Journal of Systems Architecture 130 (2022) 102683

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Towards an energy-efficient quarter-clairvoyant mixed-criticality system

Zhe Jiang [a], Kecheng Yang [b], Nathan Fisher [c], Neil Audsley [a], Zheng Dong [c,*]

[a] University of York, United Kingdom
[b] Texas State University, USA
[c] Wayne State University, USA

ARTICLE INFO

Keywords:
Real-time systems
Mixed-criticality system
Energy-efficient
Hardware–software co-design

ABSTRACT

Mode switch is the key strategy in mixed-criticality systems, enabling a dynamic balance between system performance and safety. Mode switch in conventional MCS frameworks is always triggered by *over-execution* of a task, *i.e.*, a task overrunning the less pessimistic worst-case execution time. In cyber–physical systems, the data volume generated by I/O affects and even dominates task execution time. Based on this observation, we propose a novel MCS framework, named *Pythia*-MCS, which predicts task execution time according to I/O run-time behaviors. With the new feature of *future-prediction*, *Pythia*-MCS provides more timely, but still accurate, mode switches. We specifically introduce the *Pythia*-MCS design methods, including different allocations of I/O monitoring and an efficient energy management framework. We present a new theoretical model (*quarter-clairvoyance*), which guarantees the timing predictability of the design, and a new schedulability analysis for *Pythia*-MCS, which demonstrates improved schedulability compared to conventional MCS frameworks. In addition, *Pythia*-MCS is comprehensively evaluated using a number of metrics.

1. Introduction

In modern safety-critical systems, it is increasingly important to integrate components with different levels of criticalities (*e.g.*, Automotive Safety and Integrity Levels (*ASILs*) in ISO26262 [1]), onto a shared hardware platform driven by the diverse functionalities required by modern safety-critical systems (*e.g.*, automated driving [1]) and the rapid evolution of underlying platforms [2]. Such systems are called *Mixed-Criticality Systems (MCS)s* [3].

A widely studied theoretical model for dual-criticality MCSs assumes that the Worst-Case Execution Time (WCET) of a task is estimated with different levels of confidence [3,4].[1] 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). 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 a LO-WCET [1,3]. 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 should complete execution by their deadlines [5]. To satisfy this criterion, *mode switch* is often used [3,6]. That is, a system initializes from 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 [7,8].

Many previous frameworks are based on this theoretical model, *e.g.*, Richard et al. [9], Gadepalli et al. [10], and Kim et al. [11]. These frameworks, which trigger a mode switch when they detect over-execution of a HI-task, are also called "*non-clairvoyant MCSs*" [12,13]. Different from non-clairvoyant MCSs, Baruah et al. [12] and Agrawal et al. [13] introduce *clairvoyant* and *semi-clairvoyant* MCS theoretical models, which assume that whether a HI-task will overrun its LO-WCET is known before or at release [14,15]. As shown in [12,13], both *clairvoyant* and *semi-clairvoyant* MCSs outperform non-clairvoyant MCSs. However, it is challenging to build a practical MCS framework with a degree of clairvoyance, since most run-time situations must be known beforehand, and for example, it is difficult to predict an external environmental change before it happens [12].

Inputs/Outputs (I/Os) are important in MCSs, as the volume of input data may significantly affect the execution time of a task, determining the necessity of a mode switch. Taking an autonomous vehicle as an

* Corresponding author.
E-mail address: dong@wayne.edu (Z. Dong).

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

https://doi.org/10.1016/j.sysarc.2022.102683
Received 22 January 2022; Received in revised form 3 June 2022; Accepted 25 July 2022
Available online 30 July 2022

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)[2] 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
```

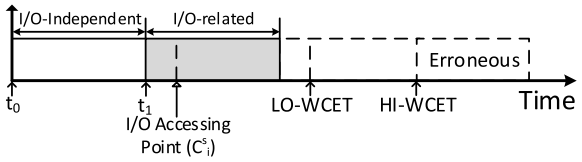


Fig. 1. Ethernet control task timing chart.

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 [1]. Therefore, a mode switch may be triggered due to more computation time being required by a task to process the additional received data. Based on these observations, we propose a novel MCS framework architecture (***Pythia-MCS*** [16]) which can acquire a certain level of clairvoyance. Based on the previous work [16], we extend the design with the following key features:

- continuously monitors and analyses I/O behaviors. The system can trigger a mode switch when a large amount of data is generated by an I/O.
- contains two optional allocations for I/O monitoring, at *interconnects* or *pins*, providing a trade-off between design compatibility and monitoring timeliness.
- integrates an energy management framework, which mitigates the power consumption introduced by I/O-driven mode switch.

Correspondingly, we also present

- Comprehensive experiments to evaluate *Pythia*-MCS in terms of overhead, power consumption and scalability.
- A real-world case study to examine benefits and prediction accuracy of *Pythia*-MCS over a conventional MCS.

The rest of this paper is organized as follows: Section 2 presents the concepts of I/O-driven MCS. Sections 3 and 4 give the system architecture and design methods of *Pythia*-MCS, followed by schedulability analysis given in 5. Section 6 evaluates *Pythia*-MCS, and Section 7 concludes.

[2] The busy-waiting is monitored by a timeout monitoring in the system, in order to avoid endless waiting.

2. Preliminary: I/O-driven MCS

We first study relationships between I/Os and task execution time, then explain concepts of I/O-driven MCSs.

2.1. I/Os and task execution time

Based on the usage of I/Os, a task can be decomposed into:

I/O-independent computation — pure software calculation without I/O access. Computation time usually depends on system micro-architectures, *e.g.*, CPU architecture.

I/O-related computation — I/O accesses and I/O-bounded calculation. Computation time is usually determined by the data volume generated by the I/Os [17].

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 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. Fig. 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, Fig. 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 even dominate, task execution time.

Therefore, we can predict execution time of an I/O-related HI-task at its I/O access point and determine 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***.

2.2. 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:

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 Y_i^L). At C_i^S, if the data volume accumulated by the task (denoted v_i) exceeds its TH-I/O (*i.e.*, $v_i > Y_i^L$), 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 5), 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 Y_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

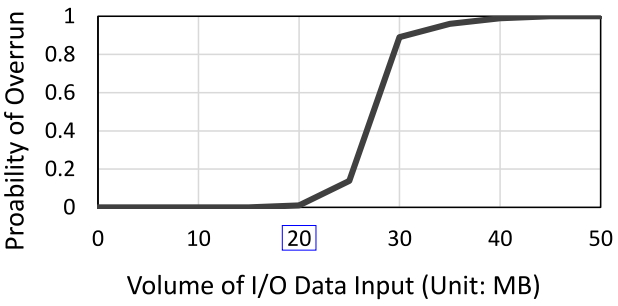


Fig. 2. Find TH-I/O for Ethernet control task.

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 [19] with the configurations introduced in Section 6) are shown in Fig. 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.

3. *Pythia*-MCS architecture

3.1. Context

In this paper, we assume: (i) The platform is an embedded Network-on-Chip (NoC). Although *Pythia*-MCS is agnostic to the types of bus, deployment of NoC can enhance the predictability of on-chip transactions [20]. (ii) *Pythia*-MCS is applicable to both single- and many-core architectures. A *fully-partitioned* scheme is adopted in a multi-/many-core *Pythia*-MCS. That is, tasks are *statically* assigned to a given processor. Existing task allocation heuristic [2] (*e.g.*, first-fit) can be applied directly for partitioning. (iii) A task can access one I/O at most, whereas an I/O can be accessed by multiple tasks.

3.2. Design concepts

We now present three design concepts for *Pythia*-MCS:

Design Concept 1: online I/O monitoring. *Pythia*-MCS introduces a coprocessor, which monitors and analyzes run-time data generated from I/Os. *Pythia*-MCS presents two options for allocating I/O monitoring: at the system interconnects (*i.e.*, NoC) or at the I/O pins. This allows the option of design compatibility or timeliness of I/O monitoring.

Design Concept 2: adaptive mode switch. *Pythia*-MCS supports both I/O-driven and conventional mode switches. In practice, tasks may only contain I/O-independent computations (*i.e.*, be I/O-independent tasks); hence, *Pythia*-MCS can execute all types of task.

Design Concept 3: efficient energy management. *Pythia*-MCS contains an energy management framework which can switch off the clocks/power of particular parts when they are not being used. Energy management effectively mitigates power consumption generated by I/O-driven mode switches.

In the context of conventional non-clairvoyant MCS theory, a number of practical frameworks have been proposed, *e.g.*, [9–11]. To ensure compatibility with the state-of-the-art, the proposed system architecture for *Pythia*-MCS is derived from conventional MCS frameworks. Therefore, Section 3.3 first reviews a conventional MCS architecture, Section 3.4 then presents the system architecture.

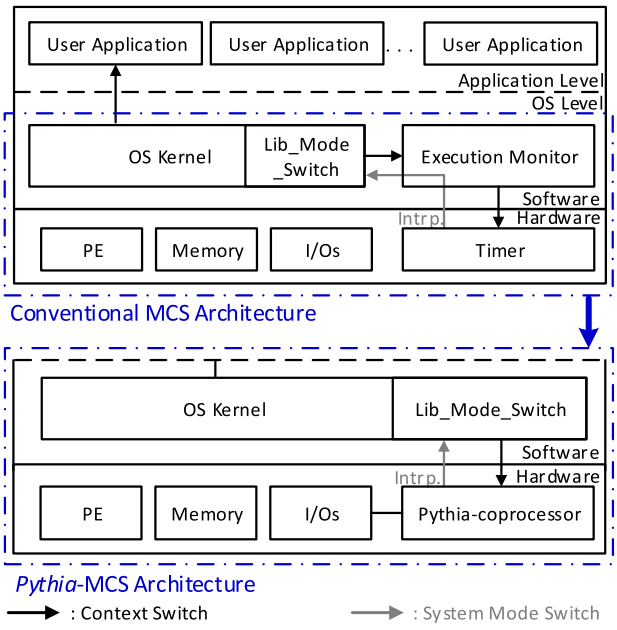


Fig. 3. System architectures of conventional MCS and *Pythia*-MCS.

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
```

3.3. Conventional MCS system architecture

The generalized architecture of a conventional MCS (shown in the upper part of Fig. 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: (i) monitoring task execution time; and, (ii) triggering a mode switch when the over-execution of a HI-task is detected. 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. [11] integrate the execution monitor with the OS kernel, while Li et al. [21] implements 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.

