

Investigating and Mitigating Contention on Low-End Multi-Core Microcontrollers

Daniel Oliveira
daniel.oliveira@dei.uminho.pt
Centro ALGORITMI, University of Minho
Guimarães, Portugal

Sandro Pinto
sandro.pinto@dei.uminho.pt
Centro ALGORITMI, University of Minho
Guimarães, Portugal

Weifan Chen
wfchen@bu.edu
Dep. of Computer Science, Boston University
Boston, MA, USA

Renato Mancuso
rmancuso@bu.edu
Dep. of Computer Science, Boston University
Boston, MA, USA

ABSTRACT

In this paper, we investigate the problem of contention and loss of predictability in modern microcontrollers (MCU). To address this issue, we first present a framework to empirically analyze and observe the impact of interference on low-end MCUs. With carefully crafted evaluation scenarios, we conduct experiments on an Arm’s Musca-A1 platform and provide sufficient evidence that even with common application setups, interference can slowdown applications by several orders of magnitude. Furthermore, we propose an architecture for a novel mitigation system that enables applications to monitor their timing progress slackness and mitigate temporal interference over shared resources. This is achieved by suspending less critical cores and reconfiguring their priority on the bus when intolerable contention delays are present. Our findings emphasize the critical importance of considering the impact of shared resources, such as interconnects and memory access patterns, on low-end multi-core MCUs. It is, therefore, crucial to design mechanisms that can allow MCU-based applications to regain control of their timeliness.

CCS CONCEPTS

• **Computer systems organization** → *Real-time system specification*.

KEYWORDS

microcontrollers, multi-core, predictability, contention

ACM Reference Format:

Daniel Oliveira, Weifan Chen, Sandro Pinto, and Renato Mancuso. 2023. Investigating and Mitigating Contention on Low-End Multi-Core Microcontrollers. In *Cyber-Physical Systems and Internet of Things Week 2023 (CPS-IoT Week Workshops ’23)*, May 9–12, 2023, San Antonio, TX, USA. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3576914.3587513>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CPS-IoT Week Workshops ’23, May 9–12, 2023, San Antonio, TX, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0049-1/23/05...\$15.00
<https://doi.org/10.1145/3576914.3587513>

1 INTRODUCTION

The increasing demand for higher computing power in embedded systems, such as automotive, avionics, and industrial automation, has led to the adoption of multi-core systems, even in low-cost devices [6, 23, 28]. These devices are typically powered by small microcontrollers (MCUs) with simple architectures that allow for predictable and deterministic operations. Thus, they are specifically useful in real-time control systems, e.g., airbag collision detection [24, 34, 35]. Leading industry players have introduced various MCU platforms with dual-core architectures, such as STMicroelectronics’ STM32H7 line and NXP’s i.MX RT1170 and i.MX RT1180, featuring different dual-core setups [22, 32]. Unfortunately, as MCUs broadly adopt multi-core architectures with more complex memory hierarchies, their predictability and suitability for scenarios with strict real-time constraints are jeopardized [8, 26, 33].

The performance interference caused by shared hardware resources can introduce timing variations that significantly hinder the predictability of applications [4, 15]. In high-end multi-core systems, common sources of contention include caches, bus interconnects, and the DRAM memory banks and controller [3, 7–9, 29, 36, 38]. In contrast, contention points have been seldom in MCUs due to their use of different on-chip memory technologies and less complex bus topologies. However, emerging applications require greater memory space and performance. This has led to the integration of CPUs with higher clock speeds, L1 instruction caches, DMA components, multiple I/O peripherals, and on- and off-chip memories (e.g., QSPI/Hyper-Flash, data/code SRAMs) [10, 26].

The real-time systems community has extensively studied issues with performance degradation due to hardware resource contention, and several techniques have been proposed to restore predictability (e.g., cache coloring [16, 18, 19], memory throttling [7, 36, 38]). However, these solutions tend to be geared towards high-end platforms and require specialized hardware features that are not present in MCUs (e.g., performance counters¹, two-stage MMUs, cache locking). As a result, the problem of observing and addressing interference in MCU systems has been largely overlooked in the literature.

