IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 16, 2018, accepted August 8, 2018, date of publication September 10, 2018, date of current version October 8, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2869094

Hypervisor-Based Multicore Feedback Control of Mixed-Criticality Systems

ALFONS CRESPO[1], PATRICIA BALBASTRE[1], JOSÉ SIMÓ[1], JAVIER CORONEL[2], DANIEL GRACIA PÉREZ[3], AND PHILIPPE BONNOT[3]

[1]Institut d'Automàtica i Informàtica Industrial, Universitat Politècnica de València, 46022 Valencia, Spain
[2]Fent Innovative Software Solutions S.L., 46022 València, Spain
[3]Thales Research & Technology, 91120 Palaiseau, France

Corresponding author: Alfons Crespo (acrespo@ai2.upv.es)

This work was supported in part by the H2020 Euro-CPS, FP7-ICT DREAMS EU Project under Grant 610640, in part by the Spanish National R&D&I Project M2C2 under Grant TIN2014-56158-C4-01/02, in part by the Spanish Government and FEDER funds (AEI/FEDER, UE) under grant TIN2017-86520-C3-1-R (PRECON-I4), and in part by the Generalitat Valenciana PROMETEOII/2014/031.

ABSTRACT One of the most promising approaches to mixed-criticality systems is the use of multi-core execution platforms based on a hypervisor. Several successful EU Projects are based on this approach and have overcome some of the difficulties that this approach introduces. However, interference in COTS systems due to the use of shared resources in a computer is one of the unsolved problems. In this paper, we attempt to provide realistic solutions to this problem. This paper proposes a feedback control scheme implemented at hypervisor level and transparent to partitions (critical and non-critical). The control scheme defines two controller types. One type of controller is oriented towards limiting the use of shared resources by limiting bus accesses for non-critical cores. A second type measures the activity of a critical core and acts on non-critical cores when performance decreases. The hypervisor uses a performance monitor unit that provides event counters configured and handled by the hypervisor. This paper proposes two control strategies at hypervisor level that can guarantee the execution of critical partitions. Advantages and drawbacks of both strategies are discussed. Control theory requires to identify the process to be controlled. In consequence, the activities of the critical partitions must be identified in order to tune the controller. A methodology to deal with controller tuning is proposed. A set of experiments will show the impact of the controller parameters.

INDEX TERMS Cyber-physical systems, feedback control, hypervisor, mixed-criticality systems.

I. INTRODUCTION

The increasing computing capacity of multi-core embedded systems allows the integration of multiple software partitions on a single shared hardware platform, even if some of them are time-critical applications. In this paper, we consider a software partition as a bundle of application code and required runtime support. So, a partition can be an application compiled to be executed on a bare-machine, a real-time application with its real-time operating system or an application running on top of a general purpose operating system.

Under these circumstances, time-critical and non-critical applications should be integrated with the same hardware, given the increased computing power of current processors in terms of the number of cores and frequency. Systems that integrate time-critical and non-critical applications are known in the community as mixed-criticality systems (MCS) [1]. In [2], a review of the MCS status is presented analyzing the research trends and challenges.

Although the concept of Integrated Modular Avionics (IMA) [3] was developed for single-core platforms, the increasing importance of multi-core platforms provides additional relevant value to this approach. In this line, researchers and industry are making significant efforts to analyze and exploit the capabilities of these platforms taking into account their advantages and limitations. Greater performance increased computing power, and stricter cost-containment are countered by the difficulty of computing exact execution time due to the effects of shared resources on code execution.

In the IMA approach, several software applications can be executed in a common platform on top of a partitioning kernel under a common application programming interface (API) [4]. A partitioning kernel is an operating system with specific extensions to increase the temporal and spatial isolation of the applications. A partitioning kernel can be replaced by a virtualization layer (hypervisor) providing

the partitioning services and including an operating system as part of the application environment.

Hypervisor-based systems have demonstrated their ability to provide basic properties and mechanisms to execute several partitions, including their operating system, in a common hardware. These properties deal with the spatial and temporal isolation of software partitions (application and operating system) running on top of the hypervisor. Hypervisor mechanisms provide the services for visualizing hardware resources to partitions (e.g. virtual CPUs, virtual interrupts, etc.). The XtratuM hypervisor [5], [6] is an open-source bare-metal hypervisor for embedded real-time systems. It has been used in several EU projects and is currently being used on several space missions. Initially developed for LEON processors, it has been adapted to other platforms such as ARM Cortex R4/R5, A9, and PowerPC. In the FP7 EU MultiPARTES project [7], XtratuM was adapted for use in multi-core heterogeneous platforms. The FP7 EU DREAMS project [8] extended XtratuM to work on multiple multi-core nodes connected through NoC (Network on Chip) and TTEthernet networks.

One of the crucial expected features when using multi-core processors for safety-critical systems is temporal isolation. While the hypervisor can guarantee the exact allocation of resources to partitions, the execution of a partition can be affected by the interferences generated by the execution of other cores. This problem, due to the use of shared resources (cache, bus, memory, etc.), can affect the execution of a critical partition jeopardizing execution in the specified deadline. These interferences introduce an unpredictable factor in the execution of a critical task and do not permit estimation of worst-case execution time (WCET) in a multi-core system introducing timing anomalies [9], [10].

In this paper, we focus on the execution control of partitioned mixed-criticality systems running on top of a hypervisor. We propose a scheduling control co-design technique with two controllers implemented at hypervisor level.

This paper extends and completes a preliminary version of the controller scheme [11] where the basic controller was presented where sections V and VI are completely new while sections IV and VII are almost entirely new.

The main technical contributions of this paper include:

- A proposed controller scheme at hypervisor level, transparent to applications, to control the execution of critical partitions in a hypervisor-based partitioned multi-core system.
- The controller scheme includes two types of controllers for critical and non-critical partitions.
- Two control strategies and their configuration parameters.
- An analysis of the cost of these strategies and the overheads introduced.
- With a set of experiments, we show that with the controller scheme and proposed tuning, we achieve the expected goals.

The rest of the paper is organized as follows. Section II presents the performance counters used by the hypervisor to implement the control. Section III provides a basic understanding of the hypervisor scheduling model. In Section IV, we present the hypervisor controller goals and the mechanisms used. Section V presents two strategies for controlling the execution of critical activities and how the controllers are tuned. Section VI describes the new hypervisor functions and analyses the cost of the proposed strategies. Section VII presents the evaluation of the control schemes in different situations and evaluates the proposed control strategies using a multi-core processor board. Section VIII reviews the state-of-the-art of related techniques. Finally, Section IX presents our conclusions and suggestions for future work.

II. PERFORMANCE MONITOR COUNTERS

Current processors can provide measurements of how the system is performing. Measurements can be global or per core in the case of multi-core platforms. The Performance Monitor Unit (PMU) is a hardware device that provides this service. It supports execution profiling [12] and can be found, among others, on the T2080 multi-core processor on the NXP QorIQ T2080RDB board. The T2080 processor includes four 64-bit e6500 cores and multi-threaded implementations of the resources for embedded processors defined by Power ISA. The e6500 core includes a PMU that provides a set of performance monitor counters (PMCs) per core for defining, enabling, and counting conditions that can trigger the performance monitor interrupt. Each core can configure up to six 32-bit counters that can count specific events.

In x86's architectures, the PMU was introduced with the Intel Pentium processor and is nowadays available in all modern processors [13]. It provides per core a set of counters for fixed events that cannot be changed and another set of programmable counters can be set to count one of the supported hardware events. A set of instructions allows to programming or reading these registers. The number of counters and the programmable events can vary depending on the specific x86 processor model.

In ARM processors, the PMU is an optional but recommended feature for A and R implementations [14]. It provides two specifications to this unit but the basic form provides one cycle counter and up to 31 programmable event counters per core. Instructions allow controlling by enabling and resetting counters and enabling interrupts on counter overflow.

For this work, all the architectures described above offer the required services. These requirements can be detailed in terms of having a set of independent counters in each core, selecting the events to be monitored, defining limits and enabling interrupts when limits are reached. The selected events in this work are available in all these processors.

The selected platform in this work is the T2080 because the scope of the study has been focused on avionics applications and it is relevant in this context. Additionally, the T2080 provides hardware extensions for virtualization and defines a *hypervisor processor* mode for executing the hypervisor.

The XtratuM hypervisor has been ported to this platform and provides, using the virtualization services, full virtualization to partitions. XtratuM handles the PMCs to count specific events during the system execution. Using the PMU, XtratuM provides the ability to count predefined events per core associated with particular operations such as processor cycles, executed instructions, L1 and L2 cache misses, data and instructions bus accesses, in order to measure the efficiency of the application running on a core. Also, performance events can be restricted to the guest or hypervisor domain.

