Multicore shared memory interference analysis through hardware performance counters

Alfonso Mascareñas González, Youcef Bouchebaba, Luca Santinelli

Multicore shared memory interference analysis through hardware performance counters

Alfonso Mascareñas González[1], Youcef Bouchebaba[1], and Luca Santinelli[1,2]

[1]ONERA - DTIS, {alfonso.mascarenas, youcef.bouchebaba, luca.santinelli}@onera.fr
[2]AIRBUS DS Germany, luca.santinelli@airbus.com

Keywords: Multicore, Measurement and statistical timing analysis, Hardware performance counters, Memory interference, Predictability, Extreme Value Theory

I. Introduction

Multicore platforms begin to be used to implement real-time systems within automotive and avionic domains. Although they bring large amount of resources and unexpected increase of performance, guaranteeing their predictability is becoming more and more complex. Multicore systems exhibit numerous behaviors, depending on the runtime environment conditions and interference that can affect task executions; multicore are difficult (or costly) to model as they embed features more oriented on performance than on predictability. Complexity and lack of models make multicore unpredictable or with over pessimistic representations. The industry is especially worried by the use of multicore for critical applications as their products have to comply with certification.

Timing analysis methods compute tasks Worst-Case Execution Time (WCET) as an upper-bound to any possible task execution time [1, 2, 3]. Static timing analysis [1] is one of the timing analysis methods, and it relies on sophisticated control flows and abstract models of the hardware architecture. While satisfactory when used with singlecore platforms, static timing analysis is currently experiencing difficulties with multicores due to the cost of obtaining accurate models. Measurement-Based Timing Analysis (MBTA), is another timing analysis method which estimates WCETs from execution time measurements [2]. MBTA does not need system models, instead it constructs empirical models from measurements of the actual platform behavior. Measurement-Based Probabilistic Timing Analysis (MBPTA) is a special case of MBTA which estimates a probabilistic version of the WCET, called probabilistic WCET (pWCET) [3]; the pWCET is a distribution that is able to upper-bound every possible task execution time, and the MBPTA computes it with the Extreme Value Theory (EVT) applied to measurements of task execution times.

The aim of this paper is to present a high precision and event-versatile MBPTA framework that we have developed for the statistical timing analysis of multicore platforms. Its use satisfactorily allows the study of complex multicore platforms from the CPU point of view, without requiring hardware or software models. This gives us an accurate real view of the platform behavior for any specific situation without using extra tools. In addition, this measurement framework is directly portable to other multicore platforms with the same CPU version and easily portable to other CPU versions within the same manufacturer. The MBPTA framework directly uses coprocessors and the Performance Monitor Unit (PMU), i.e. Performance Monitor Hardware (PMH), instead of software profilers. Hardware performance counters provide low-overhead access to a considerable amount of performance information of numerous elements such as the CPU, caches or bus. The statistical timing analysis consists in proposing average and worst-case modeling by making use of the tool diagXtrm[1] [4] applied to measurement of task execution times. Measurements obtained from the PMH are used for analyzing and quantifying the interference that can happen within a multicore platform. The potential for measurements from coprocessor and PMU, as well as its potential for statistical analysis, is shown by using an heterogeneous multicore Texas Instrument system on chip. The interference we focus on are due to the shared memory of this platform.

II. Platform and applications

A. Heterogeneous multicore platform

We use the heterogeneous multicore Keystone II model TCI6630K2L [5] within the EValuation Module (EVM) TCIEVMK2LX [6]. It is made of 6 single core processors (cores), where 2 of them are ARM15 CPUs architecture version 7 [7], and the other 4 are C66x DSPs [8]. Both types of processors have their own level 1 instruction cache (L1P) and level 1 data cache (L1D) of 32KB size. The two ARM processors (cores) share a level 2 cache (L2) of 1MB size, while each DSP processor has its own L2 cache of 1MB; the DSPs do not share L2 cache. The 6 processors are connected, with the Multicore Shared Memory Controller (MSMC) [9], which is connected to the TeraNet bus together with all the peripherals. Among the peripherals there is the DDR3 external memory of 2GB size which is shared between all the processors.

B. Real-time application

The real-time application used is divided in two kind of tasks: a critical task which is the one under observation and also supposedly the most important part of the application, and non-critical tasks which act as memory stressing sources.
For the sake of convenience, both kinds of tasks operate in the same way: they consist of loops, simple operational calculations and matrices that are the main memory demanding element. The matrices are firstly initiated using the loop counter variable. Later, they are filled again with the results of mathematical operation, where the operands are the matrices data. This refilling process changes and is repeated as function of the level of stress we desire. Algorithm 1 is an example of the simplest kind of critical task being used (in0 and in1 are the matrices, size is the length of the matrix):

[1]diagXtrm is a statistical analysis and MBPTA tool developed at the ONERA https://forge.onera.fr/projects/diagxtrm2.

Algorithm 1 Safety1 task

```
for (int i = 0; i < size; i++)
        for (int j=0; j < size; j++)
                in0[i][j] = i+j+1;
for (int i=0; i < size; i++)
        for (int j=0; j < size; j++)
                in1[i][j] = in0[i-1][j]+in0[i][j-1]+in0[i][j]+in0[i][j+1]+in0[i+1][j];
```

These tasks are not behaving as a realistic real-time application tasks would, i.e. a large number of heterogeneous tasks made up of more varied and complex operations. Instead, they are very handy to create and verify interference from shared memory. In spite of the tasks simplicity, the ease of changing the parameters makes them very flexible as stressing sources.
The critical task has up to three different stress levels. The first of them has been called safety1, indicating that is the least demanding and the last is named safety3, being the most demanding one. The critical task is placed uniquely in one ARM core (critical core); the non-critical tasks are in the other cores (non-critical cores), i.e. the other ARM and the 4 DSPs.
The ARMs are managed using the certified real time operating system PikeOS [10], making use of the integrated development environment (IDE) CODEO [11] for the code development. The PikeOS service task is set in the non-critical core to avoid any possible interference with the critical one. As the DSPs lack of the same operating system, they are programmed in bare-metal using the IDE Code Composer Studio [12].

C. Execution model

The cores execution is always synchronous. This synchrony is what causes to have arbitration schemes for accessing certain resources. A first arbitration decision is taken in the ARM pack, when both cores tries to access the shared L2 cache; a second arbitration decision is taken when the ARM pack and the DSPs try to access the MSMC. The arbitration consists in priority levels and starvation counters. Both of them are by default equal for all the cores, what leads to a fair share of the resources. This makes the non-critical task to moderately interfere with the critical task. If the priorities of the non-critical cores were increased, then a bigger interference takes place, and the opposite if the priorities of these were decreased. Future work will investigate the impact that this priority difference have, using the current implementation as a reference.
The tasks within the cores are executed repeatedly without idle time between the start and the end of the task, i.e. a duty cycle of 100%. This is to provoke constant interference between tasks.