In this paper, we present a framework to empirically analyze the reciprocal interference of shared resources on modern MCUs. Our goal is to provide evidence on the extent of the problem and highlight the need for systematically addressing it. To examine

¹The recent Armv8.1-M-based Cortex-M55 provides a Performance Monitoring Extension, but boards are still unavailable.

the effect of contention, we conducted several experiments on the Armv8-M-based Musca-A1 platform [2]. These experiments aim to provide empirical evidence about the contention for different concurrent access paths to memory and micro-architectural resources. In light of this problem, we propose a novel mitigation system that automatically instruments an application with meaningful progress milestones to be monitored in run-time. The run-time mechanism is capable of taking corrective measures to restore the application's temporal behavior when negative time slack is detected.

In summary, our main **contributions** are:

- A flexible framework that enables the definition of different evaluation scenarios specifically designed to maximize interference over shared memory resources in low-end MCUs. The framework allows to (i) use different bus masters to create contention (e.g., cores, DMAs), (ii) enable platform-dependent features (e.g., caches, performance enhancers), (iii) interfere over multiple shared resources, and (iv) configure several benchmarks and synthetic interference applications.
- An extensive empirical evaluation on the Musca-A1 platform, leveraging our framework. Our results provide evidence that interference can slow applications up to starvation values (15×) even with common memory layouts. We also unveil peculiar behaviors of micro-architectural interconnects that lead to severe slowdowns for specific memory layouts. We open-sourced all artifacts to enable independent validation of the results.
- The proposal of a novel interference mitigation framework composed of (i) a design-time tool that automatically instruments applications with progress milestones. The milestones serve as observation points for (ii) a run-time mechanism. The latter leverages widely-available hardware features to temporarily suspend interfering workloads, allowing the system to restore the timely behavior of the application.

2 RELATED WORK

Interference Assessment Frameworks. Assessing and detecting interference patterns in multi-core systems typically requires a deep understanding of the platform's underlying micro-architecture in order to fine-tune the resource-stressing synthetic benchmarks. In [27], the authors proposed a framework that utilizes a set of well-defined benchmarks to stress specific shared resources and quantify application slowdowns. Nowotsch et al. [21] evaluated multi-core systems for safety-critical applications, finding that interference can cause significant slowdowns in worst-case execution time (WCET) estimations. Iorgar et al. [11] proposed an approach for measuring multi-core interference through an auto-tuning framework that maximizes interference on shared memory resources. RT-bench [20] is an open-source framework that adds real-time features to existing benchmarks. It offers a set of automated analysis use-cases that provide key real-time metrics such as as observed WCET and the impact of contention on shared resources. While these approaches can effectively evaluate temporal interference, they solely target high-end systems. In contrast, our solution focus on low-end devices and offers the flexibility to create hostile environments that maximize interference according to MCU-specific architectural characteristics (e.g., memory access paths, micro-architectural peculiarities).

Interference Mitigation Techniques. The real-time systems community has proposed several mitigation techniques. These techniques typically target high-end platforms and rely on hardware-specific features such as performance counters [5, 29, 37], resource partitioning and virtual memory capabilities [16–19, 36], or specialized interconnects (e.g., QoS modules that are available on specific platforms [30, 31]). MemGuard leverages performance counters on x86 architectures to provide temporal isolation through memory bandwidth regulation [37]. Crespo et al. [5] proposed a feedback control scheme that uses the Performance Monitoring Unit (PMU) in a PowerPC platform to manage the execution of critical partitions. In [29], Arm's PMU and a platform-specific memory profiling unit have been utilized to implement a mechanism that dynamically regulates memory accesses of cores. Resource partitioning techniques are applied to shared resources, such as cache and DRAM. In [16], the authors proposed the Colored Lockdown concept that combined page-coloring and cache lockdown to keep frequently accessed pages in the cache. The PALLOC mechanism optimizes the use of DRAM by allocating memory pages for each application to specific banks [36]. Additionally, in [19], authors proposed a cache coloring-based technique and in [18], authors implemented support for cache coloring on Bao hypervisor. Regarding hardware support for QoS, Arm offers hardware mechanisms that aim to prevent congestion in the interconnect [31], which are evaluated on a Xilinx UltraScale+ in [30]. In another line of works, Kritikakou et al. [13] presented a formal description of a mechanism that, under interference, suspends low criticality tasks until the termination of the critical task; an implementation is presented in [14]. While this variety of mechanisms have been proposed to address interference in high-end systems, they are not directly applicable to low-end devices. To the best of our knowledge, a single work [26] addresses interference in low-end systems (Cortex-M33). The proposed static memory allocation scheme distributes code/data segments across different memory elements; however, this requires high engineering effort and a complex analysis of the platform's memory subsystem.

