Contention-Aware Dynamic Memory Bandwidth Isolation With Predictability in COTS Multicores: An Avionics Case Study*

Ankit Agrawal[1], Gerhard Fohler[2], Johannes Freitag[3], Jan Nowotsch[4], Sascha Uhrig[5], and Michael Paulitsch[6]

1 **Chair of Real-Time Systems, Technische Universität Kaiserslautern, Kaiserslautern, Germany**
agrawal@eit.uni-kl.de
2 **Chair of Real-Time Systems, Technische Universität Kaiserslautern, Kaiserslautern, Germany**
fohler@eit.uni-kl.de
3 **Airbus Innovations, Munich, Germany**
johannes.freitag@airbus.com
4 **Airbus Innovations, Munich, Germany**
jan.nowotsch@airbus.com
5 **Airbus Innovations, Munich, Germany**
sascha.uhrig@airbus.com
6 **Base Systems, Thales Austria GmbH, Vienna, Austria†**
michael.paulitsch@thalesgroup.com

Abstract

Airbus is investigating COTS multicore platforms for safety-critical avionics applications, pursuing helicopter-style autonomous and electric aircraft. These aircraft need to be ultra-lightweight for future mobility in the urban city landscape. As a step towards certification, Airbus identified the need for new methods that preserve the ARINC 653 single core schedule of a Helicopter Terrain Awareness and Warning System (HTAWS) application while scheduling additional safety-critical partitions on the other cores.

As some partitions in the HTAWS application are memory-intensive, static memory bandwidth throttling may lead to slow down of such partitions or provide only little remaining bandwidth to the other cores. Thus, there is a need for dynamic memory bandwidth isolation. This poses new challenges for scheduling, as execution times and scheduling become interdependent: scheduling requires execution times as input, which depends on memory latencies and contention from memory accesses of other cores – which are determined by scheduling. Furthermore, execution times depend on memory access patterns.

In this paper, we propose a method to solve this problem for slot-based time-triggered systems without requiring application source-code modifications using a number of dynamic memory bandwidth levels. It is NoC and DRAM controller contention-aware and based on the existing interference-sensitive WCET computation and the memory bandwidth throttling mechanism. It constructs schedule tables by assigning partitions and dynamic memory bandwidth to each slot on each core, considering worst case memory access patterns. Then at runtime, two servers – for processing time and memory bandwidth – run on each core, jointly controlling the contention between the cores and the amount of memory accesses per slot.

* The research leading to these results was funded within the EMC^2 project by the EU ARTEMIS Joint Undertaking under grant agreement no. 621429.
† The work presented here was carried out while the author was at Airbus Innovations.

29th Euromicro Conference on Real-Time Systems (ECRTS 2017).
Editor: Marko Bertogna; Article No. 2; pp. 2:1–2:22

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

As a proof-of-concept, we use a constraint solver to construct tables. Experiments on the P4080 COTS multicore platform, using a research OS from Airbus and EEMBC benchmarks, demonstrate that our proposed method enables preserving existing schedules on a core while scheduling additional safety-critical partitions on other cores, and meets dynamic memory bandwidth isolation requirements.

1998 ACM Subject Classification D.4.7 Organization and Design

Keywords and phrases dynamic memory bandwidth isolation, safety-critical avionics, COTS multicores

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2017.2

1 Introduction

For future mobility, Airbus is pursuing autonomous aircraft targeting urban landscape to ease traffic, for instance, Uber-like CityAirbus [12], and the Vahana aircraft [17]. These helicopter-style aircraft will be electrically powered, requiring ultra-light weight to boost their power-to-weight ratio. They will need most avionics applications used in current aircraft, along with a DAL-A (the highest design assurance level) sense-and-avoid application for autonomous flying, unavailable today. Further, the electronic systems used in current aircraft need to be redesigned to reduce size, weight, and power consumption (SWaP), by integrating more avionics applications on the same number of processors, which is not feasible with single-core processors. The power consumption of current electronic systems is marginal compared the envisaged electric propulsion system. However, limiting it will eliminate the need for active cooling, further reducing SWaP. It will also eliminate the risk of a failure of the cooling system. Airbus is investigating COTS multicores to meet these future demands.

Safety-critical avionics hardware and software demand certification from certification authorities, which requires that the processes used in the design of digital hardware must relate to the DAL of the intended use [9]. However, COTS multicores are designed primarily for mass market and average-case performance and do not customarily follow DAL-based design processes. The CAST-32a position paper [27] describes the issues in the certification of COTS multicores, but the concrete implementation details are still open. Airbus is aiming at an incremental transition step towards the use of full COTS multicore performance: In the first step existing safety-critical single-core avionics application will be ported to a COTS multicore by preserving the original ARINC 653 schedule as well as the source code while executing it on only one core. Additional applications must be assigned to another core of the COTS multicore. This step reduces certification cost since documentation and verification of the software is already available. In the second (future) step, an application can be distributed over all the available cores. This paper focuses on the first step.

The Helicopter Terrain Awareness and Warning System (HTAWS), selected as reference application, is a pilot supporting system rated as DAL-C. It shows the helicopter pilot the surrounding topographical layout (including large buildings, power lines) with “flyable” areas together with warnings when the helicopter approaches rough terrain, e.g., when vision is degraded. Such a system also needs to be integrated into future autonomous aircraft to allow the aircraft to perform autonomous path planning and in-flight re-planning. HTAWS application is currently implemented on a dedicated avionics computer which is not feasible for ultra-light autonomous aircraft due to their SWaP constraints.

One of the major obstacles in certifying COTS multicores for use in safety-critical avionics systems is the contention between cores. The contention between cores arises due to the

implicit sharing of hardware resources like the network-on-chip, memory controller and main memory. When left unmitigated, it can slow down the partitions, resulting in deadline misses or even system failure. For example, the authors in [20] showed in an experimental setup, using the P4080 8-core COTS multicore platform, that the latency of a single store request is increased by a factor of 25.82 when the number of active cores is increased from 1 to 8. Approaches based on static memory bandwidth throttling, such as MemGuard [31], have been shown to be useful and gained adoption. In addition to controlling the amount of memory bandwidth available to cores, they allow the use of existing scheduling algorithms with minor modifications, as the effect of throttling can be seen as a slower processor. Measurements from the HTAWS application on a COTS multicore with only one active core show that some partitions are memory-intensive. Using static memory bandwidth isolation may lead to slow down of such partitions or provide only little remaining bandwidth to the other cores.

Static memory bandwidth isolation mechanisms assign a constant amount of memory bandwidth Q_{sm_n} to each core N_n before runtime. This limits the worst-case number of contentions any partition on a core can experience at any time, which allows computation of execution time separately from scheduling. However, under dynamic memory bandwidth isolation, as scheduling impacts the contention from the other cores and the dynamic memory bandwidth, execution time computation and scheduling cannot be performed independently. Specifically, the execution time of a partition depends on (a) the time taken for memory accesses which depends on the contention from the other cores, and (b) its worst-case memory access pattern and the dynamic memory bandwidth assigned in each slot during its execution.

Contention-aware dynamic memory bandwidth throttling can solve these issues but is not straightforward as it introduces an interdependency between scheduling, execution time, memory bandwidth, and contention: Scheduling requires execution time of a partition as input, which depends on the memory bandwidth and the worst-case memory access pattern. Memory bandwidth depends on the contention between cores and the number of memory accesses from each core, which depend on scheduling. Thus, determination of dynamic memory bandwidth and scheduling of partitions cannot be done in separate steps.

