Certifiable Memory Management System for Safety Critical Partitioned System

Alexy Torres Aurora Dugo, Jean-Baptiste Lefoul, Serge Harnois, Felipe Gohring de Magalhaes, Gabriela Nicolescu

Certifiable Memory Management System for Safety Critical Partitioned System

Alexy Torres Aurora Dugo, Jean-Baptiste Lefoul, Serge Harnois, Felipe Gohring de Magalhaes and Gabriela Nicolescu

Abstract—Aerospace systems are safety-critical systems that need to respect tight constraints in terms of execution time, resource usage and predictability. This industry is currently transitioning from predictable single-core processors to less predictable multi-core architectures. This transition reveals multiple challenges due to interferences. The contention of different cores on shared resources introduces interferences. This phenomenon prevents the required isolation between applications and the estimation of their worst-case execution time. To prevent interferences and ease the certification of robust partitioned multi-core systems, guidance documents, such as the CAST-32A, provide objectives on resource isolation and management. In this paper, we propose a memory manager to mitigate memory interferences generated in shared cache, main memory and memory bus. Our results show an increase in timing predictability by 68.1%. Aside the memory manager, based on our results, we provide a set of recommendations to assist system integrators' decisions and ease the certification process by conforming to the current guidance.

Keywords—Aerospace, ARINC-653, Certification, Critical systems, Interference, Resource management, RTOS

I. Introduction

The aerospace domain relies on highly critical software to ensure the reliability and safety of the system. The first generation of computer-assisted functionalities in planes is based on the federated architecture. This type of architecture is designed such that each computer-assisted functionality has its controller or computer, called Line Replaceable Unit (LRU). LRU communicate through different networks and buses. Due to the increased number of computer-assisted features in planes, the use of federated architecture is impossible to sustain. The increase in communication nodes often results in network contention and poor fuel efficiency of the plane due to the equipment weight [1].

To leverage the size, weight and power (SWaP) of federated architectures, the Integrated Modular Avionics (IMA) architecture was proposed. The IMA architecture is designed such that multiple functionalities are gathered in the same module. This design eases the communication between applications, reduces the SWaP and facilitates the maintenance of the equipment. Multiple IMA units can be scattered in the plane and communicate via networks and buses.

IMA architectures rely on Real Time Operating Systems (RTOS) to manage the different components in the system and the different applications running concurrently. Real time systems are subject to strong constraints in terms of execution time and response latency. Hence, it is imperative to ensure the Worst-Case Execution Time (WCET) of all applications. Nowadays, IMA architectures are implemented relying on Commercial Off-The-Shelf (COTS) hardware [1]. This reduces the cost of the equipment but also reduces the design's flexibility.

Despite the determinism of single-core processors, more energy-efficient and powerful multi-core architectures slowly replace them [2]. Multi-core processors provide better performance by the ability to execute multiple applications at the same time. With the increasing production of multi-core architectures, single-core processors slowly disappear from the market. Aerospace system designers need to transition to multi-core architectures to keep the equipment up to date.

However, optimizations brought by multi-core processors make execution times less predictable and impact WCET. The execution of multiple parallel applications also introduces contention on the shared resources. These phenomena are called interferences [2]. Interference channels exist at different levels of the architecture. In this paper, we focus on memory interferences for which channels are the shared caches, shared memory bus and shared DRAM. We propose a memory manager to mitigate memory interferences. For this purpose, we extend the cache, memory and bus management methods proposed by the Single-Core Equivalent framework [3] to allow better integration with certifiable RTOS and without the use of virtualization. We propose a flexible design to accommodate with different architectures and certification processes. Our results show an increase in timing predictability by 68.1% on average while increasing the execution time of applications by 22.3% on average. The Certification Authorities Software Team (CAST) published a guidance paper, the CAST-32A, that provides objectives to be met when certifying multi-core critical systems. Based on this paper, we propose a set of rules and insights to the system integrator (SI) to improve safety and performance of the system [4]. Our work brings the following contributions:

1. We propose a novel memory manager enabling to reduce cache and memory interferences as well as the bus interferences. By integration of most efficient methods for cache and memory partitioning and bandwidth management, we obtain a solution applicable to certifiable systems in a context where hardware-assisted virtualization is not available;
2. Improving the bus bandwidth management by proposing an extension of MemGuard [5], a memory bandwidth manager, to allow more flexibility and I/O bandwidth

A. Torres, JB. Lefoul, F. Gohring de Magalhaes and G. Nicolescu are with the Department of Computer Engineering, Polytechnique Montreal, Montreal, QC, CANADA. email: alexy.torres-aurora-dugo@polymtl.ca

Serge Harnois is with Mannarino Systems & Software Inc., Montreal, QC, CANADA. email: Serge.Harnois@mss.ca

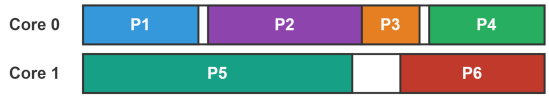

Fig. 1. Multi-core same-length MAjor Time frame defined on two cores

management by extending configuration capabilities, per application budgeting, and an initial budget pool;

3. The proposition of guidelines, named safety net as per CAST-32A [4], to ensure quality of service (QoS) and safety in the RTOS and ease the certification process in avionic systems. We further formalize the notion of robust partitioning for shared memory and bus.

II. Background and Context

To allow their certification, civilian aerospace systems must follow strict rules. Multiple standards exist to ease the development process and RTOS use (e.g., ARINC-653 [6]). In this section we present those requirements and the current challenges of certification in multicore systems.

A. Multicore Safety Critical Systems Challenges

The notion of robust partitioning implies that an application cannot impact any other applications in the system. This is true from a software perspective, where the undefined behavior of the application cannot block or impact the execution of another application [6]. This is also enforced in the hardware aspect, where an application should not change the state of the hardware to the point that it might impact another application. In this context, applications are called partitions [6].

ARINC-653 compliant systems must conform to the following constraints: (1) Time isolation: a partition must execute during a given time slot without being preempted. As shown in Figure 1, the scheduling relies on a MAjor time Frame (MAF) that is repeated indefinitely. (2) Space isolation: a partition can only access resources it has been explicitly allocated. For instance, a Memory Management Unit (MMU) can enforce space partitioning. CAST-32A [4] also proposes the integration of a safety net that should provide means to monitor and recover from failure in space and time isolation.

To allow the use of multi-core processors in critical systems two approaches are foreseen. The first is to take interferences into account during system analysis to consider their impact on timing. With this approach, one could account every access to the cache as a cache miss. This provides overly pessimistic results. The second approach is to bound or eliminate them to allow the use of known analysis methods. In this work, we allow robust partitioning of the shared resources. The objectives provided by the CAST-32A are abstract are sometimes difficult to transpose into system's constraints. Thus, we propose to formalize these objectives regarding shared memories and bus for robust partitioned systems in Section 5.

B. Interferences

In this section, we present the interferences that we consider in our work. We also describe the interference channels studied and their impact on the system's execution.

