Scheduling for Mixed-criticality Hypervisor Systems in the Automotive Domain

C. Evripidou
Department of Computer Science,
University of York, UK.
Email: ce514@york.ac.uk

A. Burns
Department of Computer Science,
University of York, UK.
Email: alan.burns@york.ac.uk

***Abstract*—Virtualisation has been proposed for use in the automotive domain as it has the potential to reduce the number of ECUs (Electronic Control Units) that are required in a modern vehicle. In this paper we first introduce a visualisation architecture that makes use of two different types of execution-time servers to provide separation, low run-time overheads but short response-times for event-triggered computation. This model is then extended to mixed-criticality systems and utilises a run-time switch between the task-server mapping to enhance schedulability (at the cost of extended response-times). An industrial case study is used to evaluate the approach.**

I. Introduction

The automotive industry has been using software in cars for over 30 years, but this is now increasing at a very fast pace [8], [9]. The software in a vehicle typically runs within Electronic Control Units (ECUs), which are embedded hardware platforms responsible for controlling different subsystems within a vehicle. Examples of ECUs are door control units, transmission control units and engine control units.

As stated by Broy [8] in 2006, about 40% of the production cost of a vehicle was spent on electronics and software. After the 30-year growth of the volume of software in vehicles, a modern premium car may contain in excess of 100 processors spread across 70 ECUs [3], [4], [8], [9]. Nolte [18] studied this trend; there is a clear indication of the increased complexity and the added hardware costs that are prominent in modern vehicles.

When they were first introduced, ECUs were functionally independent and were connected solely to sensors and actuators [9]. As ECUs were required to provide additional functionality, there was a need to establish communication channels between them. This change has resulted in multiple ECUs cooperating to provide a certain piece of functionality. In addition to numbers, ECUs can be heterogeneous and be responsible for different types of tasks; hard real-time (vehicle control) and soft real-time (infotainment). Given the large number of ECUs per vehicle, the dependencies between them and their lack of homogeneity, it can be inferred that they form a complex system that is hard to reason about.

Apart from the structural complexity of the electronic parts of modern vehicles, another challenge in software engineering for the automotive domain is the difference in lifetime of car models and ECU hardware [9], [20]. Specifically, a car model typically has a production lifetime of about 7 years, whereas a microprocessor 5 years. In addition to the 7 years of production lifetime, a car manufacturer (OEM) needs to provide service and spare parts for an additional 15 years. The result of this difference in lifetimes is that during a car's lifetime it is very likely that some ECU hardware components stop being available in the market. Given that the ECU software is usually highly optimised for the underlying hardware, porting to a newer platform can be a hard and expensive task.

In order to deal with the increased complexity that characterises modern vehicles, OEMs, suppliers and other relevant companies formed a worldwide development partnership. The result of this partnership is the Automotive Open System Architecture (AUTOSAR) [6]. AUTOSAR aims to provide a common architecture as well as a methodology that will help with the understanding of the interaction of ECUs, allow software reuse and enable the combination of multiple functions on a single ECU [14].

In this paper we consider the use of virtualisation (see Section II) for the integration of multiple ECUs into a single hardware platform. Section III outlines the general scheduling approach followed for virtualisation. The key characteristic of the scheduling approach is the use of a deferrable and a periodic server per partition for the execution of event-driven and time-driven tasks; event-driven tasks execute on deferrable servers for low response times and time-driven tasks execute on periodic servers for the improved utilisation and lower overheads. The main contribution of this paper is the extension of the proposed scheduling approach for virtualisation to support two modes of degradation (**D1** and **D2**). The mixed criticality model is presented in Section IV All the work presented was done in close cooperation with the automotive company ETAS. They provided the requirements and application code that was used to form the presented model, and derive the case study of Section V, which was used to evaluate the proposed approach.

II. Virtualisation

Virtualisation is a technique, initially developed in the early 60's [10], where logical resources are created in order to allow one or more applications to execute on the same hardware platform. The logical resources are created and managed by the hypervisor (HV), also referred to as virtual machine manager (VMM). From our experience with ETAS, there is

increasing interest in the automotive industry for the use of virtualisation to alleviate some of their current problems.

