Article

Execution Model to Reduce the Interference of Shared Memory in ARINC 653 Compliant Multicore RTOS

Sihyeong Park ¹ , Mi-Young Kwon ², Hoon-Kyu Kim ² and Hyungshin Kim ^{3,*}

¹ Department of Computer Science and Engineering, Chungnam National University, Daejeon 34134, Korea; sihyeong@cnu.ac.kr

² The 2nd R&D Institute, Agency for Defence Development (ADD), Seoul 138110, Korea; kmyadd@add.re.kr (M.-Y.K.); hunk@add.re.kr (H.-K.K.)

³ The Division of Computer Convergence, Chungnam National University, Daejeon 34134, Korea

* Correspondence: hyungshin@cnu.ac.kr

Received: 25 February 2020; Accepted: 30 March 2020; Published: 3 April 2020

Abstract: Multicore architecture is applied to contemporary avionics systems to deal with complex tasks. However, multicore architectures can cause interference by contention because the cores share hardware resources. This interference reduces the predictable execution time of safety-critical systems, such as avionics systems. To reduce this interference, methods of separating hardware resources or limiting capacity by core have been proposed. Existing studies have modified kernels to control hardware resources. Additionally, an execution model has been proposed that can reduce interference by adjusting the execution order of tasks without software modification. Avionics systems require several rigorous software verification procedures. Therefore, modifying existing software can be costly and time-consuming. In this work, we propose a method to apply execution models proposed in existing studies without modifying commercial real-time operating systems. We implemented the time-division multiple access (TDMA) and acquisition execution restitution (AER) execution models with pseudo-partition and message queuing on VxWorks 653. Moreover, we propose a multi-TDMA model considering the characteristics of the target hardware. For the interference analysis, we measured the L1 and L2 cache misses and the number of main memory requests. We demonstrated that the interference caused by memory sharing was reduced by at least 60% in the execution model. In particular, multi-TDMA doubled utilization compared to TDMA and also reduced the execution time by 20% compared to the AER model.

Keywords: avionics systems; real-time system; execution model; multicore architecture; interference analysis

1. Introduction

Owing to the increased use of sensors in avionics systems, the number of tasks to be executed also increased, and this requires more computing power. As a result, multicore architectures have been introduced in avionics systems [1]. This allows for the execution of complex tasks, but unexpected problems may occur because multiple cores share hardware resources. Safety-critical systems, especially avionics systems, require predictability and tight execution times. Commonly used avionics systems are integrated modular avionics (IMA) structures in which several systems are integrated into a single hardware component. The IMA structure provides time and space partitioning to ensure the execution of each system. The system is divided into objects called partitions, and the memory area is divided and allocated to each partition to ensure space separation. In addition, the defined execution time of each partition is guaranteed by using hierarchical schedulers. Multicore avionics systems apply asymmetric multiprocessing (AMP) in which each core executes independently to reduce inter-core

interference [2]. However, this AMP system also shares hardware resources, making it difficult to guarantee absolute execution independence.

To reduce interference from shared resources on multiple cores, methods for dividing shared resources, such as allocation of memory and input-output (I/O) to each core [3,4], or limiting the usage of hardware resources in each core [5,6], have been proposed. Further, resource-aware scheduling [7,8] has been studied to control the execution order of tasks and to prevent interference due to redundant shared resource access. Studies have also been carried out in a bare metal environment [9] or in a simulator [10]. Moreover, kernels have been modified to control hardware resources [5,11,12]. Avionics systems are very costly and take significant time to validate software to meet standards, such as ARINC 653 (AEEC, Annapolis, Maryland, 2019) [13] and DO-178C (RTCA, Washington, D.C 20036, USA, 2012) [1]. Therefore, modifying an existing system incurs additional costs. The execution models have been proposed to reduce interference without software modification. Execution models relate to task scheduling include time-division multiple access (TDMA) [14] and acquisition execution restitution (AER) execution models [9].

In this study, we implemented execution models in a commercial real-time operating system (RTOS) for avionics systems. We analyzed the interference due to memory sharing in a multicore architecture. In particular, we applied an execution model to determine how much the interference is reduced. Furthermore, we proposed a multi-TDMA model that considers the characteristics of the bus interface of the target board. Thus, we measured interference according to shared memory content with the benchmark proposed in our previous study [15]. In the preliminary study [15], we analyzed the memory interference according to the computational characteristics with MiBench automotive suite [16]. To measure interference, we measured the execution time, the L1 and L2 cache miss, and the bus interface unit (BIU) requests during benchmark executions. The objective of this work is to analyze the interference due to resource sharing between cores in AMP-based multicore avionics systems. Mainly, we focus on the interference due to cache and main memory sharing. The reason for focusing on interference due to memory sharing is that in ARINC 653, one hardware device is usually statically mapped to one partition. Therefore, even with AMP-based multicores, multiple partitions cannot access single hardware components simultaneously. However, each partition is allocated its own memory space but still uses the same hardware. Consequently, multiple partitions can access the memory at the same time, causing interference due to contention.

Experimental results show that when multiple cores access memory concurrently, the execution time increased by about 7% due to interference. We analyzed the correlation between the execution time, cache miss, and the BIU request and then confirmed that the L2 cache misses and the BIU requests had a significant effect on the execution time. When applying the TDMA and AER models, the TDMA model's execution time decreased by more than 60% due to the reduction of the L2 cache misses and BIU requests. The AER model reduced the interference rather than the contention but showed higher interference than TDMA. The AER model has more context switches as inter-partition communication for phase control. This caused more cache misses. Moreover, AER increased the execution time by 30% due to the control of execution flow between partitions on each core. The proposed multi-TDMA has an execution time similar to TDMA. Nevertheless, the L2 cache misses of multi-TDMA were about five times higher than that of TDMA, due to simultaneous memory access on two cores. Conversely, the BIU requests occurred similarly to TDMA, signifying that the execution time did not increase. In summary, we demonstrate that our proposed multi-TDMA can reduce the interference caused by memory sharing and increase multicore utilization.

The contributions of this paper to the literature are as follows:

- We analyzed the interference caused by memory sharing in a commercial operating system (OS) and commercial off-the-shelf (COTS) hardware adopted in avionics systems. We measured the interference by measuring the execution time, L1 and L2 cache miss, and the BIU request counts. Experiments have shown that L2 cache misses and the BIU requests have a significant impact on the execution time.

- We applied an execution model to reduce interference. Then, we propose a method to implement the execution model with pseudo-partition and message queuing of commodity OS, which has limited source code modification. Experimental results show that TDMA can reduce the execution time by reducing the interference due to memory sharing. In contrast, the interference of AER was reduced, but its execution time increased by about 20% due to phase control between cores.
- We propose a multi-TDMA model that takes into account the characteristics of the target hardware. We demonstrate that multi-TDMA can reduce interference due to memory sharing by 60%. Multi-TDMA has twice the utilization of TDMA and can reduce the execution time by 30% compared to AER.