In this paper, we propose a method which solves the dynamic memory bandwidth isolation problem for time-triggered systems using a number of dynamic memory bandwidth levels. It is NoC and memory controller contention-aware and is based on the existing interference-sensitive WCET computation [22] and the memory bandwidth throttling mechanism [31], which it extends by including delay due to contention in the on-chip network and the DRAM controller as well. It constructs the schedule tables offline and assigns partitions and memory bandwidth to each slot on each core. Then, at runtime, two servers – processing time server and memory bandwidth server – run on each core, jointly controlling the contention between cores in each slot.

As a proof-of-concept, we generate schedule tables performing executing time computation and scheduling in the same step using a constraint solver. Experiments on a COTS multicore P4080, using a research OS from Airbus and EEMBC benchmarks, further demonstrate the feasibility of our proposed method.

Contributions

- We introduce a new scheduling problem for COTS multicores derived from a real avionics application – HTAWS, in which some partitions are memory-intensive.
- We present a method that solves the problem for time-triggered systems using a fixed number of dynamic memory bandwidth levels, is NoC and memory contention-aware, and based on the existing interference-sensitive WCET computation and memory bandwidth throttling mechanism.

- We show the execution time computation for a partition under dynamic memory bandwidth using worst-case memory access pattern.

Paper structure. The remainder of the paper is organised as follows: Section 2 presents the HTAWS application used in helicopters from Airbus. Section 3 presents the system model and the notation used in this work. Section 4 describes how the server-based runtime mechanism provides dynamic memory bandwidth isolation. Section 5 describes the scheduling and execution time computation for a partition under dynamic memory bandwidth using the worst-case memory access pattern, in a single step. Section 6 presents the implementation and evaluation of the proposed method. Section 7 presents the related work. Section 8 concludes the paper and also presents the future work.

2 Helicopter Terrain Awareness and Warning System Application

For future mobility, Airbus is pursuing autonomous electric aircraft targeting urban landscape to ease traffic. In addition to the avionics applications used in current aircraft, DAL-A sense-and-avoid applications for autonomous flying are needed, which still need to be developed. Nevertheless, as a starting point, a non-autonomous version of a sense-and-avoid application known as Helicopter Terrain Awareness and Warning System (HTAWS) is available today. This application is an optional feature of Airbus' helicopters.

HTAWS enables safer flying by assisting the pilot especially in degraded visual environments like flying at night, poor visibility conditions due to fog, rough terrain, and low-altitudes, useful for search and rescue mission by air ambulances and coastguards. Furthermore, it contains a map of power transmission lines and other obstacles which are hard to detect even in good weather conditions. This application is a DAL-C certified safety-critical avionics application executing on a single-core processor running VxWorks 653 RTOS from Wind River [25].

2.1 Single-core Data

HTAWS includes eight partitions with a major time frame H (MAF) of 66ms. Figure 1 shows its ARINC 653 partition-level schedule. At runtime, the inter-partition scheduler schedules the partition based on a static schedule. Table 1 shows the partition-level timing constraints. I/O operations are not part of the study here.

2.2 Measured Data on COTS Multicore

Cassidian, part of the Airbus group, performed measurements on the real HTAWS application in a test setup using a dual-core COTS P5020 platform running VxWorks with only 1 active core. A core is considered active if, in the time interval under consideration, it is allowed to issue memory accesses. Table 2 shows the maximum execution time and the maximum number of memory accesses for each partition observed in 1000 runs. Partitions π_4, π_5, and π_7, are memory-intensive.

Airbus Innovations, using its proprietary OS for research, provided the maximum observed memory latencies for a different number of active cores. Table 3 lists these values for two COTS multicore platforms: the dual-core P5020 platform and the eight-core P4080 platform.

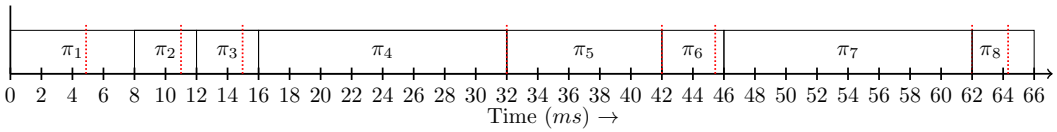

Figure 1 DAL-C certified ARINC 653 single-core schedule of the HTAWS application from Airbus. Dotted red lines indicate the maximum observed execution time of each of the partition, without contention, as shown in column 3 of Table 2.

Table 1 HTAWS application: Partition-level timing data.

Partition π_i	abs. release time r_i (ms)	abs. deadline d_i (ms)	Duration (ms)
π_1	0	8	8
π_2	8	12	4
π_3	12	16	4
π_4	16	32	16
π_5	32	42	10
π_6	42	46	4
π_7	46	62	16
π_8	62	66	4

Table 2 HTAWS application: Measurements on the dual-core COTS P5020 platform with only 1 active core.

Partition	Max. obs. num. memory accesses	Max. obs. ET (ms)
π_1	6618	4.88
π_2	2764	3.12
π_3	7381	2.97
π_4	477886	16.00
π_5	262962	10.00
π_6	4275	3.44
π_7	477886	16.00
π_8	7020	2.32

Table 3 Maximum observed memory latencies δ_j (in ns) for different number of active cores j on COTS multicores P5020 and P4080.

COTS Multicore	Mem. Lat. δ_1 (ns)	Mem. Lat. δ_2 (ns)
P5020	24.17	49.17
P4080	34.17	136.67

3 Models and Notation

This section presents the models and notation considered in this work for the COTS multicores, slots, servers, and the partitions.

3.1 COTS Multicore

We consider a COTS multicore comprising two types of hardware resources: homogeneous processing cores and a shared hardware resource consisting of a shared on-chip network and a shared DRAM sub-system including the DRAM controllers and the DRAM device.

Set $\mathcal{N}$ represents the homogeneous processing cores $\{N_1, ..., N_{|\mathcal{N}|}\}$. We consider for the shared hardware resource a set of contention-aware shared hardware resource latencies Δ, where each element δ_j denotes the maximum latency of a load/store request issued from a core under maximum contention from j active cores. A core is considered *active* if, in the time interval under consideration, it is allowed to issue memory accesses. Further, we assume that the latencies are non-decreasing with an increasing number of active cores i.e. $\frac{\delta_j}{j} \leq \frac{\delta_{j+1}}{j+1}$. This assumption allows to safely bound the time taken by a memory access accounted with e.g. δ_j latency while at runtime it may experience a lower latency δ_1 due to no contention from the $j-1$ active cores.

We assume that the COTS multicore provides a platform-level shared hardware timer and a hardware performance counter for each core to count accesses to the memory. Examples of compatible COTS multicore include Qualcomm P4080 with 8 cores [8], Qualcomm P5020 with 2 cores [23].

Figure 2 shows the hardware architecture of the 8-core COTS platform P4080 [8] from Qualcomm (initially Freescale and then NXP). On the top-left corner are the eight processing cores. The shared resource latency δ_j with j active cores includes the time taken to read/write to the DRAM device[1] for a fixed cache line size (64 bytes for P4080 and P5020 platforms) as well as the additional maximum contention delay (including arbitration time) due to j active cores in (a) the NoC – CoreNet Coherency Fabric and (b) the DRAM controller. For simplicity, we refer to the considered *shared hardware resource* as *memory* and *shared hardware resource latency* as *memory latency* in the rest of the paper.