A threshold value can be defined for any counter to trigger interrupts when a specified value is reached. Counters can be enabled or disabled under hypervisor or application needs.

III. HYPERVISOR SCHEDULING MODEL

XtratuM is a bare-metal hypervisor specifically designed for embedded real-time systems that uses para-virtualization or full-virtualization techniques depending on the hardware support. The XtratuM hypervisor enforces the logical division of software components into independent execution environments (partitions) so faults are isolated and contained. These software partitions are spatially isolated in addition to the temporal allocation of processor resources to partitions. XtratuM supports the concept of virtual CPU (vCPU). Virtual CPUs are abstractions that model hardware CPU behavior and are managed in an analogous way but can be allocated to any of the existing cores. XtratuM abstracts as many virtual CPUs on the system as physical cores. Partitions can be mono-core or multi-core when using one or several vCPUs. The allocation of vCPUS to real CPUs is decided in the configuration file where the global system is fully specified: hardware, devices, scheduling plan, communication channels, etc. Multi-core application will require an SMP guest operating system and allocate several virtual CPUs to the partition.

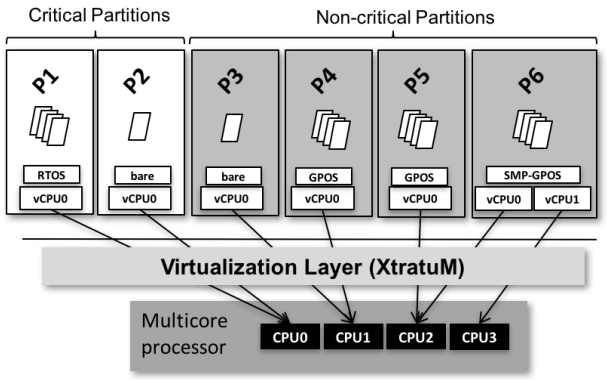


FIGURE 1. Partitioned architecture in multi-core platforms.

The software architecture in a multi-core partitioned system is presented in Fig. 1. It shows a system integrated by five mono-core partitions and one multi-core partition. In the scheduling plan, the allocation of vCPUs to real CPUs is defined and shown in the figure. P0 and P1 are allocated to the real CPU0, P3 and P4 to CPU1, P5 to CPU2 and the two vCPUs of P6 to CPU2 and CPU3, respectively. All threads in a partition are executed in the associated CPU. The SMP partition will schedule internally which threads are associated to each vCPU and, consequently, in which real CPU they will be executed.

ARINC-653 [4] defines a cyclic schedule for partitions. In a multi-core environment, the cyclic schedule can be extended to all cores by allocating partitions to cores and defining a cyclic schedule per core. XtratuM implements a multi-plan cyclic scheduling for multi-core systems.

Generation of the cyclic schedule should consider the impact on task execution due to shared resources such as bus access, L2 caches and memory. In Xoncrete tool [15], most of these issues are modeled. In [16], a methodology to generate cyclic schedule plans for multi-core mixed-criticality systems is proposed. From a set of critical and non-critical partitions, it performs a partition to core allocation and generates a cyclic schedule for each core.

A. SYSTEM MODEL

The system model described in terms of a set of partitions (P) that are composed by the dupla $P_i = (\tau, L)$, where L is the level of criticality and τ is the set of tasks of the partition.

Two levels of criticality are considered: High and Low. A partition with a higher level of criticality is defined by $\tau_i = (C_i, Pd_i, Dl_i)$ where C_i is the worst-case execution time, Pd_i is the period and Dl_i is the deadline.

Partitions with low criticality are modeled by only one task or server that integrates all internal activities. It can be seen as a server with a bandwidth defined by $\lambda = (Bs_i, Ps_i)$ where Bs_i is the budget or maximum execution time per period and Ps_i is the period. Additionally, other resources such as communication channels between partitions should be considered and avoided in this definition for the sake of clarity.

B. SCHEDULING GENERATION

Cyclic schedule generation consists of three steps:

- Step 1 Core distribution: determination of the number of physical cores for high and low criticality level partitions.
- Step 2 Allocation of partitions to cores.
- Step 3 For each core perform the schedule generation based on a limited preemption EDF with Stack Protocol Reservation (SPR) as detailed in [15].

In step 1, the number of cores is determined by computing the utilization of each partition. The algorithm computes the total utilization of the high and low critical partitions as the sum of all task utilization (C_i/Pd_i) or the (Bs_i/Ps_i) in the case of low criticality partitions. The result of these computations is U^H and U^L as global utilization of high and low criticality load which determines the number of cores N^H and N^L for high and low criticality, respectively.

The goal of step 2 is to perform a static allocation of partitions to cores. High and low-level partitions will be allocated in N^H and N^L cores, respectively. In order to perform this

allocation, bin packing techniques are used. Worst fit policies for bin packing partition to cores produces more uniform load distribution [16]. A final optimization of the allocation is performed using greedy techniques based on the discrepancy of the core loads.

The plan generated achieves:

- All critical partitions are allocated to a subset of cores. We use the term Critical Core (CC) to identify cores with critical partitions.
- Non-critical tasks are allocated to another subset of cores. These cores are considered to be Non-Critical Cores (NCC).
- Each task in a partition has its own temporal window or slot in a core.
- Slot duration of partitions takes into account the measured worst-case execution time increased by a factor (design criteria) that models the interference.

In this paper, we assume that one core can allocate all critical partitions while the non-critical partitions can use several cores. However, the proposed controller scheme is also compatible with the execution of non-critical partition on CC. Moreover, the constraint of only one critical core will be relaxed in section IV-D.

IV. GENERAL CONTROLLER SCHEME

The main goal of the controller scheme is to limit the interference of Non-Critical Partitions (NCP) on shared resources that can impact on the execution of Critical Partitions (CP).

While CP should not be limited because a worst case analysis has been performed and a static schedule fitting the temporal constraints has been generated, the execution of NCPs should be controlled to limit the interference on CP.

The control scheme depicted in Figure 2 is proposed.

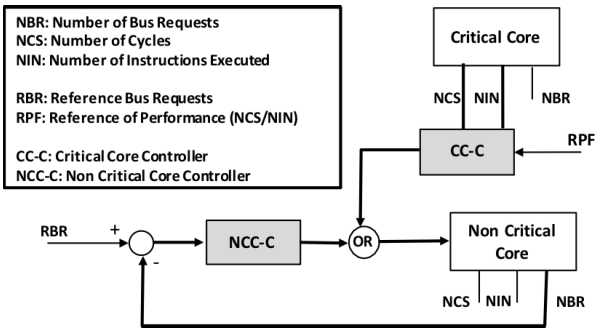


FIGURE 2. General controller scheme.

Initially, two cores are considered (we will remove this restriction in following sections), CC and NCC. Two controllers have been designed. The Non-Critical Core Controller (NCC-C) reads the performance monitor bus requests of the NCC and compares it with a specified reference of a number of bus requests. If the reference value is reached the controller can take actions on the NCC. The Critical Core Controller (CC-C) reads the performance monitor cycles and instructions from the CC, computes the cycles per instruction (CPI), compares the result with the reference for this relation and takes a decision on the NCCs. The controllers are characterised as follows:

- CC-C: an event-based controller.
  - Goal: To guarantee the execution of a critical task (partition) within the slot allocated to its service.
  - Event: Interrupt generated from the PMU based on number of instructions (NIN).
  - Inputs: Number of cycles (NCS).
  - Reference: Specified maximum relation between Cycles and Instructions. This corresponds to the cycles per instruction (CPI).
  - Action: Suspend the NCC during the remaining partition execution.
- NCC-C: an event-based controller.
  - Goal: To limit access to shared resources during execution.
  - Event: Interrupt generated from the PMU based on Number of bus requests (NBR).
  - Inputs: Number of bus access requests.
  - Reference: Specified maximum number of bus requests from the non-critical partition.
  - Action: Suspend the NCC.

As these controllers have to be implemented at hypervisor level, it is relevant to understand the impact of the controller on the partition execution. Ideally, an event-based controller enables controller activity to be limited only when a significant event occurs. This is the case of the NCC-C that will only act once during partition control in an NCC. As soon as an event arrives the core suspends the partition execution. However, CC-C requires periodic sampling of the NIN or NCS because of a lack of performance monitor for the appropriate counter. During the experimentation, we found that cycles per instruction (CPI) can be used but they must be calculated by the controller to take any appropriate action.

