Performance Monitoring Features in EPOS

Leonardo Passig Horstmann, José Luís Conradi Hoffmann, and Antônio Augusto Fröhlich
LISHA / PPGCC / UFSC
Florianópolis, SC - Brasil
{horstmann,hoffmann,guto}@lisha.ufsc.br

***Abstract*—Incorporating performance monitoring capabilities to embedded environments, especially on critical systems such as Cyber-Physical Systems, requires a negligible intrusion to ensure the CPS environment performance and data quality. The monitoring design must be tailored to fit the system's needs instead of being limited to a single monitoring approach. In this paper, we extend a monitoring framework to encompass three monitoring approaches: Periodic-, Execution-flow-, and Job-based, focusing on the evaluation of overhead, latency, and jitter for each of them. We have implemented and evaluated these approaches over the Monitoring Framework design, where none of them presented an average impact on the system execution time higher than 0.3%. While the Job-based monitoring presented better results in terms of impact over the tasks execution and memory consumption, the Execution-flow-based monitoring presented better results for jitter, both on the impact over task execution time and the monitoring latency.**

***Index Terms*—Performance Monitoring, Monitoring Framework, Time-triggered, Execution-flow-based, Job-based.**

I. Introduction

Embedded systems, such as those handling computer vision in autonomous vehicles, impose an increasing performance demand over modern Cyber-Physical Systems (CPS) platforms, which must include complex architectural features to cope with such requirements (e.g., heterogeneous cores, Single Instruction Multiple Data (SIMD) units, and task-specific accelerators interconnected by Network on Chip (NoC) technology) [1]. These increasing performance demands often include the inherent timing constraints of real-time systems and management of limited resources, such as memory, computing capabilities, and power consumption.

These platforms, however, are highly instrumented cyber-physical systems that can be monitored and controlled based on the data they produce during operation [2], [3]. The process of learning and actuating over a system behavior can be implemented to improve the system usage based on monitoring the system performance and the building of Machine Learning models to actuate on behalf of improvement of performance and energy consumption [4]–[6], or detect anomalies [7]. Nevertheless, to build reliable models of the system execution, it is fundamental that the instrumentation used for monitoring it does not disrupt the system behavior. In other words, the non-intrusiveness of the monitoring process dictates whether the Machine Learning models are accurate and the actuation is reliable.

In previous work, authors introduced a non-intrusive monitoring system capable of sampling data from sensors, performance counters, and Operating System (OS) without disrupting system constraints [8]. The system is able to sample selected variables in real-time without interfering with the execution of critical tasks in terms of deadline observance, jitter, and latency. The small overhead is constant and limited to specific OS-interaction points and can be modeled as an additional processing demand to that of tasks to ensure the system as a whole is still schedulable.

In this paper, we evaluate three different monitoring approaches. The first one is Periodic monitoring, which samples according to periodic interruptions. The second one is Execution-flow-based monitoring, the default configuration of the monitoring framework, which samples at specific OS procedures following a pre-defined sampling rate. The third one is Job-based monitoring, an event-driven solution that samples whenever a context-switch is performed, maintaining the sampling process bounded by the task-set configuration. To perform this evaluation, we extend the aforementioned framework to introduce the ability to sample data according to each of one the approaches. The evaluation focuses on providing a comparison in terms of overhead, latency, jitter, and memory consumption, which are typical metrics to evaluate interference in CPS platforms. In this way, the main contributions of this work are:

- A baseline of the performance of the monitoring system under different monitoring approaches.
- A comparison between the different monitoring approaches in terms of performance, namely, overhead, latency, jitter, and memory consumption.

The remaining of this paper is organized as follows: Section II presents the related works. Section III describes the monitoring framework architecture and the implementation of each of the monitoring approaches. Section IV describes our Case-Study scenario and the evaluation metrics. Section V presents the obtained results. Section VI presents reasoning on the major advantages and shortcomings of each of the monitoring approaches. Finally, Section VII presents our final considerations and concludes this paper.

II. Related Works

Fischmeister et al. [9] present a sampling-based monitoring solution over an instrumentation framework that extends the sampling period and reduces monitoring overhead. In their approach, the instrumentation is based on control-flow graphs. Following the control-flow graph-based instrumentation, Wu et al. [10] propose a hybrid monitoring by combining event- and time-triggered monitoring to support Runtime Verification

techniques. The main goal of the hybrid approach is to avoid redundant sampling of time-triggered monitoring that does not capture any new critical event in the system while avoiding a frequent monitor activity on periods with a higher rate of critical events occurrences. This is done through a mode-switching optimization heuristic. The optimization heuristic is built on the control-flow graph and the associated cost of the monitoring operations (event monitoring, periodic monitoring, and mode-switch). Our work also evaluates time-triggered monitoring, but we differ by evaluating two different monitoring approaches, the execution-flow- and job-based.

Woralert et al. [11] presents and measures the overhead of a periodic sampling mechanism (K-LEB, Kernel - Lineage of Event Behavior). They show that by operating as a kernel module the proposed mechanism is able to monitor at high frequencies and with a lower overhead when compared with user-level monitoring tools. In this paper, each of the monitoring approaches are integrated directly to the operating system that mediates the CPS platform. Therefore, we also take advantage of the highest-resolution timer available. Moreover, in this work we measure the impacts of three different monitoring approaches and compare their overhead in terms of time and memory.