The contention-aware memory latencies can be obtained from the hardware architecture model using static-analysis-based approach. However, Qualcomm did not provide the hardware architecture model for any of the two platforms – P4080 and P5020. An alternate approach to obtain these latencies is using measurements, described in detail in [19], previous work by Airbus Innovations (previously EADS). Table 3 and 4 show the maximum observed latencies provided by Airbus Innovations. The latencies listed in these tables for the P4080 platform have also been used in these existing works [20, 22].

3.2 Slots and Servers

We divide the timeline into fixed length time slices called slots, where slot duration of each slot S_x is $duration(S)$. It is the system designer who determines a suitable slot duration and it's time unit. For the rest of the paper, we consider slot duration in the order of ms. $\mathcal{S}$ represents the set of slots.

Our proposed method uses a server-based runtime mechanism on each core. Server τ_{sp_n} represents a processing time server on core a N_n that relates to processing resource, where

[1] Not shown in Figure 2.

Table 4 Eight-core P4080 platform: Maximum observed memory latencies δ_j (ns) for different number of active cores j.

No. act. cores j	Mem. Lat. δ_j (in ns)
1	34.17
2	136.67
3	204.17
4	385.83
5	430.83
6	614.17
7	653.33
8	839.17

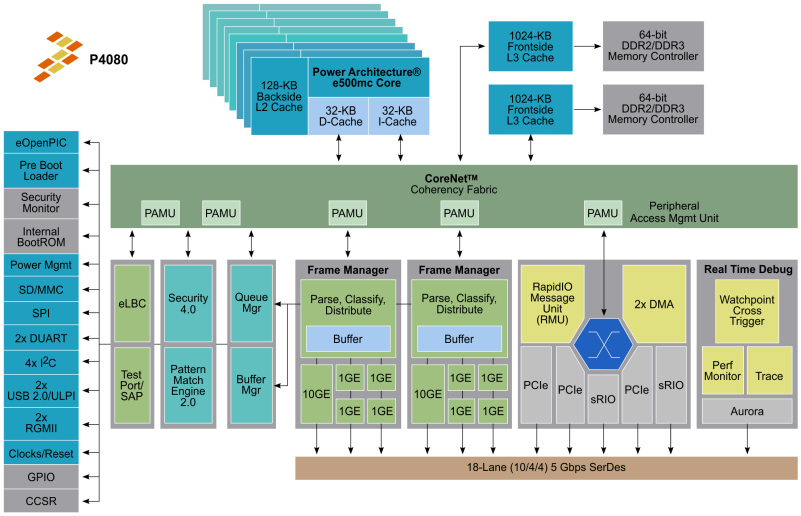

Figure 2 Architecture of the 8-core COTS platform Qualcomm P4080 [7].

sp denotes *server processing*. Server τ_{sm_n} represents the memory resource on a core N_n, where *sm* denotes *server memory*. T_{sp_n} and T_{sm_n} denote the period of each of the respective servers on a core N_n. In this work, we assume the period of all servers in the system equal to the slot duration $duration(S)$. We also assume that all servers are released synchronously on all cores.

$Q_{sp_n,x}$ (in ms) denotes the processing time server budget (simply called *processing budget*) in slot S_x on a core N_n. $Q_{sm_n,x}$ (in memory accesses) denotes the memory access server budget value (simply called *memory budget*) in slot S_x on a core N_n. Similarly, $q_{sp_n,x}$ and $q_{sm_n,x}$ denote the remaining budget during runtime for each of the corresponding servers.

3.3 Partition

The set Γ represents a set of partitions which are safety-critical i.e. their deadlines must be met. Each partition π_i is characterized by the tuple $\langle r_i, d_i, C_i^s, MA_i, C_i^m \rangle$, where

- r_i is the absolute release time,
- d_i is the absolute deadline,
- C_i^s is the core-local execution time excluding the time taken for memory accesses,

- MA_i is the maximum number of memory accesses (read/write requests) to the memory, and
- C_i^m is the multicore execution time (in slots) computed offline that includes the execution time needed for (a) C_i^s and (b) MA_i memory accesses considering possible runtime contentions from other cores, under dynamic memory bandwidth. For the rest of the paper, we use the terms – *execution time* and *multicore execution time* – interchangeably.

C_i^s and MA_i can, for instance, be acquired using a combination of static timing analysis tools like aiT and measurements as shown in [22]. We assume r_i and d_i to be integer multiples of $duration(S)$. Our proposed method is independent of the runtime memory access pattern of each partition. Thus, our partition model does not need to contain information on when and how many memory accesses a partition issues at runtime.

4 Server-Based Runtime Mechanism

Our proposed runtime mechanism provides dynamic memory bandwidth isolation, when integrated with the inter-partition scheduler. It is NoC and DRAM subsystem contention-aware and is based on MemGuard [31], an existing memory throttling mechanism .

It uses two servers per core – processing time server τ_{sp} and memory access server τ_{sm} – with a synchronised server period equal to the system-wide slot duration $duration(S)$.

4.1 Processing time server

On each core N_n a processing time server τ_{sp_n} regulates the execution time in each server instance. A processing time server budget decreases with the progression of time in a slot for each active core. During runtime, a partition on core N_n, executing in slot S_x consumes the server budget $Q_{sp_n,x}$ for the core-local execution time on core N_n and the time taken for memory accesses under contention from other cores. Due to the dynamic memory bandwidth, the processing budget may differ in distinct slots.

4.2 Memory Access Server

On each core N_n, a memory access server τ_{sm_n} regulates the total number of memory accesses in each server instance. Due to the dynamic memory bandwidth, the memory budget may differ in distinct slots. At runtime, an executing partition π_i on core N_n in slot S_x uses the memory budget $Q_{sm_n,x}$ only for memory accesses. Each access results in a decrease of remaining server budget by 1.

4.3 Runtime Behaviour

During runtime, each inter-partition-level scheduler, at the start of each slot S_x, sets the corresponding processing and memory budgets for each server based on the schedule table found in the offline phase. The server budgets need not be same in each slot. The processing time server budget decreases with the progression of time in a slot for each active core. The memory access server budget decreases by 1 on each memory access issued by an executing partition in the corresponding slot. A partition continues to execute in a slot S_x on a core N_n while both the servers have budgets greater than 0, i.e. $q_{sm_n,x} > 0 \wedge q_{sp_n,x} > 0$. If in a slot S_x on a core N_n any of the two servers exhausts its budget, the partition is stalled until the next server instance and the core is idle. If any of the two servers have a remaining server budget, it is discarded.

Jointly, the two servers on each core guarantee that the servers budgets provided for each slot in the offline schedule table, hold at runtime. This enables contention-aware dynamic memory bandwidth isolation between cores.

5 Scheduling for Dynamic Memory Bandwidth

Static memory bandwidth isolation mechanisms assign a constant amount of memory bandwidth to each core before runtime. This limits the worst-case number of contentions any partition on a core can experience at any time, which allows computation of execution time separately from scheduling. However, under dynamic memory bandwidth isolation, the execution time of a partition depends on (a) the time taken for memory accesses which depends on the contention from the other cores, and (b) its worst-case memory access pattern, which depends on the dynamic memory bandwidth assigned in each slot during its execution. As scheduling decides the amount of contention between cores which can vary between slots due to dynamic memory bandwidth, we cannot perform execution time computation and scheduling separately.

In the next sections, we show how to resolve each of the two dependencies of execution time – contention and memory access pattern. Later, we show how to perform scheduling and execution time computation together for a general case under dynamic memory bandwidth. Finally, we show an example with dynamic memory bandwidth for 2-cores.