A. CONTROLLER ACTIONS

Suspension of NCC core activity is proposed as the controller's action. This has the advantage that once the action is taken, it is definitive until the end of the execution slot. In short, only one action is taken. Initially, two cores have been considered. Subsequently, several NCC will be considered. In this latter, the action of suspension could be partial (some cores) or total (all NCC). The proposal is that the suspension action affects all NCC. Partial suspension has the disadvantage that the controller's decision must be periodically re-evaluated and future partial or total actions taken if needed. This can lead to hypervisor overhead that can affect the performance of the critical task. Since the controller's objective is to minimize the overload and impact as much as possible on the computation time of the corresponding task, the proposed action is: i) definitive: this means that there is no re-evaluation of future actions, and ii) total: the action directly affects all other cores.

An alternative to suspension may be to decrease the frequency/voltage of the other cores and, consequently, decrease their activity on the common resources. With this approach, the action needs to be reevaluated periodically. In addition, the intensity of the action in terms of the number of frequency/voltage levels must be determined. It should also be noted that not all processors can manage core frequency independently. A change in processor frequency involves all processors so that the solution would not be valid for the proposed goal.

B. CONTROLLER SCOPE

Let us assume that we have a system with four partitions and two cores. P0, P1 are critical and P2, P3 are non-critical. All partitions can have internal tasks but only tasks in critical partitions have been considered according to the task model defined in Section III. For simplicity, only one task is considered in the partition slot. Figure 3 shows a scenario that will illustrate the controller's scope.

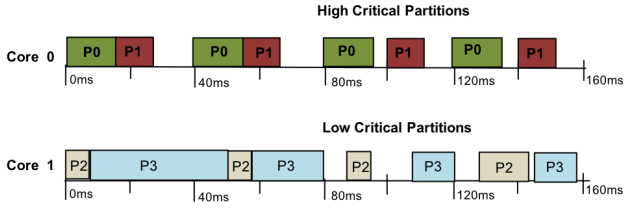


FIGURE 3. Two cores schedule.

The generated schedule has allocated some slots to each partition taking into account the worst-case execution time of the task(s) inside the partition plus additional extra time to model the interference. Thus, in isolation, the slot duration is over-estimated.

The controllers will be implemented at the hypervisor level. In order to specify their scope, a set of rules are formulated:

- Rule 1: When there is no activity in the CC, the NCC-C is disabled.
- Rule 2: When a critical partition starts the slot, both controllers (CC-C and NCC-C) are enabled. If there is no activity in the NCC, the enabled state remains and will start to act when some activity in the core is scheduled during execution of the critical partition.
- Rule 3: When the CC-C takes action to suspend NCC activity, the current partition and future partition slots are suspended.
- Rule 4: When the NCC-C takes action to suspend its activity, the current partition is suspended. Future slots will start at the specified time in the scheduling plan.
- Rule 5: When a critical partition finishes its slot, controllers are disabled. If NCC activity was suspended during its execution, NCC is resumed.
- Rule 6: Communication between cores is performed through inter-processor interrupts (IPIs).
- Rule 7: Decisions and actions taken by controllers must be performed in bounded time and low overheads.
- Rule 8: Controllers should be event based and the number of interrupts predictable.

In conclusion to these rules, the scope of the CC-C is the slot duration in critical cores of critical partitions. Note that if the scheduling plan includes non-critical partitions in the critical core, the CC-C will not act during the slots of these partitions. However, the NCC-C scope is inherited from CC-C. In other words, the NCC-C will act in the same CC-C scope.

Figure 4 shows a possible scenario where the scope of the controllers is defined. A simulated evolution of the performance parameters during controller scope is drawn. The grey areas covering the two cores determine the <scope of the controllers.

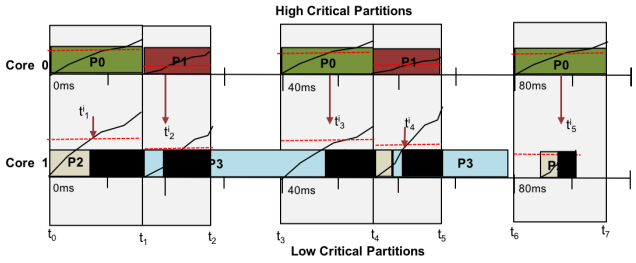


FIGURE 4. Execution scenario.

During initialization, the hypervisor detects the number of cores and creates the same number of hypervisor threads. In this case, two hypervisor threads (HT0 and HT1) are created and allocated to cores C0 and C1 respectively. At t_0, HT0 detects the slot start of P0 and identifies (through the configuration file) that it is critical. It enables both controllers and sets a threshold on the number of instructions for P0 and enables the interrupt of the PMU. Also at t_0, HT1 starts the execution of P2 and receives an IPI from HT0 and enables the controller and sets the threshold of the bus requests counter to a specified reference. In the interval $[t_0, t_1^i]$, HT0 can receive several interrupts from the PMU when the number of instructions has been completed and computes the slope of the execution (cycles/instruction). If the computed value is in a specified range, no action is performed.

At t_1^i, as consequence of the interrupt received by HT1 from the PMU informing that the maximum number of bus accesses has been reached, HT1 suspends partition P2, stores its status and sets the core to suspended.

At t_1, P0 slot finishes and HT0 executes the partition context switch. HT0 stores the status of P0, sends an IPI to HT1 to resume the core activity and identifies the next partition to be executed P1. As P1 is critical, HT0 enables the controller, sets the threshold of the performance monitor register and executes P1. HT1 has received the IPI to resume core activation, analyses the scheduling plan to know which

partitions have to be executed (P3), recover their status, sets the core to active and writes the NBR threshold.

At t_2^i, as consequence of the interrupt and computed value (out of the specified range), the action taken by CC-C is to suspend the execution of NCC (Core 1). HT0 sends an IPI to HT1 to perform the action. HT1 suspends the current partition (P3), stores its status, disables the controller and sets the status of the core to suspended.

At t_2, HT0 finishes the P1 slot, stores its status and sends an IPI to HT1 to resume the core. HT0 reads the next slot to be executed and detects that it will occur at t_3. It sets the timer to be awake at that time and waits for it. HT1, as a consequence of the IPI, resumes core activity by identifying the partition to be executed (P3) in the scheduling plan and executes it.

At t_3, HT0 starts the P0 slot and performs the actions when the slot of the critical partition slot starts (detailed at t_0). At t_3^i, HT0 receives the interrupt and performs the actions to suspend the NCC.

At t_4, HT0 stops the P0 slot and performs the actions when a slot of a critical partition is finished. HT0 selects P1 slot in the scheduling plan and perfoms the corresponding actions. HT1 receives the IPI and executes P2. At t_4^i, HT1 receives the interrupt and performs the actions to suspend itself.

C. MULTIPLE NON-CRITICAL CORES

The previous scenario can be extended to multiple NCCs. Figure 5 presents the controller extension to deal with multiple NCCs.

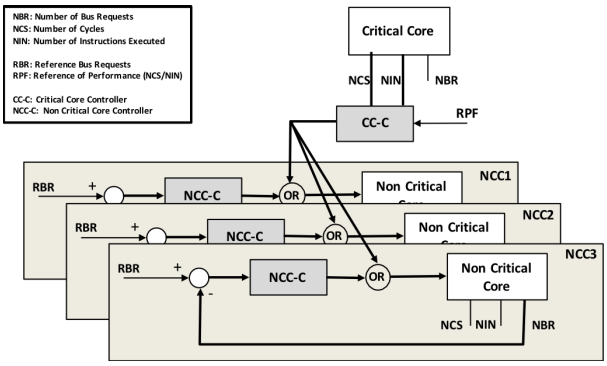


FIGURE 5. General controller scheme for multiple non critical cores.

Each NCC controller is an event-based controller. The event is generated when the number of bus requests reaches the reference value. The action of the NCC-C is self-suspending. The core executing a critical slot (critical partition) executes a CC controller. Actions taken by the CC-C are applied to all NCCs. So, when CC-C takes the decision to suspend NCCs activities, it sends IPIs to all of NCC in the same way as the two cores description. IPSs can be individually sent to each core or broadcasted. Suspending the activity of a core means that each suspended core does not execute partition code and remains in an idle state. In this state, it waits for an IPI at the end of the critical slot to resume the core activity.

D. AVOIDING CONSTRAINTS

So far we have assumed that one core is critical (it executes all critical partitions) and others are non-critical. However, that assumption is not strictly required. The main constraint is to avoid overlap in temporal slots of critical partitions. In that case, critical and non-critical partitions can be allocated in any core according to load requirements.

The generation of scheduling has to build a static schedule where critical partition slots do not overlap with other slots of other critical partitions allocated to other cores. In such cases, a core is critical when it is executing critical partitions where the remaining cores are non-critical.

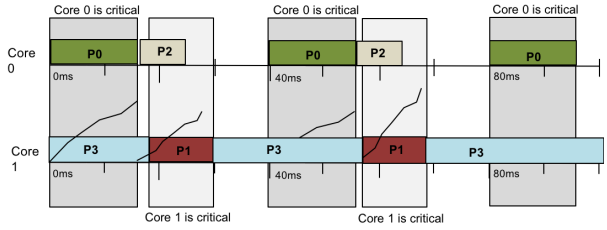


FIGURE 6. Scope extension.