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
```

3.4. 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 Fig. 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 4.

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 6.1.

Run-time behaviors. At system initialization, I/O-related HI-tasks' TH-I/Os and I/O-independent HI-tasks' LO-WCETs are preloaded to the coprocessor. During context switches, the OS kernel synchronizes the ID of the scheduled task with the coprocessor (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 3.

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 Fig. 3). Therefore, user applications designed for a conventional MCS can be mapped to the *Pythia*-MCS directly.

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

4. *Pythia*-coprocessor

Fig. 4 illustrates the typical use of the *Pythia*-coprocessor in a NoC-based many-core architecture: the coprocessor connects a router/arbiter and I/Os, which enables on-chip communication and run-time I/O monitoring, respectively. In Fig. 5, we introduce the design of the coprocessor, which comprises three main modules:

I/O Monitor Unit (*IMU*) — observes the run-time status of the connected I/O, decomposes the I/O data packets, and reports the volume to the Mode Switch Unit (MSU).

Mode Switch Unit (*MSU*) — receives messages from the IMU, and then checks whether mode switch is necessary.

Energy Management Unit (*EMU*) — synchronizes with the MSU, determining the current system status, and switches off power/clocks of unused I/Os and related parts.

A complete I/O access path usually involves different system components, *e.g.*, OS kernel, interconnects, I/O controllers and I/O pins [22], which allows I/O monitoring to be placed at any system level. As described in Design Concept 1, I/O monitoring in *Pythia*-MCS can be allocated either to the routers or the I/O pins. Placing the monitoring at the I/O pins, *i.e.*, the boundary of a system, achieves the most timeliness, but involves a more complicated design. This is because the method needs to decompose and analyze the I/O packets for different serial communication protocols. Conversely, allocating the monitoring at the routers, only requires understanding of an on-chip communication protocol. This brings compatibility to the design, but loses some of the monitoring timeliness. To support these two types of I/O monitoring, we propose two IMU variants: IMU at Routers (IMU_R) and IMU at Pins (IMU_P).

4.1. I/O Monitor Unit at Routers (IMU_R)

I/O data decomposition. Monitoring and decomposing I/O data packets at the routers requires clear understanding of the protocol specifications for on-chip communication. In this paper, we explain the I/O data decomposition using the example of AMBA AXI [23], since it is the most commonly used protocol for on-chip communications in embedded architectures [23] and is also used in our experimental platform.

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.[2] The data volume of this transaction (denoted as v^*) is calculated in Eq. (1).

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

As shown in Fig. 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_R design. The design of the IMU_R is shown in Fig. 7, which contains the main components of a run-time sampler, an access interface and memory banks.

The memory banks store the volume of unprocessed data for each task (*i.e.*, v_i). The memory address reserved for v_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.*, v^*). The sampler then adds v^* to the volume of unprocessed data for τ_d (*i.e.*, $v_d = v_d + v^*$) 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*.

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

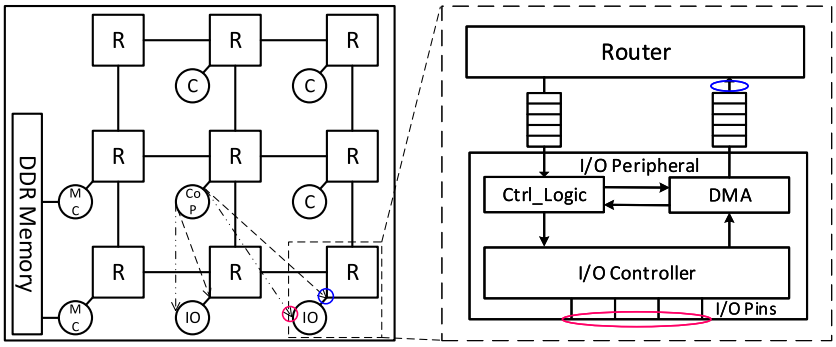


Fig. 4. *Pythia*-coprocessor in a NoC system (C: Processor core; Cop: *Pythia*-coprocessor; R: Router/Arbiter; MC: Memory controller). The blue and pink circles indicate two locations of I/O monitoring: at a router or I/O pins. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

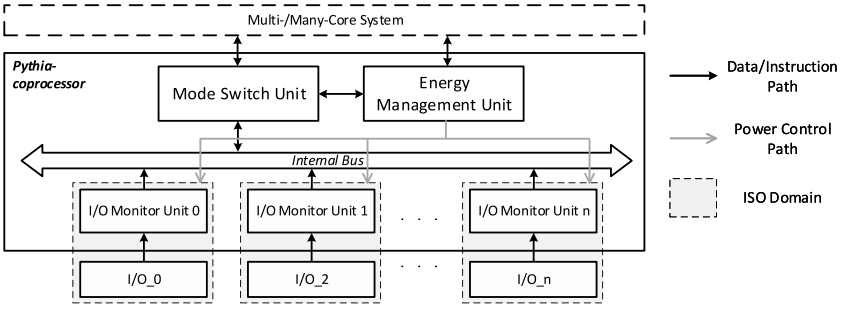


Fig. 5. Top-level design of *Pythia*-coprocessor.

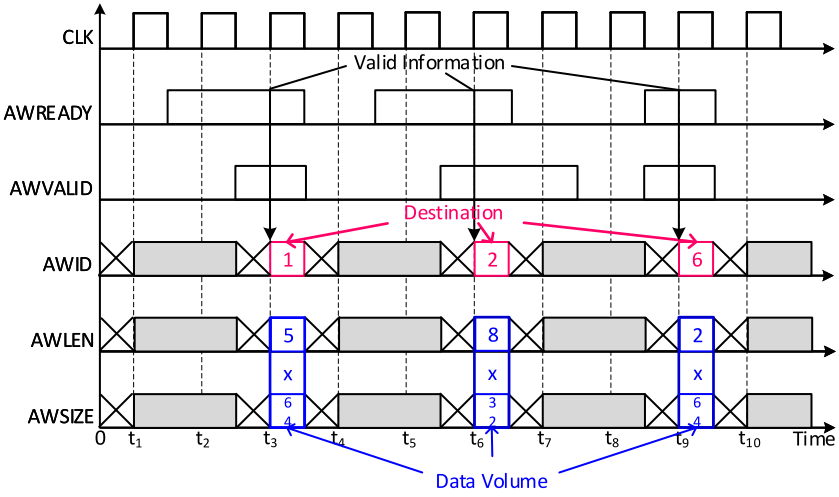


Fig. 6. Example of write address channel (Waveform) in AMBA AXI.

4.2. I/O Monitor Unit at I/O Pins (IMU_P)

I/O data decomposition. When transferring I/O data to a system, transactions are always first sent to the I/O pins via serial I/O communication protocols [17], *e.g.*, SPI, I^2C, *etc.* Unlike the on-chip communication protocols unified throughout the system, multiple serial I/O communication protocols are often involved in the same system for different application scenarios. For instance, LIN, CAN and FlexRay are three commonly used serial communication protocols in automotive systems, designed for different communication speeds. In *Pythia*-MCS, different IMU_Ps have been designed and implemented to support commonly used serial communication protocols. Here, we detail the I/O data decomposition and design methods of IMU_P using the example of FlexRay [24], which is the most complicated high-speed protocol of those we implemented.

In order to save the usage of I/O pins, serial I/O communication usually involves fewer channels than on-chip communication. This means that in serial I/O communication, most information is transferred using the same channel and organized with a restricted frame format. Therefore, we propose a 2-step method to decompose the I/O data:

Step 1 — Protocol decomposition: the 1 and 0 signals at the I/O pins are converted to valid transaction messages based on the corresponding serial communication protocol.

Step 2 — Frame format decomposition: the converted transaction messages are analyzed to extract their targets and volumes based on the corresponding frame format.

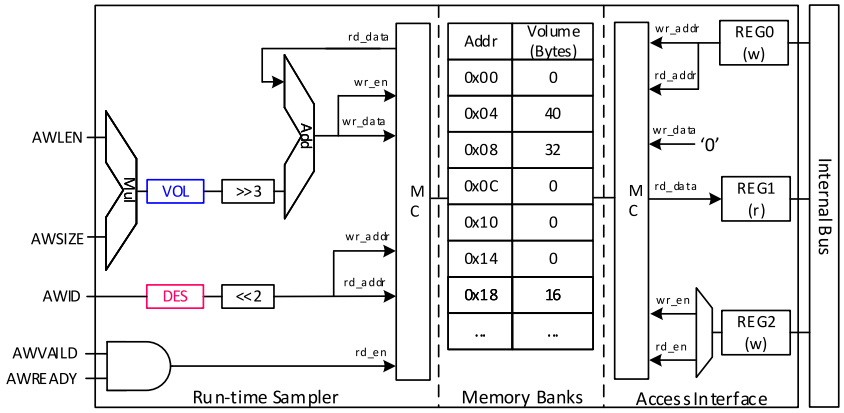


Fig. 7. Design of IMU_R (MC: Memory Controller; DES: Register for Destination; VOL: Register for Data Volume).

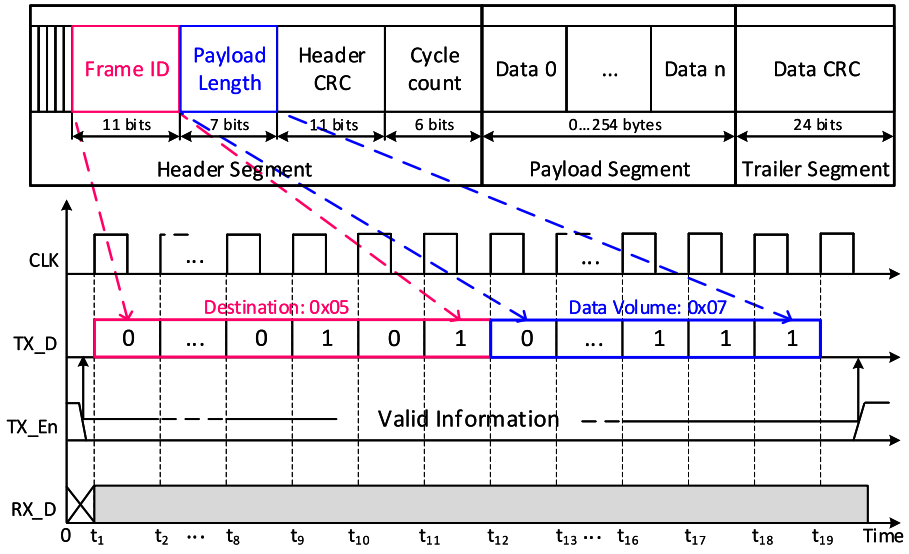


Fig. 8. Example of FlexRay transaction frames and waveform.

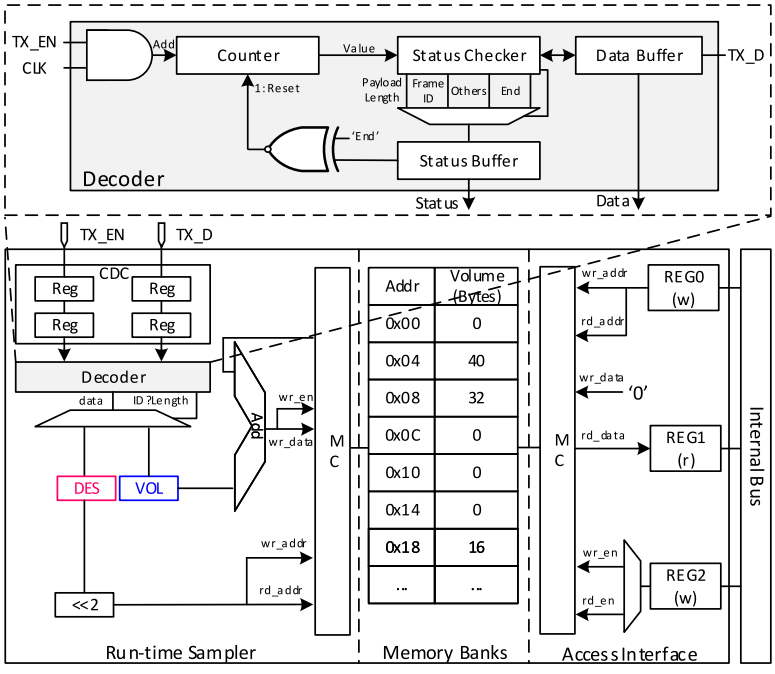


Fig. 9. Design of IMU_P. The decoder is required to be updated to support different serial communication protocols.

The FlexRay communication protocol involves four essential pins (single-bit) [24]: TX_D, RX_D, TX_EN, and RX_EN. Specifically, TX_D (RX_D) indicates the data sent from a master (slave) to a slave (master), and the TX_EN (RX_EN) determines the validness of the corresponding data line (see the lower part of Fig. 8). Therefore, to acquire valid transaction messages in protocol decomposition, IMU_P must capture these 4 I/O pin values. Moreover, the data transferred on the TX_D or RX_D always follows a fixed format (see the upper part of Fig. 8): the first 5 bits initialize a transaction, followed by header frames, payload frames and trailer frames. These frames respectively store the necessary information, payload, and the transaction CRC check. In the header frame, the 6th–17th bits and the 18th–25th bits indicate the transaction's destination and data volume (υ^*), respectively, which are desired in frame format decomposition.

As shown in Fig. 8, the example initializes an I/O data packet, which is sent to task 5 with volume 7 bytes.