5.1 Resolving Dependency – Contention

Under dynamic memory bandwidth, resolving contention dependency for execution time computation of a partition on a core requires knowledge of the number of active cores in each slot and the maximum contentions that can be introduced by each active core in each slot. In the next section, we describe a way to resolve this dependency considering $|\mathcal{N}| + 1$ dynamic bandwidth levels. Later, we show how to consider an arbitrary number of dynamic bandwidth levels.

5.1.1 $|\mathcal{N}| + 1$ Dynamic Bandwidth Levels

Instead of a constant memory budget for each core, we consider $|\mathcal{N}| + 1$ dynamic memory bandwidth levels. For simplicity, in each level, we divide the memory bandwidth equally between the active cores j. Each level corresponds to a memory budget Q^j_{sm}, which associates with a memory latency δ_j, providing contention-awareness to the memory bandwidth. It allows estimating the contention between cores in a slot just by knowing the number of active cores in that slot, without requiring knowledge of which partitions are scheduled on the other cores, and their exact number of memory accesses. Thus, for each memory access server instance $\tau_{sm_n,x}$ on a core N_n, there are $|\mathcal{N}| + 1$ possible budgets. For each processing time server instance $\tau_{sp_n,x}$ we consider only two fixed budgets: $Q^0_{sp} = 0$ and $Q^1_{sp} = X$. X is a fixed value determined by a system designer such that $0 < X \leq duration(S)$.

Equation (1) computes the different memory budgets based on the existing interference-sensitive WCET computation [22, 20] with equal memory budget distribution between the active cores in a slot. It also shows the relationship between the two servers and the memory latencies.

$$\forall \delta_j \in \Delta \ , \ Q^j_{sm} = \left\lfloor \frac{Q^1_{sp}}{\delta_j} \right\rfloor \tag{1}$$

Table 5 Per core memory access server budgets Q^j_{sm} with equal distribution of accesses corresponding to memory latencies from Table 3 for processing time server budget Q^1_{sp} of $duration(S) = 1ms$.

COTS Multicore	Mem. bud. Q^0_{sm}	Mem. bud. Q^1_{sm}	Mem. bud. Q^2_{sm}
P5020	0	41379	20338
P4080	0	29268	7317

E.g., consider a $duration(S)$ of 1ms and a processing budget Q^1_{sp} equals to $duration(S)$. Then, Table 5 shows the memory budgets Q^j_{sm} considering the memory latencies from Table 3. When a core is not active in a slot, we assume its memory budget and processing budget equal to 0. For computation purposes, we assume the memory latency δ_0 equals ∞. Thus, the use of dynamic memory bandwidth levels limits the maximum contentions each core can experience in each slot from the other cores.

5.1.2 Arbitrary Number of Dynamic Bandwidth Levels

Our proposed method also works for an arbitrary number of dynamic memory bandwidth levels and unequal distribution between active cores. A set $\mathcal{Q}_{sm} = \bigcup_{j=1}^{|\mathcal{N}|} \hat{Q}^j_{sm}$, where j is the number of active cores, represents memory budget distributions. A set $\hat{Q}^j_{sm}$ with j active cores, represents distinct valid memory budget distributions between the j active cores. For all the remaining cores, the assigned memory budget is 0. A memory budget distribution, sorted in increasing order of memory budgets, $\{y_1, ..., y_j\}$ for j active cores, is valid if the condition in Equation 2 holds.

$$\sum_{k=1}^{j} (y_k - y_{k-1}) * \delta_{j-k+1} \leq Q^1_{sp} \ , \text{ where } y_0 = 0 \, . \tag{2}$$

This is based on the interference-sensitive WCET computation [20, 22] and means that, in a slot, only a maximum of y_1 memory accesses from each active core will experience memory latency δ_j, $y_2 - y_1$ will experience latency δ_{j-1} and so on.

In slot S_x on an active core N_n with $j-1$ active cores, an offline scheduler can assign any memory budget from a valid budget distribution $\{y_1, ..., y_j\}$ to a memory access server instance $\tau_{sm_n,x}$ that has not been assigned to memory access servers of $j-1$ active cores. For example, a set $\hat{Q}^2_{sm}$ can contain $\{7000, 34137\}$ as a valid memory budget distribution for 2 active cores using P5020 memory latencies (row 2 of Table 3). If, in a slot S_2 on core N_1 an offline scheduler assigns a memory access server a budget of 7000, then the memory access server of the second other active core in slot S_2 will be assigned a budget of 34137. Another valid memory budget distribution for 2 active cores using P5020 memory latencies is $\{20338, 20338\}$ as listed in Table 5 (row 2 column 4). The set $\mathcal{Q}_{sm}$ is either provided as input to the offline scheduler by the system designer or generated by an offline scheduler based on the properties of an application.

For the rest of Section 5, to simplify the description, we limit to $|\mathcal{N}| + 1$ dynamic bandwidth levels with equal distribution of memory budgets between the active cores as described in Section 5.1.1. However, note that the ensuing description is still applicable to arbitrary bandwidth levels.

Table 6 Example slots to depict the interaction between memory access patterns and dynamic memory access server budgets.

Slot S_x	S_1	S_2	S_3	$S_{1'}$	$S_{2'}$	$S_{3'}$
Mem. ser. bud. $Q_{sm_1,x}$	45	100	15	100	45	15
Proc. ser. bud. $Q_{sp_1,x}$	1	1	1	1	1	1

5.2 Resolving Dependency – Memory Access Pattern and Dynamic Bandwidth

Static memory bandwidth isolations mechanisms assign same memory budget in each server instance on a core. Such mechanisms are agnostic of the number of active cores in each slot and need to consider the worst-case memory latency corresponding to a maximum number of active cores. Therefore, the use of a static memory budget per core considering worst-case memory latency ensures that if a partition performs a memory access in any of its assigned slots, a partition's execution time does not increase. However, under dynamic memory bandwidth, a partition may receive different memory budgets in each slot. Further, since the latencies of memory requests may differ between different memory budgets, the interaction between memory budgets and a partition's memory access pattern impacts a partition's execution time. The example in the next section highlights this interaction.

5.2.1 Example

Consider a partition π_1: $\langle r_1 = 0, d_1 = 3, C_1^s = 1, MA_1 = 60\rangle$.

An offline scheduler identifies three slots for π_1 on core 1. Let us consider two illustrative runtime memory access patterns of π_1 considering memory budgets and processing budgets as shown in Table 6 (columns 2,3 and 4):

1. If this partition uses slot S_1 for its core-local execution time C_1^s, and slot S_2 for its $MA_1 = 60$ memory accesses, then slots $\{S_1, S_2\}$ are sufficient to meet its C_1^s and MA_1 requirements.
2. On the other hand, if π_1 uses slot S_1 for 45 memory accesses, slot S_2 for its C_1^s, and slot S_3 for the remaining 15 memory accesses, then π_1 requires slots $\{S_1, S_2, S_3\}$ to meet its requirements.

If the memory budgets of the three slots were different as shown in columns 5,6 and 7 in Table 6, then using the previously considered memory access pattern 2, π_1 may use slot S_1 for 45 memory accesses and the remaining 0.55 time units in the slot for a part of its C_1^s, slot S_2 for its remaining 0.45 time units of C_1^s with the remaining slot for its 15 memory accesses. Thus, it just requires 2 slots. Similarly, using previously considered memory access pattern 1, the partition will need three slots.

Our proposed method is independent of the runtime memory access patterns of a partition as we consider the worst-case memory access pattern of a partition for assigned memory budgets in each slot.

