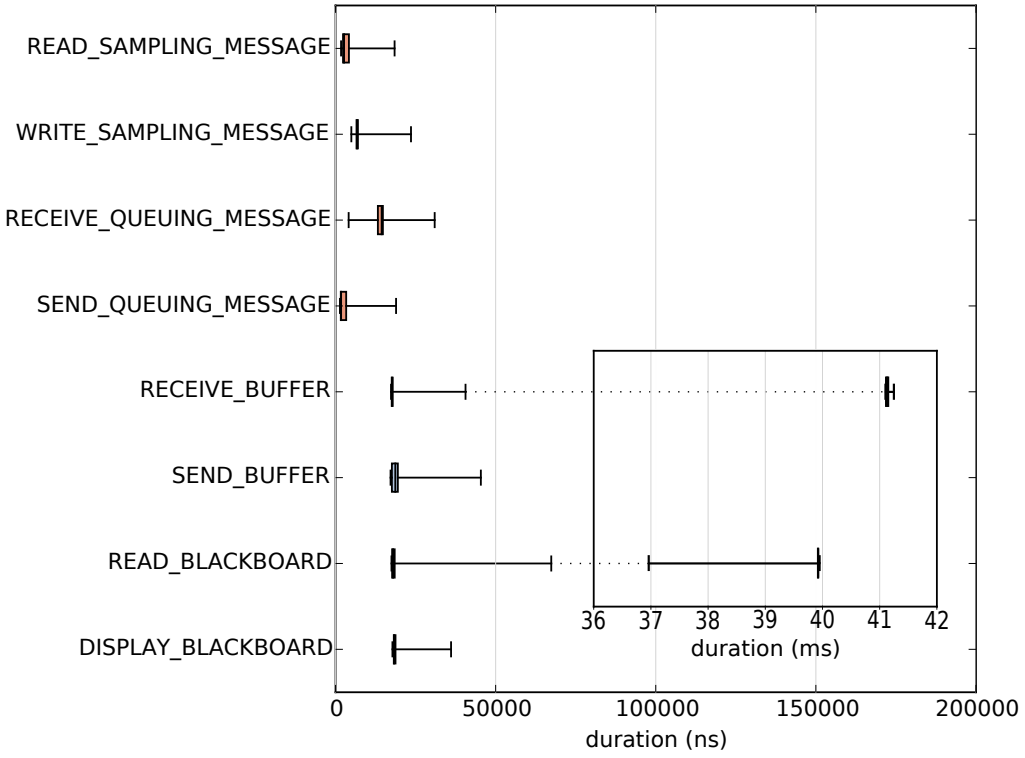


Figure 3.6: Measurements of inter-partition communication (top) and intra-partition communication services (bottom) for the deployment with intra-partition parallelism

3.5.4 Conclusion

Even tough the identified bug did not really allowed us to figure out which of the multi-core deployment is best with regards to interference, we have a proof of concept that METrICS can be used for such a study, and further can be used to identify software bugs with regards to timing.

In the next section, we will evaluate the number of total experiments that would be required to perform a full characterization of several deployments of the eDRON application while varying both the hardware target configuration and the application memory footprint.

3.6 Dealing with large Design Space: METrICS and Automation

A full characterization of a selection of multicore deployment of the eDRON application would require a number of experiments described in Table 3.4. It tests four deployment options including the three presented in previous section; three different options for the application memory footprint (fitting in L1 cache, in L2 cache or not fitting in caches); and whether or not to flush all caches between time partitions (a common practice in avionics to reduce variability).

Table 3.4: Number of runs for a full characterization

Application deployments	4
Buffer size	3
Cache Policies	2
Hardware Counter Selection	C_{41}^{2}
Number of iterations	1000
TOTAL RUNS	19.68 million

The target ARMv8 architecture provides a selection of hardware events that could be measured with performance monitor registers. Among those events, we identified a set of 41 events related to the shared hardware resource the cores may compete on. The ARM-v8 architecture of the JUNO board provides 6 performance monitor counters, meaning that testing all possible pairs of counters would require $C_{41}^2/6$ different configurations.

