An experimental approach for evaluating cache allocation policies in multicore for real-time mixed-criticality systems

Aléxis Génèrès, Michaël Lauer, Jean-Charles Fabre

An experimental approach for evaluating cache allocation policies in multicore for real-time mixed-criticality systems

Aléxis Génèrès*
* LAAS-CNRS
Université de Toulouse, UPS
Toulouse, France
Email: firstname.name@laas.fr

Michaël Lauer‡,
‡ ISAE-SUPAERO / LAAS-CNRS
Université de Toulouse
Toulouse, France
Email: fistname.name@isae-supaero.fr

Jean-Charles Fabre†,
† LAAS-CNRS
Université de Toulouse, INP
Toulouse, France
Email: firstname.name@laas.fr

***Abstract*—Embedded systems are experiencing an increasing demand for computational power. A commonly adopted solution to meet this demand involves deploying both critical and non-critical tasks on a single multicore processor. Nevertheless, the intricacy of such processors induces nondeterminism, posing potential risks to the dependability of the system. This becomes particularly pertinent in safety-critical real-time applications where temporal faults could lead to missed deadlines for high-criticality tasks. Emerging technologies like Intel's *Cache Allocation Technology* (CAT) are designed to diminish the non-determinism instigated by shared cache memory in multicore systems, by enabling dynamic cache memory allocation. In this paper, we introduce an experimental methodology to gauge the efficacy of such technology in a real-time setting. We investigate the possibility of leveraging dynamic cache memory allocation to ensure high-criticality tasks meet their deadlines while optimizing the performance of non-critical tasks. Our proposed methodology involves an exhaustive analysis of the trade-offs between various parameters in a mixed-criticality application. The effectiveness of this approach is substantiated through a sensitivity analysis on a practical use case.**

***Index Terms*—timing faults, mixed-criticality, real-time, multicore, cache allocation policies**

I. Introduction

The use of multicore processors in industrial systems, particularly embedded systems, is increasing for obvious performance and economic reasons. One of the benefits of these hardware architectures is to enable applications of mixed criticality to run on the same component. Most of such applications are safety critical and mixed criticality systems [2] must comply with real-time requirements. Timing faults [1] are a threat to dependability.

The task response time depends on the number of tasks executed in a given period of time, the processor performance, and the scheduling policy. However, in multicore systems, response time also depends on the system's state and more precisely on its shared resources (namely caches, buses, etc...). Shared resources may introduce then nondeterminism for high criticality tasks, because of their usage by lower-critical tasks. Their response time increases and may become unpredictable in case of shared resources contention (namely shared cache).

Despite best practices and techniques to estimate compliance with real-time requirements (WCET evaluation, scheduling analysis, scheduling policies), the new hardware architectures may have an impact on the timing behavior of both critical and non-critical tasks: the former may miss deadlines, the latter may observe a decrease in terms of performance.

Timing faults and timing unpredictability are major impairments to dependability in multicore-based safety critical systems, such as in avionics [17].

The current solutions to limit the effects of shared resources tend to be pessimistic, e.g., the deactivation of all cores except one during the execution of high-criticality tasks. Another proposed solution consists in reserving a large amount of shared resources (e.g., shared cache) to make sure that high criticality tasks will comply with their real-time constraints, while non-critical tasks will see their performance very much diminished. Such pessimistic approaches lead to a sub-optimal use of a multicore system.

The challenge in multicore systems is striking the right balance between critical and non-critical tasks. While high-criticality tasks must invariably meet their deadlines, non-critical tasks should harness available resources to maximize performance. However, finding the optimal trade-off is complex. Factors such as hardware architecture, task specifics, and shared resource allocation play pivotal roles. Given these multifaceted parameters, there is not a one-size-fits-all solution.

In this context, our primary focus is to delve into the following research question: **How does a specific cache allocation policy influence the performance of non-critical tasks, while ensuring the safety of high-criticality tasks?**

This paper introduces a methodology and accompanying experimental framework designed to guide safety-critical real-time system designers in selecting the most effective policy for enhanced system efficiency and dependability. Furthermore, we provide a proof of concept, highlighting the advantages of the Cache Allocation Technology (CAT) dynamic mechanism within a real-time mixed-criticality multicore system.

To summarize, we explore how different cache allocation policies can enhance mixed-criticality systems on multicore platforms. Through our research, we seek to help system designers to determine the best trade-offs between critical tasks and non-critical tasks running on the same multicore processor, using cache allocation.

In section II, we detail the problem statement. In section III,

we evaluate the suitability of CAT in a real-time context. In section IV, we describe the experimental approach designed to help system designers choose the best allocation policy for their specific system. In section V, we illustrate the experimental approach on a case-study, the side effects of dynamic cache allocation and its potential benefits. Related works are presented in section VI and, we conclude in section VII.

II. Context and Problem Statement

The parallelism brought by multicores enables the integration of more applications in a same computing unit, thus reducing the overall number of components required in an embedded systems. This is key to simplify Electrical/Electronic (E/E) architectures, facilitate maintenance, reduce weight and energy consumption, and design reusable and scalable systems.