The main use case for HV technology in the automotive domain is the reduction of ECU count by combining multiple ECUs on a single hardware platform. The key properties that must hold in a HV system is spatial and temporal isolation of the VMs. Spatial isolation is achieved by prohibiting the VMs from accessing memory areas outside of their memory space. Temporal isolation, which is the focus of the research reported in this paper, is the property under which a VM's behaviour cannot cause another VM to violate its real-time properties.

Early work on virtualisation by Popek and Goldberg [19] in 1974 identify three characteristics of an HV. First, an HV needs to be able to provide its hosted virtual machines (VMs) with an execution environment which is indistinguishable from real hardware. Second, execution is efficient by mapping a large subset of the virtual processor instruction set to a physical processor. Third, the HV has complete control over all hardware resources and is able to allow or prohibit VMs access, according to their configuration.

Popek and Goldberg [19] also identify a set of properties for HVs:

- *Efficiency:* all non-privileged instructions are executed directly on hardware.
- *Resource control:* it is impossible for a VM to interfere with any system resources that are not allocated to it.
- *Equivalence:* a VM produces the same results when executing as if it was executing without a HV.

The HV characteristics by Popek and Goldberg [19] refer to full virtualisation. Full virtualisation implies that the applications in VMs can be executed without requiring any modifications [13]. In order to maintain the properties identified above in a fully virtualised environment it is necessary to have adequate hardware support. Specifically, allowing a VM to execute directly most of the time on the underlying hardware for efficiency requires that the HV will be able to identify attempts to execute privileged instructions. The HV is then responsible for checking whether the VM is allowed to perform the operation it attempted to and act accordingly.

Paravirtualisation was introduced as means of alleviating the lack of hardware support and to simplify the development of the HV. Specifically, in a paravirtualised environment, VMs execute directly on hardware using modified versions of their application code [13], using HV calls to replace the functionality of privileged instructions. Examples of virtualisation platforms are: OKL4 [15], XtratuM [2], Xen [7] and PikeOS [21].

III. Requirements and Approach to Virtualisation

The driving requirements for the architecture described in this paper is: the need to partition the code of the applications, the need to have low run-time overheads (the automotive industry is very dependent of the efficient use of hardware resources), and the need to provide very fast response-times for a certain class of application code (namely, event-triggered computation). The latter two requirements are contradictory, fast response times require small containers, efficient implementation dictates the use of larger containers. This dilemma is resolved by the use of two different types of container.

A. Task Model

Traditional hypervisor scheduling approaches were developed assuming no visibility at the task level in the partitions. In this section we define a flexible task model, which allows exploiting visibility, where that is available, taking into account implementation overheads. First, we classify tasks as synchronous or asynchronous. Synchronous tasks are strictly periodic, whereas asynchronous are event-triggered tasks with a known minimum inter-arrival time. The main motivation behind the proposed scheduling method is that asynchronous tasks require quick response times, while synchronous ones can be serviced in a more efficient, lower overheads approach. All operations performed by the hypervisor are executed in a non-preemptive manner and are described as highly predictable pieces of code.

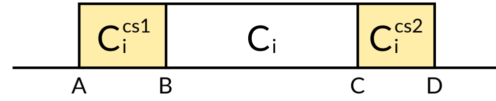

Fig. 1. Task structure.

The execution of all tasks is defined to be structured as shown in Figure 1. A task τ_i is defined by the tuple $(C_i^{cs1}, C_i, C_i^{cs2}, T_i, P_i)$:

- C_i^{cs1}: the scheduling and context switching overheads required before the execution of the main task body. In asynchronous tasks, this section is performed by the hypervisor and is therefore non-preemptible. Synchronous tasks can be preempted by the hypervisor during this section but not by other synchronous tasks of the same partition, since this operation is performed by the partition.
- C_i: the time required for the task's main body to execute. Partition tasks (synchronous and asynchronous) can be preempted while in this section, however hypervisor tasks run non-preemptively.
- C_i^{cs2}: the overheads of terminating the execution of the task. The preemption rules that apply for C_i^{cs1} also apply during the execution of this section.
- T_i: the period or minimum inter-arrival time of the task.
- P_i: the priority level of the task.