III. Measurements framework

The interference quantification of the critical task is done by using the 15th coprocessor CP15 available in the ARM cores [13]. CP15 works with the PMU [7], allowing the gather of the system statistics on the memory and processor operations. The PMU is made up of 6 general purpose counters plus a specific counter for execution cycles; these counters are in charge of keeping track of how many times an event has taken place. The number of the features or events available is extensive, but in our case there are considered only those in Table I.

Event	Event ID (hex)
Cycles	0x11
Bus Access	0x19
L1D cache refill	0x03
L1D cache access	0x04
Miss-predicted branch	0x10
Level 2 data cache access	0x16
Level 2 data cache refill	0x17

TABLE I: PMU events

Register	Name
PMSELR	Performance Counter Selection Register
PMXEVTYPER	Event Type Select Register
PMCNTENSET	Count Enable Set Register
PMCR	Performance Monitor Control Register
PMOVSR	Overflow Flag Status Register

TABLE II: PMU registers

A. PMU counter setup

To set up the PMU general purpose counters and extract the value from them, it is necessary to configure and access the PMU registers in CP15. This can be done in the following way (alternatively see [14]):

1) *Selection of the counter to use:* in ARM15, a counter among the 6 available is selected with a number between 0 and 5. From now on, the other registers are modified with respect to the selected counter. The register used to select the counter is PMSELR: $\_\_asm\_\_\ \_\_volatile\_\_("mcr\ p15,\ 0,\ \%0,\ c9,\ c12,\ 5" :: "r"((default\_register\_value \& PMSELR\_MASK) | counter\_id))$;
2) *Write into the register the event to keep track of:* the desired event we want to analyze is written using its hexadecimal tag. The register for selecting the event under study e.g. execution time, is PMXEVTYPER: $\_\_asm\_\_\ \_\_volatile\_\_("mcr\ p15,\ 0,\ \%0,\ c9,\ c13,\ 1" :: "r"((default\_register\_value \& PMXEVTYPER\_MASK) | event\_id))$;
3) *Activate the selected counter:* Activates the counter and starts counting. Register PMCNTENSET is for setting the event count: $\_\_asm\_\_\ \_\_volatile\_\_("mcr\ p15,\ 0,\ \%0,\ c9,\ c12,\ 1" :: "r"(((default\_register\_value \& PMCNTENSET\_MASK) | (1 << counter\_id)))$;
4) *Reset the selected counter value:* the 32-bit counter is reset to avoid an overflow and performance counters are enabled in case they weren't by default. We must ensure that the counters are enabled (current step) and activated (Step 3) to start counting; register PMCR is used for resetting the value and status of the counter: $\_\_asm\_\_\ \_\_volatile\_\_("mcr\ p15,\ 0,\ \%0,\ c9,\ c12,\ 0" :: "r"((default\_register\_value \& PMCR\_MASK) | 0x3))$;

5) *Read the selected counter register:* the first counter value is read from the register PMCR and stored:
   $\_\_asm\_\_\ \_\_volatile("MRC\ p15,\ 0,\ \%0,\ c9,\ c13,\ 2" : " = r"(value));$
6) *Execute the critical task:* execute the task under observation;
7) *Read again the selected counter register:* the second counter value is read and the difference with the first can be made. PMCR is the register used:
   $\_\_asm\_\_\ \_\_volatile("MRC\ p15,\ 0,\ \%0,\ c9,\ c13,\ 2" : " = r"(value));$

This procedure [2] can be used straight forward if using only one counter. When retaking a new measurement for the same condition, we can loop from steps 4 to 7. In case the task is computationally heavy, an overflow handling logic should be implemented should be checked (PMOVSR). If more counters are required, then, step 1, 2, 3 and 4 should be repeated as many times as needed while changing the counter number and the event to be analyzed. Each time a register read is required (step 5 and 7), the counter selection must be done before. Table II summarizes the most important registers involved in the previous PMU configuration.
The previous steps follow a start-read pattern. The main drawback of a start-read pattern is the overhead introduced when using more than one counter. This occurs because the already configured counters include the set up of the next ones in their statistics. This issue is relatively more noticeable with tasks less demanding in execution time. The overhead can be reduced by quantifying how much overhead is included and then remove it from the results. However, we can find other configuration patterns like the start-stop that can solve entirely this problem [16]. The start-stop pattern will be studied and compared with the start-read pattern in the future.
Other performance profiling techniques, like the use of simulators or tools that modifies the resultant application for data collection [17], can be seen as an alternative. Nevertheless, these are either slower or intrusive and their results are validity speaking questionable. PMH does not interfere with the system under analysis, assures accuracy of the studied event and quickly collects the data while the tasks are running. Therefore, despite of the PMH own drawbacks, e.g. variations due to pipeline effects, it is preferred over the other techniques. [16, 18].

B. Trace of measurements and statistical analysis

The measured data are exported from the embedded system using the UART module. The data from the different events are then grouped together forming a trace (like in Figures 1a or 5b) whose behavior is statistically analyzed to quantify the variability or predictability, among others. The main event to focus on is the execution time (pWCET study) but the other events, e.g. bus accesses or cache refills, are also analyzed to observe data correlations. In particular, the measurement traces are processed with diagXtrm to compute average models for the trace itself e.g., empirical distribution, mean, maximum, standard deviation. Alternatively to the traces, the histogram representation is used to observe the data distribution and figure out what kind of distribution function fits the best (See Figures 2a or 2d).

diagXtrm applies also the EVT to the input trace of measurements for the worst-case modeling in terms of pWCET estimates. diagXtrm tests EVT applicability with the hypothesis testing state of the art [19, 4]. The EVT hypothesis tests allow to know whether the EVT can be reliably applied or not, but also allow studying patterns and specific behaviors within traces. The EVT hypothesis to fulfill and its respective tests are the following [4]:

- *Stationary*: The measured data set must respect the stationarity, i.e. data is identically distributed and hence belonging to the same probabilistic law. Abrupt measurement changes that causes discontinuities leads to no stationarity. h_1. Check: Kwiatkowski Phillips Schmidt Shin test;
- *Short range independence*: Close in time measurements have no dependence, i.e. local samples independence. Dependency occurs when consecutive measurements have similar values. The decision rule is based on a given distance and on a threshold value. $h_{2,1}$. Check: Brock Dechert Scheinkman test;
- *Long range independence*: Far in time measurements have no dependence, i.e. extremal or far in time independence. Long time dependence are caused by periodic effects. The decision rule impose that for a measurement over a defined threshold, the probability to have other measurement over the same threshold should tend to zero. $h_{2,2}$. Check: Extreme index;
- *Matching*: The measurements belongs to the generalized Pareto distribution or the generalized extreme value distribution maximum domain of attraction. The goodness of fit is used to see how well the distributions fits the observations set. h_3. Check: Cramer von Mises criterion;