1) Cache Interferences: Cache interferences in multi-core systems appear in two contexts: shared caches and private caches [2]. Shared caches interferences occur when multiple cores share the same cache. Two types of interferences might appear in this context: (1) Contention interferences, when multiple cores try to access the cache at the same time, only one core is granted access and other core wait to access the bus. (2) Eviction interferences, when an application on a core evict data owned by another application on another core.

2) Bus Interferences: Bus or interconnect interferences occur when multiple components (cores, coprocessors, etc.) try to access the bus at the same time [5]. This contention results from the arbitration of the controller on the bus. Only one component may access and process transaction on the bus at a given time. Thus, the system puts other components that request the access on hold and wait for their turn to use the bus. Although multiple arbitration policies have been proposed to reduce or remove bus interferences, they are rarely available on COTS hardware.

3) Memory Interferences: Main memory interferences can be observed at different levels. Memory can be accessed through different independent channels without showing any interference [7]. However, if two core access the memory through the same channel, the contention on that channel generates interferences.

The second type of interferences appears at the bank level [8]. Each bank comprises a row buffer, which stores the content of the last accessed row in the DRAM. The row buffer acts as a cache that allows accessing the data contained in the same row faster. When multiple core access the same memory banks, the row buffer unpredictably changes its content to accommodate cores access patterns. Thus, affecting the execution time of partitions executing on the different cores.

C. Interferences Mitigation Means

Different mechanisms are present in COTS hardware to isolate components and make the platform more predictable. Table I provides a summary of the mitigation methods we discuss in this article. We also provide the availability of the solution, where it is said to be low when additional hardware is required and such type of hardware is usually not present in COTS processors. A high availability means that the hardware required to apply such technique is usually present in COTS processors (e.g., Performance Monitoring Counter (PMC), Memory Management Unit (MMU), etc.).

Cache way partitioning relies on hardware facilities to allocate ways of the cache to designated cores. Hardware specific registers must be set to allow segregation of the cache.

Cache set partitioning or cache coloring, relies on the structure of cache memories and the physical address of the data to know where the data will be placed in caches. Figure 3 shows the physical address layout that allows to know in which set the data will be loaded. Based on this information, the RTOS can wisely choose the virtual to physical translation to place pages in selected sets of the cache.

Finally, cache line partitioning is a combination of ways and set partitioning, where the RTOS allocate a set of cache ways and cache sets to a partition.

TABLE I
LIST OF INTERFERENCE MITIGATION MEANS AVAILABLE IN COTS HARDWARE

Mitigation method	Interference	Availability
Cache way partitioning	Shared cache eviction	Low
Cache set partitioning	Shared cache eviction	High
Cache line partitioning	Shared cache eviction	Low
Memory bank partitioning	DRAM Row buffer	High
Memory channel partitioning	DRAM channel	Low
Bus budgeting	Bus contention	High

Memory banks and channels partitioning work the same way as cache set partitioning. By wisely choosing the physical address of a data, the RTOS knows in which DRAM bank and channel the data are stored. Placing data of different partitions into different banks will remove bank interferences (row buffer interferences). Channel partitioning allows assigning unique channels to cores. When a core accesses the DRAM, it can do so without contention on its channel due to another core requesting data.

Bus budgeting relies on the monitoring of memory accesses done by a core. When the number of accesses made during a given period of time exceeds a certain amount, the core stops its execution by scheduling an idle task or halting the core. This method has the effect to stop the contending core and release the bus bandwidth it uses.

All the previously mentioned methods provide the same amount of isolation and can be used for safety-critical systems. The choice of a technique over another should be based on the availability of the hardware and the overhead it introduces.

III. RELATED WORK

In this section we present the current state of the art and position our work with regard to the interference channels we study in this article.

A. Cache Interferences

In [9], the authors present a survey of the different techniques for cache partitioning. Cache way partitioning is shown as the most used implementation. While line partitioning offers finer granularity, it also brings a higher execution and development overhead. Finally, set partitioning is known to be more complex to implement but offer better portability.

In [10], the UCP (Utility-Based Cache Partitioning) approach was introduced. It defines cache partitioning and monitoring hardware module to update the cache configuration at run-time. In the avionic domain, all configurations should be static and validated prior to system deployment as certification requires it [11]. Thus, we cannot apply dynamic approaches in this context. In [12] and [13], semi-partitioned caches are explored to improve performance while keeping the system in a more predictable state. The work of [14] extends this concept by clustering applications and partitioning caches on a per-cluster basis. In [15], the authors present a scheduling technique to account of the cache usage and reduce the performance hit introduced by cache partitioning.

In [16] and [17] cache partitioning is proposed alongside prefetching techniques. In our context, the system design often disables performance improvement facilities such as prefetching or branch prediction to reduce non-determinism produced by them. In [18], the authors use different processor management methods to reduce energy consumption. Literature also studied super pages with cache partitioning but the existing solutions use additional hardware [19].

In [20] and [21], hypervisors and the notion of virtual CPUs are proposed to ease cache partitioning implementation. The use of an hypervisor allows the application of cache partitioning to existing OS without modifying their code. However, it yields to an increased overhead in memory size and execution time.

B. Memory Interferences

In [22] the authors give a comprehensive list of memory interferences in COTS platform. The authors propose a methodology to analyze and understand the impact of the different mechanism in DRAM on predictability and latency. In [23] the authors propose a multi-policy resource allocation method. They use DRAM and cache partitioning together to leverage interferences in the system. Their approach relies on the analysis of a huge data set (2000 workloads) of execution (over 10000 experiments).

In [24], [25], [26] scheduling is explored to reduce interferences in the memory hierarchy. This approach allows reducing contention on the shared resources and thus, increase the throughput of tasks naturally. However, in [25], only one application can execute on multiple cores. Even-though scheduling reduces interferences, it does not ensure isolation between the partitions and makes certification more difficult in systems where robust partitioning is required [4].

In [27] the authors propose a software memory coloring approach to separate applications' memory between DRAM banks and channels. The method relies on a dynamic approach that changes the number of allocated banks and channels to threads at run-time. However, we cannot apply this method to hard real-time systems as the dynamic behavior would introduce non-determinism and make WCET estimation more complex due to the added factors to consider.

In [8], the authors coordinate the use of cache coloring and bank coloring. The duality of the approach and the conflict between cache and bank coloring is explained. In this paper, we review the approach proposed in [8] to allow faster integration and certification in safety-critical systems.

In [28] the authors propose a bank partitioning method to improve performance while considering the profile of the applications. The method relies on application profiling and clustering to categories their memory usage. However, we cannot use additional hardware to gather the metrics needed by the online algorithm. Similarly, in [29] the authors propose to isolate concurrent threads to use different memory banks for data sampling applications. Performances are further improved by balancing the load between the different memory banks.

C. Bus Interferences

Bus bandwidth management is a widely studied approach to leverage bus interferences. In [5] the authors present an approach called MemGuard. MemGuard regulates bus bandwidth

and propose different mechanisms to increase performances. Each core has an allowed amount of access to the bus during a quantum of time. If the core exceeds its budget, the RTOS puts it on hold until the next quantum of time. MemGuard uses per-core budget allocation and lacks configuration flexibility. These two main limitations that are further explained and addressed in Section 4.2.