The remainder of this paper is organized as follows. Section 2 describes the digital avionics system and the execution model used in this study. Section 3 discusses the methods used to implement the execution model in commercial RTOS and the proposed execution model. Section 4 shows the results of measuring the interference caused by memory sharing in the multicore architecture and the analyses of the reductions by applying the execution model. Section 5 discusses related studies that reduce the interference caused by hardware resource sharing in multicores. Finally, Section 6 provides the conclusion of the study.

2. Background

In this section, we discuss the avionics system and the execution model used in this work.

2.1. Avionics System

Until the 1990s, aircraft sensors and engine control functions were implemented as individual subsystems called line replaceable units (LRUs) [17] in avionics systems, as shown in Figure 1a. These LRUs consist of a federated architecture, connected via buses, such as, the RS-232, and 1553 analog buses. In a federated architecture, each LRU is connected to an external bus, resulting in slow communication between the LRUs. In particular, additional LRUs are needed to add functionality to avionics systems. For aircraft, only a limited number of LRUs can be added because they have space, weight, and power (SWaP) constraints. Therefore, to deal with more features in aircraft, the IMA architecture [17,18] has been proposed.

The IMA architecture integrates multiple LRUs of a federated architecture into a single hardware component with more powerful computing capability, as shown in Figure 1b. Therefore, the IMA architecture can solve the avionics systems' SWaP problem. Since the LRUs are on the same hardware backbone, the throughput is increased by communication over the external bus being reduced [19]. However, many systems operate on one hardware backbone, and it is necessary to guarantee the execution of each system. For this, the partitioning mechanism [18] following the ARINC 653 standard [13] is applied to avionics systems. Furthermore, virtualization-based methods, such as a hypervisor, provide legacy software compatibility with integrated hardware [20,21]. Hypervisor is also used to separate the running applications to ensure the security and real-time performance in cyber-physical systems [22].

The partitioning system allocates an object called a partition each subsystem. A partition consists of several processes that execute tasks [18]. The partitioning system supports spatial and temporal separation, as shown in Figure 2, to ensure the identical execution of multiple partitions on a hardware unit. As shown in Figure 2a, the memory is divided and allocated to each partition for spatial separation. Each partition has its memory region and cannot access the memory regions of other partitions. Memory access is controlled by a memory management unit (MMU) or a memory protection unit (MPU). Processes within a partition share the memory area allocated to the partition. The partitioning system hierarchically schedules partitions and their processes based on major and minor time frames, as shown in Figure 2b, for time separation. The major time frame is a scheduling sequence in which each partition runs periodically. Meanwhile, the minor time frame is the periodic scheduling sequence of the processes in each partition. The partition scheduler schedules partitions with policies such

as round-robin (RR). The process scheduler schedules processes with scheduling policies, such as first-in-first-out and earliest deadline first. The memory space, execution time, and execution period of partitions and processes based on ARINC-653 are defined before runtime. Therefore, neither configuration can be modified at runtime.

Figure 1. Architectures of the avionics system. (a) Federated architecture. (b) Integrated modular architecture.

Figure 2. Temporal and spatial separation of the partitioning system. (a) Spatial separation. (b) Temporal separation.

Recently, the amount of data to be processed in avionics systems has increased significantly. Moreover, they conduct complex operations, such as advanced signal processing. Multicore architectures were introduced into avionics systems to meet such requirements. Multicore architectures have the potential to increase throughput using the parallel processing of tasks or hardware resources. However, as shown in Figure 3, multiple cores share hardware resources (e.g., memory, I/O). In particular, hardware resource sharing among the cores can cause interference, such as increased task execution time [23]. Real-time and safety-critical systems, such as avionics systems, must deterministically bound the execution time [23–25]. Avionics systems divide the shared memory area and allocate it to each partition to reduce interference. Nevertheless, partitions are still in the same physical memory so interference can still occur. In the multicore IMA system, a model for defining and preventing conflicts between the cores has been studied to aid robust partitioning [26].

Figure 3. Memory sharing on multicore partitioning systems.

2.2. Execution Model

In this work, we apply an execution model to reduce the interference caused by memory sharing in avionics systems. Several methods have been proposed for reducing memory sharing interference. However, the software verification cost for the system and service in avionics systems is very high [27]. Therefore, we employed a model that eliminated interference without modifying the software.

Execution models are methods of defining the execution sequences of tasks. In this section, we describe two execution models that are applied to multicore avionics systems, as shown in Figure 4. Avionics systems set the execution order and time of partitions and processes before runtime, as defined in ARINC-653. Therefore, it is possible statically to design and verify the execution model.

Figure 4. Execution models. (a) Time-division multiple access (TDMA). (b) Acquisition execution restitution (AER).

TDMA [14,28] executes only one core at a time, as shown in Figure 4a. Consequently, even if resources are shared, execution interference does not occur among the cores. However, TDMA uses only one core at a time, thereby reducing the overall throughput.

In addition, AER [9,29,30] allows the parallel execution of tasks in each core to improve the limitations of TDMA, as shown in Figure 4b. Tasks are divided into three phases: acquisition, execution, and restitution (denoted by A, E, and R, respectively). The A phase brings data from the shared memory to the local memory, such as a private cache. The E phase performs operations with the data read from the previous phase, and the R phase stores the result in the shared memory. Thus, the E phase can execute concurrently with the A or R phases of other cores by accessing only the local memory. Theoretically, in the AER model, if the execution time of the E phase of one core is longer than the A and R phases of the other core, up to four cores can run simultaneously. However, legacy code may not be split into the A-E-R phases of the AER model.

The major time frames of the TDMA and AER models in a partitioning system are shown in Figure 5. Since AER has higher utilization than TDMA, it has a shorter major time frame. If the major time frame is shortened when performing the same task, the execution safety margin of the partition can be set up. Table 1 summarizes the TDMA and AER execution models.

Figure 5. Major time frame comparison of TDMA and AER execution models.

Table 1. Comparison of execution models.