As shown in Table 3.4, this represents a large number of experiments to run, and some form of automation is desirable. Therefore we developed a series of Python scripts running on the host computer to automate all aspects of running a series of experiments: selecting the right executable file for booting the target by tftp, rebooting the board using a debug probe, providing the collector with the right set of hardware counters to use, receiving the results with a telnet connection to the server, and storing the CSV file in a directory named after the experiment configuration and date. This automation infrastructure allows us to perform large measurement campaigns by iterating on the steps presented in figure 3.7 without frequent user supervision.

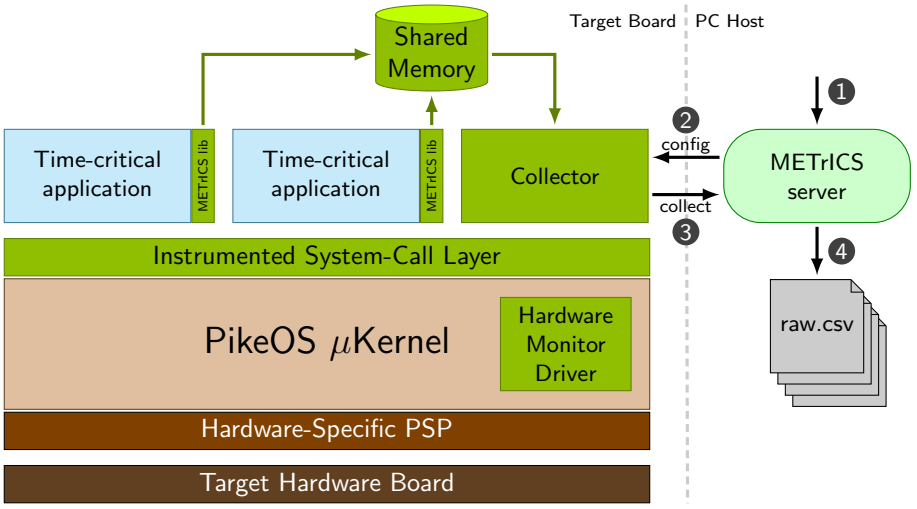


Figure 3.7: Host-side automation server, performing 1) selection of target executable and test configuration 2) configuration of hardware counters to use 3) collection of measurements and 4) storage of result files.

Such experimental campaigns generate a rather large amount of raw data, making the direct analysis quite difficult. In the next section, we present the visualization tools we developed to assist the analysis.

3.7 Supporting Analysis: Data-Mining and GUI

Considering the large amount of data collected in the experiments of previous sections (over 110GB of data collected over 14 cumulative days of runtime), it would be great to also fully automate the data mining, allowing us to analyze the collected results. This is however far beyond the scope of this project.

With METrICS we aim at providing means to perform an expert-driven analysis. In such a context, we developed a GUI providing different ways of visualizing the collected data. In this section, we will present a portfolio of the available visualizations.

3.7.1 Underlying visualization technology

As the purpose of METrICS is to collect the full distribution of events (rather than only minimum / maximum values) to study correlations between timing and shared hardware resource contention, we need the same kind of visualizations as the ones used in the statistics domain.

As we are manipulating gigabyte-large datasets, a major concern for data mining is scalability. Some other important features will be rendering speed and interactivity, to ease the expert analysis.

In the domain of data science and big data, academic research usually relies on Python language coupled with numpy [41] and pandas [29] for data analysis, coupled with matplotlib [21] and jupyter [38] for visualization.

An alternative for online and interactive data visualization is to rely on dedicated javascript libraries such as d3.js [7] or google charts [17]. These libraries render charts as SVG (Scalable Vector Graphics) which enable the user to interact with each element of the chart, typically with zooming or filtering ability.