The tests output can be seen in form of spider diagrams [4] like in Figure 4a. The spider diagram interpretation is the following: The circles of the diagram express the confidence level (cl) for a specific hypothesis from 0 to 4 (p-values of 0, 0.01, 0.025, 0.005 and 0.1 respectively). Only if the hypothesis cl value is equal or higher than the second inner circle (cl of 1), we can consider the hypothesis as passed. The confidence levels that must comply that cl $\geq$ 1 are cl1,1, cl2,1 or cl2,2 and cl3, which are associated to the stability, short range independence, long range independence and MDA hypothesis respectively. This leaves us with the following expression:

$$h_1\ AND\ (h_{2,1}\ OR\ h_{2,2})\ AND\ h_3.$$

If this expression is met, the color of the diagram turns green else it turns into red (See Figures 4c and 4d). cl1,2 and cl4 are two optional hypothesis. The first supports the stationarity hypothesis and the second indicates if the data ICDF converges, i.e. if we can determine an execution time for a zero probability risk or not. If all of the compulsory hypothesis are satisfied, thus the pWCET obtained with the EVT is a safe worst-case distribution estimate for the condition considered by the measurements. When the pWCET is achieved for a specific runtime execution condition, we refer to it as a relative pWCET, instead of an absolute pWCET,

[2]Note that the present procedure assembly code is specific for ARM CPUs. However, the steps and registers may be similar for other processors, e.g. Intel's Pentium processor with registers Event select control register and Counter configuration control register [15]

since it is able to upper bound execution times from only the condition considered. The same tests apply also to verify the existence of patterns in the traces, thus for verifying the behavior of the system. Note that the pWCETs obtained from diagXtrm are worst-case distributions only for the execution conditions considered for the measurements. Another important data representation figure is the worst-case model representation, e.g. Figure set 3. In here, it is compared the Inverse Cumulative Distribution Function (ICDF) representation of the estimated pWCET with the ICDF of the samples in the trace. This graph has two main strengths. The first one is that you can visually check hypothesis cl3 fulfillment by seeing the level of fitness of both ICDF. The second is that it can be seen the ICDF of the estimated pWCET and hence associate the probability to overpass a specific execution time.

IV. Memory interference

Among the possible interference sources, we study mainly two: the first one is based on ARM pack critical and non-critical cores relationship; the second comes from the critical ARM core and DSPs interaction with the MSMC. The results are presented in four different scenarios.

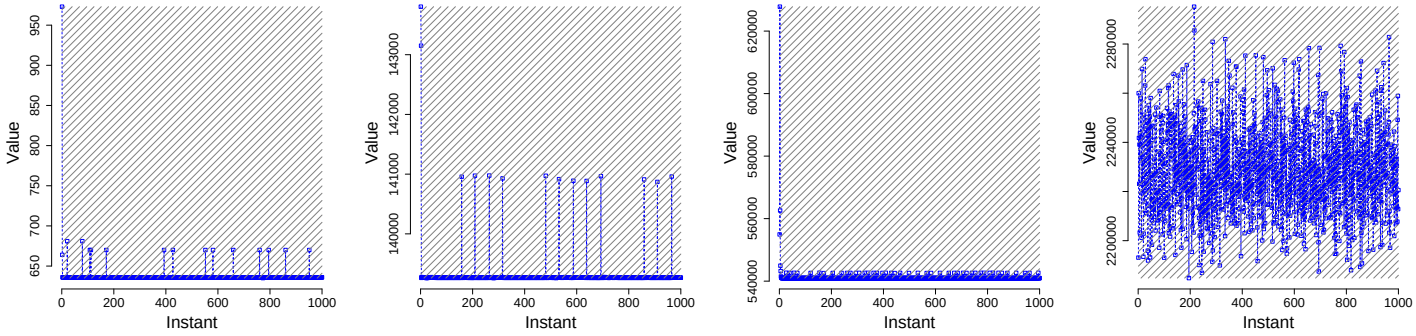
The first one (Scenario1) consists in analyzing only the critical core alone, this means that the rest of cores are in an idle state. The memory usage is gradually increased so that all the memories levels are used and hence quantified. This scenario is the reference for the other three.

The second scenario (Scenario2) consists in the study of the first source of interference, which is the one coming from the second ARM core. This interference is produced when both cores, critical and non-critical, requires the L2 cache and the use of the bus for bringing data back to the first from the DDR3.

The third scenario (Scenario3) consists in analyzing the second source of interference, which is the one produced by the DSPs when using the MSMC for accessing the DDR3. The critical ARM core and the 4 DSPs are active while the second ARM is idle.

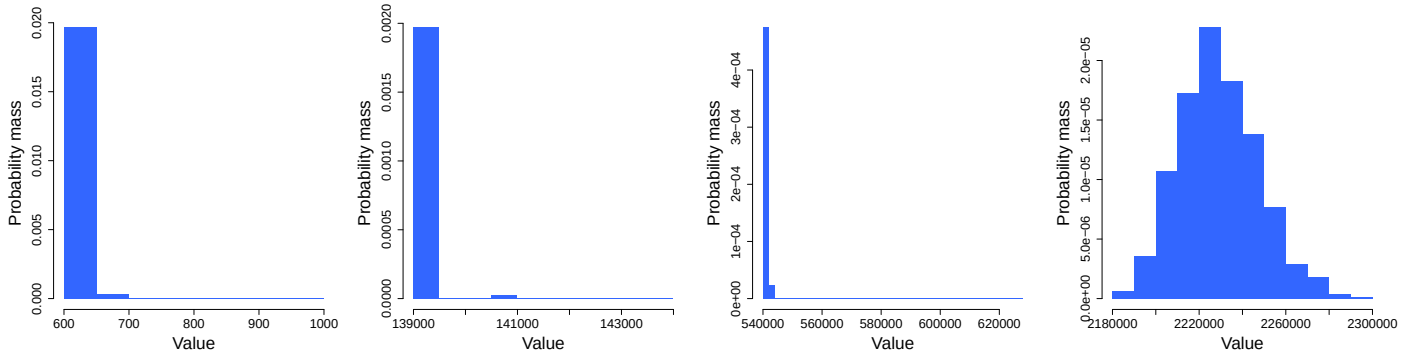
The fourth scenario (Scenario4) consists in merging scenario2 and scenario3 together, i.e. having the critical and non-critical ARMs and the DSPs running together.

The whole data presented here is a small selection of the entire data retrieved during this work[3]; the data shown here have been obtained using only 1 PMU counter for the task execution time event.