Figure 6 shows a schedule where P0 and P1 are critical partitions and P2 and P3 are non-critical ones. P0 is allocated to core 0 while P1 is allocated to core 1.

Shadowed regions represent the time intervals in which P0 or P1 are under execution and, consequently, when cores 0 and 1 are considered critical cores.

V. CONTROLLER STRATEGIES AND TUNING

In this section we attempt to provide guidelines for controller strategies and tuning.

A. NCC CONTROLLER (NCC-C)

Our proposal for these controllers is based on how a non-critical partition uses the bus. The diversity of the non-critical applications is very high. If it can be measured in isolation, a rule that has been applied in the previous example is to limit the execution of each non-critical core when it reaches a number of bus requests in isolation. As controllers are only active when critical cores are executing critical applications, the reference of an NCC-C is set to the proportional number of bus requests restricted to the overlap interval with the critical partition. Thus during the overlap interval of a non-critical partition with a critical one the number of requests non-critical partition will use in isolation during the overlap interval is calculated.

A more in-depth analysis of non-critical partition needs could improve control performance.

B. CC CONTROLLER (CC-C)

While NCC-Cs are oriented towards limiting their use of the bus, the CC-C focus should be on the deadline guarantee for the critical applications by applying the controller to their slots.

Two strategies for the CC controller have been analyzed: Limit and Linear controller.

1) LIMIT CONTROLLER STRATEGY

Figure 7 draws the slot duration allocated in the plan to the critical partition in the critical core. The grey area sketches the secure area in which partition execution can finish before the end of the slot. If at any moment the execution of the critical partition reaches the right-hand side of the secure area, there is a strong risk of reaching the end of the slot without finishing its computation.

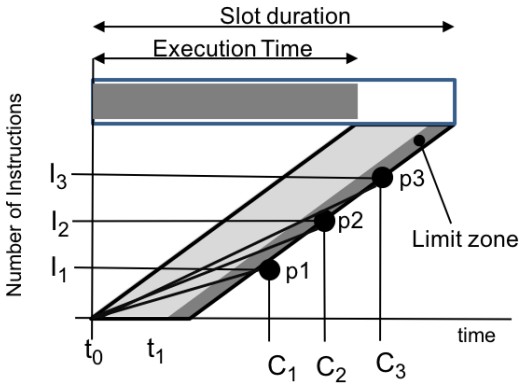


FIGURE 7. Limit controller strategy.

If the partition execution reaches that secure limit, an action suspending the other cores will enable computation to finish before the end of the slot duration.

The behavior of the CC-C, as described in Section IV, is an event-based controller and should not be periodically executed due to the overheads. In order to control execution, the hypervisor will set the threshold of the PMU register to I_1. When the event is generated, the hypervisor will read the cycles register and determine the CPI. If the value is greater than the slope of point P_1 (in the drawing), the controller suspends all cores. If execution is in the secure zone, the hypervisor sets the next threshold to I_2. These points are the limit of the secure zone. Depending on the number of I_n points, the detection of a limit situation will be too late to reach the desired deadline. In these cases, a secure margin can be defined as drawn in Figure 7 by means of parallels to the limit line.

2) LINEAR CONTROLLER STRATEGY

The action of CC-C can be considered a strong action: suspend NCCs activities for the remaining CPart execution. Another less aggressive strategy could be considered. Figure 8 shows an alternative criterion for CC-C design. It considers that the relation between the total cycles of the slot duration and the number of instructions to be executed in the worst case defines a limit line between the secure and non-secure zones. While the CPI is in the secure zone, NCCs are active. As soon as the non-secure zone is reached, the NCCs

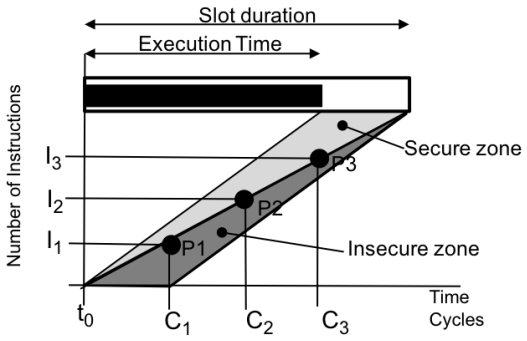


FIGURE 8. Linear controller strategy.

are suspended. As a result of this action (suspend NCC cores), the CPI can return to the secure area and the action will resume NCCs activities. These actions can generate, even using a limit line threshold, several sequential suspend and resume actions.

The advantage of this strategy is that the CPI of the experiments does not need to be determined. Directly, the slot duration in cycles permits determination of the CPI. However, this controller can produce several suspend/resume actions significantly increasing hypervisor overhead.

C. CC-C PARAMETERS

In order to control critical partitions, a process identification of the tasks included in each critical partition must be performed. As pointed out in Section IV, task execution has to be scheduled in a slot. Slot duration determines the deadline for executing the corresponding task.

The first step in the tuning is to perform a process identification based on how the task performs in isolation. All tasks in the critical partitions must be characterised. A task is identified by measuring the selected registers during execution time. In order to limit the effects of the performance monitor registers access and store, only one performance monitor counters read is performed in each slot. The mechanism used to collect the measurements during the execution is to set the event PME_PROCESSOR_CYCLES with a threshold generating an interrupt that is handled by the hypervisor logging register values. As only one interrupt per slot is allowed until the interrupt handling occurs, there is no interference due to monitoring the counters. Considering the processor frequency (1800Mhz), the increment of each slot threshold is defined to sample the counters every 10 ms in the experiments. This value can be adjusted depending on task execution time.

The log generated by the hypervisor is analyzed off-line in order to extract two parameters: the maximum number of instructions executed by the task and the cycles per instruction (CPI). From this logged information, a linear minimum mean square error is calculated. Based on the linear approximation obtained, the CPI value is determined.

The hypervisor must be booted with a configuration file. The configuration file is an XML file that specifies all the

details about the hardware platform (number of cores, frequency, memory areas, devices, etc.) as well as partitions, communication channels and the scheduling plan. It allows several scheduling plans to be specified covering systems with multiple modes. Each scheduling plan specifies, per core, the sequence of slots in a major frame (MAF). All cores share the same MAF. Details about the specification of the configuration file for the XtratuM hypervisor can be found in [17]. An execution slot is described by a set of parameters such as the slotIdentifier, the partition to be scheduled, the starting time as an offset with respect to MAF origin and slot duration.

Three optional additional parameters have been added to inform the hypervisor about the controllers. When the slot corresponds to a critical task, two parameters are included: number of instructions and the CPI. If the slot is associated with a non-critical partition, the parameter to be included is the maximum number of bus accesses allowed during the slot duration. If this parameter is not provided for non-critical partitions, the assumption is that no limitation is imposed. This will affect the NCC-C and the hypervisor will not activate the controller in this slot.

VI. HYPERVISOR PERFORMANCE ANALYSIS

The XtratuM hypervisor has been enriched with the mechanisms to incorporate the proposed controllers. XtratuM implements as many kernel threads as real cores. Each kernel thread is executed in its core and controls partition execution according to the static plan defined for each core. All kernel threads share a set of protected data structures. One of them refers to the status of the core, which includes an additional field that informs when a core has been suspended or resumed by the CC-C.

The suspension of a core implies that the kernel thread that manages it preempts the running partition and goes to the idle state. In this state, the kernel thread disables partition interrupts and runs an empty loop until it receives the interrupt (IPI) from the critical core at the end of the critical partition slot. When it arrives, the kernel thread determines the partition to be executed according to the execution plan. Figure 9 shows an example of how the action of suspending different cores might affect to NCC cores according to the execution plan. When the critical partition execution slot (*Core*0) starts, the NCC-C of each core becomes active and will remain in this state until the critical partition slot ends. *Core*1 was already running a partition. As soon as the status becomes active, the thread will set the reference value in the limit value register for the number of bus requests, taking into account the overlap between its slot and *Core*0, and enable the PMU interrupt. *Core*2 thread is not running any partition and will wait until its next partition (point a) is planned to set the reference of the bus requests. In *Core*3, the slot start coincides with the *Core*0 slot start. So, it will act as described for the others NCC-C. At the end of its slot (point b), the PMU interrupt is canceled. In point c, CC-C decides to suspend the NCCs. *Core*1 and *Core*2 preempt the

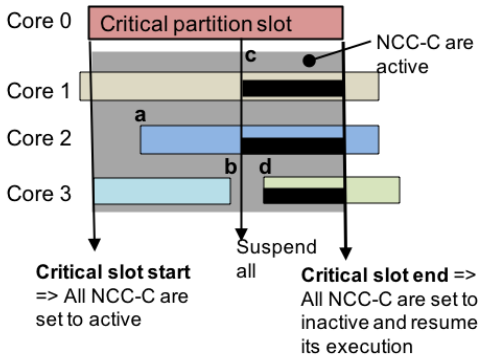


FIGURE 9. Suspension of multiple cores.

running partition. On the other hand, the core3 is not running any partition and will not run it (point d) even if it is scheduled in its plan. At the end of the critical slot, each core recovers its execution.

