Fourth Workshop on Next Generation Real-Time Embedded Systems

NG-RES 2023, January 18, 2023, Toulouse, France

Edited by

Federico Terraneo
Daniele Cattaneo

OASICS

Editors

Federico Terraneo
Politecnico di Milano, Italy
federico.terraneo@polimi.it

Daniele Cattaneo
Politecnico di Milano, Italy
daniele.cattaneo@polimi.it

ACM Classification 2012
Computer systems organization → Real-time systems; Computer systems organization → Embedded and cyber-physical systems; Software and its engineering → Software notations and tools; Networks

ISBN 978-3-95977-268-6

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-268-6.

Publication date
March, 2023

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/OASIcs.NG-RES.2023.0

OASIcs – OpenAccess Series in Informatics

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

Preface

This volume collects the papers presented at the fourth edition of the Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023). The workshop is co-located with the 2023 edition of the HiPEAC conference and was held on January 18, 2023 in Toulouse, France.

The traditional concept of embedded systems is constantly evolving to address the requirements of the modern world. Cyber-physical systems, networked control systems and Industry 4.0 are introducing an increasing need for interconnectivity. A steadily increasing algorithmic complexity of embedded software is fueling the adoption of multicore and heterogeneous architectures. As a consequence, meeting real-time requirements is now more challenging than ever.

The NG-RES workshop focuses on real-time embedded systems, with particular emphasis on the distributed and parallel aspects. The workshop is a venue for both the networking and multicore real-time communities aiming at cross-fertilization and multi-disciplinary approaches to the design of embedded systems.

Topics of interest include but are not limited to:

- Application of formal methods to distributed and/or parallel real-time systems
- Programming models, paradigms and frameworks for real-time computation on parallel and heterogeneous architectures
- Applications of approximate computing in real-time systems
- Compiler-assisted solutions for distributed and/or parallel real-time systems
- Middlewares for distributed and/or parallel real-time systems
- Networking protocols and services (e.g., clock synchronization) for distributed real-time embedded systems
- Scheduling and schedulability analysis for distributed and/or parallel real-time systems
- System-level software and technologies (e.g. RTOSs, hypervisors, separation kernels, virtualization) for parallel and heterogenous architectures

In this fourth edition of the workshop six regular papers were accepted, each of which receiving two peer reviews. In addition, we are glad to have an invited talk by Martina Maggio titled “Control Systems in the presence of Computational Problems”. We would like to thank the authors of the NG-RES 2023 papers, the members of our program committee, our publisher Schloss Dagstuhl as well as the HiPEAC organizers for contributing to the success of this workshop.

Federico Terraneo and Daniele Cattaneo

Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023).
Editors: Federico Terraneo and Daniele Cattaneo
OASICS OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Organizers of the Workshop

General Chair

- Federico Terraneo, Politecnico di Milano, Italy

Program Chair

- Daniele Cattaneo, Politecnico di Milano, Italy

Program Committee

- Luís Almeida, Universidade do Porto, Portugal
- Benny K. Akesson, TNO, Netherlands
- Ashik Ahmed Bhuiyan, University of Central Florida, United States
- Roberto Cavicchioli, Università di Modena e Reggio Emilia, Italy
- Khalil Esper, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
- Jaume Abella Ferrer, Barcelona Supercomputing Center, Spain
- Miguel Gutierrez Gaitan, Universidade do Porto, Portugal
- Leandro Soares Indrusiak, University of York, United Kingdom
- Alberto Leva, Politecnico di Milano, Italy
- Marc Pouzet, École normale supérieure, Paris, France
- Christine Rochange, Institut de Recherche en Informatique de Toulouse, France
- Marco Solieri, Università di Modena e Reggio Emilia, Italy
- Jürgen Teich, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

Control Systems in the Presence of Computational Problems

Martina Maggio
Univerisität des Saarlandes, Saarbrücken, Germany
Lund University, Sweden

Abstract

Feedback control is a central enabling technology in a wide range of applications. Control systems are at the core of energy distribution infrastructure, regulate the behaviour of engines in vehicles, and are embedded in household appliances like washing machines. Control is centred around the feedback mechanism. Sensors provide information about the current state of the physical environment. This is used to compute suitable control actions to fulfil performance requirements, that are then implemented by actuators. For example, adaptive cruise control systems use measurements from a range of sensors to determine how to adjust the throttle to automatically regulate the vehicle's speed, while maintaining a safe distance from vehicles ahead.

Control actions are often calculated using hardware and software. Hence, the computation of the new control signals is subject to accidental faults, systematic issues, and software bugs. In practice, these computational problems are often ignored. But when can this be done safely? This talk will introduce a framework for analyzing the behaviour of control software subject to computational problems. The focus will be on the development of tools that can certify when control software is able to fulfil the system requirements, despite the presence of computational problems.

2012 ACM Subject Classification Computer systems organization → Embedded systems; Software and its engineering → Software verification and validation

Keywords and phrases control systems, fault tolerance

Digital Object Identifier 10.4230/OASIcs.NG-RES.2023.1

Category Invited Talk

IRQ Coloring: Mitigating Interrupt-Generated Interference on ARM Multicore Platforms

Diogo Costa
Centro ALGORITMI / LASI, University of Minho, Portugal

Luca Cuomo
Huawei Pisa Research Center, Pisa, Italy

Daniel Oliveira
Centro ALGORITMI / LASI, University of Minho, Portugal

Ida Maria Savino
Huawei Pisa Research Center, Pisa, Italy

Bruno Morelli
Huawei Pisa Research Center, Pisa, Italy

José Martins
Centro ALGORITMI / LASI, University of Minho, Portugal

Fabrizio Tronci
Huawei Pisa Research Center, Pisa, Italy

Alessandro Biasci
Huawei Pisa Research Center, Pisa, Italy

Sandro Pinto
Centro ALGORITMI / LASI, University of Minho, Portugal

Abstract

Mixed-criticality systems, which consolidate workloads with different criticalities, must comply with stringent spatial and temporal isolation requirements imposed by safety-critical standards (e.g., ISO26262). This, per se, has proven to be a challenge with the advent of multicore platforms due to the inner interference created by multiple subsystems while disputing access to shared resources. With this work, we pioneer the concept of Interrupt (IRQ) coloring as a novel mechanism to minimize the interference created by co-existing interrupt-driven workloads. The main idea consists of selectively deactivating specific ("colored") interrupts if the Quality of Service (QoS) of critical workloads (e.g., Virtual Machines) drops below a well-defined threshold. The IRQ Coloring approach encompasses two artifacts, i.e., the IRQ Coloring Design-Time Tool (IRQ DTT) and the IRQ Coloring Run-Time Mechanism (IRQ RTM). In this paper, we focus on presenting the conceptual IRQ coloring design, describing the first prototype of the IRQ RTM on Bao hypervisor, and providing initial evidence about the effectiveness of the proposed approach on a synthetic use case.

2012 ACM Subject Classification Computer systems organization → Real-time system specification; Computer systems organization → Embedded software

Keywords and phrases IRQ coloring, Interrupt Interference, Mixed-Criticality Systems, Hypervisors, Bao, Arm

Digital Object Identifier 10.4230/OASIcs.NG-RES.2023.2

Funding *Diogo Costa*: Supported by Centro Algoritmi grant IRQ-COL-HYP-HUAWEI_01.
Daniel Oliveira: Supported by FCT grant 2020.04585.BD.
José Martins: Supported by FCT grant SFRH/BD/138660/2018.

1 Introduction

Currently, two major trends pose significant challenges for the certification of Mixed-criticality Systems (MCS) [13]. Firstly, with the increasing digitalization trend, there is a need to integrate an ever-growing number of rich functionalities for connectivity, visualization, and monitoring. Rich features have to co-exist with safety-critical workloads, and virtualization technology – due to the proven advantages for the size, weight, power, and cost (SWaP-C) – has been leveraged to provide the required spatial isolation [23, 24, 5, 11, 20]. Secondly, there is a well-established trend toward multicore. Modern high-end embedded computing platforms (typically powered by Arm Cortex-A) have evolved to highly heterogeneous designs that host multiple cores, featuring complex memory hierarchies and a myriad of accelerator such as general-purpose graphic processing units (GPUs), neural-processing units (NPUs) and even Field-Programmable Gate Array (FPGA). Although existing hypervisors have naturally evolved to support multicore designs, the problem is that the temporal isolation guarantees are generally weak due to the absence of mechanisms in the overall design to minimize the interference generated by shared microarchitectural resources [9].

The problem and difficulties deriving from the reciprocal interference generated by the contention of shared microarchitectural computational resources, such as (i) caches, (ii) bus, and (iii) main memory is a well-understood problem among the real-time systems community [11, 20, 16, 28, 27, 29, 2, 26, 7, 18, 17]. It is widely-recognized that without proper resource management strategies, interference can introduce variable delays that may hamper the timing predictability and real-time guarantees [1, 4, 3], compromising the desired Freedom From Interference (FFI) for MCSs. This issue is not linked to a particular piece of software, i.e., operating system (OS) and/or hypervisor, but rather a fundamental problem on the overall design of the lowest layer of software of the system stack. This problem is further exacerbated by interrupts and interrupt-driven workloads. Interrupts are typically asynchronous, and even synchronous events can divert the overall execution flow. Interrupt handlers normally have a different code locality, inherently leading to stress the shared components due to the expected last-level cache (LLC) misses and concurrent accesses to main memory. Even worst, a storm of interrupts (i.e., extremely high interrupt frequency) can be triggered for different reasons, leading to very effective Denial-of-Service (DoS) attacks.

Most of the proposed techniques to minimize interference created by shared resources in multicore real-time systems focus on (i) cache partitioning (via locking or coloring) [11, 20, 16, 15, 21], and (ii) memory throttling [28, 27, 2, 21, 6, 8]. Cache partitioning is a well-established technique that splits and assigns subsets of the shared cache to a specific OS process or Virtual Machine (VM). Cache locking avoids the eviction of cache lines by marking them as locked; cache coloring segments the available cache by reserving specific cache sets or ways to given cores. Memory throttling is a technique that limits the number of memory accesses of a specific workload in a given time window. In the case of virtualization, static partitioning hypervisors are the zeitgeist to implement such techniques e.g., Bao [20], Jailhouse [24], Xen Dom0-less [14], all implement cache coloring.

Despite the recognized efforts of the academic community, existing mechanisms are not perfect in terms of effectiveness and present several limitations. Firstly, cache locking support was deprecated and is not available in today's platforms powered by Armv8-A processors (and is not expected to be supported in next-generation platforms powered by Armv9-A and Armv8-R). Secondly, cache coloring has several drawbacks: (i) requires a virtual memory infrastructure (i.e., the existence of an MMU); (ii) precludes the use of (2 MiB) huge pages (which helps minimizing the overhead of the second stage translation); (iii) can impact the performance; and (iv) leads to significant memory waste and fragmentation. Finally, memory throttling completely suspends the CPU when a specific memory access threshold is reached in a given time window, thus not providing intermediate states or well-defined degradation modes. Furthermore, none of the existing mechanisms take interrupts into consideration.

In this paper, we pioneer and introduce the concept of Interrupt coloring (a.k.a. IRQ coloring), a novel mechanism to minimize the interference created by co-existing interrupt-driven workloads[1] and mitigate the effect of cascading failures when FFI cannot be completely guaranteed. The overall IRQ Coloring concept encompasses two artifacts: (i) the IRQ Coloring Design-Time Tool (IRQ DTT) and (ii) IRQ Coloring Run-Time Mechanism (IRQ RTM). The basic idea consists of selectively deactivating (or deferring) specific ("colored") interrupts if the QoS of critical workloads drops below a specific threshold. By selectively masking interrupts per the online assessment of the QoS of critical workloads, we provide fine-grained control to mitigate interference on critical workloads without fully suspending non-critical workloads. In this paper, we focus on describing the overall IRQ coloring concept, prototyping the IRQ RTM on the Bao hypervisor, and providing evidence about the effectiveness of the proposed approach on a synthetic use case mimicking an automotive application scenario. To the best of our knowledge, we are the first to propose this concept. Huawei has already filled and submitted a patent application.

2 Related Work

Several mechanisms to minimize interference have been proposed by academia and the research community. Table 1 summarizes and puts into perspective several works published in real-time venues. All works are compared across six dimensions: (i) target computer architecture (Arch); (ii) implementation leveraging COTS hardware (COTS); (iii) target system software component (System); (iv) target shared resource (Resource); (v) resource partitioning/budgeting at design-time (Static); and (vi) resource partitioning/budgeting and optimization at run-time (Dynamic). Next, we overview the related work, grouping them based on the target resource, i.e., last-level cache (LLC), memory bus (e.g., Dynamic Random Access Memory (DRAM) or SRAM), or both (cache and memory).

Table 1 Gap Analysis Table. ●: "yes". ○: "no".

	Arch	COTS	System	Resource	Static	Dynamic
MemGuard (2013) [29]	x86	●	OS	DRAM	○	●
Mancuso et al. (2013) [19]	Armv7-A	●	OS	LLC	●	●
Hassan et al. (2014) [12]	x86	○	OS	LLC	●	○
PALLOC (2014)[27]	x86	●	OS	DRAM	●	○
Kim et al. (2017) [15]	x86 & Armv7-A	●	Hyp.	LLC	●	○
Modica et al. (2018) [21]	Armv7-A	●	Hyp.	LLC & DRAM	●	●
Crespo et al. (2018) [6]	PowerPC	●	Hyp.	DRAM	●	●
Pinto et al. (2019) [22]	Armv8-M	●	Hyp.	SRAM	●	○
Kloda et al. (2019) [16]	Armv8-A	●	Hyp.	LLC & DRAM	●	○
Bao (2020) [20]	Armv8-A	●	Hyp.	LLC	●	○
BRU (2020) [8]	RISC-V	○	OS	DRAM	○	●
DNA (2021) [10]	x86	●	Hyp.	LLC+DRAM	●	●
IRQ Coloring	Armv8-A (Armv8-R)	●	Hyp.	IRQs	●	●

Cache Partitioning. Multicore platforms typically include a shared LLC and one or more levels of private caches. The cache partitioning technique splits and assigns subsets of the shared cache to a specific workload. Cache locking avoids the eviction of cache lines by

[1] processes in the case of OS and VMs in the case of hypervisor

marking them as locked, while cache coloring segments the available cache by reserving specific cache sets or ways to specific cores. R. Mancuso et al. [19] proposed a mechanism that introduces a novel mechanism known as “Colored Lockdown” by combining coloring and locking techniques. Kim et al. [15] proposed a multicore virtualization cache management system that uses the hypervisor page coloring mechanism to assign portions of the cache to VMs. M. Hassan et al. [12] address the problem of maintaining cache coherence in multicore real-time systems by modifying the Modified-Shared-Invalidate [25] coherence protocol. J. Martins et al. [20] implemented built-in support for cache coloring on Bao.

Memory Bandwidth Partitioning. Aiming to achieve temporal isolation through memory bandwidth regulation, H. Yun et al. [29] proposed MemGuard. To improve isolation and real-time performance, H. Yun et al. [27] proposed a mechanism that allocates private DRAM banks. A. Crespo et al. [6] proposed a controller technique, at the hypervisor level, to manage the execution of critical partitions for PowerPC multicore platforms. F. Farshchi et al. [8] presented a Bandwidth Regulation Unit (BRU), a customized RISC-V plug-and-play hardware module for per-core memory bandwidth control at fine-grained time intervals. From a different perspective, Pinto et al. [22] developed a lightweight hypervisor and implemented a static predictable shared resource management infrastructure for low-end Armv8-M microcontrollers.

Cache & Memory Bandwidth Partitioning. The previous works focus on mechanisms that individually target the Last-Level Cache (LLC) or the main memory. However, minimizing the interference impact in a single microarchitectural component is not enough. Hence, several authors attempted to provide methods that target the complete memory hierarchy, i.e., LLC and main memory. Modica et al. [21] proposed a cache coloring-based technique to achieve spatial isolation. Regarding the DRAM memory controller, the authors implemented a memory bandwidth reservation technique combined with the hypervisor’s scheduling logic to enhance temporal isolation. Kloda et al. [16] introduced a framework of software-based techniques to enhance memory access determinism in high-performance embedded systems. The authors proposed a Direct memory access (DMA)-friendly cache coloring combined with an invalidation-driven allocation technique. Recently, R. Gifford et al. [10] proposed a solution to mitigate two undesirable outcomes in current MCS: latency induced by shared resource interference and Worst-Case Execution Time (WCET) of critical tasks. Furthermore, the authors presented two techniques to mitigate the identified challenges: dynamic allocation (DNA) and deadline-aware allocation (DADNA).

3 Motivation: Interrupt-generated Interference in MCSs

In the context of MCS, interference generated by co-located interrupt-driven workloads is a particularly overlooked topic. Interrupts are typically used to notify the CPU about the occurrence of sporadic events, without requiring the CPU to stall while polling for that event. In the meantime, the CPU is free to perform any additional required computational operation. Generically, upon the occurrence of an interrupt (and omitting the low-level details of the overall interrupt entry process), the CPU execution is redirected to the so-called interrupt handler, which acknowledges the reception of the interrupt and then invokes an event-dependent piece of code. Typically, in safety-critical systems, interrupts are linked to application workloads, which are dispatched upon the occurrence of a particular event. For example, in Industrial Control Systems (ICS), the implementation of a PID controller, or in automotive systems, an antilock braking system.

The problem. Interrupts are typically asynchronous, and even synchronous events lead to unpredictable diversions in the overall computational execution flow. Upon the occurrence of an interrupt, the execution flow is redirected to the respective interrupt handler, which typically has a completely different code locality than the main execution path. Furthermore, interrupts are typically linked to application workloads, which are dispatched upon the occurrence of a particular event. This inherently stresses the microarchitectural shared components due to the expected LLC misses and subsequent accesses to the main memory. Even worst, a storm of interrupts can be triggered for different reasons (e.g., device driver bug, malfunction in a particular hardware device), which can create a DoS attack.

The evidence. We performed a few experiments to collect evidence supporting the identified problem. We mounted a synthetic use case of a system consisting of Bao hypervisor and two VMs running atop: a critical ASIL-D VM, and a QM VM. For the ASIL-D VM, we use a custom baremetal application that continuously writes into a buffer with the size of the LLC (1 MiB). A periodic CPU timer interrupt triggers this application. For the other QM VM, we use the very same baremetal application, triggered continuously by a software-generated interrupt. For the ASIL-D VM, we assigned 1 CPU, while for the QM VM we assigned 1, 2, or 3 CPUs, depending on the amount of interference we want to create (1 Interf VM, 2 Interf VM, and 3 Interf VM, respectively). Figure 1 depicts the assessed results. We collected the workload execution time and IRQ handling time, which corresponds to the interrupt latency, for the critical ASIL-D VM. The interference with 3 CPUs (3 Interf VM) can impact the workload execution time of the ASIL-D VM up to 2.48x.

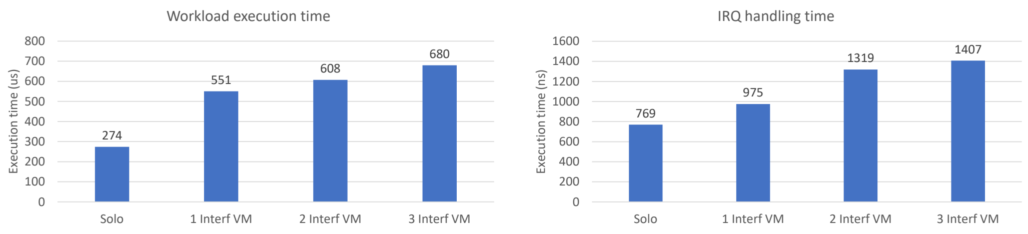


Figure 1 Interrupt-generated Interference - workload execution time and IRQ handling time with multiple VMs.

