Pythia-MCS: Enabling Quarter-Clairvoyance in I/O-Driven Mixed-Criticality Systems

Zhe Jiang*‖, Kecheng Yang†, Nathan Fisher‡, Neil Audsley*, Zheng Dong‡§
*University of York, United Kingdom, †Texas State University, USA, ‡Wayne State University, USA
‖ARM Ltd, United Kingdom

***Abstract*—In mixed-criticality systems, mode switch is a key strategy which dynamically provides a balance between system performance and safety. In conventional MCS frameworks, mode switch is triggered by the *over-execution* of a task; *i.e.*, a task overruns the less pessimistic worst-case execution time. In cyber-physical systems, the data volume generated by I/O affects and can even dominate task computation time. With this in mind, we introduce a novel MCS architecture, termed *Pythia*-MCS, which predicts task execution time according to I/O run-time behaviors. With the new feature of *future-prediction*, the *Pythia*-MCS provides more timely, but still accurate, mode switch. We also present a new theoretical model (*quarter-clairvoyance*), which guarantees the timing predictability of the design, and a new schedulability analysis for the *Pythia*-MCS, which demonstrates improved schedulability compared to conventional MCS frameworks. The *Pythia*-MCS is the first MCS framework enabling the clairvoyance functionality.**

I. Introduction

Safety-critical systems have stringent assurance and verification requirements that are absolutely essential to life-critical applications including medical, automotive, aerospace and industrial automation [7]–[9], [16], [23], [34], [57]. In safety-critical systems, integrating components with different levels of criticalities, such as *ASILs* (Automotive Safety and Integrity Levels) in ISO26262 [27], onto a shared hardware platform has become increasingly important [15], [20], as a result of the diverse functionalities required by modern safety-critical systems (*e.g.*, automated driving [27]) and the rapid evolution of underlying platforms [8]. Such systems are termed *Mixed-Criticality Systems (MCS)s* [50].

In a dual-criticality MCS,[1] which has two criticality levels (HI and LO), a widely studied theoretical model assumes that the Worst-Case Execution Time (WCET) of a task is estimated with different levels of confidence [5], [11], [23], [30], [37], [50], [59]. The high-critical WCET (HI-WCET) is confident, but extremely pessimistic (obtained by static timing analysis, for example); whereas, the low-critical WCET (LO-WCET) is much less pessimistic, but has relatively lower confidence (obtained by measurement, for example [50]). In general, a high-critical task (HI-task) is developed and verified with more rigorous procedures than a low-critical task (LO-task). Therefore, a HI-task has both HI- and LO-WECTs; whereas, a LO-task only has LO-WCET [27], [50]. The correctness criterion in this model specifies that if all tasks finish executing within their LO-WCETs, then they will all finish executing by their deadlines. However, if any task does not complete execution within its LO-WCET, then the HI-tasks at least should complete execution by their deadlines [13], [38], [50]. Therefore, only HI-tasks are guaranteed.

Mode switch [5], [50] is the key strategy used to satisfy the above criterion. The system first executes in low-critical mode (LO-mode), in which the scheduling policy assumes the execution time of each task (LO-task or HI-task) does not exceed its LO-WCET. If this assumption is violated, the system switches into high-critical mode (HI-mode), in which the scheduling policy assumes the execution time of HI-tasks may exceed their LO-WCETs, but will not exceed their HI-WCETs [17], [29], [35], [40], [49].

Within the context of the theoretical model, a number of practical MCS frameworks have been developed. These include, Richard *et al.* [52], Gadepalli *et al.* [22], Kim *et al.* [33] and Pinto *et al.* [45]. In most of these frameworks, the only way to trigger a mode switch is by executing the system until a HI-task overruns its LO-WCET [12], [22], [42], [52], [58]. This type of system framework is called a "*non-clairvoyant MCS*" [1], [4]. Although Baruah *et al.* [4] and Agrawal *et al.* [1] provide solid theoretical support to show that *clairvoyant* and *semi-clairvoyant* MCSs[2] usually outperform non-clairvoyant MCSs, they also report that establishing an MCS framework with a degree of clairvoyance is challenging. This is because most run-time situations must be known beforehand [1], [4], and for example, it is difficult to predict an external environmental change before it happens.

I/O-driven MCS. *Inputs/Outputs* (*I/O*s) are a vital part of any computer/embedded architecture [26], [43] as they connect the digital and physical worlds. The role of I/Os within an MCS must be considered carefully, as the volume of input data may significantly affect the execution time of a task, which determines the necessity of a mode switch. Taking an autonomous vehicle as an example, a sensor/lidar usually receives an additional volume of data in an urgent situation, *e.g.*, a greater number of objects to be identified and tracked compared with driving on an empty road, with no objects to be identified and tracked [27], [39]. Therefore, a mode switch may be triggered due to more computation time being required by a task to process the additional received data.

With this in mind, we propose a novel MCS framework architecture, which we term ***Pythia-MCS***. In *Pythia*-MCS, I/O behaviors are continuously monitored and analyzed, and

§Corresponding author, dong@wayne.edu.

[1]Like much of the current research on mixed-criticality scheduling, this paper restricts attention to two criticality levels.

[2]Clairvoyant and semi-clairvoyant MCSs assume that whether a HI-task will overrun its LO-WCET is known before or at its release; hence, the system can trigger a mode switch before the HI-task overruns its LO-WCET.

a mode switch can be triggered when a large amount of data is generated by an I/O. We term this ***I/O-driven mode switch***. With I/O-driven mode switch, the proposed framework architecture enables a certain level of clairvoyance, as the system can trigger a timely mode switch instead of waiting until the overrun happens.

Different from the clairvoyant and semi-clairvoyant MCS theoretical models provided in [1], [4], *Pythia*-MCS is a practical framework architecture, which has been specifically designed and implemented. *Pythia*-MCS also has a new theoretical model, which we call ***quarter-clairvoyance***. Unlike clairvoyant and semi-clairvoyant theoretical models, which assume the over-execution of a task is known before or at release, quarter-clairvoyance predicts the over-execution of a task during its execution. Therefore, with quarter-clairvoyance, an MCS can react to a run-time situation more accurately.

Contributions. This is the first practical MCS framework enabling the functionality of clairvoyance. To this end, we propose a novel system architecture, which simultaneously supports run-time I/O monitoring and I/O-driven mode switch. We describe the design details of the proposed system with two optional design methods. Corresponding to the proposed architecture, we further present a new theoretical model (quarter-clairvoyance) and schedulability analysis, which provides the timing guarantee for the system. Moreover, the theoretical analysis also demonstrates the improvements on the schedulability brought by *Pythia*-MCS, with respect to conventional MCS frameworks. We present experiments to evaluate the hardware and software overheads of *Pythia*-MCS. Additionally, we examine the benefits and prediction accuracy of *Pythia*-MCS over a conventional MCS using a real-world automotive use case.

The rest of this paper is organized as follows: Section II presents the motivation and research challenges. Sections III and IV give the system architecture and design methods of *Pythia*-MCS, followed by schedulability analysis given in Section V. Section VI evaluates the *Pythia*-MCS, and Section VII concludes.

II. Motivation: I/O-Driven MCS

As introduced in Section I, I/O is the key to establishing an MCS framework with clairvoyance. In this section, we decompose the relationships between task execution time and I/Os, then explain the concepts of an I/O-driven MCS. Lastly, we present the research challenges.

A. I/Os and Task Execution Time

To understand the relationship between I/Os and task execution time, a task can be decomposed into:

I/O-independent computation – includes pure software calculation without I/O access. Computation time usually depends on system micro-architectures [10], such as CPU architecture, branch-prediction, memory bandwidth, etc.

I/O-related computation – includes I/O accesses and I/O-bounded calculation. Computation time is usually determined by the data volume generated by the I/Os [31].

If a task involves I/O-related computation, we call it an *I/O-related task*, otherwise it is an *I/O-independent task*.

Algorithm 1: Pseudo-Code an Ethernet Control Task

```
1  RawPacket[i] = ∅; Buf = ∅;
2  if (System.Status() == Correct) then
3      while (IO.Status (Ethernet.ID) == Busy)³ NOP;
4      PacketSize = I/O.Check (Ethernet.ID, Recv);
5      if (PacketSize > 0) then
6          for i = 0; i < PacketSzie; i + + do
7              I/O.Read (Ethernet.ID, Buf[i]);
8          end
9          for i = 0; i < PacketSzie; i + + do
10             RawPacket[i] = AUTOSAR.E2E.Decoding (Buf,
                 i× PacketLen)
11         end
12     else
13         NOP;
14     end
15 else
16     Err.Ctrl();
17 end
```

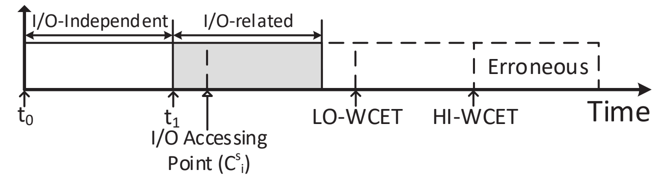


Fig. 1. Ethernet Control Task Timing Chart

Algorithm 1 uses pseudo-code to demonstrate an example of an Ethernet control task (I/O-related task) from Renesas' automotive use cases [18]. I/O-related computation is highlighted in blue, with the status check in line 3 and 4, the Ethernet read in line 7 and E2E decoding in line 10. Based on the pseudo-code, Figure 1 further illustrates the timing chart for this task. As shown, the task releases at time point t_0 with I/O-independent computation and changes to I/O-related computation at time point t_1. Additionally, Figure 1 highlights the LO-WCET and HI-WCET estimates for the task. In this example, the executing times of I/O-independent computation (*e.g.*, buffer initialization) are constant; whereas, the executing times of the I/O-related computation (Ethernet read and E2E decoding) vary with the volume of received Ethernet packets. Clearly, in an I/O-intensive system, the data volume generated by I/Os affects, and can even dominate, the task execution time.