3 INTERFERENCE IN LOW-END MCUS

In this section, we present an experimental investigation to provide conclusive evidence that MCUs are susceptible to interference when different hardware elements (e.g., cores) attempt to access shared resources (e.g., buses, flash, SRAM memories) concurrently. The results reveal that the issue of contention can lead to significant performance degradation and unpredictable execution times.

3.1 Highlighting the Problem

Our experiments were conducted on the Arm Musca-A1 platform [2]. The Musca-A1 lacks features such as multi-level shared caches or memory virtualization mechanisms, which are known to be sources of contention on high-end systems [8, 12]. However, increased demand for AI on edge devices and the proliferation of connected devices that integrate heavy-size communication protocols are leading to the need for MCUs to support various on- and off-chip memories to handle the large amount of data required for these applications [10]. To accommodate this, memory subsystems on MCUs are becoming highly heterogeneous. Different types of memory are accessed through different bus paths and controllers, clocked

at different frequencies, and shared between multiple masters in the interconnect (e.g. cores, DMAs). In the case of the Musca-A1 platform, as depicted in Figure 1, two differently-clocked CPUs have access to four distinct memory elements: (i) private-core 2KiB instruction caches; (ii) on-chip 4x32KiB data iSRAM banks, which serve as tightly-coupled memories for each corresponding core and are organized with dedicated bus connections; (iii) an external 2MiB code eSRAM, which is clocked at the same frequency as core 0; (iv) and an off-chip 8MiB Flash memory, which is accessed through the QPSI controller and therefore clocked at a much lower frequency than the CPUs. The heterogeneity of the Musca’s memory hierarchy offers the potential to create interference scenarios with varying levels of contention. This provides an opportunity to evaluate performance in both common and uncommon configurations.

3.2 Framework Methodology

In contrast to high-end systems, performance interference in multi-core MCUs is still a largely overlooked problem in the literature. Yet, they are widely adopted for real-time applications with the assumption of high predictability [24, 26, 34]. To address this gap and raise awareness in the community, we developed an open-source testing framework aimed at providing sufficient evidence of the significant impact that interference can have on these systems. Our framework is the first to support a comprehensive investigation into this issue and to enable thorough analyses of how different system configurations introduce contention on MCUs. Its primary goals are:

- **Portability:** the framework can be adapted across different Arm-based MCU platforms with the use of a BSP abstraction layer and by providing a common boot-code per architecture. The boot routine initializes platform-specific hardware (e.g., serial ports, caches, performance enhancers, bus arbitration priorities), and relocates each application to the targeted contended memory.
- **Configurability:** the framework can be easily modified to test different interference scenarios. Therefore, the framework allows for each test to: (i) place code and data regions on any memory element; (ii) enable/disable platform components that impact results; (iii) select the most effective benchmark and synthetic interfering application; (iv) and choose optimization options.
- **Reproducibility:** the framework enables consistent and accurate results to be obtained across multiple runs.

Inspired by the framework methodology proposed by Iorga et al. [11], we present our testing framework that creates a hostile environment to stress a specific shared resource and evaluate its impact on system contention. This consists of selecting the most suitable (i) *benchmark program* that will experience slowdowns due to multi-core contention caused by (ii) an optimized, synthetic *interfering program* that stresses key shared resources.

A user must primarily decide which shared resource they want to target. Let R denote the set of possible hardware resources over which contention can be created. For example, in case of the Musca-A1 ($ma1$), we define $R_{ma1} = \{\text{flash}, \text{esram}, \text{isram}, \text{ahb5mux}\}$. The framework offers two benchmarks to be run on the observed core, namely, *statemate* and *edn*. Let B denote the available benchmarks for the observed core, i.e., $B = \{\text{statemate}, \text{edn}\}$. Let I denote the set of *interfering programs* that the user has available to run

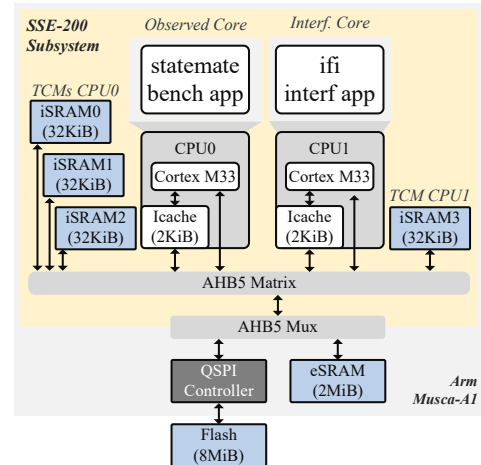


Figure 1: Musca-A1 memory hierarchy (adapted from [2]).