Leng et al. [12] applied the monitoring of the performance counters to extract fingerprints of the system execution. They selected a set of counters composed of the number of *committed instructions*, the *number of function calls*, the *number of integer instructions*, and the *number of load instructions*. They executed fault-free versions of the chosen tasks to obtain the initial execution profiles using the selected counters. They traced the chosen performance counters values following a sample interval of $10ms$, which is equivalent to a 100Hz rate. They tried two different monitoring methods, periodic sampling, and per-task-like sampling. They concluded that when performing per-task-like monitoring, they can detect abnormalities on system execution but lose the ability to determine when such abnormal behavior happens.

Aliabadi et al. [13] proposes a monitoring solution for Intrusion Detection in Cyber-Physical Systems addressing low overhead for both memory and time aspects. In their solution, the monitoring approach perform captures at a small set of locations or features defined during design-time. They reduce the overhead and memory by implementing a feature selection technique regarding attacks coverage. The selection is performed by evaluating the accuracy of a Bayesian Network over the detection of injected faults using inferences of full coverage of detection given a set of partial information. Thus, their solution follows a execution-flow based monitoring while addressing low overhead by reducing the amount of features necessary to solve the problem under investigation.

Run-DMC [6] is a runtime dynamic performance and power estimator for Heterogeneous Multicore platforms. Run-DMC is designed for non-real-time scenarios and is implemented over a Linux kernel. The method aims at thread-level prediction for both Instructions per Cycle (IPC) and Dynamic Power (DP) based on Least Square Method over performance counters. In their proposed solution, performance counters are sampled on a task basis. They sample at the granularity of tasks context-switch, where the information is fetched and summed up at thread-specific accumulators. In the worst-case scenario evaluated for Run-DMC [6], incurred on a maximum sampling overhead of $7\mu s$, $44\mu s$, and $70\mu s$, on a 4, 8, and 16 thread scenarios, and up to $869\mu s$ per actuation (sensing, estimation and prediction, optimization, and thread mapping). The same approach is also used in Donyanavard et al. [5] work, named SPARTA, a runtime, throughput-aware, energy-efficient task allocation system integrated into the Linux scheduler for heterogeneous scenarios, which presented an average sampling overhead of $28\mu s$ on all cores.

In [14], the authors propose a reinforcement learning solution to optimize thermal management in Embedded Systems. In this scenario, PMU counters and Temperature sensors are sampled periodically on a CPU-based sampling that encompasses a hyper-period. They demonstrate that the proposed reinforcement learning resulted in low overhead, requiring $2.5\mu s$ to $8\mu s$ to collect data at each period. However, they limit their analysis to CPU-based sampling in a long-term interval, which is unsuitable in some scenarios, like anomaly detection, where a fine-grain sampling is required.

Merkel et al. [15] propose a co-scheduling approach considering shared resources contention, implemented over Linux. They base their solution on monitoring Memory Bus Access, Level 2 Cache Access, and Committed Instructions rate to build an activity vector for each task. The activity vectors are updated every timer interrupt and every task switch, where the authors measured the cost for reading a performance monitoring counter to be 54 cycles on the chosen platform and did not observe any runtime increase into the execution time of the benchmarks.

III. Architecture

The analysis presented in this paper builds upon a non-intrusive monitoring system introduced by the authors in a previous work [8]. In this section, we recapitulate the main elements in that design and extend the framework to include Periodic and Job-based sampling.

A. Monitoring System

The Monitoring Framework [8] under investigation in this paper is part of a scheduling framework that supports the design of domain-specific, low-overhead resource schedulers for critical systems. The main idea for the modeling of the monitoring framework is to provide the user with a simple and lean mechanism to control the sampling of performance data at runtime while abstracting the different characteristics of the myriad of data sources (e.g., OS variables, hardware counters, and sensors), without disrupting the timing behavior of the system being monitored.

The monitoring framework is modeled around two constructs, an interface that abstracts the different data sources and a monitoring manager, namely, Clerk and Monitor. In this sense, the Clerk provides a common interface to configure the data source and acquire data. The Monitor collects data using the Clerks and automatically timestamps them with the highest-resolution timer available on the platform. The

specified Clerks and monitoring policy are used to instrument the resulting scheduler at compile-time with data sampling operations at pre-defined points of interest. Points of interest are defined by the framework for all the traditional scheduling operations, including allocation, mapping, dispatching, and accounting.

As part of a framework that supports low-overhead resource schedulers for critical systems, the Monitoring Framework addresses non-intrusiveness with a set of design decisions and their implications. The data structures that are used to store sampled data have fixed sizes and are defined at compile-time. The Clerks and data structures are initialized during the OS boot time, i.e., prior to the execution of the tasks. The code is generated with templates and all data captures execute the same procedure, avoiding incurring jitter. The buffer indexing is direct and the operation to check conditions to perform the data sampling are constant, which reduces jitter to a matter of capturing data or not. Moreover, the data sampled during execution is processed after the system execution or during the system's free time.