With this observation, we can predict the execution time of an I/O-related HI-task at its I/O access point and then determine the necessity of a mode switch before task overrun. We term this ***I/O-driven mode switch***. The MCS enabling I/O-driven mode switch is termed an ***I/O-driven MCS***.

B. I/O-driven Mode Switch

Achieving an I/O-driven mode switch based on a conventional MCS model requires two more features for each I/O-related HI-task, which must be acquired offline:

[3] The busy-waiting loop must be monitored by a timeout monitoring to bound the worst-case scenario.

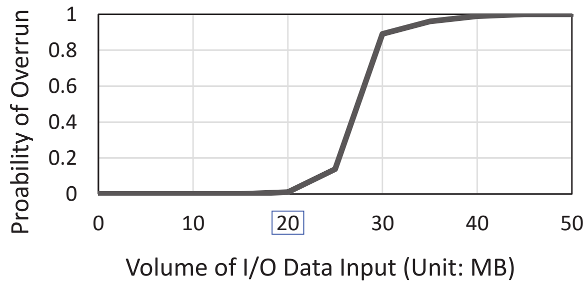


Fig. 2. Find TH-I/O for Ethernet Control Task

I/O access point (denoted C_i^{S}). I/O-related computation always starts with processing I/O accesses (*e.g.*, line 7 in Algorithm 1), which obtains the I/O data packets to be processed in the following computation. The I/O data received before/after C_i^{S} will be processed in the current/next task release.

Threshold I/O data volume (TH-I/O, denoted Υ_i^{L}). At C_i^{S}, if the data volume accumulated by the task (denoted υ_i) exceeds its TH-I/O (*i.e.*, $\upsilon_i > \Upsilon_i^{\mathrm{L}}$), we can predict that the task will exceed its LO-WCET, and therefore a mode switch is required.

Similar to the other tuples in the system (specifically described in Section V), the two introduced features can be obtained using either *static analysis* or *experimental measurements*. Here, we give a brief introduction to finding the experimental measurements.

Finding experimental measurements for C_i^{S} and Υ_i^{L}. Firstly, we removed the non-examined tasks and initialized the system without any I/O data input. We then linearly increased the volume of I/O data input and executed the system $10,000$ times under each system configuration. In each experiment, we recorded the I/O access time-point and checked whether the examined task overran its LO-WCET. Following the experiments, the probability of task over-execution under different volumes of I/O data input was plotted. The system designer was then able to select an appropriate TH-I/O for the examined task based on the experiment results. The results of the example above measured on our experimental platform (Xilinx VC709 [56] with the configurations introduced in Section VI) are shown in Figure 2. We chose 20 MB as the TH-I/O for the examined task *i.e.*, over-execution may occur when the I/O data volume is greater than 20 MB.

C. Research Challenges

Given the previously detailed concepts, it is possible to drive a mode switch based on I/O run-time behaviors and create an MCS with a certain level of clairvoyance. However, creating an I/O-driven MCS is not straightforward as several key challenges must be addressed:

- The data volume generated from each I/O must be monitored online. An I/O is often invoked by multiple tasks; hence, online monitoring must promptly and precisely determine I/O data volume for each corresponding task.
- The data volume sent to each task must be timely compared with its TH-I/O. The comparison must occur at the corresponding I/O access point (or within a small margin).
- New system architecture is required to support the features described above. In practice, tasks may only contain I/O-independent computation (*i.e.*, I/O-independent tasks); hence, the proposed system architecture must simultaneously support both I/O-driven and conventional MCS models.
- New schedulability analysis frameworks are needed to theoretically evaluate the schedulability improvement yielded by the novel I/O-driven MCS model.

Below, we introduce the new *Pythia*-MCS and the corresponding schedulability analysis, as a solution to these research challenges.

III. *Pythia*-MCS ARCHITECTURE

In this section, we give an overview of the *Pythia*-MCS, presenting the top-level design concepts and system architecture.

A. Context

In this paper, we assume:

- The platform is an embedded Network-on-Chip (NoC);
  - *Pythia*-MCS is agnostic to the types of bus;
  - Deployment of NoC can enhance the predictability of on-chip transactions [14], [46].
- *Pythia*-MCS is applicable to both single- and many-core architectures. A *fully-partitioned* scheme is adopted in a multi-/many-core *Pythia*-MCS.
  - Tasks are *statically* assigned to a given processor.
  - Existing task allocation heuristic [8] (*e.g.*, first-fit) can be applied directly for partitioning.
- A task can access one I/O at most, whereas an I/O can be accessed by multiple tasks.

B. Design Concepts

Pythia-MCS has two main design concepts:

Design Concept 1 – online I/O monitoring. The *Pythia*-MCS introduces the *Pythia*-coprocessor, which monitors and analyzes run-time data generated from the I/Os. Monitoring I/Os from the hardware layer guarantees the accuracy and timeliness of the captured transactions.

Design Concept 2 – adaptive mode switch. The *Pythia*-MCS supports both I/O-driven and conventional mode switches. Before run-time, the TH-I/O of each I/O-related HI-task and the LO-WCET of each I/O-independent HI-task are preloaded to the coprocessor. During run-time, an I/O-related HI-task exceeding the TH-I/O or an I/O-independent HI-task exceeding the LO-WCET is detected by the coprocessor and a mode switch triggers.

In the context of conventional non-clairvoyant MCS theory, a number of practical frameworks have been proposed, for example Richard *et al.* [52], Gadepalli *et al.* [22], Jiang *et al.* [32], Kim *et al.* [33], and Xi *et al.* [53]. To ensure compatibility with the state-of-the-art, the proposed system architecture for *Pythia*-MCS derived from conventional MCS

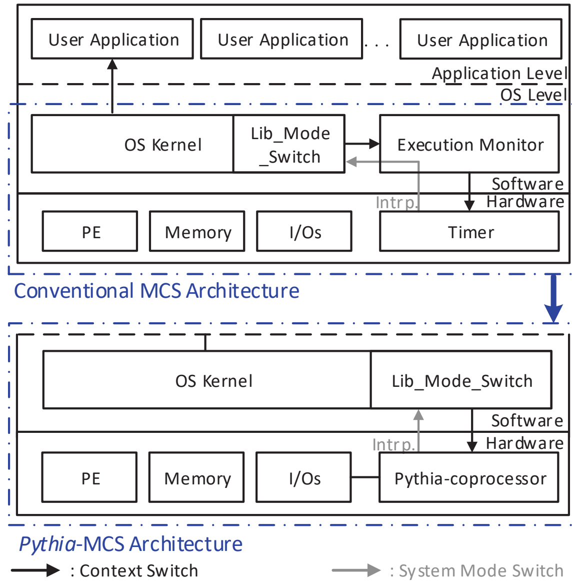


Fig. 3. System Architectures of Conventional MCS and *Pythia*-MCS

frameworks. Therefore, Section III-C first reviews conventional MCS architecture, Section III-D then presents the system architecture.

C. Conventional MCS System Architecture

The generalized architecture of a conventional MCS (shown in the upper part of Figure 3) is illustrated by considering conventional embedded/computer architectures with an additional execution monitor, usually implemented at the Operating System (OS) level to give more privileges than user applications. Two essential functionalities must be supported by the execution monitor: 1) monitoring task execution time; and, 2) triggering a mode switch when detecting the over-execution of a HI-task. These two functionalities are achieved using co-operation between a dedicated timer in the hardware and an additional library in the OS kernel (named *lib_mode_switch*). Note that the execution monitor can be implemented using different methods. For example, Kim *et al.* [33] integrate the execution monitor with the OS kernel, while Li *et al.* [41] and Xi *et al.* [53] implement the execution monitor as an independent hypervisor.

Run-time behaviors. At system initialization, the LO-WCETs of the HI-tasks are preloaded to the memory. During context switches, the OS kernel suspends the timer of the currently executing task and then (re-)activates the timer for the next executing task. If a HI-task runs over its LO-WCET, an interrupt sent from the hardware timer will trigger the execution of *lib_mode_switch* for the mode switch. The pseudo-code demonstrating this procedure is shown in Algorithm 2.

D. Pythia-MCS System Architecture

The *Pythia*-MCS has architecture changes in both the hardware and software layers compared to conventional MCS system architecture (shown in the lower part of Figure 5):

Hardware layer. As introduced in the design concepts, the run-time monitoring and the mode switch triggering in the proposed architecture are managed by the *Pythia*-coprocessor. Hence, in the hardware layer, we replace the timer (monitoring task execution time in the conventional MCS architecture) with the new coprocessor. We present the design details of the coprocessor in Section IV.

Algorithm 2: Context and Mode Switch in Conventional MCS

```
1  ▷ OS Kernel: Context Switch
2  Intrp.disable();
3  ExeMonitor.Timer.suspend (TaskSet.Current.ID);
4  ExeMonitor.Timer.activate (TaskSet.Next.ID);
5  Scheduler.run (TaskSet.Next.ID);
6  Intrp.enable();
7  ▷ Interrupt Handler: Mode Switch
8  Function Timeout_ISR(Timer.ID) :
9      Lib_mode_switch (HI-Mode);
10     Intrp.clear(Timer.ID);
11 End Function
```

Algorithm 3: Context and Mode Switch in *Pythia*-MCS

```
1  ▷ OS Kernel: Context Switch
2  Intrp.disable();
3  Coprocessor.sync (TaskSet.Next.ID);
4  Scheduler.run (TaskSet.Next.ID);
5  Intrp.enable();
6  ▷ Interrupt Handler: Mode Switch
7  Function Pythia_ISR() :
8      Lib_mode_switch (HI-Mode);
9      Intrp.clear();
10 End Function
```

Software Layer. Like the hardware timer, we also remove the execution monitor (which manages the hardware timer in the conventional MCS architecture) from the OS level. In the *Pythia*-MCS, the interrupt sent from the *Pythia*-coprocessor, triggering a mode switch, is directly routed to the *lib_mode_switch* in the OS kernel. The removal of the execution monitor effectively reduces the software overhead and system complexity compared to conventional solutions. We analyze the improvements in Section VI-A.