5.2.2 Worst-case Memory Access Pattern

The example in the previous section illustrated the need to construct a worst-case memory access pattern for each partition to ensure the slots with dynamic memory budgets under consideration will meet the partition's requirements. Static WCET analysis tools in combination with measurements can help to find a worst-case memory access pattern. However,

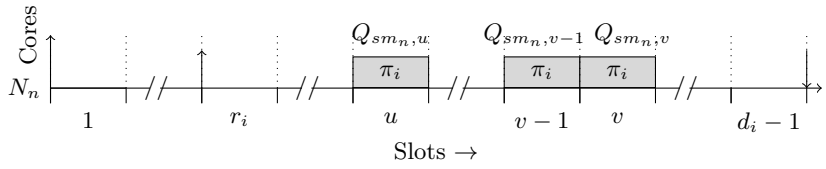

Figure 3 General case when an offline scheduler considers slots $\forall S_x \in [u, v]$ on some core N_n for partition π_i with possibly different memory budgets.

pragmatically such an approach is infeasible due to the computational complexity involved. Further such methods need to explore all valid inputs. To overcome these issues, our method uses a worst-case memory access pattern that only requires information about the partition model and the memory budget assigned to it in each slot.

The worst-case memory access pattern for a partition π_i assigned to execute on N_n in slots S_x $\forall x \in [u, v]$ manifests when a partition π_i uses $\frac{C_i^s}{Q_{sp}^1}$ slots $\in [u, v]$ with the largest assigned memory budgets for its core-local execution time C_i^s requirement, and the remaining slots for its MA_i memory access requirement. Figure 3 shows this general case with exemplary dynamic memory access server budgets for slots S_u, S_{v-1} and S_v.

Note, that the worst-case memory access pattern may not manifest at runtime. Nevertheless, considering it allows an offline scheduler to provide execution time guarantees, irrespective of the runtime memory access patterns. In the next section, we present a combined scheduling and execution time computation step.

5.3 Combined Scheduling and Execution Time

This section describes how scheduling and execution time computation are performed together.

When scheduling offline, consider a general case as shown in Figure 3 with a partition π_i: $\langle r_i, d_i, C_i^s, MA_i, C_i^m \rangle$, assigned to a core N_n. An offline scheduler considers the slots $S_x \in [u, v]$ to assign to π_i and needs to check if these slots will meet C_i^s and MA_i requirements of a partition π_i. For each slot S_x, each memory access server's budget $Q_{sm_n,x}$ relates to a valid memory budget distribution described in Section 5.1.

The offline scheduler must check if the slots S_x $\forall x \in [u, v]$ on core N_n are sufficient to meet the C_i^s and MA_i requirements of a π_i partition. Using the worst-case memory access pattern described in Section 5.2.2, the offline scheduler sorts and renumbers the slots S_x $\forall x \in [u, v]$ in descending order according to their memory budget $Q_{sm_n,x}$. $[u'v']$ represents the sorted and renumbered order of slots. Then the offline scheduler determines the number of slots needed to meet the core-local execution time requirement of π_i given by the Equation (3).

$$\mu = \frac{C_i^s}{Q_{sp}^1} \tag{3}$$

It may happen that μ is not an integer, so the offline scheduler computes the amount of memory accesses possible in the remaining part of slot $S_{u'+\lceil\mu\rceil}$ i.e.

$$\rho = \left\lceil (\lceil\mu\rceil - \mu) * Q_{sm_n,u'+\lceil\mu\rceil} \right\rceil \ .$$

Then the offline scheduler sums the memory budgets of the remaining slots from $[u' + \lceil\mu\rceil, v']$ i.e.

$$\forall x' \in [u' + \lceil\mu\rceil, v'] \ , \ \psi = \sum Q_{sm_n,x'} \ .$$

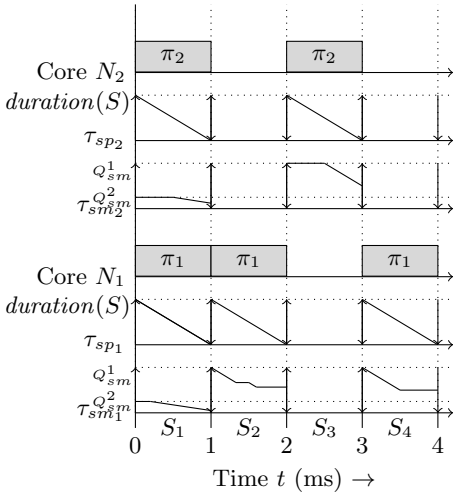

Figure 4 Illustrative example of our proposed method with three dynamic bandwidth levels.

If $MA_i \leq \rho + \psi$, then the slots $\forall S_x \in [u, v]$ will meet the partition's requirements. Thus, using the valid memory budget distributions and the worst-case memory access pattern allows scheduling and execution time computation in a single step for dynamic memory bandwidth.

5.4 Example

Figure 4 shows an illustrative example of our proposed method for 2-cores using memory latencies for P4080 platform (row 3 in Table 3). Each core N_n has two servers: processing time server τ_{sp_n} and memory access server τ_{sm_n}, with the period of each server is equal to the slot duration of 1ms. We consider 3 different dynamic bandwidth levels as described in Section 5.1.1, with valid budget distributions shown in row 3 of Table 5. The two levels relate to valid budget distributions $\{\{29268\}, \{7317, 7317\}\}$. The third level simply assigns each inactive core a 0 memory budget. For each server, the dotted horizontal lines depict the possible server budgets. During runtime, at the start of each slot, each inter-partition scheduler sets the corresponding server budgets for each server on its respective core, based on the offline schedule table (shown via budgets and partition assignment in Figure 4).

In slot S_1, i.e. at $t = 0$, both the cores are active and each inter-partition scheduler sets the corresponding memory budget $Q^2_{sm} = 7317$ and processing budget to 1ms. At the start of slot S_2 (at time $t = 1$ms), only partition π_1 is active and is assigned a memory budget of $Q_{sm_{1,2}} = Q^1_{sm} = 29268$ and processing budget of 1ms. In the time interval $[1, 1.33)$ms, the partition π_1 issues memory accesses as indicated by the corresponding decrease in remaining memory budget. In the time interval $[1.33, 1.5)$ms, π_1 does not perform any memory accesses as indicated by the memory budget being constant. Later, π_1 again briefly issues memory accesses for the next 100μs. In the time interval $[1.6, 2)$ms, since only the processing budget decreases and not the memory budget, the partition π_1 only performs computations.

At the end of slot S_3, the partition π_2 completes execution and the inter-partition scheduler of core N_2 discards the unused memory budget of the server instance $\tau_{sm_{1,3}}$. In slot S_4 (at time $t = 3$ms), as only core N_1 is active, the memory budget $Q_{sm_{1,4}}$ equals Q^1_{sm}. The partition π_1 issues memory accesses in the first half of the slot as indicated by the decrease in the remaining memory budget. Then, for the next 200μs, the partition π_1 does not issue any

memory accesses as the memory budget does not decrease and at time $t = 4$ms, π_1 completes execution.

6 Experimental Evaluation

Section 6.1 describes a proof-of-concept schedule table generation using a constraint solver. Section 6.2 presents the integration of our proposed runtime mechanism in a proprietary research OS for Qualcomm P4080 COTS multicore platform and the evaluation of our proposed method using EEMBC AutoBench benchmarks for a number of dynamic memory bandwidth levels.

6.1 Schedule Tables Generation

As a proof-of-concept, we modelled the partition allocation and scheduling problem for the Gecode [10] constraint programming (CP) solver, to find a valid schedule table for each of the two cores. The specified constraints schedule the HTAWS application on 1 core and additional partitions on the other core under dynamic memory bandwidth, while preserving the single-core HTAWS schedule.