Finally, as previously mentioned, the Monitoring Framework is part of a scheduling framework [8]. In this scheduling framework, a scheduling policy is implemented through a parameterized class that models traditional scheduling operations interacting with the scheduling queues [16] that have their ordering criteria given as the specialization of a generic criterion that provides the ranking algorithm. The integration of the Monitoring Framework to this scheduler design allows one to use the data collected by the Monitor to learn patterns from the performance traces that are sampled during the very-own system execution and model a scheduling criterion that uses Clerks and Monitors to actuate during the system execution to control platform configuration, task distribution, and other execution aspects.

B. Execution-flow-based Monitoring

This approach consists of taking advantage of the interactions of the OS on the execution by using pre-defined points of interest (see Section III-A) to perform data acquisition. In Execution-flow-based monitoring, data captures are triggered whenever the system execution reaches any point of interest. However, data is only acquired when the specified sampling rate is respected (i.e., the time spent between two data acquisitions is higher than or equal to the period specified to the Monitor).

C. Periodic Monitoring

Periodic monitoring consists of periodically interrupting the system execution to perform data acquisition. In this case, the point of interest to be instrumented is the function that handles this specific timer interruption designated to perform the data sampling. In this approach, the sampling rate defines the timer period and no extra condition needs to be checked when data capture is triggered (e.g., the sampling rate verification used in the Execution-flow-based monitoring).

Periodic monitoring is implemented in the Monitoring Framework by adding a periodic timer that triggers data acquisition instead of relying on reaching specific points of interest to capture the current system behavior. The periodic timer can be implemented either at the Clerk or the Monitor. At the Clerk, it is possible to customize different periods to each Clerk currently monitored. However, adding a specific timer to each Clerk does not scale for dissonant periods, as it may lead to trashing the system performance due to a higher interruption rate. On the other hand, adding a timer at the Monitor issues a single interruption that samples all the Clerks.

For the experiments conducted in this paper, Periodic monitoring is implemented by adding a timer to the Monitor. The timer is configured at boot time according to the specified sampling rate, and the handler for this timer is configured to trigger the data acquisitions. The customization of each Clerk's sampling rate is still possible at this level. This is done by setting the timer periodicity to the greatest common divisor of all Clerks' sampling rates. However, the periodicity precision is negatively affected when the sampling rates are dissonant (e.g., prime numbers), and should be avoided.

D. Job-based Monitoring

This monitoring approach consists of sampling data according to the execution of the jobs into the system. In this way, a sample is collected whenever a job ends its execution, and the sample is linked to the task whose job's behavior it represents.

This approach is implemented by declaring separated buffers for each of the tasks and configure the moment a context switch is going to be performed as the point of interest to be instrumented. In this way, every reschedule incurs in data acquisition and no condition is checked for the sampling rate. When a job is scheduled, the behavior of the job previously executing is sampled and the measures are accumulated in the temporary buffers until the job finishes its execution, which triggers the storage of the temporary values on the Monitor buffers. Algorithm 1 depicts the Job-based monitoring implementation.

Algorithm 1 Job-based Data Acquisition Implementation

```
1: procedure collect ( )
2:     for each c ∈ Monitor::Clerks do
3:         prev.data[c] += c.read()
4:                         − prev.data_accum[c]
5:         next.data_accum[c] = c.read()
6:     if Sampling_Trigger then
7:         Monitor :: capture()
```

IV. Case Study

In order to evaluate the effectiveness of each of the monitoring approaches implemented on the Monitoring Framework, we executed a case study using a representative task-set composed of memory-bound, CPU-bound, and mid-term tasks over an embedded multicore platform.

A. Platform Setup

The selected platform to evaluate the approaches is a Cortex-A53 processor, a widely used quad-core processor for

embedded systems. The Cortex-A53 has four homogeneous cores with frequency ranging from 0.6-1.2GHz, an 8-stage pipeline with two issued instructions per cycle, a coherent Level 1 (L1) private Cache of 32KB (16KB for Instructions and 16KB for Data), and a shared coherent L2 cache of 512KB. For the experiments conducted on this paper, the CPUs were set to the frequency of 0.6GHz.

The platform is mediated by the Embedded Parallel Operating System (EPOS - https://epos.lisha.ufsc.br/), which implements the monitoring framework presented in Section III-A. From hardware performance counters, Cortex-A53 has support to ARM Performance Monitoring Unit (PMU)v3 architecture [17], which provides 54 PMU events (e.g., Cycle Count, Committed Instructions, Branches, and Cache access). A full list of the available counters can be found at Cortex-A53 Technical Reference Manual [17]. Moreover, the PMU of Cortex-A53 enables up to six performance counters to be simultaneously monitored.

B. Task-set Setup

The experiments conducted in this paper use a synthetic task-set with distinct performance behavior, enabling a representative scenario for multicore real-time embedded systems. Moreover, the task-set also includes scenarios with concurrent architectural usage (i.e., Level 2 caches). In this way, we can evaluate the overhead of the monitoring system on a broader range of tasks. The following tasks composed the case-study task-set:

- **Bandwidth** is a benchmark implementation based on [18]. Bandwidth focuses on memory stressing and is tailored to constantly perform read and write operations in a data structure with at least the size of L2 Cache ($512KB$ in our platform).
- **Disparity map** is a task from San-Diego Visual Benchmark Studio [19], representing a real workload task of embedded systems. Disparity Map is widely used for embedded vision applications in autonomous vehicles, like cruise control, pedestrian tracking, and collision control.
- **CPU Hungry** is a loop function executing mathematical operations using Arithmetic Logical Unit (ALY). Our implementation is based on iterative Fibonacci.

