Shared Resource Contention in MCUs: A Reality Check and the Quest for Timeliness

Daniel Oliveira
Centro ALGORITMI, University of Minho, Guimarães, Portugal

Weifan Chen
Department of Computer Science, Boston University, MA, USA

Sandro Pinto
Centro ALGORITMI, University of Minho, Guimarães, Portugal

Renato Mancuso
Department of Computer Science, Boston University, MA, USA

Abstract

Microcontrollers (MCUs) are steadily embracing multi-core technology to meet growing performance demands. This trend marks a shift from their traditionally simple, deterministic designs to more complex and inherently less predictable architectures. While shared resource contention is well-studied in mid to high-end embedded systems, the emergence of multi-core architectures in MCUs introduces unique challenges and characteristics that existing research has not fully explored. In this paper, we conduct an in-depth investigation of both mainstream and next-generation MCU-based platforms, aiming to identify the sources of contention on systems typically lacking these problems. We empirically demonstrate substantial contention effects across different MCU architectures (i.e., from single- to multi-core configurations), highlighting significant application slowdowns. Notably, we observe that slowdowns can reach several orders of magnitude, with the most extreme cases showing up to a 3800x *(times, not percent)* increase in execution time. To address these issues, we propose and evaluate `µTPArtc`, a novel mechanism designed for Timely Progress Assessment (TPA) and TPA-based runtime control specifically tailored to MCUs. `µTPArtc` is an MCU-specialized TPA-based mechanism that leverages hardware facilities widely available in commercial off-the-shelf MCUs (i.e., hardware breakpoints and cycle counters) to successfully monitor applications' progress, detect, and mitigate timing violations. Our results demonstrate that `µTPArtc` effectively manages performance degradation due to interference, requiring only minimal modifications to the build pipeline and no changes to the source code of the target application, while incurring minor overheads.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases multi-core microcontrollers, shared resources contention, progress-aware regulation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2024.5

Supplementary Material
Software (Source Code): `https://github.com/danielRep/mcu-tpa-eval` [30]
archived at `swh:1:dir:ac8ce99c9ea41bf6dba2aaa197f4bda471aa5790`

Funding *Daniel Oliveira*: This work was supported by the Fundação para a Ciência e Tecnologia (FCT) within the Research and Development Units under Grant UIDB/00319/2020, and the Ph.D. Scholarship under Grant 2020.04585.BD.
Renato Mancuso: The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CNS-2238476. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

Acknowledgements We would like to express our gratitude to the reviewers for their valuable feedback and suggestions.

1 Introduction

In 2022, the microcontrollers (MCUs) market was valued at *USD 20.61 billion, with projections indicating a compound annual growth rate of 11.0% from 2023 to 2030* [36, 37, 45]. Meanwhile, there's a rising demand for MCUs with higher computing power, stemming from the proliferation and the range of functions required by Internet-of-Things (IoT) based devices. From communication and sensing [12] to machine learning algorithms [8], these devices are increasingly incorporating diverse and processing-intensive workloads [4]. To accommodate this demand, manufacturers are enhancing the architectures and microarchitectures of MCUs, notably adopting multi-core designs [1, 4, 28].

Predictability in MCUs. Unlike high-end systems powered by application processors (APU) found on desktops, servers, and mobile devices, MCUs, such as the Arm Cortex-M (CM) family, are the foundations of small embedded and IoT-based systems. MCUs are characterized by limited computing power and memory capacity, typically featuring simple microarchitectures without cache, having 2-3 pipeline stages, and lacking virtual memory support [48, 51]. For instance, the Armv6-M CM0+ and Armv8-M CM23 processors have 2-stage instruction pipelines, whereas the Armv7-M CM3/4 and Armv8-M CM33 include 3-stage pipelines. Additionally, they are designed with low interrupt latencies; for instance, CM-based devices typically range from 12 to 16 cycles. As such, MCUs are perceived as deterministic platforms due to their simplistic hardware designs, aiming to guarantee the timely behavior of deployed applications. This predictability is vital since MCUs frequently interface with the physical world and host applications with real-time and safety-critical requirements [12, 18, 34]. However, as extensively studied in high-end APUs, the introduction of multi-core in MCUs leads to unpredictability [17] due to the reciprocal interference in shared resources [19, 22, 25, 31, 49, 52].

The need for new solutions. Shared resource contention in high-end multi-core APUs has been extensively studied [22, 23]. Caches, interconnect networks, and the DRAM memory controller and banks are among noteworthy sources of interference [25, 52, 53]. However, MCUs present unique characteristics that have not been thoroughly studied so far: MCUs often lack a memory management unit (MMU); no shared caches are used, and only very few employ caches in the form of private caches with no hardware coherence; use flash memories with non-standardized controllers offering caching/prefetching capabilities; include different peripherals; and incorporate multiple Direct Memory Access (DMA) engines. Yet, empirical studies addressing contention in MCUs remain scarce. To the best of our knowledge, only two academic works have tackled this issue [31, 34]; however, their insights do not apply to a broad spectrum of MCUs since they target a specific platform (i.e., Armv8-M Musca-A1 test chip). From a different angle, contention within bus interconnects has been observed in various MCUs and leveraged to implement a side-channel attack [38]. Moreover, MCU contention is also challenging the industry; for instance, a recent study on a dual-core Arm Cortex M0+ (i.e., Raspberry Pi Pico) disclosed a priority inversion scenario resulting from contention on shared memory [49].

Timely progress assessment. To address interference, researchers have proposed different approaches (e.g., memory bandwidth regulators [53], cache coloring [19]). These techniques rely on hardware features unique to APUs and absent in MCUs (e.g., performance counters[1].

[1] Recently, Armv8.1-M-based CM55 and CM85 announced a Performance Monitoring Extension, but at the time of writing boards equipped with these processors were still not available.

and virtual memory). In another line of works [6, 21], timely progress assessment (TPA) is leveraged to live-monitor contention-induced delays and restore the system's predictability. TPA-capable systems perform a timeliness check when the program counter (`PC`) of the application under test (AUT) reaches a predefined address. Then, the system evaluates the AUT's timely progress and adjusts the resource allocation accordingly (e.g., suspending co-runner CPUs). Existing TPA methods still face a series of trade-offs: whether the source code of the AUT is required, whether the AUT needs modification, to what extent specialized hardware is necessary (e.g., trace units), and how much overhead it incurs. Nonetheless, we posit that an aptly-designed TPA-based approach is well-suited to tackle performance interference issues in MCUs. Indeed, using simple, architectural-defined hardware components (e.g., hardware breakpoints) offers a more straightforward and scalable implementation of TPA techniques compared to other, more complex approaches in the literature [6, 21].

In this paper, we shed light on the "elephant in the room" [5] for MCUs by posing two pivotal questions: *(1) How significantly does contention affect the MCU landscape?* and *(2) How can timeliness/predictability be restored in MCUs?*. Our approach to answering these questions is twofold: (i) through an extensive, empirical investigation, we seek to identify and understand the sources and effects of contention on MCUs; subsequently, (ii) we propose `µTPArtc`, a novel TPA-based mechanism specifically tailored to tackle this issue and restore timeliness in MCUs. We initially surveyed leading MCU vendors. Based on this survey, we document contention issues in three Commercial Off-The-Shelf (COTS) MCUs from distinct silicon manufacturers featuring single vs. dual-core configurations and platform-specific memories. When contention was observed, the performance degradation reached several orders of magnitude; surprisingly, one platform displayed slowdowns up to 3800x.

The proposed `µTPArtc`. The novelty of `µTPArtc` design lies in a new trade-off among the existing methods: (i) it employs a novel milestone selection algorithm that can be used without access to the source code; (ii) it only requires that `µTPArtc` is added at linking time before deployment on the AUT; (iii) it leverages ISA-defined, readily available hardware components found in MCUs; and (iv) it incurs minimal overhead. To achieve this, `µTPArtc` encompasses two steps and corresponding subsystems: (i) an offline profiling phase that constructs a timing reference profile of the AUT through an automated selection of progress milestones and (ii) an online monitoring mechanism that assesses the timely progress of AUTs at each milestone, ensuring the preservation of end-to-end timeliness. `µTPArtc` live-monitors the AUT by leveraging widely available hardware breakpoint primitives across the MCU landscape (e.g., Arm's CM family, RISC-V debug module IP). Upon reaching a milestone, the CPU is interrupted, and a debug exception is triggered, allowing `µTPArtc` to conduct TPA within the corresponding exception handler.

We implemented a prototype of `µTPArtc` on a state-of-the-art Armv8-M MCU platform, i.e., NXP LPC55S69 featuring a dual-core Arm CM33. Implementation-wise, `µTPArtc` is developed as a static library and the binary size is notably small ($\approx$ 1.3KiB). Our evaluation shows that, in general, `µTPArtc`'s overhead is tightly coupled with the number of progress milestones monitored and their granularity; nevertheless, `µTPArtc` is capable of balancing this interplay efficiently, with results showing that overhead averages around 1% for different reference application benchmarks (i.e., Embench suite). Furthermore, our results demonstrate that `µTPArtc` is able to control performance degradation within a predefined, controlled threshold even when severe contention is introduced.

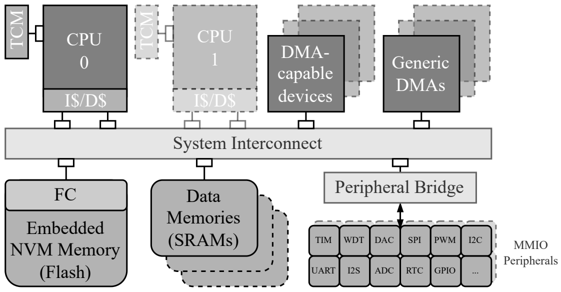


Figure 1 Generic model for an MCU-based platform, characterized by (i) single-/dual-core configuration (occasionally integrated with TCMs); (ii) several DMA controllers; (iii) a myriad of peripherals; (iv) flash and SRAM memories for instruction fetching and data, respectively; (v) and a system interconnect, typically based on a multi-layer AHB matrix.

Contributions. In summary, this paper makes four major contributions. **First**, we survey COTS MCUs and provide a generalized architectural model. With that, we identify the potential points of contention. **Second**, we empirically demonstrate the magnitude of contention on three representative, commercially available MCUs. **Third**, we propose a new hardware-assisted TPA-based mechanism called `µTPArtc` to manage contention and provide a reference implementation. And **fourth**, we conduct a comprehensive evaluation of `µTPArtc`'s performance and incurred overhead.

2 Motivation: The Contention Problem in MCUs

To shed light on the contention problem in MCUs, we conducted an empirical evaluation to provide compelling evidence of the existence of interference sources and their significant impact on performance. Given MCUs' distinct architectural characteristics when compared to high-end APU systems, our investigation is organized into four key steps:

(1) **Scoping and modeling MCU architectures.** Aiming to define the scope of our evaluation, we conducted a survey of current COTS MCU platforms, documenting their architecture and hardware features (Table 1). Based on this systematized information, we developed a generic MCU architectural model (Figure 1) as the basis for our study.