B. Execution Servers

Our approach aims to minimise the response time for event-triggered tasks, while at the same time maximise schedulability and enforce temporal protection between the different partitions[1]. The CPU time is shared between the partitions using execution servers. Temporal protection is achieved by

[1]A similar approach by Missimer et al. [16] using sporadic and priority inheritance bandwidth preserving servers was published during the development of the approach discussed in this paper. Although they also use servers, they do not have the same approach to mixed-criticality.

prohibiting partitions to execute for more than their servers' capacity. Each server is associated with a hypervisor task, which is responsible for replenishing its capacity.

Asynchronous tasks are released in response to events and therefore need to be serviced with the lowest possible latency. To facilitate this requirement, asynchronous tasks execute using deferrable servers that are assumed to always have enough capacity to service all event-driven tasks, given their WCET, hypervisor overheads and period. With the use of a deferrable server, no server capacity is expended when the system is idle, and events are serviced as they arrive, provided they have the highest priority in the system.

Deferrable servers (DS) offer low response time for the asynchronous tasks, however they are inferior in terms of schedulability in comparison to periodic servers. The time-driven or synchronous tasks in the system are execute using a periodic server (PS) in order to alleviate this trade off, therefore improving schedulability, without compromising on the low latency required by event-driven tasks.

The association between servers and tasks is defined using the association matrix M. The rows of the matrix represent the tasks in the system, whereas the columns are the servers. All elements can take the values 0 or 1. If a task τ_i is serviced by s_j, then $M_{\tau_i,s_j} = 1$, otherwise $M_{\tau_i,s_j} = 0$. Moreover, a task can be serviced by exactly one server, which implies that the sum of each row results is at most 1.

$$M = \begin{array}{c} \\ \tau_0 \\ \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \\ \tau_7 \end{array} \begin{array}{c} \begin{array}{ccccc} HV & DS_0 & DS_1 & PS_0 & PS_1 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \quad (1)$$

Equation (1) is an example configuration of an association matrix of a simple system with two partitions (applications), p_0 and p_1. Each of the partitions is associated with a deferrable and a periodic server. Specifically, DS_0 and PS_0 are associated with partition p_0 and DS_1 and PS_1 are associated with partition p_1. Each of the servers requires a hypervisor task so that its capacity is replenished periodically. The hypervisor tasks that are responsible for replenishing the server's capacity are τ_0, τ_1, τ_2 and τ_3. Partition p_0 has two tasks, τ_5 and τ_6, and p_0 also has two tasks, τ_4 and τ_7. τ_4 and τ_5 are asynchronous tasks and are therefore associated with DS_0 and DS_1 respectively. Similarly, τ_6 and τ_7 are synchronous tasks and are associated with PS_0 and PS_1 respectively.

C. Priority Space

Figure 2 shows the relationship between the execution modes in terms of their corresponding priority levels. The hypervisor executes in hypervisor mode at the highest system priority region. Since the hypervisor's code is trusted and is

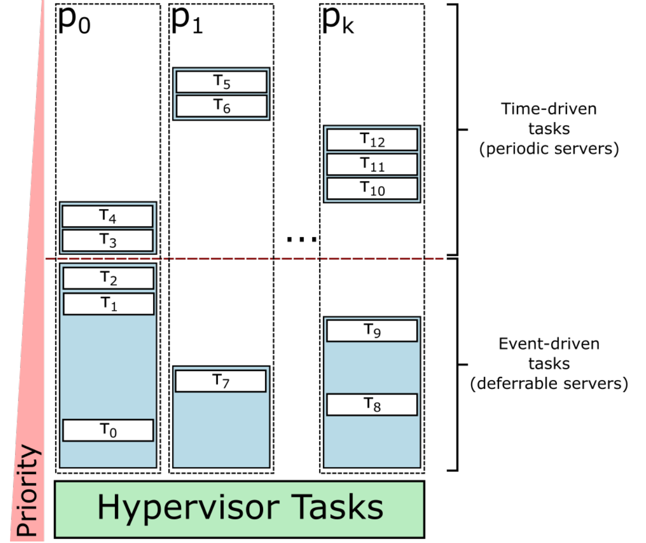

Fig. 2. Example of a k-partition system priority space.

typically consisted of short, highly predictable non-preemptive tasks. The motivation behind this approach is to allow event-triggered tasks, such as the service routines of exceptions or interrupts, to execute at a priority level that is strictly higher than any time-triggered tasks.