Data visualization in METrICS involves both charts with millions of points (usually scatterplots) as well as charts with much fewer points (e.g. boxplots) but many filtering options. As a consequence, we opted for two different rendering options: pandas coupled with matplotlib for rendering static large-scale charts, and pandas coupled with d3.js for rendering interactively filterable data. All these visualization are bundled into a single custom Qt-based GUI using the pyside Python binding: **xTRACT visualizer** (expert Timing and Resource Access Counting Trace visualizer).

3.7.2 Visualization related to user probes

User probes are typically used to monitor task runtime and resource usage. Such information allows us to build up classical Gantt charts, effectively showing what is running in parallel, but it does not help to focus on the runtime variation caused by timing interference.

To better visualize runtime variability, we build for each user probe a histogram showing the distribution of observed runtimes during the successive runs, as depicted in Figure 3.8.

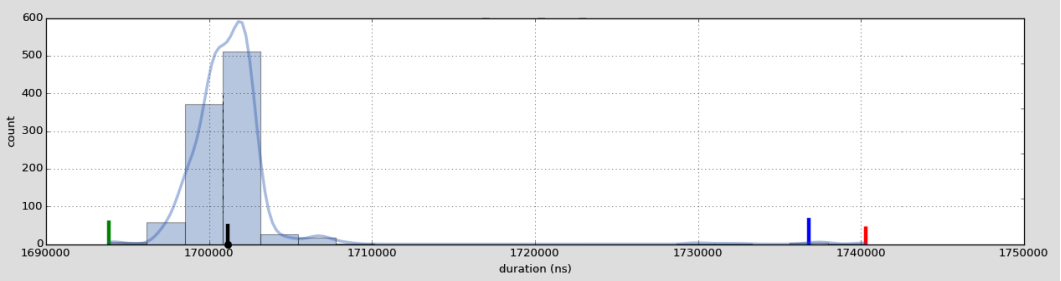


Figure 3.8: Histogram of the drone partition runtime as appearing in xTRACT visualizer

The x-axis corresponds to the observed duration while the y-axis indicates how many times each runtime has been observed. The best (shortest) runtime appears on the left, the worst (longest) observed execution time on the right, the median value being identified with a black dot.

We also added colored vertical bar markers. The green one corresponds to the best case in term of runtime, the red one to the worst case, the black one to the median and the blue one to the first iteration of the application, that frequently behaves quite differently.

To figure out correlations between runtime and hardware resource access, we also build histograms with the collected Performance Monitor Counter data, as shown in Figures 3.9 and 3.10. In these two figures, the colored vertical bar markers still correspond to the best, worst, median and first iteration case with regards to runtime. In these figures, the x-axis corresponds to the number of accesses to a particular hardware resource. In 3.9, accesses seem to somewhat correlate with execution times, whereas it is not the case for 3.10.

Potential correlations could be better observed with scatterplots such as the one appearing in Figure 3.11. Scatterplots allow to easily identify linear correlations. Each point of the scatterplot indicates that a particular run has been observed with a number of resource accesses equal to the value on the x-axis, and an observed runtime equal to the value on the y-axis.

If the points approximate a straight line, there is a linear correlation. In Figure 3.11 most of the

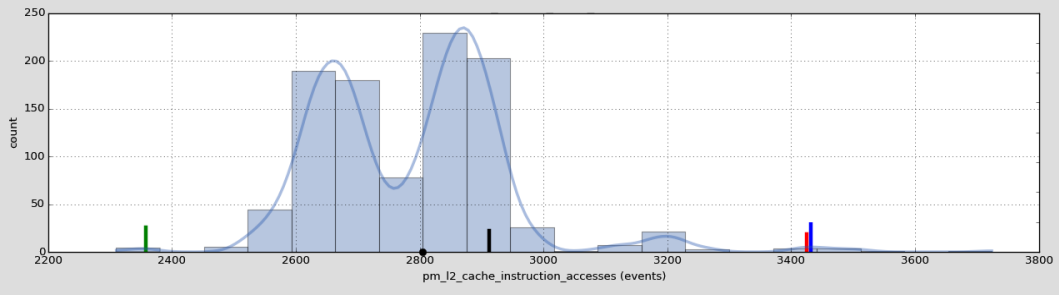


Figure 3.9: Histogram of correlating resource accesses (L2 read cache accesses) as appearing in xTRACT visualizer