4 IRQ Coloring

IRQ coloring is a new concept that dictates that interrupts assigned to workloads (e.g., VMs) are classified according to a specific criticality level. The basic idea consists of selectively deactivating/deferring interrupts of non-critical workloads if the QoS of critical workloads drops below a specific threshold. By selectively masking interrupts per the online assessment of the QoS of critical workloads, it is expected that the overall interference from non-critical VMs to critical VMs is mitigated without fully suspending non-critical workloads.

Conceptual Design. Figure 2 illustrates the conceptual design of the IRQ Coloring technique, which follows a budget-based approach. A set of budgets of interrupts (B_i) are assigned to each workload for a well-defined period of time (P). During the period (P), each VM can trigger a certain number of interrupts; however, if the VM exceeds the specified budget (B_i), the degradation mode is updated, and some interrupts are disabled $\textcircled{1}$. Since the

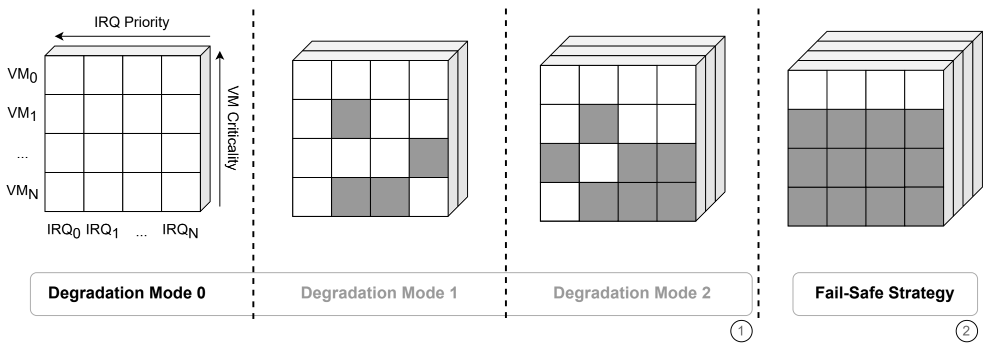


Figure 2 IRQ coloring conceptual design.

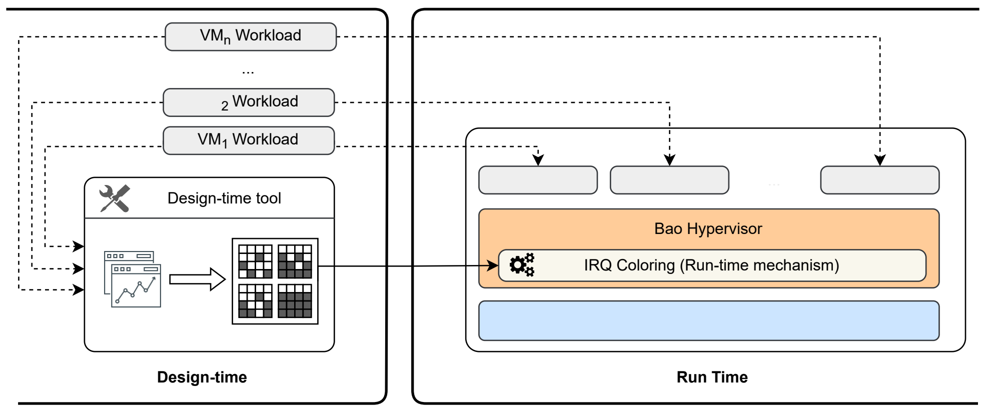


Figure 3 IRQ coloring system overview.

impact of each interrupt (assigned to a particular VM) in the overall QoS of the system is not equal and linear, the weight of each interrupt in the overall budget B_i will be measured by assessing specific QoS metrics. The conceptual design takes this into consideration and is reflected in Figure 2). Lastly, if applying different degradation modes is not enough to guarantee the QoS of the higher-criticality VM, the system will enter fail-safe mode. At this stage, only interrupts from the higher-criticality VM are kept enabled ②.

System Overview. Defining which and when interrupts must be disabled is the main research question of the IRQ Coloring technique. The design goals encompass: (i) achieving the required QoS on higher criticality VMs, (ii) maintaining intermediate execution of lower criticality VMs (intermediate states under specific degradation modes), and (iii) minimizing the performance impact incured by the overall mechanism. To achieve these three goals, we conceived a system with two major components, with the bulk of logic performed at design time. Figure 3 presents the high-level system view, encompassing the IRQ Coloring Design-Time Tool (IRQ DTT) and the IRQ Coloring Run-Time Mechanism (IRQ RTM).

1. IRQ Coloring Design-Time Tool (IRQ DTT). The IRQ DTT goal is twofold. Firstly, based on the target VM workload, set the profile of each interrupt-driven workload, which helps estimate the worst-case execution time (WCET) by providing information about the execution time and the microarchitectural state (i.e., caches, shared bus). Secondly, based on the established profile of the workload and assigned interrupts altogether with the specification of VMs criticality, produce an optimized configuration table (representing the masking map to be enforced in each degradation mode) which will feed the IRQ RTM. The output of the DTT will then be used to feed the RTM implemented at the hypervisor level, and it consists of two artifacts: (i) the budgets assigned to each degradation mode of the diverse VMs, and (ii) the masking maps to be used in the multiple degradation modes (i.e., the content of the optimized configuration table).

2. IRQ Coloring Run-Time Mechanism (IRQ RTM). The IRQ RTM, implemented as a mechanism at the hypervisor level, will mainly collect specific metrics from the hardware performance counters (e.g. Performance Monitor Unit (PMU)), and based on the optimized table produced from the IRQ DTT, will selectively disable interrupts upon the occurrence of specific events. Next, we describe the IRQ RTM implemented in Bao in more detail.

4.1 IRQ Coloring Run-Time Mechanism (IRQ RTM)

The implementation of the IRQ RTM has two main assumptions, which we highlight below.

Assumption 1. Workloads Profiling

The interference generated by co-located VMs running onto the same hardware platform is dependent on the VMs' workloads. We assume that workloads are available, known a priori, profiled offline, and this data is directly or indirectly passed to the IRQ DTT.

Assumption 2. Masking Maps

We assume that the IRQ DTT, based on the profile of the VMs workloads, produces an optimized configuration table (a.k.a. masking map) with "colored" interrupts and associated budgets.

The IRQ RTM relies on a budget-based implementation, which means that each VM can trigger a given number of interrupts in a given period of time (P). Typically, these approaches assume that a VM can process workloads until the defined budget is exhausted. After that, typically the CPU enters idle mode. However, IRQ coloring points to intermediate guarantees, i.e., VM interrupts will be gradually disabled in order to minimize the generated interference created on the critical VM. In this way, instead of assigning a budget to each VM, we assign a buffer of budgets (i.e., a buffer with D values, each representing an activation point of a different degradation mode). Thus, the system configuration must contemplate the setup of the table $B_{N,D}$, where N corresponds to the number of co-existing VMs into the system, and D corresponds to the total number of degradation modes. The high-level implementation of the IRQ RTM is shown in Algorithm 1.

Figure 4 depicts the IRQ RTM architecture, implemented as a plug-in mechanism in Bao. The IRQ RTM actuates between the GIC distributor (GICD) and the CPU Interfaces (GICC). To simplify the run-time operation, each CPU uses the PMU to track the number of triggered interrupts, triggering an interrupt, at the hypervisor level, when the counter overflows, i.e., when the assigned budget (B) is exceeded. When the PMU interrupt is triggered, the IRQ RTM is in charge of masking a set of interrupts based on the masking map generated by the IRQ DTT. At this point, the run-time mechanism is triggered, leading to the update of the degradation mode and the masking of the interrupts corresponding to the degradation mode map (IRQ_MAP_D). The process repeats until processing all the

Algorithm 1 IRQ Coloring RTM - Implementation.

```
1:  funtion periodic_timer_handler;
2:  begin
3:  for $VM = 1, 2, \ldots$ do
4:      $VM_{Degradation\_mode} = 0$
5:      Set PMU to generate overflow interrupt at $VM_{B_0}$
6:      for $IRQ = 1, 2, \ldots$ do
7:          Unmask $IRQ$
8:      end for
9:  end for
10: Re-schedule timer period
11: funtion pmu_overflow_interrupt_handler;
12: begin
13: for $IRQ = 1, 2, \ldots \in IRQ\_MAP_{Degradation\_mode}$ do
14:     Mask IRQ
15: end for
16: $VM_{Degradation\_mode} \leftarrow VM_{Degradation\_mode} + 1$
17: if $VM_{Degradation\_mode} < Max\_Degradation\_modes$ then
18:     Set PMU to generate overflow interrupt at $B_{VM,Degradation\_mode}$
19: end if
```

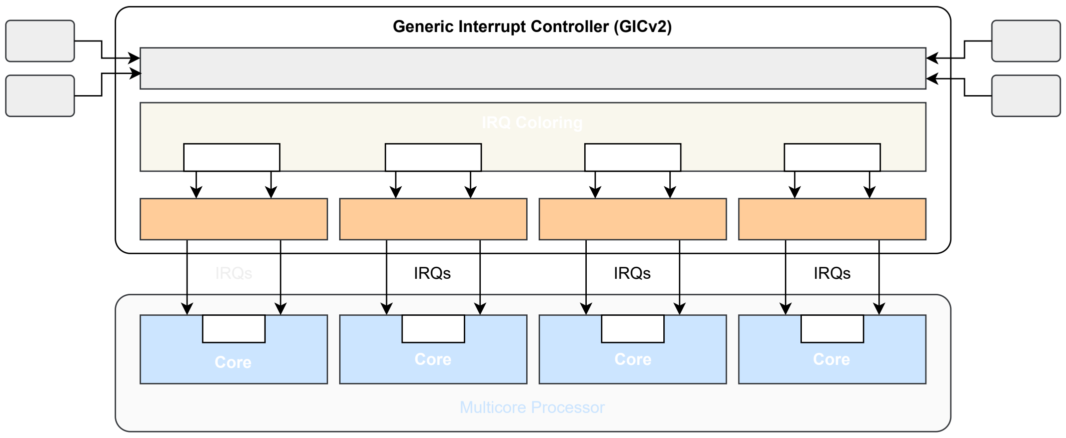


Figure 4 IRQ Coloring RTM: system architecture.

D degradation modes. If a non-critical VM reaches this point, the IRQ RTM triggers the fail-safe strategy, i.e., all interrupts from all system VMs, except the most critical one, are disabled. In this case, the interrupts are re-enabled when the periodic timer expires.

5 Evaluation

In this section we describe the evaluation setup (Section 5.1) and present and discuss the evaluation results (Section 5.2).

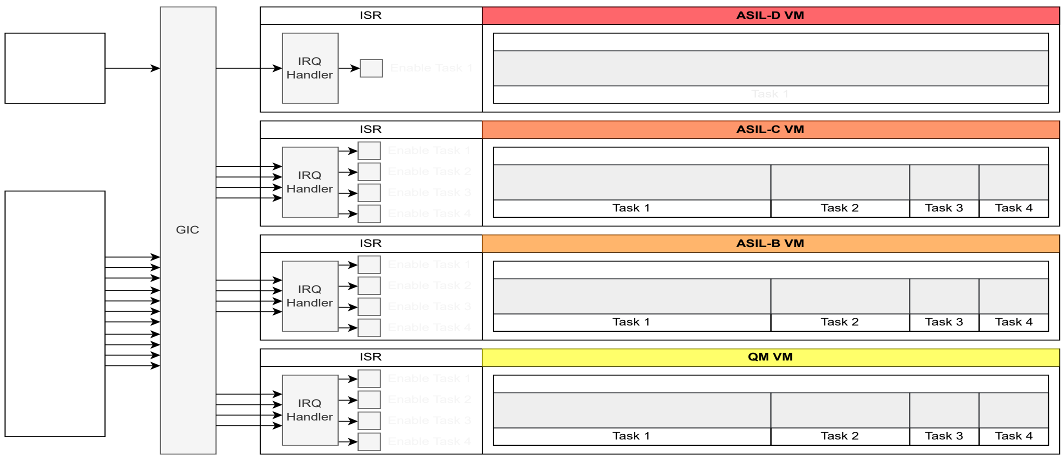


Figure 5 IRQ Coloring evaluation (synthetic) use case.

5.1 Evaluation Setup

Hardware Platform. Experiments were carried out on a Xilinx ZCU104 evaluation board, featuring a Zynq Ultrascale+ ZU7EV SoC. This platform features a quad-core Arm Cortex-A53 running at 1.2GHz. Each CPU has a private 32KiB L1 instruction and data caches and an unified L2 1MiB cache. The cluster features the GIC-400 (GICv2).

Use Case. To evaluate the raw effectiveness of the IRQ coloring mechanism, we mounted a synthetic use case, aiming at mimicking a real-world automotive ECU consisting of four different sub-systems: (i) a critical ASIL-D workload, emulating an electric power steering system; (ii) an ASIL-C workload, emulating an active suspension system or an engine management system; (iii) an ASIL-B workload, emulating for example a radar cruise control; and (iv) a QM workload, emulating a non-critical third-party software. We implemented the four VMs on top of the Bao [20]. Furthermore, to mimic the I/O interrupt generation, we have implemented a custom hardware device on the programmable logic of the Zynq UltraScale+ SoC, which can trigger up to 16 simultaneous interrupts.

VM Workload. The workload of each VM is summarized in Figure 5. For the ASIL-D VM, we use a custom baremetal application that continuously writes into a buffer with the size of the LLC (1 MiB). This application is triggered by a periodic CPU timer interrupt at each 500 us. For the other VMs, i.e., ASIL-C, ASIL-B, and QM, we use an identical baremetal application with a small difference. Each VM has assigned four different interrupts, all triggered by the custom hardware module deployed on the programmable logic of the Zynq UltraScale+ SoC. In this case, rather than writing into the entire buffer, each interrupt triggers the workload that writes in a subset of the buffer. To evaluate the effect of different workloads, the buffer is partitioned into four parts: (i) one with 512KiB (50% of the LLC); (ii) one with 256KiB (25% of the LLC); and (iii) two with 128KiB each (12.5% of the LLC). Since the Cortex-A cluster of the ZCU104 has 4 CPUs, we assign one CPU to each VM.

Measurement Tools. We use the Arm PMU to collect microarchitectural events on benchmark execution. The selected events include execution cycle count, data L1 and L2 cache misses and cycles of bus accesses. The PMU cycle counter is used to calculate the execution time.

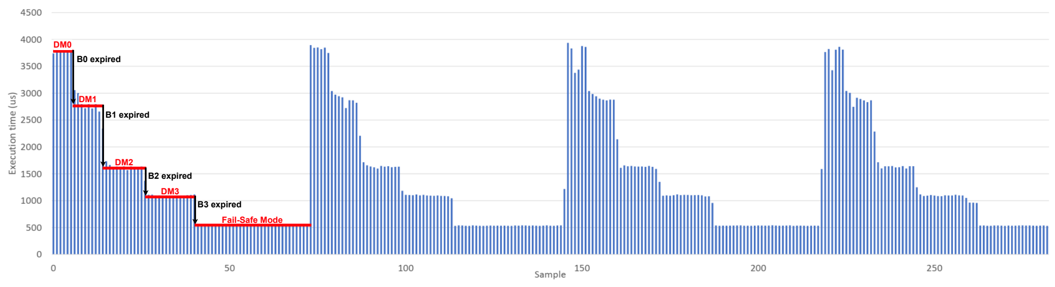


Figure 6 IRQ RTM - Execution time of ASIL-D workload for multiple degradation modes.

5.2 Evaluation Results

Intermediate degradation modes. To validate the intermediate degradation modes implemented with the IRQ RTM, we used a simplified version of the use case described above. For this particular experiment, we have used only the ASIL-D and QM VMs, with respective identified workloads. We have also defined four degradation modes: (i) degradation mode 1 (3 interrupts on QM VM active), which is activated after spending a total budget of 1000 interrupts; (ii) degradation mode 2 (2 interrupts on QM VM active), which is activated once the total budget of 4000 is reached; (iii) degradation mode 3 (1 interrupt on the QM VM active), which is activated after triggering 5000 interrupts; and, finally, (iv) fail-safe strategy (all interrupts are disabled on the QM VM) which is activated after triggering another 5000 interrupts. The budgets are re-established after a well-defined period of 150ms. Figure 6 presents the execution time of the ASIL-D VM workload. We started by assessing the ASIL-D VM running without interference, which corresponds to the baseline. For this case, the workload execution time was, on average, 547 us. By enabling the QM VM interrupts, the average ASIL-D workload execution time increases by 6.99x (3823 us) - DM0. Then, after expiring the first budget (B0), the first degradation mode is activated, reducing the performance degradation by 26.8% (2796us) - DM1. Then, after activating additional degradation modes, the ASIL-D workload performance is boosted due to reduction of the QM VM interference. For instance, when the budget of the degradation mode 1 expires, the second degradation mode - DM2 - is activated, allowing a reduction of the relative performance overhead of 57.32% (1632us); when the third degradation mode - DM3 - is activated, it reaches a reduction of 70.87% (114us). Ultimately, triggering the fail-safe strategy brings the ASIL-D workload performance to native execution, i.e., 550 us.

Relative performance overhead. After validating the effectiveness of intermediate degradation modes, we resort to the original use case with four VMs to understand the average performance impact of the IRQ coloring (*+IRQ_col*) compared to the vanilla system (baseline) and the cache coloring technique (*+cc*). For the cache coloring, we assigned two colors to each partition (25% of the L2 cache to each VM). Figure 7 depicts the ASIL-D workload execution cycles, as well as a few microarchitectural events, i.e., L1 misses, L2 misses, and bus cycles. We can draw two main conclusions. Firstly, in contrast to the cache coloring, the IRQ coloring technique does not incur any noticeable impact on the overall performance of the ASIL-D VM. Secondly, when the ASIL-D VM is under the interference of the QM VM, the average performance overhead is considerably smaller in the case of the IRQ coloring, which indicates that this technique is more effective than the cache coloring. As expected, per the results collected for the microarchitectural events, this is explained by the reduced number of L1 and L2 cache misses (the contention on the L1 and L2 caches is smaller), as well as the reduced number of accesses to the main memory.

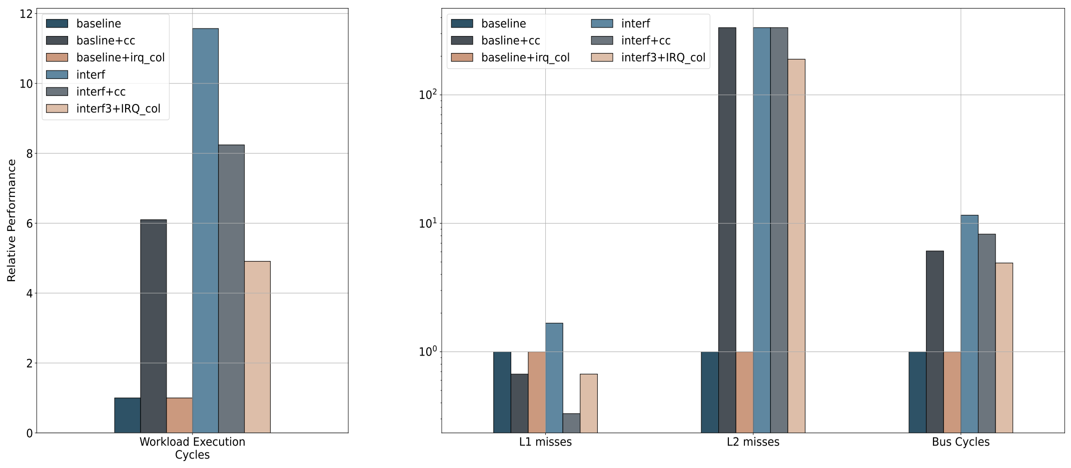


Figure 7 Average Relative performance overhead and microarchitectural events for the vanilla system, for the cache coloring, and IRQ coloring.