Synchronous tasks have the lowest priority range in the system. Specifically, a partition executes in synchronous mode if no event is pending and eligible to be handled. The eligibility of handling an event is directly associated with the asynchronous budget. A partition executing in synchronous mode is responsible for doing its own task scheduling and may therefore maintain its own internal priority space.

The analysis employed with the defined model uses standard response-time analysis to ensure:

- each server (DS and PS) is schedulable, and
- each task within each server is schedulable.

For space reasons the analysis is not given in this paper (see [12]).

IV. Approach to Mixed Criticality

Having now introduced the approach to virtualisation we can describe how this model is extended to support mixed criticality. Separation is already supported. We now wish to provide a means by which criticality mode changes can be accommodated.

The introduction of safety in the automotive industry with the ISO26262 standard on functional safety of road vehicles [1] gave rise to concerns regarding the integration of components with different ASIL (Automotive Safety Integrity Level) in the same ECU. AUTOSAR enables the integration of SWC (Software Components) from different vendors, requiring modifications to the configuration of the RTE (Runtime Environment) and BSW (Basic Software) [17]. From a safety-critical perspective AUTOSAR lacks the required separation mechanisms. Specifically a failure in an AUTOSAR SWC

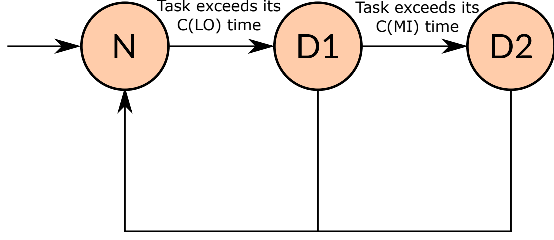

Fig. 3. State transitions for the mixed-criticality model.

typically results in an ECU reset. In the case where all SWCs are of the same ASIL this is acceptable, however in the case of mixed-criticality this could potentially allow a low criticality component to interfere with a higher criticality one. This directly violates *freedom from interference* (FFI), as dictated by ISO 26262-6:2011 Annex D [1]. Moreover, Esper et al. [11] state that suspending the execution of lower criticality tasks may hinder the certification progress due to lack of isolation unless the suspended tasks are non-critical.

A possible way of achieving FFI is to ensure that components of different criticality levels are located on separate physical ECUs. Failures in lower criticality components would therefore be isolated and not propagate to other ECUs of higher criticality. This approach could potentially increase the number of physical ECUs, which, as identified earlier, is one of the main drivers to the high development costs for software and hardware in vehicles. The use of a hypervisor can provide the necessary isolation between its partitions to allow the integration of multiple ECU images on a single physical ECU, while enforcing FFI.

The proposed mixed-criticality model supports three levels of criticality, LO, MI, HI, with $HI > MI > LO$. The task model is defined in a similar manner as the model proposed by Vestal [22]. A task τ_i is defined by the tuple $(\vec{C}_i^{cs1}, \vec{C}_i, \vec{C}_i^{cs2}, \vec{T}_i, \vec{P}_i, L_i)$, where:

- $\vec{C}_i^{cs1}$: A vector containing the WCET of the implementation overheads before the execution of the task's main body for each criticality level, $C_i^{cs1}(LO)$, $C_i^{cs1}(MI)$ and $C_i^{cs1}(HI)$.
- $\vec{C}_i$: A vector containing the WCET of the task's main body at each criticality level. $C_i(LO) \leq C_i(MI) \leq C_i(HI)$.
- $\vec{C}_i^{cs2}$: The WCET of the overheads after the execution of the task's main body for each criticality level, $C_i^{cs2}(LO)$, $C_i^{cs2}(MI)$ and $C_i^{cs2}(HI)$.
- $\vec{T}_i$: The period of the task at each level of criticality, $T(LO) \geq T(MI) \geq T(HI)$.
- $\vec{P}_i$: The priority level of the task at each criticality level, $P_i(LO)$, $P_i(MI)$, $P_i(HI)$.
- L_i: The criticality level of the task (ie. LO, MI, HI).

A. Execution Modes

Figure 3 summarises the mode transitions performed by the system to support multiple levels of criticality. A requirement of implementing the mixed-criticality model is to monitor the execution time of all partition tasks. As the figure illustrates as a task executes for more than its $C(LO)$ value the first degraded mode is entered; if necessary a second level of degradation is entered. A distinctive aspect of the model is that **D1** does not involve the dropping of any tasks. Rather, it invokes an architectural change in which the event-triggered tasks executing in deferrable servers are migrated to their corresponding partitions' periodic servers. Hence all tasks continue to execute but the event-triggered work has increased latency; the periodic work continues to meet all deadlines. If a further degradation is required then D2 does involve the dropping of low criticality work (following the model of Vestal [22]).