	Execution Model	
	Time-Division Multiple Access (TDMA)	Acquisition Execution Restitution (AER)
Description	<ul style="list-style-type: none"> Only one core runs at a time 	<ul style="list-style-type: none"> Separates tasks into three phases (A-E-R) <ul style="list-style-type: none"> The A and R phases access the memory E phase operates using only the values stored in the local memory
Advantage	<ul style="list-style-type: none"> No interference with the use of hardware resources between tasks Code reuse is possible 	<ul style="list-style-type: none"> Allows parallel execution between cores <ul style="list-style-type: none"> Only E phase can be executed in parallel with other phases Higher utilization than TDMA
Disadvantage	<ul style="list-style-type: none"> Utilization is same as the single core 	<ul style="list-style-type: none"> Reuse of existing code is not possible

3. Execution Models for Multicore RTOS

In this section, we describe the RTOS for the experiment. We also cover the implementation of the TDMA and AER models described in Section 2.2 on this RTOS.

3.1. Multicore RTOS for Avionics Systems

In this study, we used VxWorks 653 (version 3.1, Wind River, Alameda, CA, USA, 2019) [31], a commercial RTOS applied in aircraft, such as the Boeing 787 Dreamliner. The structure of the OS is shown in Figure 6. VxWorks 653 is an AMP system. In AMP systems, each core operates independently and is also applied for robust execution in multicore avionics systems [1,2]. However, since each core shares memory, interference can occur between the cores despite using the AMP system.

VxWorks 653 is a hypervisor-based multicore OS called module operating system (MOS) [31], as shown in Figure 6. This OS is an AMP system; therefore it allocates the MOS for each core. The MOS runs in privileged mode, providing spatial and temporal separation of the system. The MOS consists of an MMU manager and a TLB handler to allocate memory space for each partition that is executing on the hypervisor. During system initialization, the MOS determines each partition and associated scheduling information based on predefined configurations. After being initialized, the MOS operates on an event basis. When events occur such as inter-partition communications within

a core or interrupts that send messages to a partition on another core, the MOS handles the requests and calls the inter-processor interrupt (IPI). This IPI is also a significant cause of cache misses.

Figure 6. The architecture of VxWorks 653.

Each core of the VxWorks 653 is composed of one or more partitions, as shown in Figure 6. Each partition consists of a partition operating system (POS) and one or more processes. The POS is a hypervisor guest OS and supports VxWorks 6.3. POS manages processes and provides APIs, such as buffers and blackboards, for intra-partition communication (inter-process communication). The processes are pre-allocated to the appropriate partition memory. Further, inter-process communication continues through the corresponding memory space.

VxWorks 653 uses hierarchical scheduling for the time separation of partitions and processes in the IMA structure. The MOS of each core manages the execution time of each partition. For hierarchical scheduling, this consists of a partition scheduler and a process scheduler. The partition scheduler schedules partitions to the RR policy based on the time slice. The process scheduler applies priority-based preemptive and RR scheduling. For memory separation in VxWorks 653, the MOS allocates a predefined memory area during the initial system configuration.

3.2. Implementation of Execution Models

The goal of this work is to reduce interference without altering the source code of existing systems. Commodity OSs, such as VxWorks 653, allow limited source code modification. Moreover, the recertification of modified avionics software can be costly. Therefore, we executed models without modifying the source code.

3.2.1. TDMA

Partition scheduling in VxWorks 653, does not allow the partition start time to be specified. That is, the partitions of each core are executed in parallel simultaneously. For TDMA model, partitions on each core should not be run concurrently. To address this problem, we employed a pseudo-partition as shown in Figure 7 to control the execution order.

A pseudo-partition is a partition with a single process that executes “while (1) {}”. Based on the predefined partition execution time for each core this pseudo-partition allows only a single core at a time access to the memory. Given that ARINC 653 defines the partitions’ execution period before runtime, the pseudo-partition’s execution period can be set according to the scheduling order. We established a pseudo-partition and heuristically confirmed that the executions on each core did not overlap. This implementation required additional memory space for the six pseudo-partitions to implement TDMA model on a quad-core processor. We used pseudo-partition to control the order of execution between partitions. However, it has the potential to run non-memory jobs instead of pseudo-partitions.

Figure 7. Pseudo-partition for execution flow control.

3.2.2. AER

When implemented, the AER model must control the execution order between processes in each core's partition. VxWorks 653 is an AMP system with all core partitions running independently. For inter-partition communication, the ARINC 653 application execution (APEX) service was used to control the process flow of executions in each partition, as shown in Figure 8a. We used message queuing for inter-partition communication. Message queuing operates by allocating memory space (channels) for a message in the memory of the MOS. The messages on these channels are communicated between partitions. With message queuing, the receiving partition waits for the message to be written to the channel. We used this characteristic to control the execution flow between partitions on each core.

Figure 8. Inter-partition flow control. (a) Inter-partition flow control using message queuing. (b) Code of inter-partition flow control using message queuing.

In this work, the AER model is implemented as shown in Figure 8b. Each partition's processes have phase functions within the main function. When executing the AER model on two cores, Core 1 (1) calls the message receiving function (*RECEIVE_QUEUING_MESSAGE()*) and transitions to the block state when the partition process is executed. Core 0 executes the A phase function, and when the execution ends, (2) sends a message to Core 1 (*SEND_QUEUING_MESSAGE()*). When Core 1 receives the message, it switches from the block state to the execution state and executes the A phase function. At this time, Core 0 executes the E phase in parallel with the A phase of Core 1. If the E phase of Core 0 is terminated, (3) calls the message send function. When the A phase task of Core 1 is completed, (4) sends the message to Core 0, and the E phase is executed. The E phase of Core 1 terminates, and (5) calls the message sending function for the R phase of Core 0. Core 0 executes the R phase, (6) sends a message to Core 1, and Core 1 executes the R phase. Until the R phase of Core 1 is completed in Core 0, as shown in Figure 8a the non-executing components pause. Therefore, an additional margin is required to set the process deadline for the partition.

When sending and receiving messages, the cache is polluted or flushed when the partition calls the MOS via the system and the context is switched. In particular, to transfer messages between cores, overhead occurs when MOS checks the number of the core to which the message is sent and then writes a value in the MOS memory area of the core.

To implement the AER model, we used the pseudo-partition applied to the TDMA model. Unlike TDMA, two cores can run concurrently, thus requiring four pseudo-partitions for the major time frame, as shown in Figure 9. In a previous study that implements the existing AER model [29], the last-level cache (LLC) is disabled so that if a miss occurs in the L1 cache, the main memory is

accessed directly. However, disabling the LLC was not possible in our setup. Therefore, we used an array smaller than the size of the L1 cache to minimize the L1 cache misses in the benchmark.

Figure 9. AER execution model with pseudo-partition.

3.2.3. Proposed Model

The memory bus of our target board has two coherences in one cycle meaning that two requests for the main memory can be processed at once. We propose a multi-TDMA model that can improve the TDMA and AER models using hardware characteristics. The goal of the proposed multi-TDMA is to increase utilization over TDMA and AER while allowing slight interference due to memory sharing. However, to satisfy the deterministic characteristics of the avionics system, the balance of utilization and interference should be considered.