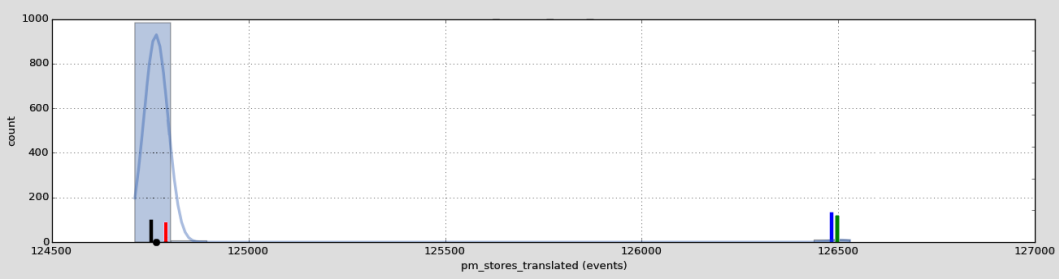


Figure 3.10: Histogram of not correlating resource accesses (issued store instructions) as appearing in xTRACT visualizer (worst and best cases are reversed)

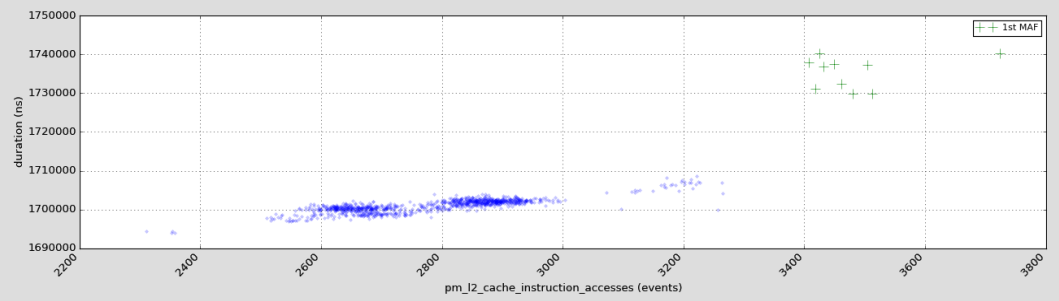


Figure 3.11: Scatterplot showing linear correlation between runtime and L2 read cache accesses as appearing in xTRACT visualizer

points are on a line except the one corresponding to the first iteration that we highlighted with a specific symbol, confirming the correlation.

Another option could be to identify correlation among performance monitor counters to eliminate redundant information provided by correlated hardware resources (like the obvious redundant information of L1 cache misses versus L2 cache accesses) to reduce the experimental design space.

3.7.3 Visualization related to the instrumented syscalls

We also rendered various charts related to the probes automatically inserted around system calls. These renderings allow the expert user to split the runtime into the classical user time (time really spent in the application) and the system time (time spent in the operating system to deal with the application I/O). Alternatively it can be used to observe the usage of kernel locks in system calls.

For instance, the top charts of Figure 3.12 show the distribution of APEX system calls in the ENGINE_R1 task of the eDRON application running on the 6-core Juno board ARM-v8 architecture. The x-axis corresponds to the time, and a gap can be observed between the SEND_BUFFER and the SET_EVENT system calls despite being called sequentially. This is due to the fact that the applica-