IMU_P design. Fig. 9 shows the design of IMU_P. The main components are a run-time sampler, an access interface and memory banks. IMU_P uses the same design of access interface and memory banks as IMU_R, ensuring compatibility between the two types of IMU — the designs abstract unified access interfaces and memory addresses for the MSU.

The run-time sampler in IMU_P contains three main modules: a Clock Domain Cross (CDC) module, a decoder and a memory control logic. Since the signals sent to the I/O pins are generated off-chip, usually belonging to an unknown frequency domain, a CDC module is required to avoid occurrences of metastable states [17]. We implement the CDC module as a two-level register chain, eliminating 99% of the metastable states [17]. The decoder decomposes the transactions captured at the I/O pins using the introduced 2-level method, returning its destination (*i.e.*, τ_d) and volume (υ^*). The decoder (shown in the upper part of Fig. 9) is based on a counter, a status checker, and two buffers. At run-time, the counter records orders of transmitted bits on TX_D (RX_D), and it increases when both TX_EN (RX_EN) and CLK are equal to '1'. At the same time, the status checker synchronizes with the counter and determines current transaction status: 'payload length', 'frame ID', 'others', or 'end'. When the payload length or frame is transmitted, the decoder stores the value received from TX_D (RX_D) in a data buffer. When the current status is 'end', the status checker resets the counter, and the decoder omits the 'others' status. At the same time, the status checker also updates the acquired transaction status to a status buffer. When both data and status are transmitted to memory control logic, the memory control logic updates the corresponding task's data volume in the memory banks after a simple calculation.

4.3. Mode Switch Unit (MSU)

As the brain of the *Pythia*-coprocessor, MSU takes charge of triggering a mode switch. As introduced in Design Concept 2, a mode switch is triggered by: (i) any I/O-related HI-task exceeding its TH-I/O at the I/O access point; or, (ii) 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 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 (Y_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 Y_i^{L}. If $\upsilon_i > Y_i^{\mathrm{L}}$, 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 (Fig. 10(a)). The hardware/software co-design propounds software executed on a ready-built processor (*e.g.*, MicroBlaze [25] or RISC-V [26]). 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 (Fig. 10(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 comparator. 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 1 when $\upsilon_i > Y_i^{\mathrm{L}}$.

In both design methods, we introduce (i) a shadow register to guarantee timing synchronization between the MSU and the entire system; and, (ii) communication interfaces to the EMU to report the current system mode (denoted L). The current system mode is stored at the last memory address.

4.4. Energy Management Unit (EMU)

Pythia-MCS brings extra hardware implementation, which potentially increases overall power consumption. To improve the energy efficiency of *Pythia*-MCS, we now propose an energy management method with a corresponding EMU.

Power Domains We partition *Pythia*-MCS into two power domains: *always-on power domain* (*AON domain*) and isolated power domain (*ISO domain*). The clocks and power of AON domain cannot be switched off during run-time. This domain contains the core system, MSU and EMU. Conversely, the clocks and power of the ISO domain can be optionally switched off by EMU. This domain includes all IMUs and the connected I/Os. Fig. 5 also shows this partitioning.

The partitioning of power domains provides a possibility to adjust energy consumption in *Pythia*-MCS. We now detail the energy management framework and EMU design.

Energy Management Framework Fig. 11 shows an overview of the energy management framework: an EMU connects the many-core system, MSU, IMUs, I/Os, and a DC-DC converter (off-chip), respectively. At run-time, EMU periodically synchronizes with the many-core system and MSU, and then controls the clocks and power of the ISO domain correspondingly. Power consumption of the system is controlled via *clock gating* and *power gating*:

Clock gating: clock gate modules are inserted between ISO domain and the clock sources. Hence, the EMU can turn the clock of any specified IMU and I/O on/off.

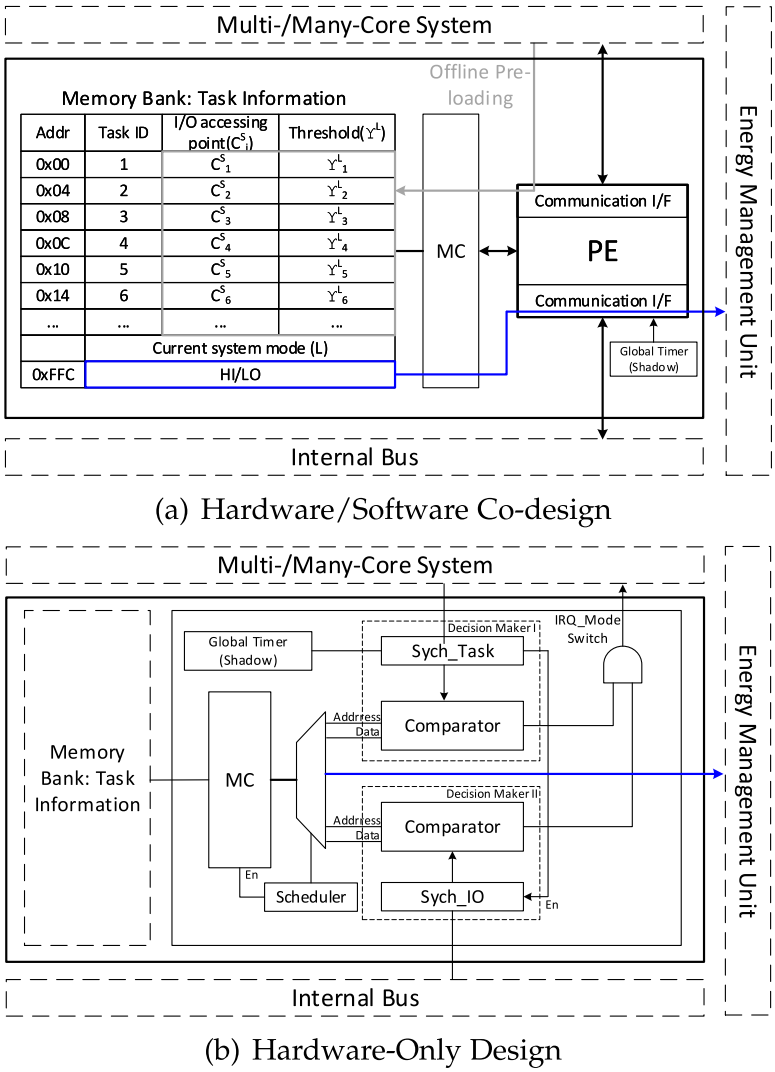


Fig. 10. Design of MSU (MC: Memory Controller).

Power gating: The EMU is connected to an off-chip DC-DC converter, which drives the voltage of ISO domain. Hence, the EMU can switch the power of the entire ISO domain on/off.

The clock and power gating can effectively reduce the static and dynamic power consumption of *Pythia*-MCS, respectively.

Energy Management Methods and EMU Design. We now introduce two energy management methods:

Passive control: The EMU receives energy control requests from the core system (applications) and then manages the clocks/power correspondingly.

Active control: The EMU checks the current system mode and then *automatically* switches off the clocks/power of the unused part(s) in the ISO domain.

Unlike passive control, which directly forwards energy control requests, active control involves more complicated execution procedures. Before system execution, the criticality of each I/O is stored (denoted as l_i^{IO}) in the EMU. Here, the *'criticality of an I/O'* indicates the highest criticality of the task which may access this I/O. During system execution, the EMU continuously synchronizes with the MSU to acquire the current system mode (L), which it then compares against each l_i^{IO}. If $l_i^{\mathrm{IO}} < L$, the EMU switches off the clock of this I/O and the associated IMU. If L is higher than all l_i^{IO}, EMU turns off the power of the entire ISO domain.

To support energy management methods, we introduce the EMU design in Fig. 12. Its main components are a power control IP, a decision-maker and a memory module. The power control IP manages low-level control of clock and power gating, which can be configured using an off-the-shelf IP, *e.g.*, an ARM power policy unit [27]. In passive control, energy control requests are directly passed through to the power control IP. In active control, the pre-loaded criticality of each I/O is stored in memory banks; the run-time synchronization and comparison are respectively handled by a synchronizer and a comparator in the decision-maker. Finally, a multiplexer is presented to switch the two energy control methods.

Conflicts Handler. The two energy control methods could cause a conflict when they manage the clocks/power at the same time. As discussed above, passive control is generated by applications during system execution, whereas active control is determined before system execution. Therefore, passive control is more flexible, as it can react to practical run-time situations, *e.g.*, malfunctions from underlying hardware or the external environment. We hence deem that passive control is more privileged than active control.