6.1.1 Preparation of the HTAWS Application Data

The numbers shown in Table 2 Section 2 are the sole input to the study presented here as actual application internals are confidential. They give maximum observed execution times and memory accesses for each partition but do not mention the distribution of read/write requests as well as how many requests end in L3 cache. Our partition model needs the core-local execution times, which are not given. Thus, the observed data had to be "reverse engineered" for this study, making safe assumptions about the number of memory accesses and core-local execution time of each partition. We use the memory latencies shown in Table 3 to obtain core-local execution times shown in column 2 of Table 7. Partition π_5 requires on average, 73% of memory bandwidth using δ_1 memory latency. Column 3 shows the minimum constant memory bandwidth that needs to be reserved using existing interference-sensitive WCET computation [20, 22], to meet each partition's C_i^s and MA_i requirements with only 1 active core. The memory latencies are the largest observed latencies for the different number of active cores under different combinations of read and write memory requests. The execution time computation further assumes that the core stalls on each memory request. As a consequence, the processed data is pessimistic, calling for more detailed information available from the original application. The processed data is, however, safe for the experiments in this study.

6.1.2 Application and Schedule Table

As no real avionics application with more than one active core at the same time exists today, we construct a scenario by replicating some partitions such that multiple cores are active at the same time. We obtained schedule tables using the Gecode constraint solver with a model specified using our proposed method. Figure 5 shows one such schedule found by the Gecode solver in which partitions π_1, π_2, and π_8 are replicated on the second core. Appendix A lists the formulation of the key constraint that checks if the slots under consideration meet a partition's π_i requirements of C_i^s and MA_i as described in the Section 5.3 using a number of bandwidth levels with equal memory budget distribution. It demonstrates the feasibility of

Table 7 "Reverse-engineered" core-local execution time and minimum constant memory bandwidth that needs to be using reserved using existing interference-sensitive WCET computation [20, 22] for the HTAWS application.

Partition	Computed core-local ET C_i^s (ms)	Min. mem. b/w (in %) reqd. throughout partition considering δ_1 latency
π_1	4.72	4.88
π_2	3.05	7.03
π_3	2.79	14.74
π_4	4.45	100.00
π_5	3.64	100.00
π_6	3.34	15.66
π_7	4.45	100.00
π_8	2.15	9.17

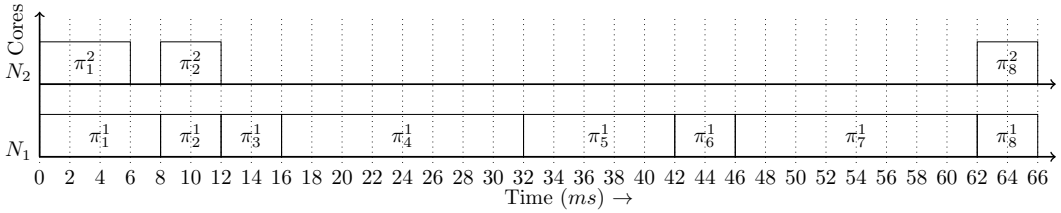

Figure 5 Partition schedule generated through Gecode constraint solver using our proposed method for the HTAWS application with some replicated partitions, such that existing HTAWS schedule is preserved.

scheduling and computing execution time offline, in the same step, under dynamic memory bandwidth, overcoming the inter-dependency challenge. At runtime, our proposed server-based runtime mechanism, described in Section 4, will execute the offline-computed scheduling decisions including assigning of the server budgets on each core in each slot.

Complexity. Slot duration plays a major role, for a larger slot duration reduces the length of the major time frame H (in slots), thereby restricting the search space. Further, the search space also depends on the number of cores, the number of partitions, and the different server budgets allowed. Due to the computational complexity of the problem, in the future, we plan to integrate our proposed method in our in-house heuristic-based offline scheduler.

6.2 Execution Time

We use the Qualcomm P4080 platform with eight e500mc cores. It is widely used by various groups conducting academic and industrial research.

6.2.1 Integration of Runtime Mechanism

We integrated our proposed runtime mechanism in a proprietary OS for research by Airbus that runs on the P4080 platform. For each core's processing time server, we use the multicore programmable interrupt controller (MPIC) timer, a hardware timer with auto-reload feature. It ensures slot-level synchronisation amongst all cores and provides a unique interrupt

Table 8 Number of memory accesses MA_i for EEMBC benchmarks obtained using configuration 1. Core-local execution time C_i^s is obtained by a "reverse-engineering" step.

Benchmark	MA_i	C_i^s (ms)
cacheb	1725	5.39
a2time	659	13.35
rspeed	591	0.621

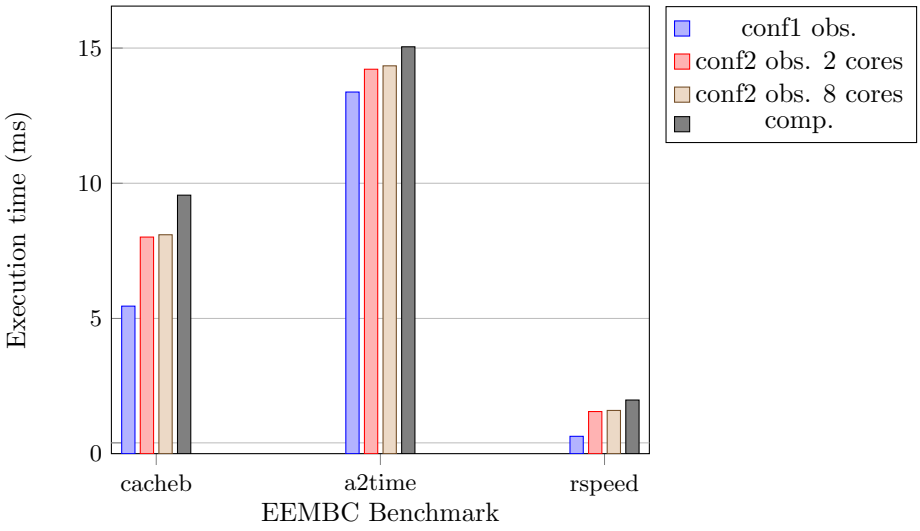

Figure 6 Execution time for EEMBC benchmarks: maximum observed values in configuration 1, maximum observed values in configuration 2 using our proposed method for 2 active cores and 8 active cores, and the computed execution time using our method based on the budgets assigned per slot in configuration 2.

instance [7] to each core on each interrupt. For the memory access server, we use the core-level hardware performance counter that counts the requests to the on-chip network [7] from each core.

6.2.2 Testbed Setup

In our experiments, we configure the platform clock frequency to 600 MHz and each core's clock frequency to 1200 MHz. The hardware timers used: MPIC timer, and Timebase timer, are configured to 37.5 MHz. The Timebase timer is used to generate timestamps for the benchmark under test. L1 and L2 caches are enabled, while L3 caches are disabled. We use both the available memory controllers.