However, to be safe, the execution time of critical tasks in a dependable system must have some degree of predictability. In particular, the worst-case execution time (WCET) of a task needs to be bounded in order to be able to prove that its deadlines are met in all cases. A weakness of multicores is that they add new sources of unpredictability with respect to single core processors [3].

Indeed, on such architecture, hardware resources like cache memory, system bus, I/O devices, are shared among the cores (see [12] for an exhaustive list of unpredictability sources specific to multicores). Each access to a shared resource is potentially concurrent with accesses from other cores and these contentions are arbitrated by low-level hardware components. Precise prediction of the impact on execution is intractable. As said in [21]: *"..., it will be extremely difficult, if not impossible, to develop such a method that can accurately capture the contention."*. This tends to make WCET estimation pessimistic which in turn degrades the tradeoff between usability and dependability in multicores processors.

In this paper, we focus on strategies based on cache allocation. Despite its limited efficiency in mitigating intra-core interference — between tasks executing on a shared core — it is recognized for substantially reducing inter-core interference in the Last-Level Cache (LLC), thereby enhancing system predictability [8]. Note that cache allocation should be regarded as only a part of the solution, since there are other contributors (memory bus and main memory for instance) to the unpredictability of task execution time, as shown in [11].

Dedicating part of the cache to critical tasks is an improvement for safety and real-time, however, a static allocation can be detrimental to non-critical tasks' performance since it reduces the amount of cache available to them at all times.

New technologies like Intel's *Cache Allocation Technology* (CAT) [18] allow to dynamically control cache allocation at run-time. Thus, allocation can be adapted at run-time to best fit the system requirements and optimise performances [9]. In this paper, we explore the benefits of this technology for mixed criticality real-time systems.

Our first goal is to evaluate the suitability of dynamic cache allocation in a real-time context. Then we show how dynamic cache allocation can improve the predictability of critical tasks while mitigating performance degradation of non-critical tasks in a mixed criticality system. However, since dynamic cache allocation may not be required for all task sets, we propose a method to analyse the impact of different cache allocation policies (dynamic, static, or even no allocation) on a given task set to help a system designer choose the right policy and configuration.

III. Dynamic Cache Allocation in Real-Time Linux

In this section, we assess whether dynamic cache allocation can be effectively employed in the context of real-time Linux.

A. Cache Allocation Technology

Our work relies on the *Cache Allocation Technology* (CAT) provided by Intel. CAT is currently available on Intel Xeon E5 V4, some Xeon E5 V3, and Celeron J3455, and is fully supported on Linux [6]. Although these processors are not designed with embedded systems constraints in mind, they are suitable testbeds to evaluate the benefits of various cache allocation strategies. Indeed, CAT is not specific to a given architecture and could be ported to (or available on) more dedicated processors.

In set-associative cache structures, the cache is divided into several sets, and each set contains multiple slots known as *cache ways*. Allocation is managed by controlling write permissions to these cache ways of the last level cache (LLC). The granularity of allocation corresponds to the number of ways of the LLC. For instance, the Celeron J3455 LLC is 8-way set-associative, meaning data can be stored in one of eight positions or ways within a set. Consequently, allocation is done in chunks equivalent to $1/8^{th}$ of the total cache size. In order to keep track of which resource can access which way, write permissions are managed through the notion of *Class Of Services* (CLOS). Software entities like threads, VMs, containers, can be grouped into a so-called CLOS and each CLOS indicates how much cache it can use. CLOS configuration can be conveniently updated at run-time through a C library or by updating some files in the Linux pseudo-filesystem `/sysfs/rescrtl`.

When using cache allocation in our experimentation, we partition the cache into two parts: one part with write permissions only for the critical tasks and one part with write permissions only for the rest of the system including the non-critical tasks.

B. Dynamic Cache Allocation Principle

Dynamic cache allocation has been shown to improve application performance in server environments [9]. These enhancements occur in the context of large-scale application servers, where the timescale for online cache management is counted in minutes. However, it remains uncertain if dynamic cache allocation could function efficiently with a control frequency in the range of 100 Hz, a characteristic frequency for real-time systems. For example, the rocket prototype discussed in [14] employs three real-time tasks for Guidance and Navigation Control, operating at 20 Hz, 50 Hz, and 125 Hz.

In this work, we evaluate a simple dynamic allocation policy: cache memory is allocated to a critical task solely during its execution - from the start of a job to its termination. Consequently, when the critical task is not running, the entire cache memory becomes available for non-critical tasks.

Briefly, the differences between static allocation and dynamic allocation are as follows:

- *Static allocation* implies a permanently reserved cache memory for the critical task.
- *Dynamic allocation* refers to the allocation of cache memory for each instance of the critical task execution (a *job*). This process results in an *allocation/execution/deallocation* cycle with each task period, i.e. for each job.

The response time for the critical task is expected to be longer with dynamic allocation compared to static allocation. Several factors contribute to this: allocating and deallocating memory every time a job begins and ends incurs overhead. Furthermore, data or instructions can be evicted from the cache from one job to the next, potentially leading to more cache misses. However, we do not expect this latency to be substantially high in the context of real-time systems. This is because the typical workflow of a job involves acquiring some data, processing it, and generating an output, with minimal carryover from one execution to the next.