With this in mind, we propose a *conflicts handler* to detect and manage conflicts: passive control can always switch on/off the clocks/power of the parts managed by active control. That means when both active

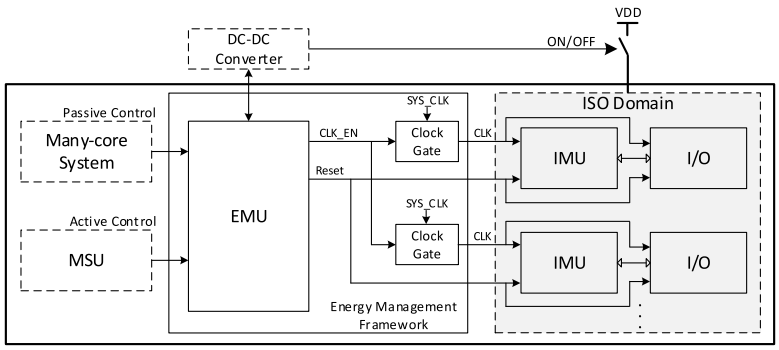


Fig. 11. Energy management framework in *Pythia*-MCS.

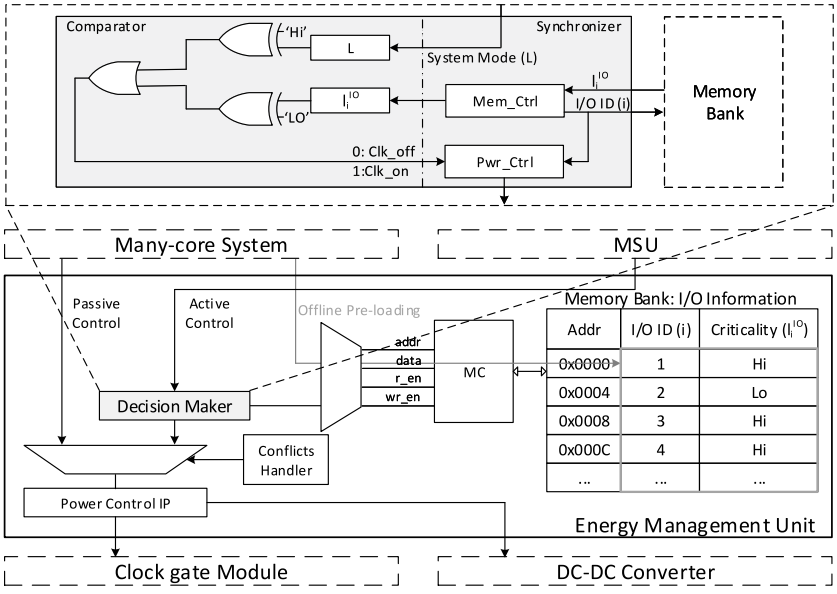


Fig. 12. Energy Management Unit (EMU).

and passive controls try to manage the clocks/power simultaneously, the conflicts handler will disable passive control as erroneous. Moreover, the system designer can also configure the conflicts handler to decide the time effectiveness of a passive control request, allowing the system to switch back to active control *automatically*. For instance, a passive control request can always be valid, or only be valid for a specific time period.

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.

5. 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 [13]. An intermediate concept of *semi-clairvoyance*, which lies between the two extremes of clairvoyance and non-clairvoyance, has been introduced [13]. 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.

5.1. 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

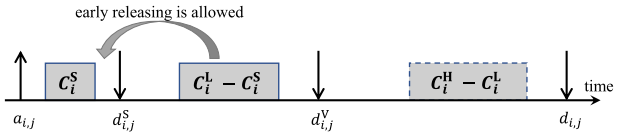


Fig. 13. An illustration for dividing an HI-job into sub-jobs in LO-mode.

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.

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 [6]. In light of the predicting coprocessor architecture presented in this paper, we introduce one more parameter, C_i^{S} (see Section 2.2 for the measurement), for each HI-task τ_i to model the certain clairvoyance our architecture brings.[3] 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[4] 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.

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

[4] 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 [13], such prediction is always made at a job's release regardless of which scheduling algorithm is applied.

5.2. Algorithm EDF-VDSD

For the traditional scheduling of implicit-deadline sporadic tasks, EDF-VD [6] 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 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,[5] 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. Fig. 13 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 [28,29].

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.

5.3. Schedulability test

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

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

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*

$$x \geq \frac{U^{\mathrm{L}}_{\mathrm{HI}}}{1 - U^{\mathrm{L}}_{\mathrm{LO}}}. \tag{3}$$

Proof Sketch. First, we consider a *fluid* schedule, where each LO-task τ_i is continuously assigned an execution rate of u^{L}_i and each HI-task τ_k is continuously assigned an execution rate of u^{L}_k/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

$$\sum_{\tau_i \in \tau_{\mathrm{HI}}} \frac{u^{\mathrm{L}}_i}{x} + \sum_{\tau_i \in \tau_{\mathrm{LO}}} u^{\mathrm{L}}_i = \frac{U^{\mathrm{L}}_{\mathrm{HI}}}{x} + U^{\mathrm{L}}_{\mathrm{LO}} \overset{\{\text{by(3)}\}}{\leq} 1.$$

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* $x \geq \frac{U^{\mathrm{L}}_{\mathrm{HI}}}{1-U^{\mathrm{L}}_{\mathrm{LO}}}$*, under EDF-VDSD, in the* HI*-mode, all* HI*-jobs meet their actual deadlines, if*

$$\sum_{\tau_i \in \tau_{\mathrm{HI}}} \max\left\{ \frac{u^{\mathrm{H}}_i}{1 - \frac{C^{\mathrm{S}}_i}{C^{\mathrm{L}}_i} \cdot x}, \frac{u^{\mathrm{L}}_i - \frac{C^{\mathrm{S}}_i}{T_i}}{1 - x} \right\} \leq 1. \tag{4}$$

Proof Sketch. Given that $x \geq \frac{U^{\mathrm{L}}_{\mathrm{HI}}}{1-U^{\mathrm{L}}_{\mathrm{LO}}}$, 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)** $t^* \leq d^{\mathrm{S}}_{i,j}$ and **(ii)** $d^{\mathrm{S}}_{i,j} < t^* \leq d^{\mathrm{V}}_{i,j}$. Note that it cannot be the case that $t^* > d^{\mathrm{V}}_{i,j}$, because in that case, either the mode switch would have been triggered by $\tau_{i,j}$ at $d^{\mathrm{S}}_{i,j}$ earlier than t^* ($\tau_{i,j}$ executes for more than C^{L}_i) or $\tau_{i,j}$ would have been completed by $d^{\mathrm{V}}_{i,j} < t^*$ ($\tau_{i,j}$ executes for at most C^{L}_i).

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

$$\frac{C^{\mathrm{H}}_i}{d_{i,j} - t^*} \leq \frac{C^{\mathrm{H}}_i}{d_{i,j} - d^{\mathrm{S}}_{i,j}} = \frac{C^{\mathrm{H}}_i}{T_i - D^{\mathrm{S}}_i} = \frac{u^{\mathrm{H}}_i}{1 - \frac{D^{\mathrm{S}}_i}{T_i}} = \frac{u^{\mathrm{H}}_i}{1 - \frac{C^{\mathrm{S}}_i}{C^{\mathrm{L}}_i} \cdot x},$$

where the last equality is because of (2).

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

$$\frac{C^{\mathrm{L}}_i - C^{\mathrm{S}}_i}{d_{i,j} - t^*} \leq \frac{C^{\mathrm{L}}_i - C^{\mathrm{S}}_i}{d_{i,j} - d^{\mathrm{V}}_{i,j}} = \frac{C^{\mathrm{L}}_i - C^{\mathrm{S}}_i}{T_i - D^{\mathrm{V}}_i} = \frac{u^{\mathrm{L}}_i - \frac{C^{\mathrm{S}}_i}{T_i}}{1 - x}.$$

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

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

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

$$u^{\mathrm{H}}_i \leq \frac{u^{\mathrm{H}}_i}{1 - \frac{C^{\mathrm{S}}_i}{C^{\mathrm{L}}_i} \cdot x} \leq f_i.$$

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*

$$\sum_{\tau_i \in \tau_{\mathrm{HI}}} \max\left\{ \frac{u^{\mathrm{H}}_i}{1 - \frac{C^{\mathrm{S}}_i}{C^{\mathrm{L}}_i} \cdot \frac{U^{\mathrm{L}}_{\mathrm{HI}}}{1-U^{\mathrm{L}}_{\mathrm{LO}}}}, \frac{u^{\mathrm{L}}_i - \frac{C^{\mathrm{S}}_i}{T_i}}{1 - \frac{U^{\mathrm{L}}_{\mathrm{HI}}}{1-U^{\mathrm{L}}_{\mathrm{LO}}}} \right\} \leq 1. \tag{5}$$

Proof. Setting $x = \frac{U^{\mathrm{L}}_{\mathrm{HI}}}{1-U^{\mathrm{L}}_{\mathrm{LO}}}$ 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. □

5.4. Discussions

We next discuss the benefits EDF-VDSD brings from an analytical perspective. Empirical studies and evaluation are presented in Section 6.

Comparison with non-clairvoyance EDF-VD. It is clear that the special case where $\forall i \in \tau_{\mathrm{HI}}, C^{\mathrm{S}}_i = C^{\mathrm{L}}_i$ 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 [30], which is also dominated by a later improved EDF-VD analysis in [6].

Unfortunately, our schedulability test does not have a strict dominance over the improved EDF-VD analysis in [6]. Nonetheless, the quarter-clairvoyance MC scheduling model and EDF-VDSD bring certain advantages over EDF-VD, even with the improved analysis in [6]. The following example is not deemed schedulable under EDF-VD, even with the analysis in [6], 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, $T_1 = 10$, $C^{\mathrm{H}}_1 = 8$, $C^{\mathrm{L}}_1 = 3$, and $C^{\mathrm{S}}_1 = 1$; for the LO-task τ_2, $T_2 = 10$, $C^{\mathrm{L}}_2 = 5$. That is, in this system, $U^{\mathrm{H}}_{\mathrm{HI}} = 0.8$, $U^{\mathrm{L}}_{\mathrm{HI}} = 0.3$, $U^{\mathrm{L}}_{\mathrm{LO}} = 0.5$, $x = (0.3)/(1-0.5) = 0.6$, and $C^{\mathrm{S}}_1/C^{\mathrm{L}}_1 = 1/3$.

Under non-clairvoyant EDF-VD,

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

which means that even the improved EDF-VD schedulability test in [6] 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+