6 Conclusion

With this paper, we propose the concept of Interrupt (IRQ) coloring as a novel mechanism to minimize the interference created by co-existing interrupt-driven workloads. We focused on presenting the conceptual IRQ coloring design, describing the prototype of the IRQ RTM on Bao, and evaluating the implemented mechanisms on a synthetic use case. Preliminary results have demonstrated advantages compared to other state-of-the-art techniques (e.g., cache coloring), and findings are encouraging additional research to advance the maturity of the technique, as well as a comprehensive evaluation under more realistic setups. Additionally, we are currently designing the DTT infrastructure. The full combination and integration of RTM and DTT have the potential to further unlock novel designs and optimize and explore trade-offs for reducing interference in multicore platforms. As part of our roadmap, we also plan to iterate and complete the formal model for the IRQ coloring mechanism.

References

1 Jaume Abella, Carles Hernández, Eduardo Quiñones, Francisco J Cazorla, Philippa Ryan Conmy, Mikel Azkarate-Askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10, 2015.

2 Michael G Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention, 2019.

3 Francisco J Cazorla, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella, and Tullio Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Computing Surveys (CSUR)*, 52(1):1–35, 2019.

4 Francisco J Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, et al. Proartis: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):1–26, 2013.

5 Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core devices for safety-critical systems: A survey. *ACM Computing Surveys (CSUR)*, 53(4):1–38, 2020.

6 Alfons Crespo, Patricia Balbastre, José Simó, Javier Coronel, Daniel Gracia Pérez, and Philippe Bonnot. Hypervisor-based multicore feedback control of mixed-criticality systems. *IEEE Access*, 6:50627–50640, 2018.

7 Dakshina Dasari, Benny Akesson, Vincent Nelis, Muhammad Ali Awan, and Stefan M Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *8th IEEE international symposium on industrial embedded systems (SIES)*, pages 39–48, 2013.

8 Farzad Farshchi, Qijing Huang, and Heechul Yun. Bru: Bandwidth regulation unit for real-time multicore processors. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 364–375, 2020.

9 Gabriel Fernandez, Jaume Abella Ferrer, Eduardo Quiñones, Christine Rochange, Tullio Vardanega, and Francisco Javier Cazorla Almeida. Contention in multicore hardware shared resources: Understanding of the state of the art. In *14th International Workshop on Worst-Case Execution Time Analysis*, pages 31–42, 2014.

10 Robert Gifford, Neeraj Gandhi, Linh Thi Xuan Phan, and Andreas Haeberlen. Dna: Dynamic resource allocation for soft real-time multicore systems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 196–209, 2021.

11 Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS)*, volume 133, pages 27:1–27:25, 2019.

12 Mohamed Hassan, Anirudh M Kaushik, and Hiren Patel. Predictable cache coherence for multi-core real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–246, 2017.

13 Thomas A Henzinger and Joseph Sifakis. The embedded systems design challenge. In *International Symposium on Formal Methods*, pages 1–15, 2006.

14 Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *5th IEEE Consumer Communications and Networking Conference*, pages 257–261, 2008.

15 Hyoseung Kim and Ragunathan Rajkumar. Predictable shared cache management for multi-core real-time virtualization. *ACM Transactions on Embedded Computing Systems (TECS)*, volume 17, 2017.

16 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 *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14, 2019.

17 Ondrej Kotaba, Jan Nowotsch, Michael Paulitsch, Stefan M Petters, and Henrik Theiling. Multicore in real-time systems–temporal isolation challenges due to shared resources. In *16th Design, Automation & Test in Europe Conference and Exhibition*, 2013.

18 Andreas Löfwenmark and Simin Nadjm-Tehrani. Understanding shared memory bank access interference in multi-core avionics. In *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

19 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, 2013.

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

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

22 Sandro Pinto, Hugo Araujo, Daniel Oliveira, Jose Martins, and Adriano Tavares. Virtualization on trustzone-enabled microcontrollers? voilà! In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304. IEEE, 2019.

23 Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. Ltzvisor: Trustzone is the key. In *29th Euromicro Conference on Real-Time Systems (ECRTS)*, 2017.

24 Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look mum, no vm exits! (almost). *arXiv preprint*, 2017. `arXiv:1705.06932`.

25 Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. In *Synthesis lectures on computer architecture*, volume 6, pages 1–212, 2011.

26 Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, et al. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010.

27 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014.

28 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *24th Euromicro Conference on Real-Time Systems*, pages 299–308, 2012.

29 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 *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

Beyond the Threaded Programming Model on Real-Time Operating Systems

Erling Rennemo Jellum
NTNU, Trondheim, Norway

Shaokai Lin
University of California at Berkeley, CA, USA

Peter Donovan
University of California at Berkeley, CA, USA

Efsane Soyer
University of California at Berkeley, CA, USA

Fuzail Shakir
University of California at Berkeley, CA, USA

Torleiv Bryne
NTNU, Trondheim, Norway

Milica Orlandic
NTNU, Trondheim, Norway

Marten Lohstroh
University of California at Berkeley, CA, USA

Edward A. Lee
University of California at Berkeley, CA, USA

Abstract

The use of a real-time operating system (RTOS) raises the abstraction level for embedded systems design when compared to traditional bare-metal programming, resulting in simpler and more reusable application code. Modern RTOSes for resource-constrained platforms, like Zephyr and FreeRTOS, also offer threading support, but this kind of shared memory concurrency is a poor fit for expressing the reactive and interactive behaviors that are common in embedded systems. To address this, alternative concurrency models like the actor model or communicating sequential processes have been proposed. While those alternatives enable reactive design patterns, they fail to deliver determinism and do not address timing. This makes it difficult to verify that implemented behavior is as intended and impossible to specify timing constraints in a portable way. This makes it hard to create reusable library components out of common embedded design patterns, forcing developers to keep reinventing the wheel for each application and each platform. In this paper, we introduce the embedded target of Lingua Franca (LF) as a means to move beyond the threaded programming model provided by RTOSes and improve the state of the art in embedded programming. LF is based on the reactor model of computation, which is reactive, deterministic, and timed, providing a means to express concurrency and timing in a platform-independent way. We compare the performance of LF versus threaded C code – both running on Zephyr – in terms of response time, timing precision, throughput, and memory footprint.

2012 ACM Subject Classification Computer systems organization → Embedded software

Keywords and phrases Real time, concurrency, reactors, Lingua Franca, RTOS

Digital Object Identifier 10.4230/OASIcs.NG-RES.2023.3

Supplementary Material *Software (Source Code)*: `https://github.com/lf-lang/lingua-franca/tree/v0.4.0`

Funding This work was supported by the Research Council of Norway (RCN) grant number 309328, Multi-Sensor Data Timing, Synchronization and Fusion for Intelligent Robots, grant number 327538, the center of excellence NTNU AMOS, grant number 223254, and the iCyPhy Research Center at UC Berkeley.

Acknowledgements We want to thank Matthew Chorlian, Arthur Deng, Abanob Bostouros and Christian Menard for their work on porting the Savina benchmark to the Lingua Franca C target, as well as Hannes Klein and Felix Wittwer for their work on the original port of the Savina benchmark to the Lingua Franca C++ target. We would also like to acknowledge Soroush Bateni and Alexander Schulz-Rosengarten for their work on the Lingua Franca C runtime.

1 Introduction

Microcontrollers are ubiquitous in embedded applications, such as consumer electronics, industrial automation, robotics, and automotive systems. Such applications are concurrent, event-driven, and in many cases, time-sensitive. Traditionally, these systems have been programmed in C, based on a limited shared-memory concurrency model offered by interrupts. However, such programs have long been known to be notoriously difficult to get right [24, 5]. This has contributed to serious bugs and vulnerabilities, sometimes with catastrophic results [11, 10, 14].

Real-time operating systems (RTOSes) offer a richer version of the shared-memory concurrency model through *threads*. Threads enable the user to decompose their application into multiple concurrent activities. The RTOS offers low-level primitives for controlling the scheduling, synchronization, timing, and communication between threads. Just like interrupts, the threaded programming model admits nondeterminism [12] and leaves the error-prone job of synchronizing the threads' accesses to shared variables to the programmer [25]. A concurrency primitive that popular operating systems like Zephyr [4] provide as an alternative to threads is *callbacks*, which are invoked from interrupt service routines [20]. Callbacks do not address the problem of nondeterminism either, but they do create the problem of control flow inversion [5], which makes programs much harder to understand [18]. The RTOS introduces a notion of time, which is useful when designing time-aware applications. However, the timing is too imprecise for many applications. The RTOS will use regular interrupts, called *systicks*, to measure time and provide timing abstractions. The interval between the systicks is user-configurable but typically in the order of milliseconds. For many applications, this level of accuracy is not sufficient [9]. Developers can resort to implementing the timed behavior through callbacks, but this will expose them to the aforementioned problems.

In this paper, we propose Lingua Franca (LF) [16] as a means to enable intuitive concurrent, reactive, and time-aware programming of microcontrollers. LF offers lightweight concurrent objects called reactors, which can be composed in a network. Reactors do not share state but communicate through message passing via named ports. Deterministic concurrency is ensured on the basis of the topology of the reactor network. LF has a semantic notion of time that enables specification of arbitrary timing behavior of a program. Being a coordination language, not a general-purpose programming language, LF lets the programmer express computation in a target language of their choice. The Zephyr support implemented and evaluated in this paper is based on the C target. We compare the implementation of typical patterns in embedded system software using the timed reactor-based concurrency model against achieving the same kinds of behavior with threads.

The remainder of the paper is organized as follows. Section 2 briefly introduces the reactor-oriented paradigm followed by LF and its suitability for embedded system design. Section 3 takes a look at related work. Section 4 introduces the Zephyr support layer we developed for LF. In Section 5 we compare the performance of LF on Zephyr versus threaded C code running directly on Zephyr. Conclusions are provided in Section 6.

2 Embedded Programming in Lingua Franca

In the reactor-oriented programming paradigm followed by LF, concurrency is exposed by the programmer through the composition of stateful components called *reactors* that encapsulate event-triggered *reactions*, leaving the task of concurrency management to the LF compiler and runtime. Programmers need not deal with locks, semaphores, or other low-level synchronization mechanisms. Any two simultaneously triggered reactions that do not have any data dependencies between them may execute in arbitrary order (or in parallel, on a multicore system) without execution order affecting the effects of those reactions. It is the runtime system, not the programmer, that is in charge of ensuring that data dependencies are observed. This is what makes LF programs deterministic by default.

When events are not logically simultaneous, the LF runtime guarantees that they will appear to be processed in order according to their positions, called *tags*, on a totally ordered logical timeline. Through its correspondence to physical time, as measured by the system clock, this logical timeline also determines the program's real-time behavior. Events with well-defined logical times are produced by hooking up reactors to *timers* that produce events periodically, or by scheduling events in reaction code via *actions*. LF's timing constructs can be parameterized to let the runtime control the offsets, periods, and spacing between events. Logical delays and deadlines can be used to specify timing constraints and to handle possible timing violations.

LF separates a declarative specification of a program's coordination fabric from the program's imperative code, which is contained in reaction bodies and written in a target language of choice. The LF toolchain capitalizes on this by providing automatically synthesized interactive diagrams in the IDE. In the diagrams, reactors are shown as rounded boxes. Small black triangles denote ports, and reactions are represented by chevrons. Reactions can be triggered by events originating from upstream ports or actions, connected via dashed lines. There are also distinguished startup and shutdown triggers that, respectively, have events at the very first and very last time instant of program execution. Dashed lines also connect a reaction to its *effects*, which are downstream ports or actions that it can produce events on. Events produced on ports are always instantaneous; those produced on actions are always scheduled to occur at a future time.

To express modal behaviors, where the software has to switch between different modes of operation, LF also has syntax for grouping a reactor's elements in mutually exclusive regions called *modes*. A modal reactor has a single initial mode and can have several other modes. A transition from one mode to another is initiated in reaction code and requires the reaction to specify the target mode among its effects. A mode can be entered via a reset transition (the default) or a history transition. The former initializes all timers and resets any state marked to be reset; the latter resumes operation where it left off when the mode was last exited. Modes and transitions are rendered visually in the diagrams as highlighted areas with arcs between them. The initial mode is marked by a thicker border.

The LF runtime can be compared to an RTOS kernel. Like a kernel, it supports message-passing and context-switching between concurrently executing components. The message-passing is done by setting up direct pointers from input ports to the output ports they are connected to on. Ports carry immutable data by default which enables zero-copy communication. There are two types of context-switches performed by the LF runtime. The first, referred to as *reaction-switching*, involves scheduling a new reaction at the same tag a the previously executed reaction. The second type of context-switch requires advancing the logical time of the program. This is referred to as *logical time advancement*. It might include putting the system into low-power sleep to wait for the physical time to reach that particular value.

2.1 Embedded design patterns

In the following sections we will show how three common design patterns in embedded systems can be expressed easily using Lingua Franca.

```
target C;

reactor ButtonDebounce {
    preamble {=
        void * button_press_action; // Pointer to physical action
        void gpio_pin_isr() {
            lf_schedule(button_press_action, 0);
        }
    =}

    physical action button_press(0, 10 msec)

    reaction(startup) -> button_press {=
        // Setup GPIO pin and register ISR
        ...
        // Make physical action available externally
        button_press_action = button_press;
    =}

    reaction (button_press) {=
        // Handle
    =}
}
```

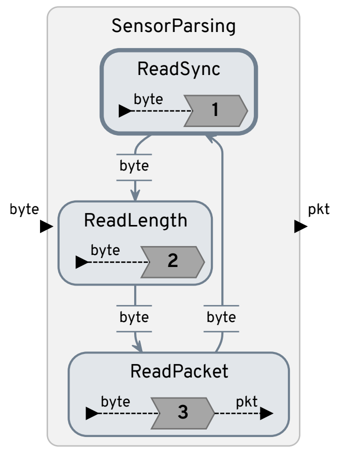
Figure 1 Button debouncing in Lingua Franca.

2.1.1 Button debouncing

When a button is pressed, it may generate multiple spurious open and close transitions. Debouncing is the process of filtering out transitions that are so close in time that they likely are not due to an actual button press or release. LF's timed semantics makes such patterns simple to express. To capture asynchronous events, like button presses, and pin them to LF's logical timeline, a *physical* action is used, which can be scheduled from an external thread or ISR. The tag of such an event is derived not from the runtime's current logical time (as an event scheduled on a *logical* action would), but from the systems's physical time. An action can be assigned a minimum spacing, which will make the scheduler reject events scheduled too close to one another. Fig. 1 shows how this is expressed in the C target of Lingua Franca. The physical action *button_press* has a minimum spacing of 10 ms, and the runtime system handles the debouncing.

2.1.2 Sensor parsing

Consider the common application of parsing an incoming packet from an external sensor connected through a digital communication interface. Figs. 2b and 2a show the implementation and diagram of a generic packet parser reactor in LF. Parsing a stream of bytes is conveniently expressed as a state machine. LF has native support for encoding state machines through modal reactors. The *SensorParsing* reactor has an input port for incoming bytes and an output port for the parsed packet. How the reactor responds to data on its input port depends on its current mode. In the initial mode, *ReadSync*, the incoming bytes are compared against the known sync ID of the packet. Upon finding the sync bytes, the reactor performs a mode change to *ReadLength*, where it reads the length field of the packet. When the length has been decoded, the reactor moves to the *ReadPacket* mode, reads the packet, and outputs the result on the *pkt* output port.



(a) Diagram.

```
reactor SensorParser {
  input byte:char
  output pkt:sensor_pkt_t;

  initial mode ReadSync {
    reaction(byte) -> ReadLength {=
      // Search for sync-byte ...
      lf_set_mode(ReadLength);
    =}
  }

  mode ReadLength {
    reaction(byte) -> ReadPacket {=
      // Read length field ...
      lf_set_mode(ReadPacket);
    =}
  }

  mode ReadPacket {
    reaction(byte) -> pkt {=
      // Read packet into p ...
      lf_set(pkt, p);
    =}
  }
}
```

(b) Source code.

Figure 2 A sensor packet parser in Lingua Franca.

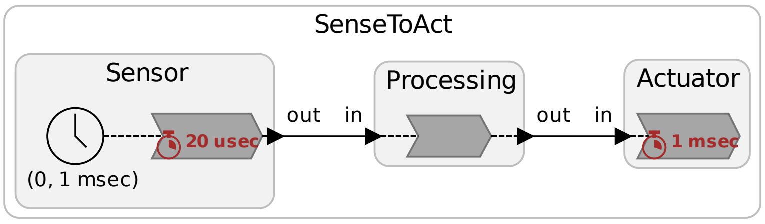


Figure 3 Tight control loop application in Lingua Franca.

2.1.3 Control loops

Another common design pattern is that of tight control loops in, for example, robotics and control. Consider the LF program depicted Fig. 3. It consists of a *Sensor* reactor that reads from a sensor every 1 ms. The sensor values are passed downstream to the *Processing* reactor, which implements a control algorithm. Finally, the control set points calculated by *Processing* are applied to the actuators in the *Actuator* reactor.

Notice that there are two deadlines. First, the *Sensor* reaction is associated with a 20 us deadline. This is a way of expressing a bound on how much jitter we accept in our sensor readings. Likewise, the *Actuator* reactor has a deadline of 1 ms. This corresponds with the period of the control loop and expresses that the overhead starting from obtaining a reading from *Sensor* up to right before invoking *Actuator* should be less than 1 ms. The timed semantics of LF provides an intuitive way to express such control loops and their timing requirements.

3 Related Work

There is a rich body of work exploring ways of raising the abstraction level for embedded software design. SynchronVM [20] is a virtual machine running on top of Zephyr OS or ChibiOS targeting microcontrollers. The underlying concurrency model is based on message passing and resembles communicating sequential processes (CSP) [6]. In SynchronVM one can specify a time window in which a synchronous message passing should occur. The time window informs the scheduling of the processes but does not have a well defined semantics. The underlying model of computation, CSP, is fundamentally nondeterministic as it allows processes to wait for events on multiple channels and synchronize with the first available.

MicroPython [19] is a project bringing the interpreted and garbage-collected high-level language Python to microcontrollers. The MicroPython interpreter supports REPL (Read-Evaluate-Print-Loop) interaction, which greatly simplifies prototyping. However, MicroPython requires FLASH memory on the order of 256KB, and its garbage collection and dynamic typing make it unsuited for real-time and safety-critical applications.

Medusa [2] is a concurrent programming language based on the Actor model [1]. Medusa is an extension of Python and runs an Owl Python runtime for microcontrollers. The Owl runtime has a code size of 38KB, and each Actor requires 100B of RAM. In Medusa, the hardware is also modeled as actors, and the peripherals communicate with the rest of the system through message passing performed by the interrupt service routines. Medusa does not have a semantic notion of time, and the underlying actor model is nondeterministic.

xC [22] is a C-based programming language designed for the Xcore [17] CPU architecture from XMOS. xC also has CSP as its underlying model, but it targets a CPU architecture with hardware support for processes and rendezvous. As such, there is no runtime providing a message-passing API. xC has language primitives representing hardware timers, which provide sleeping and waiting. The Xcore architecture is timing-predictable, and the tools support worst-case execution time analysis.

4 Design and Implementation

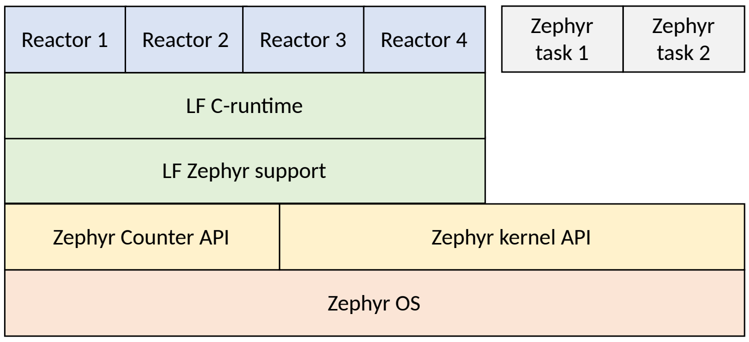


Figure 4 Lingua France runtime stack.