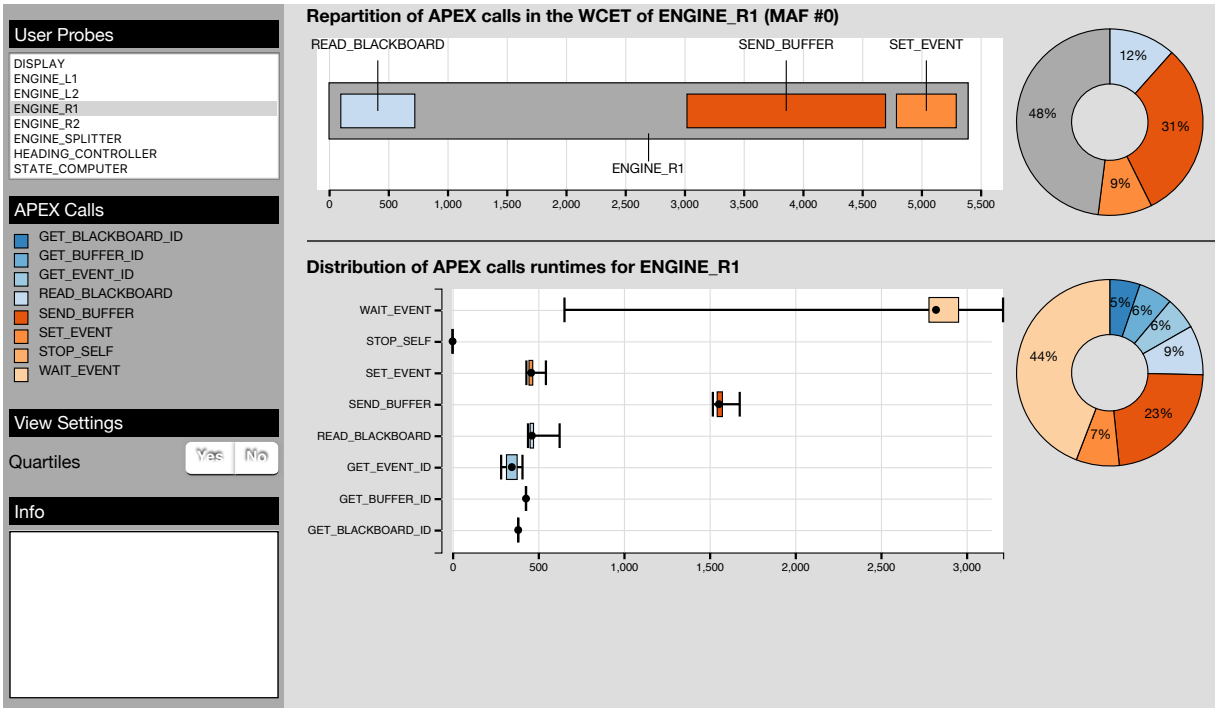


Figure 3.12: Visualizing ARINC-653 syscalls in ENGINE_R1 task with xTRACT visualizer

tion is no more schedulable during system calls, and therefore at least two time-consuming context switches occur between the two function calls.

The bottom part of Figure 3.12 shows with boxplots the variability of the execution time of APEX system calls for different runs of the ENGINE_R1 task, actually showing that the RTOS is also affected by timing interference, while the left navigation bar allows a degree of iteractiveness by proposing to filter out some of the calls.

3.8 Conclusion

In this chapter, we presented and evaluated METrICS, a toolsuite dedicated to perform fine-grain time and resource access measurements in safety critical systems in terms of accuracy, intrusiveness, and ability to measure timing interference.

This toolsuite does not claim to be a solution to deal with the timing interference problems of multi-cores with regards to safety critical systems, but as a reliable and non intrusive way to measure these interference, and therefore a way to evaluate different solutions guaranteeing time integrity.

Beyond the layer running on top of PikeOS on the target system, METrICS is also a host-side server than can automatically drive large-scale test campaigns, and xTRACT visualizer, a GUI that can help the expert to analyze collected results.

Chapter 4 Summary

Contention in hardware shared resources of multicore processors challenges the integration of real-time critical tasks, which need strict guarantees on their WCET. This deliverable provides complementary means to obtain tight bounds (multicore contention model) and to analyze performance so that tasks can be consolidated smartly (METrICS).