In [25], [30] and [31], scheduling frameworks to leverage predictability in multi-core systems are proposed. The approaches use memory bandwidth throttling mechanisms to ensure bus and memory interferences bounding. In [32], the authors propose to isolate bus resources when hard real-time applications execute, thus, giving them exclusive access to the bus. In [33], critical applications are mapped to a single core, leaving the other cores to use by best effort applications.

In [21], the authors design a resource management framework based on virtualization to leverage cache partitioning and memory bandwidth limitation. The approach is based on the use of virtual CPU (vCPU).

Execution models were introduced to manage bus and memory interferences. In [34] and [35] the authors use the Acquisition Execution Restitution model (AER). The objective of this method is to separate the computation (Execution phase) from memory accesses (Acquisition and Restitution) during run-time and schedule the memory phases so only one partition can be in those phases at a time. Thus, the RTOS schedules access phases to ensure no contention occurs.

Other works rely on hardware mechanism to reduce, bound and remove interferences. In [24], [36], [37] and [38] memory controllers and arbiters are proposed to improve predictability while increasing the performance. However, these approaches are not compatible with COTS hardware as it requires additional hardware that is not certifiable in some cases. In [39] the authors study three techniques to manage memory bandwidth: thread packing, clock modulation and Intel MBA technology.

We cannot apply the exclusive use of resources at a given time such as the method proposed in [32]. We consider all the partitions in the system as highly critical. Applying this method would be useless as only one application could execute at a time. The approach provided in [30] reduces the overhead and improves the performance but we cannot afford to change the criticality of the applications nor provide different execution modes. Unfortunately, we cannot rely on the additional hardware components used in [40]. Moreover, contrary to the proposed method, isolation must be maintained at any moment in the execution window. Finally, our objective is to reduce applications WCET contrary to optimizing the average performance. Work such as [39] does not provide strict isolation of resources and thus, makes certification more tedious. In [3] the Single-Core Equivalence principle was proposed. We base our work on this approach and refine it to enable easier integration with non-virtualized environments. While virtualization allows a better management of resources, this method is not applicable in our context. We aim to rely on a bare-metal RTOS that does not allow virtualizing the system's resources and to reduce the overhead introduced by these techniques.

Using the AER model is not applicable in our context as we need to keep compatibility with legacy applications that do not make use of such model.

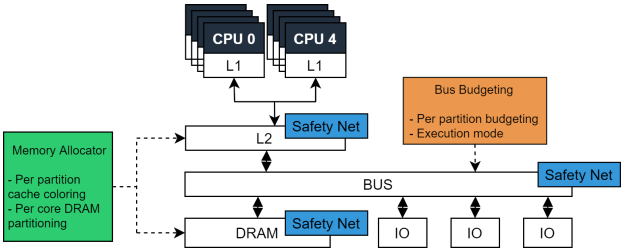

Fig. 2. Memory hierarchy with key points where our management framework applies.

In [41], the authors show that per application budgeting increases the system's performance and scheduling feasibility. In this paper, we provide an extension to MemGuard that allows per application budgeting instead of the per core budgeting previously allowed by MemGuard.

In this paper, we choose to use set based cache partitioning alongside with DRAM bank partitioning. Both approaches are portable to any architecture with virtual to physical address translation and are applicable to COTS hardware. For the same reason, we choose to extend the MemGuard approach that relies on COTS available features such as PMC (Performance Monitoring Counters) [5]. To enable portability, our solution can be extended for cache way partitioning use.

IV. Global Memory Manager

In this section, we present our centralized memory manager. This module extends the state-of-the-art approaches to ease their certification in the context of aerospace systems. Figure 2 provides an overview of the system's architecture with key points where our method applies. While the memory allocator manages how shared caches and DRAM is allocated across partitions, the bus budgeting module monitors and controls bus usage by the partitions. Finally, the safety net module is scattered across the memory hierarchy, to monitor the system's health. In case of an error, the safety net triggers an error to the heal monitoring manager. Throughout this section, we explain in detail each module and how they interact with each other.

A. Memory Management

Three main approaches exist to partition the cache. Way partitioning allows segregating caches in ways, by allocating them across cores. Set partitioning, also named cache coloring, profits from the virtual to physical translation mechanism to arrange the memory layout and allocate cache sets to different applications. Finally, line partitioning allows allocating cache lines to applications. Line partitioning is available when both set and way partitioning are possible. Set partitioning is privileged over way partitioning as it allows more cache partitions and is easily portable to different architectures. It is worth to be mentioned that our approach can be extended to use way or line cache partitioning with few modifications.

As discussed in [8], cache coloring (set partitioning) may interfere with memory coloring. Figure 3 shows the different memory layouts encountered during our study on architectures.

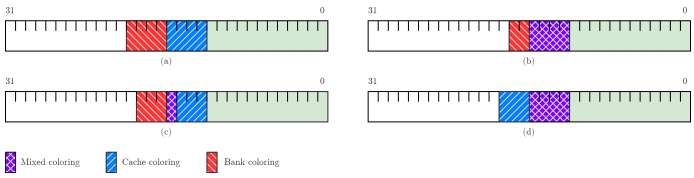

Fig. 3. Memory coloring layouts found in 40 different architectures. (a) allows separate allocation of banks and cache partitioning, (c) show overlapping bits while (b) and (d) show that bank coloring completely overlaps cache coloring and vice versa.

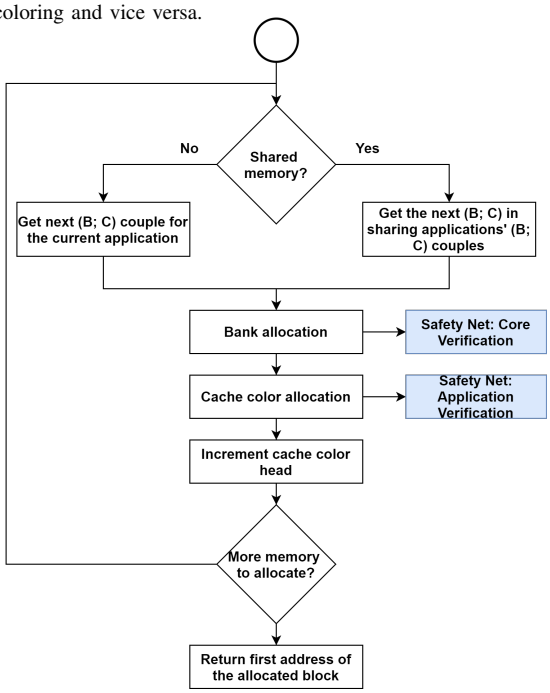

Fig. 4. Two-level memory allocator flow. We use the same process for both private and shared memory.

[8] also discuss the bank bit randomization technique used in modern processors. Such process is not present in all architecture but should be considered by the system designer when studying the memory address layout of the system. Our allocator takes into account overlapping bits in memory coloring to ensure complete isolation of the resources.