Multi-TDMA runs two cores in parallel, as shown in Figure 10. This implementation of the multi-TDMA model reduces the number of pseudo-partitions by two compared to the TDMA model. With the same major time frame as the AER model, an additional safety margin in partition execution can be set. Furthermore, multi-TDMA can diminish memory usage by reducing the pseudo-partitions. Because with TDMA two cores access the memory simultaneously, more interference is expected to affect the execution time.

Figure 10. Multi-TDMA.

4. Evaluations

This section describes the results of measuring the interference due to memory sharing between cores by executing TDMA, AER, and multi-TDMA on the target hardware described in Section 3. Moreover, we describe the benchmark [15] for this experiment.

4.1. Measurement Environments

Figure 11 shows the NXP T2080QDS board (NXP, Eindhoven, Netherlands) used to measure the memory sharing interference between cores. The target board is equipped with QorIQ architecture-based quad-core PowerPC E6500 central processing unit (CPU). Each core has a 32 KB private data and instruction cache, and a 2 MB L2 cache is shared among the cores. The L2 cache is 16-way banked with 512 KB for each of the four cores. The main memory and the core transmit data through the CoreNet coherence fabric (eq. CoreNet BIU). The target board employs a 512 KB CoreNet platform cache and a pre-fetched L3 cache to improve access time to the main memory. The main memory is a DDR3 8 GB chip. The CoreNet BIU provides high concurrency by providing multiple and parallel address access. For this purpose, two coherence granules are processed per cycle.

Figure 11. Architecture of T2080QDS.

We measured the number of L1 data cache misses, the L2 cache misses, and the CoreNet BIU requests (L3 and main memory access) to identify the memory sharing interference. As shown in Figure 11, each core shares the L2 cache and LLC. Therefore, cache eviction may occur depending on the cache access of each core. This factor can cause inter-core interference. In particular, we measured the effect of inter-core interference on the partitions' process execution times. Each cache misses count was read from the core's performance monitor register (PMR). The PMR consists of a performance monitor global control (PMGC), a performance monitor local control register (PMLC), and a performance monitor counter (PMC). We set event flags for the L1, L2 cache misses rete, and BIU requests in the PMGC and the PMLC and the read values counted in the PMC during benchmark execution. The PMGC and PMLC can only be configured in hypervisor mode, while VxWorks 653's guest (POS) operates in guest-hypervisor mode and user mode. Therefore, since the PMGC is not set up in the partition process, it is necessary to set up the event flag in the initialization of MOS. The PMC is accessible in user mode. We used the *move instruction* of PMR (*mfpmr*) in the assembly code to access the values of the PMC. The latency of the *mfpmr* instruction was four cycles. The execution time of the benchmark was measured with the POSIX timer library *clock_gettime()*. We measured the corresponding cache misses only in Core 0 due to the limited the number of registers; the execution time was measured in the partitions of all four cores. All experiments were repeated a hundred times. In addition, only one process was allocated to each partition.

4.2. Benchmark

We experimented with the TDMA, AER, and multi-TDMA models applied the micro-benchmarks proposed in the previous study [15]. The benchmark followed the phase structure of the AER model described in Section 3.2.2, implemented with one main function and each phase function. Each phase function was called from the main function. The detailed benchmark operation is shown in Figure 12. The A phase read the array stored in shared memory and stored it in the local variable. Then, to minimize L1 cache misses, the array was accessed by 32 KB, the size of L1 on the target board. The E phase carries out a simple addition operation using local variables stored during the A phase. During the R phase, the value calculated in the E phase was updated in the main memory.

Before experimenting with the execution model, we measured the number of cache misses that occurred at each phase of the benchmark. The measured values for each phase are shown in Table 2. In the E phase, the L1 misses were approximately 15 times fewer than in the A phase, and 31 and 55 times less for the L2 misses and BIU requests, respectively. The cache misses that occurred in the E phase were due to the context switch caused by the calling of each phase function in the main function. The low number of L2 misses and BIU requests in the R phase is because the data is already stored in the cache during the A phase. These results show that the cache misses and BIU request counts in the E phase are acceptable.

Figure 12. Benchmark for evaluation.

Table 2. Benchmark measurement results per phase.

	Phase		
	A	E	R
L1 data cache misses	8802	583	8739
L2 cache misses	592	19	17
BIU requests	763	14	13

4.3. Experimental Results

To examine the interference reduction in the execution model, we measured the memory sharing interference among the cores. For this purpose, the benchmark was run concurrently on all cores without phase control and the memory was accessed from all partitions simultaneously, for the ensuing bottlenecks to affect the execution time. We measured the memory sharing interference by varying the number of cores accessing the memory concurrently. We also measured the number of cache misses for each benchmark phase function. We also set a 2 ms deadline for executing a process on each core resulting in the same for each partition. The interference measurement results, according to memory sharing are shown in Figure 13.

Figure 13. Benchmark measurement results. (a) Execution time. (b) Cache misses and bus interface unit (BIU) requests.

Figure 13a is a violin graph of the benchmark execution times with the width of each graph representing the distribution. The X-axis represents the number of cores, and the Y-axis is the execution time. When executing the benchmark on one core, the execution time was about 0.64 ms, and on four cores, it increased by about 7%. When running the benchmark on only one core, the variation in execution times was 0.01 ms. On 3 cores, the execution time variations were greatest, at 0.04 ms. This demonstrates that as the number of cores accessing the shared memory increases execution times become non-deterministic.

The cache misses measurement results of the benchmark are shown in Figure 13b, where the X-axis represents the variation in the number of cores, and the Y-axis represents the numbers of each item. The L1 cache misses occurred 18,000 times, regardless of the number of cores. At this time, most of the misses occurred in the A and R phases while accessing the shared memory. Approximately 1000 L2 cache misses occurred while executing the benchmark on one core and increased approximately 4.5-fold as the number of cores increased. The BIU requests showed a similar trend and increased more than seven-fold with the increase in the core count being similar to the L2 cache misses. As the number of concurrently running cores increased, the L2 cache misses and the BIU requests in the E phase increased. This means that the execution between processes in each core is not controlled, for it to be executed concurrently with the A and R phases of the other cores. The numbers of L2 misses, and BIU requests during the R phase were small since the data had already been cached in the preceding A and E phases.

Based on these results, we analyzed the correlation of each measurement item, as shown in Figure 14. Based on the measured values, we calculated the Pearson correlation coefficient using MATLAB. From this figure, it appears that execution times are interrelated with the L2 cache misses and the BIU requests. The L2 cache misses are affected by the number of cores running concurrently. Further, the BIU requests were found to be associated with L2 cache misses. In conclusion, the L2 cache misses should be minimized to reduce the impact of memory-sharing interference between cores during execution.