C. Overhead Evaluation

To better understand if dynamic cache allocation is suitable for real-time systems, we evaluate the overhead of this mechanism. Fig. 1 illustrates the overhead of using dynamic cache allocation for a real-time task, mapped against the frequency of the task. This overhead comprises the time necessary to allocate some cache at the beginning of each job and the time it takes to deallocate this cache upon the job's completion.

For these experiments, we employ an image processing task. This task is fed images that are sized such that its execution time is approximately half of the task's period, resulting in an average system utilization of 50%. The selected frequencies are chosen to avoid periodic system activities that might introduce noise. The experiments are conducted on an Intel Celeron J3455 system running Linux with kernel 5.15 (low-latency).

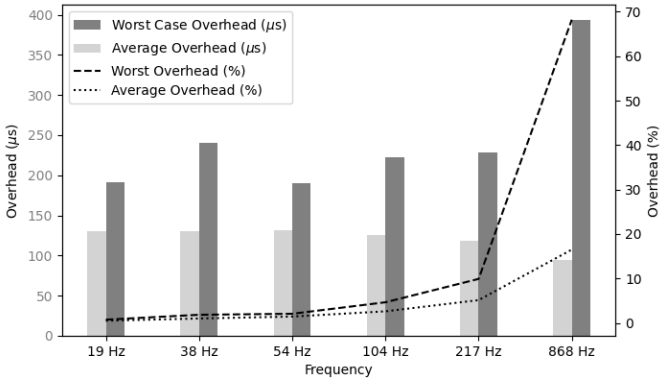

Fig. 1: Dynamic allocation overhead w.r.t. task frequency

The overhead remains stable in the range of 19 Hz to 217 Hz, with an average of 130 µs and a worst-case of 250 µs. The overhead becomes significant with respect to the execution time of the task for 200 Hz and higher, with 5.1 % average and 9.9 % worst-case overhead at 217 Hz for instance. Further increasing the frequency to 868 Hz results in a worst-case allocation and deallocation time of 393 µs, corresponding to a 68 % overhead. With such overhead, it seems unlikely that dynamic allocation might provide advantages over a static allocation. Therefore, at such high frequencies, we would recommend opting for a static allocation strategy.

To test if the load of the system impact the overhead, we also run these experimentation under different workloads, up to 95 % utilization of the CPU. We have not observed significant impact on the overhead.

From our experimentation, we conclude that the overhead of dynamic cache allocation is manageable for real-time embedded systems implemented on Linux. In particular since high frequency task ($>$ 1 kHz), such as sensor fusion are usually executed on dedicated microcontrollers.

IV. Experimental Approach

We propose an experimental approach to analyse the impact of different cache allocation policies (dynamic, static, and no allocation) on a given task set to help a system designer choose the right policy, and allocation configuration. We apply it in section V for illustration and show how a simple dynamic allocation policy can help with mixed criticality system.

Fig. 2 summarises the four main steps of the approach. Outputs of the decision diagram are either a suitable policy and its associated configuration or "failure" meaning that the system designer should explore additional means to improve the system predictability and/or performance. This may involve using different hardware or having more drastic control on non-critical tasks as in [16] for instance.

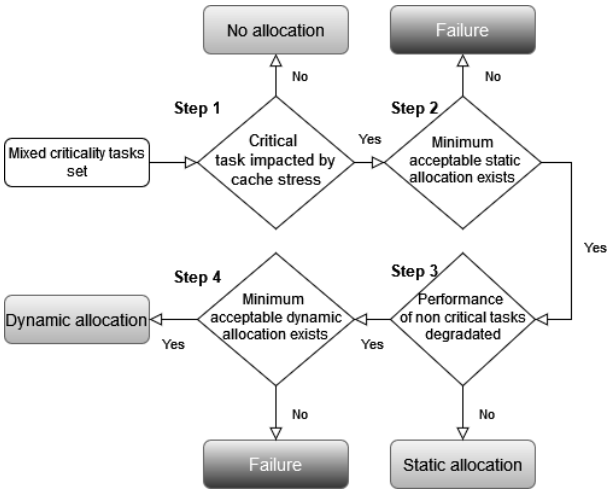

Fig. 2: Decision diagram to configure the allocation

Step 1 tests if the shared cache is actually a significant bottleneck for the performance of the task set. If not, then experiment stops and "no allocation" is chosen because cache allocation is not required.

Otherwise, we proceed to *step 2* in order to find a static allocation allowing safe execution of the critical tasks. We look for a static allocation before a dynamic one because it is the easiest implementation. In addition, the response time with dynamic allocation can only be greater or equal than with static allocation for the critical task. If no allocation is found then the experiment stops and the system designer needs to find other means to improve determinism.

When static allocation is found, then we go to *step 3* and evaluate performance degradation of non-critical tasks. If performance is satisfactory for non-critical tasks, then the experiment stops and an acceptable configuration is found.

If *step 3* does not conclude, we go to *step 4* to evaluate performance improvement achievable thanks to dynamic allocation. In the following the different steps are detailed.

A. Step 1: Do I need cache allocation?

In this first step, we focus on the critical task. The objective of this step is to conclude on the effect of shared cache memory on the critical task. To achieve this, we need to consider best and worst-case scenarios.

