A High-Resilience Imprecise Computing Architecture for Mixed-Criticality Systems

Zhe Jiang[†§], Xiaotian Dai[†*], Alan Burns[†], Neil Audsley[¶], Zonghua Gu[‖], Ian Gray[†]
[†]University of York, United Kingdom, [§]University of Cambridge, United Kingdom
[¶]City, University of London, United Kingdom, [‖]Umeå University, Sweden

I. Introduction

Mixed-Criticality Systems (MCS) are systems in which components can be developed under multiple assurance or criticality levels and integrated on a shared hardware platform. This is common in, for example, the automotive industry in which an Advanced Driver Assistance System (ADAS) may involve many modules developed under various criticality levels. The collision avoidance module will be high-criticality, whereas route-planning may be developed to a lower criticality. In this domain, criticality levels are defined as Automotive Safety and Integrity Levels (ASILs). Designing a modern complex embedded system in domains such as this is challenging because systems usually have to satisfy weight, power and cost (SWaP-C) requirements [2], producing a trend towards the integration of myriad software components and a huge driver towards mixed-criticality systems.

The most accepted approach is the widely studied dual-criticality MCS model (Vestal's model [3]). In this model, tasks are high-criticality and low-criticality (HI- and LO-tasks), and have estimated Worst-Case Execution Times (WCET)s with different confidence levels according to their criticality. The high-critical WCET (HI-WCET) is obtained using strict or formal methods, *e.g.*, static timing analysis, resulting in high confidence but often significant pessimism. On the other hand, low-criticality WCET (LO-WCET) may be obtained with measurement-based methods [3], which are more representative, but much less confident. To ensure timing correctness, it is necessary that: (i) if all tasks finish executing within their LO-WCETs, then they will all finish executing by their deadlines; (ii) if any task does not complete execution within its LO-WCET, then HI-tasks at least should complete execution by their deadlines [3].

Conventional dual-mode MCSs therefore define two *system modes*, LO-mode and HI-mode. In LO-mode (the optimistic mode) the system assumes that the execution time of every task (LO-task or HI-task) does not exceed its LO-WCET. If this assumption is violated, the system switches into HI-mode, in which it assumes the execution time of HI-tasks may exceed their LO-WCETs, but will not exceed their HI-WCETs [2].

Imprecise MCS. The key difficulty associated with conventional dual-mode MCS is the termination of the LO-tasks affecting system functionality [2], especially if the system still relies on the computation results of LO-tasks even in HI-mode. To extend the "life cycle" of the LO-tasks, a more practical model is possible: without dropping LO-tasks directly during mode switch, the system can continue to execute the LO-tasks as much as possible but with degraded computation precision, effectively accelerating LO-tasks' execution. Such a system model is called Imprecise MCS (IMCS) [4], [5]. In an IMCS, the imprecise computation tends to be used on Floating Point (FP) computations because: (i) Modern LO-tasks heavily rely on FP calculations, *e.g.*, image processing [6], signal processing [7], and Deep Neural Networks (DNNs) [6]; (ii) FP calculations consume significantly more clock cycles compared to other operations, which dominate the tasks' computation time [8].

This paper presents a new IMCS framework, *HIART*-MCS, which, mitigates the computation errors caused by imprecise computation for LO-tasks; achieves near-conventional MCS real-time performance for HI-tasks; and supports dynamically adjustable fine-grained levels of approximation of individual tasks at run-time without the need for software adaptations. Specifically, we present:

- A new FPU design, supporting approximation, accelerating *all* types of floating point computation.
- A novel system architecture that enables run-time configuration of software tasks' approximation degrees.
- A practical measurement-based method for exploring all potential configurations of the tasks in the system.
- A theoretical model, schedulability analysis, and optimisation approach, ensuring both the system's real-time performance and computation correctness.
- Comprehensive experiments examining the system under different configurations using various metrics.

II. *HIART*-FPU Overview

As task execution is affected by components at different system levels (language, compiler, OS, *etc.*), approximations can be potentially deployed at any system level. In *HIART*-MCS, we implement the approximation at the hardware level because of three reasons: (i) hardware-level approximation ensures *source compatibility* of software tasks, as the approximation is transparent to the software; (ii) hardware-level approximation occurs at run-time and so does not require input data to be known prior; (iii) hardware-level approximation can provide finer bit-width granularity, allowing run-time reconfiguration of the degree of approximation for each task.