A. KERNEL THREAD COMMUNICATIONS

Communication between kernel threads is performed by means of IPIs. A kernel thread can send an inter-processor interrupt to another core which is attended by the associated kernel thread. IPIs to several cores can be broadcasted avoiding the need for loops in the code to send as many IPIs as NCCs. So, CC-C will broadcast only one IPI to inform the NCCs. Modifications in the hypervisor affect the partition context switch (PCS) between partitions. When the end of a slot is reached in the CC scheduling plan, the associated kernel thread performs the PCS as described in the Listing 1 pseudocode.

```
1   function CPS_CC() is
2     if CCC is active then
3         Broadcast IPI to NCC(resume_core)
4     Store_Partition_State
5     Select_Plan_Next_Partition
6     Restore_Next_Partition_State
7     Setup_Resources_Next_Partition
8     if Slot is Critical then
9         Setup_CC_Controller
10    Setup_Timer_Slot_Duration
11    Flush_Cache
12    Return_To_Next_Partition
13  end CPS_CC
```

LISTING 1. Partition context switch.

The function to setup the critical core and the interrupt handler to deal with the PMU interrupt are described below, and their pseudo-code presented in Listing 2. Note that variables MARGIN and CCC_LEVELS are defined during hypervisor configuration before its compilation. The hypervisor binary holds the configured parameters.

NCC kernel threads have to deal with the interrupts from CC-C to activate the NCC-C and suspend and resume the partition activities. It is important to point out that the kernel thread of the NCCs does not suspend kernel activities during core suspension. Thus the kernel thread remains under

```
1  function Setup_CC_Controller(nInst, CPI) is
2    nxtInstThrhld = nInst / CCC_LEVELS
3    Set_NCCC_Active()
4    Set_Threshold_PMU(nxtInstThrhld)
5    Send IPI to NCC(startCC)
6  end Setup_CC_Controller
7
8  function IRQ_Handler_PMU() is
9    Reads_PMU_Registers
10   nxtInstThrhld := Increase_Level
11   Computes_Current_CPI
12   if Current_CPI >= Limit_CPI(MARGIN) then
13        Broadcast IPI to NCC(suspend_core)
14   else
15        Set_Next_Threshold_PMU(nxtInstThrhld)
16 end IRQ_Handler_PMU
```

LISTING 2. Setup CC controller.

execution but suspend and resume only affects the partition execution defined in their scheduling plan.

The PCS of the NCC kernel threads works in the same way as the CC except that the threshold is set to the bus access requests counter rather than the instruction counter. Interrupt management includes three interrupts: from the PMU (bus access requests) and from the CC to suspend and resume partition activities.

B. OVERHEAD ANALYSIS

The analysis focuses on the CC kernel thread because it executes the critical activities in the system. It has to consider two costs: controller setup and controller execution. Controller setup is included in the PCS and introduces very few instructions. The cost in terms of cycles has been measured with an increase of 268 cycles with respect to the 39653 cycles (approximately 22 μsec at 1800MHZ) of the original PCS.

The cost of execution impacts on critical activity execution. The worst case corresponds to exhausting all levels of the number of interrupts. Currently, this number of interrupt levels is configurable at hypervisor compilation time. When an interrupt occurs, the CC kernel threads execute the previously defined interrupt handler. The measured cost in terms of cycles is 64 when no IPIs are sent to NCC and 182 when they are sent. As the number of interrupts that can be generated is defined at compilation time, the overload due to this mechanism is bounded to the number of levels. The worst scenario corresponds to the case where the last level performs the suspending actions in the rest of the cores. In that case, all levels except the last one have been handled and the last one performs the actions.

On the basis of the costs of the operations, it is possible to evaluate the performance of the two proposed schemes. Let L be the number of instruction levels configured for the CC-C. The control based on the limit strategy will execute the action of suspending all the NCC cores only once. The worst-case scenario will be to run it after exhausting the L-1 interrupts. The cost for this situation is $(L-1)*64+182$ cycles. The best case, from the point of view of critical partition, would take the action when the first interrupt occurs. In this case, the best case would be 182 cycles. In the case of the linear controller, the worst case is that every time an interruption occurs, an action of suspending or resuming is performed alternately. In this case, the cost is $L*182$ cycles. The best case would be that no action is taken on any of the L interrupts ($L*64$ cycles). Assuming an L = 15, the worst cases in the two schemes would be 0.5 and 1.5 $\mu secs$, respectively.

This analysis corresponds to the impact on the critical partition. However, a higher impact can be produced on the non-critical partitions. In the limit controller, only one suspension can be produced. It implies that the partition context is saved and the kernel thread goes to idle. The cost is approximately less than half PCS (approximately 8 μsec). In the Linear controller, the partition in the NCC can be suspended and resumed several times. A suspension and resume actions have a similar cost of one PCS. So, it means that the non-critical partition can suffer L partition context switches during the critical partition execution. It can imply a reduction of performance of non-critical partitions. A more thorough analysis of the proposed strategies and new ones are considered as future work,

VII. EXPERIMENTAL RESULTS

This section describes the scenarios evaluated considering the proposed controllers. Controllers use PMU counters. After comprehensive analysis of the more than 60 events available in the PMU of the platform used, we selected three events for controller implementation. These events are: PME_PROCESSOR_CYCLES that counts the number of cycles, PME_INSTR_COMPLETED that counts the number of instructions executed and BIU_MASTER_REQUESTS that provides the number bus accesses requested. The number of bus requests is strongly dependent on the bus collision generated. This measure is significantly higher when the core is executed at the same time as other cores than when it is executed in isolation. All events can be applied to the guest, hypervisor or both domains. Depending on the domain, they count the selected domain(s) processor cycles, instructions executed, or bus requests.

The experimentation has analysed the behaviour of the proposal using a Flight Control demonstrator provided by Thales comprising two partitions and several tasks. We selected specific tasks for the experimentation. Although there are tasks with a computation time of milliseconds, we selected one task with a computation time of two hundred of milliseconds to show better the effects of the interference of other cores. Also, the selected task was more representative of control tasks where different sections in the execution correspond to data acquisition (input) or data delivery (output). Input and output phases have more intensive access to memory while computation performs fewer external accesses (data can be cached).

Each scenario was executed on the T2080 platform with a critical partition (CPart) and a range of 0 to 3 NCC executing dummy applications. These dummy applications or partitions are executed in temporal windows that can overlap with *CPart* partition in an interval. CPart starts in all scenarios at

time 0. The goal is to measure the response time of this partition. Four scenarios have been defined. In *SC*1 controllers are not active. *SC*2 uses NCC-Cs while *SC*3 defines a CC-C. In *SC*4, both types of controllers are active.

A. SCENARIOS EVALUATION

In all scenarios, *CPart* is executed in core 0 which is considered the critical core while Dummy partitions are executed in cores 1, 2 and 3 (non-critical cores). We use *DP* to refer to all Dummy Partitions and DP_n when a specific Dummy Partition is referenced.

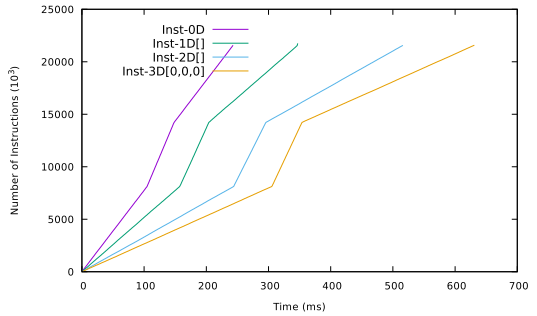


FIGURE 10. Execution evolution when controllers are not active.

Figure 10 shows the execution of Scenario SC1. Time measurements are performed in processor cycles but we present it as time (processor frequency is 1800MHz) to facilitate plot representation. The x-axis represents the time in milliseconds while the y-axis draws the number of *CPart* instructions. As can be seen, if *CPart* is executed without the interference of other partitions (identified in the plot as Inst-0D), it finishes its computation at 243 milliseconds (ms). When *CPart* is executed with DP_1 starting at 0 ms (Inst-1D in the plot), the interference introduces a delay in the *CPart* execution which requires 346 ms to complete its number of instructions (computation). Inst-2D shows the evolution of *CPart* instruction count when 2 *DP* are started at the same time as CPart. In this case, *CPart* finishes its computation at 517 ms. Inst-3D plots the evolution of *CPart* when 3 DP are executed starting at time 0 ms. *CPart* completes its execution in 634 ms.

