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

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

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

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

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

CPS-IoT Week Workshops '23, May 9–12, 2023, San Antonio, TX, USA

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 counters[1], 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

[1]The 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} = {`flash`,`esram`,`isram`,`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 = {`statemate`,`edn`}. Let I denote the set of *interfering programs* that the user has available to run

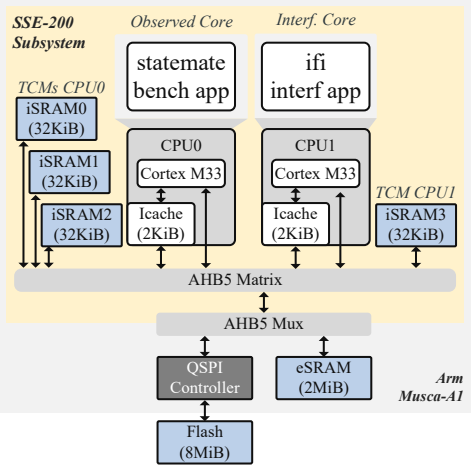


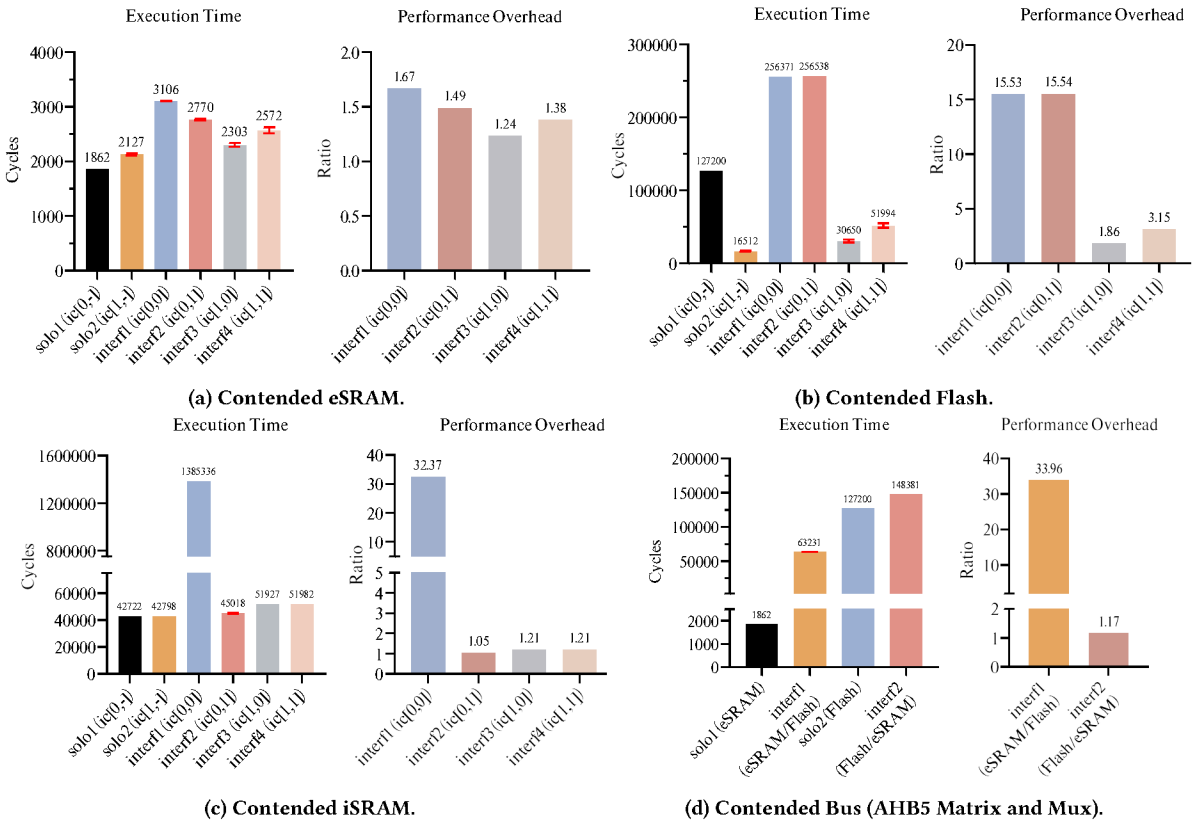
Figure 1: Musca-A1 memory hierarchy (adapted from [2]).

on the interfering core, i.e., I = {`ifi`,`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} = {`flash, esram`}, i.e., code memories, a more branch-intensive pair of applications should be selected; therefore, b_r = `statemate` $\wedge$ i_r = `ifi`. On the other hand, if r_{ma1} = {`isram`}, i.e., data memory, a more memory-intensive pair of applications should be selected; therefore, b_r = `edn` $\wedge$ i_r = `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 $\texttt{perf}(b_r, e)$, which represents the execution of b_r on core 0 while e is running on the other core. Let $\texttt{interf}(b_r, i_r)$ 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*):

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

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



(a) Contended eSRAM. (b) Contended Flash.

(c) Contended iSRAM. (d) Contended Bus (AHB5 Matrix and Mux).

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:

(a) *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.
(b) *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.
(c) *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.
(d) *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