This synthetic task-set is based on the one used in [4] to depict a representative set of tasks for multicore real-time embedded systems, in which the behavior of the task is affected by shared resource contention due to the parallelism of Bandwidth and Disparity map tasks.

The underlying OS was set to schedule tasks following Partitioned scheduling with Earliest Deadline First (EDF) criterion to define tasks priority. Additionally, the scheduler was configured to run with a periodic scheduling verification every $10ms$. The task-set configuration is depicted in Table I, where, in this scenario, tasks have their Deadline equal to their Period. Moreover, we have considered CPU0 as the CPU that handles all the system interrupts on the selected OS, possibly affecting the performance measurements of each of the monitoring approaches due to the extraneous interruptions. Thus, CPU0 measurements were discarded, and only CPUs 1 to 3 run the task-set.

TABLE I
TASK-SETS CONFIGURATION.

CPU	Period/WCET	Task	Parallel to
1	$500ms/100ms$	T1 Bandwidth	T3, T5
	$500ms/200ms$	T2 Disparity	T4, T6
2	$500ms/200ms$	T3 Disparity	T1, T5
	$500ms/200ms$	T4 CPU Hungry	T2, T6
3	$500ms/200ms$	T5 CPU Hungry	T1, T3
	$500ms/200ms$	T6 Disparity	T2, T4

C. Evaluation Metrics

To evaluate the level of intrusion added to the embedded system, five metrics were selected to describe the impact on the system execution, mainly regarding its temporal behavior and memory usage. They are:

- **System Overhead**: The overhead O_m on the system is a measure of the extra time added to the system execution when enabling data acquisition through the monitoring m. Instead of measuring the overhead added on specific OS operations, we can in fact measure the overall impact m introduced in the system execution by comparing the accumulated idle time (i.e., the amount of time the CPU expends idling during the system execution, also called free time) with and without m. Thus, we can measure the overhead O_m as $I - I_m$, where I is the accumulated idle time with m disabled and I_m with m enabled.
- **Task Overhead**: The task overhead α_t is a measure of the extra time added to the specific task execution. This measure is obtained by monitoring the task execution time. The task overhead is given by $\alpha_t = AET_{t,m} - AET_t$, where AET_t represents the task Average Execution Time (AET) with m disabled and $AET_{t,m}$ with m enabled. This measure is focused on capturing the impact of possible preemption made to run the monitor (e.g., when running with interrupts enabled) and the architectural impact of the monitoring execution. Lastly, the task overhead metric is divided into three measures, one for each of the tasks: α_{cpu} for the CPU-bound (i.e., CPU Hungry), α_{mem} for the memory-bound (i.e., Bandwidth) α_{mid} for the mid-term one (i.e., Disparity).
- **Latency**: The latency L_m of the monitoring approach m is a measure of the average execution time taken by the data acquisitions performed by m. Thus, the latency of m can be defined as $L_m = \frac{1}{N}\sum_{i=1}^{N} t_i$, where $t_i \in T_m$, T_m is the set of the measured execution time of the data acquisitions performed by m, and $N = |T_m|$, the number of data acquisitions performed.
- **Jitter**: The Jitter J_m of the monitoring approach m is measure as $J_m = stdev(t_i \ \forall \ t_i \in T_m)$.
- **Memory Consumption**: The memory consumption M_m of the monitoring approach m is the amount of memory required by the data structures used by m to compute and store the samples.

D. Monitoring Setup

Each of the monitoring approaches was configured to sample seven different Clerks simultaneously, six of which for PMU performance counters and an extra one for the running task's identifier, exploring the maximum capacity of the PMU while tagging the samples with an identifier for the running task. In addition, the `Monitor` automatically timestamps samples with the highest resolution timer available on the platform. In the following, we describe the specific configurations for each of the monitoring approaches in terms of sampling rate and the expected number of captures, considering 30-seconds executions.

1) Execution-flow-based: The Execution-flow-based monitoring' sampling rate is dependent on the frequency the system execution passes through a point of interest. Considering the configured scheduler, this period is expected to not exceed $10ms$. Therefore, the sampling rate s for the Execution-flow-based monitoring was set to $100Hz$. In this way, given an execution length of 30 seconds, the expected number of samples to be collected is given by $\Theta_s = \Theta_e * 1/s * CPUS$, where Θ_e is the length of the execution, and $CPUS$ is the number of monitored CPUs. Thus, for the task-set depicted in Table I, $\Theta_s = 9000$, i.e., the product of the sampling rate and execution time.