(a) Matrices occupation: 512B (b) Matrices occupation: 128KB (c) Matrices occupation: 512KB (d) Matrices occupation: 2MB

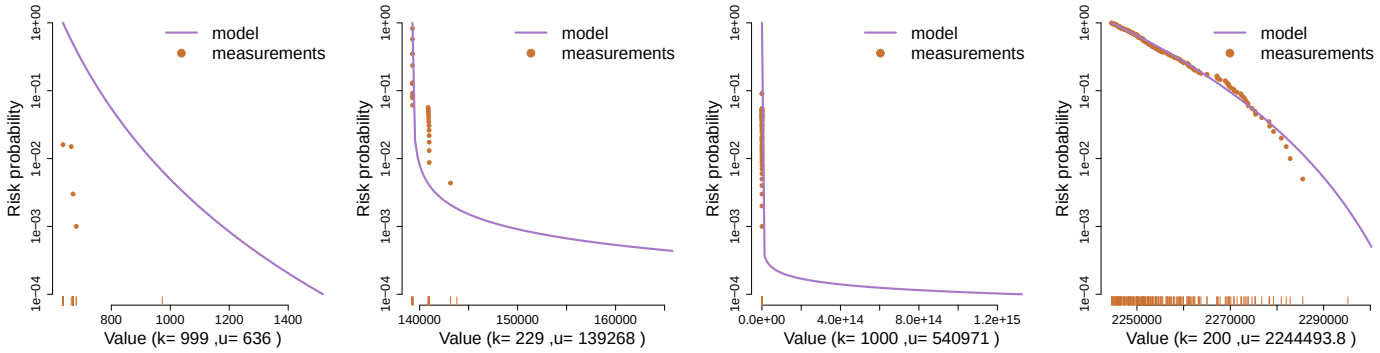
Fig. 1: Isolated critical task safety1 cycle traces results.



(a) Matrices occupation: 512B (b) Matrices occupation: 128KB (c) Matrices occupation: 512KB (d) Matrices occupation: 2MB

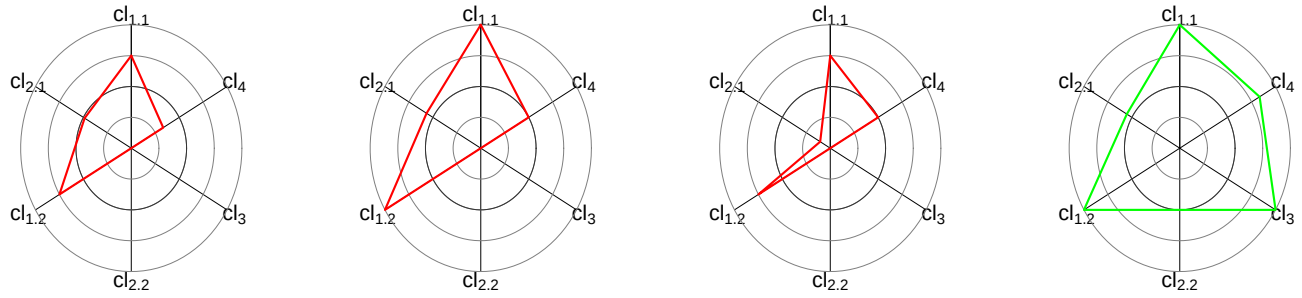
Fig. 2: Isolated critical task safety1 cycle pWCET results represented with histograms.

[3]The data is fully available in the ONERA repositories (https://forge.onera.fr/projects/diagxtrm2)



(a) Matrices occupation: 512B (b) Matrices occupation: 128KB (c) Matrices occupation: 512KB (d) Matrices occupation: 2MB

Fig. 3: Isolated critical task safety1 cycle pWCET results represented with ICDFs.



(a) Matrices occupation: 512B (b) Matrices occupation: 128KB (c) Matrices occupation: 512KB (d) Matrices occupation: 2MB

Fig. 4: Isolated critical task safety1 hypothesis fulfillment.

A. *Critical application*

The analysis of the critical core run in solitary (Scenario1), allows us to validate the effect of the memory levels on the data behavior. This can be seen in Figures 1a, 1b, 1c and 1d, where each level of memory is reached, L1, two times L2 and DDR3 respectively from left to right. The quantity of memory used by the sum of the matrices is indicated for each figure. The Y axis of the sample trace graphs shows the quantity of the event under study e.g. the execution time measured in cycles, while in the X axis the sample number.

The first trace of execution time, Figure 1a, shows several peaks. One large at the beginning (973 cycles) due to the cold start of the processor, and smaller ones (681 and 670 cycles) appearing randomly owing to the branch predictor misprediction. The rest of samples have the same execution cycle value, 636 cycles, coinciding with the mean. Figure 2a is the histogram representation of the measurements trace from which we derive the average modeling of the task behavior.

The behavior of the second trace, Figure 1b, is similar to the previous with small variations on the nominal value. These variations are low compared to the cold start initial spike (143804 cycles); branch misprediction few peaks are still present. The mean is settle to 139294 cycles, very close to the rest of the values measured.

Figure 1c is like Figure 1b (both uses the L2 cache) but with more variability due to the increase of memory occupation (variance of 68375 against 8316715 respectively). The cold start is relatively higher than both, the nominal value (540971 cycles) and the other peaks (542626 cycles) produced by mispredictions and L2 cache maintenance operations. The average value is 541176 cycles.

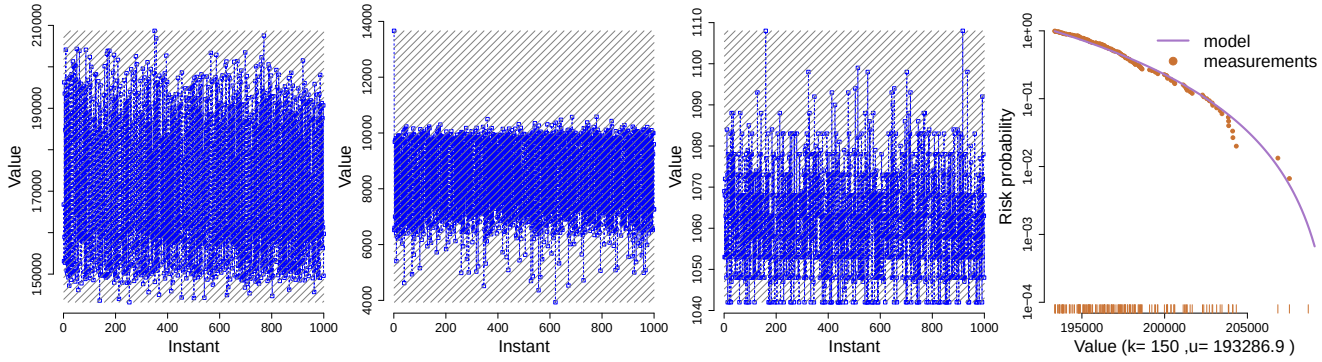
In Figure 1d it is represented the execution condition in which the DDR3 and the bus are exercised. This provokes the large variation on execution time that can be seen. The cold start and the branch misprediction effects are relatively reduced, because the variability due to memory interference from one execution to another is strong. The mean and maximum are 2229078 and 2295192 cycles, respectively.