[*]A full version of this paper was published in the IEEE Transactions on Computers (TC) [1].

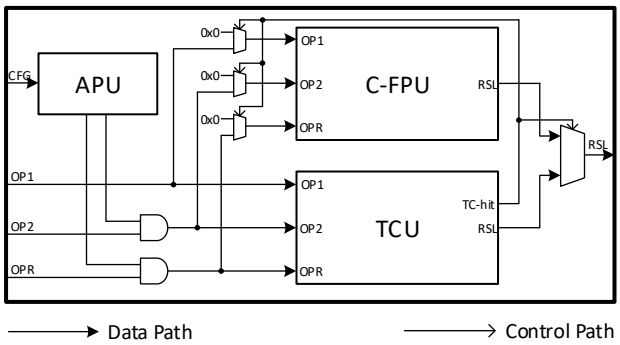


Fig. 1. Overview of *HIART*-FPU (CFG: configure interface).

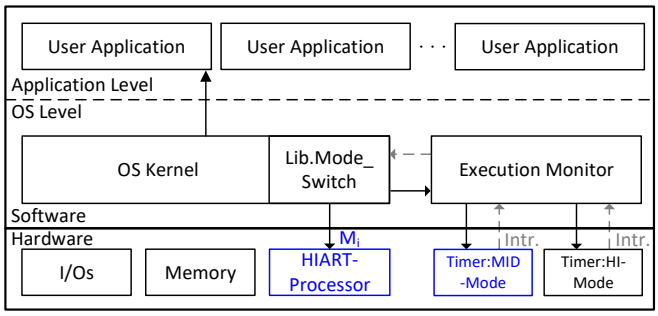


Fig. 2. System Architecture of *HIART*-MCS (The blue boxes show the new parts in *HIART*-MCS).

Top-level micro-architecture. The design has two features:

- **Run-time configurability**: The FPU can be configured at run-time to execute transparently with different approximation schemes. This is particularly important for IMCS.
- **FP-cache:** the FPU contains a dedicated FP-cache, recording the recent calculation results of the mantissa bits. During an FP-cache-hit, the FPU returns the mantissa bits' calculation in a single clock cycle, accelerating the FP computations. The benefits of this cache will be explained in the following sections.

We designed the C-FPU using three parts (see Fig 1): an Approximation Processing Unit (APU) that controls the approximation degrees of the operands; a Cached-FPU (C-FPU) and a Trivial Calculation Unit (TCU).

Computation Procedures. During an FP calculation, two operands (filtered by the APU) and one operator are first transmitted to the TCU. The TCU checks whether the calculation matches a Trivial Cases (TCs): if TC hits, the TCU directly returns the calculation results. Otherwise, the TCU generates a TC-miss signal to the C-FPU, requesting the C-FPU to proceed with the corresponding FP calculation.

III. System Architecture of HIART-MCS

Unlike the conventional dual-mode MCS framework, *HIART*-MCS uses three system modes:

- LO**-mode**: *HIART*-MCS initialises in LO-mode and stays in this mode if none of the HI-tasks exceeds LO-WCETs.
- MID**-mode**: when a HI-task (τ_i) overruns its LO-WCET (denoted as C_i^{LO}), at the moment of over-execution, the *HIART*-MCS immediately switches to MID-mode. In this mode, LO-tasks can still be executed with approximation, in which case their execution times (denoted as C_i^{AP}) should be less than C_i^{LO}.
- HI**-mode**: if the HI-task continues to overrun and exceeds a specific time point, *HIART*-MCS immediately switches to HI-mode while all LO-tasks are terminated. The time point triggering mode switch from MID-mode to HI-mode is termed MID-WCET (denoted as C_i^{MI}).

To enable these three system modes, we introduce architectural changes at both the hardware and the software levels. At the hardware level, we deploy our novel processor that supports FP imprecision (detailed previously in Sec. II) of the LO-task in the MID-mode. Also, we deploy two hardware timers to monitor the HI-tasks execution times for LO-WCETs and MID-WCETs.

The software-level structure is comprised of kernel and user spaces. In the kernel space, we present an implementation of both the *Lib.mode_switch* and the execution monitor. Specifically, we propose a new control function in the *Lib.mode_switch*, managing the approximation degree of LO-tasks in MID-mode. For the execution monitor, we operate the hardware timers during context switches corresponding to the new mode switch strategy. In user space, we need no changes to the original OS interfaces (see Fig. 2), thereby ensuring *source compatibility* and allowing tasks designed for a conventional MCS framework to be directly migrated.