Run-time behaviors. The new system architecture also brings different run-time behaviors compared to the conventional MCS frameworks. At system initialization, the I/O-related HI-task LO-WCETs and the I/O-independent HI-task TH-I/Os are preloaded to the coprocessor. During context switches, the OS kernel synchronizes the ID of the scheduled task with the coprocessor (see line 3 of Algorithm 3). If an I/O-independent HI-task exceeds its LO-WCET or an I/O-related HI-task exceeds its TH-I/O, the coprocessor generates an interrupt to trigger mode switch by invoking *lib_mode_switch*. The pseudo-code demonstrating this procedures is shown in Algorithm 2.

Compatibility. Although the *Pythia*-MCS introduces a new system architecture, the design minimizes modifications to the software (shown in the comparison of Algorithms 2 and 3). Moreover, the design maintains the original OS-application interfaces presented by the traditional MCS (shown in Figure 3). Therefore, user applications designed for a conventional MCS can be mapped to the *Pythia*-MCS directly.

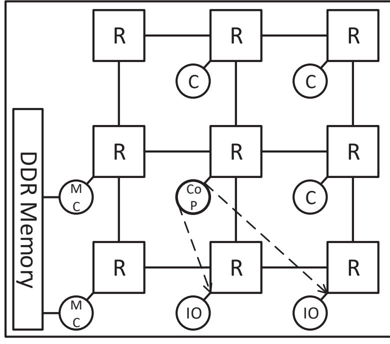


Fig. 4. *Pythia*-coprocessor in a NoC System (C: Processor Core; Cop: *Pythia*-coprocessor; R: Router/Arbiter; MC: Memory Controller)

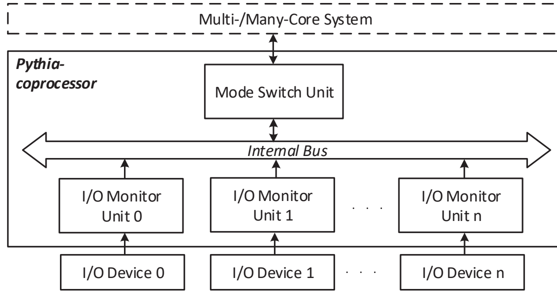


Fig. 5. Top-level Design of *Pythia*-coprocessor

In the new system architecture, acquiring the functionality of clairvoyance relies on the coprocessor; we therefore present the *Pythia*-coprocessor design details in the next section.

IV. *Pythia*-COPROCESSOR

The typical use of the *Pythia*-coprocessor in a NoC-based many-core architecture is shown in Figure 4, where the coprocessor is physically connected to a router/arbiter for on-chip communication, and different I/O devices for run-time I/O monitoring. The design of the *Pythia*-coprocessor (see Figure 5) is modularized, comprising two primary modules:

I/O Monitor Unit (*IMU*) – observes the run-time status of the connected I/O, and decomposes the I/O data packets, then reports the volume to the Mode Switch Unit (MSU). The design of the IMU is generic, and can be directly applied to different I/Os in the same system.

Mode Switch Unit (*MSU*) – checks the necessity of a mode switch. The design of the MSU is generic, and can be directly applied to different systems.

These two modules are introduced in the following sections.

A. I/O Monitor Unit (IMU)

The I/O monitoring methods and data decomposition, which are the main concern of the IMU are explained first. The design details of the IMU are then discussed.

I/O monitoring. From the perspective of embedded/computer architecture, accessing I/Os from software tasks usually involves following system components like the OS kernel, I/O drivers, bus interconnects and I/O controllers [24], [47]. Therefore, it is possible to monitor the behaviors of an I/O from any system level. I/O monitoring, which is considered to guarantee the timeliness of monitored transactions and the compatibility of data decomposition (Design Concept 1), is placed in the hardware layer. This is between the I/O controllers and the bus interconnects (shown in Figure 4).

From the selected system level, we have a unified method for I/O data decomposition, which is introduced below.

I/O data decomposition. I/O data decomposition returns the volume and destination of each I/O packet.[4] Decomposing an I/O data packet from the selected level (*i.e.*, between I/O controllers and bus interconnects) requires a clear understanding of the protocol specifications for on-chip communications. Here, we explain the I/O data decomposition using the example of AMBA AXI [3], which is the most commonly used protocol for on-chip communications in embedded architectures [3] and is also used in our experimental platform.

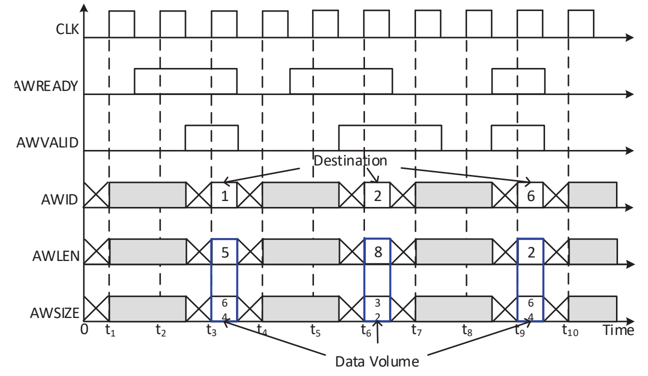


Fig. 6. Example of Write Address Channel (Waveform) in AMBA AXI

The AMBA AXI protocol contains five communication channels: write/read address channels, write data channel and write/read response channels. An on-chip transaction always initializes from the write/read address channel, which presents the necessary information of the transaction. Hence, the IMU is only required to monitor these two channels. For example, in the write address channel, a transaction initializes by setting the AWVALID and AWREADY signals to 1. At the same time, the control signals AWID, AWLEN and AWSIZE become valid for representing the destination, length and size of the transaction.[5] The data volume of this transaction (denoted as v^*) is calculated in Equation 1.

$$v^* = \text{AWLEN} \times \text{AWSIZE}, \text{ if AWVALID \& AWREADY} = 1 \quad (1)$$

As shown in Figure 6, the example initializes three I/O data packets, which are sent to tasks 1, 2 and 6 with volume $(5 \times 64 \div 8 =)$ 40, 32 and 16 bytes, respectively.

IMU design (Figure 7). The design of an IMU contains: a run-time sampler, an access interface and memory banks.

[4]The *destination* of an I/O packet means the task receiving the I/O packet.

[5]The relation between AWID and a task ID is defined by the system designer. In this paper, we consider these two IDs are always equal.

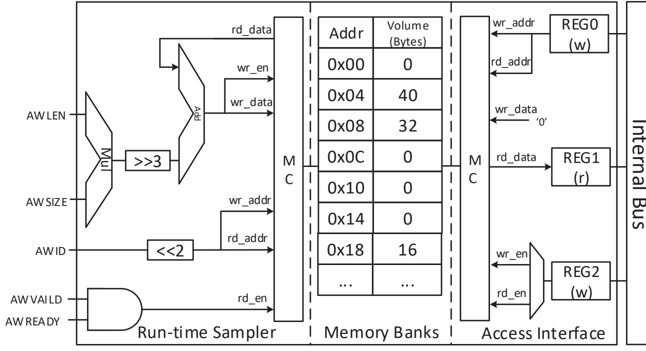
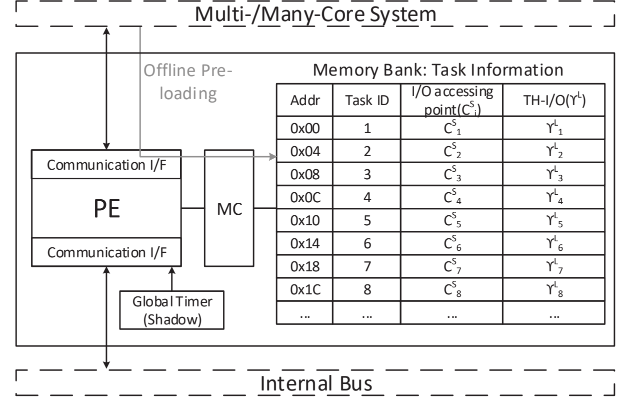
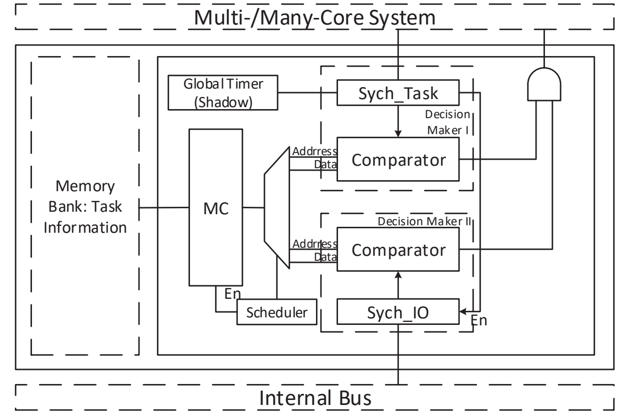


Fig. 7. Design of IMU (MC: Memory Controller)



Addr	Task ID	I/O accessing point(C^S_i)	TH-I/O(Υ^L_i)
0x00	1	C^S_1	Υ^L_1
0x04	2	C^S_2	Υ^L_2
0x08	3	C^S_3	Υ^L_3
0x0C	4	C^S_4	Υ^L_4
0x10	5	C^S_5	Υ^L_5
0x14	6	C^S_6	Υ^L_6
0x18	7	C^S_7	Υ^L_7
0x1C	8	C^S_8	Υ^L_8
...	...	...	...

(a) Hardware/Software Co-design



(b) Hardware-Only Design

Fig. 8. Design of MSU (MC: Memory Controller)

The memory banks store the volume of unprocessed data for each task (*i.e.*, υ_i). The memory address reserved for υ_i is calculated as $i \times 0x04$. During system execution, the sampler decomposes each captured I/O packet using the previously introduced method, returning its destination (*i.e.*, task τ_d) and volume (*i.e.*, υ^*). The sampler then adds υ^* to the volume of unprocessed data for τ_d (*i.e.*, $\upsilon_d = \upsilon_d + \upsilon^*$) and stores the calculated result back in the corresponding address in the memory (*i.e.*, $d \times 0x04$).