(2) **Identifying potential sources of contention.** Using the model formulated in (1), we discuss and characterize which shared resources may be prone to contention, while highlighting their differences from other high-end APUs.

(3) **Designing resource-stressing scenarios.** We select three distinct platforms for our evaluation: (i) two representative of the most recent multi-core MCUs and (ii) one exemplifying a mainstream single-core MCU (Figure 2). For each target MCU, we mount different system configurations designed to stress-test contention on the resources identified in (2).

(4) **Assessing contention impact.** We evaluate the impact of interference on MCUs by using the previously devised setups. We provide empirical evidence of how a seemingly inincuous system misconfiguration or a deliberate attack aimed at maximizing contention can lead to performance slowdowns from 2x up to 3800x.

2.1 Scope and Platform Model

COTS platforms. We target MCU-based platforms specifically designed for embedded systems and IoT devices. Arm's CM family is the dominating 32-bit MCU architecture, with almost 30 billion chips shipped in 2021 [2]. In Table 1, we present a selection of 17

Table 1 Available MCU-based COTS platforms on the low-end market segment.

Class	Platform			32-bit Cortex-		Memory				DMA	Peripherals (subset)	Sys. Bus
	vendor	*series*	*acron*	*CPUs*	*cache (I/D)*	*flash*	*ACC*	*SRAM*	*TCM*			
tier1	ST	STM32C0x	stm1	M0+ (42MHz)	–	32KiB	●	6KiB	–	1	GPIO, ADC, USART, SPI, I2C, WDT, Timers	M-AHB
	Renesas	RA2E2	ren1	M23 (48MHz)	–	64KiB	●	8KiB	–	1	GPIO, ADC, USART, SPI, I3C, WDT, Timers, AES, TRNG	M-AHB
	NXP	LPC860	nxp1	M0+ (60MHz)	–	64KiB	●	8KiB	–	1	GPIO, ADC, USART, SPI, I2C, WDT, Timers	M-AHB
tier2	Infineon	XMC4200	inf1	M4 (80MHz)	–	256KiB	●	40KiB	–	1	GPIO, ADC USART, CAN, Timers, SPI, I2C, I2S, CAN, USB	M-AHB-L
	ST	STM32L4	stm2	M4 (80MHz)	–	1MiB	●	128KiB	–	2	GPIO, ADC, Timers, USART, SPI, CAN, I2C, WDT, RNG	M-AHB
	NXP	KV30F	nxp2	M4 (100MHz)	–	128KiB	●	16KiB	–	1	GPIO, ADC, USART, SPI, I2C, WDT, Timers	N/P
tier3	NXP	Kinetis K26	nxp3	M4 (180MHz)	8KiB	2MiB	●	256KiB	–	1	ADC, Timers, USB, USART, GPIO, I2C, SPI, CAN, I2S,	M-AHB
	ST	STM32F7	stm3	M7 (216MHz)	2x 16KiB	2MiB	●	512KiB	128KiB	1	GPIO, UART, CAN, Timers, USB, ADC, I2C, SPI, Crypto	M-AHB
	Renesas	S7G2	ren2	M4 (240MHz)	–	4MiB	●	64KiB	–	1	GPIO, Eth, SPI, CAN, ADC, Crypto, HMI, USB, I2C, USART	N/P
nextgen	Raspberry	RP2040	rpb1	2xM0+ (133MHz)	–	16MiB	●	264KiB	–	1	GPIO, UART, SPI, I2C, USB, Timers, WDT	M-AHB-L
	Infineon	CY8C63x6	inf2	M4 (150MHz) & M0+ (100MHz)	–	1024KiB	●	288KiB	–	2	Crypto, Timers, I2C, SPI, UART, I2S, SD, USB, BLE, GPIO	M-AHB
	NXP	LPC55S6x	nxp4	2xM33 (150MHz)	–	640KiB	●	320KiB	–	2	AES, SD, USB, GPIO, SPI, I2C, UART, Crypto, Timers, I2S	M-AHB
	Nordic	nRF54H	nrd1	?xM33 (320MHz)	–	2MiB	N/P	1MiB	–	N/P	CAN, BLE, ADC, I3C, USB	N/P
	Infineon	TRAVEO T2G	inf3	2xM7 (320MHz) & M0+ (100MHz)	M7:2x 16KiB	6336KiB	●	640KiB	M7:128KiB	3	Crypto, CAN, UART, I2C, SPI, SD, Timers, GPIO, I2S, Eth	M-AHB & AXI-I
	ST	STM32H7	stm4	M7 (480MHz) & M4 (240MHz)	M7:2x 16KiB	2x 1MiB	●	1MiB	M7:192KiB	2	I2C, USART, SPI, SD, CAN, USB, Eth, Timers, Crypto, LCD	M-AHB & AXI-I
	NXP	i.MX RT1180	nxp5	M7 (800MHz) & M33 (240MHz)	M7:2x 32KiB M4:2x 16KiB	N/P	N/P	1.5MiB	M7:512KiB M33:256KiB	2	UART, WDT, Timers, Crypto, SD, GPIO, USB, I3C, CAN	N/P
	NXP	i.MX RT1170	nxp6	M7 (1GHz) & M4 (400MHz)	M7:2x 32KiB M4:2x 16KiB	N/P	N/P	2MiB	M7:512KiB M4:256KiB	2	UART, WDT, Timers, Crypto, SD, GPIO, USB, I3C, CAN	N/P

We have classified well-established MCU platforms into three classes based on their price per ≈10,000 units: *tier1* (<1$), *tier2* ([1,10[$), and *tier3* (≥10$). The *nextgen* class refers to next-generation multi-core MCUs, some still in pre-production. *N/P*: not provided. *ACC.*: flash acceleration (e.g., cache, prefetchers). *M-AHB*: multilayered AHB matrix. *M-AHB-L*: M-AHB Lite. *AXI-I*: AXI Interconnect. *acron.*: acronym used across the document.

COTS MCU families, with a focus on the industry leaders by sales (i.e., NXP, ST, Infineon, and Renesas) [45]. Among the surveyed vendors, it is noteworthy that, in comparison to well-established platforms (i.e., *tier1,2,3*), the next-generation platforms (i.e., *nextgen*) are consistently moving towards adopting multi-core configurations. In contrast, traditional platforms feature simple, single-core architectures without cache or virtual memory support, and with elementary 2-3 stages pipelines (e.g., *stm1/2*, *ren1/2*, *nxp1/2*, *inf1*). Fewer platforms feature longer pipelines, optional caches (e.g., *stm3*), and special memory controllers (e.g., *nxp3*) for performance purposes. Notwithstanding, *nextgen* devices are standing out with their adoption of multi-core architectures, featuring CPUs with higher clock frequencies (e.g., *nxp6* goes up to 1GHz), along with the inclusion of instruction and data caches (e.g., *nxp5* features 2x 32KiB caches). Simultaneously, memory capacities are increasing, while tightly-coupled memories (TCM) are becoming ubiquitous. Additionally, the number of DMAs is also rising, and at the microarchitecture level, AXI interconnects are being placed alongside the conventional AHB system buses to meet the increasing bandwidth demands.

MCU characteristics and platform model. Figure 1 depicts the MCU model, generalized from the unique characteristics presented in Table 1. MCUs feature single- to dual-core configurations and optionally incorporate instruction/data caches and TCMs with zero-wait states. They often support peripherals with DMA capabilities (e.g., Ethernet, USB), as well as generic DMA bus masters. Key architectural components, such as CPUs, memories, DMA controllers, and peripherals, are interconnected usually by a multi-layer AHB matrix. Flash memories are used for code and read-only data, while SRAMs are used for data. DRAMs are generally not used in MCUs. Due to the relatively slow speed of flash, most devices

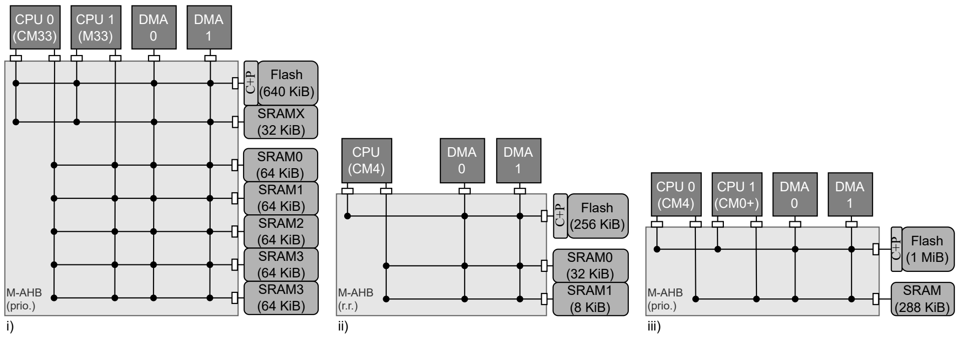


Figure 2 Block diagram of selected platforms from Table 1. (i) *nxp4* (SRAMX is reserved for code); (ii) *stm2* (SRAM1 is reserved for code); (iii) *inf2*.

feature acceleration mechanisms within the flash controller (FC) to expedite flash memory access (e.g., small internal caches and prefetchers). SRAM memories are faster memories and typically apply very few wait-states. A myriad of peripherals (e.g., timers, SPI, I^2C, UART, CAN controllers, etc.) is connected through a bridge module (e.g., AHB-to-APB bridge) that interfaces the system bus with a slower bus connected to each memory-mapped peripheral. Across the spectrum, MCUs adhere to this architectural model.

2.2 Potential Sources of Contention

MCU-based systems are perceived as immune to interference effects, while high-end APU systems face recognized challenges in dealing with interference in real-time applications [22]. As previously mentioned, high-end systems experience interference effects precisely in components typically absent from MCUs (e.g., shared caches, interconnect networks, and DRAM memory controller and banks [9,10,22]). However, as shown in Table 1, recent *nextgen* MCUs exhibit two notable trends that could serve as sources of contention: (i) a higher number of integrated bus masters capable of creating interference exacerbated by the integration of dual-core configurations and additional DMAs, all connected to a typical bus with round-robin arbitration policy; and (ii) the enhancement of memory resources, such as FCs being equipped with internal caches and prefetchers, to address growing performance demands.