Operation. At system initialisation, LO-tasks' approximation degree ($\mathcal{M}_i$) and HI-tasks' LO-WCETs and MID-WCETs are loaded. During context switches, the execution monitor suspends the timers of the currently executing task and then (re-)activates the timers for the next executing task to monitor the over-execution of LO-WCET and MID-WCET.

If a HI-task exceeds its LO/MID-WCET, the corresponding timer generates an interrupt to trigger a mode switch (to MID/HI-mode) by invoking *Lib.mode_switch*. Also, the system configures the approximation degree of LO-tasks at context switch if the system stays in MID-mode. The pseudo-code of the mode switch and context switch of *HIART*-MCS can be found in [8].

IV. Timing Analysis and Optimisation

To give an analytical bound for *HIART*-MCS, we introduce response time analysis (RTA) for a *HIART*-enabled platform in order to test the schedulability of given a taskset with constrained deadlines. With the introduction of the MID-mode, the model and analysis have to be tailored to reflect the new design. Specifically, the analysis for *HIART* uses a standard dual-criticality analysis, with an additional criticality level, and with the MID-mode execution times of LO-criticality tasks being *smaller* than that of LO-mode. Based on the schedulability analysis, the system's overall utility can then be optimised by applying either a fixed, or varying level of computational imprecision. This is discussed in detail in [1].

V. Experimental Evaluation

Platform setup. We built 4/8/16-core *HIART*-MCS variants on a Xilinx VC709 evaluation board. *HIART*|nc denotes the system without any FP-cache (which is the system framework presented in [8]), whereas *HIART*|N-way ($N \in \{2, 8\}$) contains an N-way FP-cache with 256 entries. The FP-cache address length is set to be 16, in which case only operands with fewer than 8 valid mantissa bits ($\mathcal{M}_i < 16$) will be buffered. This is because when $\mathcal{M}_i < 16$, the FP-cache hit rate decreases dramatically, thus the improvement from a cache-hit will be modest given the resources it would otherwise cost. We implemented the *HIART*-processors based on the SiFive Freedom E31, an open-source 32-bit RISC-V processor (5-stage pipeline). We implemented the proposed FPU (see Sec. II) in Verilog. The processors are connected to memory and I/O peripherals using a 5×5 mesh open-source NoC [9]. The hardware was synthesised and implemented by Xilinx Vivado (v2020.2). The software executing on the processors (OS kernels, drivers and user applications) was compiled using a RISC-V GNU toolchain. FreeRTOS (v.10.4) was the OS kernel for all processors, with the minimal modifications that were described in Sec. III.

Existing MCS frameworks usually implement run-time monitoring and mode switches either in the OS kernel (OSK) or a dedicated hypervisor (HYP). Therefore, we built two baseline systems (BS) on similar hardware platforms using conventional RISC-V processors. BS|OSK is a baseline MCS framework implementing execution monitoring and mode switches in OS kernels. BS|HYP is a baseline MCS framework using virtualisation technology, including real-time patches and I/O enhancement [10]. BS|HYP implements an execution monitor and mode switches in its hypervisor.All systems ran at 100 MHz because of the use of an FPGA as a prototyping.

A. Theoretical Evaluation

As one of the major design goals, the *HIART* should increase survivability of LO-tasks. To evaluate this, we used simulation based on synthetic tasksets using analytical evaluation. We defined an evaluation metric, survivability ($\mathcal{S}$), which is the percentage of LO-tasks that survive during an overload:

$$\mathcal{S} = \frac{\#\_of\_survived\_cases}{\#\_of\_all\_cases} \times 100\% \quad (1)$$

The task utilisation was generated with UUniFast, and half of the taskset was selected to be HI-tasks. In each trial, one of the HI-tasks was randomly selected to overrun and trigger a mode switch, with overrun execution time uniformly distributed from C_i^{LO} to C_i^{HI}. Task priority was assigned by deadline-monotonic priority ordering. We note that for the traditional dual-criticality model (without AP-mode), the survivability was 0% as all LO-tasks are dropped. Each utilisation consisted of 50 trials, and optimal $\mathcal{M}_i$ were selected.

Obs. 1. The proposed MCS model with the addition of the AP-mode significantly improved LO-tasks' survivability.