First, we need to characterise the critical task in the best-case. To do so, we measure the task response time in the most favorable context: the execution of the critical task alone in the system. The ideal manner to characterise the critical task Response Time (RT) is to repeatedly execute the task until the variance of the measured RT is constant. However, measuring RT and estimating the variance at each repetition might distort the results. So we run batch experimentation, only capturing timestamps at the start and end of the task each period. At the end of a batch, we determine the RT of each execution and the variance. We run batches until the variance stabilizes. Once that's the case, we can have some important experimental information on the critical task: its Worst Case Response Time (WCRT), its Average Case Response Time (ACRT), and its percentage of LLC misses. This WCRT is optimistic due to the favorable conditions of this scenario.

Then, we need to characterise the critical task in the worst-case. To do so, alongside the critical task, we execute *stress-ng* [10] as a worst-case non-critical workload with options to target the LLC and workers on each core. With the full utilization of the CPU by *stress-ng*, we expect the most aggressive pressure on the shared cache.

At this point, we can compare metrics, in particular the WCRTs, of both scenarios with and without stress on the cache. If the probability of deadline violation increases too much with stress, then we move to step 2 to evaluate how cache allocation might help.

B. Step 2: Find a safe static allocation

In this step, we are looking for a cache allocation configuration that prevents degradation of the WCRT of our critical task due to interference with non-critical tasks. Since CAT enables *way-based* partitioning, we partition the cache ways between the critical task and the rest of the system. Allocation is exclusive, we do not allow overlapping partitions.

To check if a partition is acceptable, we execute the critical task alongside *stress-ng*, as previously, but we activate the CAT mechanism. Considering *stress-ng* as a worst-case workload, if an acceptable configuration is found, it will also be safe for any real workload.

To reduce experimental iterations, we employ a binary search to identify the minimal cache ways needed for the critical task. Each experiment continues until the Response Time (RT) variance stabilizes. The search concludes upon finding the smallest cache allocation meeting the designer's Worst Case Response Time (WCRT) criteria.

C. Step 3: Non-critical tasks performance degradation

Since static allocation can be very detrimental to the performance of the non-critical tasks, step 3 checks if the performance level is acceptable. To establish a baseline for the performance of non-critical tasks, we launch critical and non-critical tasks with no allocation. Of course, different performance metrics can be used, but the percentage of LLC misses, and the distribution of response time are recommended.

Then, we run the experiment with the amount of ways determined in step 2 for the static allocation and we compare the results with the baseline obtained without allocation. The output depends very much on the task set selected. In some cases, the use of static allocation (reducing thus the cache size allocated to non-critical tasks) will degrade the response time of non-critical tasks. This is true when these tasks are highly cache sensitive. If the system designer estimates that degradation of performance of non-critical tasks is too important with static allocation, a dynamic allocation strategy will be tried in step 4. If not, we stop the experimentation because we have already ensured safety for the critical task and acceptable performance for the non-critical ones.

D. Step 4: Dynamic cache allocation

Step 4 proposes to use dynamic allocation to limit the detrimental effect of static allocation on non-critical tasks. Dynamic allocation can take many forms. For illustration, we choose a very simple policy. We allocate to the critical task a fixed number of ways. Then, we activate the allocation when the task starts, and we deactivate it when the task terminates, until the next task period. To determine the minimum acceptable number of ways for the critical task, in line with the system designer's WCRT criteria, we use the same method as in step 2: a binary search combined with *stress-ng*. Finally, we conduct an experiment with the non-critical tasks to evaluate if their performance level is satisfactory for the intended system. Typical criteria for non-critical tasks can include percentage of deadline misses, Last Level Cache (LLC) misses, Instructions per Cycle (IPC), and Average Case Response Time (ACRT).

To conclude, we have presented a four steps experimental approach to take advantage of cache allocation technology in the integration of a mixed criticality system. With this approach, as will be shown in section V, the high criticality task complies with its deadline while non-critical tasks increase their performance, at least in the experiments we have

conducted. Of course, the benefits of cache allocation depend greatly on the task set and the targeted CPU. Tasks are more or less cache sensitive and so cache allocation might not always be a solution to predictability and performance issues.

V. Implementation and Experimental results

In this section, we implement our approach on an illustrative example. Our computing unit is a Celeron J3455 which supports CAT. We use two cores that share an 8-way set-associative, 1 MB, LLC. Due to its known cache sensitivity [7], we select an image corner detection task, *Susan* from Mibench [5] as the critical task. We've modified the task in order to schedule it with the Linux Deadline scheduling class and to record its own response time. We use a *period* of 9.6 ms which is equivalent to 104 Hz and is characteristic of real-time systems. Since we want to observe response time variability, we choose a larger than necessary *runtime* and *deadline* parameters, 9 ms, in order to never trigger deadline misses or overrun runtime budget.

A. Illustrative example of Step 1

As indicated in section IV-A, the goal of the first step of our experimental approach is to determine whether our critical task is sensitive to LLC stress or not. To do so, we first run experiments to establish a best-case baseline, i.e. without stress on the LLC, then we establish the worst-case behaviour by imposing a maximum stress on the LLC.