We propose a two-level memory allocator. First the allocator selects a memory bank based on the core the application is running on. In our design, we allocate banks on a per-core basis while allocating cache partition on per-application basis. Using private cache partitions for each applicative partition allows the systems to skip cache flush and invalidate when switching between partitions as recommended by the CAST-20 [42]. This method allows us to use cache more efficiently while reducing the partition switch time.

We define an allocation unit by the couple $(B; C)$ where B is the bank identifier and C the cache color. An application can have one or more assigned $(B; C)$ couples. Shared memory regions (between application on the same or different cores) use the $(B; C)$ couples of all applications sharing the region. Figure 4 depicts the execution flow of the allocation process. Our method ensures compatibility with all layouts depicted in Figure 3. It is also extensible to other future layouts. The safety net is explained in the next section of this paper.

When applications' private and shared memory have multiple allowed couples, the allocator select the $(B; C)$ couple to evenly distribute the workload on all available memory.

Once the algorithm selected the memory bank, the allocator provides one or more cache colors to the application. Each memory bank contains at least one cache color. We compute the number of cache colors per bank by subtracting the number of overlapping bits between the cache color bits and the bank color bits to the number of cache color bits.

To provide a faster allocation, we do not rely on linked list to represent free memory. We represent the memory banks with the *bank_color* structure.

```
struct bank_color {
    uint8_t  bank_id;
    uint32_t free_mem;
    ptr_t    cache_color_head[nb_cc];}
```

In this structure, the *cache_color_head* array represents the next free address in memory for a given cache color. We also use *nb_cc* to represent the number of cache color associated with the current bank. Using this structure, the allocation process is straightforward. When allocating a page to an application, the allocator sets the cache color head referred by $(B; C)$ to the next free page in $(B; C)$.

B. Bus bandwidth budgeting

We propose to extend the MemGuard [5] approach by adding the following features: (1) Per application budget allocation: each application receives a static amount of budget for the duration of its time window in the MAF; (2) An initial reclaiming [1] pool budget to accommodate with the scenario where all applications are critical; (3) Introduction of four application modes that we present later in this section;

1) Reclaiming modes: The computation of the initial reclaiming pool size as well as the budget allocation are out of the scope of this article and can be computed using the method proposed in [41].

To accommodate with the critical nature of ARINC-653 systems and render our approach more flexible, we propose to classify applications in four categories to extend the reclaiming feature:

- **Full reclaiming** mode, which allows an application to reclaim budget and provide budget to the reclaiming pool. Soft real-time partitions can use this mode. We also propose to further constrain the budget prediction to provide bounds on the budget removal of the application.
- **Greedy** mode for which an application does not provide any budget to the global budget pool but can reclaim budget from this same pool. This is suitable for any hard and soft real-time partitions, however, hard real-time applications are more prone to use this mode.
- **Altruist** mode, which allows an application to provide budget to the global pool but not reclaim budget from this pool. This mode is suitable for best-effort applications which can run in degraded mode.
- **Strict** mode, which forbids budget reclaiming and providing budget to the global pool.

[1] Reclaiming budget means that an application can request more budget than it was allocated by the means of a shared budget pool.

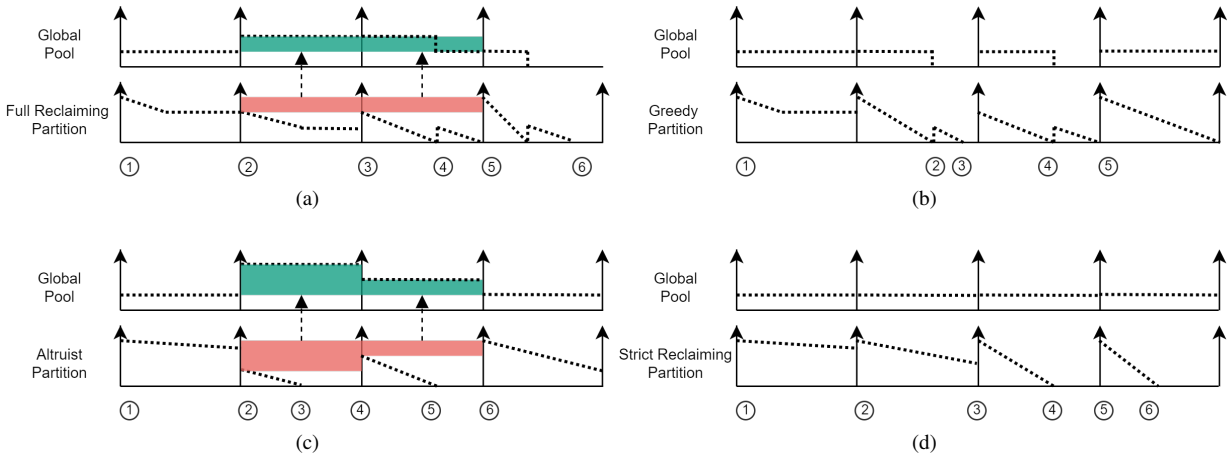

Fig. 5. Partitions' scheduling behavior when using full reclaiming mode (a), greedy mode (b), altruist mode (c) and strict reclaiming mode (d).

Figure 5 (a) depicts the execution of a partition under full reclaiming mode, which means that it can reclaim budget from the global pool and provide budget to it. At time ① the partition starts executing. At the end of the budgeting periods ② and ③, it was predicted that the partition will use less budget than initially provided. Budget is removed from the partition to provide it to the global pool. At ④ the partition exceeded its budget and reclaimed a fixed amount of free budget from the global pool. At ⑤, the does not remove budget from the partition. Finally, at ⑥ the partition exceeds its budget but the global pool is empty, the partition is removed from execution until the next timer's period.

Figure 5 (b) shows the execution of a partition under greedy mode. This time the partition's budget is exceeded at ②, ③ and ④ and is replenished at ② and ④ by reclaiming budget from the global pool and at ① and ⑤ from the bus budgeting replenish server. In greedy mode, no prediction is made and the partition does not provide any budget to the global pool.

Similarly Figure 5 (c) shows the execution of a partition under altruist mode. The budget can only be replenished by the bus budgeting server (at times ①, ②, ④ and ⑥). The partition can provide budget to the global pool and gets removed from execution when it exceeds its budget at times ③ and ⑤.

Finally, Figure 5 (d) depicts the execution for a partition using strict mode. Here, the partition does not give any budget to the global pool but also cannot reclaim any. The only replenish events occur at ①, ②, ③ and ⑤ and the partition is removed from execution at times ④ and ⑥.

The proposed reclaiming modes allow a complete flexibility on the system bus management. The system can change the mode at any moment and for any partition during the execution of the RTOS. Changing the reclaiming more is done through an API proposed by the RTOS. This API allows reconfiguring the budgeting mode while ensuring highly critical partitions are still correctly isolated. The API can be called at any moment by the partition during its execution.

Algorithm 1 shows the execution flow of the periodic server interrupt. This process is executed before restoring the application's context. Algorithm 2 depicts the process executed when an application exceeds its allocated budget. As proposed in MemGuard, when all cores become idle because of budget exceeding, the mechanism will reschedule all applications. Finally, we explain the $HealthCheck$ block in the next section and refers to the safety net feature proposed by the CAST-32A document.