BIU Request	0.993	0.825	0.041	0.999	1.000
L2 Cache Miss	0.994	0.828	0.062	1.000	0.999
L1 Cache Miss	0.029	0.381	1.000	0.062	0.041
Elapsed Time	0.787	1.000	0.381	0.828	0.825
Number of Core	1.000	0.787	0.029	0.994	0.993
	Number of Core	Elapsed Time	L1 Cache Miss	L2 Cache Miss	BIU Request

Figure 14. Correlation by measurement elements.

4.3.1. Application of the Execution Model

Based on the results of the interference due to memory sharing in the multicores measured earlier, we apply each execution model to find interference reductions. To compare the results with the execution models, we set the baseline to run the benchmarks concurrently on four cores. The results of applying the TDMA and AER models are shown in Figure 15.

Figure 15a shows the execution time for each model. The TDMA model executes by core partition sequentially and similarly accesses the memory, so there is no interference. Therefore, the variation of the benchmark execution time was the least, and the execution time was reduced by about 0.1 ms, compared to the baseline. Conversely, the AER model had a 0.82 ms, execution time which is approximately 20% longer than the baseline and TDMA results. The increase in the execution time of the AER model is due to the waiting time of the main function while controlling the A, E, and R phases among the cores, as shown in Figure 15b. Moreover, the AER model takes time to send and receive messages to control the order of phases and the IPI overhead for this process. The baseline and TDMA only calls each phase function from the main function, so there is a slight overhead.

The measurement results of L1, L2 cache misses, and the BIU requests according to each execution model are shown in Figure 15c. TDMA execution model incurred the fewest L1 cache misses. As with

the previous results, most of the L1 cache misses occurred in the A and R phases in TDMA and AER. The L2 cache misses count was of the same order as the baseline, AER, and TDMA. The baseline had about 4300 L2 cache misses, mostly in the A phase when reading values from the memory. At this time, because there was no execution control for the processes of each core, the E phase was affected, and the L2 cache misses occurred. There were 2000 L2 cache misses when the TDMA model was run. This is caused by the cold cache when a value initially is read from the shared memory to the local memory. The AER model's L2 cache misses decreased by more than 60%, compared to the baseline. However, approximately 2000 additional L2 cache misses occurred in the E and R phases due to the message calls context switching and each phase control's send and receive functions. The BIU requests of each model appeared in the order of baseline, AER, and TDMA. In TDMA, the BIU requests were also lower because there were minimal L2 cache misses during the E and R phases. Consequently, the TDMA and AER models reduced interference. However, in the case of AER, 23% of the execution time was required for each phase control action.

Figure 15. Benchmark measurement results by the execution model. (a) Execution time. (b) Execution time by each phase. (c) Cache misses and BIU requests.

We set the partition execution times for all cores to 2 ms for the experiment. To implement each model, the major time frames of the baseline, TDMA, and AER were 2 ms, 8 ms, and 4 ms, respectively. The AER model increased the execution time of the benchmark, but the major time frame was shorter than TDMA by increasing the utilization of the multicore. In particular, the safety margin in the partition of each model was about 67%, 69%, and 60%, respectively. Consequently, higher safety margins can be obtained as memory interference is reduced.

4.3.2. Multi-TDMA

Our experimental results show that the TDMA and AER models can be used to reduce interference due to memory sharing. The TDMA model reduces interference by running only one core at a time on

multiple cores. Thus, TDMA has the same utilization as a single core. AER increased the utilization of multicores by splitting the task into three phases and executing both non-memory and memory phases concurrently. However, AER took about 20% more time to control each phase. We propose a multi-TDMA that runs two cores simultaneously, considering the shortcomings of existing models and the characteristics of the target hardware.

The result of applying the multi-TDMA model is shown in Figure 16. The execution times of the TDMA and multi-TDMA models were 0.65 ms and 0.63 ms, respectively on average, as shown in Figure 16a. The execution time for the A, E and R phases were 0.24 ms, 0.21 ms, and 0.18 ms respectively in both models, as shown in Figure 16b. The variations in the execution times of the models were 0.02 ms and 0.01 ms, respectively. These values are negligible in the cache misses results of Figure 16c, multi-TDMA showed the greatest value in all categories, particularly in the number of L2 cache misses. The L2 cache misses of the TDMA and multi-TDMA were 911 and 4741, respectively. Thus, the L2 cache misses in the multi-TDMA were approximately five times more. This caused more cache misses, as two cores accessed the memory simultaneously. In Figure 13b, when two cores ran the benchmark concurrently, there were about 2000 L2 cache misses, but about twice as many occurred in the multi-TDMA. In the previous experiment each core had one partition but in the multi-TDMA experiment, two partitions were allocated per core. We considered this difference as an overhead for partition scheduling. The number of the BIU requests in the multi-TDMA model was approximately 2000, which is similar to the TDMA model as shown in Figure 16c. Because the BIU requests differences in the multi-TDMA model were insignificant, it appears that this will not affect execution times.

Figure 16. Benchmark measurement results by TDMA and multi-TDMA. (a) Execution time. (b) Execution time by each phase. (c) Cache miss and BIU request.

To implement execution models with the quad-core, six pseudo-partitions and four pseudo-partitions were used for implementing TDMA and multi-TDMA, respectively. This reduced

the memory space required to implement the multi-TDMA model. In addition, the major time frame of multi-TDMA was 4 ms, which is 50% shorter than in TDMA.

We show that multi-TDMA reduces the execution time by 1% and 23% respectively compared to TDMA and AER. The multi-TDMA model has more L2 cache misses than TDMA, but does not affect the execution times due to the minor increase in the BIU requests. Consequently, on this study's target hardware the multi-TDMA had a higher resource utilization than TDMA and shorter execution time than AER.

5. Related Works

In this section, we describe related studies that focus on reducing interference due to memory sharing.

Cache partitioning. Cache partitioning is a technique of allocating the cache capacity to the cores or processes. This reduces cache misses by ensuring that the cache used by the process is not evicted. In multicore architecture, there may be unexpected cache misses due to execution occurring in other cores. Cache partitioning includes hardware-assisted cache lockdown (cache locking) [4] and software-based cache coloring [3].

Cache lockdown [4] divides the cache into way units and allocates available cache area to each core (or process). Cache lockdown is categorized as static locking or dynamic locking [32–34]. In the first case, dynamic locking [33] adaptively locks regions by profiling cache regions that are accessed while the system is executing. In the other case, static locking [34] creates a cache memory map to be allocated to processes based on profiled log data before runtime. However, dynamic locking can reduce the determinism of real-time systems, such as avionics systems, because it modifies cache locking information at run time [35]. Moreover, there is data or instruction cache locking [33,36] as well as user space or kernel space locking [37]. These cache lockdown methods depend on variables, such as the number of cache lockdown entries supported by the hardware.