For the best-case scenario, i.e. without stress and non-critical tasks, the critical task is the only task running on the CPU. We use `perf-stat` to record the LLC misses of the critical task. Timestamps used for response time estimation are captured by the task itself. We execute the critical task for five minutes. Then we extract statistics on LLC misses and the task response time. We use the same stopping condition as before. Then, we execute our worst-case scenario by launching *stress-ng* running on the two cores alongside the critical task. The graph in Fig. 3 represents the results of our two scenarios via a cumulative distribution function which represent the probability of deadline violation with respect to a given deadline. In this example with a deadline of 5.1 ms, we observe a probability of deadline violation of less than 0.01% in the scenario without stress, while the probability of deadline violation goes up to 87% in scenario with stress. We observe an increase of 11% of the critical task worst-case response time between the without stress scenario and with stressed one. This effect is confirmed by the increase of average LLC misses which go from 10,51% to 30.14% with stress.

With a probability of 87% of deadline violation in a stressed scenario, we need to protect the critical task with cache memory allocation.

B. Finding a safe static allocation (Step 2)

The second step of our experimental approach focuses on identifying the optimal configuration to mitigate the impact of stress on task deadline violations and LLC misses. We employ static cache allocation to minimize excessive LLC misses.

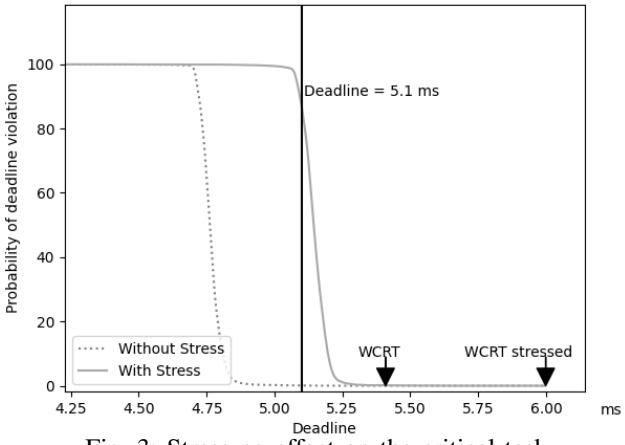

Fig. 3: Stress-ng effect on the critical task

We investigate all potential configurations, ranging from no allocation to allocations spanning up to 7 ways. An allocation of 8 ways (which would be a complete cache allocation) is restricted by the Operating System. Using the same worst-case experimental conditions from step 1, Fig. 4 illustrates the probability of deadline violation for the critical task relative to the number of ways statically allocated.

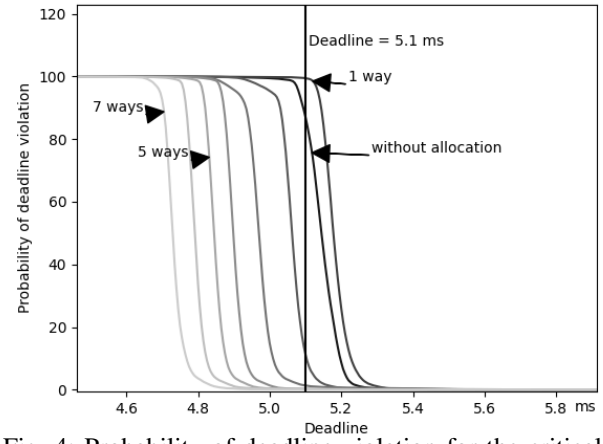

Fig. 4: Probability of deadline violation for the critical task w.r.t. the number of ways allocated

In Fig. 4, it's evident that the allocation of more ways (or cache memory) enhances the performance of the critical task, which aligns with our expectations. The curve with no allocation matches the stressed curve depicted in Fig. 3. Importantly, allocating merely one way yields inferior outcomes. This constrained cache space for the critical task results in increased cache misses, primarily because the task ends up evicting its own cache lines. Such eviction subsequently affects its response time. For the critical task to maintain a sufficiently low probability of violating the 5.1 ms deadline, we allocate 5 ways. With this allocation, the violation probability is 0.21%, and the LLC misses stand at 17.38%.

C. Performance degradation on non-critical task (Step 3)

Once a satisfactory configuration is identified, and the probability of deadline violation meets the designer's criteria, the next step focuses on assessing the impact of this allocation on the performance of non-critical tasks.

For our practical scenario, we selected a cache-sensitive task as our non-critical workload, which has an average execution time of 85 ms when no allocation is applied. For simplicity, the critical and non-critical tasks are executed on different cores. This approach helps to prevent scheduling effects from degrading the performance of non-critical tasks. Note that this is not a limitation of the proposed approach. The non-critical workload runs continuously and is managed by Linux's default scheduler, albeit at a higher priority relative to standard Linux processes, in order to focus on the delays incurred by the cache allocation policy.

To evaluate the performance of the non-critical task, we measure its response time and the LLC misses. We run two scenarios: with and without the static allocation paired with the critical task. In Fig. 5 we have plotted the probability of the response time of a non-critical job to be lesser than a given value X for both scenarios. The higher the probability the better for the performance of the task. We can then compare the effect on the non-critical task performance.