2) Job-based: As described in Section III-D, Job-based monitoring does not require the specification of a sampling rate for each Clerk. Instead, a sampling approach must be specified. In this case study, a sample is taken for each job of a task. This is done by accumulating the Clerks every time a context switch is performed and capturing the sample at the end of a job. This is implemented on the underlying OS by setting the $Sampling\_Trigger$ in Algorithm 1 to verify whether the job execution has finished, for instance, checking if the state of the job leaving the CPU is to wait for its next period to start.

In this sense, the expected number of samples for a Job-based monitoring is given by $\Theta_s = \sum_{i=1}^{N} \frac{\Theta_e}{T_i.P}$, where N is the size of the task-set, and $T_i.P$ is the period of the ith task in the task-set. Thus, for the task-set depicted in Table I, $\Theta_s = 360$.

3) Periodic: In opposition to Execution-flow-based- and Job-based monitoring, Periodic monitoring has no inherent limitations in terms of the maximum sampling rate. Instead of such limitation, as previously mentioned, the Periodic monitoring approach configures a timer to trigger data sampling according to the desired sampling rate. In order to conduct a fair comparison in terms of the chosen metrics IV-C, the Periodic sampling rate was set to $100Hz$. In this way, the expected number of samples is equal to the Execution-flow-based approach, where for the task-set depicted in Table I, $\Theta_s = 9000$.

V. Results

Each of the metrics was evaluated for both all the monitoring approaches and the baseline configuration. Considering the execution time of each execution (i.e., 30 seconds), and the task-set configuration I, each task is scheduled to run 60 times per execution. Additionally, a set of 10 executions were collected for each of the monitoring approaches, providing a total of 600 measurements for the tasks' execution time. In the following subsections, we analyze each of the metrics evaluated.

A. Memory Consumption

In terms of memory utilization, as described in Section III-A, all the data structures required for the execution of the monitoring system (e.g., Clerk's buffers) are statically allocated during OS's boot time, which allows us to estimate memory consumption before the system execution based on the expected number of samples to be collected during runtime. As previously mentioned, the monitoring system tags the data with high-resolution timestamps, which use 8 bytes. Moreover, in this architecture, PMU performance counters readings are given in 64 bits (8 bytes) unsigned integers. Let us consider the union of a timestamp and a data point as a Snapshot. In this way, the memory consumption for each monitoring approach is given by equation (1), where C is the number of monitored Clerks. The resulting memory consumption for each of the monitoring approaches is depicted in Table II.

$$M_m = \Theta_s(m) * |Snapshot| * C \tag{1}$$

Moreover, for Job-based monitoring, an auxiliary accumulator is needed, as depicted in Algorithm 1. Thus, for the Job-based monitoring, the memory consumption is given as:

$$M_{Job-based} = \Theta_s(m) * |Snapshot| * C + |Data| * C * N \tag{2}$$

where N is the number of tasks in the system.

TABLE II
Memory Consumption

Method	Memory
Execution-flow-based	984.4 KBytes
Periodic	984.4 KBytes
Job-based	39.7 KBytes

B. System Overhead

The system overhead is measured as the impact on the idle time of the system, as described in Section IV-C. To estimate this impact, we first calculate the average CPU_time of each of the cores during all executions, where the CPU_time of execution is defined as the sum of all the execution time of the jobs running on a core (3).

$$CPU\_time(x,k) = \sum_{i=1}^{n\_jobs} time(j_i),\ \forall\ job\ j \mid j \in CPUx,k \tag{3}$$

where, x is the CPU number, and k is the execution number.

Table III presents the results for CPU_time for each of the executing cores on each of the monitoring approaches. Once the average CPU_time is obtained, $Idle\_time$ on a core is

defined as the expected execution time of 30 seconds minus the average CPU_time of this core (4).

$$Idle\_time(x) = 30s - \frac{\sum_{i=0}^{k} CPU\_time(x,i)}{k} \quad (4)$$

where k is the number of executions of the system.

TABLE III
CPU TIME

CPU	Method	Avg. Time	Std. Dev.
1	Baseline	$17817796\mu s$	0.4191%
	Execution-flow-based	$17683931\mu s$	0.4333%
	Periodic	$17651836\mu s$	0.4516%
	Job-based	$17609303\mu s$	0.4568%
2	Baseline	$23682966\mu s$	0.0035%
	Execution-flow-based	$23779260\mu s$	0.2745%
	Periodic	$23711761\mu s$	0.0538%
	Job-based	$23701492\mu s$	0.2050%
3	Baseline	$23140117\mu s$	0.0219%
	Execution-flow-based	$23169998\mu s$	0.0708%
	Periodic	$23152417\mu s$	0.0633%
	Job-based	$23111943\mu s$	0.0677%

Table IV presents the results for the $Idle\_time$ for each of the executing cores on each of the monitoring approaches. With the average $Idle\_time$ calculated for all cores, system overhead is calculated according to IV-C. Table V depicts the average of the system overheads on the average $Idle\_time$.