```
if U_LO^L + U_HI^H ≤ 1 then
    Apply ordinary EDF from the beginning (i.e., no MC and no mode
      switch at all), and declare SUCCESS;
else
    if U_HI^L / (1−U_LO^L) ≤ (1−U_HI^H) / U_LO^L then
        Apply EDF-VD, and declare SUCCESS;
    else
        if Σ_{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
            Apply EDF-VDSD, and declare SUCCESS;
        else
            Declare FAILURE.
        end
    end
end
```

6. Experimental evaluation

In this section, we conduct extensive experiments and a real-world case study to evaluate *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 [31] and connected to a 7×7 mesh type open-source NoC [20]. As well as the *Pythia*-coprocessor, the NoC also contained 32 MicroBlaze processors [25], memory and I/O peripherals. The software executing on the processors (OS kernels and user applications) was compiled using a Xilinx MicroBlaze GNU tool-chain [25]. We selected FreeRTOS (v.9.0.0) as the OS kernel for all processors, with the modifications introduced in Section 3.4. The IMU in *Pythia*-coprocessor was implemented using the methods described in Section 4.1 to support I/O monitoring at the routers (denoted as *PY_R*) and pins (denoted as *PY_P*). The MSU in the coprocessor was implemented using the method described in Section 4.3, with hardware/software co-design (denoted *PY|hs*) and hardware-only design (denoted *PY|hw*). Following this naming strategy, we denote the implementation of *Pythia*-MCS as *PY_A|B*, where A and B indicate the implementation methods of the IMU and MSU, respectively. For instance, *PY_R|hw* represents the system monitoring I/Os at the routers, designed using the hardware-only method. To enable comparison, we also built a conventional MCS framework (reviewed in Section 3.3) on a similar hardware architecture (denoted *MC|conv*) without *Pythia*-coprocessor. The *MC|conv* system architecture is shown in the upper part of Fig. 3. All architectures ran at 100 MHz.

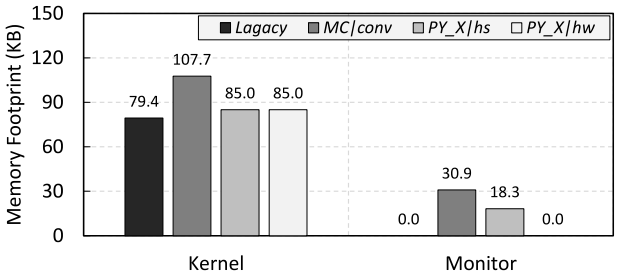


Fig. 14. Run-time software overhead. The software overhead is evaluated via memory footprint (unit: KB).

6.1. Software overhead

In this section, we compare software overheads of the legacy system,[6] with *MC|conv* and all the variants of *Pythia*-MCS.

Experimental Setup. The software overhead was evaluated using the run-time memory footprint [32], with specific consideration of the OS kernel and execution monitor (memory size tool: Xilinx MicroBlaze GNU tool-chain [25]). The legacy OS kernel was fully-featured with essential I/O drivers [33]. Since *PY_R* and *PY_P* abstract a unified interface to software level (described in Section 4), adopting different methods of I/O monitoring does not affect the software overhead of *Pythia*-MCS. In experimental results, we use *PY_X* to denote the *Pythia*-MCS configured as either *PY_R* or *PY_P*.

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 Fig. 14. In *MC|conv*, the introduction of an execution monitor and the modifications to OS kernel bring an additional 60 KB (75.9%) memory footprint compared to the legacy system. By contrast, in both *PY_X|hs* and *PY_X|hw*, run-time monitoring and mode switch triggers rely on the 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_X|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).

6.2. Hardware overhead

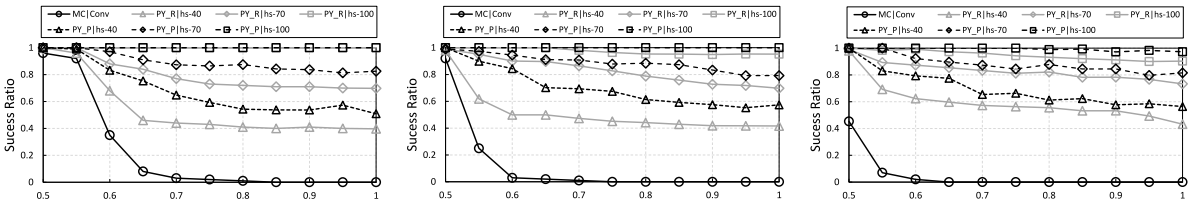
Pythia-MCS requires additional hardware implementation for the coprocessor. Hence, in this section, we evaluate the hardware overhead of the *Pythia*-coprocessor.

Experimental Setup. We first configured *Pythia*-coprocessor to monitor two I/Os (FlexRay); and then evaluated the coprocessor and both 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) [19] and compared using Look Up Tables (LUTs), registers, DSPs, Block RAMs (BRAMs) and average power consumption [34].

Obs.2. The design of *Pythia*-coprocessor was resource-efficient compared to generic CPUs. Its hardware consumption was slightly higher than evaluated I/O controllers.

As shown in Table 1, when we implemented MSU using the hardware-only method, the coprocessor in *PY_R|hw* and *PY_P|hw*, required less hardware than either MB-F (12.0% and 14.5% LUTs; 9.0% and 12.0%

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



(a) 8-core Architecture (b) 16-core Architecture (c) 32-core Architecture

Fig. 15. Case study: success ratios of the HI-tasks in conventional MCS and *Pythia*-MCS (the x-axis denotes the target utilization).

Table 1
Hardware overhead (Implemented on FPGA).

	LUTs	Registers	DSP	RAM (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_R|hw	587	396	0	16	109
PY_R|hs	973	583	0	16	133
PY_P|hw	714	527	0	16	122
PY_P|hs	1093	773	0	16	140

registers; 42.2% and 47.3% power consumption) or MB-B (68.7% and 83.6% LUTs; 74.9% and 99.6% registers; 85.8% and 96.1% power consumption). Due to the integration of a mature processor, *PY_R|hs* (*PY_P|hs*) consumed more hardware than *PY_R|hw* (*PY_P|hw*). When compared to the CAN and SPI controllers, all variants of *Pythia*-MCS had similar consumption of both LUTs and registers, but additional memory consumption. The memory consumed additional power for refresh [17]; hence, the coprocessors consumed more than 20 times the power of the I/O controllers.

Obs.3. In *Pythia*-MCS, placing I/O monitoring at pins consumed more hardware than placing it at routers.

This observation can be seen by comparing *PY_R|hs* (*PY_R|hw*) and *PY_P|hs* (*PY_P|hw*) in Table 1. This is because monitoring I/O data at the pins requires a more complicated decomposition of protocol and frame format (see Section 4.2). The IMUs in *PY_P* hence involve more control logic and state machines compared to *PY_P* (detailed in Section 4.2) to support these procedures.

6.3. Automotive case study

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

Systems Configuration. To analyze the benefits brought by *Pythia*-MCS with different I/O monitoring methods, *MC|conv*, *PY_R|hs* and *PY_P|hs* were examined. We configured *PY_X|hs* as *PY_X|hs*-40/70/100, enabling 40/70/100% of I/O-related tasks using *I/O-driven mode switch*. In other words, *PY_X|hs*-z indicated the system was z% of *Pythia*-MCS.

Task sets. We introduced two sets of I/O-related tasks[7]:

- 20 automotive safety tasks, selected from Renesas automotive use case database [18], *e.g.*, CRC, RSA32, *etc.*.
- 20 automotive function tasks, selected from EEMBC benchmark [35], *e.g.*, fast Fourier transform.
- synthetic workloads (in the LO-task category), selected from EEMBC benchmark, which could be added into system to control overall system utilization.

The HI-tasks had been certified as ASIL-D tasks [1], 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 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 Y_i^{L}) were obtained using the method described in Section 2. The *MC|conv* also contained a simulated HI-task for the execution monitor (described in Section 3.2), which was not required by *PY_X|hs*. Each task had a defined period, with overall system utilization in both LO- and HI-mode approximately 50%. Following Section 5, we adopted implicit deadlines for all tasks.

Notably, in practice, execution time of a task is affected by diverse 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 8/16/32 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 s, 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.

Experimental metrics. We used two metrics to evaluate the examined systems under each target utilization, *success ratio* and *number of services* (NoS). The success ratio recorded the percentage of an examined trail executed without deadline miss of any HI-task, and the NoS measured the number of LO-tasks executions. Figs. 15 and 16 demonstrate the experiment results.

Obs.4. Introducing *I/O-driven mode switch* is beneficial.