Figures 3a, 3b, 3c and 3d show the measurement traces pWCETs (worst-case models) in the ICDF form and in a logarithmic scale. The distribution graph in purple is the pWCET distribution obtained with EVT, while the brown spots are the measured data above a certain threshold. In the Y axis the cumulative probability for overcoming the samples, and for the pWCET; the X axis is the value of the sample (execution cycles in these figures). Figures 3a, 3b and 3c show a clear mismatch between the measurements data distribution (See the distribution as histograms in Figures 2a, 2b and 2c) and the pWCET distribution, which means that the EVT fails to comply the MDA hypothesis. Instead, Figure 3d represents a nice fit of the measured data: the MDA is satisfied. This happens because the data follows a distribution similar to a Gaussian (See histogram in Figure 2d). The MDA hypothesis requires that the distribution of the data belongs to the MDA of an EVT resulting distribution, i.e. generalized Pareto distribution or a generalized extreme value distribution.

Figures 4a, 4b, 4c and 4d show the hypothesis fulfillment of the traces. Only for the last case (Figure 4d) the EVT can be applied and its ICDF be reliably used for the worst-case modelling. Therefore, in Figure 3d we find a reliable and accurate worst-case model for the considered conditions.

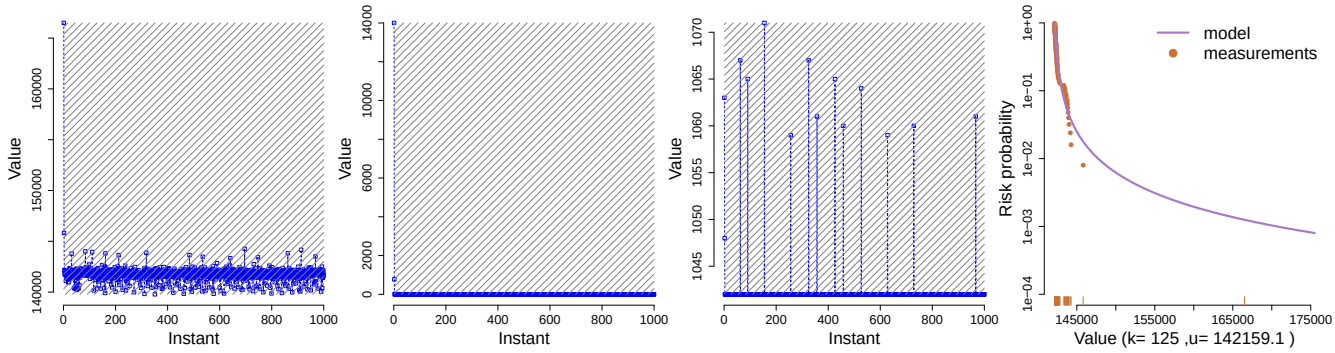
B. ARM critical and non-critical application

The inclusion of the second ARM core (Scenario2), acts as a source of interference for the critical core in the L2 cache and in the DDR3. This happens if the memory used by the non-critical task in the second ARM is sufficiently high to start using the shared memory levels. These interference are produced mainly due to the sharing of the L2 cache stack among the two ARM cores. This leads to have less space available and forces cache evictions of the critical task information: the non-critical task consumes L2 memory of the critical task, and produces evictions which imply future DDR3 accesses that are forced on the critical task. The consequences can be seen in Figures 5, 6 and 7, in which there are represented different measured events and the pWCET model for a specific execution condition. Due to the shareable L2 cache or DDR3 accesses by the two ARM cores, the samples traces exhibit variability from lower memory requirements than Scenario1; the EVT can be applied from lower memory requirements, fulfilling the stability, the short and long range independence and the MDA hypothesis (Figures 5d, 6d and 7d); both average and worst-case models can be reliably and accurately computed.



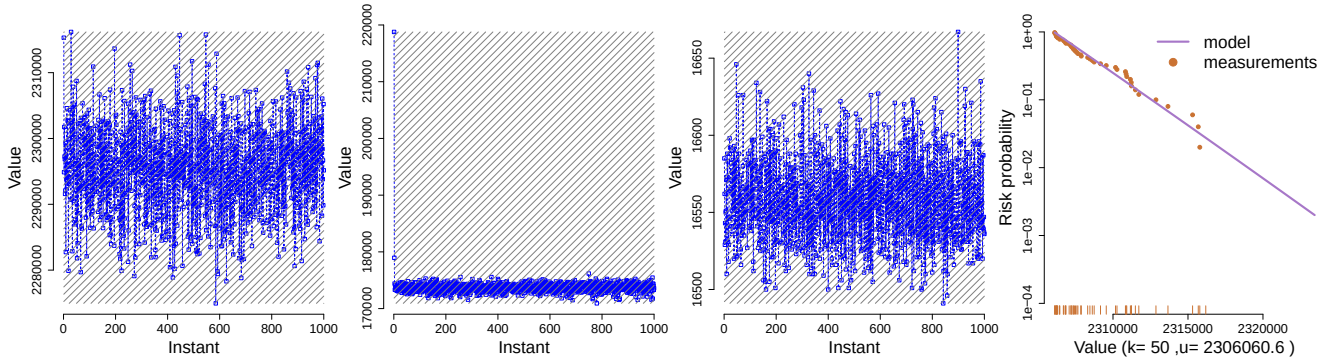
(a) Event: Cycle (b) Event: Bus accesses (c) Event: Cache refill (d) Distribution

Fig. 5: Critical and non-critical ARM tasks safety1 results with matrices size of 128KB and 2MB respectively.



(a) Event: Cycle (b) Event: Bus accesses (c) Event: Cache refill (d) Distribution

Fig. 6: Critical and non-critical ARM tasks safety1 results with matrices size of 128KB and 128KB respectively.



(a) Event: Cycle (b) Event: Bus accesses (c) Event: Cache refill (d) Distribution

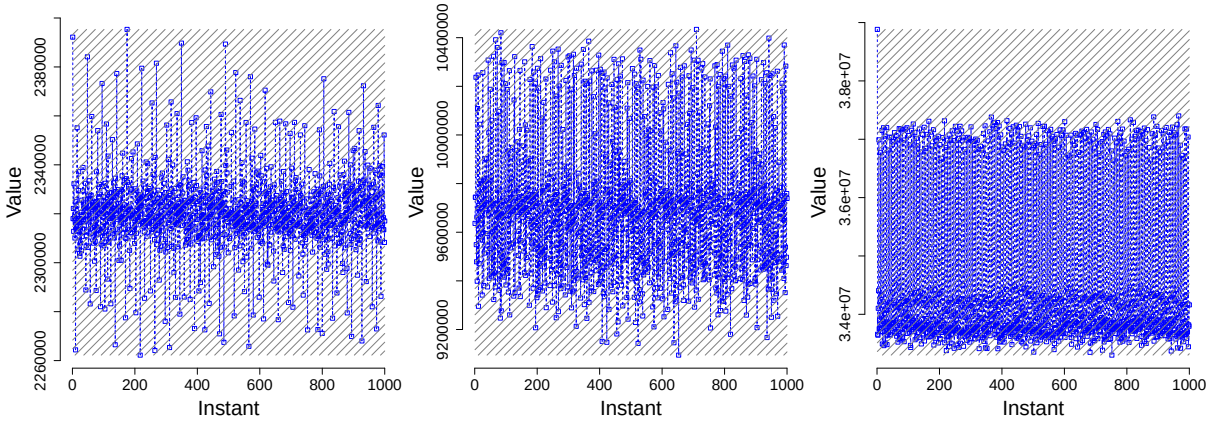
Fig. 7: Critical and non-critical ARM tasks safety1 results with matrices size of 2MB and 128KB respectively.