TABLE 1. NCC-C active. CPart response and action time.

No.	Init time	No active R. time	NCC-C active R. time	NCC-C active Act. time
1	0	346	286	130[1]
2	0,80	447	327	154[1],235[2]
3	0,80,120	503	387	179[1],297[2],325[3]

Time units in milliseconds

Tables 1 and 2 present the results when NCC controllers and CC controllers are independently active. The scenario includes three cases with 1, 2 or 3 *DP*s starting at 0, 80 and 120 ms. The third column provides the response time of *CPart* when no controllers are active. In the case that *CPart* is executed in isolation, the response time is 243 ms. Columns 4 and 5 present the results when only NCC Controllers are

TABLE 2. CC-C active. CPart response and action time.

No.	Init time	No active R. time	CC-C active R. time	CC-C active Act. time
1	0	346	290	140 [1]
2	0,80	447	324	174 [1,2]
3	0,80,120	503	337	187 [1,2,3]

Time units in milliseconds

active. In that case, the response time of *CPart* is 286, 327 and 387 ms, as result of the actions taken by NCC Controllers shown in column 5. For instance, when 2 *DP*s are defined, actions are taken at 154 ms by NCC-C in core 1 and 235 ms by NCC-C in core 2.

Columns 4 and 5 in table 2 show the results when only a CC Controller is active. In that case, *CPart* finishes at 290, 324 and 337 ms, respectively. Actions have been taken at 140, 174 and 187 ms suspending running non-critical cores. Column 5 shows the time the action is taken and the affected cores (in brackets).

TABLE 3. CC-C and NCC-C active. CPart response and action time.

No.	Init time	NCC-C and CC-C active R. time	NCC Act. time	CC Act. time
1	0	286	130[1]	
2	0,80	317	154[1]	167[2]
3	0,80,120	336	179[1]	186 [2,3]

Time units in milliseconds

Table 3 show the results when CC and NCC controllers are active. Using the same scenarios, columns 3, 4 and 5, show the response time of *CPart*, the action taken by NCC and CC Controllers. In the case of 2 *DP*s starting at 0 and 80 ms, *CPart* finishes its execution at 317 ms. At 154 ms, the NCC-C of core 1 suspends its execution. At 167 ms, the CC-C decides the suspension of the rest of the cores (core 2 in this case).

A more detailed analysis and justification for the instant the action is taken by CC-C is presented in the next section.

The conclusion of this experimentation is that: i) it is possible to control the execution of critical cores, ii) a characterization of *CPart* and *DP* is required to adjust the references. The reference for NCC-Cs should limit the number of bus accesses. So, an estimation of the number of accesses in isolation restricted to the overlap interval with critical partition is a good starting point for setting the reference that limits the NCC bandwidth.

The CC-C tuning has to consider the number of control points (increment of the number of instructions) and the relation between cycles and instructions (CPI). These values should take into account the way *CPart* executes the instructions.

B. STRATEGY ANALYSIS

Using the previous example of the critical partition, the worst observed execution time of a task is 243 ms. As it has to be executed in parallel with other cores, at design phase an

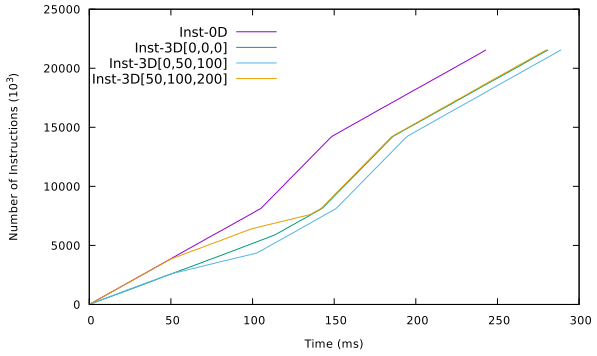


FIGURE 11. Local and Global controller. 15 instruction levels.

interference overload should be assumed. Let us suppose that the system integrator of the partitioned system decides that execution time plus an additional 20% of its time is allocated to the task. In that case, when the cyclic scheduling plan is generated, the slot to allocate the task will be 290 ms.

TABLE 4. CC-C and NCC-C active. CPart response and action time.

	All active CC [5 levels]		
Init time	R. time	NCC-C Actions	CC-C Action
0,0,0	286	130[1]	137[2,3]
0,50,100	333	169[1]	183[2,3]
50,100,200	382	232[1],307[2],375[3]	

Time units in milliseconds

TABLE 5. CC-C and NCC-C active. CPart response and action time.

	All active CC [15 levels]		
Init time	R. time (usec)	NCC-C Actions	CC-C Action
0,0,0	286	130[1]	137[2,3]
0,50,100	309		136[1,2,3]
50,100,200	290		158[1,2,3]

Time units in milliseconds

Tables 4 and 5 show the behavior of two controllers with different configurations in the same set of scenarios. The scenarios are composed of three DSs with starting times at [0, 0, 0], [0, 50, 100] and [50, 100, 200], ms respectively. Columns 3, 4 and 5 show the response time of *CPart*, the time where the action suspend itself is taken by each NCC-C and the time where the action applied by the CC-C, suspending the running cores, is executed. This CC Controller is configured with 5 instruction levels. Same columns (4, 5 and 6) of table 5 show the control results when the CC Controller is configured with 15 instruction levels. Figure 11 plots the evolution of *CPart* execution in the case of the CC Controller with 15 levels.

C. ALTERNATIVE STRATEGY

Table 6 summarises three cases with the start times, the instants at which control decisions to suspend and resume NCCs are taken and NCC controller decisions (time and core). In second case, slots of NCC start at 0, 50, and 100 ms. A 10 ms, the CC-C decides to suspend all NCCs which are resumed at 24 ms. This situation is repeated several

TABLE 6. Time results when Linear controller is used.

Init time	R. time	CC-C Suspend CC-C Resume	NCC-C Suspend
0,0,0	274	10,35,60,77, 95,119 25,49,67,84,109,126	195 [0]
0,50,100	333	10,41, 92,274,306 24,76,131,195,321	298 [0], 328 [1]
50,100,150	344	103,136,262,332 117,148,312,—	

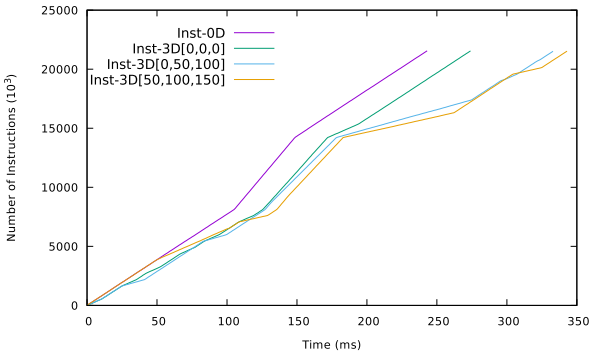


FIGURE 12. CC: Linear controller.

times until the end of the slot is reached or the cores are self-suspended. In this case, the actions of suspending and resuming NCCs are taken up to five times, resulting in an overload higher than when using the Limit Controller strategy. Figure 12 plots these scenarios in which the Linear Controller has been applied.

The advantage of this strategy is that the CPI of the experiments does not need to be determined. Directly, the slot duration in cycles permits determination of the CPI. However, this controller can produce several core suspension and resume significantly increasing hypervisor overhead.

D. ANALYSIS OF THE CONTROLLER PARAMETERS

In order to select the appropriate parameters for the CC-C, a set of experiments with random scenarios has been performed. The scenario is generated by defining random start times for NCC slots. DP_1 will randomly start in the interval [0, end of the first third], DP_2 in [0, end of the second third] and DP_3 in [0, end of the execution of CPart]. All the scenarios were executed 10 times with a configuration of N levels of Instructions (N = [2, 18, 2]) and decision limit ranged in [1.0, 0.9, 0.2]. As an example, the configuration N = 10 and decision limit = 0.92, means that the total number of instructions has been split in 12 limits and the decision to stop cores is taken if the CPI relation is higher than 0.92*CPI calculated as the limit of the secure zone.