This observation is supported by the results in Figs. 15(a), 15(b), and 15(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_X|hs*-100) consistently outperformed the partial *Pythia*-MCSs (*PY_X|hs*-70 and *PY_X|hs*-40). This means that having a higher percentage of the system involving *I/O-driven mode switch* introduces more benefits.

Obs.5. Increasing the number of processors significantly reduced the success ratio of the conventional MCS framework. Such issues were effectively eliminated by *Pythia*-MCS.

This observation is shown in the comparison between Figs. 15(a) and 15(c). In a 8-core *MC|conv*, a significant drop in the success ratio occurred at 65% of target utilization, whereas this drop moved to 50% of target utilization in a 32-core *MC|conv*. This observation mainly results from the additional on-chip interfaces and resource contention generated by the introduced processors and tasks.

In the *Pythia*-MCS, run-time monitoring is placed in hardware level; hence, the *Pythia*-coprocessor can detect abnormal behaviors due to

[7] In order to demonstrate wide applicability, and the value of a true mixed-criticality system, we *randomly* selected tasks from both real automotive software and open-source benchmarks, in an attempt to capture a wide range of tasks.

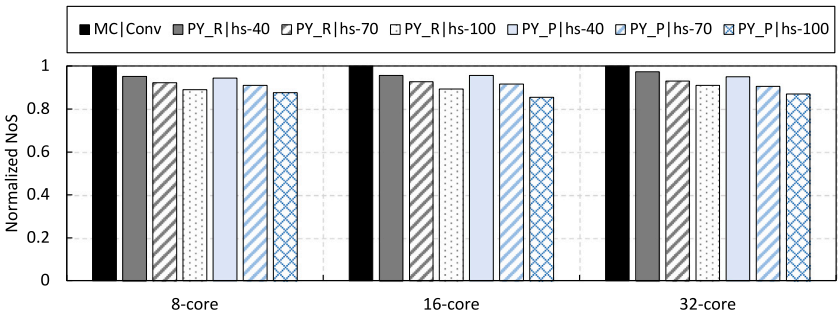


Fig. 16. Case study: Average NoS of LO-tasks (normalized by *MC|conv*).

large amounts of data generation promptly, and trigger a mode switch. In a 32-core system (Fig. 15(c)), when target utilization approaches 100%, *PY_X|hs*-100 maintains a success ratio which is still higher than 95%. This observation demonstrates the benefits and applicability of introducing the *Pythia*-MCS in multi-many-core architectures.

Obs.6. In *Pythia*-MCS, placing I/O monitoring at pins brought more benefits than placing monitoring at routers.

As shown in Fig. 15, for the *Pythia*-MCSs with the same settings, *PY_P|hs* always outperformed *PY_R|hs*. This is because monitoring I/O data at the pins ensures the most timely mode switches, increasing the overall success ratios.

Obs.7. Introducing the I/O-driven mode switch in *Pythia*-MCS decreased the LO-tasks' NoS. The decrement is caused by the miss-predictions of the mode switches.

This observation is given by Fig. 16. Compared to the conventional MCS (*MC|conv*), the *PY_X|hs*-40 decreased the LO-tasks' NoS by 7%. Such decrement was further magnified in *PY_X|hs*-70 and *PY_X|hs*-100. In the worst case, *i.e.*, in *PY_P|hs*-100, the decrement of the LO-tasks' NoS reached up to 14.7%. This decrement is caused by the miss-prediction of the mode switches, where the *Pythia*-MCS triggers a mode switch when it is not necessary (*i.e.*, false-positive). In the following section, we specifically evaluate the accuracy of the prediction in *Pythia*-MCS.

6.4. Accuracy of prediction

Although Section 6.3 demonstrates the benefits brought by I/O-driven mode switch in *Pythia*-MCS, we acknowledge that *accuracy* of the prediction mechanism finally determines feasibility of the proposed design. We now examine the accuracy of the prediction mechanism considering two scenarios:

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

Scenario II: false positive 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 6.3 with *MC|conv* and *PY_X|hs* (*PY_X|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_X|hs* also triggers the switch. Secondly, for all executing cases where *PY_X|hs* triggers mode switch, *accuracy of overrun prediction* calculates the percentage of executing cases where *MC|conv* also triggers the switch. From the results (Fig. 17), we observe:

Obs.8. The prediction mechanism does not introduce additional safety concerns in *Pythia*-MCS, as the system never missed a required mode switch.

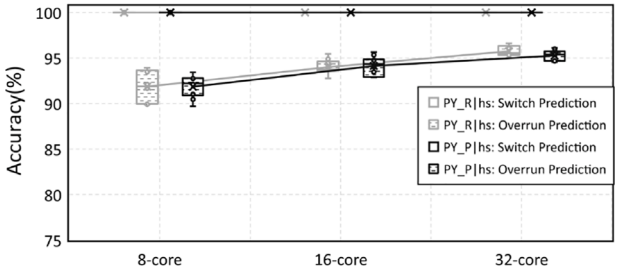


Fig. 17. Prediction accuracy of *Pythia*-MCS.

As shown in Fig. 17, 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_X|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 2.2.

Obs.9. **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.**

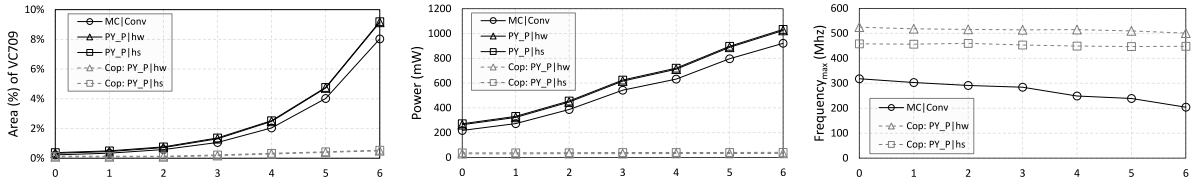
As shown in Fig. 17, in a 8-core *PY_X|hs*, the accuracy of overrun prediction averaged around 90%, which means the *Pythia*-MCS has about 10% probability of triggering an unrequired mode switch. Therefore, *Pythia*-MCS does not completely avoid Scenario II.

Fortunately, with an increasing number of processors, this weakness can be effectively alleviated. As shown, *PY_X|hs* raises the accuracy of overrun prediction to around 93% for the 16-core system, and 95% for the 32-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 the observation, we conjecture that the accuracy of overrun prediction in *Pythia*-MCS would approximate to 100% with more processors.

6.5. Scalability

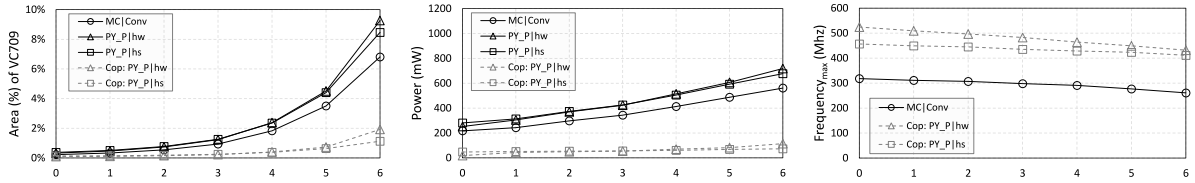
We now examine the scalability of *Pythia*-MCS using a varying number of processors and I/Os.

Experimental Setup. As seen in Section 6.2, placing I/O monitoring at the pins usually consumes more resources than monitoring at the routers under the same settings. Hence *Pythia*-MCS was configured as *PY_P|hs* and *PY_P|hw*. We adopted the same method described in Section 6.2 to synthesize and implement *PY_P|hs*, *PY_P|hw* and conventional MCS, firstly with a scaling number of basic MicroBlaze processors and then with a scaling number of I/Os (Ethernet). In the experiments,



(a) Area consumption v.s. η_{core}. (b) Power v.s. η_{core}. (c) Maximum frequency v.s. η_{core}.

Fig. 18. Area, power, and the maximum frequency v.s. scaling factor η_{core} (CoP: *Pythia*-coprocessor, the x-axis denotes η_{core}).



(a) Area consumption v.s. η_{io}. (b) Power v.s. η_{io}. (c) Maximum frequency v.s. η_{io}.

Fig. 19. Area, power, and the maximum frequency v.s. scaling factor η_{io} (CoP: *Pythia*-coprocessor, the x-axis denotes η_{io}).

we introduced two scaling factors: η_{core} to control number of processors ($2^{\eta_{core}}$) and η_{io} to control number of I/Os ($2^{\eta_{io}}$).

Scalability of area consumption. We compared the area consumption of the evaluated systems with varying η_{core} and η_{io}. The evaluated results were normalized by using the overall area of the experimental platform (Xilinx VC709).

Obs.10. The area consumption of both the conventional MCS and *Pythia*-MCS is linearly scaled by η_{core} and η_{io}.

This observation is supported by Figs. 18(a) and 19(a). As shown, when the system scaled with η_{core}, the area consumption of *PY_P|hs* and *PY_P|hw* was consistently similar to *MC|conv*. For instance, in a 64-core system ($\eta_{core} = 6$), both *PY_P|hs* and *PY_P|hw* introduced less than 0.3% additional area consumption with respect to the *MC|conv*. When the system scaled with η_{io} (Fig. 19(a)), both *PY_P|hs* and *PY_P|hw* suffered from slight increment on area consumption with respect to *MC|conv*, since the *Pythia*-coprocessor required to integrate additional IMUs for I/O monitoring. For example, in a 64-I/O system ($\eta_{io} = 6$), *PY_P|hw* and *PY_P|hs* brought an additional 2% and 3% area consumption compared to *MC|conv*, respectively. This observation also illustrates the area-efficiency of the proposed design.

Obs.11. When the system scales with η_{io}, *PY_P|hs* has better area consumption scalability than *PY_P|hw*.

This observation is given by Fig. 19(a). This is because that *PY_P|hs* always adopted a fixed MSU design (based on a mature processor), whereas *PY_P|hw* required additional hardware implementation when η_{io} increased. Please see Section 4 for the design details of the MSU.

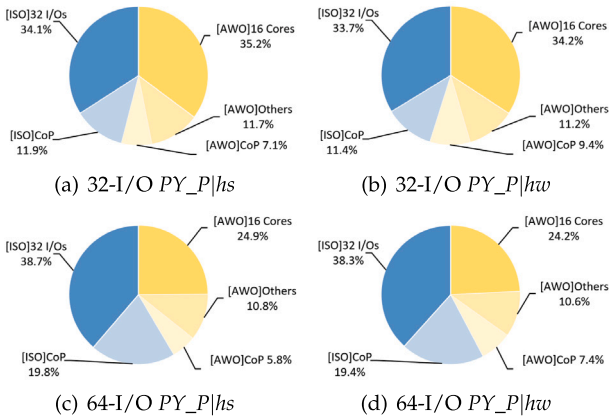
Scalability of power. We now compare the power consumption of the evaluated systems, calculated as the sum of static and dynamic power, by varying η_{core} and η_{io}.

Obs.12. η_{core} and η_{io} linearly scale the power consumption of both conventional MCS and *Pythia*-MCS.