TABLE IV
IDLE TIME

CPU	Method	Avg. Idle Time
1	Baseline	$12182204\mu s$
	Execution-flow-based	$12316069\mu s$
	Periodic	$12348164\mu s$
	Job-based	$12390697\mu s$
2	Baseline	$6317034\mu s$
	Execution-flow-based	$6220740\mu s$
	Periodic	$6288239\mu s$
	Job-based	$6298508\mu s$
3	Baseline	$6859883\mu s$
	Execution-flow-based	$6830002\mu s$
	Periodic	$6847583\mu s$
	Job-based	$6888057\mu s$

TABLE V
SYSTEM OVERHEAD

Method	CPU	System Overhead	Avg. Impact
Execution-flow-based	1	$-133865\mu s$	0.2870%
	2	$96294\mu s$	
	3	$29881\mu s$	
Periodic	1	$-165960\mu s$	0.1248%
	2	$28795\mu s$	
	3	$12300\mu s$	
Job-based	1	$-208493\mu s$	-0.1540%
	2	$18526\mu s$	
	3	$-29174\mu s$	

C. Task Overhead

The Task Overhead $\alpha_{t,m}$ on a task type t caused by the monitoring approach m is given by the average overhead on each instance of such task. Table VI presents the AET for each task considering a baseline (i.e., without monitoring) and each of the monitoring approaches. The table also presents the standard deviation of the tasks AET. This metric demonstrates the variability imposed into the execution of these tasks. Moreover, the table also includes the proportional overhead for each task when compared to the baseline AET.

The average overhead of T4 and T5 composes the overhead of the CPU-bound tasks $\alpha_{cpu,m}$, while T2, T3, and T6 compose the average overhead of mid-term task $\alpha_{mid,m}$, and T1 the overhead of Memory-bound tasks $\alpha_{mem,m}$, as T1 is the only instance of this type of task. The $\alpha_{t,m}$ for each task and monitoring approach is presented in Table VII.

TABLE VI
TASK OVERHEAD

Task	Method	AET	Std. Dev.	Overhead
T1	Baseline	$99427\mu s$	0.1844%	$-$
	Execution-flow-based	$96857\mu s$	0.1564%	-2.5848%
	Periodic	$95968\mu s$	1.3910%	-3.4789%
	Job-based	$98498\mu s$	0.3648%	-0.9344%
T2	Baseline	$198498\mu s$	0.0937%	$-$
	Execution-flow-based	$196612\mu s$	0.0515%	-0.9501%
	Periodic	$196390\mu s$	0.0750%	-1.0620%
	Job-based	$195772\mu s$	0.0884%	-1.3733%
T3	Baseline	$200429\mu s$	0.0604%	0%
	Execution-flow-based	$201184\mu s$	0.0564%	0.3767%
	Periodic	$200205\mu s$	0.3436%	-0.1118%
	Job-based	$201037\mu s$	0.0692%	0.3033%
T4	Baseline	$194292\mu s$	0.0011%	$-$
	Execution-flow-based	$194551\mu s$	0.0109%	0.1333%
	Periodic	$194489\mu s$	0.0246%	0.1014%
	Job-based	$194303\mu s$	0.0021%	0.0057%
T5	Baseline	$194293\mu s$	0.0011%	$-$
	Execution-flow-based	$194596\mu s$	0.0111%	0.1560%
	Periodic	$194431\mu s$	0.0283%	0.0710%
	Job-based	$194307\mu s$	0.0020%	0.0072%
T6	Baseline	$194561\mu s$	0.4191%	$-$
	Execution-flow-based	$194362\mu s$	0.4333%	-0.1023%
	Periodic	$194843\mu s$	0.4516%	0.1449%
	Job-based	$193876\mu s$	0.4568%	-0.3521%

TABLE VII
OVERHEAD PER TASK TYPE

Task Type	Execution-flow-based	Periodic	Job-based
CPU-bound	0.1446%	0.0862%	0.0064%
Memory-bound	-2.5848%	-3.4789%	-0.9344%
Mid-term	-0.2252%	-0.3429%	-0.4740%

D. Monitoring Latency and Jitter

The latency of each monitoring approach L_m on a given execution is calculated following the description in Section IV-C. The average monitoring latency considers the average L_m on all different executions. In this sense, jitter J_m is calculated using the values observed on all executions and the formulation on Section IV-C. The results for latency and jitter are presented in Table VIII.

TABLE VIII
MONITORING APPROACH LATENCY AND JITTER

Method	Latency	Jitter
Execution-flow-based	$5\mu s$	$1.9461\mu s$
Periodic	$6\mu s$	$2.0924\mu s$
Job-based	$3\mu s$	$2.1482\mu s$

VI. Discussion

When designing the monitoring approaches over the Monitoring Framework proposed in [8], Execution-flow-based stands out as an approach less intrusive than adding interruptions. However, it does not guarantee an exact moment the sampling will be taken considering its sampling rate, as it only collects data when the system execution passes through a point of interest. In terms of computation at each data acquisition, the least amount of computation performed on Execution-flow-based monitoring is the sampling rates verification itself, which, in this case, is represented by $Period + Last\_Capture < T_{now}$.

On the other hand, the Periodic approach is more deterministic in terms of sampling periodicity, as it incurs an interruption every period. Nevertheless, it can interfere with the execution of a critical task or even trash the system execution if the sampling rate of each Clerk is dissonant. The Job-based is the one with fewer interactions with the system, incurring into a data acquisition only at context-switches.