6.2.3 Experiments

We ran benchmarks from the EEMBC AutoBench benchmark suite [28] in two different configurations for 10 runs each. Configuration 1 runs the benchmark in isolation with no memory throttling to obtain largest observed execution times and number of memory accesses. This configuration is required to obtain core-local execution time for each benchmark. Configuration 2 integrates our proposed server-based mechanism with $duration(S) = Q_{sp}^1 = 1ms$ and runs each of the benchmark on one core and a contending benchmark (matrix) replicated on the rest of the active cores. The memory budgets for the core executing the

benchmark under test are generated randomly for each of the 10 slots which are repeated in a schedule table. Thus, the schedule table contains 10 dynamic memory bandwidth levels. The memory budgets for the other active cores in each slot relate to the first core's memory budgets computed using Equation (2) with equal budget assignment. Table 8 shows the maximum number of memory accesses observed for each benchmark using configuration 1. This data is used to obtain core-local execution times by "reverse-engineering" using memory latencies from Table 3 (row 3). Figure 6 shows the largest observed execution time in configuration 1, in configuration 2 with 2 active cores and 8 active cores, and the computed execution time using our method based on the budgets assigned per slot in configuration 2. It shows that the observed execution times are less than or equal to the computed execution times using our method. We also observed per stall overhead due to the processing time server and memory access server of 10μs which is 1% of the considered slot duration of 1ms [21]. All measured execution times shown in Figure 6 exclude these stall overheads.

7 Related Work

Over the last years, several different scheduling approaches for COTS multicores have been presented. Schranzhofer et al. [26], Pellizzoni et al. [24], Boniol et al. [4] proposed deterministic execution models to control the access to shared resources. The basic concept is to divide program execution into multiple phases and restrict their capabilities. Schranzhofer et al. [26] proposed to divide each task into superblocks where each superblock consists of three phases: acquisition phase, execution phase and replication phase. A task is allowed to access the shared hardware resource only in acquisition and replication phases. Further, no two co-executing tasks on different cores can have overlapping acquisition and replication phases. Pellizzoni et al. [24] proposed the PRedictable Execution Model (PREM) for single-core COTS processors, introducing co-scheduling for shared resources. It splits each task into a sequence of non-preemptable intervals: predictable intervals, and compatible intervals. Predictable intervals are used to pre-load all data and instructions into local caches, while system calls and interrupt preemptions are prohibited. System calls and interrupt preemptions are, however, allowed in compatible intervals. Each predictable interval is further divided into two phases: execution phase and memory phase. Traffic from peripheral devices is only permitted during the execution phase of a predictable interval, resulting in an architecture with very few contentions for accesses to the shared resources. Boniol et al. [4] presented a sliced execution model. It splits each task into sub-tasks and each sub-task further into execution slices and communication slices. The access to a shared resource is only allowed in a communication slice. Such approaches are known to poorly utilise the respective resource [15]. In addition, to support transition from single-core processors to COTS multicore processors, such approaches require modification of the source code of legacy applications [11], which is a non-trivial step.

Another popular approach is joint analysis. To address sharing of resources those approaches analyse the program flows on all cores using the considered shared resource. Therefore, detailed knowledge of the state of execution is required. Yan and Wang [29] applied WCET analysis to multicore processors with shared L2 caches by analysing inter-thread dependencies. The analysis is based on the program control flow and accounts for all possible conflicts on the shared cache. Li et al. [16] extended the analysis by identifying possibly overlapping threads. Hardy et al. [13] further extended it by reducing the number of possible conflicts between overlapping threads. Chattopadhyay et al. [6, 5] and Kelter et al. [14] proposed combined shared cache and shared bus analysis and applied a TDMA bus

arbitration. Chattopadhyay et al. [5] combine cache and bus analysis with other architectural features such as pipelines and branch prediction. Overall, the joint analysis approaches have to explore huge state spaces due to the various possible interactions between different tasks, resulting in huge computational complexity.

With respect to monitoring, Bellosa [3, 2] introduced the idea to leverage built-in processor counters to acquire additional task runtime information. Yun et al. [32] proposed controlling memory accesses from all but one cores to limit contentions experienced by hard real-time tasks executing on one core to ensure that they meet their deadlines. Yun et al. extended the work in [31] by introducing a memory throttling mechanism – MemGuard, that regulates memory accesses using a memory server on each core. It assumes that memory bandwidth is statically partitioned between cores before runtime for safety-critical systems. Behnam et al. [1] propose a method to isolate the behaviour of different cores. They apply a hierarchy of servers to all cores using a server-based approach which assigns a certain limit on the amount of cache misses. Yao et al. [30] present a method to bound variability in execution time of each task on all cores considering round-robin arbitration between cores for memory accesses. Mancuso et. al. [18] present a method to compute the WCET under static partitioning of memory bandwidth between cores, which takes into account the amount of locked data in caches. The main difference of our proposed method against these works is our focus on dynamic memory bandwidth isolation and guarantees for safety-critical systems. Also, the goal of preserving static schedule on a core differentiates this paper from other server-based approaches. Moreover, most of these works do not consider additional arbitration and contention delay introduced by the shared on-chip network and the DRAM memory controller in the analysis.

The authors in [20, 22] consider a process frame that consists of co-executing tasks and introduced the interference-sensitive WCET (is-WCET) computation that computes, in offline phase, the execution time of each process considering contentions from the worst-case number of memory accesses from each of co-executing processes in a process-frame. Our proposed method is based on the same is-WCET computation, but allows a finer granularity of resource control (slots) instead of partition level. This enables our method to further relax the strong start time constraints imposed by [20, 22], that is, a new process can only be scheduled after the completion of all processes in a process frame.

8 Conclusion and Future Work

In this paper, we presented a method for a step in the investigation of Airbus using COTS multicores for safety-critical avionics applications. It addresses the need of preserving the ARINC 653 single core schedule of a Helicopter Terrain Awareness and Warning System application while scheduling additional safety-critical partitions on the other cores. As some partitions are memory-intensive, dynamic memory bandwidth isolation is needed, which requires performing the computation of execution times and scheduling together.

Our method solves this problem for slot-based time-triggered systems using a number of dynamic memory bandwidth levels. The method is NoC and DRAM controller contention-aware and is based on the existing interference-sensitive WCET computation and memory bandwidth throttling mechanisms and does not require application source-code modifications. It constructs schedule tables assigning partitions and dynamic memory bandwidth to each slot on each core. Then at runtime, two servers – for processing time and memory bandwidth – run on each core, jointly controlling the contention between the cores and the amount of memory accesses per slot. Thus, the number of active cores can vary over time.

As a proof of concept, we generated schedule tables performing executing time computation and scheduling in the same step using a constraint solver. The basic concepts can be included in a variety of offline algorithms for schedule table construction. We considered a generic case for execution time computation with scheduling that can be used, e.g., in search based algorithms.

We implemented our proposed runtime mechanism part in a proprietary research operating system from Airbus and EEMBC benchmarks, executed on a Qualcomm P4080 multicore platform, demonstrating its practicality. We evaluated our runtime mechanism and execution time computation under dynamic memory bandwidth levels and showed that the observed execution times are less than or equal to the computed execution times.

For future work, we will turn the focus away from the specific application and hardware settings in this paper to generalise our method. Instead of the proof-of-concept constraint solver used here, we will extend our in-house heuristic-based offline scheduling framework to integrate the proposed offline phase. This will allow us to focus on the complexity performance of the offline part of the method and its integration in existing tool chains.

Acknowledgements. We are grateful to Bernd Koppenhöfer and Max Gapp from Cassidian for the case study. We also thank the anonymous reviewers for their valuable feedback. We would also like to express our gratitude to Björn Brandenburg and the members of the RTS group at TUK for their helpful suggestions. We also thank Claire Pagetti, Daniel Gracia Pérez, and Rob Davis for the enriching discussions.

References

1 Moris Behnam, Rafia Inam, Thomas Nolte, and Mikael Sjödin. Multi-core composability in the face of memory-bus contention. *SIGBED Rev.*, 10(3):35–42, October 2013. doi:10.1145/2544350.2544354.