The different technologies are assessed on COTS platforms such as the DragonBoard, the Juno board and the AURIX TC27x. Results on benchmarks and avionics prototypes prove the reliability of these technologies and their appropriateness for use on industrial platforms and applications.

In particular, our work shows that the DragonBoard cannot be used for real-time industrial applications unless further documentation is made available. Instead, the Juno board, despite some limitations can be practically used. Finally, the AURIX TC27x automotive platform allows devising tighter contention bounds due to its controllability and, therefore, is a very convenient platform for industrial use.

A detailed toolset, METrICS, has been developed and integrated on top of the Juno board, showing through an avionics prototype that timing interferences can be analyzed. In particular, METrICS shows its ability to reveal the sources of interference, which is a critical information for end users for an efficient consolidation of real-time critical tasks on multicores.

Overall, this deliverable provides both, mechanisms to upper-bound interference on COTS platforms and tools to analyze interference so that consolidation can be optimized, thus achieving lower bounds.

Table 4.1: Summary of integrations on prototypes and use cases.

Technology	**Avionics prototype (Juno)**	**Telecom (DragonBoard)**	**Automotive multicore (AURIX)**	**Automotive networked**
Multicore contention model	YES	NO(*)	YES	N/A
METrICS	YES	NO(*)	NO	N/A

For completeness, we provide the table of technologies integrated in use cases and industrial prototypes. As shown, both technologies are integrated on the avionics prototype. None of them is directly integrated in the telecom use case, which was the original target instead of the avionics prototype. However, an assessment of what could be obtained with the limitations of the DragonBoard platform and a best effort analysis will be performed. The multicore contention model is currently being integrated in the automotive multicore use case, whereas METrICS is not. In fact, the original plan was integrating none of both technologies in this use case. Finally, multicore analysis does not apply to the automotive networked use case.

Chapter 5 List of Abbreviations

AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
CE	Consumer Electronics (market)
COTS	Commercial Off the Shelf
CPA	Cycles Per Access
CRTES	Critical Real-Time Embedded Systems
DSPR	Data ScratchPad RAM
DSU	Debug Support Unit
ECU	Electronic Control Unit
eDRON	embedded Directed Rotodrone Operated Network (application parts of the WP4 avionic prototype)
HPclus/LPClus	High- (2xA72) and Low-Power (4xA53) Clusters
FMS	Flight Management System (application parts of the WP4 avionic prototype)
IP	Intellectual Property
ISA	Instruction Set Architecture
LMU	Local Memory Unit
METrICS	Measurement Environment for Multi-Core Time Critical Systems
PLL	Phase-Locked Loop (clock frequency)
PMC	Performance Monitoring Counters
PMU	Performance Monitoring Unit
PrMU	Program Memory Unit
PSPR	Program ScratchPad RAM
RTOS	Real-Time Operating System
SB	(Resource) Stressing Benchmark
SoC	System on Chip
SRI	Shared Resource Interface
SWaP	Size, Weight and Power
WCET	Worst-Case Execution Time
xTRACT	expert Timing and Resource Access Counting Trace visualizer (GUI)

Bibliography

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, pages 483–485, Atlantic City, April 1967. ACM.

[2] ARM. ARM Cortex-A53 MPCore Processor. Revision: r0p4. Technical Ref. Manual, 2013.

[3] ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade. Technical report, ARM, 2015.

[4] Thomas G. Baker. Lessons learned integrating COTS into systems. In *Proceedings of the First International Conference on COTS-Based Software Systems*, ICCBSS '02, pages 21–30, 2002.

[5] J. Bin et al. Studying co-running avionic real-time applications on multi-core COTS architectures. In *ERTS*^{2}, 2014.

[6] E. Bost. Hardware Support for Robust Partitioning in Freescale QorIQ Multicore SoCs (P4080 and derivatives), White Paper , 2013.

[7] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, December 2011.