Cache coloring [3] divides pages into colors based on sets in the cache and maps them to virtual and physical memory addresses. Cache coloring maps page addresses at the OS, compiler, or application level, requiring modifications to the OS's virtual memory [11]. In particular, if the context switch occurs when coloring dynamically at runtime, the page color has to be adjusted [38].

In avionics systems, several studies have explored bounding the worst-case execution time (WCET), by using cache partitioning to reduce the memory sharing interference. In the single core ARINC-653-based partitioning system, a method has been proposed to reduce the intra-partition cache misses by applying cache locking [35]. In this study, greedy selection and genetic selection algorithms were applied to select the cache areas in each partition for static cache locking. The work measures the number of misses and hits for cache blocks accessed by partitions. Based on this, the weight was calculated, and cache blocks with large weights were locked, reducing the cache misses by up to 45%. Investigations of cache and memory partitioning in the multicore ARINC 653 RTOS have also been done [39]. The study allocated memory buses based on the bandwidth required by cores, and banked the main memory to each core. Notably, the memory access latency was improved by partitioning on the set and way basis for the LLC. The proposed resource partitioning on an octa-core based system was measured, and it demonstrated an improvement in memory access latency of up to 56%.

Execution model. The execution model is a method of controlling task scheduling to reduce the interference caused by hardware resource sharing, such as the predictable execution model (PREM) [7]. The PREM coordinates the execution of tasks to avoid shared resources such as I/O and memory from contending on each core. In particular, memory-centric scheduling (MCS) [8] has been proposed to avoid or limit concurrent access to shared memory.

TDMA [14,28] is a typical MCS, which executes only one process in a single time slot. Even in multicore architectures, only one core runs in a time slot, eliminating inter-core interference due to shared memory usage. The purpose of TDMA is to reduce the contention caused by memory sharing,

and to bound tightly the process's WCET to increase predictability. However, TDMA reduces multicore architecture utilization.

To complement the low utilization of TDMA, multi-phase models have been proposed that allow tasks to be executed concurrently by dividing them into memory-centric and computational phases (denoted as M and C phases, respectively) [10,40,41]. For this, the task is divided into a pre-fetch phase that reads data and instructions from the shared memory (global memory) to the local memory, and a computation phase that computes the data in the local memory. The computational phase does not access the shared memory, allowing it to run concurrently with the pre-fetch phase. In Reference [10], the schedulability of sequential task systems and models with contention were analyzed. However, this study statically analyzed the schedulability for a single core architecture. In the multicore direct memory access (DMA) architecture, the WCET was bound to around 10%, according to the increase in the number of cores in the two-phase (M and C phases) models [40]. In Reference [41], the execution and write-back (P-E-W) phases from the COTS multicore model were implemented by allocating cores for global scheduling. In particular, a synchronous model that executes P, E, and W phases on the same core and an asynchronous model that dedicates one core to the P or W phase with cache partitioning was discussed. However, the user API (system call) and kernel API needed to be modified for the PEW model. Similar to the PEW model, there are studies on the AER models [9,29,30]. The AER model, similar to other models, executes the A and R phases that access memory in parallel with the E phase.

MCSs, such as the AER model, were applied to avionics systems. A study on applying AER to a flight management system (FMS) proposed an implementation method that considered aperiodic and restartable tasks [9]. To control each phase of the AER model, a message-passing buffer was allocated to the on-chip SRAM for direct access between cores. This study implemented the system on a bear-metal executive to provide real-time properties with a minimal footprint.

One problem with the multi-phase model is that developers must manually port the existing code as a multi-phase task. To solve this problem, tools have been studied for analyzing code and converting it for multi-phase models [42,43]. In particular, there is a study that proposes a C-language-based tool to eliminate compiler dependence in code generation [42]. In this study, memory access is recorded for code refactoring and based on this, a processor-accessible memory range (chunk) is detected. The memory access coverage of the automatically generated code and the manual code was about 10% on average. A tool for automatically converting code into an AER model for avionics systems has also been proposed [44]. In this study, the optimization programming language solver automatically generates code by statically mapping tasks onto each core and the memory. Furthermore, based on this tool, the FMS system code [9] was automatically generated to fit the AER model. Experiments have shown that available memory, by reducing interference, is increased by up to approximately 50%.

In many-core architectures, there was a study that applied an execution model to reduce interference due to memory and I/O sharing [45]. In particular, four rules of the execution model were established to reduce the inter-partition interference at the memory bank level. This improves memory throughput by reducing the memory access delays.

Memory regulation. Memory regulation is a method for limiting the available capacity or bandwidth of shared memory per core.

There are studies where shared memory interference was reduced by isolating the available memory bandwidth on each core [5,6]. MemGuard [5] has been proposed to reserve the available memory bandwidth for each core in the kernel space. To solve the problem caused by the quality of service (QoS) degradation that occurs when DRAMs are simultaneously accessed in multiple cores, memory bandwidth was allocated based on the memory access cycles and budgets required for each core. Experimental results show that the IPC decreased with the limiting of memory bandwidth, but the throughput increased because the interference due to memory separation was reduced. In Reference [6], the per-core memory bandwidth mechanism of single core equivalence (SCE) was

applied to the multicores. Additionally, an adaptive mixed-criticality (AMC) scheduling algorithm was applied, and the interference due to the shared memory was reduced by this stall-aware scheduling.

Memory bank-aware methods have also been proposed to reduce memory interference. A method of allocating a bank of memory to each core by applying a page-based kernel-level memory allocator has been studied [12,46]. These methods implement page allocators by extending Linux's buddy algorithm. The works were conducted by using an Intel Xeon to measure interference when multiple cores were allocated to the same memory bank and when each was allocated to a separate memory bank. When allocated to the divided bank, the memory access latency remained constant even when the number of cores executed concurrently increased. Moreover, when allocated to the same bank, memory access latency increased by about 150 ms.

It was shown in Reference [47] that memory contention can be eliminated in avionics systems with a dynamic memory bandwidth level. In this study, the schedule table was configured offline and assigned to each core through interference-sensitive WCET computation and memory bandwidth throttling mechanisms.

6. Conclusions

In this work, the task execution model was applied to reduce the interference due to memory sharing in a multicore avionics system. In safety-critical systems, such as avionics systems, the system must be deterministic by reducing the effects of such interference. We applied the proposed execution model to COTS hardware and commercial RTOS. We proposed a multicore flow control method with pseudo-partition and message queues to implement the TDMA and AER execution models in VxWorks 653. Then, we showed that the execution model reduced interference due to the shared memory access between cores.