Fig. 4 shows the software stack supporting LF on embedded systems. The Zephyr OS [4] forms the foundation of this stack. This was chosen over a bare-metal target for multiple reasons. First of all, the APIs of the Zephyr kernel and drivers are platform independent, and, at the time of writing, Zephyr supports more than 400 boards. Zephyr also comes with many useful drivers for communication protocols like Bluetooth Low Energy (BLE), Ethernet, USB, and CAN buses. This makes Zephyr an attractive framework even for single-threaded

Table 1 LF code and data size (B is bytes).

	Runtime	Reactor	Reaction	Timer	Action	Mode	Port (In/Out)	Event
Code size	9.8KB(1KB)	96B	62B	88B	174B	36B	32B/16B	–
Data size	266B(170B)	72B	80B	60B	72B	28B	104B/40B	32B

applications. Using Zephyr also enables simple integration of LF programs with existing libraries like TCP/IP stacks or clock synchronization implementations. These can simply execute in separate threads and be scheduled by the Zephyr kernel. Lastly, for running LF on multicore platforms, a threading library is needed in order to make use of all the cores. Zephyr implements a POSIX-compliant threading library.

In this paper we focus on the single-threaded LF C runtime. Like any LF target port, this one must implement a few key functions to (1) enter and leave critical sections; (2) sleep for a duration or until interrupted asynchronously; and (3) read a monotonically increasing physical clock. A critical section is created by disabling all interrupts; this functionality is exposed by the kernel API. Sleeping can be achieved in various ways, but the Zephyr kernel provides the function *k_sleep* which puts the calling thread to sleep for a number of clock ticks. A physical clock can be read using *k_cycle_get_*32. However, on many platforms, this clock, which underlies the kernel services, is a low-frequency 32KHz clock. This might not be sufficient for high-precision control or highly dynamic robots [9]. Most platforms will also have high-frequency clocks which can drive hardware timers. Such hardware timers can be used through the platform-independent Counter API. The Counter API is not part of the kernel API and might not be implemented for all platforms.

For multi-threaded support, a full POSIX-like implementation must be provided. Zephyr does implement the POSIX API, but again the granularity of the timing services is too coarse. Here, we use the Counter API again to provide high-precision sleeps.

For platforms that do not have hardware timers or do not implement the Counter API, the Zephyr kernel's timing primitives are used. This drastically reduces precision, but it enables testing and running LF programs on QEMU emulations [3].

5 Evaluation

The evaluation was done on a NRF52832 development kit from Nordic Semiconductor [21]. It has an Arm Cortex M4 microcontroller with 64KB RAM and 512KB flash clocked at 64MHz. Benchmarks were compiled with the west build tool version 0.14.0 using Zephyr version 3.2.0.

5.1 Memory footprint

Memory footprint is a critical property for most microcontroller applications. In the following we provide a detailed evaluation of both the flash and RAM footprints of LF.

5.1.1 Code size

The top row of Table 1 shows the code size of different LF components. The runtime itself consumes less than 10KB and can be further reduced by 3KB if the program has no modal reactors. The 1KB in the parenthesis is the implementation of the Zephyr Counter driver. The Zephyr kernel adds around 19KB which makes a total of less than 30KB. For comparison, Medusa [2] requires a total of 38KB, Espruino [23] and MicroPython [19] requires 100KB-200KB and SynchronVM [20] requires 32KB.

```
target C;

reactor Unbounded {
    logical action a;
    logical action b
    state i:int(0)

    reaction(a,b) -> a,b {=
        lf_schedule(a,0);
        lf_schedule(b,++self->i);
    =}
}
```

(a) LF program with unbounded event queue.

```
target C;

reactor Bounded {
    timer t1(0, 10 msec)
    timer t2(0, 50 msec)
    reaction(t1) {= =}
    reaction(t2) {= =}
    reaction(t1,t2) {= =}
}
```

(b) LF program with bounded event queue.

Figure 5 Memory requirements for LF programs.

A minimal reactor with just a single reaction triggered at startup only requires 96B. Adding reactions, timers and modes all add less than 100B of overhead. The user code in the reaction bodies, as well as the SDK components they depend on, will add to this.

As we will see next, the limiting factor for platforms like the NRF52 is not the size of the program memory but rather the size of RAM.

5.1.2 Data size

The bottom row of Table 1 shows the RAM footprint of LF. It consists of three parts. First, there is the constant size of the runtime, which only amounts to 265B plus 170B for the Counter implementation. Then there is the memory footprint for each LF component like reactions, actions and ports. This is known at compile-time but is currently being allocated dynamically in the initialization routine. Last, we have the events being communicated between the reactors over the ports through the event queue. These events are allocated as needed at run-time. When an event has been consumed, it is recycled to avoid memory fragmentation. Due to the expressiveness of the Reactor model of computation, there is, in general, no upper bound to the size of the event queue. There exist syntactically valid pathological LF programs for which the event queue can grow without bound. For instance, consider the reactor *Unbounded* in Fig. 5a. Each time the action a triggers, two events will be added to the event queue. The first triggers a at the same tag, while the second triggers b at a future tag. Only events triggering a will ever be handled and the events triggering b will accumulate. Such a program is said to exhibit a *stuttering Zeno* condition [13].

However, there also exist classes of "safe" LF programs for which upper bounds on the event queue can be computed. Consider for instance the reactor *Bounded* in Fig. 5b. It is a simple reactor with two periodic timers and three reactions triggered by them. The size of the event queue for this reactor will never exceed two. At any time during execution, there is one event per timer on the event queue. In fact, any LF program that uses only timers to drive logical time forward will have a statically computable bound on the size of its event queue.

5.2 Timing precision

One of the advantages of using LF for embedded applications is its timed semantics. An implementation of LF has good timing precision if the physical time at which a reaction is executed is close to the logical time at which it was scheduled to execute.

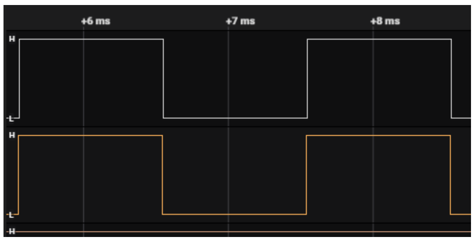


Figure 6 Logical analyzer dump.

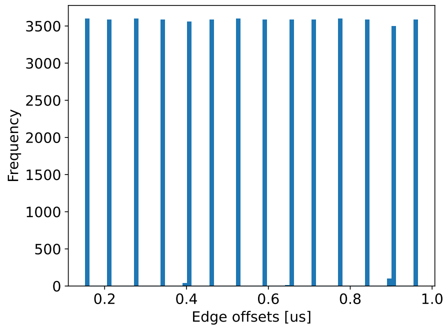


Figure 7 Actuator driving precision.

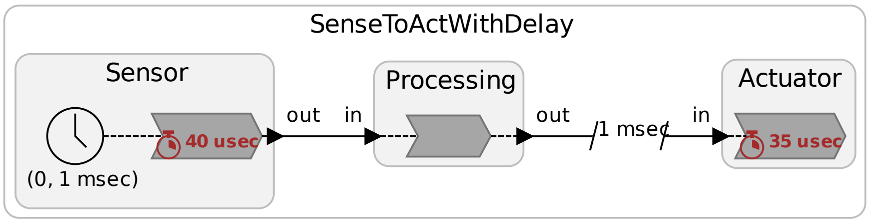


Figure 8 Tight control loop with after-delay.

5.2.1 Square-wave generator

To test the timing precision, the NRF52 was configured to generate a 1 KHz square wave on an output pin using hardware timers. We created a simple LF program to generate the same signal using a timer and a reaction that toggles a GPIO pin.

Fig. 6 shows the two resulting square waves as measured by a Salaea Logic Pro logic analyzer. Fig. 7 shows a histogram of the offset between the LF square wave and the hardware generated square wave. Clearly, the LF runtime can deliver precise and repeatable timing. All observed offsets were less than 1 μs. The multi-modality of the distribution is due to inaccuracies associated with waking up from sleep. This could be improved by targeting a lower-level timer API; however, such an approach would not be platform-agnostic. An important observation is that this timing precision can be affected by independent long-running reactions due to barrier synchronization at each time tag. We are working on new scheduling strategies and language primitives to improve timing precision.

5.2.2 Tight control loops with load

Consider again the SenseToAct example from Fig. 3 in Section 2.1.3. An interesting performance metric is the load a system can sustain while still meeting its deadlines. By introducing busy-waits in the reaction bodies, different workloads can be emulated. Our experiments show that the runtime overhead for SenseToAct is only 70 μs. This leaves 93% of the 1 ms period for actual computation.

An important observation is that the semantics of an LF deadline only puts an *upper bound* on the lag of a reaction invokation. In fact, for Fig. 3 there is a full 1 ms interval in which the *Actuator* reaction can legally be invoked. To introduce a *lower* bound on the invokation time we can use logical delays. In Fig. 8 we have introduced a 1 ms logical

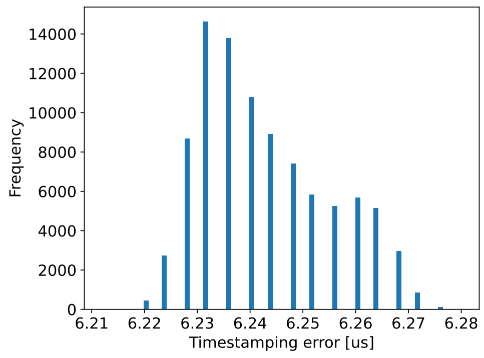


Figure 9 Physical action precision.

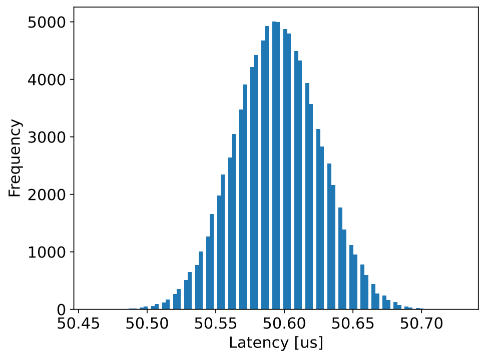


Figure 10 Physical action latency.

delay on the connection between *Processing* and *Actuator*. This means that any output produced by *Processing* will be visible at *Actuator* one *logical* ms later. This, together with the deadline, specifies an interval of 35 μs in which it is acceptable to invoke the *Actuator* reaction. This is a portable way of explicitly specifying bounds on jitter. Notice also that this will coincide with the next evaluation of *Sensor*, so we have, in effect, made a pipeline.

With this configuration, we can execute *Processing* for 885 μs, without missing any deadlines. This yields a utilization of 88.5%. Notice that both the utilization is lower and the deadlines are higher. The reason for this is that the logical delay on the connection is syntactic sugar for a reactor consisting of two reactions and a logical action, which introduces some additional overhead [15].

The execution of this program without any deadline misses means that the runtime claims that timing is not being violated. To verify this, we instrumented hardware timers to generate events on GPIO pins and compared with events generated from within the reaction bodies of *Sensor* and *Actuator*.

The results show offsets of around 28 μs and 35 μs for sensing and actuating, respectively. This is well below their deadlines. *Sensor* is lagging more than *Actuator*, this is expected as their reactions are logically simultaneous. On single-core platforms, simultaneous reactions are ordered based on earliest deadline first scheduling.

5.2.3 Asynchronous events

The ability to react to asynchronous events in a timely manner is crucial for many embedded systems. For instance, a sensor might signal that it has made a new sample by creating a transition on a pin. There are two important timing properties of a reaction to such an event. (1) The precision with which the system *timestamps* the occurrence of the event and (2) *latency* from occurrence to being handled by the system.

Fig. 9 shows the latency between the invokation of an interrupt handler that schedules a physical action and LF's timestamping of the resulting event. The low jitter indicates that a constant offset of around 6.22 μs could be subtracted from the timestamp associated with the physical action.

Fig. 10 shows the latency from asynchronously scheduling a physical action until a reaction triggered by the physical action is invoked. The latency is normally distributed with a mean of 50 μs.

The reported overheads are in addition to the interrupt latency of the underlying platform.

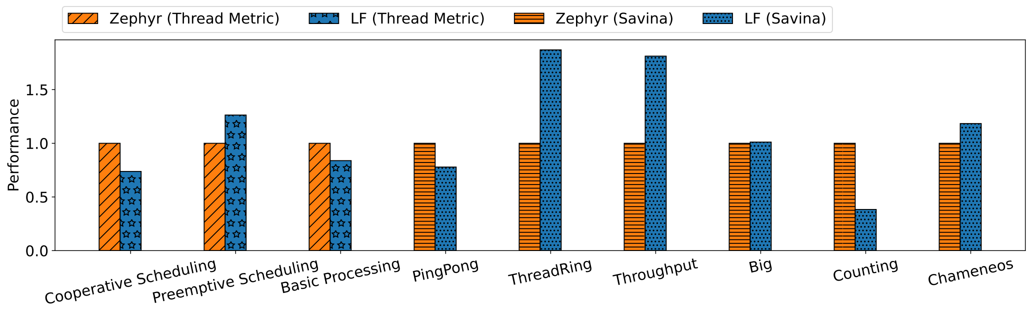


Figure 11 Results of Thread-Metric and Savina benchmark suites for Lingua Franca and a threaded implementation targeting Zephyr kernel API.

5.3 Performance

To assess the general performance of LF on microcontrollers, we run a relevant subset of the Thread-Metric and Savina benchmarks against a baseline implementation using Zephyr's thread APIs. At a fundamental level, LF's scheduling and context-switching overhead is compared to Zephyr's overhead of message-passing and context-switching. This is evaluated for multiple different program topologies and workloads. In the following, we will use the term *actor* to refer to a concurrent component. In the threaded baseline, it corresponds to a thread, and in the LF implementation, it corresponds to a reactor.

5.3.1 Thread-Metric

Thread-Metric [8] is a benchmark for measuring performance of RTOSes. It consists of seven benchmarks that measure overheads associated with different RTOS services like message passing, synchronization, scheduling, interrupt latency and memory allocation. LF does not expose low-level APIs for interrupts, memory allocation and message buffers to the user. In this evaluation, we have thus excluded Message Processing, Synchronization Processing, Interrupt Processing, and Memory Allocation.

The left part of Fig. 11 shows the results of the remaining Thread-Metric benchmarks normalized to the threaded baseline. Performance is measured as the number of iterations through the program during a fixed period. In Cooperative Scheduling, LF achieves 74% of the performance of Zephyr threads.

We measured the cost of a Zephyr thread yield to be 8 μs while LF's reaction-switching and logical time advancement take 4 μs and 29 μs respectively. This means that the LF implementation would need eight low-overhead reaction switches to make up for the relatively expensive time advancement. Cooperative Scheduling only consists of five actors and thus renders LF's performance worse than the baseline.

In the Preemptive Scheduling benchmark, LF outperforms threads by 26%. The reason is the long chain of preemptions occurring when the lowest priority thread executes first. In LF this is not a problem as there is no priority-based scheduling, rather, the topology of the reactor network imposes well-defined restrictions on scheduling. In LF, the preemption is translated into reaction-switching followed by a logical time advancement in each iteration. The threaded baseline must perform twice as many context-switches as reaction-switches in the LF version, due to the priority-based preemption.

Basic Processing only involves a single actor, in a while loop, performing a long-running computation before incrementing a counter. LF is outperformed as it performs logical time advancement after each iteration.

5.3.2 Savina

The Savina benchmark suite is an actor-oriented benchmark suite [7]. The Savina benchmarks focus on compute-intensive rather than IO-intensive applications and put emphasis on message-passing overhead. We evaluate LF on a subset of the Savina benchmarks called the micro benchmarks. The right portion of Fig. 11 shows the result normalized to the Zephyr baseline.

PingPong only consists of two actors sending a message back and forth. The Zephyr implementation uses a message queue [26], which incurs around 8 μs overhead for each communication. In LF the same communication only takes around 6 μs. However, LF advances logical time after each round of back and forth. This takes almost 30μs. In other words, LF performs better when there are long chains of reactions between each logical time advancement. Counting suffers from the same problem. While PingPong has three reactions per logical tag, Counting only has two. Moreover, the threaded implementation of Counting allows for two iterations per context-switch where PingPong requires a context-switch between each ping and pong. This is because we are using a message queue of size 1 which can buffer one element and thus enable two elements communicated per context-switch.

Throughput and ThreadRing are examples at the other extreme. In Throughput, a single producer sends a message to 60 consumers, each performing some floating-point arithmetic on the received data. In LF, this constitutes a chain of 60 reactions per logical time advancement. The Zephyr baseline performs a normal context-switch for each communication. ThreadRing is similar only there is a a chain of consumers rather than the one-to-many topology of Throughput. Again the low reaction-switching overhead of LF enables a 87% and 81% improvement over Zephyr, respectively.

Big is a many-to-many benchmark where each actor sends a ping to a random actor and waits for a response. Due to limitations in RAM we only use 10 actors. This constitutes a chain of reactions just long enough to make up for the expensive logical time advancement.

Chameneos measures how contention on a shared resource affects the performance. In this benchmark, a large number of creatures called chameneos send messages to a single mall, which pairs up two chameneos for a meeting. The Zephyr threading API and the LF runtime both allow the mall to process messages in batches without context-switching between each message. This lets both implementations attain good performance because the mall processes a large volume of messages in comparison to the other actors in the program.

6 Conclusion

We have introduced the embedded target of Lingua Franca, which brings deterministic concurrency and portable specification of timing behaviors to resource-constrained platforms like microcontrollers. The embedded target is built on top of the popular Zephyr RTOS. The runtime is evaluated in terms of memory footprint, timing precision and general performance against baselines using Zephyr’s shared memory concurrency primitives directly. We show that the LF runtime adds little overhead and in several cases outperforms the baseline.

References

1 Gul Agha and Carl Hewitt. *Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming*, pages 49–74. MIT Press, Cambridge, MA, USA, 1987.

2 Thomas W. Barr and Scott Rixner. Medusa: Managing concurrency and communication in embedded systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC’14, pages 439–450, USA, 2014. USENIX Association.

3 Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. Califor-nia, USA, 2005.

4 The Linux Foundation. Zephyr RTOS. `https://zephyrproject.org/`, 2022.

5 Philipp Haller and Martin Odersky. Event-Based Programming Without Inversion of Control. In David E. Lightfoot and Clemens Szyperski, editors, *Modular Programming Languages*, Lecture Notes in Computer Science, pages 4–22. Springer, 2006. `doi:10.1007/11860990_2`.

6 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. `doi:10.1145/359576.359585`.

7 Shams M. Imam and Vivek Sarkar. Savina – An actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, pages 67–80, 2014.

8 iSotEE. Thread Metric Test Suite. `https://github.com/iSotEE/thread_metric_test_suite`, 2020.

9 Erling Rennemo Jellum, Torleiv Håland Bryne, Tor Arne Johansen, and Milica Orlandíc. The syncline model–analyzing the impact of time synchronization in sensor fusion. *arXiv preprint*, 2022. `arXiv:2209.01136`.

10 Phil Koopman. A case study of Toyota unintended acceleration and software safety. *Presentation. Sept*, 2014.

11 Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462, 2010. `doi:10.1109/SP.2010.34`.

12 Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

13 Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach.* MIT Press, Cambridge, MA, USA, second edition, 2017. URL: `http://LeeSeshia.org`.

14 Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

15 Hendrik Marten Frank Lohstroh. *Reactors: A deterministic model of concurrent computation for reactive systems.* University of California, Berkeley, 2020.

16 Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. Toward a lingua franca for deterministic concurrent systems. *ACM Trans. Embed. Comput. Syst.*, 20(4), May 2021. `doi:10.1145/3448128`.