[8] D. Dasari et al. Identifying the sources of unpredictability in COTS-based multicore systems. In *SIES*, 2013.

[9] Alan Eustace and Amitabh Srivastava. Atom: A flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 25–25, Berkeley, CA, USA, 1995. USENIX Association.

[10] Jay Fenlason and Richard Stallman. GNU gprof. November 1998.

[11] G. Fernandez et al. Resource usage templates and signatures for COTS multicore processors. In *DAC*, 2015.

[12] G. Fernandez et al. Computing safe contention bounds for multicore resources with round-robin and FIFO arbitration. *IEEE Transactions on Computers*, 66(4):586–600, 2017.

[13] Sylvain Girbal, Guy Durrieu, Madeleine Faugere, Daniel Gracia Perez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. In *ERTS*^{2}, 02 2014.

[14] Sylvain Girbal, Xavier Jean, Jimmy Le Rhun, Daniel Gracia Pérez, and Marc Gatti. Deterministic Platform Software for hard real-time systems using multi-core COTS. In *Proceedings of the 34th Digital Avionics Systems Conference*, DASC'2015, 2015.

[15] Sylvain Girbal, Daniel Gracia Pérez, Jimmy Le Rhun, Madeleine Faugère, Claire Pagetti, and Guy Durrieu. A complete toolchain for an interference-free deployment of avionic applications on multi-core systems. In *Proceedings of the 34th Digital Avionics Systems Conference*, DASC'2015, 2015.

[16] S. Girbal et al. On the convergence of mainstream and mission-critical markets. In *DAC*, 2013.

[17] GOOGLE. Google charts: Interactive charts for browsers and mobile devices. https://developers.google.com/chart/.

[18] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[19] High-Performance Embedded Architecture and Compilation. HiPEAC vision, 2011, 2013, 2015 and 2017.

[20] Hitex. *AURIX Application Kit TC277 TFT*.

[21] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[22] International Electrotechnical Commission. IEC 61508: Functional safety of electrical, electronic, or programmable electronic safety-related systems, 2011.

[23] International Organization for Standardization (ISO). ISO 26262: Road Vehicles Functional Safety, 2011.

[24] J. Jalle et al. Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP). In *ERTS*, 2015.

[25] Raimund Kirner and Peter Puschner. Obstacles in worst-case execution time analysis. In *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 333–339, 2008.

[26] S. Law and I. Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *ECRTS*, 2016.

[27] John Levon. *OProfile - A System Profiler for Linux*. Victoria University of Manchester, 2004.

[28] J. Liedtke et al. OS-controlled cache predictability for real-time systems. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 1997.

[29] Wes McKinney. pandas: a foundational python library for data analysis and statistics. In *PyHPC 2011: Python for High Performance and Scientific Computing*, Seattle, USA, November 2011.

[30] E. Mezzetti and T. Vardanega. On the industrial fitness of wcet analysis. In *Proceedings of the 11th International Workshop on Worst Case Execution Time Analysis (WCET2011)*. 2011.

[31] F. Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 1995.

[32] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Programming Language Design and Implementation Conference*, 2007.

[33] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *EDCC*, 2012.

[34] J. Nowotsch et al. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.

[35] Pine64. Pine64 website, 2016.

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

[37] Renesas. R-Car H3, 2017. https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html.

[38] Kluyver Thomas, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. Jupyter notebooks—a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing*, ELPUB, Göttingen, Germany, 06/2016 2016.

[39] J.W. Tukey. *Exploratory Data Analysis*. Behavioral Science: Quantitative Methods. Addison-Wesley, Reading, Mass., 1977.

[40] P.K. Valsan et al. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS*, 2016.

[41] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: a structure for efficient numerical computation. *CoRR*, abs/1102.1523, 2011.

[42] R. Wilhelm et al. The worst-case execution time problem: overview of methods and survey of tools. *ACM TECS*, 7(3):1–53, 2008.

[43] Ben Wun. Survey of software monitoring and profiling tools.