Figure 5a shows a different behavior than Figure 1b due to the interference of the non-critical task, forcing the use of the DDR3. This leads to a change in variability and in execution time (168864 average cycles). The number of bus accesses can be seen in Figure 5b (8349 average accesses). The L1D cache refilling is seen in Figure 5c (1062 average refills), being almost the same for Figure 1b as the quantity of data to be brought to the critical core L1 cache is the same in both case.

In Figure 6 we can see the case where the L2 cache is used by the critical and non-critical cores but not sufficiently for requiring the DDR3. Hence, the bus accesses are zero except for the cold start (Figure 6b). The variability remains despite being lower than in Figure 5 (variance of 9257042 against 4977282016).

Figure 7 is similar to Figure 5 but with higher cycles, bus accesses and refilling values, with 2295342, 173693, 16557 in average and 2316186, 218773, 16667 as maximum values respectively. This is owing to the higher critical task memory in use (2MB instead 128KB).

The effect of the different stress levels can be seen in the set of figures found in Figure 8. We appreciate that as more demanding the execution is, the more relative maximums we find. The quantity of cycles in Figure 8c (Safety3) is more than an order of magnitude bigger than Figure 8a (Safety1), being 34506605 and 2320258 average cycles respectively.

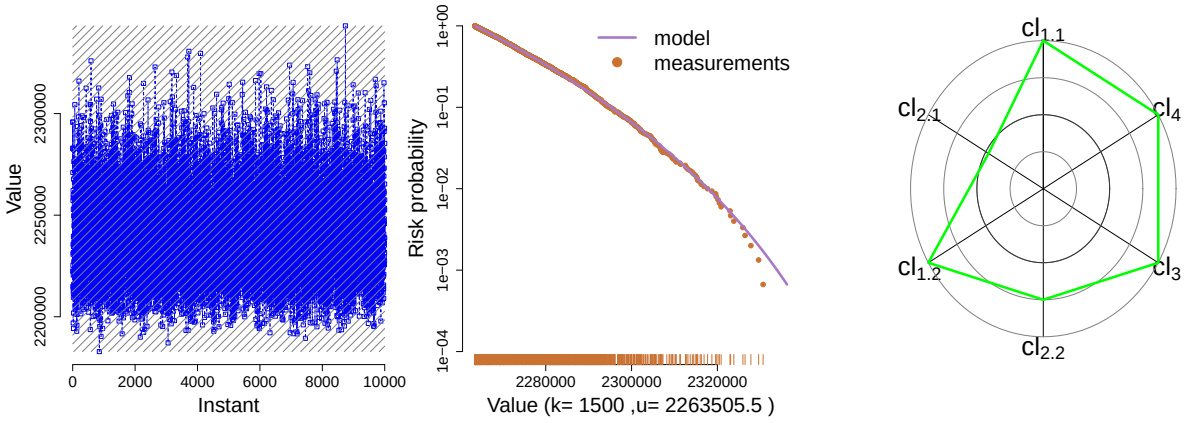


(a) Safety1 (b) Safety2 (c) Safety3

Fig. 8: Critical and non-critical ARM tasks cycles results with matrices size of 2MB and 2MB respectively.

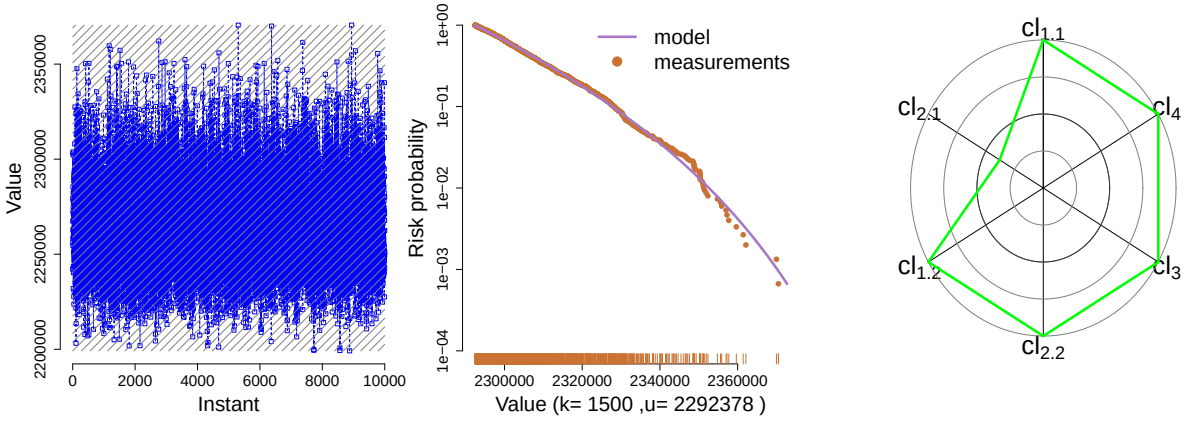
C. ARM critical and DSPs non-critical application

The DSPs act as another source of interference for the critical ARM core, more concretely in the MSMC when trying to access the DDR3. Therefore, the critical and non-critical tasks must fill their correspondent L1 and L2 caches in order to encounter this problem. Figures 9, 10, 11 and 12 corresponds to the critical task output interfered by 1, 2, 3 and 4 DSPs running in parallel respectively. To create a continuous source of interference, the DSPs are not using their L1D and L2 caches and each core pack, ARM and DSP, have the same access priority to the shared resources. Besides, in contrast to Sections IV-A and IV-B, to show the entire behavior of the critical task, the number of samples per runtime condition are 10000 (10 times bigger).