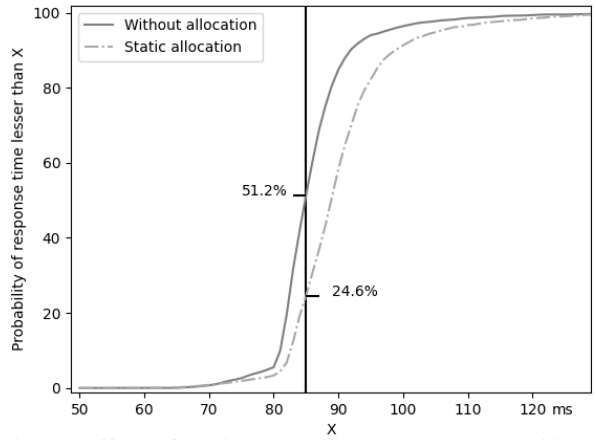

Fig. 5: Effect of static cache allocation on non-critical task

We observe a degradation of performance (i.e. increase of the response time of the non-critical task). For instance, without allocation, 51.2% of the non-critical jobs have a response time lesser than 85 ms, with respect to only 24,6% when allocating 5 ways of cache memory to the critical task. It is clear that the response time of a majority of non-critical jobs is greater with static allocation. Concerning the LLC misses, we have 9.82% without allocation and we have 15.51% with the static allocation.

In order to mitigate the performance degradation of the non-critical workload caused by the static allocation, we jump into the last step to improve the trade-off between critical and non-critical tasks performance using dynamic allocation.

D. Dynamic cache allocation benefits (Step 4)

In this experimental step, we implement a very simple policy. We allocate a fixed number of ways (5 ways) to the critical task and, as explained before, we activate the allocation when one of its jobs starts, we deactivate it when the job terminates. Thanks to the libraries provided for Linux [6], dynamic policies are easy to implement. More sophisticated policies are under consideration.

1) Impact on Critical Task Performance: As presented in section III, dynamic allocation may be less favorable than static allocation for the critical task, in part because of the allocation overhead; to check this, we run an experiment executing our dynamic allocation paired with *stress-ng*. The results obtained should illustrate how much performance degradation the dynamic allocation approach implies with respect to static allocation for the critical task. No allocation is a baseline we consider as well.

The corresponding experiments enable comparing three scenarios: without allocation, with 5 ways static allocation, then 5 ways dynamic allocation. We present the results we obtained in Fig. 6 and Tab. I.

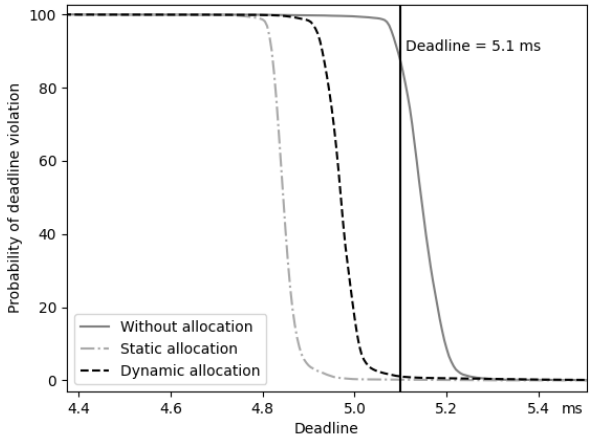

Fig. 6: Comparison between three policies for the **critical task**

	No alloc.	Static alloc.	Dynamic alloc.
Prob(RT>D)	87 %	0.21 %	0.37 %
WCRT	6 ms	5.4 ms	5.7 ms
LLC misses	30.14 %	17.38 %	19.99 %

TABLE I: Critical task performance

As expected, we have a degradation of the performance due to the dynamic allocation mechanism, compared to the static allocation, for the critical task. However, the results are quite acceptable compared to the no allocation policy which is clearly worse.

2) Impact on Non-Critical Task Performance: Lastly, we run another experiment to analyze the impact of our dynamic allocation mechanism for non-critical tasks. The critical task still has access to 5 ways of cache memory when it is

executing. And with the dynamic policy, the non-critical task has access to $3/8^{th}$ of the cache memory during the execution of a critical job and gets full access to the cache memory in the interval between the completion of a critical job and the beginning of the next one. We compare the performance results of the non-critical tasks for the three policies. We display the results in Fig. 7 and Tab II.

We observe some performance improvements (i.e. decrease of the response time of the non-critical task). For instance, with static allocation, 24.6% of the non-critical jobs have a response time lesser than 85 ms, with respect to 41.2% with dynamic allocation. It is clear that the response time of a majority of non-critical jobs is lesser with dynamic allocation. Concerning the LLC misses, we have 15.51% with static allocation and we have 10.76% with the dynamic allocation.