The observation can be seen in Fig. 3. The survivability of LO-tasks was significantly improved when utilisation was

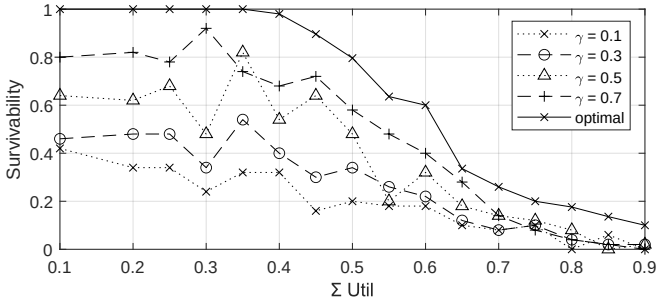


Fig. 3. Survivability of LO-tasks vs total utilisation

delrelatively low. For example, the survivability was around 100% when $\sum U_i \leq 0.35$ and was 60% when $\sum U_i = 0.6$. Survivability gradually decreased as utilisation increased, with survivability of 20% even when $\sum U_i \leq 0.8$. Eventually, survivability became close to 0, where nearly no system can maintain its schedulability. In general, a higher γ could prolong the AP-mode duration and thus improve survivability.

B. Real-time Performance and Computation Quality

As described in Sec. I, real-time performance and computation quality are the most important metrics for the IMCSs. In this section, we use real-world use cases to evaluate the real-time performance and computation quality of the systems.

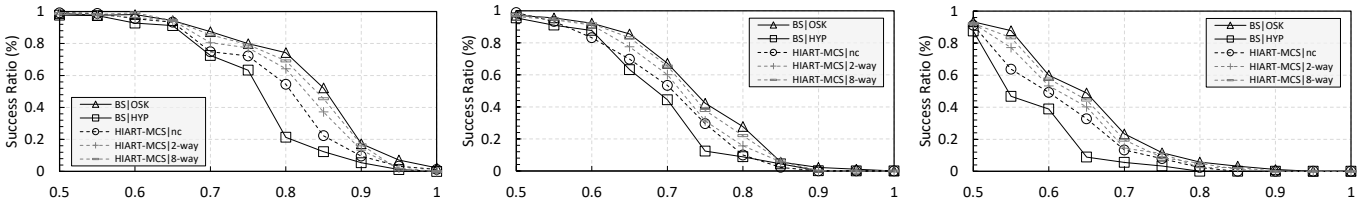
Hardware and software deployment. We configured the examined systems with 4/8/16 processors and deployed two sets of software tasks:

- 18 HI-tasks, selected from Renesas functional safety automotive use case database (*e.g.*, CRC and RSA32).
- 18 LO-tasks, including 6 DNN tasks, 6 image processing tasks, and 6 automotive function tasks selected from EEMBC benchmark.

In the LO-task set, the DNN and image processing tasks can be approximated at the MID-mode. The DNN tasks were classified into two categories, established upon LeNet-5 [11] and SqueezeNet (a variant of AlexNet) architectures. In each category, three tasks were respectively trained using MNIST, EMNIST and CIFAR-10 training datasets. The image processing tasks were implemented to perform Sobel filter, Canny filter, Scharr filter, Prewitt filter, Roberts filter and Sharpen filter on Oxford-IIIT Pet Dataset [12], respectively.

Experimental setup. In the experiments, the raw data for processing by the tasks was generated off-chip and sent to the evaluated systems via two Ethernet controllers (1 Gbps) at run-time. The experimental results were stored in the dedicated addresses of DRAMs. For each experimental setup, we executed the examined systems 200 times (500 seconds for each run) under varying target utilisation from 45% to 100%, with intervals of 5% increments.

Metrics. We examine the systems under each target utilisation using two metrics: (i) real-time performance; (ii) computation quality. For real-time performance, we used Success Ratio (SR) to report the percentage of the trials executed without any deadline miss of HI-tasks. For computation quality, we



(a) 4-core systems. (b) 8-core systems. (c) 16-core systems.

Fig. 4. HI-tasks: success ratio (x-axis: utilisation).