We choose to perform separated monitoring approaches once we aim at assessing the overhead of each specific technique. Therefore, we did not address mixed sampling approaches, such as [10], which adopted two monitoring policies that are executed in different moments of the execution. Moreover, when considering an event-driven trigger for monitoring, the execution-flow monitoring approach performs similarly to the hybrid methods proposed by [10], once periodic sampling will be dismissed whenever a sufficiently close event-driven capture was recorded. Other than that, control-flow graphs are not easily generated for complex multicore real-time systems and all our monitoring approaches are independent of such mechanisms.

In terms of memory consumption, Job-based monitoring consumes significantly less memory for data storage (nearly 96% less), presenting a trade-off between granularity of collected data and memory consumption. For some domains of application regarding the monitored data usage, such loss on data granularity are not ideal. For instance, Leng et al. [12] claim that Job-based monitoring allowed anomaly detection but without the ability to determine the moment an abnormal behavior happens. On the other hand, works on performance optimization, like SPARTA [5], RMC [6], and Hoffmann and Fröhlich [4], demonstrate the ability to learn system behavior and actuate based on Job-based monitoring. Nevertheless, memory consumption tends to be a limiting factor only for very limited architectures or for sufficient long executions. Furthermore, for runtime actuation, the memory consumption is usually considerably reduced in every monitoring approach, as only data required for actuation is kept in memory.

Due to the characteristics of each approach configuration, Execution-flow-based and Periodic approaches perform the same number of captures. This is demonstrated in Section IV-D, where Θ_s for both monitoring approaches were equal.

The results in terms of overhead are related to the number of times data is collected. The Job-based sampling implies a data acquisition only when a context switch is performed, thus, fewer data acquisitions are performed when compared to the other approaches. Moreover, Periodic monitoring deals with timer handling, and Execution-flow-based deals with time condition checking, increasing their latency when compared with the Job-based one, as presented in Table VIII.

The presence of negative overheads shown in Table VII for the Memory-bound and the Mid-term tasks is partially explained by the presence of resource contention between these tasks for the shared L2 Cache and memory buses. Shared resource contention is an issue commonly observed in multicore scenarios, especially for memory-bound tasks concurrency over shared elements of the Memory hierarchy like caches [20], [21]. The task-set used in this case study (Table I) is an example of a task-set with shared resource contention over the Memory Bound task (Bandwidth) and the Mid-term task (Disparity). The contention over these tasks has already been explored by other works [4], [22], which shown to be advantageous to avoid the parallel execution of these tasks.

The presence of the Monitoring Framework, and subsequently, of the monitoring approaches incur on extraneous computation to be performed at each interaction point (e.g., interruptions, dispatches, context-switches), which, even though presenting a negligible latency, as presented in Table VIII), still presents effects over the contention by reducing the time these two tasks interacted.

Regardless of positive or negative overhead, the less intrusive monitoring approach is the one with the task overhead closer to 0. Table VII shows that the Job-based approach is the one that yields the overhead per task type closest to 0, except for the Mid-Term task. So, when comparing to the Execution flow-based and Periodic monitoring, it provided the task behavior closest to the original. This is expected since this approach is the one that performs the lowest amount of data acquisitions overall. Nevertheless, in terms of jitter, Execution-flow-based monitoring achieved the best results for both Memory-bound and Mid-term tasks, as shown by the Standard Deviation column in Table VI.

The main difference when comparing Execution-flow-based to Periodic monitoring is the additional interrupts incurred by the Periodic monitoring, as Execution-flow-based takes advantage of the very own OS interactions to perform the data acquisition. The impacts of such difference are depicted by both monitoring latency and jitter and at the tasks average execution time standard deviation, where Execution-flow-based presented a lower standard deviation for all tasks when compared to the Periodic monitoring.

Execution-flow-based monitoring yields the best results regarding jitter in both system's and tasks' overhead at the cost of incurring a 67% higher monitoring latency when compared to Job-based. It also presents a significantly higher impact on tasks execution time regarding the baseline average for both CPU- and Memory-bound tasks. Nevertheless, it presents the smallest impact for the Mid-term task.

VII. Conclusion

In this paper, we evaluated the impact of monitoring the system behavior through a monitoring framework configured

under three different data sampling approaches, namely Periodic, Execution-flow-based, and Job-based sampling. The evaluation was performed through a set of metrics, including jitter, latency, overhead, and memory consumption, that were measured on executions that account for memory-bound, CPU-bound, and mid-term tasks.

The results obtained point that the intrusion levels for any of the monitoring approaches are low. In fact, none of them presented an average impact over the system execution time higher than 0.3%. Job-based monitoring presented better results for overall overhead and memory consumption. However, its usage depends on the application goal since it implies a trade-off between performance and data granularity. For sampling in higher frequencies, Execution-flow-based monitoring achieved the lowest jitter for the average execution time of every task when compared with the Periodic monitoring. For CPU-bound ones, Job-based shown the lowest jitter. When analyzing the impact on tasks execution time, Execution-flow-based monitoring also presented better results over the Periodic monitoring for the tasks affected by shared-resource contention. However, it is important to remember that, the Execution-flow-based monitoring is dependent on the OS behavior since data acquisition is only executed when a point of interest is reached by the execution, which limits the maximum sampling rate.