Additionally, the access interface introduces two control registers and a data register, accessed by the MSU via an internal bus. Write-only *Register 0* determines the operated address of the memory bank, *Register 2* controls operations (*e.g.*, value clear), and *Register 1* (read-only) reports the unprocessed data volume of the selected memory address given by *Register 0*. For example, to acquire the task τ_4 unprocessed data volume, the MSU first sets *Register 0* to $0x10$, then reads data from *Register 1*.

B. Mode Switch Unit (MSU)

The MSU is the brain of the *Pythia*-coprocessor. It triggers mode switch during run-time. As introduced in Design Concept 2, a mode switch is triggered by: 1) any I/O-related HI-task exceeding its TH-I/O at the I/O access point; or, 2) any I/O-independent HI-task exceeding its LO-WCET. To optimize the design of the MSU, we set a virtual I/O access point and a virtual TH-I/O for each I/O-independent HI-task, where the virtual I/O access point was the LO-WCET and the TH-I/O was -1. Therefore, when an I/O independent task executes at its virtual I/O access point, the task will always exceed the corresponding TH-I/O. This method unifies the criteria for mode switch for both I/O-related and I/O-independent HI-tasks.

The MSU determines the necessity of a mode switch using three executing phases:

Phase 1 - Offline preloading: before run-time, the (virtual) I/O access point (C_i^{S}) and (virtual) TH-I/O (Υ_i^{L}) of each HI-task (τ_i) are grouped and stored in the MSU.

Phase 2 - Online synchronization: during run-time, the MSU continuously synchronizes with the OS kernel, which updates the computation time (C_i) of the currently executing HI-task (τ_i), and the IMUs, which update the currents HI-task's unprocessed I/O data volume (υ_i).

Phase 3 - Decision making: at C_i^{S} of each τ_i, the MSU compares the υ_i against Υ_i^{L}. If $\upsilon_i > \Upsilon_i^{\mathrm{L}}$, the MSU triggers an interrupt for mode switch. After comparison, the MSU resets υ_i to 0, as the data will be now processed by τ_i.

To support these three executing phases, we introduce two possible MSU design methods:

Hardware/software co-design (Figure 8(a)). The hardware/software co-design propounds software executed on a ready-built processor (*e.g.*, MicroBlaze [55] or RISC-V [51]). The preloaded (virtual) I/O access point and (virtual) TH-I/O of each HI-task (Phase 1) are stored in a memory unit; the run-time synchronization and comparison (Phases 2 and 3) is handled by the software executed on the processor.

Hardware-only design (Figure 8(b)). Compared to the hardware/software co-design, the hardware-only method retains the memory unit, but replaces the processor with two decision-makers. Each decision-maker contains a synchronizer and a compactor. Decision-maker I synchronizes with the OS kernel and then compares the synchronized result with the (virtual) I/O access point. Decision-maker II synchronizes with the IMU and then compares the synchronized result with the (virtual) TH-I/O. When both decision-makers return 1, an interrupt for mode switch is generated. Note that Decision-maker I returns 1 when $C_i = C_i^{\mathrm{S}}$. Decision-maker II, returns $\upsilon_i > \Upsilon_i^{\mathrm{L}}$.

In both methods we introduce a shadow register to guarantee timing synchronization between MSU and the entire system.

Until now, we have described the system architecture and the design methods of the *Pythia*-MCS. In the next section, we study the benefits for schedulability analysis that can be obtained from enabling clairvoyance in the *Pythia*-MCS.

V. Quarter-Clairvoyance Schedulability Analysis

Although clairvoyance in general indicates the ability to look into the future, in MC scheduling, a few different degrees of clairvoyance are investigated in the recent literature [1]. An intermediate concept of *semi-clairvoyance*, which lies between the two extremes of clairvoyance and non-clairvoyance, has been introduced [1]. The terms are briefly explained below:

Clairvoyance. Whether *any* job will overrun its LO-WCET is *known from the beginning*, *i.e.*, at time 0. That is, whether this system run is in LO- or HI- mode would have been known before the system started.

Semi-Clairvoyance. Whether a job will overrun its LO-WCET becomes known right at the release of a job. The system is notified of a mode switch from LO to HI *at the release* of the first job that will overrun its LO-WCET.

Non-Clairvoyance. Whether a job will overrun its LO-WCET remains unknown until an overrun is *observed* during run-time. The system can only be notified of a mode switch from LO to HI when a job *misses its* LO-*WCET*, but has not completed.

In terms of the above terminology, our system architecture provides a certain degree of clairvoyance, as it falls between the two extremes of clairvoyance and non-clairvoyance. However, the limitations of the clairvoyance our architecture provides does not exactly match the limitations defined by semi-clairvoyance. In particular, our architecture enables *looking-into-the-future*, but not when a job releases; the job needs to execute for a certain amount of time first (up to LO-WCET). Therefore, we position the degree of clairvoyance our system architecture provides between semi-clairvoyance and non-clairvoyance. We call this degree of clairvoyance ***quarter-clairvoyance***, specified in more detail below.

A. System Model

We consider the scheduling of a set of n MC tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ on a single processor to which τ is assigned. Each MC *sporadic* task τ_i releases a (potentially infinite) sequence of jobs with a minimum separation of T_i time units between any two consecutive jobs of τ_i, where T_i is the *period* of τ_i. The jth job of task τ_i is denoted $\tau_{i,j}$. It is released at time $a_{i,j}$ and has an absolute deadline at $d_{i,j} = a_{i,j} + D_i$ where D_i is the relative deadline of task τ_i. We focus on implicit deadlines, *i.e.*, $D_i = T_i$ for all i.

We consider a dual-criticality task system, where each task in τ is a HI-task or a LO-task. That is, $\tau_{\text{HI}} \cup \tau_{\text{LO}} = \tau$ and $\tau_{\text{HI}} \cap \tau_{\text{LO}} = \emptyset$ where τ_{HI} denotes the set of HI-tasks and τ_{LO} denotes the set of LO-tasks. A HI-task τ_i has two WCET estimates: one extremely pessimistic but safe one (*e.g.*, by static timing analysis and/or inflated by a safety-margin factor) denoted C_i^{H}, and a less pessimistic one (*e.g.*, by measurement) denoted C_i^{L}, where it is clear that $C_i^{\text{H}} \geq C_i^{\text{L}}$. By contrast, the WCET of a LO-task τ_k has only one (less-pessimistic) estimate denoted C_k^{L}. Please note, a HI-task (LO-task) job is also called a HI-job (LO-job, respectively) in the paper.

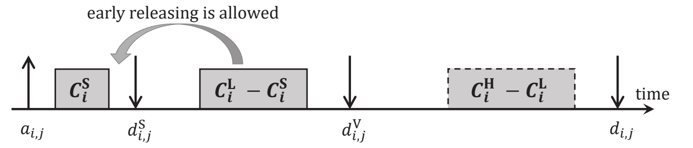


Fig. 9. An illustration for dividing an HI-job into sub-jobs in LO-mode.

Each HI-task τ_i has LO-*utilizations* ($u_i^{\text{L}} = C_i^{\text{L}}/T_i$) and HI-*utilizations* ($u_i^{\text{H}} = C_i^{\text{H}}/T_i$), while LO-tasks τ_k have only a LO-utilization ($u_k^{\text{L}} = C_k^{\text{L}}/T_k$). We also denote:

$$U_{\text{HI}}^{\text{L}} = \sum_{\tau_i \in \tau_{\text{HI}}} u_i^{\text{L}},\ U_{\text{HI}}^{\text{H}} = \sum_{\tau_i \in \tau_{\text{HI}}} u_i^{\text{H}},\text{ and } U_{\text{LO}}^{\text{L}} = \sum_{\tau_i \in \tau_{\text{LO}}} u_i^{\text{L}}.$$

Schedulability criteria. The MC sporadic task system τ is deemed MC-schedulable if and only if it is guaranteed that:

- all (HI- and LO-) jobs meet their deadlines if every job $\tau_{i,j}$ completes within C_i^{L} time units of execution; and,
- all HI-jobs meet their deadlines if every HI-job $\tau_{i,j}$ completes within C_i^{H} time units of execution.

Any HI-job $\tau_{i,j}$ having executed C_i^{H}, or any LO-job having executed C_i^{L}, but not completing is terminated immediately, or the system is considered erroneous.

Quarter-clairvoyance. So far, the above task model matches the traditional MC sporadic task model introduced in [5]. In light of the predicting coprocessor architecture presented in this paper, we introduce one more parameter, C_i^{S} (see Section II-B for the measurement), for each HI-task τ_i to model the certain clairvoyance our architecture brings.[6] Specifically, it is not necessary to wait until observing the behavior of a HI-job $\tau_{i,j}$ overrunning C_i^{L} to switch the system to HI-mode; once a HI-job $\tau_{i,j}$ has completed $C_i^{\text{S}} \leq C_i^{\text{L}}$ time units execution, our proposed architecture can predict[7] whether $\tau_{i,j}$ is able to complete within C_i^{L} time-unit accumulative execution or may need up to C_i^{H} time-unit accumulative execution to finish. That is, the scheduler may *foresee a future* HI-*job overrun and make the mode switch earlier to obtain better schedulability.* In addition, please note that in the special case where $C_i^{\text{S}} = C_i^{\text{L}}$ for every HI-task τ_i (*e.g.*, an I/O-independent task), quarter-clairvoyance MC scheduling reduces to traditional non-clairvoyance MC scheduling.