on the interfering core, i.e., $I = \{\text{ifi}, \text{dai}\}$. Depending on the $r \in R$ to interfere, a $b \in B$ must be selected to run on core 0 (b_r) and a $i \in I$ must be selected to run on core 1 (i_r). For example, if $r_{ma1} = \{\text{flash}, \text{esram}\}$, i.e., code memories, a more branch-intensive pair of applications should be selected; therefore, $b_r = \text{statemate} \wedge i_r = \text{ifi}$. On the other hand, if $r_{ma1} = \{\text{isram}\}$, i.e., data memory, a more memory-intensive pair of applications should be selected; therefore, $b_r = \text{edn} \wedge i_r = \text{dai}$. Later, we detail each of these applications. We can now define an *environment* as a setup of a b_r and a i_r , each assigned to its respective core. The performance of a b_r in environment e is represented by $\text{perf}(b_r, e)$, which represents the execution of b_r on core 0 while e is running on the other core. Let $\text{interf}(b_r, i_r)$ denote the interference value associated with executing program b_r in isolation on core 0 and executing it in parallel with an instance of i_r on core 1. Therefore, we have the following equation (nop denotes the absence of an interfering program):

$$\text{interf}(b_r, i_r) = \frac{\text{perf}(b_r, i_r)}{\text{perf}(b_r, \text{nop})} \quad (1)$$

3.3 Evidence Results

Experimental Setup. We conducted several experiments on the Musca-A1 platform using our framework to showcase the existence of multi-core contention issues on MCUs. We configure the Data Watchpoint and Trace (DWT) cycle counter to obtain 1000 measurement samples in each test, providing the wall-clock execution time. The most suitable b_r and i_r were selected based on the specific characteristics of the target r . The synthetic *ifi* and *dai* interfering applications are designed to have a varying degree of impact depending on the target r . The *ifi* application forces the interfering core to bypass the cache, using a sequence of nop instructions and branching between multiple blocks that fill up a cache line. On the other hand, the *dai* application is optimized to create interference through data-intensive operations by reading and writing constantly to a buffer array. Moreover, we carefully integrated a set of branch- and memory-intensive benchmarks from