V. Evaluation

The proposed approach was heavily motivated by the requirements of the automotive industry and takes into account implementation overheads. In this section a case study, derived from application code provided by ETAS Ltd., is used to evaluate the proposed approach.

The case study evaluation was performed using ECU application code that was provided by ETAS Ltd. The application code consisted of a set of AUTOSAR TASKs of a Mercedes-Benz M160 engine controller. The functionality of the application code includes controlling the air flow, idle speed, and fuel injection. Each TASK has a unique period and a set of sub-tasks that are to be executed at that period. The provided taskset was split into two partitions with different criticality levels by an expert with respect to the individual task functionality. Each task was then classified as asynchronous or synchronous, with respect to their real-time requirements.

A. Task Measurement

1) Application Tasks: The next step for the case study is to analyse the provided ECU code, in order to realise the proposed model. The source contained only application code, without the underlying OS. The minimum information that is required for the proposed model is the period of each task, which was provided and the execution time. The execution times of the tasks were obtained using a measurement-based approach. The first part of the timing analysis was to study the provided code. Upon inspection, the code was primarily linear with minimal branching. Numeric calculations and variable conversions were the primary operations performed by the code. The input/output of the tasks was made by reading/writing to external variable, each with a clearly defined range of valid values.

Measuring the task execution times required modifications to the provided code, since the OS code was not available. Specifically the code was modified with definitions of all the external variables and structures, that were initialised using valid values. The granularity of the timer was insufficient for the measurement of the short task execution times, therefore each task execution time was estimated by measuring the amount of time required to execute it one million times in sequence. Each such measurement was then taken 500 times,

Task	Partition	T (ms)	Max	Mean	Std
τ_0	p_0	100	516.06	518.49	0.42
τ_1	p_0	10	3351.14	3641.27	64.65
τ_2	p_0	100	955.6	959.29	0.74
τ_3	p_1	10	188.55	189.35	0.16
τ_4	p_0	100	612.29	615.18	0.52
τ_5	p_0	1	221.49	222.28	0.16
τ_6	p_1	1000	444.86	447.13	0.24
τ_7	p_1	100	123.99	124.65	0.13
τ_8	p_0	10	361.33	362.95	0.31
τ_9	p_0	10	361.36	363.6	0.37
τ_{10}	p_0	1	497.33	499.7	0.45
τ_{11}	p_0	1	361.41	363.16	0.34
τ_{12}	p_0	100	420.06	424.44	0.38
τ_{13}	p_1	10	361.39	363.38	0.34
τ_{14}	p_0	100	419.97	422.01	0.34
τ_{15}	p_0	100	1035.55	1039.08	0.59
τ_{16}	p_0	100	248.28	249.26	0.18
τ_{17}	p_0	100	1084.03	1088.36	0.63
τ_{18}	p_1	100	2523.01	2537.88	2.97
τ_{19}	p_1	50	361.39	363.18	0.32
τ_{20}	p_1	100	338.45	340.07	0.27
τ_{21}	p_0	100	372.68	374.8	0.32
τ_{22}	p_0	1	343.02	344	0.19
τ_{23}	p_0	20	1325.86	1330.68	0.89
τ_{24}	p_1	10	480.63	483.01	0.4
τ_{25}	p_0	1	460.15	462.09	0.24
τ_{26}	p_0	100	173.33	174.15	0.14
τ_{27}	p_0	10	203.72	204.47	0.16
τ_{28}	p_0	10	492.54	493.55	0.21
τ_{29}	p_0	10	505.74	507.78	0.24
τ_{30}	p_0	20	2351.58	2373.03	3.73
τ_{31}	p_0	100	755.56	758.3	0.26

TABLE I
APPLICATION TASK TEMPORAL CHARACTERISTICS AND PARTITION MAPPING.

therefore obtaining a sample of 500 estimated execution times per task.