Algorithm 1 Periodic server timer handler function.

```
1: function TIMERHANDLER
2:     App = ScheduledApp()
3:     nextBud = App.initBud
4:     if App is (full reclaiming or altruist) then
5:         nextBud = PredictNextBudget()
6:         AddSlackBudgetToPool(App.initBud - nextBud)
7:     end if
8:     if HealthCheck() is Failure then
9:         RaiseHMError()
10:        Return
11:    end if
12:    UpdatePMC(nextBud)
13:    UnlockCore()
14: end function
```

Algorithm 2 Interrupt handler executed when an application exceeds its budget.

```
1: function PMCHANDLER
2:     App = ScheduledApp()
3:     if HealthCheck() is Failure then
4:         RaiseHMError()
5:         Return
6:     end if
7:     if App is (full reclaiming or greedy) then
8:         nextBud = ReclaimBudget()
9:         UpdatePMC(nextBud)
10:        Schedule(App)
11:    end if
12:    if AllCoresIdle() then
13:        ForceResetTimer()
14:    else
15:        ScheduleIdle()
16:    end if
17: end function
```

In our context, we consider systems with hard real-time applications only. We cannot afford removing budget from a partition using s. Thus, we introduce an initial reclaiming pool containing free budget for any application. The global pool contains budget that the partitions can reclaim when

their budget is exceeded. This global pool is replenished at each budget server's tick. This approach allows improving the overall system's performance as presented in our result section. The benefits of this method are presented in the results section.

Both the periodic replenish server and the PMC interrupt handler are integrated from scratch in the RTOS. We have chosen to implement the different modules from scratch and not use a patched version of MemGuard because the architecture of an ARINC-653 compliant RTOS greatly differs from the Linux kernel. The periodic replenish servers consists in an interrupt service routine called every time the global timer triggers its periodic interrupt. In the same manner, when the PMC detect that the budget is exceeded, an interrupt handler is called and an idle task is scheduled to replace the running application as presented in Algorithm 2.

V. Safety Net

To ease the certification process, we define a set of rules that should always be verified during the system's execution. Apart from the CAST-32A guideline, we provide a way to ensure synchronization between cores regarding the MAF timings.

A. Core Verification

The RTOS architecture we use offers an Asymmetric Multi-Processing paradigm to manage the CPU. In this architecture, each CPU executes a separate instance of the RTOS. Small shared memory region enables the inter-core communications.

Each CPU has its own timer and uses a tick-less scheduler. This means no periodic tick is present in the system and timers interrupt occur only when needed (for instance, when the next application should be executed). To ensure synchronization, we add constraints to the MAF length. All CPUs can have different MAF length; however, we define a global hyper period based on the least common multiple Q between all CPUs MAF.

We define q_i the length of the MAF M_i defined for CPU i. M_i will be executed $\frac{Q}{q_i}$ times before reaching a synchronization point. When reaching such point, the CPU waits for the synchronization barrier release. When all CPUs reach the barrier, the kernel releases the CPU and immediately start a new MAF. This process corrects the drift between CPU clocks while ensuring real-time constraints are met on all cores. Our results show that the overhead introduced by the synchronization mechanism does not break any real-time constraint nor introduces deadline miss in the system. We provide the overhead considerations and experiments in Section 6.3. Q may be further constrained to reduce the drift between CPUs clocks. The smaller Q is, the faster the synchronization module correct the time drift.

B. Runtime Verification

To ensure safety of the proposed memory manager, we define a set of constraints that are verified at critical points during execution. Table II defines the set of variables we use to formalize the system. We further define the following rules:

TABLE II
List of variables associated with safety net constraints

Q	System's hyperperiod, defined as the least common multiple between all CPUs MAF
q_i	The length of the MAF M_i defined for CPU i.
A_i	Application i
$\mathcal{C}_{Ai}$	The memory colors set allocated to the application i.
A_i^C	The CPU identifier where A_i is mapped.
A_i^B	The bus bandwidth budget allocated to A_i.
A_i^M	The maximum number of bus access done by A_i during a single MAF.
$Acc(A_i, t)$	Accessed number done by an A_i at time t.
$\mathcal{S}_i^C$	The memory colors set allocated to the shared memory region i.
$\mathcal{S}_i^A$	Applications set sharing the memory region i.
$G(t)$	The free budget in the global pool at t.
$R(A_i, t)$	1 when A_i executes at time t, 0 otherwise.
B_{max}	The maximal bus bandwidth.

TABLE III
Formalized safety net rules

Rule 1	$A_i^C \neq A_j^C \Rightarrow \mathcal{C}_{A_i} \bigcap \mathcal{C}_{A_j} = \emptyset$
Rule 2	$\mathcal{S}_i^C \subseteq \bigcup_{\forall A_j \in \mathcal{S}_i^A} \mathcal{C}_{A_j}$
Rule 3	$\forall t \in [0; Q], G(t) + \sum_{\forall A_i} A_i^B \times R(A_i, t) \leq B_{max}$
Rule 4	$\forall A_i, \sum_{t=1}^{q_{A_i^C}} Acc(A_i, t) \leq A_i^M$

Rule 1: At any time during the system's execution, co-running applications must not share memory or cache color.

Rule 2: Shared memory regions color set must only contain colors allocated to the applications that share this region.

Rule 1 and rule 2 are verified during the system startup when allocating memory. It is further verified during a page fault, when the RTOS handles the page fault, it ensures that the address generating the fault is mapped to a physical address in the application's colors set. This rule does not apply to explicitly shared memory.

Rule 3: At any time t during execution, the sum of the bus bandwidth budget of the executing applications and the global budget pool must not exceed the maximal bandwidth permitted by the bus.

Rule 4: For safety purpose and fault detection, we define a maximum number of bus access for each application and IO in the system. The system should never reach this number to ensure the system's stability.

To verify rule 3, we add the *HealthCheck* function in Algorithm 1. Every time the system changes the budget configuration, the health monitor ensures that rule 3 is satisfied. If not, the health monitor handles the error. Rule 4 is verified in Algorithm 1 when handling applications budget overflow.

B_{max} can be provided by the processor's manufacturer or can be empirically measured with benchmarks. In this paper, we rely on the work presented in [43] to compute the value of B_{max} for our architecture.

VI. Results

In this section we present the study of our global approach and compare it to the current literature.

Our test bench comprises an NXP T2080RDB-PC board with four PowerPC e6500 cores running at 1.8GHz. We rely on a proprietary ARINC-653 RTOS (M-RTOS [44]) to

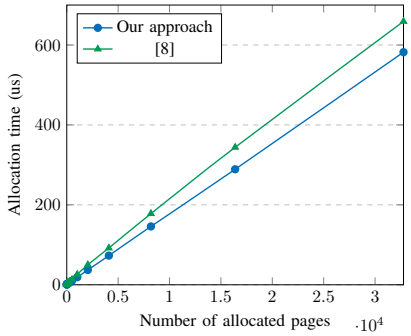

Fig. 6. Allocation time for different allocation sizes. The results show that our allocator allows faster allocation time.