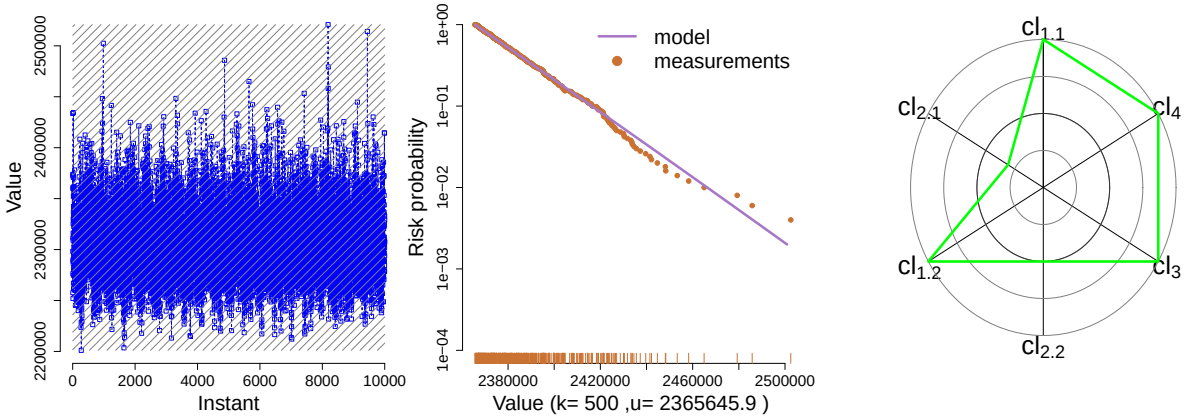
(a) Cycles (b) Distribution (c) Hypothesis fulfillment

Fig. 9: Critical and 1 DSP non-critical tasks safety1 results with matrices size of 2MB and 12MB respectively.



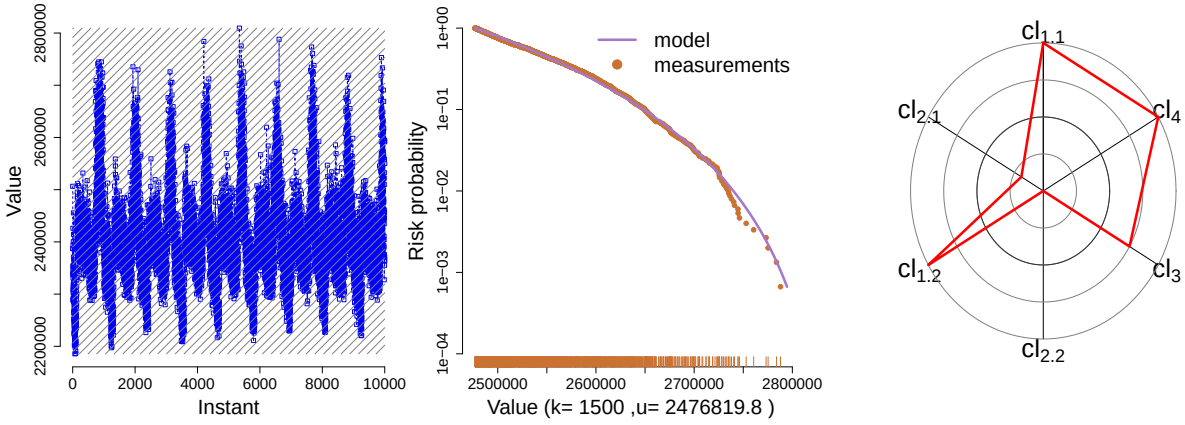
(a) Cycles (b) Distribution (c) Hypothesis fulfillment

Fig. 10: Critical and 2 DSPs non-critical tasks safety1 results with matrices size of 2MB and 12MB respectively.



(a) Cycles (b) Distribution (c) Hypothesis fulfillment

Fig. 11: Critical and 3 DSPs non-critical tasks safety1 results with matrices size of 2MB and 12MB respectively.



(a) Cycles (b) Distribution (c) Hypothesis fulfillment

Fig. 12: Critical and 4 DSPs non-critical tasks safety1 results with matrices size of 2MB and 12MB respectively.

From the cycles traces (Figures 9a, 10a, 11a and 12a) we can see that a DSP addition increases exponentially the execution time, being more notable when using 3 and 4 DSP cores concurrently. The variance also tends to increase following this exponential behavior. The mean, maximum and variance can be numerically seen in Table III, where the 0 DSP case is included as the reference.

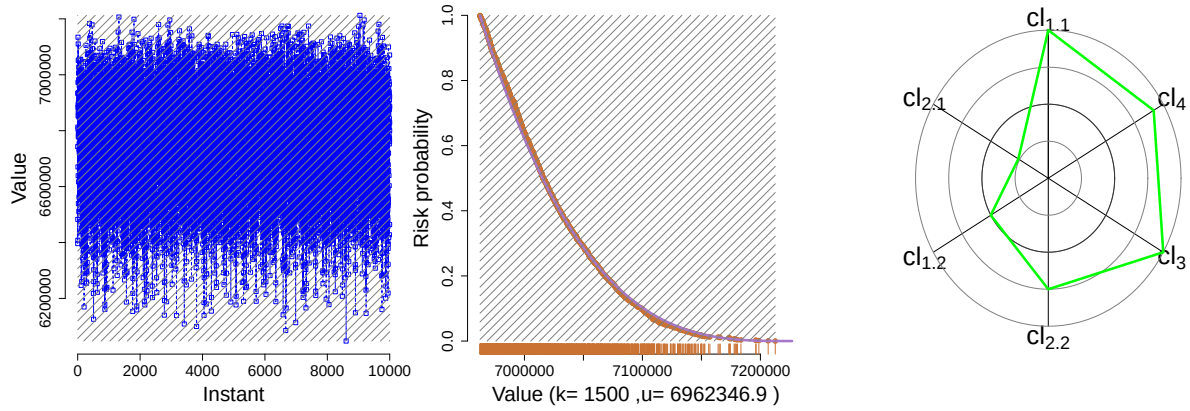
DSPs	Mean (cycles)	Maximum (cycles)	Variance (cycles)
0	2224420	2356814	343967865
1	2241536	2343183	465614203
2	2266079	2370502	629619565
3	2306344	2520805	1177881735
4	2400577	2809433	7534655301

TABLE III: Critical and DSPs non-critical tasks statistics

Figures 9b, 10b, 11b and 12b indicate that the data and pWCET ICDF fit is optimal for all of the cases. However, the spider diagram indicates whether the estimated pWCET can be used or not. The cases with 1, 2 and 3 DSPs fulfill the EVT hypothesis (Figures 9c, 10c and 11c) but not when using 4 DSPs (Figure 12c). The latest fails in both of the independence hypothesis, short and long range. These two hypothesis levels of confidence decrease as new DSP cores are being added (See the spider diagrams). These DSP core additions also change the measurement trace pattern. This is clearly visible in Figure 12a when the 4 DSPs are concurrently working. In this plot we appreciate a pattern with distinguishable relative minimums and maximums sections, e.g. from measurement 0 to 130 and from measurement 780 to 850 respectively. The minimums represent the situation where the critical core has very small interference. It behaves as if there were no DSPs or only if 1 or 2 DSPs were being executed (relative minimum values do not overpass the maximum value of DSPs 0, 1 and 2 shown in Table III). The maximums behaves as if 3 or 4 DSPs were intensively interfering the critical task from loading data from the DDR.