B. Algorithm EDF-VDSD

For the traditional scheduling of implicit-deadline sporadic tasks, EDF-VD [5] has been widely studied. Under EDF-VD, each HI-job is assigned a *virtual* deadline, which is earlier than its actual deadline. In LO-mode, both HI- and LO-tasks

[6] The "s" in C_i^{S} stands for triggering mode switch.

[7] We would also like to note that given the definition of C_i^{S}, the specific time instant at which a prediction can be made also depends on the specific scheduling algorithm that is applied. In contrast, in the semi-clairvoyance model [1], such prediction is always made at a job's release regardless of which scheduling algorithm is applied.

are scheduled by EDF according to the *virtual* deadlines of HI-jobs and actual deadlines of LO-jobs. On a mode switch to HI-mode, LO-tasks are dropped and HI-jobs are then scheduled by EDF according to their *actual* deadlines.

With quarter-clairvoyance MC tasks, we propose a new scheduling algorithm, called EDF-VDSD,[8] to improve schedulability by leveraging the clairvoyance obtained from the coprocessor.

Pre-runtime processing. Similar to EDF-VD, EDF-VDSD also calculates a relative virtual deadline for each HI-task τ_i using $D_i^{\text{V}} = x \cdot T_i$, where $x = \frac{U_{\text{HI}}^{\text{L}}}{1-U_{\text{LO}}^{\text{L}}}$. Furthermore, a relative *switching deadline*, D_i^{S}, for each HI-task is calculated using

$$D_i^{\text{S}} = \frac{C_i^{\text{S}}}{C_i^{\text{L}}} \cdot D_i^{\text{V}} \implies \frac{C_i^{\text{S}}}{D_i^{\text{S}}} = \frac{C_i^{\text{L}}}{D_i^{\text{V}}} = \frac{u_i^{\text{L}}}{x} \tag{2}$$

That is, each HI-job $\tau_{i,j}$ has a virtual deadline at $d_{i,j}^{\text{V}} = a_{i,j} + D_i^{\text{V}}$ and a switching deadline at $d_{i,j}^{\text{S}} = a_{i,j} + D_i^{\text{S}}$.

Run-time scheduling. During run-time, a deadline-based scheduling scheme is applied. In LO-mode, every LO-job is scheduled using its *actual* deadline as the priority, and every HI-job is considered as split into two sub-jobs. In particular, for every HI-job $\tau_{i,j}$, its first C_i^{S} time units execution is considered as the first sub-job and scheduled by the *switching* deadline $d_{i,j}^{\text{S}}$ as the priority; any execution beyond C_i^{S} time units up to C_i^{L} is considered as the second sub-job with a *pseudo-release* time at $d_{i,j}^{\text{S}}$ and a maximum execution of $C_i^{\text{L}} - C_i^{\text{S}}$ time units. The second sub-job is scheduled by the *virtual* deadline $d_{i,j}^{\text{V}}$ as the priority. Figure 9 illustrates sub-job splitting. Please note that during run-time, the second sub-job may be executed even *before* its pseudo-release, $d_{i,j}^{\text{S}}$ without jeopardizing any schedulability result, because *early released* sub-jobs have no impact on schedulability analysis under preemptive EDF scheduling, as long as their deadlines (and therefore, priorities) are not altered [2], [28].

A mode switch from LO-mode to HI-mode may happen at the moment when a HI-job $\tau_{i,j}$ has completed C_i^{S} time units of execution, *i.e.*, at the time instant when its first sub-job has completed. At that moment, it would be revealed to the scheduler whether $\tau_{i,j}$ needs to execute for more than C_i^{L} time units to complete, and therefore the scheduler decides whether a mode switch should be triggered. On a mode switch to HI-mode during run-time, all LO-jobs are immediately discarded, and all (pending and to-be-released) HI-jobs are henceforth scheduled by EDF according to their *actual* deadlines. That is, all *switching* and *virtual* deadlines are disregarded and do not have an effect in HI-mode.

C. Schedulability Test

We now analyze schedulability under EDF-VDSD and propose a schedulability test running in polynomial time.

Lemma 1. *Under EDF-VDSD, in* LO*-mode, all* LO*-jobs meet their* actual *deadlines, all* first sub-jobs *of* HI*-jobs meet their* switching *deadlines, and all* second sub-jobs *meet their* virtual *deadlines, if*

$$x \geq \frac{U_{\text{HI}}^{\text{L}}}{1 - U_{\text{LO}}^{\text{L}}}. \tag{3}$$

[8]EDF-VDSD stands for "earliest-deadline-first with virtual deadlines and switching deadlines."

Proof Sketch. First, we consider a *fluid* schedule, where each LO-task τ_i is continuously assigned an execution rate of u_i^{L} and each HI-task τ_k is continuously assigned an execution rate of u_k^{L}/x. It is clear that in this fluid schedule all LO-jobs meet their *actual* deadlines, all *first sub-jobs* of HI-jobs meet their *switching* deadlines, and all *second sub-jobs* meet their *virtual* deadlines. By viewing these LO-jobs, first sub-jobs, and second sub-jobs as just a set of "jobs" with each "job" having its "deadline" at their corresponding actual, switching, and virtual deadline in the three cases, all "jobs" meet their "deadlines." Furthermore, the total assigned rates are

$$\sum_{\tau_i \in \tau_{\text{HI}}} \frac{u_i^{\text{L}}}{x} + \sum_{\tau_i \in \tau_{\text{LO}}} u_i^{\text{L}} = \frac{U_{\text{HI}}^{\text{L}}}{x} + U_{\text{LO}}^{\text{L}} \overset{\{\text{by (3)}\}}{\leq} 1.$$

Therefore, this fluid schedule is feasible.

On the other hand, under EDF-VDSD, the "job set" of these LO-jobs, first sub-jobs, and second sub-jobs is scheduled exactly, following EDF, where their "deadline" is defined by their corresponding actual, switching, and virtual deadlines, respectively. Due to the optimality of EDF in preemptive uniprocessor scheduling, the existence of a feasible fluid schedule implies that EDF-VDSD also guarantees that all "deadlines" of the "jobs" are met. The lemma is as follows: □

Lemma 2. *Given that $x \geq \frac{U_{\text{HI}}^{\text{L}}}{1-U_{\text{LO}}^{\text{L}}}$, under EDF-VDSD, in the* HI*-mode, all* HI*-jobs meet their* actual *deadlines, if*

$$\sum_{\tau_i \in \tau_{\text{HI}}} \max\left\{ \frac{u_i^{\text{H}}}{1 - \frac{C_i^{\text{S}}}{C_i^{\text{L}}} \cdot x}, \frac{u_i^{\text{L}} - \frac{C_i^{\text{S}}}{T_i}}{1 - x} \right\} \leq 1. \tag{4}$$

Proof Sketch. Given that $x \geq \frac{U_{\text{HI}}^{\text{L}}}{1-U_{\text{LO}}^{\text{L}}}$, by Lemma 1, the switching deadline is the latest time instant for each HI-job to trigger a mode switch.

We consider the *density* (*i.e.*, the ratio of the remaining workload to the remaining time units until its deadline) of each carry-over (*i.e.*, released before t^* but has not completed by t^*) HI-job at the mode switch time instant t^*. Then, an arbitrary carry-over HI-job $\tau_{i,j}$ must be in one of the following two cases: **(i)** $t^* \leq d_{i,j}^{\text{S}}$ and **(ii)** $d_{i,j}^{\text{S}} < t^* \leq d_{i,j}^{\text{V}}$. Note that it cannot be the case that $t^* > d_{i,j}^{\text{V}}$, because in that case, either the mode switch would have been triggered by $\tau_{i,j}$ at $d_{i,j}^{\text{S}}$ earlier than t^* ($\tau_{i,j}$ executes for more than C_i^{L}) or $\tau_{i,j}$ would have been completed by $d_{i,j}^{\text{V}} < t^*$ ($\tau_{i,j}$ executes for at most C_i^{L}).

In case **(i)**, the density of $\tau_{i,j}$ is at most

$$\frac{C_i^{\text{H}}}{d_{i,j} - t^*} \leq \frac{C_i^{\text{H}}}{d_{i,j} - d_{i,j}^{\text{S}}} = \frac{C_i^{\text{H}}}{T_i - D_i^{\text{S}}} = \frac{u_i^{\text{H}}}{1 - \frac{D_i^{\text{S}}}{T_i}} = \frac{u_i^{\text{H}}}{1 - \frac{C_i^{\text{S}}}{C_i^{\text{L}}} \cdot x},$$

where the last equality is because of (2).

In case **(ii)**, $\tau_{i,j}$'s total execution time is at most C_i^{L}; otherwise, it would have triggered the mode switch earlier. In addition, it must have executed C_i^{S} time units by t^* which is after $d_{i,j}^{\text{S}}$. Therefore, the density of $\tau_{i,j}$ is at most

$$\frac{C_i^{\text{L}} - C_i^{\text{S}}}{d_{i,j} - t^*} \leq \frac{C_i^{\text{L}} - C_i^{\text{S}}}{d_{i,j} - d_{i,j}^{\text{V}}} = \frac{C_i^{\text{L}} - C_i^{\text{S}}}{T_i - D_i^{\text{V}}} = \frac{u_i^{\text{L}} - \frac{C_i^{\text{S}}}{T_i}}{1 - x}.$$

Thus, in a *fluid* schedule in HI-mode, if each HI-task τ_i is assigned a constant execution rate

$$f_i = \max\left\{\frac{u_i^{\mathrm{H}}}{1 - \frac{C_i^{\mathrm{S}}}{C_i^{\mathrm{L}}} \cdot x}, \frac{u_i^{\mathrm{L}} - \frac{C_i^{\mathrm{S}}}{T_i}}{1 - x}\right\},$$

then all deadlines in HI-mode must be met. Please note that all non-carry-over HI-jobs in HI-mode will also meet their deadlines due to

$$u_i^{\mathrm{H}} \leq \frac{u_i^{\mathrm{H}}}{1 - \frac{C_i^{\mathrm{S}}}{C_i^{\mathrm{L}}} \cdot x} \leq f_i.$$