Memory contention. Unlike APUs that predominantly utilize DRAM for main memory, MCUs rely on flash and SRAM memories. Flash memory serves as non-volatile storage, from which code and read-only data are directly fetched when firmware/applications are executed. Known for its slower access times, FCs typically implement proprietary, black-box architectures [15]. Moreover, to cope with rising CPU frequencies, the number of wait-states that cause CPU idling introduced during slow flash reads is becoming prohibitive [11]. A common solution to this issue is to include a zero-wait state cache on the FC that ensures immediate access to the most recently used instruction(s). However, these caches are typically small and implement unusual organizations seldomly found in high-end L1/L2 caches. For example, ST's flash ART accelerator does not cache the most recently fetched instructions. Rather, it caches only the 16 bytes at the target of a recent branch instruction [11, 47]. Consequently, while wait states may delay the initial branch to a specific address, subsequent branches to the same address do not experience the same delay. On the other hand, SRAM memories are typically single-ported with an arbiter in place to handle collisions [38].

Table 2 Memory allocation layouts and system configuration for the *common* and *catastrophic* experiments. In the *catastrophic* configuration, we use two DMAs.

(a) *Common* experiment configuration.

		read-only		*read+write*	
test	board	CPU0	CPU1	CPU0	CPU1
cmn1	*nxp4*	Flash		SRAM0	SRAM1
cmn2	*nxp4*	Flash		SRAM0	
cmn3	*inf2*	Flash		SRAM	

(b) *Catastrophic* experiment configuration.

		read-only		*read+write*				
test	board	CPU0	CPU1	CPU0	CPU1	DMAs-src	DMAs-dst	BP (High Prio.)
ctt1	*nxp4*	Flash		SRAM0		Flash	SRAM0	CPU0
ctt2	*nxp4*	Flash		SRAM0		Flash	SRAM0	CPU1, DMAs
ctt3	*inf2*	Flash		SRAM		Flash	SRAM	CPU0
ctt4	*inf2*	Flash		SRAM		Flash	SRAM	CPU1, DMAs
ctt5	*stm2*	Flash	-	SRAM0	-	Flash	SRAM0	N/A

DMAs-src and *DMAs-dst* denote the DMAs source and destination buffer's memory locations, respectively. *N/A*: not applicable, since *stm2* applies fixed round-robin arbitration. *BP*: bus prioritization.

Bus contention. As depicted in Figure 1, memories and peripherals are interconnected to the CPU(s) and DMA(s) controllers through a system bus. A multi-layer AHB matrix is the most widespread bus topology. It allows different bus masters to execute several non-blocking, full-bandwidth transfers to non-shared slaves concurrently. However, when multiple masters attempt to access the same slave, the bus must arbitrate them, which inherently introduces contention [38]. Typically, the bus arbitration policy is round-robin. Still, recent platforms are implementing priority-based policies: *nxp4* and *inf2* provide a priority-based policy, while *stm2* adopts a round-robin approach. Attached to the system bus, more and more MCUs support both general-purpose and I/O-specific DMA controllers (e.g., Ethernet modules, SDMMC card interfaces, USB drivers, etc) that autonomously access memory without the intervention of the CPU. As observed in high-end APUs, such devices can further exacerbate the load on the system bus and thus worsen contention effects [33, 55].

2.3 Methodology and Experimental Setup

Our series of experiments conducted on three MCUs aim to illustrate the impact of contention on performance due to shared hardware resources. We devised two evaluation scenarios, as outlined in Table 2: *common* and *catastrophic*. Each experiment implies different system configurations that result in either manageable or severe contention-related delays.

Selected platforms. Based on the potential sources of contention described in Subsection 2.2, we have selected for our empirical study three distinct platforms: *nxp4* [29], *inf2* [14], and *stm2* [47]. These platforms are representative of a subset of different MCU-based architectural organizations, including (i) single- or dual-core designs; (ii) distinct ISAs and processors (Armv6-M CM0+, Armv7-M CM4, and Armv8-M CM33); (iii) bus topologies with differing arbitration policies (round-robin or priority-based); (iv) different memory organizations (e.g., vendor-specific FCs, single- vs. multi-block SRAMs); and (v) implementation-specific DMA controllers with different functionalities. In Figure 2, we illustrate a simplified version of the main block elements of each selected platform architecture (i.e., bus masters and slaves, and bus interconnect). More details on each platform can be found in Table 1.

System configuration. During the evaluation of each experimental scenario, we made an effort to maintain the consistency of the platform's configuration. This included (i) configuring CPU clocks at their maximum frequencies, which inherently required the maximum number of wait states in each platform's FC, and (ii) activating any platform-specific performance-enhancing features available for the FC, such as small internal caches and prefetchers. Table 2 summarizes the memory layout of each system configuration used in the two experiments.

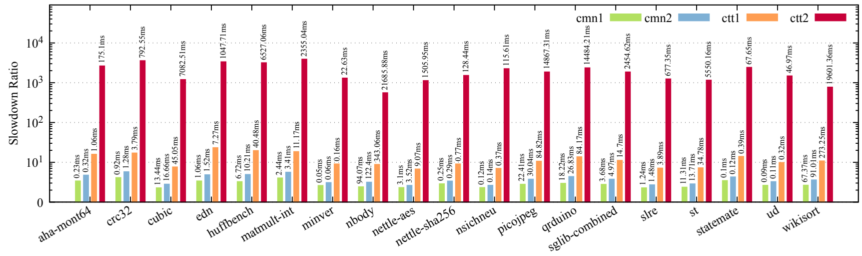
The observed CPU (i.e., *CPU0*) runs an Embench's benchmark [32], while the co-runner (i.e., *CPU1*), as the interfering CPU, runs a synthetic memory-intensive application. This synthetic application is designed to stress the target shared memory resources by performing branch-intensive operations of multiple non-sequential blocks of `nop` instructions. We emphasize that while we have deliberately parameterized the interference application to create a significant level of contention, the observed effects can possibly be aggravated with further fine-tuning.

Experimental methodology. The evaluation is designed to assess performance degradation in the CPU executing the Embench benchmark. The degradation arises due to contention from other bus masters, such as additional CPUs or DMAs, that concurrently compete for the same hardware resources used by the CPU under test. In the first experiment, i.e., the *common* scenario – see Table 2(a), we instantiate a typical multi-core memory layout on the *nxp4* and *inf2* boards. The configuration separates read-only sections (code in `.text` and constants in `.rodata`) from read+write sections (`.data`, `.heap`, and `.stack`) segments depending on the available memory units on each platform. As illustrated in Figure 2, the *inf2* board is limited to a single code and data memory; hence, in our experiments, memory contention arises from both flash and SRAM (*cmn3* configuration). In contrast, the *nxp4*, with its multiple SRAM memories, opened up the possibility of performing two distinct experiments (*cmn1* and *cmn2*). In both experiments, the CPUs' read-only segments are stored in flash memory. Then, their read+write segments are either split between two SRAMs (*cmn1*) or combined into a single SRAM memory (*cmn2*).

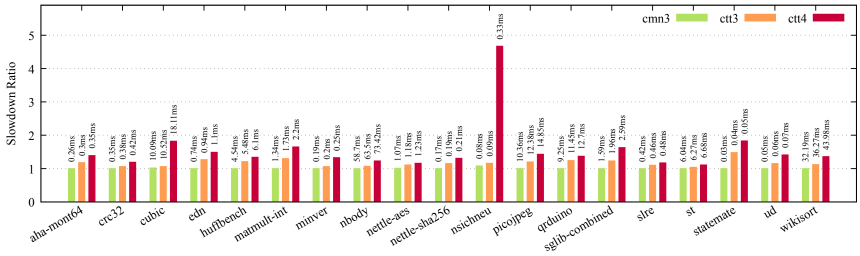
In the second experiment, i.e., the *catastrophic* scenario – see Table 2(b), we aim to demonstrate the most extreme form of performance degradation resulting from contention. We adopt a victim-vs-attacker system model, where the attacker, operating from a secondary, compromised CPU, aims to disrupt the performance execution of the victim running in the primary CPU. The attacker has full access to the DMAs interface and the bus arbitration; however, it cannot directly use platform-enabled functions to stall/pause the victim's CPU (e.g., *CPU1* in *nxp4* cannot reset or disable *CPU0*'s clock [29]). Therefore, the attacker configures the system to intentionally (i) use DMAs in a way that stresses the victim's access to its assigned memories and (ii) prioritize all other bus masters (i.e., *CPU1* and DMAs) over the victim's CPU. This scenario is demonstrated in the *ctt(1-5)* configurations. For *nxp4* and *inf2*, we showcase the resulting contention from (i) *CPU1* and DMAs interference (*ctt1* and *ctt3*), while *CPU0* is configured with the highest priority bus mater in the BP and (ii) the inverse configuration, i.e., *CPU1* and DMAs have the highest priority (*ctt2* and *ctt4*). In the case of *stm2*, only DMAs are enabled, as the platform has a fixed round-robin arbitration policy.

Result validity. The conducted experiments involved two researchers who independently carried out each experiment. To maintain consistency between each configuration, both researchers have used the same evaluation framework. Notwithstanding, we argue that some results could present inconsistencies when repeated. Therefore, we strongly encourage other independent users to validate them through the framework that we are open-sourcing[2]. While we made efforts to select a diverse range of platforms across the MCU spectrum, we acknowledge that the results in this study might not immediately generalize to other MCU platforms. MCUs are increasingly heterogeneous in their architectures, and even seemingly similar platforms can yield different results due to their unique microarchitectures.

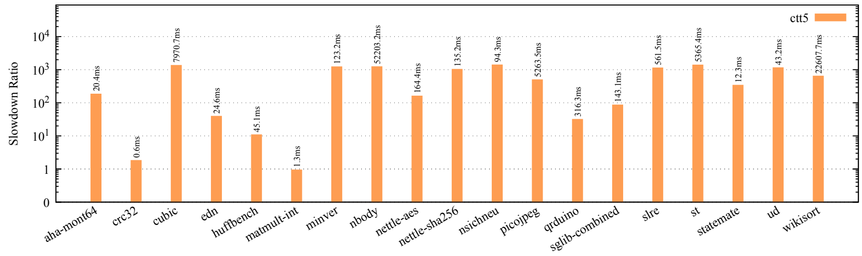
[2] GitHub repository: `https://github.com/danielRep/mcu-tpa-eval`



(a) Multi-core MCU: *nxp4* with 2xCM33 (logarithmic scale).



(b) Multi-core MCU: *inf2* with 1xCM4 and 1xCM0+.



(c) Single-core MCU: *stm2* with 1xCM4 (logarithmic scale).

Figure 3 Slowdown ratios across platforms and scenarios, normalized to the baseline execution time of each benchmark running in single-core.