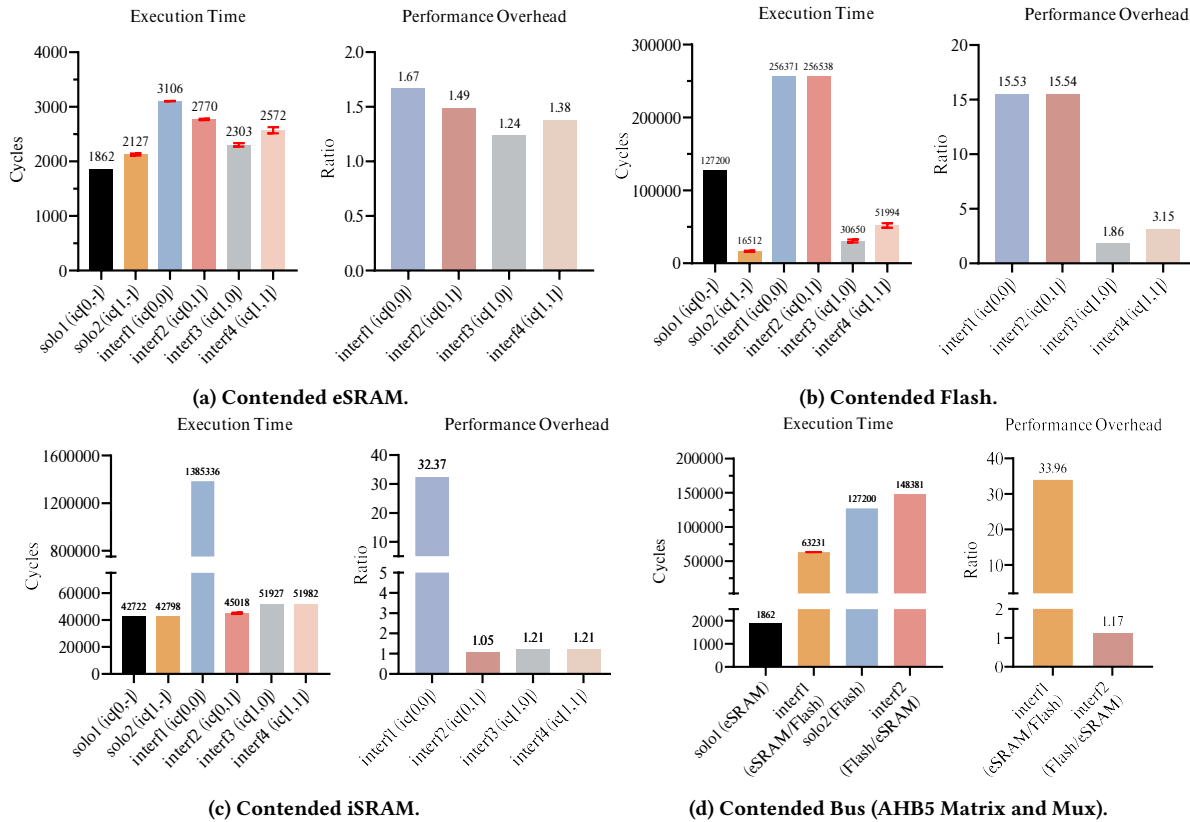


Figure 2: Execution time and performance overhead of the benchmark for different interfering setups. The baseline selected for the performance overhead ratio is always the *solo* experiment that achieves the lowest execution time. The *ic[.,.]* denotes if the instruction cache (IC) is enabled on the experiment by core, i.e. if *ic[1,0]* the IC is enabled on core 0 and disabled on core 1.

the Embench suite [25]. The selected benchmarks are *statemate* for stressing code memories and *edn* for stressing data memories.

Our investigation comprises four evaluation scenarios that were designed to determine the impact of interference on the following shared resources: the data and code memories (i.e., iSRAM, eSRAM, and Flash) and a dedicated interconnect bus connecting both code memories to the AHB5 Matrix, i.e., the AHB5 Mux (Figure 1). The evaluation scenarios are organized as follows:

- Contended eSRAM:** core 0 and core 1 fetch instructions from the faster code memory, i.e., the on-chip eSRAM. Data is placed in separate iSRAM banks.
- Contended Flash:** core 0 and core 1 fetch instructions from the slower code memory, i.e., the off-chip QSPI Flash. Data is placed in separate iSRAM banks.
- Contended iSRAM:** core 0 and core 1 issues data read/writes from a single iSRAM bank (iSRAM1). Code is split between eSRAM (core 0) and Flash (core 1) memories.
- Contended Bus (AHB5 Mux):** core 0 fetches instructions from eSRAM and core 1 from Flash, and vice-versa. The expected contention arises from the AHB5 Mux interconnect rather than the memory controllers. Data is placed in separate iSRAM banks.

Contended eSRAM. The results of the experiments on the impact of eSRAM contention are presented in Figure 2a. The evaluation

is based on the execution time and performance overhead. To begin, the execution time of the *statemate* benchmark without contention was measured (i.e., both *solo* tests). The experiments were conducted in two scenarios, one with the core 0 instruction cache (IC) enabled and the other with it disabled. The results show that, despite the low 6% miss ratio of the *statemate* benchmark, the IC has an unexpected negative impact on the performance, i.e., a 14.2% slowdown. We hypothesize that the extra cycles required to perform a cache lookup followed by a memory fetch for the 6% of the instructions that resulted in cache misses might induce a higher overhead than directly accessing the SRAM memory. To validate this hypothesis, we conducted an experiment in which we measured the execution time of cache-friendly code, i.e., a segment of code with 0% cache misses. The results demonstrated that the eSRAM performs similarly to cache memory, with no discernible difference in execution time regardless of cache enablement. We then iteratively enable/disable the IC across the four experiments to measure the slowdown due to contention. The results show that disabling the ICs on both CPUs leads to a 67% increase in performance (*interf1*). It is noteworthy that while the IC can have a negative impact on single-core execution, it has a positive effect on a contended eSRAM by reducing the contention cost by up to 35% (*interf3*). Furthermore, our experiments show that enabling the

ICs results in a variation in the execution time, thereby eliminating the deterministic behavior observed in the solo run with the IC disabled (*solo1*).