We used the micro-benchmark to measure inter-core interference due to memory sharing in a multicore system. We measured the benchmark execution times, L1 and L2 cache misses, and the BIU requests in a quad-core architecture. We showed that the execution time variation increased when all cores accessed the memory and contention occurred. Based on this, when the TDMA model was applied, the non-determinism of the execution time was decreased by reducing the interference caused by shared memory. At this time, the L2 cache and the BIU requests decreased by about 60%. The AER model increased the execution time by about 20% due to the overhead for controlling the execution of processes running on each core. Memory queuing for the process' phase control of this model caused context switches, resulting in more cache misses than in TDMA. However, in the AER model, the number of cache misses was 50% less than when four cores concurrently accessed the shared memory. We also proposed a multi-TDMA, in which two cores run concurrently while the target hardware bus processes two coherency granules in one cycle. Furthermore, our experiments demonstrated that the TDMA and multi-TDMA models produce insignificant differences in the execution times and cache misses. In particular, multi-TDMA unlike AER does not require phase control. Therefore, the multi-TDMA model has a higher utilization than the TDMA model and a shorter execution time than the AER model.

Author Contributions: Conceptualization, S.P. and H.K.; methodology, S.P. and H.K.; software, S.P.; writing—original draft preparation, S.P.; writing—review and editing, S.P. and H.K.; supervision, H.K.; project administration, M.-Y.K. and H.-K.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Agency for Defense Development under the contract UD180002ED.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Huyck, P. ARINC 653 and multi-core microprocessors—Considerations and potential impacts. In Proceedings of the 2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC), Williamsburg, VA, USA, 4–18 October 2012; pp. 1–16.
2. Agrou, H.; Sainrat, P.; Gatti, M.; Toillon, P. Mastering the behavior of multi-core systems to match avionics requirements. In Proceedings of the 2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC), Williamsburg, VA, USA, 4–18 October 2012; pp. 6E5-1–6E5-12.
3. Liedtke, J.; Hartig, H.; Hohmuth, M. OS-controlled cache predictability for real-time systems. In Proceedings of the Third IEEE Real-Time Technology and Applications Symposium, Montreal, QC, Canada, 9–11 June 1997; pp. 213–224.
4. Mittal, S. A survey of techniques for cache locking. *ACM Trans. Des. Autom. Electron. Syst. (TODAES)* **2016**, *21*, 1–24. [\[CrossRef\]](#)
5. Yun, H.; Yao, G.; Pellizzoni, R.; Caccamo, M.; Sha, L. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), Philadelphia, PA, USA, 9–11 April 2013; pp. 55–64.
6. Awan, M.A.; Souto, P.F.; Bletsas, K.; Akesson, B.; Tovar, E. Mixed-criticality scheduling with memory bandwidth regulation. In Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 19–23 March 2018; pp. 1277–1282.
7. Pellizzoni, R.; Betti, E.; Bak, S.; Yao, G.; Criswell, J.; Caccamo, M.; Kegley, R. A predictable execution model for COTS-based embedded systems. In Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, Chicago, IL, USA, 11–14 April 2011; pp. 269–279.
8. Yao, G.; Pellizzoni, R.; Bak, S.; Betti, E.; Caccamo, M. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Syst.* **2012**, *48*, 681–715. [\[CrossRef\]](#)
9. Durrieu, G.; Faugère, M.; Girbal, S.; Gracia Pérez, D.; Pagetti, C.; Puffitsch, W. Predictable Flight Management System Implementation on a Multicore Processor. In Proceedings of the Embedded Real Time Software (ERTS'14), Toulouse, France, 5–7 February 2014.
10. Melani, A.; Bertogna, M.; Bonifaci, V.; Marchetti-Spaccamela, A.; Buttazzo, G. Memory-processor co-scheduling in fixed priority systems. In Proceedings of the 23rd International Conference on Real Time and Networks Systems, Lille, France, 4–6 November 2015; pp. 87–96.
11. El-Sayed, N.; Mukkara, A.; Tsai, P.A.; Kasture, H.; Ma, X.; Sanchez, D. KPart: A hybrid cache partitioning-sharing technique for commodity multicores. In Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), Vienna, Austria, 24–28 February 2018; pp. 104–117.
12. Yun, H.; Mancuso, R.; Wu, Z.P.; Pellizzoni, R. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In Proceedings of the 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), Berlin, Germany, 15–17 April 2014; pp. 155–166.
13. Diniz, N.; Rufino, J. ARINC 653 in space. In Proceedings of the DASIA 2005—Data Systems in Aerospace, Edinburgh, Scotland, 30 May–2 June 2005; Volume 602.
14. Schranzhofer, A.; Chen, J.J.; Thiele, L. Timing analysis for TDMA arbitration in resource sharing systems. In Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium, Stockholm, Sweden, 12–15 April 2010; pp. 215–224.
15. Park, S.; Song, D.; Jang, H.; Kwon, M.Y.; Lee, S.H.; Kim, H.K.; Kim, H. Interference Analysis of Multicore Shared Resources with a Commercial Avionics RTOS. In Proceedings of the 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), San Diego, CA, USA, 8–12 September 2019; pp. 1–10.
16. 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 Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), Austin, TX, USA, 2 December 2001; pp. 3–14.
17. Watkins, C.B.; Walter, R. Transitioning from federated avionics architectures to integrated modular avionics. In Proceedings of the 2007 IEEE/AIAA 26th Digital Avionics Systems Conference, Dallas, TX, USA, 21–25 October 2007; pp. 2.A.1-1–2.A.1-10.