2.4 Evidence Results

Degree of contention-induced delays in nextgen MCUs. Figure 3a and Figure 3b depict the slowdown ratio of each benchmark running atop the selected *nextgen* MCUs, i.e., the *nxp4* and *inf2*, respectively. In the *common* experiment, both platforms exhibit varying degrees of impact across three configurations: (i) *cmn1* and *cmn2* show significant performance slowdown on *nxp4* (ranging from 2.27x to 5.77x); (ii) *cmn3* results in negligible impact on *inf2*, with a maximum of 10% degradation in the *nschineu* benchmark. Regarding the *catastrophic* experiment, the results are, again, orders of magnitude different on each platform. The *inf2* exhibits a maximum slowdown of 4.67x (*nsichneu* benchmark in *ctt4*), whereas *nxp4* experiences an almost complete stall of *CPU0*, with execution time reaching 3800x in the worst case (*matmult-int* benchmark in *ctt2*).

▶ **Finding 1.** Contention-induced slowdowns in MCUs are a reality, with effects varying widely across platforms and configurations, sometimes leading to the complete CPU stall.

Influence of FC's architecture on contention. The obtained results indicate that contention is predominantly rooted in shared flash memory, influenced by unique properties of the FC. For example, the *inf2*'s FC provides an 8KiB instruction cache for each CPU, while *nxp4* implements a notably smaller cache. This variance in cache size is evident in the *cmn3* results of *inf2*, where only the *nsichneu* benchmark exhibits contention since its code size surpasses the cache capacity (≈15KiB of code). This stands in contrast to the other benchmarks, with an average code size of ≈2KiB, fitting within the cache and thus avoiding any performance degradation due to cache misses. Moreover, *nxp4*'s uses a priority-based bus that, by default, configures *CPU0* with a higher priority than the co-runner *CPU1*, suggesting that the primary contention source is the FC rather than the shared bus.

▶ **Finding 2.** Flash memory is acutely susceptible to contention, with the FC specific characteristics, notably internal cache sizes, playing a pivotal role on the extent of slowdowns.

DMAs add fuel to contention effects. In the *catastrophic* experiments, the DMAs are enabled and configured to continuously move data between two memory buffers located in *CPU0* assigned memories (from flash to SRAM). On *nxp4* and *inf2* platforms, the benchmark performance drops significantly, showing an increase in memory contention when DMAs create contention on such memories (e.g., in *ctt1*, a 23x performance slowdown is observed in the *edn* benchmark). Moreover, this can be even aggravated if we change the bus arbitration policies (*ctt2* and *ctt4* configurations). Configuring *CPU0* as the low-priority bus master has a catastrophic impact on benchmark performance, particularly in the *nxp4* platform.

▶ **Finding 3.** Enabling DMAs significantly increases memory contention, causing notable performance drop across platforms. Adjusting bus priorities exacerbates the issue.

Contention in classical single-core MCUs. In Figure 3c, the *catastrophic* experimental results for the *stm2* reveal significant slowdowns, reaching up to 1390x, leading to near CPU starvation due to intensive memory access by both DMAs. The *stm2*'s FC includes a 1KiB instruction cache and a 256B data cache for fetching instructions and read-only constants, respectively. CPUs fetch instructions and literal pools (constant/data) from flash memory through dedicated buses in the FC (I-code and D-code, respectively); DMAs also utilize the D-code bus. According to the platform's manual [47], D-code accesses are prioritized over I-code accesses to limit the time lost due to fetches from literal pools that lead to stalls during the execution stage of the CPU pipeline. However, the continuous data requests from the two DMAs saturate the D-code bus, meaning that the CPU can only fetch instructions during the intervals when the DMA transaction is restarted. We conducted additional experiments to probe this behavior further. These tests involved running a sequence of 100 `nop` instructions while varying the number of active DMAs and FC's acceleration settings, such as enabling instruction or data caches. The results indicate that a single active DMA incurs an overhead of approximately 30%. With the activation of a second DMA, the CPU experiences starvation unless either the flash instruction or data cache is activated, which aligns with the results observed in *ctt5*.

▶ **Finding 4.** Contention is also present in single-core MCUs, where DMAs commonly operate alongside the CPU. Our observations reveal that flash memory constitutes the primary bottleneck, with certain configurations resulting in significant performance slowdowns.

3 µTPArtc: A MCU-powered TPA-based Mechanism

In the previous section, we empirically demonstrated how contention arises in single- and multi-core MCUs when bus masters concurrently access shared memory resources. Following that study, we present a solution to address those contention issues in this section. We propose `µTPArtc`, a µcontroller-specialized, `TPA`-based `runtime` control mechanism designed to ensure timeliness specifically in multi-core environments. µTPArtc monitors an application's timely progress in run-time, and in the presence of contention delays, it takes proper regulatory actions in the secondary, interfering CPU.

3.1 Background

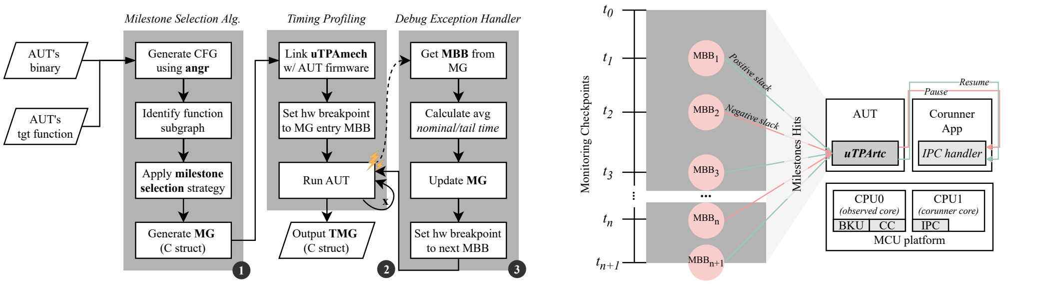
Timely progress assessment (TPA). The term TPA is initially introduced in Chen et al. [6], "TPA refers to the ability of a system to live-monitor the positive/negative slack – with respect to a known reference – at key milestones throughout an application's lifespan." The *milestones* could be instructions or source code at specific addresses or lines. When the program counter (PC) reaches a milestone, a.k.a., a *milestone hit*, it can be concluded that the application has made some sizable progress. For example, the exit point of a loop could be a milestone candidate. Upon a milestone hit at runtime, the system emits various information, including the wall clock time. Kritikakou et al. [20, 21] firstly design and implement TPA-like mechanism. One important usage of TPA is to enforce the application's *timely progress integrity* (TPI). The latter captures the idea that, as long as the progress rate of an application throughout its lifespan is sufficient, execution can finish before the deadline. Several works have demonstrated the effectiveness of TPA in enforcing TPI [6, 20, 21, 43]. All TPA methods start by analyzing the control flow graph (CFG) of the AUT. The following points provide a brief overview of CFGs.

Basic block and branch instructions. A *basic-block* (BB) in this work refers to a consecutive series of non-branching assembly instructions, concluding with a branching instruction. This means that, except for the final instruction, every other instruction in the BB increments the PC to point to the next sequential instruction. The number of target BBs of a branch instruction varies depending on the type. Unconditional branch instructions have one target, for example, B and BL in Armv7/8-M. Conditional instructions (e.g., B.BEQ) have two targets. The targets of return instruction (e.g., RET) can be enumerated by inspecting its call sites.

Control flow graph (CFG). The control flow transfer within a program can be modeled as a directed graph $\mathcal{G} = (V, E)$. A node in $\mathcal{V}$ corresponds to a BB, and an edge $(u, v) \in \mathcal{E}$ represents that the ending branch instruction in u has v as a target. For the purpose of TPA, it is unnecessary to construct the complete CFG. Instead, a partial CFG for a function of interest func is constructed as func-CFG. func-CFG has the following property: for $(u, v) \in$ func-CFG, if u ends with a function call instruction, then v is the BB starting with the function call return address, instead of the actual branch target BB of that call.

3.2 System Overview

Figure 4 presents the µTPArtc system divided into its two main subsystems: (i) an offline profiling tool that automates the selection of milestones and adds reference timing information, and (ii) an online monitoring mechanism that, at each milestone and employing TPA, evaluates the application's timely progress, ensuring that end-to-end timeliness is preserved. µTPArtc's online monitoring mechanism conducts TPA by setting hardware breakpoints on milestones, and the corresponding exception handler is programmed for TPA logic. The logic includes checking whether the current progress is ahead or behind the reference timing to take proper regulation actions and updating the breakpoints for the next milestones.



(a) Offline profiling tool flow chart. **(b)** Online monitoring mech. architecture overview.

Figure 4 System overview of µTPArtc, consisted by the profiling tool and monitoring mechanism.

3.2.1 Offline Profiling Step

The TPA logic executed in the debug exception handler is not part of the application logic. In other words, a milestone hit would incur an overhead. Thus, the overhead might become excessive if the selected milestones are hit too frequently. If the selected milestones are too "close" to each other, µTPArtc might not be counterproductive. When this is the case, it is beneficial to de-select some of them to reduce overhead. On the other hand, if the hits are too sparse, µTPArtc could be ineffective due to coarse monitoring granularity. So, choosing an appropriate set of milestones is essential in conducting effective TPA. The milestone selection is an optimization problem and a challenging one at that. In this work, we provide an initial milestone selection algorithm that is not meant to be the absolute best. When used in conjunction with profiling, the proposed algorithm produces satisfactory solutions.

Initial milestone selection. Intuitively, an application spends most of its time executing loops. Thus, given a function CFG, e.g., `main`-CFG, two types of location could be good candidate milestones. The first type is the return address of function calls in `main` because these functions usually take a sizable portion of the execution time. The second type is the successor(s) of a *strongly connected component* (SCC) because the control flow might iterate through the loops in the SCC, taking a sizable amount of time. The set of successors of an SCC is the union of all successors of nodes in the SCC minus all nodes in the SCC. The initial milestone selection algorithm works as follows: (i) given the AUT binary, construct the `main`-CFG. We use the open-source `angr` [42] tool. (ii) Color the entry node, the return BBs of function calls, and successors of SCCs red. Color all the other nodes white. (iii) Remove all the white nodes and connect the red nodes according to the topology of the `main`-CFG. This is done by removing each white node and directly connecting all the edges from its predecessors to its successors. The resulting directed graph is a *milestone graph* (MG). Nodes in the MG form a complete set of milestones for the AUT. At runtime, when a milestone is hit at node u, the next milestone is guaranteed to be one of the successors of u in the MG.

Timing profiling. Given an edge (u, v) in MG, define a random variable *transfer time* T_{transfer} for the time it takes for the execution to transfer from u to v without other milestones hit in between. Although, by construction, an edge in the MG indicates that the execution will encounter functions or loops during the transfer, the actual T_{transfer} might still be too short or too large, leading to the aforementioned overhead/granularity problems. Timing profiling tackles this issue and adds reference timing information in the MG. The first step is to gather the timing information. This is done in three steps. (i) The MG information is stored as a struct in `C` and compiled/linked to the firmware; (ii) the hardware breakpoint is set to the starting address of the entry node in MG; (iii) the application is run multiple times. Upon a milestone hit, the exception handler logs the current wall clock time and configures the next set of milestones to monitor according to the MG.