Table I summarises the sampled execution times in ns. From the table, there is a small difference between max and mean as well as a small standard deviation of the execution times. This confirms that there is little variation between each measured execution time, as expected from the minimal branching that was observed during the inspection of the code. Analysis using Pearson's r showed no correlation between the WCET and the period of tasks, $r(30) = -0.0622$, $p = 0.735$ (two-tailed test).

B. Hypervisor Overheads

Hypervisor Overhead	WCET
Forward interrupt	363ns
Return from interrupt	139ns
Replenish server capacity	553ns
Mode change	645ns

TABLE II
HYPERVISOR OVERHEADS FOR THE MIXED-CRITICALITY MODEL.

To calculate the hypervisor overheads, a partial hypervisor implementation was built. The hypervisor overheads in the system are responsible for the replenishment of the server capacity and handling forwarding and returning from interrupts. The WCETs of the developed hypervisor components were calculated using the maximum cycles required for the ARM1176JZF-S instructions to execute [5]. Table II summarises the hypervisor overheads. These values are deemed applicable to all criticality modes. Therefore, the WCET for all operations performed by the hypervisor are not scaled up for the MI and HI criticality levels.

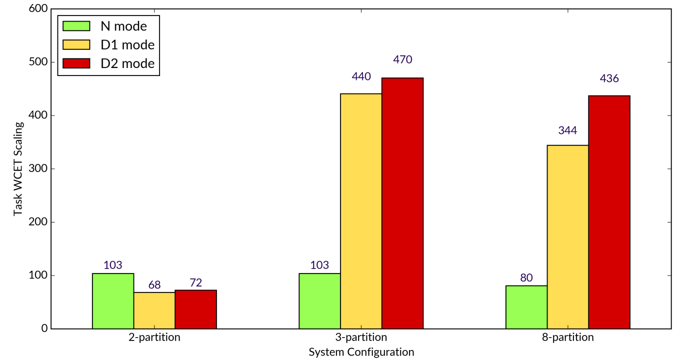

Fig. 4. Application task WCET scaling with 2, 3 and 8-partition configurations.

C. Experiments

The case study was used to undertake sensitivity analysis using the response-time analysis developed for the two server type model. The case study taskset specification in conjunction with the system overhead parameters are used to produce a system configuration that is marginally schedulable at all criticality levels.

Sensitivity analysis is then repeated for the two degraded modes, **D1** and **D2**, to see what improvement in schedulability results. The processor used for these experiments was considerable more powerful that the one from which the code WCET times were derived. High scaling factors were therefore observed. A scaling factor of 100 implies that all code could be increased 100 times and the system would still (just) be schedulable.

Three methods of supporting **D1** were investigated:

- 2-partition – here each of the two applications uses just one DS and one PS; in **D1** all tasks in the DS are moved to the PS.
- 3-partition – here each application uses just one DS and one PS; in **D1** the DS is converted to a new, distinct, PS (i.e. another partition is added).
- 8-partition – here each application uses a number of DSs and PSs; in **D1** all DSs are converted to PSs.

D. Results

In this section we discuss the results of the sensitivity analysis on the three identified system configurations.

Figure 4 shows the scaling that was achieved by each system configuration for all criticality levels. The 2-partition configuration while the system executes in **N** mode achieved a scaling factor of 103. At the degraded modes there is significant capacity loss, which makes the proposed model ineffective for the 2-partition configuration. The reason for the poor performance in **D1** and **D2** is the large variation

of the temporal requirements of the application tasks served by individual periodic servers. This variation in temporal requirements results in wasted server capacity.

The 3-partition configuration achieves the same scaling as the 2-partition configuration while the system executes in **N** mode. After the mode switch to **D1** there is a significant increase of the WCET scaling, by a factor of 3. Assigning the asynchronous tasks in a separate partition eliminated the server parameter shortcoming of the 2-partition configuration. Specifically, in the degraded modes the asynchronous tasks execute using the original periodic server. All asynchronous tasks in the system share the same minimum inter-arrival time of $1ms$. This allows for no wasted server capacity, therefore increasing the achieved scaling in the degraded modes. Switching to the second degraded mode, **D2**, there is a small increase in the scaling factor. The small increase is as expected, since the 95% of application task utilisation is used by HI criticality tasks.