17 David May. The XMOS architecture and XS1 chips. *IEEE Micro*, 32(6):28–37, 2012.

18 Tommi Mikkonen and Antero Taivalsaari. Web applications – spaghetti code for the 21st century. In *2008 Sixth international conference on software engineering research, management and applications*, pages 319–328. IEEE, 2008.

19 George Robotics. MicroPython – Python for microcontrollers. `micropython.org`, 2022.

20 Abhiroop Sarkar, Bo Joel Svensson, and Mary Sheeran. Synchron – an API and runtime for embedded systems, 2022. `doi:10.48550/arXiv.2205.03262`.

21 Nordic Semiconductor. nRF52 DK. `https://www.nordicsemi.com/Products/Development-hardware/nRF52-DK`, 2022.

22 Douglas Watt. *Programming XC on XMOS devices.* XMOS Limited, 2009.

23 Gordon F. Williams. *Making Things Smart: Easy Embedded JavaScript Programming for Making Everyday Objects into Intelligent Machines.* Maker Media, Inc., 2017.

24 W. Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2):28–34, February 1973. `doi:10.1145/953353.953355`.

25 Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 163–176, USA, 2010. USENIX Association.

26 Zephyr. Message Queues. `https://docs.zephyrproject.org/3.2.0/kernel/services/data_passing/message_queues.html`, 2022.

Efficient Abstraction of Clock Synchronization at the Operating System Level

Alessandro Sorrentino
DEIB, Polytechnic University of Milano, Italy

Federico Terraneo
DEIB, Polytechnic University of Milano, Italy

Alberto Leva
DEIB, Polytechnic University of Milano, Italy

Abstract

Distributed embedded systems are emerging and gaining importance in various domains, including industrial control applications where time determinism – hence network clock synchronization – is fundamental. In modern applications, moreover, this core functionality is required by many different software components, from OS kernel and radio stack up to applications. An abstraction layer devoted to handling time needs therefore introducing, and to encapsulate time corrections at the lowest possible level, the said layer should take the form of a timer device driver offering a *Virtual Clock* to the entire system. In this paper we show that doing so introduces a nonlinearity in the dynamics of the clock, and we design a controller based on feedback linearization to handle the issue. To put the idea to work, we extend the Miosix RTOS with a generic interface allowing to implement virtual clocks, including the newly designed controller that we call FLOPSYNC-3 after its ancestor. Also, we introduce the resulting virtual clock in the TDMH [20] real-time wireless mesh protocol.

2012 ACM Subject Classification Computer systems organization → Real-time operating systems; Networks → Time synchronization protocols; Computer systems organization → Embedded software

Keywords and phrases Clock synchronization, Real-time operating systems, Embedded software, Real-time control

Digital Object Identifier 10.4230/OASIcs.NG-RES.2023.4

1 Introduction

The world of embedded systems is evolving from isolated to distributed systems. This move can be observed in several research and market trends such as the Industrial Internet of Things (IIoT) [16]. As a result, clock synchronization is becoming a key technology to enable both real-time industrial applications as well as low energy wireless protocols [23]. At the application level, synchronization in distributed embedded systems allows the execution of coordinated tasks among multiple devices [13], allows to perform sensing and reconstruction of spatially distributed phenomena [10, 2], while the availability of synchronization at the network level enables the use of TDMA protocols [1, 20, 7], being thus fundamental for real-time communication among devices. Since in modern embedded operating systems both applications and OS components – such as the radio stack – can benefit from clock synchronization, an abstraction layer that handles time correction directly at the OS level is therefore needed. Moreover, from a software engineering perspective, the presence in the OS codebase of both corrected and uncorrected time values is a potential source of errors. Therefore, it becomes desirable to encapsulate time correction at the lowest possible level, such as the timer device driver.

However, using corrected times in the entire OS codebase introduces an issue: the uncorrected time is usually required by the clock synchronization algorithm itself. Efficient clock synchronization schemes such as FLOPSYNC-2[22] require uncorrected timestamps of received synchronization packets, and performing clock synchronization using corrected timestamps is challenging as it makes the model of the clock synchronization problem nonlinear.

This work introduces a new clock synchronization scheme, FLOPSYNC-3, that is capable of operating with timestamps corrected by the previous iteration of the algorithm itself. As a result of this improved capability, the Miosix RTOS was extended with a generic interface allowing to implement clock correction at the hardware timer level.

The FLOPSYNC-3 controller is here tested both in simulation and on a network of nodes running the Miosix operating system and the TDMH [20] real-time wireless mesh protocol.

This paper is organized as follows: Section 2 presents a brief overview on the state of the art in clock synchronization for distributed embedded systems. Section 3 discusses how the Miosix OS has been extended with a virtual clock abstraction that enables transparent clock corrections. Section 4 briefly mentions the design of the Miosix subsystem for performing timestamp measurements, a key feature used to precisely timestamp clock synchronization packets. Section 5 presents the FLOPSYNC-3 clock synchronization scheme that can perform clock corrections using the virtual clock as actuator while operating on corrected timestamps only. Finally, Section 6 presents simulation and experimental results, and Section 7 outlines future research directions.

2 Related Works

Clock synchronization is a classical problem in distributed systems [11, 15], but also one where research is still ongoing to produce clock synchronization schemes fine-tuned to changing application requirements and hardware capabilities. Many works related to clock synchronization in distributed embedded systems come from the Wireless Sensor Network research community, focusing on several aspects including low power synchronization [18, 22], propagation delay compensation [12, 19], efficient synchronization information dissemination [14, 8].

When considering accuracy, a major differentiating factor is whether a clock synchronization scheme only performs offset corrections or it also performs skew corrections. Simple schemes such as TPSN [9] and DMTS [17] only correct for offset. When implemented at the OS level, this correction can be efficiently performed by overwriting the hardware counter with the required correction at every synchronization [9]. The disadvantage is however that after each correction the hardware clock keeps counting at the incorrect frequency, and thus a time error accumulates over the synchronization period, which reaches its maximum value immediately before the next correction. Another issue is that the value returned by the clock exhibits a discontinuity at every synchronization [22], a matter that can introduce errors in interval measurements, especially for short intervals.

More advanced clock synchronization schemes perform skew (also known as frequency or rate) corrections. The synchronization scheme produces both an offset and a frequency correction value at every synchronization. As altering the frequency of a crystal oscillator requires additional hardware [5] which is usually unavailable in off-the-shelf boards, the frequency correction is preferably performed by applying an algorithm every time the OS or applications request the time. In this paper we refer to such algorithm as a virtual clock. For a given synchronization period, frequency correction allows for lower synchronization errors compared to offset correction. Additionally, clock synchronization schemes that perform frequency correction can make the corrected clock continuous and monotonic [22], thus avoiding clock jumps.

Real-time embedded systems also face increasing power and energy constraints [3], especially if battery operated. Clock synchronization may thus be required to operate also when the processor enters a deep sleep state which includes turning off the main oscillator. In such cases, time is kept using a low power Real-Time Clock (RTC), and this introduces the need to synchronize both the RTC and high-frequency timebase [21], a matter that we account for by designing our virtual clock to support multiple corrections.

In this paper we address the clock synchronization problem from the perspective of implementing it at the real-time OS level. Software engineering considerations suggest us to completely encapsulate time correction, and since this makes uncorrected time unavailable to the clock synchronization scheme, we design a new scheme that can operate with corrected timestamps.

3 Virtual Clock

A real-time OS typically requires two main time-related primitives: one to get the current time, whose use is obvious, and one to set an interrupt in a given future time instant, to be used to handle context switches as well as sleeping tasks wakeup. This chapter describes the design and implementation of a virtual clock to make these primitives synchronization-aware.

3.1 Design

An uncorected clock t_{nc} fed by an oscillator with nominal frequency f_0, affected by (possibly time-varying) frequency error δ_s will progressively diverge from an ideal one as

$$t_{nc}(t) = \int_0^t \frac{f_s(\tau)}{f_0} d\tau = t + \int_0^t \frac{\delta_s(\tau)}{f_0} d\tau \tag{1}$$

where f_s is the instantaneous oscillator frequency, and the integral accounts for the accumulated frequency error. Accordingly, the accumulated frequency error $\Delta(k)$ over one clock synchronization period k of duration T is

$$\Delta(k) = \int_{(k-1)T}^{kT} \frac{\delta_s \tau}{f_0} d\tau \tag{2}$$

A virtual clock VC is a piece wise linear function (Figure 1) that applied to the uncorrected clock t_{nc} produces a corrected one. Virtual clocks allow to perform not only offset corrections, but also frequency corrections. Said otherwise, it is possible to control a virtual clock to count time faster or slower than the underlying hardware clock to better approximate a reference clock. A virtual clock is however just an actuator, it provides the *means* to correct a hardware clock, but requires at every synchronization period updated parameters. A clock synchronization scheme uses a controller and time information from an external reference to adjust the virtual clock rate trying to align it to the reference clock. By defining the virtual clock rate separately on each synchronization interval, it can be demonstrated by induction that the value of the virtual clock (that is the corrected time t_c) on a generic time t_{nc} inside a synchronization interval $[\,kT, (k+1)\,T\,]$ can be expressed as

$$t_c = VC(t_{nc}) = VC(k) + \dot{VC}(k)\big(t_{nc} - t_{nc}(k)\big) \tag{3}$$

where $\dot{VC}(k)$ is the rate of the virtual clock. More specifically, if $t_{nc} = t_{nc}(k+1)$, its definition simplifies as

$$VC(k+1) = VC(k) + \dot{VC}(k)\,(T + \Delta\,(k)) \tag{4}$$

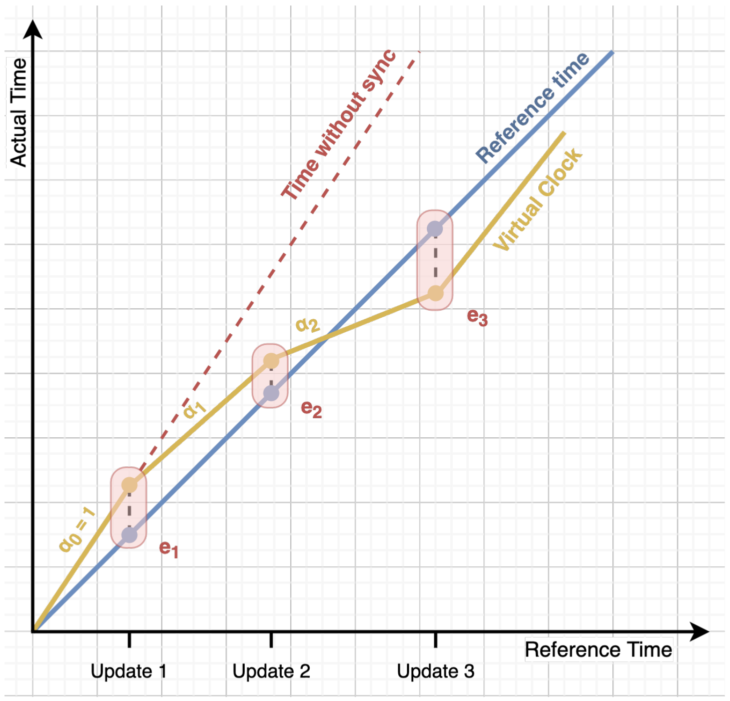


Figure 1 Virtual clock correcting clock rate to align itself to a reference clock.

We can further generalize the virtual clock expressing (3) as $f = a_k x + b_k$ by algebraic manipulation

$$\begin{cases} a_k & = \dot{VC}(k) \\ b_k & = VC(k) - \dot{VC}(k)t_{nc}(k) \end{cases} \tag{5}$$

where a_k is the rate correction of the clock in the synchronization period k and b_k is the offset. This rate is adjusted by the controller to align the current clock to the reference, and is related to the mean skew over the synchronization period.

3.2 Implementation

The virtual clock was implemented in C++ as part of the Miosix RTOS, as shown in Figure 2. To support clock synchronization as well as deep sleep operation which entails transitions from a board RTC to an high resolution clock (a technique called VHT [21]), a virtual clock may need to perform multiple clock corrections f_i combined. The software design of the virtual clock thus supports multiple corrections as a *Variable Length Correction Stack* (VLCS). This design allows for an arbitrary number of *Correction Tiles*, each with their own correction parameters $a_{k,i}$ and $b_{k,i}$. For performance reasons, the number of correction tiles is configured at compile time as a template parameter. Having n distinct corrections chained together as $f_0 \circ \cdots \circ f_n$, the combined correction parameters can be calculated as

$$a_{k,vc} = \prod_{i=0}^{n-1} a_{k,i} \tag{6}$$

$$b_{k,vc} = \sum_{i=0}^{n-2} \left\{ b_{k,i} \cdot \prod_{j=i+1}^{n-1} a_{k,j} \right\} + b_{k,n-1} \tag{7}$$

where index 0 is the correction closer to the hardware timer, and n the furthest.

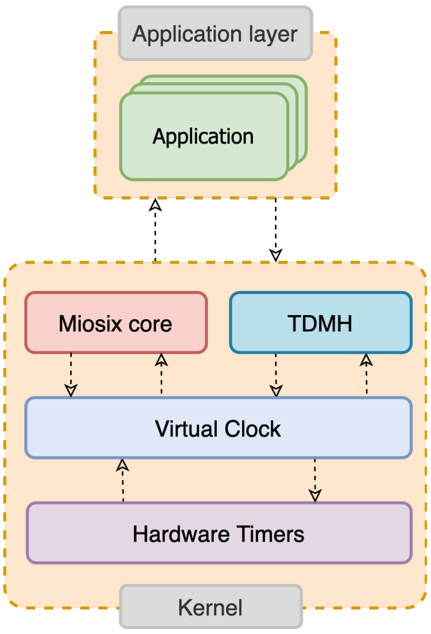


Figure 2 Virtual clock interface.

As reading the current time is a more frequent operation than changing the correction coefficients, the combined parameters are precomputed when a new clock correction is produced (Figure 3). Conversely, to set a time interrupt the corrected time coming from the OS will need to be back-converted as the hardware timer still works using uncorrected time.

$$t_c = a_{k,vc} \cdot t_{nc} + b_{k,vc} \tag{8}$$

$$t_{nc} = (t_c - b_{k,vc}) \,/\, a_{k,vc} \tag{9}$$

3.3 Optimization

As the typical skew of quartz clocks is in the order of tens of parts per million (*ppm*), the $a_{k,vc}$ coefficient should be very close to 1. Since many microcontrollers lack a Floating Point Unit (FPU), we need an efficient way (exploiting the range of $a_{k,vc}$ as just identified) to

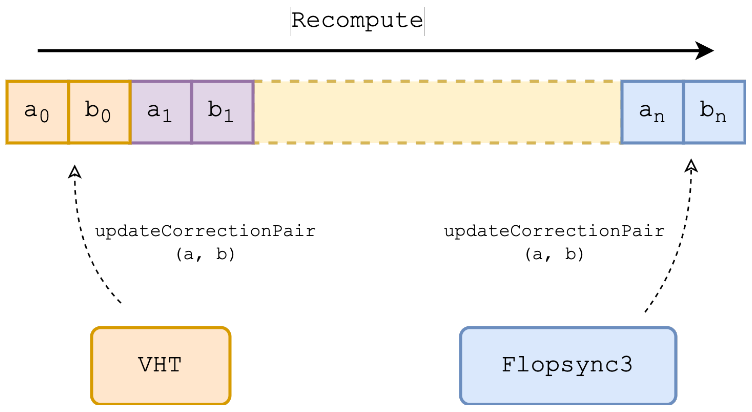


Figure 3 Virtual clock recomputing aggregated parameters a_{vc} and b_{vc}.

perform the multiplication $a_{k,vc} \cdot x$, as the time retrieval is one of the most critical path of the operating system. For this purpose, a template class *Fixed* was designed. This is capable of representing a fixed point number with an arbitrary number of bits for the decimal part. Given a few compile-time optimized functions able to handle multiplication between a 64-bit integer and a fixed point 32.32, a specialization of the said class – called *fp32_32* and representing a fixed point number as a 64-bit integer with 32-bit for both decimal and integer part – was used. With fp32_32, the multiplication $a_{k,vc} \cdot x$ can be performed in just 60 clock cycles bringing the total time to get the current time to 170 clock cycles, a 37% improvement compared to the previous implementation. Regarding the uncorrection, we can note from (9) that a division by $a_{k,vc}$ is needed. There is no nice properties to perform fast division using fixed point, so it was implemented as a multiplication for the inverse. The inverse value is precomputed using 64-bit floating point numbers at every update of the $a_{k,vc}$ parameter and converted to fp32_32. The pre-computation is optimized using a modified version of the fast inverse square root algorithm [6], adapted to perform $1/x$ instead of $1/\sqrt{x}$ as follows. The optimization relies on the fact that an *IEEE754* double precision number is very similar to an Logarithmic Number System (LNS) number, as they never differ for more than a small factor. An interesting property of LNS numbers is that it makes implementations for multiplications and divisions very efficient. In particular, the inverse of an LNS number v is $-v$. The bit representation of a floating point number u can approximated as the LNS number $x = 2^{u/2^{52}-1023}$, and using this representation the inverse q can be computed as follows

$$2^{q/2^{52}-1023} = 2^{-(u/2^{52}-1023)} \tag{10}$$

which solving the implicit equation results in

$$q = \mathrm{0x7FE0000000000000} - u \tag{11}$$

Performing a sweep with sufficient precision, it was possible to elaborate a quadratic regression model to approximate faster and with more precision the hexadecimal value. Having a closer approximation, less Newton steps are necessary making the inversion faster. This whole inversion process is called *optimizedFastInverse*.

4 Hardware Events

Although what presented above is sufficent to support the time-related requirements of an OS, performing clock synchronization requires accurate timestamping of received radio packets. Moreover, advanced radio transmission techniques such as constructive interference require accurate packet transmission times [22]. To abstract hardware-accelerated event timestamping and generation, Miosix was extended with an *Eventstamping* interface. This abstraction introduces the concept of *event channels* that abstract the event sources or sinks of a given platform. Every event channel can be configured as *input*, for external event timestamping, or as *output* to trigger events. When configured in input mode, a thread can block and wait for an event to happen on the chosen channel, with an optional timeout (Figure 4). When configured as output, a thread can generate a hardware event in the future, blocking until that time point. This design simplifies the realization of TDMA networking protocols. Since events are measured/generated in hardware, the achievable time granularty is that of the hardware timer (in our implementation 21ns), and is unaffected by software interrupt latencies.

5 FLOPSYNC-3

The redesign of the Miosix OS timing subsystem in order to only operate in terms of corrected time in the entire OS –except for the timer driver– required the design of a new clock synchronization scheme. Previously, the clock synchronization packets were timestamped using the uncorrected clock, as this was needed by the FLOPSYNC-2 algorithm [22]. The previous approach required to deal with both corrected and uncorrected times and was causing code maintainability issues from a software engineering standpoint. However, performing clock synchronization using timestamps corrected by the previous round of clock synchronization makes the problem nonlinear. The FLOPSYNC-3 controller was designed to address the aforementioned nonlinearity, and implemented at the OS level.