Timed milestone graph (TMG). After milestone timing has been gathered, per-edge and per-node timing information, i.e., *nominal time* $T_n(u, v)$ and *tail time* $T_t(u)$ respectively, are added to the MG. $T_n(u, v)$ is the average transfer time for edge (u, v), indicating the expected time the application needs to transfer from one milestone to the other. The time elapsed between the application's start time and the last time u is hit is a random variable. The tail time $T_t(u)$ is the maximum of said random variable. This term can be experimentally derived (as in our evaluation) or computed using static analysis. The tail time is necessary complementary information to nominal time because the latter only expresses the expected transfer time but not how many times the transfer should occur. Consider a loop. At runtime, each iteration could be timely from the perspective of nominal time. However, the number of iterations might be significantly larger than those observed during the profiling phase. In this scenario, checking against tail time could spot a potential TPI violation. The nominal time is also used to refine the graph and enhance milestone granularity; however, it is imperative to balance granularity with the potential runtime overhead introduced, which we discuss further in Section 4.1. When $T_n(u, v)$ is notably large, the refinement logic depends on the edge type. If u ends with a function call, the function can be analyzed by the profiling tool in the same way, and the resulting function graph can be merged into the upper-level graph. If v is the successor of an SCC, more milestones can be placed inside the SCC, and a new iteration of the timing profiling will be performed. Conversely, if the nominal time is too small, the associated milestones can be removed to reduce overhead. The final satisfactory MG with nominal and tail time embedded within is called *timed milestone graph* (TMG). For the AUTs in this work, the final refinement is small: after the initial selection, this step only adds/removes a couple of milestones.

3.2.2 Online Monitoring Mechanism

Consider an AUT that runs on an MCU with other co-runners. `µTPArtc` conducts online monitoring to pause/resume the co-runner(s) and mitigate contention based on the delta between the actual runtime and the reference time. `µTPArtc` implements a controlled degradation mechanism similar to the one proposed in [6]. The same symbols are chosen to stay consistent. While the operational principle is similar, recall that `µTPArtc` does not rely on the presence of a trace unit nor uses a secondary CPU. As such, `µTPArtc` is not subject to the problem of trace blackout that was a key limiting factor in [6].

Two times are tracked during the runtime: (i) the *actual time* $\Theta(i)$ and (ii) the *running nominal time* $N(i)$. Let $\texttt{MBB}_i$ be the milestone reached with the i-th hit. $\Theta(i)$ is updated with the current time when the i-th hit occurs. Therefore, it indicates the time between the first milestone hit and the i-th milestone hit. The running nominal time $N(i)$ is updated as $N(i) = N(i-1) + T_n(\texttt{MBB}_{i-1}, \texttt{MBB}_i)$. When the i-th milestone hit occurs, $\min(T_t(\texttt{MBB}_i), N(i))$ indicates the expected timely behavior when the AUT runs in isolation. If co-runners are active, the actual time is usually larger than expected due to contention. Thus, for controlled degradation, $\alpha > 1$ is introduced to provide a *set-point* for the timely behavior of the AUT. Upon the i-th milestone hit, the slack is calculated as:

$$slack(i) = \alpha \min\{T_t(\texttt{MBB}_i), N(i)\} - \Theta(i) \tag{1}$$

Table 3 Hardware primitives support for Arm-based architectures and *nextgen* platforms. Filled circle: supported. Empty circle: not supported.

Hardware Primitives		**Arm Architectures**							**Available**
ISA-	*Platform-*	*v6-M*		*v7-M*			*v8-M*		***nextgen***
defined	*defined*	CM0	CM0+	CM3	CM4	CM7	CM23	CM33	**Platforms**
CC		●[1]	●[1]	●	●	●	●	●	all
BPU		○[2]	○[2]	●	●	●	○[2]	●	all, except *rpb1*
	IPC				–				all

1) While DWT cycle counter is not supported, the system timer can be used (SysTick) in these processors.
2) BPU is supported, but not accessible by the CPU only the debug access port.

If a negative slack is detected, i.e., the timely progress falls behind, the `µTPArtc`'s exception handler pauses the co-runners. This allows the application to recover some slack. When enough positive slack is accumulated, the co-runner(s) can be resumed. An *aggressiveness* parameter $\beta \in [0, 1]$ is introduced to decide when to resume. The co-runners are resumed when $slack(i) > \beta\alpha N(i)$. A smaller β causes the resume to be as early as possible. A larger β causes the regulation to be more conservative.

3.3 System Implementation

Hardware primitives and support. `µTPArtc` is designed for MCU-based platforms, and our current prototype implementation targets the Armv8-M *nxp4* platform. To conduct TPA, `µTPArtc` resorts on three hardware primitives (Figure 4b): (i) a Breakpoint Unit (BPU) for setting hardware breakpoints at each milestone and redirect execution to `µTPArtc`; (ii) a Cycle Counter (CC) to serve as a wall-clock timer; and (iii) an Inter-Process Communication (IPC) module as a way to intervene with the co-runner upon detecting a TPI violation. Table 3 summarizes the availability of each hardware primitive across Arm's CM-based processors and the *nextgen* platforms listed in Table 1. These features are widely available in MCU-based platforms, with both the BPU and CC typically defined at the Instruction Set Architecture (ISA) level. In Arm architectures, the BPU is implemented as the Flash Patch and Breakpoint (`FPB`) unit, which exposes hardware breakpoint functionality via, typically, 8 comparators [41]. The FPB implemented by the *nextgen* platforms features 8 comparators, which, from our tests, is typically sufficient for monitoring the usual number of milestone edges, i.e., 2 to 3 edges. When an instruction fetch matches a comparator's address, the core halts or a debug exception is triggered – `µTPArtc`'s runs the monitoring logic in this exception handler. Moreover, most CM-based processors implement a 32-bit CC in the Data Watchpoint and Trace (DWT) unit. However, for those processors that do not support DWT, the 24-bit system timer (i.e., SysTick) can serve as an alternative. In another example, RISC-V-based platforms feature a system CC and address/data match triggers to implement hardware breakpoints. Additionally, we use a platform-specific IPC module (Inter-CPU Mailbox in the case of *nxp4*) to send pause/resume commands to the co-runner.

Profiling and TMG construction. The offline profiling tool requires two inputs: (i) the unmodified binary of the AUT, which is analyzed by a Python script that utilizes `angr` to implement the milestone selection algorithm; and (ii) AUT's function of interest, for which a CFG is to be constructed. Figure 4a shows `µTPArtc`'s sequence of operations. Initially, in the *milestone selection algorithm* step (❶), the AUT's binary is analyzed, and an MG is

Algorithm 1 µTPArtc's monitoring logic (Arm-based implementation).

```
input : Current MBB address, addr
input : Timed Milestone Graph, TMG
begin Run µTPArtc debug monitor handler
    i ← Get current MBB id based on addr, TMG(addr)
    Θ_i ← Get actual time from DWT cycle counter
    N(i) ← Calculate running nominal time
    slack(i) = α min(T_t(MBB_i), N(i)) − Θ(i) // Calculate slack;
    if slack ≥ 0 then // Positive slack -> no TPI violation
        if Corunner = SUSPENDED then
            if slack > (βαN(i)) then
                Send IPI to the co-runner to resume
                Corunner ← RUNNING
    else // Negative slack -> TPI violation
        if Corunner = RUNNING then
            Send IPI to the co-runner to suspend
            Corunner ← SUSPENDED
    end
    FPB ← Clean MBB_prev edges breakpoints // Reconfigure hw breakpoints;
    FPB ← Set next MBB_i edges breakpoints
end
```

generated, following the strategy outlined in Subsubsection 3.2.1. Then, µTPArtc's profiler (❷) is integrated into AUT's firmware by modifying the debug exception vector table entry and linking the system's library to the executable. During runtime, the profiler starts by configuring a hardware breakpoint at the MG's $\texttt{MBB}_0$ entry point. This involves programming the $\texttt{MBB}_0$ address into an FPB's instruction address comparators. A debug exception is triggered when the PC reaches the generic $\texttt{MBB}_n$. Within the *debug exception handler* (❸), the $\texttt{MBB}_n$ responsible for the exception is identified. Then, the wall-clock time is read from the DWT cycle counter, and the *nominal* and *tail* times are calculated; the MG's $\texttt{MBB}_n$ entry is then updated accordingly. Before handling control back to the AUT, the profiler sets new hardware breakpoints for all the MBBs that can be directly reached from $\texttt{MBB}_n$. The analysis is repeated across multiple runs of the AUT. Finally, the tool outputs the TMG. The TMG comprises an array of `mbb_t` structures. This structure contains information about each milestone: (i) entry point address, (ii) tail time, and (iii) a list of edges to adjacent milestones, each with the edge's nominal time and index of the target milestone.

Monitoring logic. Algorithm 1 illustrates µTPArtc monitoring handler. Before entering the monitoring logic, µTPArtc resorts to a pre-handler routine that recovers the PC value from the *exception stack frame*[3]. Using the PC address, the $\texttt{MBB}_i$ that caused the exception is identified. Then, the *actual time* Θ_i is read from the cycle counter, and the running nominal time $N(i)$ is calculated. Next, the slack is evaluated (Equation 1), and according to this value, two operations can be performed: (i) if the slack is positive, no TPI violation is detected, and the co-runner is resumed based on the β parameter; (ii) if the slack is negative, the co-runner is signaled to suspend operations. Before returning from the handler, the previously-set hardware breakpoints are cleared, and the successor edges of $\texttt{MBB}_i$ are configured as the new set of breakpoints in the BPU, preparing the system for the next monitoring checkpoint.

[3] On CM architectures, the CPU automatically pushes a number of registers to stack memory on exception entry.

Table 4 System binary size and run-time overhead. *Ovhd (Δ%)* refers to the run-time overhead percentage of `µTPArtc` (T_{time}) compared to the baseline execution time.