That is, if $\sum_{\tau_i \in \tau_{\mathrm{HI}}} f_i \leq 1$, then the fluid schedule (starting from t^*) is feasible. Due to the optimality of EDF in preemptive uniprocessor scheduling, the existence of a feasible fluid schedule implies that EDF scheduling (by actual deadlines) the HI-tasks starting from t^*, which is exactly what EDF-VDSD does, also guarantees that all deadlines (of HI-tasks) are met in HI-mode. Thus, the lemma follows. □

Theorem 1. *The task system is MC-schedulable if*

$$\sum_{\tau_i \in \tau_{\mathrm{HI}}} \max\left\{\frac{u_i^{\mathrm{H}}}{1 - \frac{C_i^{\mathrm{S}}}{C_i^{\mathrm{L}}} \cdot \frac{U_{\mathrm{HI}}^{\mathrm{L}}}{1-U_{\mathrm{LO}}^{\mathrm{L}}}}, \frac{u_i^{\mathrm{L}} - \frac{C_i^{\mathrm{S}}}{T_i}}{1 - \frac{U_{\mathrm{HI}}^{\mathrm{L}}}{1-U_{\mathrm{LO}}^{\mathrm{L}}}}\right\} \leq 1. \quad (5)$$

Proof. Setting $x = \frac{U_{\mathrm{HI}}^{\mathrm{L}}}{1-U_{\mathrm{LO}}^{\mathrm{L}}}$ and by the above two lemmas, the theorem follows. It directly implies a sufficient schedulability test running in $\mathcal{O}(n)$ time, where n is the number of tasks. □

D. Discussions

We next discuss the benefits EDF-VDSD brings from an analytical perspective. Empirical studies and evaluation are presented in Section VI.

Comparison with non-clairvoyance EDF-VD. It it clear that the special case where $\forall i \in \tau_{\mathrm{HI}}, C_i^{\mathrm{S}} = C_i^{\mathrm{L}}$ reduces quarter-clairvoyance to the conventional non-clairvoyance MC scheduling model. By investigating this special case, we find our schedulability test dominates the first EDF-VD analysis in [6], which is also dominated by a later improved EDF-VD analysis in [5].

Unfortunately, our schedulability test does not have a strict dominance over the improved EDF-VD analysis in [5]. Nonetheless, the quarter-clairvoyance MC scheduling model and EDF-VDSD bring certain advantages over EDF-VD, even with the improved analysis in [5]. The following example is not deemed schedulable under EDF-VD, even with the analysis in [5], while it is deemed schedulable under EDF-VDSD by our analysis.

Example 1. *Consider a system with only two tasks τ_1 and τ_2, where τ_1 is a HI-task and τ_2 is a LO-task. For the HI-task τ_1, $T_1 = 10$, $C_1^{\mathrm{H}} = 8$, $C_1^{\mathrm{L}} = 3$, and $C_1^{\mathrm{S}} = 1$; for the LO-task τ_2, $T_2 = 10$, $C_2^{\mathrm{L}} = 5$. That is, in this system, $U_{\mathrm{HI}}^{\mathrm{H}} = 0.8$, $U_{\mathrm{HI}}^{\mathrm{L}} = 0.3$, $U_{\mathrm{LO}}^{\mathrm{L}} = 0.5$, $x = (0.3)/(1-0.5) = 0.6$, and $C_1^{\mathrm{S}}/C_1^{\mathrm{L}} = 1/3$.*

Under non-clairvoyant EDF-VD,

$$\frac{U_{\mathrm{HI}}^{\mathrm{L}}}{1 - U_{\mathrm{LO}}^{\mathrm{L}}} = \frac{0.3}{1 - 0.5} = 0.6 > 0.4 = \frac{1 - 0.8}{0.5} = \frac{1 - U_{\mathrm{HI}}^{\mathrm{H}}}{U_{\mathrm{LO}}^{\mathrm{L}}},$$

which means that even the improved EDF-VD schedulability test in [5] fails.

By contrast, under EDF-VDSD,

$$\frac{u_1^{\mathrm{H}}}{1 - \frac{C_1^{\mathrm{S}}}{C_1^{\mathrm{L}}} \cdot \frac{U_{\mathrm{HI}}^{\mathrm{L}}}{1-U_{\mathrm{LO}}^{\mathrm{L}}}} = \frac{0.8}{1 - \frac{1}{3} \times 0.6} = 1,$$

$$\text{and} \quad \frac{u_1^{\mathrm{L}} - \frac{C_1^{\mathrm{S}}}{T_1}}{1 - \frac{U_{\mathrm{HI}}^{\mathrm{L}}}{1-U_{\mathrm{LO}}^{\mathrm{L}}}} = \frac{0.3 - 0.1}{1 - 0.6} = 0.5.$$

Thus, by Theorem 1, this system is schedulable by EDF-VDSD.

An integrated algorithm EDF-VDSD+. Because schedulability can be determined offline by system parameters that are known prior to run-time, we can integrate algorithms EDF, EDF-VD, and EDF-VDSD to achieve even better schedulability. The resulting integrated algorithm, called EDF-VDSD+, is presented in Algorithm 4. Intuitively, by exploring the respective schedulability tests, EDF-VDSD+ will select the simplest of the three algorithms which can guarantee schedulability.

Algorithm 4: Pseudo-Code for EDF-VDSD+

```
1  if U_LO^L + U_HI^H <= 1 then
2  |  Apply ordinary EDF from the beginning (i.e., no MC
      and no mode switch at all), and declare SUCCESS;
3  else
4  |  if U_HI^L / (1 - U_LO^L) <= (1 - U_HI^H) / U_LO^L then
5  |  |  Apply EDF-VD, and declare SUCCESS;
6  |  else
7  |  |  if sum_{τ_i ∈ τ_HI} max{ u_i^H / (1 - (C_i^S/C_i^L) · U_HI^L/(1-U_LO^L)),
   |  |       (u_i^L - C_i^S/T_i) / (1 - U_HI^L/(1-U_LO^L)) } <= 1 then
8  |  |  |  Apply EDF-VDSD, and declare SUCCESS;
9  |  |  else
10 |  |  |  Declare FAILURE.
11 |  |  end
12 |  end
13 end
```

VI. Experimental Evaluation

We now conduct extensive experiments to evaluate the *Pythia*-MCS.

Experimental Platform. We built the *Pythia*-MCS on a Xilinx VC709 evaluation board. Specifically, the *Pythia*-coprocessor was implemented using BlueSpec System Verilog [54] and connected to a 5×5 mesh type open-source NoC [46]. As well as the *Pythia*-coprocessor, the NoC also contained 16 MicroBlaze processors [55], memory and I/O peripherals. The software executing on the MicroBlaze processors (OS kernels and user applications) was compiled using a Xilinx MicroBlaze GNU tool-chain [55]. We selected FreeRTOS (v.9.0.0) as the OS kernel for all processors, with the modifications introduced in Section III-D. Note that the *Pythia*-coprocessor was implemented using the two methods described in Section IV-B, hardware/software co-design (denoted *PY*|*hs*) and hardware-only design (denoted *PY*|*hw*). To enable comparison, we also built a conventional MCS framework (reviewed in

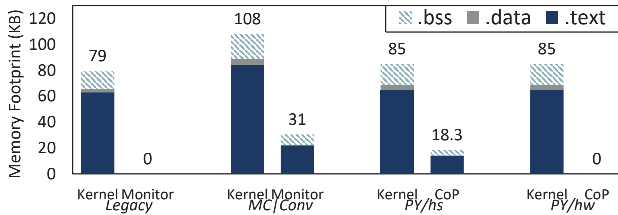


Fig. 10. Run-time Software Overhead (CoP: *Pythia*-coprocessor). The software overhead is evaluated via memory footprint (unit: KB), containing segments of BSS, data and text.

TABLE I
HARDWARE OVERHEAD (IMPLEMENTED ON FPGA)

	LUTs	Registers	DSP	BRAM (KB)	Power (mW)
MB-B	854	529	0	16	127
MB-F	4908	4385	6	128	258
CAN	711	604	0	0	5
SPI	632	427	0	0	4
PY|*hw*	587	396	0	16	109
PY|*hs*	973	583	0	16	133

Section III-C) on a similar hardware architecture (denoted *MC*|*conv*), without the *Pythia*-coprocessor. The *MC*|*conv* system architecture is illustrated in the upper part of Figure 3. All architectures ran at 100 MHz.

A. Software Overhead

In this section, we compare the software overheads of the legacy system,[9] with *MC*|*conv*, *PY*|*hs* and *PY*|*hw*.

Experimental Setup. The software overhead was evaluated using the run-time memory footprint [48], with specific consideration of the OS kernel and execution monitor (memory size tool: Xilinx MicroBlaze GNU tool-chain [55]). The legacy OS kernel was fully-featured with essential I/O drivers [21].

Obs.1. An additional software overhead was sustained by the conventional MCS framework compared to the legacy system. This is effectively reduced in *Pythia*-MCSs.

This observation is shown in Figure 10. In *MC*|*conv*, the introduction of an execution monitor and the modifications to the OS kernel bring an additional 60 KB (75.9%) memory footprint compared to the legacy system. By contrast, in both *PY*|*hs* and *PY*|*hw*, run-time monitoring and mode switch triggers rely on the *Pythia*-coprocessor. Hence, the implementation of the execution monitor was not required. The removal of the execution monitor significantly reduced the run-time memory footprint to 85 KB, which is slightly higher than the memory footprint in the legacy system (7.6% extra). Please note, *PY*|*hs* requires an 18.3 KB memory footprint for the software execution on the coprocessor, which is not counted in the software overheads of the main CPU(s).

B. Hardware Overhead

Pythia-MCS requires additional hardware implementation for the coprocessor. Hence, in this section, we evaluate the hardware overhead of the *Pythia*-coprocessor.