18. Littlefield-Lawwill, J.; Kinnan, L. System considerations for robust time and space partitioning in integrated modular avionics. In Proceedings of the 2008 IEEE/AIAA 27th Digital Avionics Systems Conference, St. Paul, MN, USA, 26–30 October 2008; pp. 1.B.1-1–1.B.1-11.
19. Bieber, P.; Boniol, F.; Boyer, M.; Noulard, E.; Pagetti, C. New Challenges for Future Avionic Architectures. *AerospaceLab* **2012**, *4*, 1–10.
20. VanderLeest, S.H. ARINC 653 hypervisor. In Proceedings of the 29th Digital Avionics Systems Conference, Salt Lake City, UT, USA, 3–7 October 2010; pp. 5.E.2-1–5.E.2-20.
21. Gaska, T.; Werner, B.; Flagg, D. Applying virtualization to avionics systems—The integration challenges. In Proceedings of the 29th Digital Avionics Systems Conference, Salt Lake City, UT, USA, 3–7 October 2010; pp. 5.E.1-1–5.E.1-19.
22. Prehofer, C.; Horst, O.; Dodi, R.; Geven, A.; Kornaros, G.; Montanari, E.; Paolino, M. Towards Trusted Apps platforms for open CPS. In Proceedings of the 2016 3rd International Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC), Vienna, Austria, 11 April 2016; pp. 23–28.
23. 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 Proceedings of the 2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC), Prague, Czech Republic, 13–17 September 2015; pp. 8D4-1–8D4-15.
24. Jean, X.; Gatti, M.; Faura, D.; Pautet, L.; Robert, T. A software approach for managing shared resources in multicore IMA systems. In Proceedings of the 2013 IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC), East Syracuse, NY, USA, 5–10 October 2013; pp. 7D1-1–7D1-15.
25. Löfwenmark, A.; Nadjm-Tehrani, S. Challenges in future avionic systems on multi-core platforms. In Proceedings of the 2014 IEEE International Symposium on Software Reliability Engineering Workshops, Naples, Italy, 3–6 November 2014; pp. 115–119.
26. Jean, X.; Faura, D.; Gatti, M.; Pautet, L.; Robert, T. Ensuring robust partitioning in multicore platforms for ima systems. In Proceedings of the 2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC), Williamsburg, VA, USA, 4–18 October 2012; pp. 7A4-1–7A4-9.
27. Strunk, E.A.; Knight, J.C.; Aiello, M.A. Distributed reconfigurable avionics architectures. In Proceedings of the 23rd Digital Avionics Systems Conference (IEEE Cat. No. 04CH37576), Salt Lake City, UT, USA, 28 October 2004; Volume 2, p. 10.B.4-101.
28. Andrei, A.; Eles, P.; Peng, Z.; Rosen, J. Predictable implementation of real-time applications on multiprocessor systems-on-chip. In Proceedings of the 21st International Conference on VLSI Design (VLSID 2008), Hyderabad, India, 4–8 January 2008; pp. 103–110.
29. Maia, C.; Nogueira, L.; Pinho, L.M.; Pérez, D.G. A closer look into the aer model. In Proceedings of the 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), Berlin, Germany, 6–9 September 2016; pp. 1–8.
30. Maia, C.; Nelissen, G.; Nogueira, L.; Pinho, L.M.; Pérez, D.G. Schedulability analysis for global fixed-priority scheduling of the 3-phase task model. In Proceedings of the 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Hsinchu, Taiwan, 16–18 August 2017; pp. 1–10.
31. Parkinson, P.; Kinnan, L. Safety-critical software development for integrated modular avionics. *Embed. Syst. Eng.* **2003**, *11*, 40–41.
32. Suhendra, V.; Mitra, T. Exploring locking & partitioning for predictable shared caches on multi-cores. In Proceedings of the 45th annual Design Automation Conference, Anaheim, CA, USA, 8–13 June 2008; pp. 300–303.
33. Ding, H.; Liang, Y.; Mitra, T. WCET-centric dynamic instruction cache locking. In Proceedings of the 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 24–28 March 2014; pp. 1–6.
34. Puaud, I.; Decotigny, D. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In Proceedings of the 23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002, Austin, TX, USA, 3–5 December 2002; pp. 114–123.
35. Dugo, A.T.A.; Lefoul, J.B.; De Magalhaes, F.G.; Assal, D.; Nicolescu, G. Cache locking content selection algorithms for ARINC-653 compliant RTOS. *ACM Trans. Embed. Comput. Syst. (TECS)* **2019**, *18*, 1–20. [[CrossRef](#)]

36. Vera, X.; Lisper, B.; Xue, J. Data Cache Locking for Tight Timing Calculations. *ACM Trans. Embed. Comput. Syst.* **2007**, *7*. [[CrossRef](#)]
37. Gracioli, G.; Alhammad, A.; Mancuso, R.; Fröhlich, A.A.; Pellizzoni, R. A Survey on Cache Management Mechanisms for Real-Time Embedded Systems. *ACM Comput. Surv.* **2015**, *48*. [[CrossRef](#)]
38. Ye, Y.; West, R.; Cheng, Z.; Li, Y. COLORIS: A dynamic cache partitioning system using page coloring. In Proceedings of the 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), Edmonton, AB, Canada, 24–27 August 2014; pp. 381–392. [[CrossRef](#)]
39. Pak, E.; Lim, D.; Ha, Y.M.; Kim, T. Shared Resource Partitioning in an RTOS. In Proceedings of the 13th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPRT), Dubrovnik, Croatia, 14 April 2017; Volume 14.
40. Alhammad, A.; Wasly, S.; Pellizzoni, R. Memory efficient global scheduling of real-time tasks. In Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA, 13–16 April 2015; pp. 285–296.
41. Rivas, J.M.; Goossens, J.; Poczekajlo, X.; Paolillo, A. Implementation of Memory Centric Scheduling for COTS Multi-Core Real-Time Systems. In Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Stuttgart, Germany, 9–12 July 2019.
42. Mancuso, R.; Dudko, R.; Caccamo, M. Light-PREM: Automated software refactoring for predictable execution on COTS embedded systems. In Proceedings of the 2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications, Chongqing, China, 20–22 August 2014; pp. 1–10.
43. Fort, F.; Forget, J. Code generation for multi-phase tasks on a multi-core distributed memory platform. In Proceedings of the 2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Hangzhou, China, 18–21 August 2019; pp. 1–6.
44. Girbal, S.; Pérez, D.G.; Le Rhun, J.; Faugère, M.; Pagetti, C.; Durrieu, G. A complete toolchain for an interference-free deployment of avionic applications on multi-core systems. In Proceedings of the 2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC), Prague, Czech Republic, 13–17 September 2015; pp. 7A2-1–7A2-14.
45. Perret, Q.; Maurere, P.; Noulard, E.; Pagetti, C.; Sainrat, P.; Triquet, B. Temporal isolation of hard real-time applications on many-core processors. In Proceedings of the 2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, 11–14 April 2016; pp. 1–11.
46. Yun, H.; Pellizzoni, R.; Valsan, P.K. Parallelism-aware memory interference delay analysis for COTS multicore systems. In Proceedings of the 2015 27th Euromicro Conference on Real-Time Systems, Lund, Sweden, 7–10 July 2015; pp. 184–195.
47. Agrawal, A.; Fohler, G.; Freitag, J.; Nowotsch, J.; Uhrig, S.; Paulitsch, M. Contention-aware dynamic memory bandwidth isolation with predictability in cots multicores: An avionics case study. In Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), Dubrovnik, Croatia, 27–30 June 2017; Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik: Wadern, Germany.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).