5.1 Design

Given (4), we can define the clock synchronization error at the end of each synchronization period as

$$e(k) = VC(k) - kT \tag{12}$$

To observe the evolution of the error across synchroniation periods, we can compute the next error as a function of the previous, resulting in

$$e(k+1) = e(k) + T\left(1 - \dot{VC}(k)\right) - \Delta(k)\dot{VC}(k) \tag{13}$$

where the $\Delta(k)\dot{VC}(k)$ term makes the model nonlinear. To perform the control synthesis we used *Feedback Linearization* [4] to linearize this process using the output $u(k)$ of a linear controller and express the new error as

$$e(k+1) = \beta\, e(k) + (1-\beta)\, u(k) \tag{14}$$

and as a consequence have the new output $u(k)$ of the controller provide $\dot{VC}(k)$ from

$$\dot{VC}(k) = \frac{e(k)(1-\beta) + u(k)(\beta-1) + T}{T + \Delta(k)}. \tag{15}$$

The mean skew value at the synchronization period k is of course not available and needs to be approximated with the previous one $(k-1)$, i.e.,

$$\hat{\Delta}(k) = \Delta(k-1) = \frac{VC(k) - VC(k-1)}{\alpha(k-1)} - T \tag{16}$$

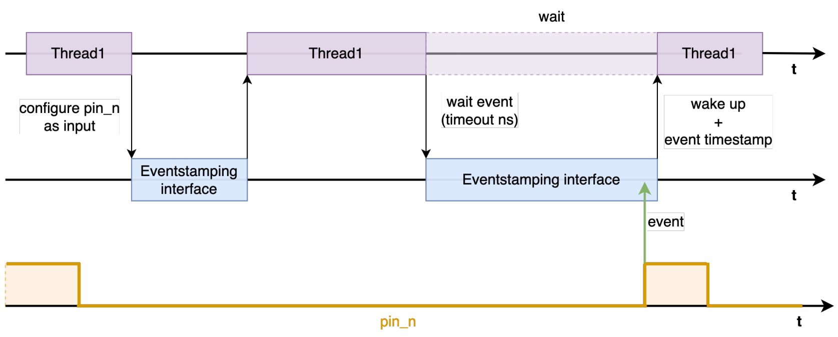


Figure 4 Eventstamping, wait event.

We can then obtain the transfer function $\mathscr{H}(z)$ of the imposed dynamic (14) as

$$\mathscr{H}(z) = \frac{E(z)}{U(z)} \Rightarrow \frac{1-\beta}{z-\beta} \tag{17}$$

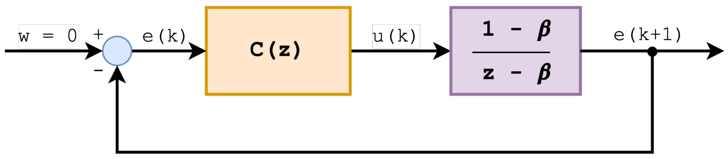


Figure 5 FLOPSYNC-3 Control scheme.

The controller $C(z)$ used to work in conjuction with the feedback linearization is a proportional one having a gain of 0.15. The full FLOPSYNC-3 control scheme is shown in Figure 5.

6 Simulation and Experimental Results

The operation of the FLOPSYNC-3 controller and virtual clock were first assessed through simulations performed using the *Modelica* language.

Figure 6 shows one such simulation, where the clock synchronization period T was set to 10 seconds and β was chosen to be 0.025. The left part of the figure shows the simulated clock skew profile, that starts from 10 ppm and increases to 50 ppm from $T = 150$ seconds, approximating in the simulation the effect of an ambient temperature change. The right part of the figure shows the clock synchronization error. The blue line is the instantaneous error of the virtual clock, thus the time error exposed to the operating system and application. As a node in the network can measure its error only at discrete intervals, corresponding to when synchronization packets are received, the red line shows the measured error that feeds the FLOPSYNC-3 controller.

As can be seen, the initial 10 ppm skew causes a 100 μs error that is quickly corrected by the FLOPSYNC-3 controller. The frequency change caused by the simulated temperature change, although higher in amplitude than the initial skew causes a lower peak error, less than 75 μs, due to its slower nature.

The FLOPSYNC-3 controller as specified by equation (15) and (16) has been implemented in Miosix acting on the the variable length correction stack of the virtual clock. Since deep sleep support was not implemented, the correction stack was configured to perform the FLOPSYNC-3 correction only. Synchronization parameters T and β were configured as in the simulations. The clock synchronization error measurement was taken from the TDMH networking stack using the eventstamping interface to provide the timestamps of received synchronization packets. FLOPSYNC-3 was implemented using the fp32_32 type to perform intermediate calculations efficiently. Because of the limited range of this type, pre-scaling was necessary to avoid overflows.

Clock synchronization experiments were performed with a network of nodes running the TDMH networking stack on top of Miosix. Figure 7 shows the clock synchronization error of one such node. The top part of the figure shows the measured clock synchronization error in the first three minutes after synchronization. The initial clock synchronization error of 40 μs occurs when the node is booted and joins the network. This value is the accumulation of the oscillator frequency error over the entire first synchronization period, as the FLOPSYNC-3 algorithm, being a feedback one, requires a first error measure to compute a correction. The

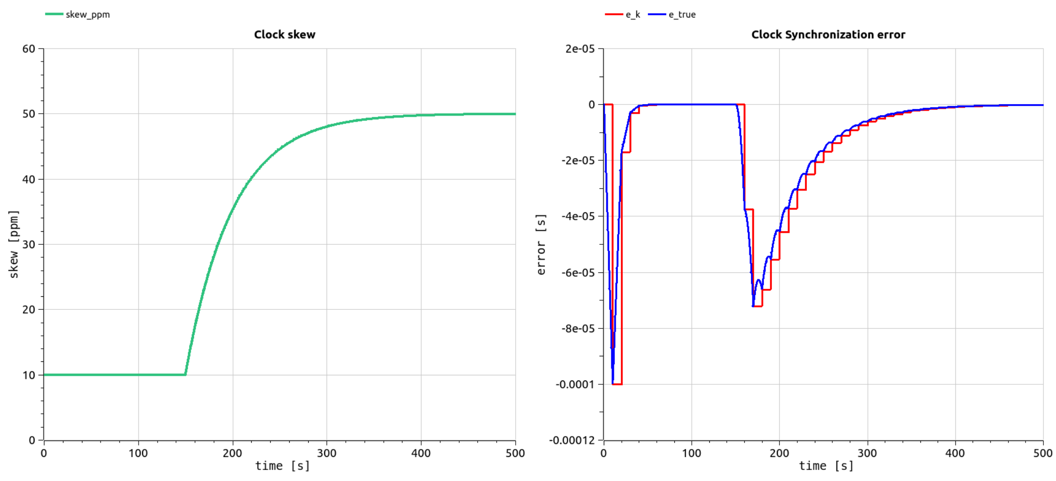


Figure 6 Simulated clock skew profile (left) and clock synchronization error (right). The blue line shows the instantaneous synchronization error, while the red line shows the measured error.

bottom part of the figure shows the error after the initial synchronization, over a period of approximately 24 hours, to better appreciate the error dynamics after the initial skew is corrected. The observed stochastic nature of the clock synchronization error, not present in the simulations, is caused by the measurement noise of packet timestamps. The error standard deviation, excluding the first transient, is 137 ns.

7 Conclusions

This work addressed abstracting clock synchronization at the operating system level. To achieve this goal a virtual clock was introduced as an efficient abstraction allowing a hardware timer driver to provide a time reference whose rate can be changed compared to the one of the underlying oscillator. Support for multiple corrections sources was accounted for, allowing the implementation of deep sleep solution such as VHT [21]. Encapsulating time correction allows reducing bugs and problems during development since all components are just using the same time source (corrected), but makes the uncorrected synchronization packet timestamps unavailable to the clock synchronization algorithm. The FLOSPYNC-3 controller was thus designed specifically to overcome this issue.

The Miosix real-time OS was extended with a flexible, efficient and modular timing subsystem based on the virtual clock design, capable of internalizing the clock correction and only exposing corrected time to all kernel and application tasks. This new timing subsystem was designed from the start to be general allowing to easily port the Miosix to different microcontrollers.

Future research directions will focus on further improving clock synchronization resilience to temperature variations, while future improvements of the Miosix timing subsystem will address completing the support for maintaining clock synchronization during deep sleep periods using the variable length correction stack.

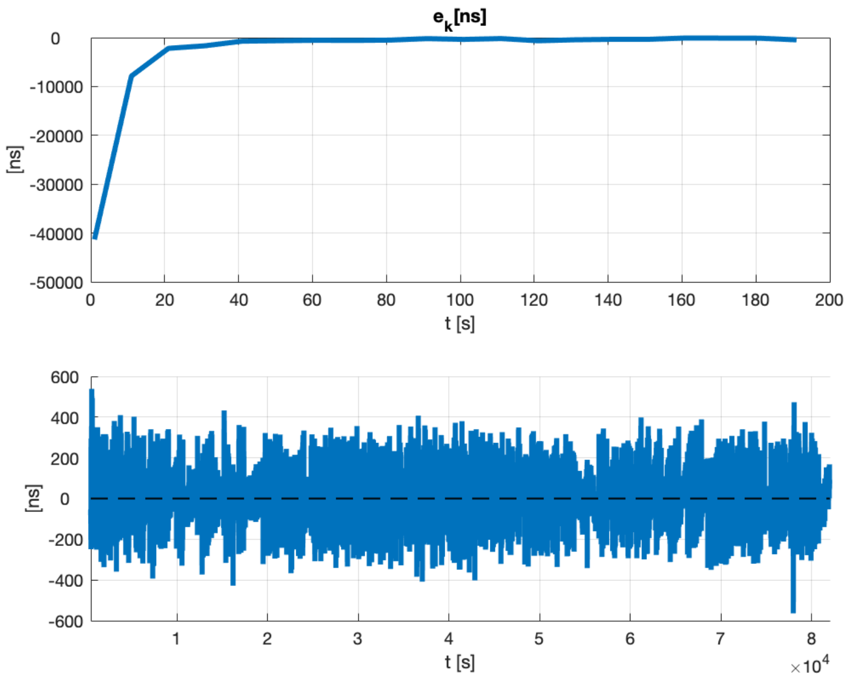


Figure 7 Clock synchronization error during experimental evaluation. Top plot includes the initial clock skew compensation, bottom plot shows the synchronization error after the initial transient.

References

1 Diogo Almeida, Miguel Gaitán, Pedro d'Orey, Pedro Santos, Luis Ramos Pinto, and Luís Almeida. Demonstrating RA-TDMAs+ for robust communication in WiFi mesh networks. In *RTSS@work workshop co-located with the 42nd IEEE Real-Time Systems Symposium*, 2021.

2 Riad Azzam and Nabil Aouf. Visual information to enhance time difference of arrival based acoustic localization. In *2014 IEEE International Conference on Aerospace Electronics and Remote Sensing Technology*, pages 77–82, 2014. `doi:10.1109/ICARES.2014.7024400`.

3 Ashikahmed Bhuiyan, Federico Reghenzani, William Fornaciari, and Zhishan Guo. Optimizing energy in non-preemptive mixed-criticality scheduling by exploiting probabilistic information. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3906–3917, 2020. `doi:10.1109/TCAD.2020.3012231`.

4 Roger W Brockett. Feedback invariants for nonlinear systems. *IFAC Proceedings Volumes*, 11(1):1115–1120, 1978.

5 Maxim Buevich, Niranjini Rajagopal, and Anthony Rowe. Hardware assisted clock synchronization for real-time sensor networks. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 268–277, 2013. `doi:10.1109/RTSS.2013.34`.

6 John Carmack. Fast inverse square root. URL: `https://blog.timhutt.co.uk/fast-inverse-square-root/`.

7 Julius Degesys, Ian Rose, Ankit Patel, and Radhika Nagpal. DESYNC: Self-Organizing Desynchronization and TDMA on Wireless Sensor Networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, pages 11–20, 2007. `doi:10.1145/1236360.1236363`.

8 F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. Efficient network flooding and time synchronization with glossy. In *Information Processing in Sensor Networks (IPSN)*, 2011.

9 S. Ganeriwal, R. Kumar, and M. Srivastava. Timing-sync protocol for sensor networks. In *International Conference on Embedded Networked Sensor Systems*, 2003.

10 Grzegorz Krukar, Marco Wenzel, Piotr Karbownik, Norbert Franke, and Thomas von der Grün. Proof-of-concept real time localization system based on the UWB and the WSN technologies. In *2014 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 756–757, 2014. `doi:10.1109/IPIN.2014.7275559`.

11 L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978.

12 Roman Lim, Balz Maag, and Lothar Thiele. Time-of-flight aware time synchronization for wireless embedded systems. In *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, EWSN '16, pages 149–158, USA, 2016. Junction Publishing.

13 L. Maillet and C. Fraboul. Scheduling complex real-time tasks in an embedded distributed system. In *Proceedings Seventh Euromicro Workshop on Real-Time Systems*, pages 62–65, 1995. `doi:10.1109/EMWRTS.1995.514293`.

14 M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *Conference On Embedded Networked Sensor Systems*, 2004.

15 D.L. Mills. Internet time synchronization: the network time protocol. *IEEE Trans. on Communications*, 39(10):1482–1493, 1991.

16 Dinh C. Nguyen, Ming Ding, Pubudu N. Pathirana, Aruna Seneviratne, Jun Li, Dusit Niyato, Octavia Dobre, and H. Vincent Poor. 6G Internet of Things: A Comprehensive Survey. *IEEE Internet of Things Journal*, 9(1):359–383, 2022. `doi:10.1109/JIOT.2021.3103320`.

17 Su Ping. Delay measurement time synchronization for wireless sensor networks. In *Intel Research*, 2003.

18 A. Rowe, V. Gupta, and R. Rajkumar. Low-power clock synchronization using electromagnetic energy radiating from ac power lines. In *Sensys*, pages 211–224, 2009.

19 Federico Terraneo, Alberto Leva, Silvano Seva, Martina Maggio, and Alessandro Vittorio Papadopoulos. Reverse Flooding: Exploiting Radio Interference for Efficient Propagation Delay Compensation in WSN Clock Synchronization. In *2015 IEEE Real-Time Systems Symposium*, pages 175–184, 2015. `doi:10.1109/RTSS.2015.24`.

20 Federico Terraneo, Paolo Polidori, Alberto Leva, and William Fornaciari. TDMH-MAC: Real-Time and Multi-hop in the Same Wireless MAC. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 277–287, 2018. `doi:10.1109/RTSS.2018.00044`.

21 Federico Terraneo, Fabiano Riccardi, and Alberto Leva. Jitter-Compensated VHT and Its Application to WSN Clock Synchronization. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 277–286, 2017. `doi:10.1109/RTSS.2017.00033`.

22 Federico Terraneo, Luigi Rinaldi, Martina Maggio, Alessandro Vittorio Papadopoulos, and Alberto Leva. FLOPSYNC-2: Efficient Monotonic Clock Synchronisation. In *2014 IEEE Real-Time Systems Symposium*, pages 11–20, 2014. `doi:10.1109/RTSS.2014.14`.

23 Hüseyin Yiğitler, Behnam Badihi, and Riku Jäntti. Overview of time synchronization for iot deployments: Clock discipline algorithms and protocols. *Sensors (Switzerland)*, 20(20):1–58, 2020. `doi:10.3390/s20205928`.

Response Time Analysis for RT-MQTT Protocol Grounded on SDN

Ehsan Shahri[1]
Department of Electronics, Telecommunications and Informatics (DETI), University of Aveiro, Portugal
Institute of Telecommunications, Campus de Santiago, Aveiro, Portugal

Paulo Pedreiras
Department of Electronics, Telecommunications and Informatics (DETI), University of Aveiro, Portugal
Institute of Telecommunications, Campus de Santiago, Aveiro, Portugal

Luis Almeida
Research Center in Real-Time and Embedded Computing Systems (CISTER), Porto, Portugal
Faculty of Engineering, University of Porto (FEUP), Portugal

Abstract

The current industry trend is to replace the use of custom components with standards-based Commercially available Off-The-Shelf (COTS) based hardware and protocols. Furthermore, the emergence of new industrial paradigms, such as Industry 4.0 and the Industrial Internet of Things, sets additional requirements regarding e.g. scale, transparency, agility, flexibility and efficiency. Therefore, in these domains, application layer protocols such as Message Queuing Telemetry Transport protocol (MQTT) are gaining popularity, in result of their simplicity, scalability, low resource-usage and decoupling between end nodes. However, such protocols were not designed for real-time applications, missing key features such as determinism and latency bounds. A recent work proposed extending MQTT with real-time services, taking advantage of Software Defined Networking (SDN) to manage the network resource. These extensions allow applications to specify real-time requirements that are then captured by a resource manager and used to reserve the necessary resources at the network layer. This paper shows that such MQTT extended architecture is analyzable from a worst-case timing perspective. We derive a system model that captures the real-time features and we present a response-time analysis to assess the schedulability of the real-time traffic. Finally, we validate the analysis with a set of experimental results.

2012 ACM Subject Classification Networks

Keywords and phrases Real-time systems, OpenFlow, fixed-priority non-preemptive scheduling, response time analysis, MQTT

Digital Object Identifier 10.4230/OASIcs.NG-RES.2023.5

Funding This work is funded by Portuguese national funds through FCT/MCTES and, when applicable, co-funded by European Community funds, under projects IT-UIDB/50008/2020-UIDP/50008/2020 and CISTER-UIDB/04234/2020, as well as the FCT scholarship PD/BD/137388/2018.

1 Introduction

Real-time computing systems have been widely employed in industrial domains for many years, from robotics [9] to industrial control [26, 24] and industrial automation [14]. Recent trends, pushed by Industry 4.0 and the Industrial Internet-of-Things (IIoT), have brought Information and Communication Technologies (ICT) into industrial operations towards

[1] Corresponding author

increased scalability, transparency, agility, flexibility and efficiency. However, while improving these properties, ICT have typically disregarded their own timing behavior, conflicting with the real-time requirements found in industrial operations, namely quick response times and high predictability and stability. Another difficulty that came with ICT is their typical high overhead that may conflict with deployments within embedded resource-constrained hardware, commonly found in industrial settings. Moreover, Industry 4.0 and IIoT imply an increase in the complexity and heterogeneity of the data exchanges, both in terms of nature and requirements, putting together low and high data-rate flows, low and high bandwidth flows, and critical and non-critical flows.

The Message Queuing Telemetry Transport protocol (MQTT) [20] is one of the most popular application-layer protocols within the scope of the Internet-of-Things. It was developed already with a perspective of applying ICT in operations and it is finding a growing use within the IIoT, too. Main factors that contribute to its popularity include its simplicity, low footprint, scalability and effective publisher-subscriber messaging model. Standard MQTT relies on TCP/IP networks that provide ordered and lossless bi-directional channels. However MQTT misses support for real-time requirements, having Quality-of-Service (QoS) policies that only address message delivery. This limitation has been tackled by the scientific community [21, 16, 7, 18]. Specifically the work in [19] proposes an architecture called RT-MQTT that extends MQTT with real-time services, allowing applications to define real-time requirements that are translated to network reservations which are enforced using Software Defined Networking (SDN), particularly using the OpenFlow protocol.

Despite the referred efforts to reconcile real-time requirements with MQTT, formal traffic schedulability analysis is still lacking. In this paper we build on RT-MQTT and detail and formalize its system model and provide a response-time analysis for critical real-time traffic that assumes fixed-priority non-preemptive packet transmissions. The analysis and its implementation in a practical setup are validated with a set of experimental results that also allow assessing the tightness of the response-time analysis. Our aim is to show the analyzability of RT-MQTT. To the best of our knowledge, this is the first schedulability analysis for MQTT-based networks.

The rest of the paper is structured as follows. Section 2 discusses traffic schedulability analysis for multi-hop networks. Section 3 describes the OpenFlow switches as the networking elements in RT-MQTT. Section 4 introduces the multi-hop network architecture and system model. The response-time analysis is presented in Section 5 and validated with experiments in Section 6. Section 7 concludes the paper and discusses future work.

2 Related Work

Using ICT in the industrial operations domain came with several challenges, such as timing analysis. Particularly, analyzing the worst-case timing behavior of ICT protocols is frequently unfeasible. For this reason, we did not find a concrete schedulability analysis for real-time traffic on MQTT, which is not a surprise given the lack of protocol support. However, as we show in this paper, the extensions of RT-MQTT leverage the real-time network capabilities of SDN and make the framework analyzable.

Worst-case response-time analysis methods for real-time multi-hop networks fall in three main categories [28, 17]: Network Calculus (NC) [10], Trajectory approach [13] and holistic analysis [23]. NC [30, 29] considers that network elements and arrival flows are characterized by a service curve and arrival curve, respectively. Provided with this information, NC allows computing the maximum delay that each flow can suffer at each network element, as well as the maximum size of the waiting queues and the corresponding departure curves. NC provides deterministic results, but requires the determination of arrival and service curves, usually in the form of bounds, which introduces some degree of pessimism.