Benchmark	Total MBBs in TMG	Monitoring Checkpoints		Binary Size (bytes)			Run-time Overhead (cycles)				
		Total Hits	*Max. (<1%)*	*TMG*	*Total*	*µTPArtc (Δ%)*	*min*	*max*	*mean*	T_{time}	*Ovhd (Δ%)*
cubic	13	38	47	1872	43316	**7.23%**	122	190	155	5889	**0.66%**
nbody	8	213	310	1152	32328	**7.31%**	122	172	151	32094	**0.55%**
nettle-aes	11	40	11	1584	44180	**6.34%**	121	151	134	5221	**2.62%**
huffbench	15	13	17	2160	42644	**8.11%**	107	151	128	1662	**0.54%**
st	10	405	42	1440	33816	**7.95%**	120	168	150	60871	**8.28%**
matmult-int	8	7	5	1152	40408	**5.76%**	107	139	130	909	**1.00%**
wikisort	9	20	220	1296	42392	**5.86%**	124	171	143	7006	**0.17%**

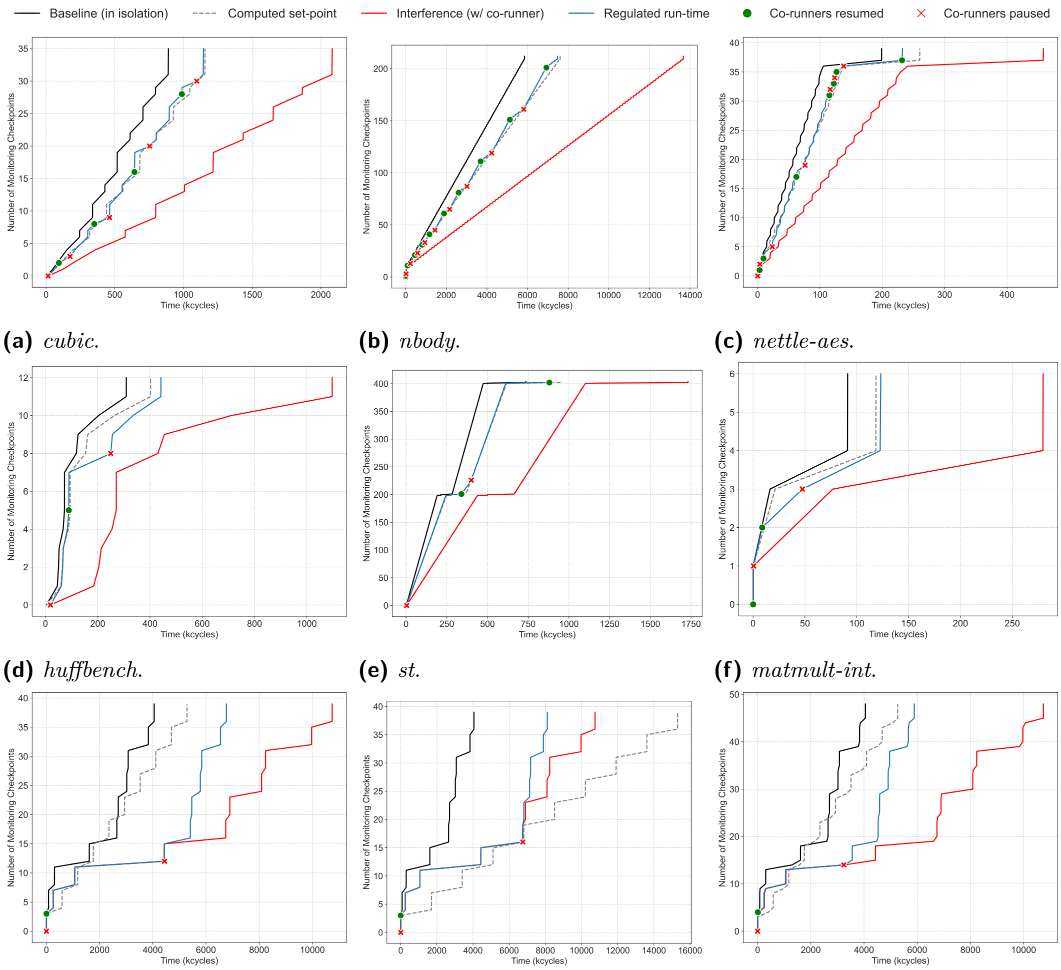
4 Evaluation

Our evaluation of `µTPArtc` was conducted on the *nxp4* platform using the same framework mentioned in Section 3. Firstly, we evaluate the run-time overhead imposed in a `µTPArtc`-enabled system and measure the memory footprint of `µTPArtc`. Following this, we assess the mechanism's ability to mitigate contention effects, aiming to control the performance degradation of a target application within an acceptable slowdown threshold.

4.1 Performance Overhead and Memory Considerations

Monitoring checkpoints overhead. `µTPArtc` conducts TPA by preemptively interrupting the core each time a milestone is reached, a routine that we refer to as a *monitoring checkpoint*. This process inherently incurs a run-time overhead. As a result, increasing the number of monitoring checkpoints leads to larger cumulative overheads. If the overhead grows too large, it might exacerbate contention delays but also impair `µTPArtc`'s ability to enact effective TPA without fully halting the co-runner. This interplay highlights the challenge of balancing the granularity of milestone placement with the imperative to maintain manageable overhead levels, ensuring that `µTPArtc` interventions are both effective and as minimally intrusive as possible. Table 4 reports the variable impact of `µTPArtc`'s run-time overhead across different benchmarks, with overheads ranging from a minimal 0.17% in *wikisort* to a moderate 8.28% in *st*. Noteworthy, benchmarks such as *st* and *nettle-aes* exhibit the highest overheads, primarily due to the number of monitoring checkpoints surpassing the pre-computed maximum threshold intended to limit impact to less than 1% (as seen in the *Total Hits* column surpassing the *Max (<1%)*) – see Section 5. This observation underscores the importance of strategic milestone placement within `µTPArtc` to mitigate unnecessary performance degradation while ensuring effective TPA.

Memory footprint. The `µTPArtc` is developed as a static library, allowing it to be directly linked with a target application. The binary size of `µTPArtc` is notably small, approximately 1.3KiB when compiled with `-O2` optimizations (GNU toolchain version 11.2.1.). The configuration requires a C source file that describes the TMG through a data structure, which can increase the binary size (around 150 bytes of data is allocated for each milestone defined). According to Table 4, which details the binary size of various setups used within the controlled performance degradation use-case (Subsection 4.2), the increase is between 6% to 8%.



(a) *cubic.* **(b)** *nbody.* **(c)** *nettle-aes.*

(d) *huffbench.* **(e)** *st.* **(f)** *matmult-int.*

(g) *wikisort.* **(h)** *wikisort* (w/ max. T_n). **(i)** *wikisort* (w/ fine granularity).

Figure 5 Selected benchmarks behavior when regulated by `µTPArtc`.

4.2 Use-case: Controlled Performance Degradation

Experimental setup. We selected *nxp4* based on its observed significant performance degradation in the *common* experiment, making it an ideal target for assessing `µTPArtc`'s ability to restore timeliness. To recap, the *common* experiment applies a typical multi-core setup, where the interplay between CPUs accessing common memory resources generates contention (i.e., the flash and SRAM). The observed primary CPU runs the target benchmark, while the second CPU runs the synthetic memory-intensive application used in Section 2. In all the performance tests, both CPU clocks are configured at a 150MHz maximum frequency, and the flash acceleration is enabled to speed up its performance.

Benchmark selection. The benchmarks from Embench chosen for our use-case study are reported in Table 4. With each `µTPArtc` intervention consuming an average of 142 cycles (0.95µs), we pinpointed 10 benchmarks where adding at least one monitoring checkpoint results in no more than 1% of total overhead. Due to limitations in space, three benchmarks – *qrduino*, *picojpeg*, and *sglib-combined* – were further excluded from this selection.

Algorithm 2 `WikiSort` benchmark marked with MBB placement.

```
begin Run wikisort benchmark
    ← MBB_1
    Create data array to be sorted (D⃗)
    Initialize funcGenPointers (F⃗) with different data generator functions
    Initialize the random-number generator
    foreach Function f in F⃗ do
        ← MBB_2
        foreach Index d in D⃗ do
            d ← Generate data by calling f(d)
        end
        ← MBB_3
        Call WikiSort(D⃗)
        ← MBB_4
    end
    ← MBB_5
end
```

Results overview. Figure 5 depicts the behavior of the selected benchmarks when regulated by `µTPArtc`, which strives to maintain their timely behavior with a set-point of $\alpha = 1.3$. This regulation entails intervening in the AUT's operation – pausing or resuming interfering CPUs – when contention occurs, thereby allowing the AUT to recover slack. We analyze the AUT's progress in three scenarios: (i) operating in isolation (*Baseline (in isolation)*), (ii) under the impact of interference from a co-runner (*Interference w/ co-runner*), and (iii) when regulated by `µTPArtc` (*Regulated run-time*). Additionally, we display the computed set-point to explain `µTPArtc`'s regulation actions at each monitoring checkpoint (*Computed set-point*). The y-axis chronologically details the timeline of each monitoring checkpoint, whereas the x-axis indicates the time interval between these checkpoints. Furthermore, the figure reports the decisions to pause (red $\times$) or resume (green ●) the co-runner made by `µTPArtc` based on its slack monitoring. Co-runners are resumed promptly as soon as the AUT recovers some slack ($\beta = 0.05$).

Evaluation results. As depicted in Figure 5, in the first three cases, i.e., *cubic*, *nbody*, and *nettle-aes*, `µTPArtc` successfully enforces a runtime very close to the set-point. As soon as negative slack is detected, a corrective adjustment of the co-runner is performed, allowing for the recovery of performance slack. For *huffbench*, a minor deviation from the baseline is observed at the final checkpoint. Increasing milestone granularity between checkpoints 7 and 8 could improve regulation, avoiding the notable slowdown observed in this interval; `µTPArtc` only applies a corrective measure in 8 after *huffbench* has been substantially affected by contention. Regarding *st*, `µTPArtc` aligns the regulated performance with the set point. However, as reported in Table 4, *st* exhibits a large 8.28% overhead. This excessive overhead arises from a milestone positioned within a loop, prompting `µTPArtc` to intervene 405 times – far exceeding the threshold established for maintaining overhead at 1% (specifically, 42 hits). To address such scenarios, we suggest implementing a countdown counter that selectively bypasses monitoring checkpoints in extensive loops, as detailed in Section 5. The *matmult-int* benchmark could also benefit from a thin granularity between checkpoint 2 and 3; nevertheless, as soon as `µTPArtc` pauses the co-runner in 3, the performance is prevented from dropping further, and *matmult-int* recovers close to the set-point.

The *wikisort* demonstrates certain limitations of `µTPArtc`. *Wikisort* is based on a block merge sort algorithm, as detailed in Algorithm 2. It consists of a two-level nested loop. The inner loop prepares the data, and the outer loop executes the `WikiSort` algorithm for each prepared data set. The execution time of each inner and outer loop iteration varies, depending on the data type to be generated (e.g., random, equal, etc.) and the time elapsed in the `WikiSort` function. Consequently, T_{transfer} for ($\texttt{MBB}_2$,$\texttt{MBB}_3$) and ($\texttt{MBB}_3$,$\texttt{MBB}_4$) has a large variance, which causes the nominal and tail time to no longer be accurate representations of the true behavior. This inaccuracy makes `µTPArtc` overly conservative. The intersection between the baseline and progress set-point in Figure 5g illustrates one such case. To mitigate the issue, one could use the maximum T_{transfer} to calculate the nominal time. When doing so, as shown in Figure 5h, said intersection disappears. Nonetheless, the set-point still appears far from the actual timely behavior due to the overestimation of the nominal time. In Section 5, we propose a strategy for addressing timing imbalances in loops. Additionally, the granularity-overhead trade-off problem (see Section 3.2.1) also appears at the 11th milestone hit. The flat slope here suggests coarse monitoring. An attempt is made to add one more milestone in the outer loop, resulting in a much improved albeit imperfect TPI as shown in Figure 5i. When more milestones are placed in the inner loop for better TPI, the large number of iterations results in impractical overhead, a scenario also noted in *st*.