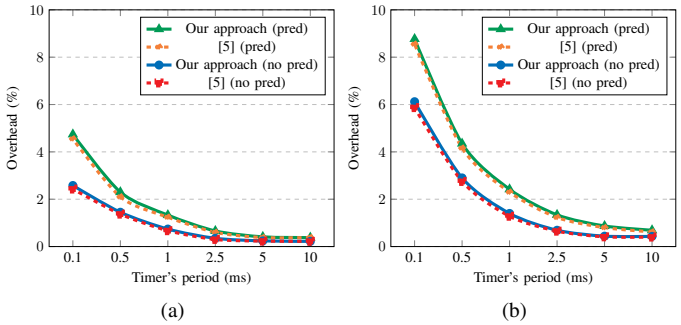

Fig. 7. Bandwidth limitation mechanism overhead depends on the timer's period.

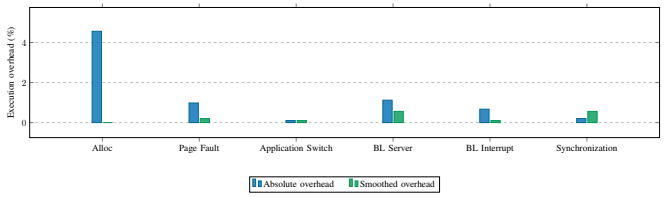

Fig. 8. Absolute overhead of the safety net compared to the smoothed overhead introduced at run-time.

retrieve our experiments results. We study the system using the TACLeBench [45] benchmark suite. We do not show results for the safety net other than its overhead. This is because the safety net is a part of the environment that is evaluated. All measures and results presented here are conducted with the safety net enabled.

A. Memory Allocation Overhead

We compare the execution time of our memory allocator for different allocation sizes to the one proposed in [8]. Our allocator initialization time is 2 us compared to 7120 us for the one proposed in [8]. This is because we do not have to create a linked list of available pages at initialization. Furthermore, we do not have to maintain this list during allocation. Figure 6 shows the allocation time reduction that benefits our method.

Memory allocation only occurs at the system's startup, which implies that the allocator's overhead only impacts the system's performance during boot time. In our test environment, memory allocation represents 9.39% (2.04ms) of the total startup time (21.74ms), which justifies the need to reduce its impact as much as possible. Our allocator reduces the startup time of the RTOS we use in our experiments by 24.8% (21.74ms) compared to the method proposed in [8] (28.91ms). After allocation, the cache partitioning technique has no overhead. Indeed, it relies on the MMU to perform a controlled translation of addresses. This process also occurs when not partitioning the cache.

B. Bandwidth Limitation Overhead

Our second experiment evaluates the overhead introduced by the bandwidth limitation module. We compare our implementation that adds reclaiming modes with the overhead introduced by MemGuard. Figure 7 present the overhead of our bus bandwidth management technique.

We based our test environment in the same context as what is proposed in [5]. To measure the overhead introduced by bandwidth limitation mechanisms, we use the following settings:

- For server's timer overhead, we disable bandwidth limitation mechanism by disabling PMCs interrupts. The rest of the mechanism is enabled. Figure 7 (a) shows our results.
- For server's timer overhead and budget exceeding mechanism, we enable bandwidth limitation mechanism by enabling PMCs interrupts. We also make sure that the application only exceeds its budget once every server's period expires. The RTOS directly reschedule the application once its budget is exceeded to measure only the limitation mechanism's overhead. Figure 7 (b) shows our results.

We compare our approach when using a predictor to add budget to the global budget pool and when not using any predictor. Results show that our extension of MemGuard to support per applications budgeting as well as reclaiming modes adds a small overhead (less than 0.5%). Our experiments show that the bus usage predictors introduce the most overhead.

Finally, we studied the impact that the mode change API has on the execution time of partitions. The mode change was implemented to only impact the calling core. Thus, this API call is no different than any system call made by the user to the RTOS (e.g. mutex acquisition, display print, etc.). The API performs a constant number of accesses to the memory. Based on the measurements of the execution time of common API and system calls in the used RTOS, changing mode is 4.6 times faster than a semaphore signaling, 8.5 times faster than getting the current partition's status and takes sensibly the same time as retrieving the current process's identifier.

C. Safety Net Overhead

We further study the impact of the safety net on the performance. We provide two metrics to express the overhead. We define the absolute overhead, which gives the absolute execution time added by the safety net. We also define the smoothed overhead, which describes the impact of the safety net on the applications execution time. It is important to note that the safety net overhead should be reduced as much as possible at its routines frequently executed in the system (e.g., TLB miss handlers, context switch, etc.).

Figure 8 presents the absolute and smoothed overhead of the safety net. *BL* stands for "Bandwidth Limitation". While we manage to maintain the overhead under 1% for most safety nets, the allocation safety net (Rule 1 and Rule 2) overhead reaches 4.56%. However, this overhead only occurs during the system startup and slows down the boot process by 1.5%. Therefore, we do not provide any allocation smoothed overhead metric, as it is nonexistent.

The time synchronization overhead presented in Figure 8 refers to the overhead added by the core MAF synchronization mechanism. Two factors cause this overhead. First, the synchronization routine itself must access shared data atomically to know the state of the synchronization barrier. The second factor comes from the potential synchronization of the cores. The more a core drifts, the more it is prone to reach the synchronization barrier late. In this test, the synchronization primitive causes the absolute overhead while the smoothed overhead accounts for the total overhead (synchronization primitive and drift of the cores clocks). The second factor is the more impacting, hence the bigger smoothed overhead. The results were gathered by measuring the number of cycles the cores stay in the synchronization routine at every MAF renewal and compared to the number of cycles the complete MAF renewal routine takes. The synchronization happens at most once every MAF start. Current timing analysis methods can take this overhead into account in the MAF renewal time. The overhead added by the synchronization was measured to be 0.56% of the MAF renewal time in the worst case. To gather the results, we measured 100,000 MAF renewal.

D. Scalability of the solution

The partitions-related safety net as well as the bus bandwidth limitation mechanism are designed to be independent for each core and are based on an AMP RTOS architecture. The cache coloring run time mechanism solely relies on the MMU and, thus, is independent of each core that has a private MMU. By construction, those mechanisms are not impacted by the number of cores in the system.

The startup memory allocation mechanism is impacted by the number of cores in the system. Memory allocation must keep global structures to ensure correct memory allocation between cores, thus synchronization primitives are used to ensure exclusive use of these structures. The worst case happens when all cores allocate memory at the same time. To access the shared data structures, a core might wait until all other cores have finished their allocation. Thus, the allocation time can be multiplied linearly by the number of cores in the system. To validate our assumption, we artificially created such a case by adding a barrier before each core allocates the partitions' memory. Each core has the same amount of memory to allocate. Our experimental measures show a maximal allocation time of 233μs on 1 core, 429μs on 2 cores, and 849μs on 4 cores. Finally, the core synchronization mechanism itself is not impacted by the number of cores. However, the time waited for all cores to reach the barrier may differ and is likely to increase with the number of cores. The equation $\sum_{i=1}^{N} D_i$ provides a way to compute the worst synchronization time where N is the number of cores and D_i is characterized by the amount of time the clock of the core i drifts between two synchronizations.