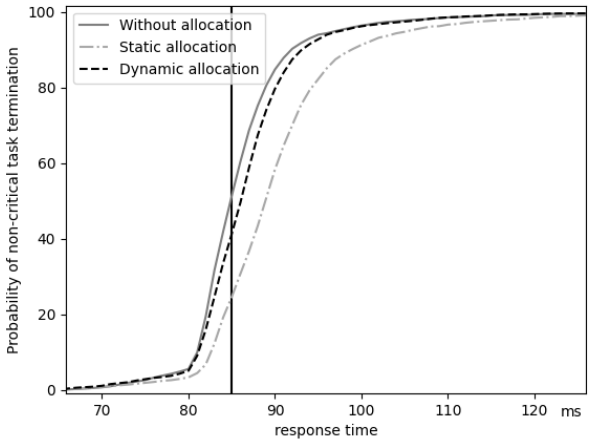

Fig. 7: Comparison between three policies for the **non-critical task**

	No alloc.	Static alloc.	Dynamic alloc.
Prob(RT<85ms)	51.2%	24,6%	41.2%
LLC misses	9.82 %	15.51 %	10.76 %

TABLE II: Non-critical task performance

3) Finding the right trade-off: In summary:

- *critical task*: the best strategy is static allocation, the worst is no allocation.
- *non-critical task*: the best strategy is no allocation, the worst is static allocation.

Clearly, the proposed dynamic allocation provides the best compromise, satisfaction of deadline for the critical task while increasing the performance of the non-critical task. The dynamic allocation executed for the critical task has a cost, but it is not a heavy penalty at least for the practical use case we have used. As illustrated previously, this depends very much on the tasks set and several parameters we have identified. The experimental approach we propose clearly helps the system designer to take appropriate decisions regarding the allocation strategy to use.

VI. Related work

Dynamic cache allocation has been shown to improve application performance in server environments [9]. However, to the best of our knowledge, the application of dynamic cache allocation at the job level in real-time systems has not yet been explored.

The state of the art in managing interference between cores for safe real-time embedded systems is rich, offering solutions that range from shared resource allocation strategies to strict access scheduling, or even new hardware design.

Time partitioning [19], implemented in systems like PikeOS or as used by Girbal et al. [4], allows scheduling to be configured such that no non-critical tasks can be executed while critical tasks are running. Consequently, contention on the shared resources can be entirely avoided. Although this approach provides high safety, it can result in suboptimal processor utilization and is most suited for scenarios where the highest performance level is not a requirement.

Kritikakou et al. [13] and Loche et al. [16] propose a more optimistic approach. Recognizing that deadline violation is an infrequent event, the system operates without any interference prevention measures to optimize performance. However, the system is continuously monitored, and if a risk of a deadline miss is detected, non-critical tasks are halted to avoid contention and ensure the critical tasks complete on time.

Suzuki et al. [22] suggest dedicating a subspace of the cache memory to each core. Critical tasks are then all assigned to one specific core, while non-critical tasks are delegated to the remaining cores. While this method improves predictability, it could lead to processor under-utilization, as the core assigned to critical tasks may not be fully occupied. Additionally, it inherits the drawbacks of static allocation.

Other research, such as that by Liu et al. [15], proposes new CPU architecture paradigms that can avoid unpredictable contention on shared resources. While such solutions may be ideal, their large-scale adoption could prove challenging.

The real-time community has shown interest in mechanisms that enhance determinism in multicore systems. Sohal et al. [20] conducted a comprehensive analysis of all mechanisms proposed by Intel within the Resource Director Technology (RDT) framework. Specifically, regarding the Cache Allocation Technology (CAT) – a component of RDT – they introduce a method to estimate the efficacy of CAT across different CPU models. While their focus remains on static allocation, they demonstrate that although cache allocation is effective, it cannot fully mitigate the side effects caused by shared resources.

VII. Conclusion

The experimental approach proposed in this paper has been applied in a practical use case involving a critical image corner detection task extracted from the MiBench benchmark. The primary goal of this example is to offer a proof of concept in terms of the efficiency of dynamic allocation and the various parameters that need consideration. Although this use case doesn't encapsulate all potential application characteristics, it

aids in understanding the various steps needed to perform the analysis and decide which cache allocation strategy suits the application characteristics and requirements.

Prior to initiating our experimental approach, we examined the overhead of the CAT mechanism in relation to the activation frequency. We concluded that the cost of the CAT allocation mechanism remains acceptable until 200 Hz.

In the *first step*, we verified that the chosen critical task is sensitive to LLC stresses. In the *second step*, we analyzed the temporal behavior of our critical task based on the number of ways allocated statically. In the *third step*, we evaluated the impact of the static allocation, chosen in step 2, on a non-critical task as compared to the case without allocation. The non-critical task was observed to run more slowly with static allocation, thus motivating the strategy proposed in *step 4*.

The dynamic allocation strategy evaluated in step 4 operates on an *allocation/execution/deallocation* profile for the critical task, i.e., LLC space is allocated when the job starts and released upon job completion. The results demonstrated an interesting trade-off between the performance of the non-critical task and the behavior of the critical task from a real-time perspective, despite a minor runtime overhead.

In this work, we found that even a simple dynamic cache allocation policy, under certain conditions, significantly improve the performance of non-critical tasks. This was achieved without compromising the real-time behavior of critical tasks, showcasing the usefulness of dynamic cache allocation.

However, it is worth noting again that the impact of these cache allocation policies greatly depends on several parameters, including task set, multicore CPU, critical task sensitivity to LLC stress, frequency/period to consider, etc.