5 Discussion

Timing high variance nominal in loop. The variance problem exhibited in *wikisort* shows only one possible case of complex control flow that the proposed method cannot satisfactorily solve. Although Kritikakou et al. [20, 21] provide a treatment for the timing variance, this work adopts Chen et al. [6]'s TMG to store timing information, which does not provide a treatment yet. Thus, we extend the TMG to provide a solution for the *wikisort*-like problems. The following notations are used to better explain the peculiarity of *wikisort*. Given an edge, let $\mathcal{T}$ denote the set containing all the measured T_{transfer} for the edge throughout the profiling phase. $t_{i,j} \in \mathcal{T}$ represents the measured T_{transfer} for the j-th iteration of the edge during the i-th run of the AUT. The offending edge whose transfer time exhibits high variance, in *wikisort*, is part of a loop, and its nominal time represents the expected time for the `WikiSort` function call. At profiling, the number of iterations is found to be precisely 9. Define a set $P_k = \{t_{i,j} \in \mathcal{T} | i = k, j \in \mathbb{N}^+, 1 \leq j \leq 9\}$, the nine measurements for the k-th run. If the variance $\sigma^2(P_k)$ of P_k is high, neither the average nor the maximum of P_k is suitable for nominal time. Interestingly, let m be the total number of profiling runs and define $P'_k = \{t_{i,j} \in \mathcal{T} | i \in \mathbb{N}^+, 1 \leq i \leq m, j = k\}$, all the measured transfer times for the k-th iteration in the loop across all the runs. The data indicates $\sigma^2(P'_k)$ is sufficiently small for all k. The implication is that given the k-th iteration of the loop, the average $\mu(P'_k)$ of P'_k is a suitable representation of the timely behavior. Thus, instead of a single nominal time, a list of nominal times $[\mu(P'_1), ..., \mu(P'_9)]$ should be added to the edge. Then, `µTPArtc` chooses the corresponding nominal time and conducts TPA based on the current iteration. Two lessons are learned: (i) when profiling a loop, one must check whether $\sigma^2(P_k)$ is sufficiently small; (ii) by extrapolation, a non-loop edge can also be associated with a set P'_1. When $\sigma^2(P'_1)$ is sufficiently small, the nominal time is an accurate representation. Last but not least, (iii) instead of solely relying on the variance, the profiling phase should also consider the range of $\mathcal{T}$ to improve the mechanism's reliability.

Reducing overhead for loop with large iterations. *st* exhibits a relatively large overhead (8.28%) due to one milestone in a loop being hit 405 times, as shown in Table 4. The overhead can be reduced by not conducting TPA every time the milestone is hit. Upon a hit, the overhead is the sum of (a) entering/leaving the exception handler and (b) TPA/TPI logic. The overall overhead per hit is 150 cycles. However, the measured overhead for (a) is only 15

Table 5 Gap analysis table of mitigation techniques for shared resources interference.

Ref	Year	Platform			Shared Resource	Mitigation Technique	Hw Support
		Target HW Platform	*ISA*	*SoC*			
[3]	2019	Raspberry Pi 3[1]	Armv8-A	4xCortex-A53	LLC	CC	PMC
[6]	2023	Xilinx UltraScale+	Armv8-A/-R	4xCortex-A53 & 2xCortex-R5	LLC	TPA	PMC, TU
[7]	2023	Zynq Ultrascale+	Armv8-A	4xCortex-A53	IRQ	IRQ Coloring	PMC
[26]	2020	Zynq Ultrascale+	Armv8-A	4xCortex-A53	LLC	CC	MMU
[53]	2013	Desktop Machine[1]	Intel x86	Core2Quad Q8400	Bus & DRAM	MBR	PMC
[50]	2016	Odroid XU4	Armv7-A	4x Cortex-A7 & 4x Cortex-A15	LLC	CC	MMU
[24]	2013	Pandaboard	Armv7-A	2xCortex-A9	LCC	CC & Cache Locking	MMU & Cache Lockdown
[52]	2014	Desktop Machine Freescale P4080	Intel x86 PowerPC	Xeon W3530 8xe500mc	DRAM	Bank-aware Mem. Allocator	MMU
[27]	2018	Raspberry Pi 2	Armv7-A	4xCortex-A7	LLC & DRAM	MBR & CC	MMU, PMC
[19]	2019	Nvidia Tegra TX1	Armv8-A	4x4xCortex-A57	LLC & DRAM	DRAM & CC	Virtualization Extensions
[56]	2023	Zynq Ultrascale+	Armv8-A	4xCortex-A53	DRAM	MBR	PMC, CoreSight
[44]	2020	NXP S32V234	Armv8-A	4xCortex A53	DRAM	MBR	DRAM
[39]	2022	NXP S32V234	Armv8-A	4xCortex A53	DRAM	MBR	QoS-PU & PMC
[55]	2022	Zynq Ultrascale+	Armv8-A	4xCortex-A53	DRAM	IO-related MBR	QoS-PU
[34]	2019	Musca-A1	Armv8-M	2xCortex-M33	Flash & SRAM	Mem. Static Allocation	-
[31]	2023	Musca-A1	Armv8-M	2xCortex-M33	Flash & SRAM	-	-
Our work	2024	LPC55S69	Armv8-M	2xCortex-M33	Flash & SRAM	TPA	FPB, DWT

[1] Here, we have only detailed a subset of the platforms tested. *CC*: Cache coloring. *MBR*: Memory bandwidth regulation. *MMU*: Memory management unit (i.e., virtual memory) *QoS-PU*: Quality-of-Service profiling units.

cycles. Thus a counter-down counter can be defined in the handler for the milestone, such that only if the counter value equals to zero, the TPA would be executed and the counter is replenished, otherwise it merely decrements the counter and exit the handler. For example, if the replenishment value is 81, the number of fully handled TPA events for the same milestone would be reduced from 405 to 5.

Extending `µTPArtc` regulation to DMAs. In Subsection 2.4, the *catastrophic* scenario illustrates how DMAs can be instrumental in exacerbating contention on shared memory. Although `µTPArtc`'s current regulatory actions are directed at interfering CPUs, we believe our methodology is broadly applicable and, with minor engineering modifications, can be expanded to include DMAs. Notably, several *nexgen* platforms (e.g., *nxp5*, *nxp6*, *stm4*) offer DMA controllers that allow software to suspend/resume each channel at any time, presenting a practical way to integrate DMA management into `µTPArtc`'s resources allocation strategy.

6 Related Work

To the best of our knowledge, only two works have specifically measured interference in MCUs [31, 34]. In [34], temporal interference is observed in the access to shared memories. The authors propose a memory allocation strategy for statically segregating code and data; however, this approach offers limited scalability and flexibility. On the other hand, [31] introduces a framework for analyzing interference in low-end MCUs and proposes a monitoring system to manage contention; however, they lack implementation and evaluation details. Both contributions target the Arm Musca-A1 test chip, which does not include common features of commercial platforms (e.g., DMAs and flash acceleration). This section summarizes significant research on shared resource contention and explores studies on TPA techniques.

Measuring multi-core interference. Assessing interference in multi-core systems requires a comprehensive knowledge of a system's s underlying microarchitecture in order to identify sources of contention and fine-tune synthetic benchmarks that stress-test those resources. Radojkovic et al. [35] designed a set of benchmarks to stress core- and memory-level shared resources (e.g., L1 caches, memory bandwidth through L2 cache) in 3 multi-core Intel processors. In [16], D. Iorga, et al. proposed techniques for empirically testing interference across different chips (Arm Cortex-A53, -A7, and Intel Atom); authors are able to cause slowdowns up to 3.8x larger than prior work. In [3], authors carefully crafted write-intensive applications that are capable of originating a DDoS attack on a particular internal structure of a Cortex-A53-based last-level cache (LLC). While these works offer insights into interference assessment methods, MCUs represent a distinct computing paradigm with unique challenges and limitations.

Addressing multi-core interference. Table 5 highlights several works proposing techniques to address interference, mainly in the LLC and DRAM resources. These approaches utilize specific hardware features, including performance monitor counters (PMC) [3, 7, 27, 39, 53, 54], as well as resource partitioning and virtual memory capabilities [19, 24–27, 34, 50, 52]. Additionally, some techniques take advantage of debug facilities [56] or specialized interconnects [13, 40, 44, 46, 55]. Memguard [53, 54] presented the initial effort using PMCs to regulate memory bandwidth, and subsequent works have included cache coloring features [27] and reinforced the actual memory utilization by using memory's QoS-PU [39]. In another perspective, MemPol [56] uses debug facilities to monitor interference outside the core. PALLOC [52] optimizes the use of DRAM by allocating memory pages for each application to specific banks, while [19] leverages virtualization extensions to extend a hypervisor with memory coloring. In cache partitioning, different techniques have been proposed [24, 26, 50]. Mancuso et al. [24] proposed "Cache Lockdown," combining cache coloring and locking techniques. Other works propose mechanisms to address interference from interrupts [7] and DMA-enabled I/O peripherals [55]. While effective in APU systems, these methods are less applicable to MCUs which lack the necessary hardware.

Existing TPA methods. Existing methods to achieve TPA fall into two categories, i.e., software instrumentation [20,21,43] and hardware monitoring [6]. All software instrumentation approaches require the source code of AUTs available, and the source code needs to be modified to add a portion of TPA logic. These requirements are constraints on the AUT side but make software approaches implementable on a wider range of platforms because the modified AUT carries out necessary TPA logic, which needs to be handled by the platform otherwise. On the other hand, the hardware monitor method only needs the binaries of the AUTs without code instrumentation, but the platform needs a trace unit infrastructure. From an overhead perspective, all software approaches rely on interrupting or at least "preempting" AUT from its payload to conduct TPA, which inevitably introduces overhead; on the contrary, the hardware approach introduces negligible to no overhead, and the AUT can run as it is. Our work achieves a new trade-off: (a) in contrast to [6] relying on a trace unit, ours leverage ISA-defined hardware breakpoints, which allows (b) to operate on black-box binary AUTs with minor modifications to the linking step, but by (c) introducing some overhead due to the nature of hardware breakpoints. Table 6 shows the trade-off. On the TPI side, all works derive a reference time for each milestone via offline profiling and measure the slack at runtime. Kritikakou et al. [20, 21] utilize the slack to enforce the timing requirement for hard real-time systems, Sinha et al. [43] to run a scheduler upon milestone hits to prevent unnecessary

Table 6 The trade-off among different methods.