Contended Flash. Figure 2b depicts the assessed results when interference is created on the Flash memory. As expected, in this case, enabling the IC largely improves performance by almost an order of magnitude (7 \times) compared to the *solo1* scenario. Upon examining the results of the *interf1* and *interf2* experiments, we observe a similar performance overhead ratio that can reach up to 15 \times . Moreover, both the *interf1* and *interf2* experiments show similar execution times as the *if1* interfering application forces core 1 to evict cache lines, causing core1 to fetch instructions from Flash. Regarding the counter-intuitive results observed in *interf4*—where a 1.7x degradation is seen compared to *interf3*—we attribute this to the Flash controller’s poor handling of non-sequential blocks of code. This is likely due to the controller’s constant buffering of new pages from Flash memory in response to requests from core 1 at each cache miss. Additionally, we confirm this hypothesis by comparing the performance between running a sequential number of nop instructions from a single Flash page and running the same number of instructions from multiple Flash pages in a non-sequential manner. The results show that the latter scenario incurs a higher performance cost, around 41.8%.

Contended iSRAM. The impact of iSRAM contention on performance is presented in Figure 2c. The results show that the execution time of the *solo* experiments are similar with or without the IC enabled, as the *edn* benchmark is highly memory-intensive and exhibits good spatial locality, with a 1% cache miss ratio. The *interf1* experiment highlights an unexpected measurement compared to the remaining interfering tests (core 0 is starving). This severe performance degradation can be attributed to an undocumented behavior of the Musca-A1 system. By observing Figure 1, we noticed that core 0 and core 1 access the eSRAM and Flash memories through the AHB5 Mux, which acts as an arbiter for their access. Therefore, the starvation of core 0 in the *interf1* experiment suggests that core 1’s stream of Flash instructions may be prioritized over core 0’s stream of eSRAM instructions, leading to a slowdown of approximately 32 \times . The same behavior is not observed in other interfering setups (i.e., *interf2*, *interf3* and *interf4*) since the access to the memories is less concurrent, given the good spatial locality of the workloads, resulting in a smaller impact on performance. We further investigate this hypothesis by designing the next experiment, which applies a hybrid memory layout that uses eSRAM and Flash memories but avoids data contention.

Contended Bus (AHB5 Mux). In the fourth evaluation scenario, we investigate the contention over the AHB5 Mux interconnect. Both cores fetch instructions from eSRAM and Flash memories, respectively. First, we performed the *solo* experiments for each memory element, i.e., *solo1* (eSRAM) and *solo2* (Flash). Then, we induce interference on the AHB5 Mux by having each core fetch instructions from different memories, i.e., if core 0 executes from eSRAM, core 1 executes from Flash, and vice-versa. Figure 2c shows the results of the experiments with the IC always disabled throughout the experiments, allowing a clearer understanding of the results. The results show that, despite eSRAM being a fast memory, when

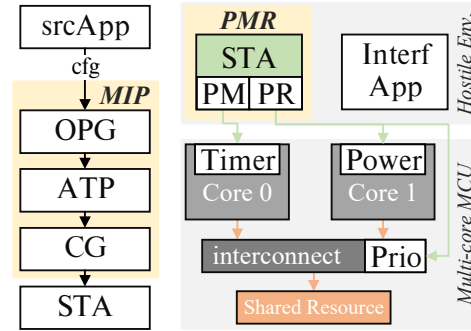


Figure 3: System overview of the proposed architecture.

core 1 executes from Flash (*interf1*), there is a significant 34 \times increase in performance observed in core 0. This result is similar to what was observed in the previous evaluation scenario, where core 1 once again starves core 0. However, when the cores switch between memory blocks (*interf2*), the level of contention observed is significantly less severe (1.2 \times). We conclude that the interconnect prioritizes transactions to/from Flash over eSRAM; however, the Musca-A1 manual does not corroborate this hypothesis. Nevertheless, these results emphasize the need for careful memory layout configuration in multi-core applications.

4 PROPOSED MITIGATION SYSTEM

Figure 3 presents the foreseen architecture for a system that enables MCU-based applications to restore their timing behavior in the presence of contention delays induced by hostile environments. The solution is composed of two subsystems: (i) the design-time Milestone Identification and Profiling (MIP) tool, a framework to automate the instrumentation of an application with significant milestones to be monitored by (ii) the run-time Performance Monitor and Regulator (PMR) mechanism that evaluates the application’s timely progress, ensuring that end-to-end timeliness is preserved.