This implies that such an approach should be tailored by each mixed-criticality system designer to fit their specific operational context.We've demonstrated that there isn't a universal solution suitable for all scenarios, which is why the proposed experimental approach is so valuable.

A limitation of the proposed approach is that we divide the LLC cache into two partitions—one for the critical tasks and one for the rest of the system. This implies that all critical tasks operate within the same partition, potentially leading to cache line eviction amongst them. Future work will explore using to multiple partitions for the critical tasks.

As a perspective, our goal is to incorporate additional criticality levels to better align with industrial standards.

REFERENCES

[1] Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing **1**(1), 11–33 (2004). https://doi.org/10.1109/TDSC.2004.2

[2] Burns, A., Davis, R.I.: Mixed Criticality Systems - A Review : (13th Edition). Pure, The University of York (February 2022), https://eprints.whiterose.ac.uk/183619/, this is the 13th version of this review now updated to cover research published up to the end of 2021.

[3] Cullmann, C., Ferdinand, C., Gebhard, G., Grund, D., Reineke, J., Triquet, B., Wilhelm, R.: Predictability considerations in the design of multi-core embedded systems. In: 4th Embedded Real Time Software and Systems, ERTS^2 2010. pp. 36–42 (2010)

[4] Girbal, S., Jean, X., Le Rhun, J., Pérez, D.G., Gatti, M.: Deterministic platform software for hard real-time systems using multi-core cots. In: 2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC). pp. 8D4–1 to 8D4–15 (2015)

[5] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop. p. 3–14. WWC '01, IEEE Computer Society, USA (2001)

[6] Intel: intel-cmt-cat. https://github.com/intel/intel-cmt-cat (2023)

[7] Joshi, A., Phansalkar, A., Eeckhout, L., John, L.: Measuring benchmark similarity using inherent program characteristics. IEEE Transactions on Computers **55**(6), 769–782 (2006)

[8] Kim, N., Ward, B.C., Chisholm, M., Anderson, J.H., Smith, F.D.: Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. Real Time Syst. **53**(5), 709–759 (2017)

[9] Kim, Y., More, A., Shriver, E., Rosing, T.: Application performance prediction and optimization under cache allocation technology. In: 2019 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 1285–1288 (2019)

[10] King, C.I.: stress-ng. https://github.com/ColinIanKing/stress-ng (2023)

[11] Kloda, T., Solieri, M., Mancuso, R., Capodieci, N., Valente, P., Bertogna, M.: Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 1–14 (2019). https://doi.org/10.1109/RTAS.2019.00009

[12] Kotaba, O., Nowotsch, J., Paulitsch, M., Petters, S.M., Theiling, H.: Multicore in real-time systems – temporal isolation challenges due to shared resources. In: Design, Automation and Test in Europe, DATE 2013 (2013)

[13] Kritikakou, A., Marty, T., Roy, M.: DYNASCORE: DYNAmic Software COntroller to Increase REsource Utilization in Mixed-Critical Systems. ACM Tran. on Design Automation of Electronic Systems **23** (2017), http://dl.acm.org/citation.cfm?doid=3149546.3110222

[14] Linsen, R., Listov, P., de Lajarte, A., Schwan, R., Jones, C.N.: Optimal thrust vector control of an electric small-scale rocket prototype. In: 2022 International Conference on Robotics and Automation (ICRA). pp. 1996–2002 (2022)

[15] Liu, I., Reineke, J., Lee, E.A.: A pret architecture supporting concurrent programs with composable timing properties. In: 2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers. pp. 2111–2115 (2010)

[16] Loche, D., Génèrès, A., Lauer, M., Fabre, J.: Run-time monitoring and control for temporal fault prevention in mixed-criticality systems. In: 17th European Dependable Computing Conference, EDCC 2021, Munich, Germany, September 13-16, 2021. pp. 53–60. IEEE (2021), https://doi.org/10.1109/EDCC53658.2021.00015

[17] Löfwenmark, A., Nadjm-Tehrani, S.: Fault and timing analysis in critical multi-core systems: A survey with an avionics perspective. J. Syst. Archit. **87**, 1–11 (2018). https://doi.org/10.1016/j.sysarc.2018.04.001, https://doi.org/10.1016/j.sysarc.2018.04.001

[18] Nguyen, K.: CAT Cache Allocation Technology (2016), https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html

[19] Rushby, J.: Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Tech. rep., NASA Langley Technical Report Server (1999)

[20] Sohal, P., Bechtel, M., Mancuso, R., Yun, H., Krieger, O.: A closer look at intel resource director technology (rdt). In: Proceedings of the 30th International Conference on Real-Time Networks and Systems. p. 127–139. RTNS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3534879.3534882, https://doi.org/10.1145/3534879.3534882

[21] Suhendra, V., Mitra, T.: Exploring locking & partitioning for predictable shared caches on multi-cores. In: 2008 45th ACM/IEEE Design Automation Conference. pp. 300–303 (2008)

[22] Suzuki, N., Kim, H., Niz, D.d., Andersson, B., Wrage, L., Klein, M., Rajkumar, R.: Coordinated bank and cache coloring for temporal protection of memory accesses. In: 2013 IEEE 16th International Conference on Computational Science and Engineering. pp. 685–692 (2013)