Experimental Setup. We first configured the *Pythia*-coprocessor to monitor two I/Os (with two IMUs) and then evaluated the coprocessor and both the basic and full-featured MicroBlaze processors (MB-B and MB-F), as well as two mainstream I/O controllers (SPI and CAN). All components were synthesized and implemented by Vivado (v2019.2) [56] and compared using Look Up Tables (LUTs), registers, DSPs, Block RAMs (BRAMs) and power consumption [44].

Obs.2. The design of the *Pythia*-coprocessor was resource-efficient compared to the generic CPUs. Its hardware consumption was similar to commonly used I/O controllers.

As shown in Table I, *PY*|*hw* required significantly less hardware than either MB-F (12.0% LUTs, 9.0% registers, 42.2% power consumption) or MB-B (68.7% LUTs, 74.9% registers, 85.8% power consumption). Due to the integration of a mature processor, *PY*|*hs* consumed slightly more hardware than *PY*|*hw* (165.8% LUTs, 147.2% registers, 122.0% power consumption). When compared to the CAN and SPI controllers, both *PY*|*hs* and *PY*|*hw* had similar consumption of both LUTs and registers, but additional memory consumption. The memory consumed additional power for refresh [25]; hence, both *PY*|*hs* and *PY*|*hw* consumed more than 20 times the power of the I/O controllers.

C. Automotive Case Study

We now use an automotive case study to examine the benefits of the *Pythia*-MCS over a conventional MCS framework.

Systems Configuration. To analyze the benefits brought by the *Pythia*-MCS, *MC*|*conv* and *PY*|*hs* were examined. We configured *PY*|*hs* as *PY*|*hs*-40/70/100, which enabled 40%/70%/100% of I/O-related tasks using *I/O-driven mode switch*. In other words, *PY*|*hs*-x indicated the system was x% of *Pythia*-MCS.

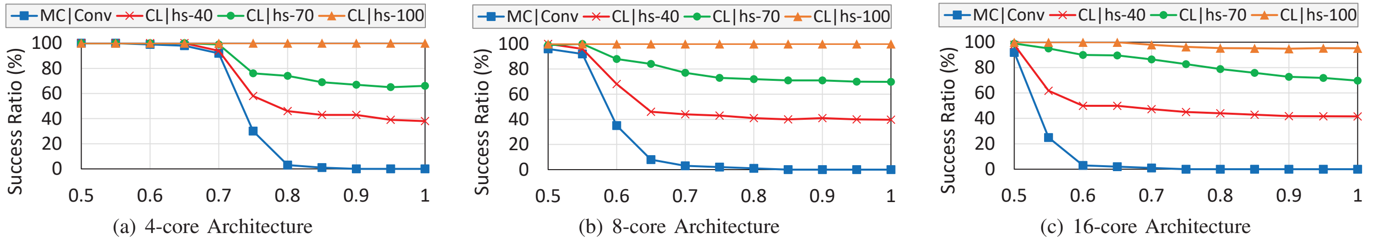
Task sets. We introduced two sets of I/O-related tasks:

- 20 HI-tasks, selected from Renesas functional safety automotive use case database [18], (*e.g.*, CRC and RSA32).
- 20 LO-tasks, selected from the EEMBC automotive benchmark [19], (*e.g.*, Fast Fourier Transform (FFT) and road speed calculation).

The HI-tasks had been certified as ASIL-D tasks [27], with analyzed WCETs (C_i^{HI}). Additionally, we employed a hybrid-measurement approach [36] to obtain measured WCETs for all tasks (C_i^{LO}). The raw data for processing by the 40 tasks was randomly generated off-chip and sent to the evaluated systems via two Ethernet controllers (10 Gbps) at run-time. The HI-tasks experimental measurements (C_i^{S} and Υ_i^{L}) were obtained using the method described in Section II. The *MC*|*conv* also contained a simulated HI-task for the execution monitor (described in Section III-B), which was not required by *PY*|*hs*. Each task had a defined period, with overall system utilization in both LO- and HI-mode approximately 50%. Following Section V, we adopted implicit deadlines for all tasks.

Synthetic workloads. We also introduced synthetic workloads (in the LO-task category), which can be optionally added into the system to control overall utilization in the experiments. Like other LO-tasks, the synthetic workloads were also selected from the EMBC automotive benchmark [19], but only contained I/O-independent tasks, without data input. Notably, in practice, the execution time of a task is affected by diverse

[9] A naive system, which does not support any MCS features.



(a) 4-core Architecture (b) 8-core Architecture (c) 16-core Architecture

Fig. 11. Success Ratios of the Conventional MCS and *Pythia*-MCS in Automotive Case Study. (The x-axis denotes the target utilization)

factors (*e.g.*, cache miss rate); hence, adding synthetic workloads to a system only gives the system a *target utilization*, which may be different from the actual system utilization.

Experimental Setup. We introduced three groups of experimental setups, which activate $4/8/16$ processors to execute the experimental task sets and synthetic workloads. In each experimental group, we executed the examined systems 500 times under varying target utilization from 50% to 100% (at intervals of 5% increases). Each execution lasted 100 seconds, which guaranteed all tasks could execute at least 250 times. For fair comparison, we also ensured the data input to the examined systems was identical in each execution.

We evaluated the examined systems using *success ratio*, which records the percentage of an examined trial executed successfully (*i.e.*, without deadline miss of any HI-task), under a specified target utilization. Note that, we ignored the deadline miss of LO-tasks after mode switch. Figure 11 shows the experiment results. According to the results, we extend the following observations:

Obs.3. Introducing *I/O-driven mode switch* is beneficial.

This observation is supported by the results in Figures 11(a), 11(b) and 11(c). As shown, with the same configuration, the *Pythia*-MCSs always outperform the conventional MCS. Moreover, we also obverse that a full *Pythia*-MCS ($PY|hs$-100) consistently outperformed the partial *Pythia*-MCSs ($PY|hs$-70 and $PY|hs$-40). This means that having a higher percentage of the system involving *I/O-driven mode switch* introduces more benefits.

Obs.4. Increasing the number of processors significantly reduced the success ratio of the conventional MCS framework.

This observation is shown in the comparison between Figures 11(a) and 11(c). In a 4-core $MC|conv$, a significant drop in the success ratio occurred at 75% of target utilization, whereas this drop moved to 55% of target utilization in a 16-core $MC|conv$. This observation mainly results from the additional on-chip interfaces and resource contention generated by the introduced processors and tasks.

Obs.5. The *Pythia*-MCS, maintains high success ratios when the number of processors increases, effectively eliminating the issues in *Obs.4.*

In the *Pythia*-MCS, run-time monitoring is placed close to I/O devices; hence, the *Pythia*-coprocessor can detect abnormal behaviors due to large amounts of data generation promptly, and trigger a mode switch. In a 16-core system (Figure 11(c)), when target utilization approaches 100%, $PY|hs$-100 maintains a success ratio which is still close to 100%. This observation demonstrates the benefits and applicability of introducing the *Pythia*-MCS in multi-many-core architectures.

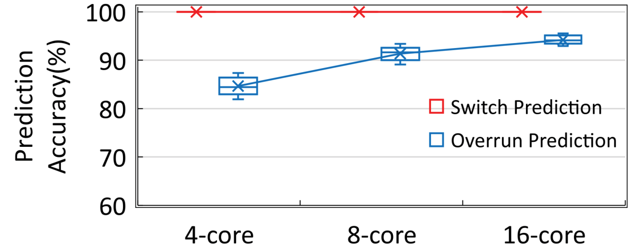


Fig. 12. Prediction Accuracy of *Pythia*-MCS.

D. Accuracy of Prediction

Although Section VI-C demonstrates the benefits brought by I/O-driven mode switch in the *Pythia*-MCS, we acknowledge that the *accuracy* of the prediction mechanism finally determines the feasibility of the proposed design. We now examine the accuracy of the prediction mechanism considering two scenarios:

Scenario I. The *Pythia*-MCS misses a required mode switch. This scenario causes safety hazards, since the LO-tasks cannot be terminated in time.

Scenario II. The *Pythia*-MCS triggers a mode switch when it is not necessary. This scenario leads to system performance loss, since LO-tasks are terminated unexpectedly.

Experimental Setup. We adopted the same experimental setup and methods introduced in Section VI-C with $MC|conv$ and $PY|hs$ ($PY|hs$-100) being executed. Prediction accuracy was calculated using two measures. Firstly, for all executing cases where $MC|conv$ triggers mode switch, *accuracy of switch prediction* calculates the percentage of executing cases where $PY|hs$ also triggers the switch. Secondly, for all executing cases where $PY|hs$ triggers mode switch, *accuracy of overrun prediction* calculates the percentage of executing cases where $MC|conv$ also triggers the switch. From the results (Figure 12), we observe:

Obs.6. The prediction mechanism does not introduce additional safety concerns in *Pythia*-MCS, as the system never missed a required mode switch.

As shown in Figure 12, the accuracy of switch prediction was constant at 100% without experimental variance. This means that in all cases where $MC|conv$ triggered a mode switch, $PY|hs$ also triggered the mode switch. Therefore,

Pythia-MCS successfully avoids Scenario I. This observation benefited from the conservative selection of TH-I/O for each HI-task introduced in Section II-B.

Obs.7. The prediction mechanism leads to a certain level of system performance loss, as the *Pythia*-MCS may pessimistically trigger a mode switch when it is not required.

As shown in Figure 12, in a 4-core $PY|hs$, the accuracy of overrun prediction averaged around 85%, which means the *Pythia*-MCS has about 15% probability of triggering an unrequired mode switch. Therefore, *Pythia*-MCS does not completely avoid Scenario II.

Fortunately, with an increasing number of processors, this weakness can be effectively alleviated. As shown, $PY|hs$ raises the accuracy of overrun prediction to around 91% for the 8-core system and 94% for the 16-core.