The Trajectory approach [22, 25] computes the latest starting time of a packet on the last node visited, then moving backwards through the sequence of nodes visited by the packet (i.e. the packet trajectory), identifying the preceding packets and busy periods that affect the latency of the packet in each node. This approach is the tightest among the three categories, in general, but is is more complex to implement and validate, particularly when considering unconstrained routing schemes in which message flows can cross multiple times.

The holistic analysis [8] is meant for distributed systems and it computes the minimum and maximum response-time of the tasks and uses this jitter to compute the minimum and maximum response-times of the messages they generate. This jitter is then used to reassess the response time of the tasks triggered by the messages. The whole process is repeated until the response times of tasks and messages converge. This method can be simplified if we focus on the network, only, particularly for a multi-hop switched network. At each hop, we use release jitters to compute both the response times of the messages and their jitters, which will be used as release jitters of the next hop. Thus, the response times are computed in just one pass, from source to destination. This approach is more pessimistic than the trajectory approach, as it considers worst-case scenarios on every node that often cannot occur in operation. However, it is simpler to implement and verify, reason why it was adopted in this work as a first approach to show the real-time analyzability of RT-MQTT.

3 OpenFlow Switch Structure

The core network elements of RT-MQTT are OpenFlow switches that forward packets (a.k.a frames) in an SDN environment and can be implemented in software or hardware. SDN decouples the network data and control planes, the former being implemented in the switch to process packets, while the latter is implemented in a separate SDN controller that makes high-level packet handling decisions and configures switches accordingly. OpenFlow is the de facto protocol used for switch-controller interactions (a.k.a. southbound interface). Figure 1 shows the OpenFlow pipeline in the switches that receives frames at ingress ports and sends them to a set of flow tables installed by a controller. Each flow table contains a set of Flow Entries with: i) a priority to sort matching; ii) filters to identify incoming frames; iii) associated instructions; and iv) fields for statistics.

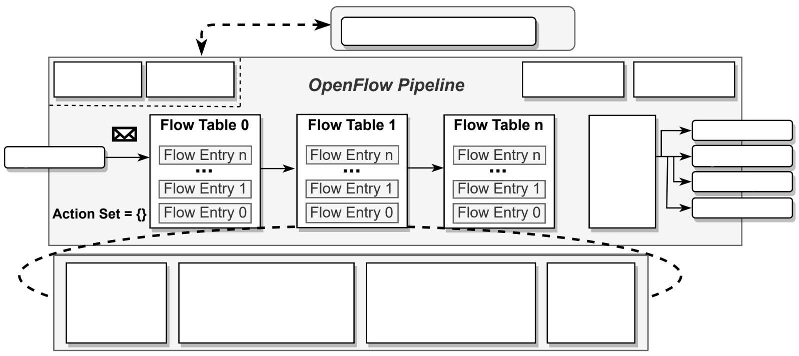


Figure 1 Overview of the OpenFlow pipeline.

When a packet matches the filters of a given flow entry, the switch performs the associated instructions. Once the packet reaches the last table or is not directed to a subsequent one by the matched entry, the switch executes the current action set that may send the packet to a group table or forward it to an OpenFlow egress port. Conversely, if a packet does not match any flow entry, it is handled as a table-miss. The actions executed in this case are also configurable including dropping the packet, forwarding to a subsequent table, or sending to the controller on a packet-in message via the control channel. If the table-miss flow entry does not exist, unmatched packets are dropped by default. Sending the controller a flow request (packet-in message) for every unknown packet can overwhelm the controller since it has to determine the forwarding path and forwarding rules for every new packet and then install them in the flow tables in all the involved switches. Finally, each egress port features several prioritized queues to enforce traffic segregation for time-sensitive flows.

4 RT-MQTT Network Architecture

The RT-MQTT network architecture (Figure 2) is based on the MQTT application layer protocol supported by OpenFlow switches. MQTT permits the use of multiple brokers for fault-tolerance and load balancing purposes, allowing the system to scale and tolerate broker failures. Similarly, Openflow also features mechanism that allow networks to scale and tolerate faults [2]. However, scalability and fault-tolerance mechanisms are out of the scope of this paper and will not be further addressed.

The RT-MQTT topology comprises an OpenFlow controller (OF-Controller), OpenFlow switches (OF-Switches), an MQTT broker, a real-time network manager (RT-NM) and MQTT clients (IIoT nodes). The OF-Controller is connected to the OF-Switches, having a global view of the network and storing all information in the OF-DataBase (OF-DB). RT-MQTT allows applications to explicitly specify real-time requirements in the User Properties of their MQTT packets. The RT-NM intercepts all MQTT messages to extract possible real-time requirements data. It is logically placed between MQTT clients and the broker, desirably executing in the same node. These requirements are subsequently conveyed to the OF-Controller that processes them and manages the flow tables of the OF-Switches to create corresponding real-time channels.

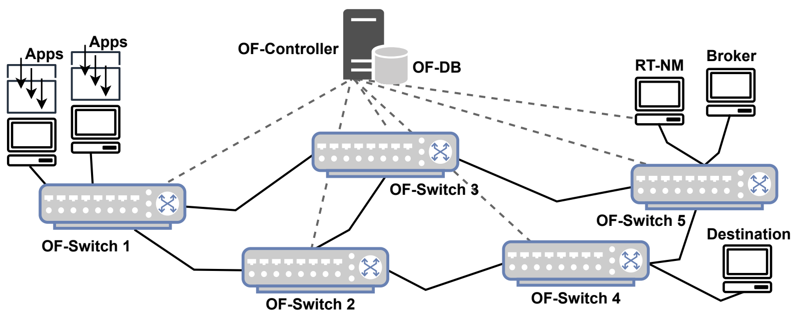


Figure 2 High-level RT-MQTT system architecture.

4.1 Message Model

RT-MQTT classifies packet flows (a.k.a messages) in `real-time`, or time-sensitive, and `non-real-time`, such as normal MQTT messages and general background traffic. Client nodes are assumed to run an operating system with minimal real-time capabilities (we use real-time enabled Linux with regular network stack) and can generate real-time and non-real-time traffic concurrently. We model each real-time message m_i as sporadic. The message set Γ, composed of N messages, is defined as follows:

$$\Gamma = \{m_i(C_i, PS_i, D_i, T_i, P_i, S_i, DS_i, \mathcal{L}_i, n_i), i = 1...N\} \tag{1}$$

The semantics of the parameters are the following:

$\boldsymbol{i}$	: message index used as identifier;
$\boldsymbol{C_i}$	: total transmission time of the message;
$\boldsymbol{PS_i}$	: maximum packet size among the packets that compose m_i;
$\boldsymbol{D_i}$	: deadline, maximum allowed time between transmission and reception of a message;
$\boldsymbol{T_i}$	: minimum interval between consecutive message source publications, with $D_i \leq T_i$;
$\boldsymbol{P_i}$	: message priority;
$\boldsymbol{S_i}$	: source node;
$\boldsymbol{DS_i}$	: destination node;
$\boldsymbol{\mathcal{L}_i}$	: set of links that m_i passes through, including local-links and inter-links;
$\boldsymbol{n_i}$	: number of links that m_i crosses, i.e., $n_i = \|\mathcal{L}_i\|$.

As common in IIoT applications we consider single packet messages. Concerning the path, each element in $\mathcal{L}_i$ has a duplet $l =< x, y >$ representing a link l between node/switch x and node/switch y. A link between a node and a switch is called local-link, while the link between two switches is an inter-link. The direction of message transmission in that link is indicated by the sequence within the duplet. The set of links in the route of m_i is $\mathcal{L}_i = \{l_k | k = 1..n_i\}$. Each message has a defined priority assigned in ascending order (larger value of P_i means higher priority). Since MQTT generally relies on unicasting [15], we restrict the analysis to unicast streams, with only one destination port per message.

4.2 Scheduling Model

As common in current networks, packet transmission is non-preemptive. Thus, we use non-preemptive fixed priorities scheduling with FIFO strategy within each priority level. When a message arrives at an ingress port it is processed by the pipeline that defines the interaction with the flow tables in that switch and (in the normal case) places the packet in the output port queue determined by the message path and priority. Note that each output port corresponds to a link in the message path. Generally, message m_i can suffer two types of delays in any output queue, namely blocking and interference delays as explained next:

Blocking delay is the longest time that m_i may have to wait when it arrives at an output port and message m_j with lower priority ($P_j < P_i$) is already being transmitted in that port. This delay can be computed considering the longest packet among all lower priority messages that share the same output port in each switch. For simplicity of analysis we assume that the blocking delay can be as long as the transmission time of the longest configured packet length, i.e. the Maximum Transmission Unit (MTU). We also consider the MTU to be the same in all nodes.

Interference delay is the longest time that message m_i must wait due to the transmission of all messages m_j with higher or same priority ($P_j \geq P_i$) that share the same output port. The worst-case interference delay requires that all messages with similar priority arrive just before m_i arrives and that all higher priority messages arrive with a pattern defined in the next section.

Figure 3 shows the scheduling model of RT-MQTT with its logical channels and the corresponding physical links between source and destination nodes, conveying real-time and non-real-time traffic concurrently. Once the logical channel is established, the route is determined by a sequence of physical links, each one with origin at a specific output port of a particular switch.

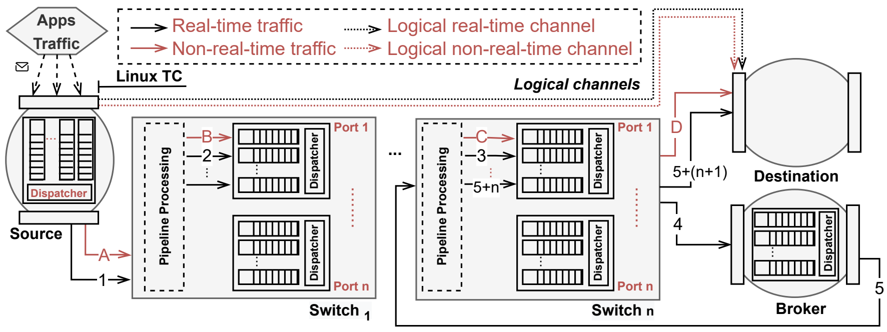


Figure 3 Scheduling model of RT-MQTT.

5 Response Time Analysis

To analyze RT-MQTT we follow a response time analysis method for fixed priorities. We consider the response time of a real-time message as being composed of three parts: (i) the response time from the `source` to the `broker`; (ii) the response time inside the `broker`; and (iii) the response time from the `broker` to the `destination`. While the response times of parts (i) and (iii) are communication delays, (ii) is a computational delay. Both communication and computation delays are interdependent. The holistic approach [23] solves this interdependence iteratively to find the global worst-case response times but, in this case, it requires knowing the computational structure of the broker, which is generally unknown.

Although there is some work in the literature addressing the real-time behavior of the broker, e.g. applying offset-based response time analysis [12], we opted for leaving it outside our current work and focusing on network delays, only. Moreover, the response time analysis for parts (i) and (iii) is similar, just switching the source in (i) with the broker in (iii) and the broker in (i) with the destination in (iii). Therefore, hereinafter we focus only on part (i), i.e., developing a response time analysis for the traffic from source nodes to the broker.

5.1 Response Time Analysis From Source to Broker

We analyze the full path from source to broker as a series of links, each contributing with additional delay. The total response time is then obtained by adding the response times of all individual links in the message path plus the time taken by the switches to move messages across, through their pipelines. We refer to this last component as the switching delay. For the sake of simplicity of analysis, we consider worst-case conditions in all links and switches, at the expense of additional pessimism.

5.1.1 Single link response time analysis

In each link, messages are serialized in the output port of its source, be it a source node or an OF-Switch. As discussed in the previous section, each output port enforces fixed-priorities non-preemptive packet scheduling, with FIFO handling within the same priority level. To analyze this model we apply the conventional computation of response time without preemption in uniprocessors based on cumulative delays [4, 5, 6].

A crucial aspect of this analysis is determining the critical instant, i.e., the message release pattern that leads to the worst-case interference that a message can suffer. In our case, we use as critical instant a synchronous pattern in which a message m_i is released immediately after the release of a lower priority message with maximum size (maximum blocking) and immediately after all equal priority messages and together with all higher priority messages considering their maximum release jitter (maximum interference).

Non-preemptive transmissions are subject to the push-through effect according to which a message m_i can delay higher priority messages through blocking that in turn will generate higher interference in the following instances of m_i itself. For this reason, the worst-case response time of m_i in a given output port may occur at instances beyond the one that is released at a critical instant, within the so-called occupied period. The number of instances following a critical instant that have to be checked to determine the worst-case response time is given by $Q_i = \lceil (w_i + J_i)/T_i \rceil$, where J_i is the release jitter and w_i is the length of the level-i busy period. This period is computed as in Equation 2 using fixed point iteration, starting with $w_i^0 = B_i + C_i$ and ending when $w_i^{n+1} = w_i^n$. The summation term labeled `hep` represents the total interference due to invocations of higher and equal priority messages released strictly before the end of the busy period.

$$w_i^{n+1} = B_i + \sum_{\forall\, j \,\in\, hep(i)} \lceil \frac{w_i^n + J_j}{T_j} \rceil C_j \tag{2}$$

The so-called level-i occupied period starts at the critical instant and extends until the following level-i idle period, i.e., when the queues of priorities P_i and higher of the output port become empty. This period includes the q previous instances of m_i and it represents the maximum delay that the next instance of m_i can suffer before starting transmission. Equation 3 gives an upper bound to its length $v_i(q)$ and can also be solved through fixed-point iteration with a possible initial value $v_i^0(q) = B_i + qC_i$ and ending when either $v_i^{n+1}(q) = v_i^n(q)$ or when $v_i^{n+1}(q) + C_i - qT_i > D_i - J_i$ in which case the deadline cannot be guaranteed.

$$v_i^{n+1}(q) = \ B_i + qC_i + \sum_{\forall\ j \in\, hep(i)} (\lfloor \frac{v_i^n(q) + J_j}{T_j} \rfloor + 1) C_j \tag{3}$$

The worst-case response time of an instance preceded by q instances can then be obtained with $RT_i(q) = v_i(q) + C_i - qT_i$. Equation 4 gives us the worst-case response time for m_i.

$$RT_i = \ max_{q=0,1\ldots Q_i-1}(v_i(q) + C_i - qT_i) \tag{4}$$

5.1.2 Switching delay calculation

As referred before, the switching delay affects all messages crossing a switch even without blocking or interference by other messages. Figure 4 [27] helps understanding the switching delay of OpenFlow switches, particularly implemented in software, as in our system.

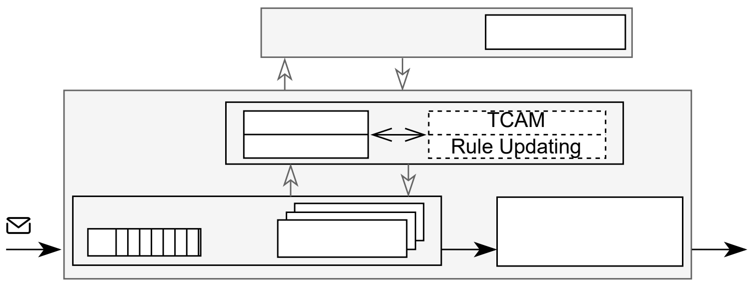


Figure 4 The switching delay in an OF-Switch.

We consider the switching delay as resulting from three components, the time to manage the input queues (essentially memory operations), the time to process the flow tables and the time to execute the action set. We refer to the switching delay affecting m_i as SD_i. The switching delay must be bounded for a bounded response time, but this bound can be different for each switch because it depends on the input load, on the number of flow tables and on the complexity of the action set. These aspects have been studied in the literature. For example, the switching delay increases with increasing flow arrival rate since it requires more bandwidth from the local CPU in the SDN switch [27]. The switching delay may also increase with decreasing packet size because it typically promotes higher arrival rate again imposing higher processing burden on the switch CPU [3]. The forwarding table is another source of overhead. The longer it is the higher the switching delay. In practice, the SD is normally measured and then used in the analysis.

5.1.3 Response time calculation algorithm

Algorithm 1 shows the computation of the worst-case response time for m_i for the total route from source to broker, taking as input the network topology and the message set.

Algorithm 1 WCRT calculation for m_i.

```
Input: $G, \Gamma$
Output: $RT_i^{TotalRoute}$
/* A. Compute delays and jitter at source node:        */
1   $RT_i^{TotalRoute} = ResponseTimeCalc(i, 1)$
2   $J_i^{acc} = J_i + J_{i,1}^{QP}$
3   $k = 2$
/* B. Compute delays and jitter for each switch:       */
4   while $k \leq n_i$ do
5       $SD_{i,k} = SwitchingDelayCalc(i, k)$
6       $J_i^{acc} = J_i^{acc} + J_{i,k}^{SD}$
7       $RT_{i,k} = ResponseTimeCalc(i, k)$
8       $J_i^{acc} = J_i^{acc} + J_{i,k}^{QP}$
9       $RT_i^{TotalRoute} = RT_i^{TotalRoute} + RT_{i,k} + SD_{i,k}$
10      $k = k + 1$
11  end while
```

The algorithm has two parts. The first part (A) processes the output link of the source node (corresponding to k=1), including the computation of the response time ($ResponseTimeCalc(i, 1)$ according to Equation 4) and output jitter (J_i plus $J_{i,1}^{QP}$ which is the response time jitter). These values are used to initialize two accumulators that will allow building the response time for the total route ($RT_i^{TotalRoute}$) and the release jitter (J_i^{acc}) that will affect each subsequent link along the path. The second part in the algorithm (B) is very similar to the first one with the only difference that it also computes the switching delay introduced by the switch under analysis and the additional jitter it may cause. This is repeated from the second to the last link in the path of m_i.

6 Performance Assessment

We validate the analysis presented before with an empirical study using the Mininet emulation framework applied in multiple scenarios of different complexity. For consistency, we also focus on the publishers side, measuring the time intervals from the publication instant (writing to the respective socket) to the respective reception at the broker and comparing with the corresponding analytical response times.

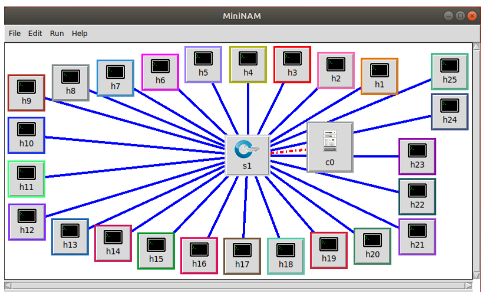
6.1 Emulation Setup

We used the Mininet virtual network emulator, version 2.3.0d6 `http://mininet.org/`, together with Eclipse Mosquitto [11] (v2.0.10) and Eclipse Paho MQTT library to create MQTT clients and broker. Mininet is executed on a laptop computer featuring a 4.9 GHz Intel Core i7 processor and 16 GB of RAM. The SDN controller is the RYU OF-Controller [1] and it is executed on the same laptop computer.

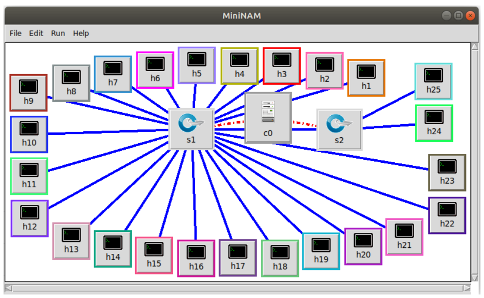
In the emulation experiments, the QoS of all MQTT messages is set to 1 (deliver at least once). This QoS level favors reliability over timeliness given its positive acknowledge and retry mechanism, and it was used since fault-tolerance is important for many IIoT applications. However, this is not expected to have a significant impact on cabled Ethernet networks, given their low error rate.