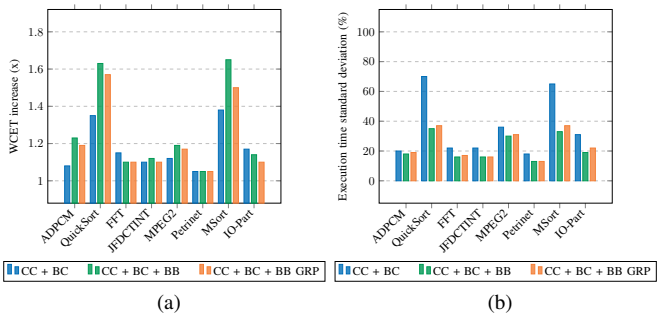

Fig. 9. WCET (a) and predictability improvements (b) provided by the memory manager. We compare the proposed metrics when using cache and bank coloring (CC + BC), bus budgeting (CC + BC + BB) and the initial reclaiming pool (CC + BC + BB GRP)

E. Effect on Predictability

We validate our approach by studying the predictability of eight concurrent applications. We define the predictability of an application based on the standard deviation of the execution time. We compare the predictability with interference management to the predictability without interference management. A lower value means the partitions' execution time is more predictable.

To approximate the measured WCET[2], we analyze each application under the same input data for all executions. We further executed each application multiple times until no higher execution time is detected after 1000 executions. In a safety-critical context, a more formal approach should be used, however, timing analysis in presence of interference in multicore system is not yet attainable and provides results that are too pessimistic [35].

This method provides a way to analyze the impact our work has on the applications' actual WCET. The WCET increase entailed by the interference mitigation mechanism is compared to the applications' WCET measured with a single-core setup, where only one application exclusively uses the system's resources. The lower the WCET increase, the better our method performs.

Finally, the bus budgets as well as the initial reclamation pool size are computed using the method proposed by [41] where an ILP formalism to the budget allocation that fits our experimental environment is defined.

Figure 9 (a) shows the relative increase of measured WCET compared to the WCET measured in single-core execution. Lower values are better and means the application suffered less slow down because of the interference mitigation means. We used 8 applications from the TacleBench suite and co-run them on 2 cores. Interferences are naturally introduced by the memory usage of each application. Table IV sump up our test

[2]We use a measured approximate WCET because current state of the art in multicore WCET analysis does not provide precise means of analysis in presence of interference. Our work is a step towards the use of conventional analysis methods, but other interferences exist and need to be addressed before being able to use regular analysis methods.

TABLE IV
TEST ENVIRONMENT APPLICATIONS

Application	Description	Core
ADPCM	Analog to digital filter Memory intensity: medium	0
QuickSort	Array sorting algorithm Memory intensity: high	0
FFR	Fast Fourrier Transform implementation Memory intensity: medium	0
JFDCTINT	Forward discrete cosine transform implementation Memory intensity: low	0
MEPG2	MPEG2 manipulation Memory intensity: medium	1
Petrinet	Petrinet simulation implementation Memory intensity: low	1
MSort	Array sorting algorithm Memory intensity: high	1
IO-Part	Application performing IOs on the serial port Memory intensity: high	1

payload. The period of each partition on the core 0 is 50 ms while periods for the core 1 are 75 ms. This ensures that with enough executions of the system, all partitions of the core 1 are co-run at least once with all partitions of the core 0. Our approach reduces the impact of isolation on the WCET while drastically increasing the predictability. In Figure 9 (b), the predictability corresponds to the standard deviation of execution time where 100% is the standard deviation in multi-core processors with the same setup but without interference mitigation. We show that in our test bench, we reduce the standard deviation of execution times by 68.1% on average.

Isolation is ensured by design. When using cache coloring, shared caches are strictly segregated into different areas and partitions using strictly different cache colors cannot share any cache area. It is the same for memory banks. Bus budgeting will also detect when a budget is exceeded and stop the partition at the very instruction where the budget is exceeded. Thus, isolation on cache and bus is ensured. However, in multicore systems, far more interference channels exist. For instance, cache coherence protocols will introduce delays when in need to update private caches [2]. Other hardware mechanisms such as the Miss Status Holding Registers (MSHR) will create interferences [46]. Thus, is it impossible to reach the same predictability in multi-core systems compared to single-core ones. Our objective is to remove the most critical one to lower the analysis pessimism and further conduct the system's analysis considering the remnant interferences.

It must be noted that the increase in predictability as well as the increase of the WCET is not the same for all applications. Indeed, in the system, some applications are more prone to interferences (usually applications that use more memory). Thus, memory-intensive applications (in our experiments MSort, QuickSort) will see their execution time increased more than other applications because of the resource restriction cache coloring and bus budgeting apply to them. For the same reason, the increase of the WCET is not proportional to the decrease of the standard deviation of execution time.

We also show that for highly critical system with only hard real-time applications, the initial reclaiming pool allows improving the overall system performance. Based on experiments with the proposed benchmark, we use an initial global reclaiming pool containing 10% of the maximal bus budget. This configuration yields to the best performance by providing additional budget for hard real-time applications using full-reclaiming and greedy modes while avoiding budget waste. Our study shows that while using an initial global reclaiming pool slightly reduces predictability, it allows to reduce the WCET and reduce the impact of bus budgeting.

Finally, it is important to discuss the tradeoff between WCET increase and predictability. An increase of less than 2 on dual-core architecture means that the system will be able to host more partition than a single-core system. Similarly, an increase of less than 4 on a 4-core architecture would mean the same. The goal of our interference mitigation means is to reduce the WCET impact as much as possible to ensure isolation but also allow more petitions to be hosted by the system. Based on our results, the method we propose increases applications' WCET by 22.3% on average on 2 cores, which is deemed acceptable to provide an increase of predictability of 68.1% on average. We also applied our approach on a 4-core setup, which yielded the same results, following the scalability analysis we provide in Section 6.5. For the sake of space, we do not present these results in this paper.

VII. CONCLUSION

Safety-critical systems rely on partitioned system to ensure complete isolation between applications. However, with the advent of multi-core processors, such isolation is impossible when using regular resource management methods. The contention on resources shared between the different cores generates interferences. Interferences are non-deterministic delays in execution time that prevent certification of safety-critical systems. The delay interferences introduce at run-time cannot be predicted. Thus, when measuring the Worst-Case Execution Time of applications, one must assume the worst case when sharing resources with other applications. This yields to overly pessimistic results that render the system unusable.

Research in the domain proposes multiple solutions to isolate different components and improve predictability. However, such solutions are not always applicable to safety-critical systems, where the configuration and implementation of interference mitigation methods must be deterministic and known during system design to allow certification.