2 Frank Bellosa. Memory access – the third dimension of scheduling. Technical report, University of Erlangen, 1997.

3 Frank Bellosa. Process cruise control: Throttling memory access in a soft real-time environment. Technical report, University of Erlangen, 1997.

4 Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic Execution Model on COTS Hardware. In Andreas Herkersdorf, Kay Römer, and Uwe Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2012: 25th International Conference, Munich, Germany, February 28 – March 2, 2012. Proceedings*, pages 98–110, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-28293-5_9.

5 S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A Unified WCET Analysis Framework for Multi-core Platforms. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 99–108, April 2012. doi:10.1109/RTAS.2012.26.

6 Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling Shared Cache and Bus in Multi-cores for Timing Analysis. In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES'10, pages 6:1–6:10, New York, NY, USA, 2010. ACM. doi:10.1145/1811212.1811220.

7 Freescale Semiconductor. *e500mc Core Reference Manual Rev. 3*, 2013.

8 Inc. Freescale Semiconductor. P4080: QorIQ P4080/P4040/P4081 Communications Processors with Data Path, 2013. URL: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080.

9 R. Fuchsen. How to address certification for multi-core based IMA platforms: Current status and potential solutions. In *29th Digital Avionics Systems Conference*, pages 5.E.3–1–5.E.3–11, Oct 2010. doi:10.1109/DASC.2010.5655461.

10 Generic constraint development environment. URL: http://www.gecode.org/.

11 S. Girbal, X. Jean, J. Le Rhun, D. G. Perez, and M. Gatti. Deterministic platform software for hard real-time systems using multi-core COTS. In *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*, pages 8D4–1–8D4–15, Sept 2015. doi:10.1109/DASC.2015.7311481.

12 Airbus Group. Future of urban mobility: My kind of flyover. URL: http://www.airbusgroup.com/int/en/news-media/corporate-magazine/Forum-88/My-Kind-Of-Flyover.html.

13 D. Hardy, T. Piquet, and I. Puaut. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *2009 30th IEEE Real-Time Systems Symposium*, pages 68–77, Dec 2009. doi:10.1109/RTSS.2009.34.

14 T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 3–12, July 2011. doi:10.1109/ECRTS.2011.9.

15 Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk. Evaluation of resource arbitration methods for multi-core real-time systems. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–10, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2013/4117, doi:10.4230/OASIcs.WCET.2013.1.

16 Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *2009 30th IEEE Real-Time Systems Symposium*, pages 57–67, Dec 2009. doi:10.1109/RTSS.2009.32.

17 Rodin Lyasoff. Welcome to Vahana, Sept. 2016. URL: https://vahana.aero/welcome-to-vahana-edfa689f2b75#.fz78tkigh.

18 R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun. WCET(m) Estimation in Multi-core Systems Using Single Core Equivalenc. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 174–183, July 2015. doi:10.1109/ECRTS.2015.23.

19 J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 132–143, May 2012. doi:10.1109/EDCC.2012.27.

20 J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 109–118, July 2014. doi:10.1109/ECRTS.2014.20.

21 Jan Nowotsch. *Interference-sensitive Worst-case Execution Time Analysis for Multi-core Processors*. PhD thesis, University of Augsburg, 2014.

22 Jan Nowotsch and Michael Paulitsch. Quality of service capabilities for hard real-time applications on multi-core processors. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS'13, pages 151–160, New York, NY, USA, 2013. ACM. doi:10.1145/2516821.2516826.

23 NXP. P5020: QorIQ® P5020 and P5010 64-bit Dual- and Single-Core Communications Processors, Sept. 2016. URL: http://www.nxp.com/products/microcontrollers-and-processors/power-architecture-processors/qoriq-platforms/p-series.

24 R. Pellizzoni, E. Betti, S. Bak, Gang Yao, J. Criswell, M. Caccamo, and R. Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *Real-Time and*

Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE, pages 269–279, April 2011. doi:10.1109/RTAS.2011.33.

25 Wind River. Wind River VxWorks 653 Platform, 2015. URL: http://www.windriver.com/products/product-overviews/vxworks-653-product-overview/vxworks-653-product-overview.pdf.

26 A. Schranzhofer, R. Pellizzoni, Jian-Jia Chen, L. Thiele, and M. Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 213–222, April 2011. doi:10.1109/RTAS.2011.28.

27 Certification Authorities Software Team. *Position Paper CAST-32A Multi-core Processors*. US Federal Aviation Administration, Nov. 2016.

28 Inc. The Embedded Microprocessor Benchmark Consortium. EEMBC AutoBench 1.1 benchmark software, 2013. URL: http://www.eembc.org/benchmark/automotivesl.php.

29 J. Yan and W. Zhang. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 80–89, April 2008. doi:10.1109/RTAS.2008.6.

30 Gang Yao, Heechul Yun, Zheng Pei Wu, R. Pellizzoni, M. Caccamo, and Lui Sha. Schedulability analysis for memory bandwidth regulated multicore real-time systems. *Computers, IEEE Transactions on*, 65(2):601–614, Feb 2016. doi:10.1109/TC.2015.2425874.

31 H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, Feb 2016. doi:10.1109/TC.2015.2425889.

32 Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308, July 2012. doi:10.1109/ECRTS.2012.32.

A Core-local Execution Time and Memory Accesses Constraints

Constraint (4), specified in our constraint-solver model, is a key constraint that ensures that the number of slots allocated to a partition meet its requirements of core-local execution time and the number of memory accesses, under fixed number of dynamic memory bandwidth levels with equal budget distribution between the active cores in each slot.

$slot1(i)$ represents the number of slots with memory budget $Q^1_{sm} = 41379$ considering 1 active core assigned to a partition π_i. $slot2(i)$ represents the number of slots with memory budget $Q^2_{sm} = 20338$ considering 2 active core assigned to a partition π_i.

$$
\begin{aligned}
&\forall i \in \Gamma\ , \\
&\left(slot1(i) == 0\ \wedge\ slot2(i) == \left\lceil \frac{C^s_i}{Q^1_{sp}} + \frac{MA_i}{Q^2_{sm}} \right\rceil\right) \\
\vee &\left(slot1(i) == \left\lceil \frac{C^s_i}{Q^1_{sp}} + \frac{MA_i}{Q^1_{sm}} \right\rceil\ \wedge\ slot2(i) == 0\right) \\
\\
&\quad (slot1(i) > 0\ \wedge\ slot2(i) > 0) \\
\vee &\quad \wedge \left(slot1(i) \geq C^s_i\ \wedge\ part(i) == slot1(i) * Q^1_{sm} - C^s_i * Q^1_{sm}\right) \\
&\quad \wedge \left(slot2(i) * Q^2_{sm} + part(i) - MA_i < Q^2_{sm}\right) \\
&\quad \wedge \left(slot2(i) * Q^2_{sm} + part(i) - MA_i \geq 0\right) \\
\\
&\quad slot1(i) < C^s_k \\
&\quad \wedge \left(part(i) == -slot1(i) * Q^1_{sm} + C^s_i * Q^1_{sm}\right) \\
\vee &\quad \wedge \left(slot2(i) + \frac{part(i)}{Q^1_{sm}}\right) * Q^2_{sm} - MA_i < Q^2_{sm} \\
&\quad \wedge \left(slot2(i) + \frac{part(i)}{Q^1_{sm}}\right) * Q^2_{sm} - MA_i \geq 0
\end{aligned}
\tag{4}
$$