The scaling achieved with the 8-partition configuration was 80, which is significantly lower than the 2 and 3-partition configurations. The lower scaling achieved for the 8-partition configuration was attributed to the large number of hypervisor tasks, which reside at the highest priority band in the system. Specifically, server capacity replenishment tasks are assigned strictly higher priorities than application tasks, therefore having a greater impact on the optimality of the priority assignment algorithm. Another contributing factor to this is the context switching overheads of asynchronous tasks. After the switch to the first degraded mode, **D1**, the achieved scaling is 310, which is the highest achieved in all three configurations. The use of solely periodic servers servicing tasks of the same period made the priority assignment algorithm very effective. Switching to **D2** results in more spare capacity than the 3-partition configuration, since additional capacity is freed due to the number of server replenishment tasks that are no longer in use.

VI. Conclusions

In this paper we have shown how two forms of execution-time servers (namely deferrable server and periodic server) can be used (with a hypervisor) to support multiple applications (partitions) running on the same processor/core. The approach derived has been in response to the needs of the automotive domain. The servers provide the necessary level of separation, and the different characteristics of the two servers allow low latencies for event-triggered tasks and low overheads for time-triggered (periodic) tasks to be manifest. The proposed approach was extended to support three levels of criticality, featuring two degraded modes (**D1** and **D2**). The mixed-criticality model aims to provide the partitions with additional CPU capacity by providing an alternative task-server mapping (**D1**) before the suspension of lower criticality tasks (**D2**). An industrial case study was used to evaluate the approach. This demonstrated the need to keep the overheads of the hypervisor as low as possible.

Future work will move the scheme to multi-core platforms, this is a strong requirement of the automotive domain, although it presents a number of significant challenges.

References

[1] ISO 26262-2:2011 road vehicles – functional safety.

[2] XtratuM, August 2012. [Accessed: 16 Feb 2016].

[3] U. Abelein, H. Lochner, D. Hahn, and S. Straube. Complexity, quality and robustness-the challenges of tomorrow's automotive electronics. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 870–871. IEEE, 2012.

[4] H. André and H. Gernot. Operating systems technology for converged ECUs. In *6th Embedded Security in Cars Conference*. ISITS, 2008.

[5] ARM Information Center. ARM1176JZF-S^{TM}technical reference manual, 2009. [Accessed: 14 May. 2016].

[6] AUTOSAR. AUTomotive Open System ARchitecture, August 2012. [Accessed: 30 Sep. 2012].

[7] P. Barham and et al. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM, 2003.

[8] M. Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.

[9] M. Broy, I.H. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356 –373, February 2007.

[10] J. P. Buzen and U. O. Gagliardi. The evolution of virtual machine architecture. In *Proceedings of the National Computer Conference and Exposition*, pages 291–299. ACM, 1973.

[11] A. Esper and et al. How realistic is the mixed-criticality real-time system model? In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 139–148, 2015.

[12] Chiristos Evripidou. *Scheduling for Mixed-criticality Hypervisor Systems in the Automotive Domain*. PhD thesis, University of York, Department of Computer Science, 2016.

[13] Robert Kaiser. Combining partitioning and virtualization for safety-critical systems. *Embedded World Conference*, January 2009.

[14] F. Kirschke-Billern and et al. AUTOSAR - A worldwide standard: Current developments, roll-out and outlook. In *15th International VDI Congress Electronic Systems for Vehicles*, Baden-Baden, Germany, 2011.

[15] Open Kernel Labs. OKL4 microvisor, August 2012. [Accessed: 01 Oct. 2012].

[16] E. Missimer, K. Missimer, and R. West. Mixed-criticality scheduling with I/O. In *Proc. 28th ECRTS*, pages 120–130, 2016.

[17] G. Morgan. Safety and security with hypervisor technology. In *Embedded World Conference*, Nuremberg, Germany, February 2016. Design & Elektronik.

[18] T. Nolte. Hierarchical scheduling of complex embedded real-time systems. In *Ecole d'Ete Temps-REel (ETR)*, 2009.

[19] G.J. Popek and R.P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[20] A. Pretschner, M. Broy, I.H. Kruger, and T. Stauner. Software engineering for automotive systems: A roadmap. In *2007 Future of Software Engineering*, FOSE, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[21] SYSGO AG. PikeOS RTOS and virtualization concept, August 2012. [Accessed: 16 Feb 2016].

[22] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS)*, pages 239–243, 2007.