The figures below show the results of this evaluation. Figure 13 shows the number of the CPart misses. A missed deadline is considered when CPart execution does not finish during the slot duration. Figure 14 shows NCC activity. The maximum CPU time is calculated as the sum of the three NCCs during the intervals in which they should be executed

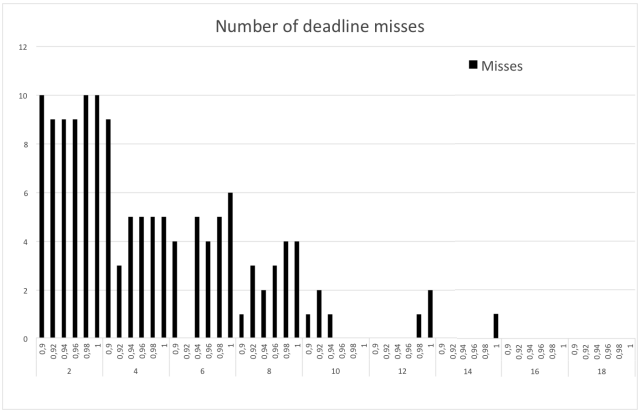


FIGURE 13. Number of misses in CPart.

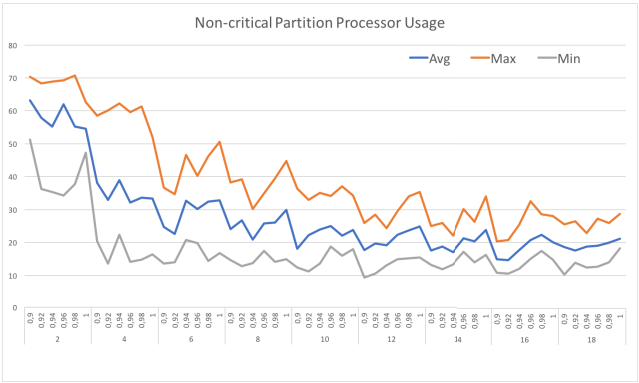


FIGURE 14. CPU utilisation of NCCs.

and overlaps with the CPart slot. The minimum time corresponds to the minimum CPU utilisation of NCC during the interval of which are active during the execution of CPart. Average value is the mean execution time of NCCs in all experiments under the same configuration parameters.

As can be seen, if N is higher than 10 and the margin is lower than 0.95 then the controller can allow the CPart computation to finish on time.

VIII. RELATED WORK

Different methods have been proposed or are pursued to derive guarantees for the timeliness of sets of tasks in a parallel workload setting when performance isolation is not given. Techniques can be static or measurement-based. The static analysis of an entire set of concurrently executed applications may deliver a sound and precise guarantee for timing behaviour. The problem is the huge complexity of this approach. Measurement-based methods are, in general, not able to derive guarantees in either the monocore or the multicore case. However, some works have demonstrated that with some constraints they can generate results good enough to be used in industry.

To cope with the resource constraints in computer systems control and scheduling codesign has been considered as a relevant topic. The integration of feedback control and real-time constraints has been extensively studied in the literature. Scheduling control co-design is a topic that combines the scheduling of real-time tasks and feedback control solutions [18], [19]. One of the first works [20] considered the period selection problem by means of a cost function. Other issues related to the control of real-time activities were analysed in several works such as task rate optimization with deadline constraints [21], feedback-based control of thermal management and control of media tasks [22], the use of linear quadratic Gaussian controllers for controlling the response time, memory as resource [23] and adjusting the task periods in given intervals to minimize the hyperperiod [24].

While previous works considered the control or adjustment of task-related parameters (period or deadline), other works focused on CPU utilisation as a resource. In [25] a framework was presented to deal with CPU control using PID controllers. The goal was to achieve higher CPU usage with a lower deadline miss ratio. A Constant Bandwidth Server (CBS) was proposed in [26] as a way of controlling the processing time of some tasks. In [27] a control server model based on the CBS approach was developed. Extensions to this CBS have been proposed in the literature [28]. In [29] an extension for multi-processors is presented. In [30] the optimal choice of scheduling parameters for control tasks is presented, assuming CBS scheduling.

Several works have pointed out the problems of shared resources in multi-core systems. In [31] and [32], there are reviews of the impact of shared buses, caches, and other resources on performance and performance prediction. In [33], an approach for WCET computation considering variable access delay due to shared resources is proposed. It also introduces techniques to analyse how applications use resources and compute the interference delay. In [34], it is proposed a technique to determine the bounded interference in multi-core systems. This technique requires to generate a profile of the execution and determine the WCET. In [35], a bus protocol based on TDMA-based memory arbiter jointly with a second, dynamic arbitration layer facilitates the interference-free integration of mixed-criticality applications. In [36] a global time-triggered scheduling approach with barrier synchronization is proposed for multicores.

In [37], a control of the running tasks accessing shared resources is presented. In [38], a memory guard mechanism is defined that regulates memory accesses. It is a regulation oriented mechanism that allocates a maximum bandwidth usage per timeslot. In [39], a distributed run-time WCET controller is proposed that stops low-criticality tasks (running on other cores) whenever it determines that their continued execution could cause a high-criticality task to fail to meet a deadline. In [40] this controller is combined with quality-of-services strategies to improve processor utilisation based on controller execution history.

IX. CONCLUSIONS

In this paper, we have proposed a feedback control implemented at hypervisor level that can control the execution of the critical applications on a multi-core platform under hypervisor execution. The controller scheme and its scope

have been defined. The proposed scope enables effective control only when critical applications are executed. The actions proposed to control the execution of critical applications are simple: suspend the execution of non-critical cores.

Two control strategies have been proposed and compared. First, it makes decisions to suspend non-critical cores as late as possible. This allows a single control action to be taken and reduces overload. The second one makes the decision to suspend when the performance line (CPI) is separated from the expected one, while if it is recovered, the non-critical cores are resumed.

The action of suspending all non-critical cores has two motivations: hypervisor decisions must be extremely simple in order to facilitate the future certification and to avoid complex decisions that can increase overheads.

Also, we proposed a controller tuning technique for the controllers. Experimentation has made it possible to analyze the influence of the controller configuration parameters on the control of the response time achievement. Experimentation has been carried out in a representative computer system used in avionics. Finally, we detailed the impact in terms of execution cycles that the proposed scheme adds to the original hypervisor services.

Future work is focused on: i) other control strategies for critical and non-critical cores, ii) different controller actions such as core frequency reduction, iii) how to reduce or remove the constraint of only one critical core executing critical partitions at any time, iv) extend the evaluation to other platforms, and vi) to analyse the effect of enabling/disabling cache can impact on the approach.

REFERENCES

[1] A. Burns and R. I. Davis, "Mixed criticality systems—A review," Dept. Comput. Sci., Univ. York, York, U.K., Internal Rep., 2017. [Online]. Available: www-users.cs.york.ac.uk/burns/review.pdf

[2] R. Ernst and M. di Natale, "Mixed criticality systems—A history of misconceptions?" *IEEE Design Test*, vol. 33, no. 5, pp. 65–74, Oct. 2016.

[3] J. Rushby, "Partitioning in avionics architectures: Requirements, mechanisms, and assurance," NASA Langley Res. Center, Hampton, VA, USA, Tech. Rep. NASA/CR-1999-209347, 1999.

[4] *Avionics Application Standard Software Interface*, Standard ARINC-653, Airlines Electronic Engineering Committee, Mar. 1996.

[5] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The XtratuM approach," in *Proc. EDCC*, 2010, pp. 67–72.

[6] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "XtratuM: A hypervisor for safety critical embedded systems," in *Proc. 11th Real-Time Linux Workshop*, Dresden, Germany, Sep. 2009, pp. 263–272.

[7] *Multi-Cores Partitioning for Trusted Embedded Systems*, document FP7-ICT-287702, MultiPARTES, 2011.

[8] *Distributed Real-Time Architecture for Mixed Criticality Systems*, document eU FP7-ICT-610640, DREAMS, 2013.

[9] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1999, pp. 12–21.

[10] C. Cullmann *et al.*, "Predictability considerations in the design of multi-core embedded systems," in *Proc. Embedded Real Time Softw. Syst.*, 2010, pp. 1–10.

[11] A. Crespo, A. Soriano, P. Balbastre, J. Coronel, D. Gracia, and P. Bonnot, "Hypervisor feedback control of mixed critical systems: The XtratuM approach," in *Proc. Workshop Oper. Syst. Platforms Embedded Real-Time Appl. (OSPERT)*, Dubrovnik, Croatia, Jun. 2017, pp. 35–40.

[12] *User Manual*, FreeScale, Austin, TX, USA, 2010.

[13] *Intel 64 and IA-32 Architectures. Software Developer's Manual*, vol. 3C, Intel Corp., Santa Clara, CA, USA, 2011.

[14] *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R Edition, Issue C*, document ARM DDI0406C, 2012.

[15] V. Brocal, M. Masmano, I. Ripoll, A. Crespo, P. Balbastre, and J.-J. Metge, "Xoncrete: A scheduling tool for partitioned real-time systems," in *Proc. Embedded Real-Time Softw. Syst.*, 2010, pp. 1–8.