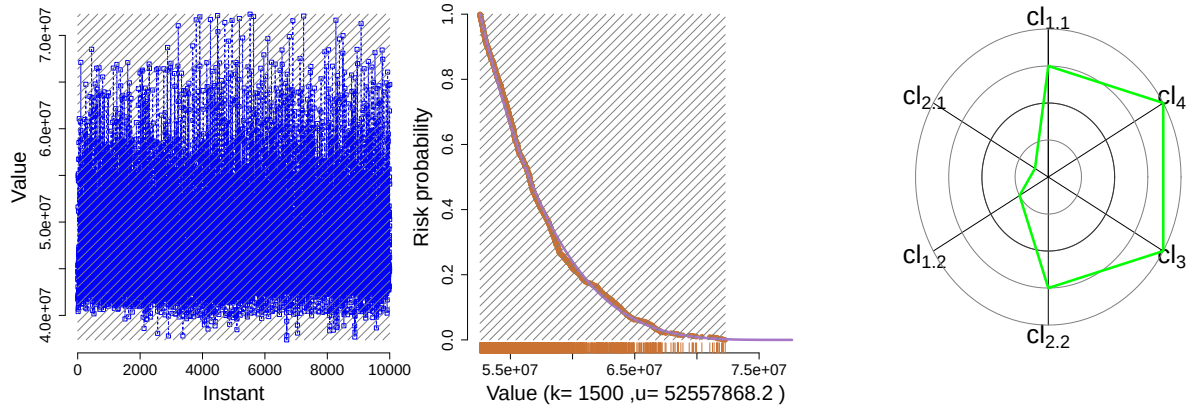
D. ARM critical and non-critical and DSPs non-critical application

Scenario4 has all the cores of the platform activated and running. Therefore, the critical task will be interfered (1) in the L2 cache by the non-critical ARM and (2) in the DDR by the non-critical ARM core and the DSPs. The results found in Figures 13 and 14 can be used to determine whether or not the pWCET model can be applied in a complete case or not. As it can be seen, both cases fulfill the EVT hypothesis and hence allowing us to apply the probabilistic model for each case. The results can be found counter-intuitively as Figure 12a in Scenario3 did not pass the EVT tests. In here, the non-critical ARM removes the dependency (visible patterns) caused by the 4 DSPs by adding variability.



(a) Cycles (b) Distribution (c) Hypothesis fulfillment

Fig. 13: ARM critical and non-critical and 4 DSPs non-critical tasks safety3 results with matrices size of 512KB, 2MB and 12MB respectively.



(a) Cycles (b) Distribution (c) Hypothesis fulfillment

Fig. 14: ARM critical and non-critical and 4 DSPs non-critical tasks safety3 results with matrices size of 2MB, 2MB and 12MB respectively.

Note that the stress level 3 (safety3) is used. By using Figure 8c as a reference (Scenario2, stress level3, 2MB for both ARM cores), we see that the overhead caused by introducing the 4 DSPs is around 36.76%. Recall that no data caches are being used for the DSPs and that all cores have an equal shared resource access priority.

V. Conclusions and future work

In this work we show the feasibility and the potential of using the processors PMH for embedded systems timing analysis purposes. We propose a low-overhead measurement framework and a statistical analysis for average and worst-case modeling of task execution times under different execution conditions. The statistical analysis is applied on measured execution time, and accurate models are computed. This framework is used to investigate interference that tasks receive from shared memory in multicore real-time embedded systems.

Future work will extend the timing analysis framework to other system parameters and other system interference sources e.g., communication buses or partitioning tools. Also, the statistical analysis will be integrated with other metrics to increase the quality of the representations obtained. The analysis will also apply to realistic real-time applications and other possible industrial multicore platforms.

References

[1] Reinhard Wilhelm et al. "The worst-case execution-time problem - overview of methods and survey of tools." In: *ACM Trans. Embedded Comput. Syst.* 7 (Jan. 2008).

[2] Stephen Law and Iain Bate. "Achieving Appropriate Test Coverage for Reliable Measurement-Based Timing Analysis". In: *28th Euromicro Conference on Real-Time Systems, ECRTS*. 2016, pp. 189–199.

[3] Francisco J. Cazorla et al. "PROARTIS: Probabilistically Analysable Real-Time Systems". In: *ACM Transactions on Embedded Computing Systems (TECS)*. Vol. 12. 2. 2012, pp. 1–26.

[4] F. Guet, L. Santinelli, and J. Morio. "On the Reliability of the Probabilistic Worst-Case Execution Time Estimates". In: *8th European Congress on Embedded Real Time Software and Systems (ERTS)*. 2016.

[5] Texas Instruments. *TCI6630K2L Multicore DSP+ARM KeyStone II System-on-Chip (SoC)*. Jan. 2015.

[6] Einfochips. *Keystone2 EVM Technical Reference Manual*. Dec. 2014.

[7] ARM. *Cortex-A15 Revision: r2p1. Technical Reference Manual*. Dec. 2011.

[8] Texas Instruments. *TMS320C66x DSP CorePac*. July 2013.

[9] Texas Instruments. *KeyStone II Architecture Multicore Shared Memory Controller (MSMC)*. Nov. 2012.

[10] SYSGO. *PikeOS Hypervisor certified according to Common Criteria*. URL: https://www.sysgo.com/news-events/news-articles/article/pikeos-hypervisor-certified-according-to-common-criteria/.

[11] SYSGO. *PikeOS Hypervisor Eclipse-based CODEO*. URL: https://www.sysgo.com/products/pikeos-hypervisor/eclipse-based-codeo/.

[12] Texas Instruments. *Code Composer Studio (CCS) Integrated Development Environment (IDE) CCSTUDIO (ACTIVE)*. URL: http://www.ti.com/tool/CCSTUDIO.

[13] ARM. *About the coprocessor interface*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0333h/Bgbhbiah.html.

[14] ARM. *Using the PMU Event Counters in DS-5*. URL: https://developer.arm.com/tools-and-software/embedded/legacy-tools/ds-5-development-studio/resources/tutorials/using-the-pmu-event-counters-in-ds-5.

[15] B. Sprunt. "Pentium 4 performance-monitoring features". In: *IEEE Micro* 22.4 (July 2002), pp. 72–82.

[16] D. Zaparanuks, M. Jovic, and M. Hauswirth. "Accuracy of performance counter measurements". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. Apr. 2009, pp. 23–32.

[17] B. Sprunt. "The basics of performance-monitoring hardware". In: *IEEE Micro* 22.4 (July 2002), pp. 64–71.

[18] Vincent M Weaver and Sally McKee. "Can hardware performance counters be trusted?" In: Oct. 2008, pp. 141–150.

[19] L. Santinelli, F. Guet, and J. Morio. "Revising Measurement-Based Probabilistic Timing Analysis". In: *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2017, pp. 199–208.