In this paper, we presented a statically configurable memory manager that isolates the memory hierarchy between applications. Our approach enables to mitigate shared cache, DRAM and bus interferences with deterministic and certifiable methods. We further provided safety net rules to accommodate with the CAST-32A guidance document. Finally, we studied the impact of our method in terms of performance improvement and overhead. Our approach increases predictability by 68.1% while reducing the impact of mitigation methods.

As explained in Section 3, other interferences exist. Our approach restricted interference mitigation to shared caches, memory bank and bus interferences. Future work could integrate the use of channel partitioning in our mitigation framework and study its impact.

REFERENCES

[1] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, 2007, pp. 2.A.1–1–2.A.1–10.

[2] A. Löfwenmark and S. Nadjm-Tehrani, "Challenges in future avionic systems on multi-core platforms," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, Nov 2014.

[3] L. Sha *et al.*, "Single core equivalent virtual machines for hard real—time computing on multicore processors," 2014.

[4] C. A. S. Team, "Cast-32, multi-core processors," USA, November 2016.

[5] H. Yun *et al.*, "Memory bandwidth management for efficient performance isolation in multi-core platforms," *IEEE Transactions on Computers*, vol. 65, no. 2, feb 2016.

[6] ARINC, "Arinc specification 653: Avionics application software standard interface," Maryland, USA, 2015.

[7] S. P. Muralidhara *et al.*, *Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning*, 2011.

[8] N. Suzuki *et al.*, "Coordinated bank and cache coloring for temporal protection of memory accesses," in *Proceedings - 16th IEEE International Conference on Computational Science and Engineering, CSE 2013*, 2013.

[9] S. Mittal, "A survey of techniques for cache partitioning in multicore processors," *ACM Comput. Surv.*, vol. 50, no. 2, May 2017.

[10] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[11] R. SC-205, "Do-178c, software considerations in airborne systems and equipment certification," 2011.

[12] G. Kedar *et al.*, "Space: Semi-partitioned cache for energy efficient, hard real-time systems," *IEEE Transactions on Computers*, vol. 66, no. 4, 2017.

[13] J. Brock *et al.*, "Optimal cache partition-sharing," in *2015 44th International Conference on Parallel Processing*, 2015.

[14] N. El-Sayed *et al.*, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[15] G. Aupy *et al.*, "Co-scheduling amdahl applications on cache-partitioned systems," *The International Journal of High Performance Computing Applications*, vol. 32, 2017.

[16] G. Sun *et al.*, "Combining prefetch control and cache partitioning to improve multicore performance," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[17] J. Xiao *et al.*, "Cppf: a prefetch aware llc partitioning approach," 08 2019.

[18] M. Nejat *et al.*, "Coordinated management of processor configuration and cache partitioning to optimize energy under qos constraints," *ArXiv*, vol. abs/1911.05114, 2019.

[19] Z. Cui *et al.*, "A swap-based cache set index scheme to leverage both superpage and page coloring optimizations," in *Proceedings of the 51st Annual Design Automation Conference*, 2014.

[20] X. Jin *et al.*, "A simple cache partitioning approach in a virtualized environment," in *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2009.

[21] M. Xu *et al.*, "Holistic multi-resource allocation for multicore real-time virtualization," in *Proceedings - Design Automation Conference*, jun 2019.

[22] M. Hassan, "Managing dram interference in mixed criticality embedded systems," in *2019 31st International Conference on Microelectronics (ICM)*, 2019, pp. 253–257.

[23] L. Liu *et al.*, "Going vertical in memory management: Handling multiplicity by multi-policy," in *Proceedings - International Symposium on Computer Architecture*. Institute of Electrical and Electronics Engineers Inc., 2014.

[24] M. Hassan *et al.*, "A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, vol. 2015-May, may 2015.

[25] W. Ali and H. Yun, "RT-Gang: Real-time gang scheduling framework for safety-critical systems," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, vol. 2019, apr 2019.

[26] J. Fang *et al.*, "A memory scheduling strategy for eliminating memory access interference in heterogeneous system," *Journal of Supercomputing*, vol. 76, apr 2020.

[27] L. Liu *et al.*, "BPM/BPM+: Software-based dynamic memory partitioning mechanisms for mitigating dram bank-/channel-level interferences in multicore systems," in *Transactions on Architecture and Code Optimization*, vol. 11, no. 1, 2014.

[28] T. Ikeda and K. Kise, "Application aware DRAM bank partitioning in CMP," in *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*. IEEE Computer Society, 2013.

[29] J. Zhou and J. Wang, "Archsampler: Architecture-aware memory sampling library for in-memory applications," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018.

[30] M. A. Awan *et al.*, "Mixed-criticality scheduling with dynamic memory bandwidth regulation," in *Proceedings - 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2018*, jan 2019.

[31] J. Kim *et al.*, "Reducing Memory Interference Latency of Safety-Critical Applications via Memory Request Throttling and Linux Cgroup," in *International System on Chip Conference*, vol. 2018-September. IEEE Computer Society, jan 2019.

[32] H. Yun *et al.*, "BWLOCK: A dynamic memory access control framework for soft real-time applications on multicore platforms," *IEEE Transactions on Computers*, vol. 66, no. 7, jul 2017.

[33] A. Agrawal *et al.*, "Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, 2017.

[34] G. Durrieu *et al.*, "Predictable Flight Management System Implementation on a Multicore Processor," in *Embedded Real Time Software (ERTS'14)*, TOULOUSE, France, Feb. 2014.

[35] S. Park *et al.*, "Execution model to reduce the interference of shared memory in arinc 653 compliant multicore rtos," *Applied Sciences*, vol. 10, 04 2020.

[36] M. Hassan and H. Patel, "Criticality- and Requirement-Aware Bus Arbitration for Multi-Core Mixed Criticality Systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2016 - Proceedings*, apr 2016.

[37] A. Kostrzewa *et al.*, "Safe and dynamic traffic rate control for networks-on-chips," in *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2016.

[38] A. Dabaghi and H. Farbeh, "High performance and predictable memory controller for multicore mixed-criticality real-time systems," *IET Computers & Digital Techniques*, jun 2019.

[39] J. Park *et al.*, "HyPart: A hybrid technique for practical memory bandwidth partitioning on commodity servers," in *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, nov 2018, pp. 1–14.

[40] Y. Xiang *et al.*, "EMBA: Efficient memory bandwidth allocation to improve performance on intel commodity processor," in *ACM International Conference Proceeding Series*. Association for Computing Machinery, aug 2019.

[41] M. A. Awan *et al.*, "Uneven memory regulation for scheduling IMA applications on multi-core platforms," *Real-Time Systems*, vol. 55, apr 2019.

[42] C. A. S. Team, "Cast-20, addressing cache in airborne systems and equipment," USA, July 2003.

[43] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Tech. Rep., 1991-2007.

[44] M. S. . S. I. MSS, "M-rtos," 2020. [Online]. Available: https://www.mss.ca/m-rtos/

[45] H. Falk *et al.*, "TACLeBench: A benchmark collection to support worst-case execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, 2016.

[46] P. K. Valsan *et al.*, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.