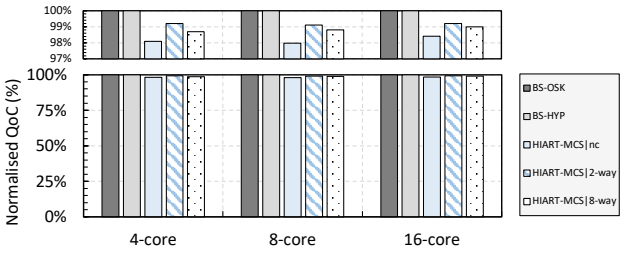


Fig. 5. System: average QoC.

examined the systems using system-level QoC to record the percentage of correct execution.

Obs. 2. IMCS decreased the HI-tasks' success ratio compared to the conventional MCS. This issue was effectively mitigated by *HIART*|2-way and *HIART*|8-way.

This observation is given by Fig. 4. When the systems were configured with the same settings (*i.e.*, core number and utilisation), *HIART*|nc suffered from a reduction of success ratio compared to the conventional MCS (BS|OSK), as the *HIART*|nc is designed upon BS|OSK but executes more tasks in a specific time period (*i.e.*, MID-mode). Unlike *HIART*|nc, *HIART*|2-way and *HIART*|8-way achieved similar success ratios as the BS|OSK. This is benefited by the deployment of the FP-cache (Sec. II), accelerating the tasks' executions at MID-mode.

Obs. 3. Although using imprecise computation in IMCS decreases system QoC, *HIART*|N-way could mitigate them.

As reported by Fig. 5, deploying imprecise computation in *HIART*|nc decreased the system-level QoC. Although such loss was caused by the nature of imprecise computing, introduction of FP cache accelerates the computation, and such brings the opportunity for finding more suitable parameters, including deferring switching points and with better approximation degrees of each LO-tasks.

VI. Conclusion

Imprecise Mixed Criticality (IMCS) has been recently studied as a more practical model than conventional dual-mode mixed criticality, because it could effectively improve the survivability of low-criticality tasks. The IMCS model raises three challenges to be solved. How to minimise the number of system errors caused by computational imprecision, how to minimise pessimism when compared with standard dual-mode mixed criticality, and how to adapt existing legacy systems and code to use the IMCS model. To solve these problems, we present a new processor design which implements transparent hardware-level computational approximation. The amount of approximation is run-time configurable and can be adjusted on a per-task basis to accelerate floating point computation. The cost of this approach is additional hardware area and power use. We propose a novel IMCS framework that takes advantage of this hardware support with an expanded system model, schedulability analysis, and optimisation approach that ensures both the system's real-time performance and minimises the impact of computational imprecision. As shown in our evaluations, the proposed system significantly extends the LO-tasks' survivability whilst ensuring that HI-tasks' real-time performance remains near to that of a conventional MCS.

For more details regarding the paper, please see [1].

References

[1] Z. Jiang, X. Dai, A. Burns, N. Audsley, Z. Gu, and I. Gray, "A high-resilience imprecise computing architecture for mixed-criticality systems," *IEEE Transactions on Computers*, 2022.

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

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

[4] L. Huang, I.-H. Hou, S. S. Sapatnekar, and J. Hu, "Graceful degradation of low-criticality tasks in multiprocessor dual-criticality systems," in *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, 2018, pp. 159–169.

[5] X. Gu and A. Easwaran, "Dynamic budget management and budget reclamation for mixed-criticality systems," *Real-Time Systems*, 2019.

[6] S. Bateni and C. Liu, "ApNet: Approximation-aware real-time neural network," in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018, pp. 67–79.

[7] M. Rosenthal, A. Weiss, and A. Mazloumian, "Realtime signal processing on embedded GPUs," in *Embedded Computing Conference (ECC2018), Winterthur, 5. Juni 2018*, 2018.

[8] Z. Jiang, X. Dai, and N. Audsley, "Hiart-mcs: High resilience and approximated computing architecture for imprecise mixed-criticality systems," in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2021, pp. 290–303.

[9] G. Plumbridge, J. Whitham, and N. Audsley, "Blueshell: a platform for rapid prototyping of multiprocessor nocs and accelerators," *ACM SIGARCH Computer Architecture News*, 2014.

[10] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in xen," in *2011 Proceedings of the Ninth ACM International Conference on Embedded Software*. IEEE, 2011.

[11] Y. LeCun *et al.*, "LeNet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, vol. 20, no. 5, p. 14, 2015.

[12] O. M. Parkhi, A. Vedaldi, A. Zisserman, and C. Jawahar, "Cats and dogs," in *2012 IEEE conference on computer vision and pattern recognition*. IEEE, 2012, pp. 3498–3505.