Category	Work	Preemption	Code Instrumentation	TU	Breakpoint
Software	Kritikakou et al.	●	●		
	Sinha et al.	●	●		
Hardware	Chen et al.			●	
	Ours	●			●

abortions of low critical tasks, and Chen et al. [6] to conduct controlled degradation. Ours also adopts the controlled degradation scheme. Importantly, as presented in Table 5, this is the first work demonstrating how TPA-based regulation can be applied to MCUs.

7 Conclusion

As MCUs evolve to embrace multi-core architectures with complex memory hierarchies and integrate multiple DMAs, their predictable and deterministic nature faces new challenges due to the interference within shared hardware resources. Recognizing the scarcity of research on contention issues in these platforms, we embarked on an empirical investigation to assess the impact of contention delays on application performance on three different COTS platforms. Our research uncovers that contention can cause application slowdowns by factors ranging from 2x to an extreme 3800x. To address this, we proposed `µTPArtc`, a mechanism designed to enforce TPI on applications. We demonstrated `µTPArtc`'s ability to regulate performance degradation within a predetermined threshold while incurring a minimal overhead. This establishes `µTPArtc` as a promising strategy for achieving timeliness in modern MCUs.

References

1 Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I. Davis. A Comprehensive Survey of Industry Practice in Real-Time Systems. *Real-Time Systems*, 58(3):358–398, 2022.

2 Arm. Arm Delivers Record Revenues and Record Profits in FY21, 2022. URL: `https://newsroom.arm.com/news/arm-delivers-record-revenues-and-record-profits-in-fy21`.

3 Michael G. Bechtel and Heechul Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 357–367, 2019.

4 Jacob Beningo. Why and How to Get Started with Multicore Microcontrollers for IoT Devices at the Edge, 2020. URL: `https://www.digikey.com/en/articles/why-and-how-to-get-started-with-multicore-microcontrollers`.

5 Marko Bertogna. A View on Future Challenges for the Real-Time Community, 2019. URL: `https://www.irit.fr/rtns2019/wp-content/uploads/2019/11/bertogna_keynote.pdf`.

6 Weifan Chen, Ivan Izhbirdeev, Denis Hoornaert, Shahin Roozkhosh, Patrick Carpanedo, Sanskriti Sharma, and Renato Mancuso. Low-Overhead Online Assessment of Timely Progress as a System Commodity. In *Euromicro Conference on Real-Time Systems (ECRTS)*, volume 262, pages 13:1–13:26, 2023.

7 Diogo Costa, Luca Cuomo, Daniel Oliveira, Ida Maria Savino, Bruno Morelli, José Martins, Fabrizio Tronci, Alessandro Biasci, and Sandro Pinto. IRQ Coloring: Mitigating Interrupt-Generated Interference on ARM Multicore Platforms. In *Next Generation Real-Time Embedded Systems (NG-RES)*, volume 108, pages 2:1–2:13, 2023.

8 Miguel Costa, Diogo Costa, Tiago Gomes, and Sandro Pinto. Shifting capsule networks from the cloud to the deep edge. *ACM Transactions on Intelligent Systems and Technology*, 13(6), September 2022.

9 Dakshina Dasari, Benny Akesson, Vincent Nélis, Muhammad Ali Awan, and Stefan M. Petters. Identifying the Sources of Unpredictability in COTS-based Multicore Systems. In *IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 39–48, 2013.

10 Kinjal Dave. Optimizing ARM Cortex-A and Cortex-M Based Heterogeneous Multiprocessor Systems for Rich Embedded Applications. In *Embedded World Conference*, 2017.

11 Jack Ganssle. On Wait States, 2015. URL: `https://www.embedded.com/on-wait-states/`.

12 Hamed HaddadPajouh, Ali Dehghantanha, Reza M. Parizi, Mohammed Aledhari, and Hadis Karimipour. A Survey on Internet of Things Security: Requirements, Challenges, and Solutions. *Internet of Things*, 14:100129, 2021.

13 Přemysl Houdek, Michal Sojka, and Zdeněk Hanzálek. Towards Predictable Execution Model on ARM-based Heterogeneous Platforms. In *International Symposium on Industrial Electronics (ISIE)*, pages 1297–1302, 2017.

14 Infineon. *PSoC 6 MCU: CY8C63x6, CY8C63x7 Architecture Technical Reference Manual*, 2020.

15 Gideon Intrater. On Wait States, 2019. URL: `https://embeddedcomputing.com/technology/storage/execute-in-place-xip-an-external-flash-architecture-ideal-for-the-code-and-performance-requirements-of-edge-iot-and-ai`.

16 Dan Iorga, Tyler Sorensen, John Wickerson, and Alastair F. Donaldson. Slow and Steady: Measuring and Tuning Multicore Interference. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 200–212, 2020.

17 Ian Johnson Joseph Yiu. Multi-core Microcontroller Design with Cortex-M Processors and CoreSight SoC. Technical report, Arm, 2013.

18 Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, X. Zhang, and Dongyan Xu. Securing Real-Time Microcontroller Systems through Customized Memory View Switching. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

19 Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14, 2019.

20 Angeliki Kritikakou, Claire Pagetti, Olivier Baldellon, Matthieu Roy, and Christine Rochange. Run-Time Control to Increase Task Parallelism In Mixed-Critical Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 119–128, 2014.

21 Angeliki Kritikakou, Christine Rochange, Madeleine Faugère, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and Daniel Gracia Pérez. Distributed Run-Time WCET Controller for Concurrent Critical Tasks in Mixed-Critical Systems. In *International Conference on Real-Time Networks and Systems (RTNS)*, pages 139–148, 2014.

22 Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesus Carretero. A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms. *IEEE Access*, 10:21853–21882, 2022.

23 Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Computing Surveys*, 52(3), 2019.

24 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-Time Cache Management Framework for Multi-Core Architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, 2013.

25 José Martins and Sandro Pinto. Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023.

26 José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Next Generation Real-Time Embedded Systems (NG-RES)*, volume 77, pages 3:1–3:14, 2020.

27 Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms. In *International Conference on Industrial Technology (ICIT)*, pages 1651–1657, 2018.

28 Fabio Montagna, Giuseppe Tagliavini, Davide Rossi, Angelo Garofalo, and Luca Benini. Streamlining the openmp programming model on ultra-low-power multi-core mcus. In *International Conference on Architecture of Computing Systems (ARCS)*, volume 12800, pages 167–182, 2021.

29 NXP. *LPC55S6x/LPC55S2x/LPC552x User manual*, 2022.

30 Daniel Oliveira, Weifan Chen, Sandro Pinto, and Renato Mancuso. MCU Contention Evaluation Framework. Software, version 0.1., swhId: `swh:1:dir:ac8ce99c9ea41bf6dba2aaa197f4bda471aa5790` (visited on 2024-06-19). URL: `https://github.com/danielRep/mcu-tpa-eval`.

31 Daniel Oliveira, Weifan Chen, Sandro Pinto, and Renato Mancuso. Investigating and Mitigating Contention on Low-End Multi-Core Microcontrollers. In *Real-time And IntelliGent Edge Computing Workshop (RAGE)*, pages 221–226, 2023.

32 David Patterson, Jeremy Bennett, Palmer Dabbelt, Cesare Garlati, G. S. Madhusudan, and Trevor Mudge. Embench: A Modern Embedded Benchmark Suite. URL: `https://www.embench.org/`.

33 Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst Case Delay Analysis for Memory Interference in Multicore Systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 741–746, 2010.

34 Sandro Pinto, Hugo Araujo, Daniel Oliveira, José Martins, and Adriano Tavares. Virtualization on TrustZone-Enabled Microcontrollers? Voilà! In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304, 2019.

35 Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-Critical Environments. *ACM Transactions on Architecture and Code Optimization*, 8(4), 2012.

36 Grand View Research. Microcontroller Market Size & Share and Trends Analysis Report By Product (8-bit, 16-bit, 32-bit), By Application (Consumer Electronics & Telecom, Automotive, Industrial, Medical Devices, Aerospace & Defense), By Region, And Segment Forecasts, 2023 - 2030, 2022. URL: `https://www.grandviewresearch.com/industry-analysis/microcontroller-market`.

37 Precedence Research. Microcontroller (MCU) Market - Global Market Size, Trends Analysis, Segment Forecasts, Regional Outlook 2023 - 2032, 2023. URL: `https://www.precedenceresearch.com/microcontroller-mcu-market`.

38 Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto. BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect. In *Security and Privacy (S&P)*, pages 62–62, 2024.

39 Ahsan Saeed, Dakshina Dasari, Dirk Ziegenbein, Varun Rajasekaran, Falk Rehm, Michael Pressler, Arne Hamann, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 133–145, 2022.

40 Alejandro Serrano-Cases, Juan M. Reina, Jaume Abella, Enrico Mezzetti, and Francisco J. Cazorla. Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In *Euromicro Conference on Real-Time Systems (ECRTS)*, volume 196, pages 3:1–3:26, 2021.

41 Haoqi Shan, Dean Sullivan, and Orlando Arias. When Memory Mappings Attack: On the (Mis)use of the ARM Cortex-M FPB Unit. In *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2023.

42 Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Security and Privacy (S&P)*, 2016.

43 Soham Sinha, Richard West, and Ahmad Golchin. PAStime: Progress-Aware Scheduling for Time-Critical Computing. In *Euromicro Conference on Real-Time Systems (ECRTS)*, volume 165, pages 3:1–3:24, 2020.

44 Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management. In *Real-Time Systems Symposium (RTSS)*, pages 345–357, 2020.

45 Statista. Microcontroller Unit (MCU) Market Revenues from 2005 to 2021 by Type. Technical report, Statista, 2021.

46 Ashley Stevens. Quality of Service (QoS) in ARM Systems: An Overview. Technical report, Arm, 2014.

47 STMicroelectronics. *STM32L42xxx advanced Arm-based 32-bit MCUs*, 2018.

48 Xi Tan, Zheyuan Ma, Sandro Pinto, Le Guan, Ning Zhang, Jun Xu, Zhiqiang Lin, Hongxin Hu, and Ziming Zhao. Where's the "up"?! A Comprehensive (bottom-up) Study on the Security of Arm Cortex-M Systems, 2024. `arXiv:2401.15289`.

49 Ken Tindell. Raspberry Pi Pico Priority Inversion, 2021. URL: `https://kentindell.github.io/2021/03/05/pico-priority-inversion/`.

50 Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

51 Joseph Yiu. *Definitive Guide to Arm Cortex-M23 and Cortex-M33 Processors*. Newnes, 1st edition, 2021.

52 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014.

53 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

54 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2016.

55 Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms. *Software: Practice and Experience*, 52(5):1095–1113, 2022.

56 Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. MemPol: Policing Core Memory Bandwidth from Outside of the Cores. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–248, 2023.