Design-time MIP tool. The control flow graph (CFG) of an application is a directed graph $C = \{N, \mathcal{E}\}$. A node $n \in N$ represents a contiguous instruction block ending with a branch instruction, and an edge $(n_p, n_s) \in \mathcal{E}$ indicates the ending branch instruction in n_p is capable of transferring program counter (PC) to n_s . Several tools are capable of generating a CFG from an application [1]. In Figure 3, the MIP subsystem first reads the CFG of the target application. Then, the Observation Points Generator (OPG) uses the CFG to decide the milestones - nodes - to be monitored during run-time. These milestones represent instrumentation points that trigger the PMR monitoring logic. Therefore, when PC reaches a milestone, an overhead would incur; thus, the OPG has to strategically place milestones in order to balance monitoring granularity and overhead. Next, the Application Timing Profiler (ATP) profiles the application. For each milestone, the ATP associates an expected execution time, or *nominal time*. Finally, the Code Generator (CG) uses the timing information to instrument the application’s source code with Program Monitor (PM) calls placed at each milestone. These allow the application to remain aware of its timeliness. Thus, we refer to an instrumented application as a Self-aware Timed Application (STA).

Run-time PMR mechanism. During run-time, an STA is capable of self-monitoring its timely progress at each milestone by invoking the PM. The PM checks the current slack—i.e., the difference between the target completion time (or deadline) and the expected execution time based on current progress information. The Progress Regulator (PR) is triggered if the slack is insufficient to regulate contention over shared resources. One commonly used technique is to stall interfering cores (and/or DMA-capable devices). The stalled cores are resumed when/if sufficient slack is restored at a future milestone. Another approach involves reconfiguring the bus masters' priority on the bus. When the slack is insufficient, the priority of interfering cores can be reduced to decrease their usage of the shared resource, allowing STA to have a higher priority on the bus.

5 CONCLUSION

In this paper, we address the issue of contention-induced temporal interference on low-end MCU devices. We presented an open-source testing framework to analyze the impact of interference and provide undeniable evidence of significant slowdowns in common application setups. We also unveil a proposal for a novel mitigation system that allows applications to monitor their timing progress slackness and mitigate interference over shared resources, a key requirement to ensure predictability on modern multi-core MCUs.

ACKNOWLEDGMENTS

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. The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant number CCF-2008799. 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.