The power consumption of a system is usually determined by four factors [37]: voltage, clock frequency, toggle rate and design area. Because the same voltage, clock frequency and simulated toggle rate were assigned to the evaluated systems, the design area dominated the overall power consumption. As expected, in Figs. 18(b) and 19(b), we observed linearly increased power consumption in the evaluated systems when either η_{core} or η_{io} increased.

Scalability of maximum frequency. We examine the maximum frequency of the *Pythia*-coprocessor (in *PY_P|hs* and *PY_P|hw*) and *MC|conv* using varying η_{core} and η_{io}.

Obs.13. Introducing *I/O-driven mode switch* in conventional MCS does not affect the system's maximum performance.



(a) 32-I/O *PY_P|hs* (b) 32-I/O *PY_P|hw*

(c) 64-I/O *PY_P|hs* (d) 64-I/O *PY_P|hw*

Fig. 20. Power distribution in *Pythia*-MCS (CoP: *Pythia*-coprocessor).

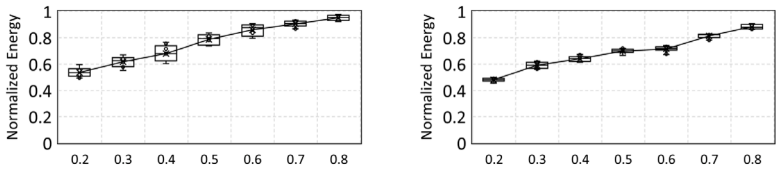
As shown in Figs. 18(c) and 19(c), when the system scaled with η_{core} or η_{io}, the maximum frequency of the coprocessor was always greater than the *MC|conv*. This indicates that the coprocessor did not become a critical path and could not reduce maximum system performance.

6.6. Power distribution

Although the design of *Pythia*-MCS keeps energy-efficiency in mind, the introduction of a coprocessor still increases its power consumption compared to conventional MCS, especially when the number of I/Os increases (*Obs. 11.*). In this section, we report the power distribution in *Pythia*-MCS with different numbers of I/Os. We then examine its energy-efficiency with different workloads.

Experimental Setup. We first configured *PY_P|hs* and *PY_P|hw* to support 16 cores and 32/64 I/Os. We then use the same method described in Section 6.2 to synthesize and implement the system. Lastly, we decompose and analyze the relative power consumption of the system. The experimental results show the power distribution in *Pythia*-MCS and determine the percentage of switchable power consumption.

Obs.14. In 32-I/O *Pythia*-MCS, more than 45% of the power consumption can be switched off by power management; this percentage increases to about 58% in a 64-I/O system.



(a) 32-I/O *PY_P|hw* (b) 64-I/O *PY_P|hw*

Fig. 21. Energy-efficiency of *PY_P|hw* with different numbers of I/Os (normalized by *MC|conv*). *x*-axis: target I/O utilization.

This observation is given by Figs. 20. In 32-I/O *PY_P|hs* and *PY_P|hw*, 45.1% and 46.0% of power consumption is generated by I/Os and their associated modules. With the proposed energy management (detailed in Section 4.4), we can switch the clocks/power of these portions off. This means the proposed energy management can maximally save more than 46% of power consumption in *Pythia*-MCS. When it comes to 64-I/O *PY_P|hs* and *PY_P|hw*, this percentage increases to 57.7% and 58.5%, which demonstrates the effectiveness of energy management.

6.7. Energy-efficiency: Synthetic I/O workloads

Experimental Setup. We used *PY_P|hw* synthesized and implemented in Section 6.6, and deployed the processors as I/O requesters, generating synthetic I/O workloads without processing. At the same time, we introduced a *utilization checker* to indicate the utilization of each I/O. During experiments, the requesters continuously checked the utilization of each I/O: if an I/O did not reach a *target utilization*, the requesters continuously generated synthetic workloads for this I/O. The I/O then operated the workloads and acknowledged the requesters; if all I/Os were executed under the *target utilization*, the requesters paused. We recorded the clock frequency of each I/O and its associated modules (*i.e.*, ISO domain), and calculated their *dynamic* energy consumption using the energy model presented in [17]. We executed the experiments for 100 times.

Obs.15. Implementing EMU in *Pythia*-MCS effectively reduced the overall *dynamic* energy consumption.

As shown in Fig. 21, in 32-I/O *PY_P|hw* (*i.e.*, Fig. 21(a)), introducing the EMU saved about 40% energy consumption. This improvement increased to about 50% in a 64-I/O *PY_P|hw* (*i.e.*, Fig. 21(b)). However, in both experimental groups, we also reported that such benefits were slightly reduced with the increase of I/O utilization. This is because the EMU could not gate the clocks when the I/Os were busy.

6.8. Energy-efficiency: Case study

Experimental Setup. We now examine the energy-efficiency of *Pythia*-MCS using real-world use cases. We first configured *MC|conv*, *PY_P|hs*, and *PY_P|hw* with 8/16/32 processors and 2 I/Os (Ethernet controllers). We then executed the case study described in Section 6.3 with different target processors utilization [50%, 100%], and recorded the clock frequency of each I/O and its associated modules. With the recorded clock frequencies, we calculated the *dynamic* energy consumption of the ISO domain using the energy model presented in [17]. We executed the experiments 100 times. In Fig. 22, we report the average energy consumption of *PY_P|hs* and *PY_P|hw*. The experimental results are normalized by *MC|conv*.

Obs.16. Deploying EMU in *Pythia*-MCS effectively reduced the ISO domain's dynamic energy consumption while running the use cases. The improvement was reduced when the number of processors or the volume of workloads increased.

This observation is given by Fig. 22. With 8-core configurations, the ISO domain in *PY_P|hw* and *PY_P|hs* only consumed about 20% dynamic energy compared to *MC|conv*. This benefited from deploying

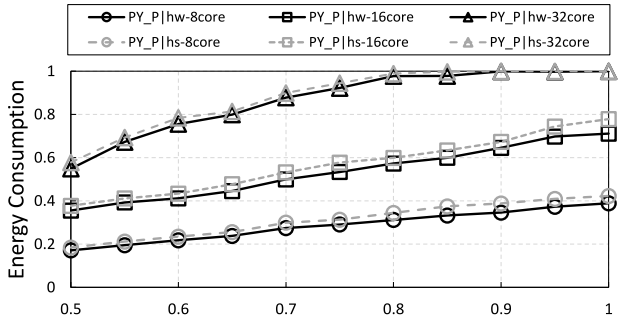


Fig. 22. Energy efficiency of *PY_P|hs* and *PY_P|hw*, examined using use case. *x*-axis: target processor utilization.

the EMU in *Pythia*-MCS, gating the source clocks when the I/Os were idle. The improvement was reduced when the number of processors or the volume of workloads increased, since they led the I/Os to be operated for the longer time duration, and the EMU had fewer opportunities to gate their clocks. As observed in Fig. 22, with 32-core configurations, the ISO domain in *PY_P|hw* and *PY_P|hs* consumed nearly 100% dynamic energy compared to *MC|conv*, since the I/Os were always busy. This observation aligns with experimental results using synthetic workloads, *i.e.*, Obs. 15.

Obs.17. With the same experimental setting, *PY_X|hs* consumed slightly more energy than the *PY_X|hw*.

This observation is shown in the comparison between *PY_X|hs* and *PY_X|hs* (in Fig. 22). Under the same experimental setting, *PY_X|hs* usually consumed 3%–7% extra energy than *PY_X|hw*. This is because the IMUs are designed differently in *PY_X|hs* and *PY_X|hs*, where *PY_X|hs* deploys a mature processor in the IMU, and the processor consumes slightly more energy during execution.

7. Conclusion

In this paper, we proposed a novel MCS framework (named *Pythia*-MCS), which simultaneously supports run-time I/O monitoring and I/O-driven mode switch. With these new features, *Pythia*-MCS achieves *future-prediction*, being able to foresee the over-execution of a task and triggering a timely mode switch. Moreover, we proposed two possible methods of allocating I/O monitoring, *i.e.*, at routers or pins, which provides a trade-off between design compatibility and monitoring timeliness. We also proposed an energy management framework to mitigate the power consumption caused by the new features. Correspondingly, we presented a new theoretical model (quarter-clairvoyance) and schedulability analysis to provide a timing guarantee for *Pythia*-MCS and to demonstrate improved schedulability compared to conventional MCS frameworks. As shown in the evaluation, *Pythia*-MCS outperformed the state-of-the-art MCS frameworks with varying hardware architectures. In addition, *Pythia*-MCS is resource- and energy-efficient.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive and helpful feedback. This work was supported in part by the U.S. National Science Foundation under Grants CNS-2103604, CNS-2140346, CNS 2113817, CNS-2038609, IIS-1724227, CCF-2118202 and CNS-2104181, in part by a start-up Grant from Wayne State University, USA, in part by start-up and REP grants from Texas State University, USA.