[16] A. Crespo, P. Balbastre, J. Simo, and P. Albertos, "Static scheduling generation for multicore partitioned systems," in *Proc. Int. Conf. Inf. Sci. Appl. (ICISA)*, vol. 376, 2016, pp. 511–522.

[17] *XtratuM User Manual*, fentISS, València, Spain, Dec. 2015.

[18] K.-E. Årzén and A. Cervin, "Software and platform issues in feedback control systems," in *Cyber-Physical Systems*. Reading, MA, USA: Addison-Wesley, 2017, pp. 165–195.

[19] R. S. Fernández, J. E. S. Ten, J. L. N. Herrero, J. Poza-Lujan, and J. Posadas-Yagüe, "Núcleo de control: Controladores modulares en entornos distribuidos," *Rev. Iberoamer. Autom. Inf. Ind.*, vol. 13, no. 2, pp. 196–206, 2016.

[20] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin, "On task schedulability in real-time control systems," in *Proc. 17th IEEE Real-Time Syst. Symp. (RTSS)*, Washington, DC, USA, Dec. 1996, pp. 13–21.

[21] E. Bini and M. Di Natale, "Optimal task rate selection in fixed priority systems," in *Proc. 26th IEEE Real-Time Syst. Symp. (RTSS)*, Miami, FL, USA, Dec. 2005, pp. 399–409.

[22] M. Lindberg and K.-E. Årzén, "Feedback control of cyber-physical systems with multi resource dependencies and model uncertainties," in *Proc. 31st IEEE Real-Time Syst. Symp. (RTSS)*, San Diego, CA, USA, Nov./Dec. 2010, pp. 85–94.

[23] A. Marchand, P. Balbastre, I. Ripoll, R. Masmano, and A. Crespo, "Memory resource management for real-time systems," in *Proc. 19th Euromicro Conf. Real-Time Syst. (ECRTS)*, Pisa, Italy, Jul. 2007, pp. 201–210.

[24] V. Brocal, P. Balbastre, R. Ballester, and I. Ripoll, "Task period selection to minimize hyperperiod," in *Proc. IEEE 16th Conf. Emerg. Technol. Factory Automat. (ETFA)*, Toulouse, France, Sep. 2011, pp. 1–4.

[25] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao, "Feedback control real-time scheduling: Framework, modeling, and algorithms," *Real-Time Syst.*, vol. 23, nos. 1–2, pp. 85–126, 2002.

[26] L. Abeni and G. C. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. 19th IEEE Real-Time Syst. Symp.*, Madrid, Spain, Dec. 1998, pp. 4–13.

[27] A. Cervin and J. Eker, "The control server: A computational model for real-time control tasks," in *Proc. 15th Euromicro Conf. Real-Time Syst. (ECRTS)*, Porto, Portugal, Jul. 2003, pp. 113–120.

[28] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo, "IRIS: A new reclaiming algorithm for server-based real-time systems," in *Proc. 10th IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Toronto, ON, Canada, May 2004, pp. 211–218.

[29] S. K. Baruah, J. Goossens, and G. Lipari, "Implementing constant-bandwidth servers upon multiprocessor platforms," in *Proc. 8th IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, San Jose, CA, USA, Sep. 2002, pp. 154–163.

[30] D. Fontanelli, L. Palopoli, and L. Greco, "Optimal CPU allocation to a set of control tasks with soft real–time execution constraints," in *Proc. 16th Int. Conf. Hybrid Syst. (HSCC)*, Philadelphia, PA, USA, Apr. 2013, pp. 233–242.

[31] A. Abel *et al.*, "Impact of resource sharing on performance and performance prediction: A survey," in *Proc. 24th Int. Conf. CONCUR*, Buenos Aires, Argentina, Aug. 2013, pp. 25–43.

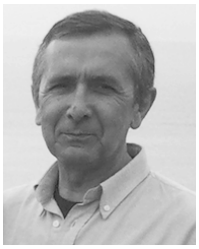
[32] J. Reineke and R. Wilhelm, "Impact of resource sharing on performance and performance prediction," in *Proc. DATE*, Dresden, Germany, Mar. 2014, pp. 1–2.

[33] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement," in *Proc. 26th Euromicro Conf. Real-Time Syst. (ECRTS)*, Madrid, Spain, Jul. 2014, pp. 109–118.

[34] S. Esposito, M. Violante, M. Sozzi, M. Terrone, and M. Traversone, "A novel method for online detection of faults affecting execution-time in multicore-based systems," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 4, pp. 94:1–94:19, 2017.

[35] B. Cilku, A. Crespo, P. Puschner, J. Coronel, and S. Peiro, "A TDMA-based arbitration scheme for mixed-criticality multicore platforms," in *Proc. Int. Conf. Event-Based Control, Commun., Signal Process. (EBCCSP)*, Krakow, Poland, 2015, pp. 17–19.

[36] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *Proc. Int. Conf. Embedded Softw. (EMSOFT)*, Montreal, QC, Canada, 2013, pp. 17:1–17:15.
[37] S. Girbal, X. Jean, J. L. Rhun, D. G. Pérez, and M. Gatti, "Deterministic platform software for hard real-time systems using multi-core cots," in *Proc. 34th Digit. Avionics Syst. Conf. (DASC)*, Prague, Czech Republic, Sep. 2015, pp. 8D4-1–8D4-15.
[38] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Proc. 19th IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Philadelphia, PA, USA, Apr. 2013, pp. 55–64.
[39] A. Kritikakou *et al.*, "Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems," in *Proc. 22nd Int. Conf. Real-Time Netw. Syst. (RTNS)*, Versaille, France, Oct. 2014, p. 139.
[40] S. Heywood, "Scheduling mixed-criticality multi-core systems to maximise resource utilisation," Ph.D. dissertation, Dept. Comput. Sci. Eng., Univ. Gothenburg, Gothenburg, Sweden, 2016.



ALFONS CRESPO received the Ph.D. degree in computer science from the Universitat Politècnica de València (UPV), Spain, in 1984. He is currently a Professor with the Department of Computer Engineering, UPV. He also leads the group of Industrial Informatics and has been the responsible of several national and European research projects including OCERA (coordinator), FRESCOR, ARTIST2, OVERSEE, MultiPARTES, and DREAMS. He has published over 100 papers in specialized journals and conferences in the area of real-time systems. His main research interests include different aspects of the real-time systems (scheduling, virtualization techniques, scheduling, and control integration).

PATRICIA BALBASTRE received the degree in electronic engineering and the Ph.D. degree in computer science from the Universitat Politècnica de València (UPV), Spain, in 1998, and 2002, respectively. She is currently a Professor with the Department of Computer Engineering, UPV. She has participated in various Spanish and European projects including OCERA, FRESCOR, VOS4ES, MultiPARTES, and DREAMS. Her main research interests include real-time operating systems, dynamic scheduling algorithms, and real-time control.

JOSÉ SIMÓ received the M.S. degree in industrial engineering and the Ph.D. degree in computer science from the Universitat Politècnica de València (UPV), Spain, in 1990 and 1997, respectively. Since 1990, he has been involved in several Spanish and European research projects mainly related to Real-Time and Embedded Systems and Industrial Collaborations. He is currently a Professor with the Department of Computer Engineering, UPV. His current research is focused on the development of real-time embedded systems, autonomous systems, and robotics.

JAVIER CORONEL received the Ph.D. degree in computer science from the Universitat Politècnica de València, Spain, in 2014. He is currently a Software Engineer Specialist in virtualization for real-time multicore embedded systems. He is also in charge of the hardware, validation, and development activities. He has published several papers in the topic of virtualization and scheduling for real-time systems.

DANIEL GRACIA PÉREZ received the Ph.D. degree in computer architecture from Paris XI University and degrees in engineering from the Kungliga Teckniska Högskolan, Sweden, and the Universitat Politècnica de Catalunya, Spain. He has participated in various French and European projects including ANR SoCLib (workpackage coordinator), ANR Hecosim (workpackage coordinator), CATRENE COMCASS, OPEES, ITEA TWINS, FP7 Certainty, and FP7 DREAMS. He is currently a Research Engineer with Thales Research & Technology, France. His current research consists on the development and application of new multi-core architectures for safety-critical systems.

PHILIPPE BONNOT received the degree from the Ecole Nationale Supérieure des Télécommunications de Paris in 1988. He has experience of complex SOC design, parallel architecture of digital signal processor, and associated development tools developed in Thales Communications. He notably involved in SIMD architectures designed for space on-board signal processing. He has been a Manager of European Project MAGICFPU. He was a Chief Architect with the ATMEL DSP Design Center. He initiated design of massively parallel architectures for image processing implemented on FPGA. He has been a Coordinator of IST FP6 MORPHEUS integrated project about reconfigurable architecture and tools. He is currently the Leader of the Critical Embedded Systems Lab, Thales Research & Technology.