An explanation for this observation may be that although the *Pythia*-MCS cannot provide 100% accuracy of overrun prediction for every single task, the increasing number of tasks from the introduced processors raises the likelihood that more than one task triggers a mode switch simultaneously (and at least one actually overruns C_i^L execution), which effectively mitigates the prediction gap from the perspective of the entire system. With this observation, we conjecture that the accuracy of overrun prediction in *Pythia*-MCS would approximate to 100% with an increasing number of processors.

VII. Conclusion

In this paper, a novel MCS framework (*Pythia*-MCS), which simultaneously supports run-time I/O monitoring and I/O-driven mode switch, is proposed. With the introduced features, the *Pythia*-MCS achieves *future-prediction*, being able to foresee the over-execution of a task and triggering a timely mode switch. The *Pythia*-MCS system architecture and two options of design methods are detailed. A new theoretical model (quarter-clairvoyance) and schedulability analysis are also presented to provide a timing guarantee for the *Pythia*-MCS and demonstrate improved schedulability compared to conventional MCS frameworks. As shown in the evaluation, the *Pythia*-MCS outperforms state-of-the-art MCS frameworks with varying hardware architectures. In addition, the *Pythia*-MCS is resource-efficient.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive and helpful feedback. This work is supported by the U.S. National Science Foundation under Grant Nos. CNS-1618185 and IIS-1724227, start-up and REP grants from Texas State University and a start-up grant from Wayne State University. The authors would also like to thank Dr. Xiaotian Dai, Prof. Ian Gray and Prof. Qingling Zhao, for their consultancy and discussion during the system development.

References

[1] K. Agrawal, S. Baruah, and A. Burns. Semi-clairvoyance in mixed-criticality scheduling. In *40th IEEE Real-Time Systems Symposium (RTSS 2019)*. York, 2019.

[2] J. H. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, 2004.

[3] ARM. Amba axi and ace protocol specification, 2012.

[4] S. Baruah, V. Bonifaci, G. d'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, 2011.

[5] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 145–154. IEEE, 2012.

[6] S. K. Baruah, V. Bonifaci, G. d'Angelo, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. Mixed-criticality scheduling of sporadic task systems. In *European Symposium on Algorithms*, pages 555–566. Springer, 2011.

[7] A. Bhuiyan, S. Sruti, Z. Guo, and K. Yang. Precise scheduling of mixed-criticality tasks by varying processor speed. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, pages 123–132, 2019.

[8] A. Burns and R. Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, pages 1–69, 2013.

[9] A. Burns and A. J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[10] A. Burns and A. J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[11] J. Caplan, Z. Al-Bayati, H. Zeng, and B. H. Meyer. Mapping and scheduling mixed-criticality systems with on-demand redundancy. *IEEE Transactions on Computers*, 67(4):582–588, 2017.

[12] J.-J. Chen, W.-H. Huang, Z. Dong, and C. Liu. Fixed-priority scheduling of mixed soft and hare real-time tasks on multiprocessors. In *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2017.

[13] D. De Niz, B. Andersson, M. Klein, J. Lehoczky, A. Vasudevan, H. Kim, and G. Moreno. Mixed-trust computing for real-time systems. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–11. IEEE, 2019.

[14] P. Dong, A. Burns, Z. Jiang, and X. Liao. Tzdks: A new trustzone-based dual-criticality system with balanced performance. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 59–64. IEEE, 2018.

[15] P. Dong, Z. Jiang, A. Burns, Y. Ding, and J. Ma. Build real-time communication for hybrid dual-os system. *Journal of Systems Architecture*, page 101774, 2020.

[16] Z. Dong and C. Liu. An efficient utilization-based test for scheduling hard real-time sporadic dag task systems on multiprocessors. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 181–193. IEEE, 2019.

[17] A. Easwaran. Demand-based scheduling of mixed-criticality sporadic tasks on one processor. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 78–87. IEEE, 2013.

[18] R. Electronics. Renesas: Automotive Use Cases. https://www.renesas.com/eu/en/solutions/automotive/technology/safety.html. Accessed May 5, 2020.

[19] EMBC. EMBC benchmark. https://www.eembc.org/autobench/.

[20] R. Ernst and M. Di Natale. Mixed criticality systems—a history of misconceptions? *IEEE Design & Test*, 33(5):65–74, 2016.

[21] FreeRTOS. Freertos official website. http://www.freertos.org/. Accessed September 27, 2017.

[22] P. K. Gadepalli, G. Peach, G. Parmer, J. Espy, and Z. Day. Chaos: a system for criticality-aware, multi-core coordination. In *RTAS*, 2019.

[23] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 13–23. IEEE, 2011.

[24] M. Hassan and H. Patel. Criticality-and requirement-aware bus arbitration for multi-core mixed criticality systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11. IEEE, 2016.

[25] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[26] K. Hwang and A. Faye. Computer architecture and parallel processing. 1984.

[27] I. ISO. 26262: Road vehicles-functional safety. *International Standard ISO/FDIS*, 26262, 2018.

[28] K. Jeffay and S. Goddard. A theory of rate-based execution. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No. 99CB37054)*, pages 304–314. IEEE, 1999.

[29] Z. Jiang, N. Audsley, and P. Dong. Blueio: A scalable real-time hardware i/o virtualization system for many-core embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(3):1–25, 2019.

[30] Z. Jiang, N. Audsley, P. Dong, N. Guan, X. Dai, and L. Wei. Mcs-iov: Real-time i/o virtualization for mixed-criticality systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 326–338. IEEE, 2019.

[31] Z. Jiang and N. C. Audsley. Gpiocp: Timing-accurate general purpose i/o controller for many-core real-time systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 806–811. IEEE, 2017.

[32] Z. Jiang, N. C. Audsley, and P. Dong. Bluevisor: A scalable real-time hardware hypervisor for many-core embedded systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 75–84. IEEE, 2018.

[33] N. Kim, S. Tang, N. Otterness, J. H. Anderson, F. D. Smith, and D. E. Porter. Supporting i/o and ipc via fine-grained os isolation for mixed-criticality real-time tasks. In *RTNS*, 2019.

[34] A. Kritikakou, C. Pagetti, O. Baldellon, M. Roy, and C. Rochange. Run-time control to increase task parallelism in mixed-critical systems. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 119–128. IEEE, 2014.

[35] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez. Distributed run-time wcet controller for concurrent critical tasks in mixed-critical systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, pages 139–148, 2014.

[36] S. Law, M. Bennett, S. Hutchesson, I. Ellis, G. Bernat, A. Colin, and A. Coombes. Effective worst-case execution time analysis of do178c level a software. *Ada User Journal*, 36(3), 2015.

[37] J. Lee, H. S. Chwa, L. T. Phan, I. Shin, and I. Lee. Mc-adapt: Adaptive task dropping in mixed-criticality scheduling. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–21, 2017.

[38] B. Lesage, I. Puaut, and A. Seznec. Preti: Partitioned real-time shared cache for mixed-criticality real-time systems. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, pages 171–180, 2012.

[39] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, et al. Towards fully autonomous driving: Systems and algorithms. In *2011 IEEE Intelligent Vehicles Symposium (IV)*, pages 163–168. IEEE, 2011.

[40] J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, and C. Lu. Mixed-criticality federated scheduling for parallel real-time tasks. *Real-time systems*, 53(5):760–811, 2017.

[41] Y. Li, M. Danish, and R. West. Quest-v: A virtualized multikernel for high-confidence systems. *arXiv preprint arXiv:1112.5136*, 2011.

[42] D. Liu, J. Spasic, N. Guan, G. Chen, S. Liu, T. Stefanov, and W. Yi. Edf-vd scheduling of mixed-criticality systems with degraded quality guarantees. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 35–46. IEEE, 2016.

[43] M. M. Mano. *Computer system architecture*. Prentice-Hall of India, 2003.

[44] E. Monmasson and M. N. Cirstea. Fpga design methodology for industrial control systems—a review. *IEEE transactions on industrial electronics*, 54(4):1824–1842, 2007.

[45] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares. Virtualization on trustzone-enabled microcontrollers? voilà! In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304. IEEE, 2019.

[46] G. Plumbridge, J. Whitham, and N. Audsley. Blueshell: a platform for rapid prototyping of multiprocessor nocs and accelerators. *ACM SIGARCH Computer Architecture News*, 41(5):107–117, 2014.

[47] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo. Is your bus arbiter really fair? restoring fairness in axi interconnects for fpga socs. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–22, 2019.

[48] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system principles*. John Wiley & Sons, 2006.

[49] J. Singh, L. Santinelli, F. Reghenzani, K. Bletsas, and Z. Guo. Non-preemptive scheduling of periodic mixed-criticality real-time systems. In *10th European Congress on Embedded Real-Time Systems*, 2020.

[50] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*, 2007.

[51] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, and V. I. U. level Isa. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2, 2014.

[52] R. West, Y. Li, E. Missimer, and M. Danish. A virtualized separation kernel for mixed-criticality systems. *ACM Transactions on Computer Systems (TOCS)*, 34(3):1–41, 2016.

[53] S. Xi, J. Wilson, C. Lu, and C. Gill. Rt-xen: Towards real-time hypervisor scheduling in xen. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 39–48. IEEE, 2011.

[54] Xilinx. Bluespec System Verilog. https://bluespec.com.

[55] Xilinx. Microblaze. https://www.xilinx.com/products/design-tools/microblaze.

[56] Xilinx. Xilinx official website. https://www.Xilinx.com.

[57] D. Yang, X. Li, X. Dai, R. Zhang, L. Qi, W. Zhang, and Z. Jiang. All in one network for driver attention monitoring. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2258–2262. IEEE, 2020.

[58] K. Yang and Z. Dong. Mixed-criticality scheduling with varying processor supply in compositional real-time systems. In *Proceedings of the 7th International Workshop on Mixed Criticality Systems (WMC)*, 2019.

[59] Y. Zhao and H. Zeng. An efficient schedulability analysis for optimizing systems with adaptive mixed-criticality scheduling. *Real-Time Systems*, 53(4):467–525, 2017.