References

[1] ISO26262, Road vehicles-Functional safety, International Standard, 2018.
[2] A. Burns, R. Davis, Mixed Criticality Systems-A Review, Tech. Rep, Department of Computer Science, University of York, 2013.
[3] S. Vestal, Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance, in: RTSS, 2007.
[4] J. Caplan, Z. Al-Bayati, H. Zeng, B.H. Meyer, Mapping and scheduling mixed-criticality systems with on-demand redundancy, IEEE Trans. Comput. (2017).
[5] D. De Niz, B. Andersson, M. Klein, J. Lehoczky, A. Vasudevan, H. Kim, G. Moreno, Mixed-trust computing for RTS, in: RTCSA, 2019.
[6] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti, et al., The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems, in: ECRTS, 2012.
[7] A. Easwaran, Demand-based scheduling of mixed-criticality sporadic tasks on one processor, in: RTSS, 2019.
[8] J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, C. Lu, Mixed-criticality federated scheduling for parallel real-time tasks, Real-Time Syst. 53 (5) (2017) 760–811.
[9] R. West, Y. Li, E. Missimer, M. Danish, A virtualized separation kernel for mixed-criticality systems, Trans. Comput. Syst. (2016).
[10] P.K. Gadepalli, G. Peach, G. Parmer, J. Espy, et al., Chaos: a system for criticality-aware, multi-core coordination, in: RTAS, 2019.
[11] N. Kim, S. Tang, N. Otterness, J.H. Anderson, F.D. Smith, D.E. Porter, Supporting I/O and IPC via fine-grained OS isolation for mixed-criticality real-time tasks, in: RTNS, 2019.
[12] S. Baruah, V. Bonifaci, G. d'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, L. Stougie, Scheduling real-time mixed-criticality jobs, IEEE Trans. Comput. (2011).
[13] K. Agrawal, S. Baruah, A. Burns, Semi-clairvoyance in mixed-criticality scheduling, in: RTSS, York, 2019.
[14] A. Burns, R.I. Davis, Schedulability analysis for adaptive mixed criticality systems with arbitrary deadlines and semi-clairvoyance, in: 2020 IEEE Real-Time Systems Symposium, RTSS, IEEE, 2020, pp. 12–24.
[15] Q. Zhao, M. Qu, B. Huang, Z. Jiang, H. Zeng, Schedulability analysis and stack size minimization for adaptive mixed criticality scheduling with semi-clairvoyance and preemption thresholds, J. Syst. Archit. 124 (2022) 102383.
[16] Z. Jiang, K. Yang, N. Fisher, N. Audsley, Z. Dong, Pythia-MCS: Enabling quarter-clairvoyance in I/O-driven mixed-criticality systems, in: 2020 IEEE Real-Time Systems Symposium, RTSS, IEEE, 2020, pp. 38–50.
[17] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, Elsevier, 2011.
[18] R. Electronics, Automotive Use Cases, https://www.renesas.com/eu/en/solutions/automotive/technology/safety.html.
[19] Xilinx official website, https://www.xilinx.com/.
[20] G. Plumbridge, Blueshell: a platform for rapid prototyping of multiprocessor NoCs, Comput. Archit. News (2014).
[21] Y. Li, M. Danish, R. West, Quest-V: A virtualized multikernel for high-confidence systems, in: RTSS, 2011.
[22] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, G. Buttazzo, Is your bus arbiter really fair? restoring fairness in AXI interconnects for FPGA SoCs, ACM TECS (2019).
[23] ARM, AMBA AXI and ACE Protocol Specification, ARM Ltd. 2012.
[24] R. Shaw, B. Jackman, An introduction to FlexRay as an industrial network, in: ISIE, 2008.
[25] Xilinx, Microblaze, https://www.xilinx.com/products/microblaze.
[26] A. Waterman, et al., The RISC-V instruction set manual, 2014, Volume I: User-Level ISA', Version.
[27] ARM official website, https://www.arm.com/.
[28] J. Anderson, A. Srinivasan, Mixed Pfair/ERfair scheduling of asynch-ronous periodic tasks, J. Comput. System Sci. (2004).
[29] K. Jeffay, S. Goddard, A theory of rate-based execution, in: Real-Time Systems Symposium, 1999.
[30] S.K. Baruah, V. Bonifaci, G. d'Angelo, A. Marchetti-Spaccamela, S. Van Der Ster, L. Stougie, Mixed-criticality scheduling of sporadic task systems, in: European Symposium on Algorithms, 2011.
[31] Bluespec, Bluespec System Verilog, https://bluespec.com.
[32] A. Silberschatz, P.B. Galvin, G. Gagne, Operating System Principles, John Wiley & Sons, 2006.
[33] FreeRTOS, FreeRTOS official website, http://www.freertos.org/.
[34] E. Monmasson, M.N. Cirstea, FPGA design methodology for industrial control systems—A review, IEEE Trans. Ind. Electron. 54 (4) (2007) 1824–1842.
[35] EEMBC, EEMBC benchmark, https://www.eembc.org/.
[36] S. Law, M. Bennett, S. Hutchesson, I. Ellis, G. Bernat, A. Colin, A. Coombes, Effective worst-case execution time analysis of DO178C level A software, Ada User J. 36 (3) (2015).
[37] A. Bellaouar, M. Elmasry, Low-Power Digital VLSI Design: Circuits and Systems, Springer Science & Business Media, 2012.

Zhe Jiang received his Ph.D. from University of York (2019). He is currently working as the system design engineer of Central Engineering Department in ARM Ltd and visit research associate in University of York. He is research interests include safety-critical system, system architecture, and system micro-architecture. He can be reached at: zhe.jiang@arm.com or zhe.jiang@york.ac.uk.

Kecheng Yang received the BE degree in computer science and technology from Hunan University in 2013, and the MS and PhD degrees from the University of North Carolina at Chapel Hill in 2015 and 2018, respectively. He is an assistant professor in the Department of Computer Science at Texas State University. His research interests include real-time systems and scheduling algorithms. He received an Outstanding Paper Award and the Best Student Paper Award at the 40th IEEE RTSS, and an Outstanding Paper Award at the 26th RTNS.

Nathan Fisher received the Ph.D. from the University of North Carolina at Chapel Hill in 2007, and M.S. degree from Columbia University in 2002, and the B.S. degree from the University of Minnesota in 1999, all in computer science. He is an Associate Professor with the Department of Computer Science, Wayne State University, Detroit, MI, USA. His research interests include real-time and embedded computer systems, sustainable computing, resource allocation, game-theory, and approximation algorithms. His current funded research projects are on composability of real-time applications, multiprocessor real-time scheduling theory, thermal-aware real-time system design, and algorithmic mechanism design in competitive real-time systems. Prof. Fisher was the recipient of the NSF CAREER Award in 2010 and the best paper awards from publication venues such as RTSS, ECRTS, and the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS.

Neil C. Audsley is a professor in the Department of Computer Science at University of York, where he leads a team researching Real-Time Embedded Systems. He is currently serving as the Head of Department of Computer Science. Specific areas of research include high performance real-time systems (including aspects of big data); real-time operating systems and their acceleration on FPGAs; real-time architectures, specifically memory hierarchies, Network-on-Chip and heterogeneous systems; scheduling, timing analysis and worst-case execution time; model-driven development. His research has been funded by a number of national (EPSRC) and European (EU) grants, including TEMPO, eMuCo, TouchHMore, MADES, JEOPARD, JUNIPER, T-CREST, DreamCloud and Phantom.

Zheng Dong received his BSc degree from Wuhan University, China, in 2007, an MSc from the University of Science and Technology of China, in 2011, and a Ph.D. degree from the University of Texas at Dallas, USA, in 2019. He is an assistant professor with the Department of Computer Science, Wayne State University, Detroit, Michigan. His research interests are in real-time embedded computer systems and connected autonomous driving systems. His current research focus is on multiprocessor scheduling theory and hardware–software co-design for real-time applications. He received the Outstanding Paper Award at the 38th IEEE RTSS. He is a member of the IEEE Computer Society.