The low interference presented by the monitoring approaches are mostly due to the low-intrusiveness aimed at the design of the Monitoring Framework itself, including statically allocating data structures at OS boot time, statically instrumenting code on already known OS interactions points (except for periodic monitoring), and processing data speculatively at the idle time or the end of the execution.

Future works include the extension of this evaluation for different sampling approaches such as hybrid ones. Moreover, extending the evaluation for ML models running over each of the monitoring approaches, for instance, obtained ML results and performance to assess the relation between data quality and monitoring overhead as well as assessing the impacts generated by each of the ML algorithms.

REFERENCES

[1] S. Sarma, N. Dutt, P. Gupta, N. Venkatasubramanian, and A. Nicolau, "Cyberphysical-system-on-chip (cpsoc): A self-aware mpsoc paradigm with cross-layer virtual sensing and actuation," in *Proceedings of the 2015 Design, Automation and Test in Europe Conference and Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, p. 625–628.

[2] T. Mück, A. A. Fröhlich, G. Gracioli, A. M. Rahmani, J. a. G. Reis, and N. Dutt, "Chips-ahoy: A predictable holistic cyber-physical hypervisor for mpsocs," ser. SAMOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 73–80. [Online]. Available: https://doi.org/10.1145/3229631.3229642

[3] N. Dutt, A. Jantsch, and S. Sarma, "Toward smart embedded systems: A self-aware system-on-chip (soc) perspective," *ACM Transactions on Embedded Computing Systems*, vol. 15, no. 2, Feb. 2016. [Online]. Available: https://doi.org/10.1145/2872936

[4] J. L. Conradi Hoffmann and A. A. Frohlich, "Online machine learning for energy-aware multicore real-time embedded systems," *IEEE Transactions on Computers*, pp. 1–1, 2021.

[5] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt, "SPARTA: Runtime task allocation for energy efficient heterogeneous manycores," *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 1–10, 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7750975

[6] T. Mück, S. Sarma, and N. Dutt, "Run-dmc: Runtime dynamic heterogeneous multicore performance and power estimation for energy efficiency," in *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES '15. IEEE Press, 2015, p. 173–182.

[7] J. L. C. Hoffmann, L. P. Horstmann, and A. A. Fröhlich, "Anomaly detection in multicore embedded systems," in *2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2019, pp. 1–8.

[8] L. P. Horstmann, J. L. C. Hoffmann, and A. A. Fröhlich, "A framework to design and implement real-time multicore schedulers using machine learning," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2019, pp. 251–258.

[9] S. Fischmeister and Y. Ba, "Sampling-based program execution monitoring," in *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 133–142. [Online]. Available: https://doi.org/10.1145/1755888.1755908

[10] C. W. W. Wu, D. Kumar, B. Bonakdarpour, and S. Fischmeister, "Reducing monitoring overhead by integrating event- and time-triggered techniques," in *Runtime Verification*, A. Legay and S. Bensalem, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 304–321.

[11] C. Woralert, J. Bruska, C. Liu, and L. Yan, "High frequency performance monitoring via architectural event measurement," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 2020, pp. 114–122.

[12] E. W. L. Leng, M. Zwolinski, and B. Halak, "Hardware performance counters for system reliability monitoring," in *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, 2017, pp. 76–81.

[13] M. Raiyat Aliabadi, M. Seltzer, M. Vahidi Asl, and R. Ghavamizadeh, "Artinali#: An efficient intrusion detection technique for resource-constrained cyber-physical systems," *International Journal of Critical Infrastructure Protection*, vol. 33, p. 100430, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1874548221000226

[14] A. Das, B. M. Al-Hashimi, and G. V. Merrett, "Adaptive and hierarchical runtime manager for energy-aware thermal management of embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 15, no. 2, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2834120

[15] A. Merkel, J. Stoess, and F. Bellosa, "Resource-conscious scheduling for energy efficiency on multicore processors," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 153–166. [Online]. Available: https://doi.org/10.1145/1755913.1755930

[16] G. Gracioli and A. Fröhlich, "Towards a Shared-data-aware Multicore Real-time Scheduler," in *Real-Time Scheduling Open Problems Seminar (RTSOPS)*, Paris, France, Jul. 2013. [Online]. Available: http://www.lisha.ufsc.br/pub/Gracioli_RTSOPS_2013.pdf

[17] ARM, *ARM Cortex-A53 MPCore Processor*. ARM, 2016.

[18] Heechul Yun. (2019) Misc micro-benchmarks & tools. [Online]. Available: https://github.com/heechul/misc/

[19] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "SD-VBS: The San Diego vision benchmark suite," *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 55–64, october 2009.

[20] N. Akram, Y. Zhang, S. Ali, and H. M. Amjad, "Efficient task allocation for real-time partitioned scheduling on multi-core systems," in *2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*. IEEE, Jan. 2019.

[21] G. Gracioli and A. A. Frohlich, "CAP: Color-aware task partitioning for multicore real-time applications," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. IEEE, Sep. 2014.

[22] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, "Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Quinton, Ed., vol. 133, Dagstuhl, Germany, 2019, pp. 27:1–27:25.