REFERENCES

- [1] angr. 2023. *CFG Documentation*. <https://docs.angr.io/built-in-analyses/cfg>
- [2] Arm. 2018. *Arm® Musca-A*. Technical Report. Arm Ltd.
- [3] Michael G. Bechtel and Heechul Yun. 2019. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. *ArXiv CoRR* (2019).
- [4] Diogo Costa Costa, Luca Cuomo, Daniel Oliveira, Ida Savino, Bruno Morelli, Jose Martins, Fabrizio Tronci, Alessandro Biasci, and Sandro Pinto. 2023. IRQ Coloring: Mitigating Interrupt-generated Interference on ARM Multicore Platforms. In *Proc. NG-RES*.
- [5] Alfons Crespo, Patricia Balbastre, José Simó, Javier Coronel, Daniel Gracia Pérez, and Philippe Bonnot. 2018. Hypervisor-Based Multicore Feedback Control of Mixed-Criticality Systems. *IEEE Access* (2018).
- [6] Rolf Ernst and Marco Di Natale. 2016. Mixed Criticality Systems—A History of Misconceptions? *IEEE Design & Test* (2016).
- [7] Farzad Farshchi, Qijing Huang, and Heechul Yun. 2020. BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors. *Proc. IEEE RTAS* (2020).
- [8] Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. 2019. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *Proc. ECRTS*.
- [9] Mohamed Hassan and Rodolfo Pellizzoni. 2020. Analysis of Memory-Contention in Heterogeneous COTS MPSoCs. In *Proc. ECRTS*.
- [10] Gideon Intrater. 2019. *Execute in place (XiP): An external flash architecture ideal for the code and performance requirements of Edge Iot and ai*. <https://embeddedcomp.com/technology/storage/execute-in-place-xip-an-external-flash-architecture-ideal-for-the-code-and-performance-requirements-of-edge-iot-and-ai>
- [11] Dan Iorga, Tyler Sorensen, John Wickerson, and Alastair F. Donaldson. 2020. Slow and Steady: Measuring and Tuning Multicore Interference. In *Proc. RTAS*.
- [12] Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. 2019. Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. *Proc. IEEE RTAS* (2019).
- [13] Angeliki Kritikakou, Claire Pagetti, Olivier Baldellon, Matthieu Roy, and Christine Rochange. 2014. Run-Time Control to Increase Task Parallelism In Mixed-Critical Systems. *Proc. ECRTS*.
- [14] Angeliki Kritikakou, Christine Rochange, Madeleine Faugere, Claire Pagetti, Matthieu Roy, Sylvain Girbal, and Daniel Gracia Perez. 2014. Distributed Run-Time WCET Controller for Concurrent Critical Tasks in Mixed-Critical Systems. In *Proc. RTNS*.
- [15] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. 2019. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Comput. Surv.* (2019).
- [16] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. 2013. Real-time cache management framework for multi-core architectures. In *Proc. IEEE RTAS*.
- [17] José Martins and Sandro Pinto. 2023. Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems. In *Proc. IEEE RTAS*.
- [18] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. 2020. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Proc. NG-RES*.
- [19] Paolo Walter Modica, Alessandro Biondi, Giorgio C. Buttazzo, and Anup Patel. 2018. Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms. *IEEE International Conference on Industrial Technology*.
- [20] Mattia Nicoletta, Shahin Roozkhosh, Denis Hoornaert, Andrea Bastoni, and Renato Mancuso. 2022. RT-Bench: An Extensible Benchmark Framework for the Analysis and Management of Real-Time Applications. In *Proc. RTNS*.
- [21] Jan Nowotzsch and Michael Paulitsch. 2012. Leveraging Multi-core Computing Architectures in Avionics. *Proc. EDCC*.
- [22] NXP. 2022. *i.MX RT*. <https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus:IMX-RT-SERIES>
- [23] Daniel Oliveira, Miguel Costa, Sandro Pinto, and Tiago Gomes. 2020. The Future of Low-End Motes in the IoT: A Prospective Paper. *Electronics* (2020).
- [24] Runyu Pan, Gregor Peach, Yuxin Ren, and Gabriel Parmer. 2018. Predictable Virtualization on Memory Protection Unit-Based Microcontrollers. In *Proc. IEEE RTAS*.
- [25] David Patterson, Jeremy Bennett, Palmer Dabbelt, Cesare Garlati, G. S. Madhusudan, and Trevor Mudge. 2020. Embench. www.embench.org/.
- [26] Sandro Pinto, Hugo Araujo, Daniel Oliveira, José Martins, and Adriano Tavares. 2019. Virtualization on TrustZone-Enabled MCUs? Voilà!. In *Proc. IEEE RTAS*.
- [27] Petar Radjokovic, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. 2012. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Trans. Archit. Code Optim.* (2012).
- [28] Manuele Rusci, Marco Fariselli, Martin Croome, Francesco Paci, and Eric Flamand. 2022. Accelerating RNN-based Speech Enhancement on a Multi-Core MCU with Mixed FP16-INT8 Post-Training Quantization. *arXiv cs.SD* (2022).
- [29] Ahsan Saeed, Dakshina Dasari, Dirk Ziegenbein, Varun Rajasekaran, Falk Rehm, Michael Pressler, Arne Hamann, Daniel M. Gritschneider, Andreas Gerstlauer, and Ulf Schlichtmann. 2022. Memory Utilization-Based Dynamic Bandwidth Regulation for Temporal Isolation in Multi-Cores. In *Proc. IEEE RTAS*.
- [30] Alejandro Serrano-Cases, Juan M. Reina, Jaume Abella, Enrico Mezzetti, and Francisco J. Cazorla. 2021. Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In *Proc. ECRTS*.
- [31] Ashley Stevens. 2014. *Quality of Service (QoS) in ARM Systems: An Overview*. Technical Report. Arm Ltd.
- [32] STMicroelectronics. 2020. *STM32H72*. <https://blog.st.com/stm32h723-stm32h732-stm32h725-stm32h735-stm32h730/>
- [33] Rohan Tabish, Renato Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S. Phatak, Rodolfo Pellizzoni, and Marco Caccamo. 2016. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In *IEEE RTAS*.
- [34] Ioan Ungurean. 2020. Timing Comparison of the Real-Time Operating Systems for Small Microcontrollers. *Symmetry* (2020).
- [35] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. 2022. RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone. In *Proc. IEEE S&P Symp.*
- [36] Heechul Yun, Renato Mancuso, Zheng Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. *Proc. IEEE RTAS*.
- [37] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2013. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proc. IEEE RTAS*.
- [38] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Raymond Sha. 2013. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. *Proc. IEEE RTAS* (2013).

Received 15 February 2023; revised 12 March 2009; accepted 5 June 2009