The operational environment included heterogeneous data exchanges mimicking the diversity of industrial scenarios created with the *Distributed Internet Traffic Generator* (D-ITG) `http://traffic.comics.unina.it/software/ITG/` for TCP packets, *VLC media player* to generate audio/video streams and *vsftpd* to transfer files using the File Transfer Protocol (FTP) `https://linuxconfig.org/how-to-setup-and-use-ftp-server-in-ubuntu-linux`. These were all non-real-time traffic sources with bandwidth limited to 10 Mbit/s, 32 kbit/s, and 800 kbit/s for D-ITG, VLC, and vsftpd, respectively. All links are configured with 100 Mbit/s capacity as still commonly found in industry.

The experiments consider two network topologies with different levels of complexity, named `Single-Switch` and `Dual-Switch`, comprising 1 and 2 OF-Switches (Figure 5). For each topology we generate three different load-levels, labeled `A`, `B`, and `C`, involving publications in different real-time topics. The real-time messages were published with a nominal period chosen to cover the interval $[2\ 15]ms$ and using a single Ethernet packet with maximum size (1500 bytes), leading to a transmission time of $123\mu s$. The priority assignment was Deadline-Monotonic. Since we were interested in assessing the accuracy of the analysis, we considered schedulable message sets, only.


MiniNAM
File Edit Run Help
s1
c0
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25

(a) Single-Switch network topology.


MiniNAM
File Edit Run Help
s1
c0
s2
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25

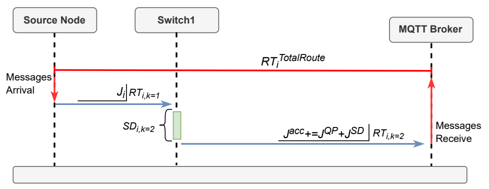
(b) Dual-Switch network topology.

Figure 5 Network topologies used in Mininet.

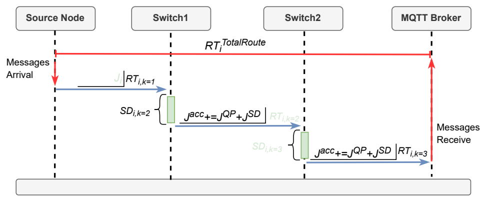
Both topologies include the OF-controller (node `c0`), as well as the `RT-NM` and the MQTT broker (both in node `h24`), plus up to 20 MQTT publishers (nodes `h1` to `h20`), each publishing a single real-time message (m_1 to m_{20}, resp.) to the broker, favoring contention in the network and not in the nodes network stacks. In `Load-Level A` just 5 publishers are active (nodes $h1$ to $h5$), `Load-Level B` uses 10 publishers (nodes $h1$ to $h10$) and `Load-Level C` uses all the 20 publisher nodes. All configurations considered the non-MQTT traffic including D-ITG data from node `h21`, VLC streams from `h22` and *vsftpd* data from `h23`, all directed to node `h25` which is also the subscriber of all MQTT topics. On our measurement host, we use `libpcap` to timestamp the packets for each flow, measuring the associated jitters and the switching delays. For reduced interference, we first log the timestamps in memory and when the experiment is completed, the results are dumped to disk and processed, enabling us to calculate the switching delay and response times.

6.2 Experimental Results

The measurement points in the experiments are shown in Figure 6. The experiments tested all combinations of topology {Single-Switch & Dual-Switch} and load-levels {A, B, C}. Each combination was executed 1000 times with each publisher generating 100 messages per run.



(a) Response time in the Single-Switch topology.

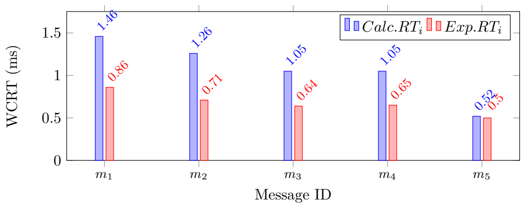


(b) Response time in the Dual-Switch topology.

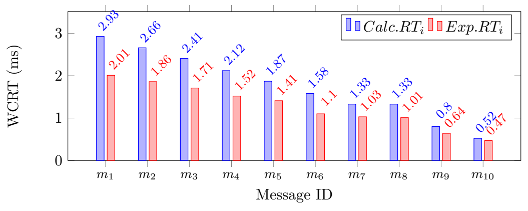
Figure 6 Response time measurements.

Figures 7 and 8 show the analytical worst-case response times ($CalcRT_i$) against the maximum observed response times in the experiments ($ExpRT_i$) for the real-time messages.

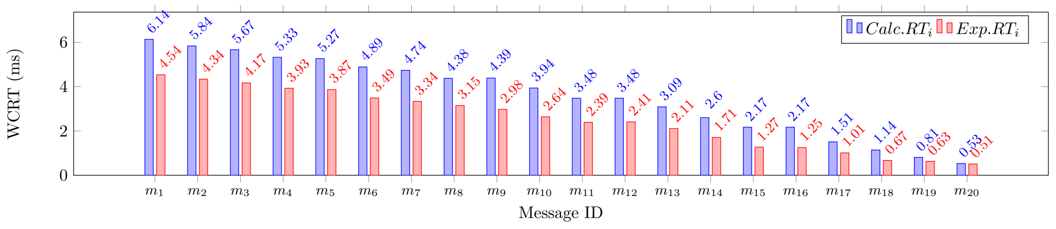
From these figures we can observe three main aspects. Firstly, for all real-time messages we see $CalcRT > ExpRT$, meaning the proposed analysis is safe. Secondly, the difference $CalcRT - ExpRT$, which approximates the pessimism of the analysis, is relatively small for higher priority messages, increasing as the priority decreases. This is expected since the analysis of the lower priority messages includes more pessimistic assumption, e.g. in the interference. Finally, both $CalcRT$ and $ExpRT$ increase similarly with the load level and



(a) Load-level A.

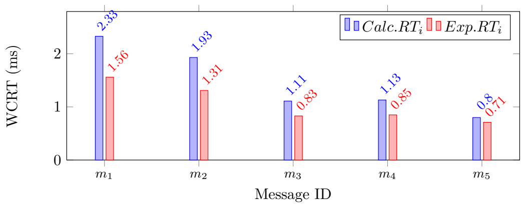


(b) Load-level B.

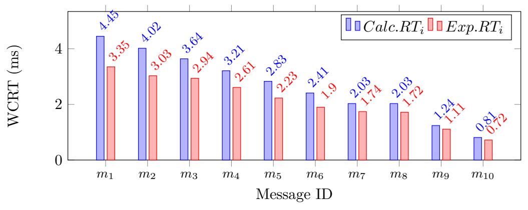


(c) Load-level C.

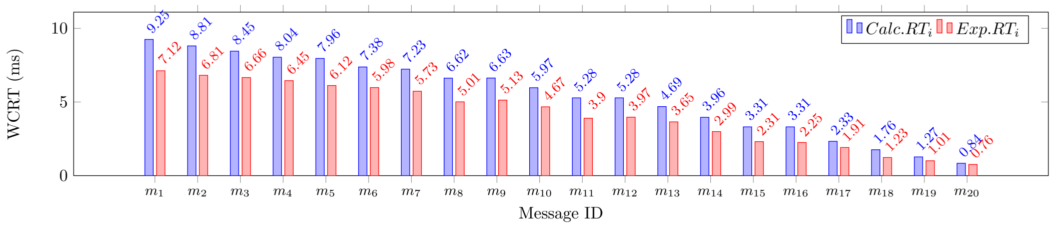
Figure 7 Analytical (CalcRT) versus observed (ExpRT) WCRT for Single-Switch topology.



(a) Load-Level A.



(b) Load-Level B.



(c) Load-Level C.

Figure 8 Analytical (CalcRT) versus observed (ExpRT) WCRT for the Dual-Switch topology.

with the number of switches. Indirectly we can also see a consistent behavior of the real-time messages, despite the reasonable load of non-real-time traffic in the background. All details concerning the message sets and the analytical and experimental results are shown in tabular form in Annex A, in Tables 1–6.

7 Conclusions and Future Work

Despite the increasing popularity of MQTT in the scope of IoT and IIoT, its Quality-of-Service policies do not support timeliness requirements. A recent work addressed this limitation proposing a set of extensions to the MQTT protocol that allow applications to specify real-time requirements (RT-MQTT). Such specifications are then used by a resource manager, implemented in SDN/Openflow, to create real-time channels with suitable attributes, thus instantiating adequate network reservations to enforce the desired temporal behavior.

In this paper, we show that it is possible to apply existing response time analysis to RT-MQTT on the multi-hop SDN/OpenFlow switched network to derive worst-case response time upper bounds to the real-time traffic. In particular, we used the standard response time analysis for non-preemptive fixed-priority scheduling of sporadic messages. The analysis is validated empirically within the Mininet emulator framework, being safe and with relatively low pessimism, particularly for the higher priority traffic. Future work includes the analysis of the broker temporal behavior to support an end-to-end (publisher-to-subscriber) delay model. We will also apply other analytical techniques, e.g., the trajectory approach, in an attempt to reduce the analysis pessimism.

References

1 What's ryu. URL: https://ryu-sdn.org/.

2 M. Bala Krishna and Pascal Lorenz. Proactive replication scheme for resilient content delivery in software defined networks. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2019. doi:10.1109/GLOBECOM38437.2019.9013441.

3 Andrea Bianco, Robert Birke, Luca Giraudo, and Manuel Palacin. Openflow switching: Data plane performance. In *2010 IEEE International Conference on Communications*, pages 1–5. IEEE, 2010.

4 Reinder J Bril, Johan J Lukkien, and Wim FJ Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pages 269–279. IEEE, 2007.

5 Reinder J Bril, Johan J Lukkien, and Wim FJ Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems*, 42(1):63–119, 2009.

6 Robert I Davis and Alan Burns. Response time upper bounds for fixed priority real-time systems. In *2008 Real-Time Systems Symposium*, pages 407–418. IEEE, 2008.

7 Yong-Seong Kim et al. MQTT Broker with Priority Support for Emerg. Events in IoT. *Sensors and Materials*, 2018.

8 J Javier Gutiérrez, J Carlos Palencia, and Michael Gonzalez Harbour. Holistic schedulability analysis for multipacket messages in afdx networks. *Real-Time Systems*, 50(2):230–269, 2014.

9 Hamidreza Kasaei and Mohammadreza Kasaei. Mvgrasp: Real-time multi-view 3d object grasping in highly cluttered environments. *arXiv preprint*, 2021. arXiv:2103.10997.

10 Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet.* Springer, 2001.

11 Roger A Light. Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, 2(13), 2017.

12 Jukka Mäki-Turja, Kaj Hänninen, and Mikael Sjödin. On sustainability for offset based response-time analysis. In *7th Conference on the Engineering of Computer Based Systems*, pages 1–7, 2021.

13 Steven Martin and Pascale Minet. Schedulability analysis of flows scheduled with fifo: application to the expedited forwarding class. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 8–pp. IEEE, 2006.

14 Alessandro Massaro, Giuseppe Mastandrea, Luigi D'Oriano, Giuseppe Rocco Rana, Nicola Savino, and Angelo Galiano. Systems for an intelligent application of automated processes in industry: a case study from "pmi iot industry 4.0" project. In *2020 IEEE International Workshop on Metrology for Industry 4.0 IoT*, pages 21–26, 2020. doi:10.1109/MetroInd4.0IoT48571.2020.9138231.

15 Jun-Hong Park, Hyeong-Su Kim, and Won-Tae Kim. Dm-mqtt: An efficient mqtt based on sdn multicast for massive iot communications. *Sensors*, 18(9):3071, 2018.

16 Changheon Oh Seongjin Kim. A Study on Method for Message Processing by Priority in MQTT Broker. *JKIICE-Journal of the Korea Institute of Information and Communication Engineering*, Jul. 2017.

17 Lui Sha, Tarek Abdelzaher, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, Aloysius K Mok, et al. Real time scheduling theory: A historical perspective. *Real-time systems*, 28(2):101–155, 2004.

18 Ehsan Shahri, Paulo Pedreiras, and Luis Almeida. Enhancing mqtt with real-time and reliable communication services. In *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, pages 1–6. IEEE, 2021.

19 Ehsan Shahri, Paulo Pedreiras, and Luis Almeida. Extending mqtt with real-time communication services based on sdn. In *2022 Sensor Applications in Industrial Automation (ISSN 1424-8220)*, pages 1–6. Sensors SAIA SI, 2022.

20 OASIS Standard. Mqtt version 5.0. *Retrieved June*, 22:2020, 2019.

21 Hiroshi Mineno Takuma Tachibana, Tetsuo Furuichi. Implementing and Evaluating Priority Control Mechanism for Heterogeneous Remote Monitoring IoT System. *MOBIQUITOUS '16 Adjunct Proceedings, Hiroshima, Japan*, December,01,2016.

22 Xueqian Tang, Qiao Li, Guangshan Lu, and Huagang Xiong. A revised trajectory approach for the worst-case delay analysis of an afdx network. *IEEE Access*, 7:142564–142573, 2019. doi:10.1109/ACCESS.2019.2943543.

23 Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and microprogramming*, 40(2-3):117–134, 1994.

24 Jorge Otávio Trierweiler. Real-time optimization of industrial processes. In *Encyclopedia of Systems and Control*, pages 1827–1836. Springer, 2021.

25 Long Yan, Zexiong Luo, Xueqian Tang, and Yunwen Kong. Timing analysis of rate-constrained traffic in ttethernet using extended trajectory approach. In *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*, pages 1039–1042. IEEE, 2020.

26 Shen Yin, Juan J Rodriguez-Andina, and Yuchen Jiang. Real-time monitoring and control of industrial cyberphysical systems: With integrated plant-wide monitoring and control framework. *IEEE Industrial Electronics Magazine*, 13(4):38–47, 2019.

27 Ting Zhang and Bin Liu. Exposing end-to-end delay in software-defined networking. *International Journal of Reconfigurable Computing*, 2019, 2019.

28 Luxi Zhao, Paul Pop, and Silviu S. Craciunas. Worst-case latency analysis for ieee 802.1qbv time sensitive networks using network calculus. *IEEE Access*, 6:41803–41815, 2018. doi:10.1109/ACCESS.2018.2858767.

29 Luxi Zhao, Paul Pop, Qiao Li, Junyan Chen, and Huagang Xiong. Timing analysis of rate-constrained traffic in TTEthernet using network calculus. *Real-Time Systems*, 53(2):254–287, March 2017. doi:10.1007/s11241-016-9265-0.

30 Boyang Zhou, Isaac Howenstine, Siraphob Limprapaipong, and Liang Cheng. A survey on network calculus tools for network infrastructure in real-time systems. *IEEE Access*, 8:223588–223605, 2020.

A Detailed Experimental Results

Tables 1–3 for {Single-Switch} network topology and Tables 4–6 for {Dual-Switch} network topology show the message set details and associated worst-case response time analytical and observed values in three load-levels {A, B, C}, corresponding to Figures 7 and 8.

Table 1 Single-Switch network topology/load-level A.

Experiment	Message Set Parameters						Response Time Analytical Values								Results	
	m_i	PS_i	IL_i	C_i	D_i	T_i	h_i/P_i	B_i	I_i	J_i	Q_i	v_i	TS_i	SD_i	$Calc.RT_i$	$Exp.RT_i$
	$m_{\#}$	$(Bytes)$	$(Mbps)$	(μs)	(ms)	(ms)	$h\#/p\#$	(μs)	(ms)	(μs)	$q\#$	(ms)	$\#$	(μs)	(ms)	(ms)
Single-Switch Load-Level: A	m_1	1500	12.3	123	14	15	h1/p1	123	1.03	37	0	1.27	6	31	1.46	0.86
	m_2	1500	12.3	123	14	15	h2/p2	123	0.77	36	0	1.01	6	32	1.26	0.71
	m_3	1500	12.3	123	5	6	h3/p3	123	0.52	36	0	0.77	6	31	1.05	0.64
	m_4	1500	12.3	123	5	6	h4/p3	123	0.52	37	0	0.77	6	32	1.05	0.65
	m_5	1500	12.3	123	1	2	h5/p5	123	0	37	0	0.24	6	32	0.52	0.50

Table 2 Single-Switch network topology/load-level B.

Experiment	Message Set Parameters						Response Time Analytical Values								Results	
	m_i	PS_i	IL_i	C_i	D_i	T_i	h_i/P_i	B_i	I_i	J_i	Q_i	v_i	TS_i	SD_i	$Calc.RT_i$	$Exp.RT_i$
	$m_{\#}$	$(Bytes)$	$(Mbps)$	(μs)	(ms)	(ms)	$h\#/p\#$	(μs)	(ms)	(μs)	$q\#$	(ms)	$\#$	(μs)	(ms)	(ms)
Single-Switch Load-Level: B	m_1	1500	13.53	123	14	15	h1/p1	123	2.33	48	0	2.67	6	34	2.93	2.01
	m_2	1500	13.53	123	14	15	h2/p2	123	2.13	48	0	2.38	6	34	2.66	1.86
	m_3	1500	13.53	123	11	12	h3/p3	123	1.88	47	0	2.13	6	35	2.41	1.71
	m_4	1500	13.53	123	11	12	h4/p4	123	1.59	46	0	1.84	6	34	2.12	1.52
	m_5	1500	13.53	123	8	9	h5/p5	123	1.34	47	0	1.59	6	33	1.87	1.41
	m_6	1500	13.53	123	8	9	h6/p6	123	1.06	47	0	1.30	6	34	1.58	1.10
	m_7	1500	13.53	123	5	6	h7/p7	123	0.81	46	0	1.05	6	34	1.33	1.03
	m_8	1500	13.53	123	5	6	h8/p7	123	0.81	46	0	1.05	6	35	1.33	1.01
	m_9	1500	13.53	123	1	2	h9/p9	123	0.28	47	0	0.53	6	35	0.80	0.64
	m_{10}	1500	13.53	123	1	2	h10/p10	123	0	46	0	0.24	6	35	0.52	0.47

Table 3 Single-Switch network topology/load-level C.

Experiment	Message Set Parameters						Response Time Analytical Values								Results	
	m_i	PS_i	IL_i	C_i	D_i	T_i	h_i/P_i	B_i	I_i	J_i	Q_i	v_i	TS_i	SD_i	$Calc.RT_i$	$Exp.RT_i$
	$m_{\#}$	$(Bytes)$	$(Mbps)$	(μs)	(ms)	(ms)	$h\#/p\#$	(μs)	(ms)	(μs)	$q\#$	(ms)	$\#$	(μs)	(ms)	(ms)
Single-Switch Load-Level: C	m_1	1500	111	123	14	15	h1/p1	123	5.60	61	0	5.85	6	42	6.14	4.54
	m_2	1500	111	123	14	15	h2/p2	123	5.31	61	0	5.55	6	42	5.84	4.34
	m_3	1500	111	123	11	12	h3/p3	123	5.14	60	0	5.38	6	42	5.67	4.17
	m_4	1500	111	123	11	12	h4/p4	123	4.80	61	0	5.04	6	41	5.33	3.93
	m_5	1500	111	123	8	9	h5/p5	123	4.74	60	0	4.98	6	42	5.27	3.87
	m_6	1500	111	123	8	9	h6/p6	123	4.35	60	0	4.60	6	43	4.89	3.49
	m_7	1500	111	123	6	7	h7/p7	123	4.25	59	0	4.49	6	43	4.74	3.34
	m_8	1500	111	123	6	7	h8/p8	123	3.85	60	0	4.09	6	41	4.38	3.15
	m_9	1500	111	123	4	5	h9/p9	123	2.98	59	0	4.10	6	42	4.39	2.98
	m_{10}	1500	111	123	4	5	h10/p10	123	3.40	59	0	3.65	6	42	3.94	2.64
	m_{11}	1500	111	123	4	5	h11/p11	123	2.95	61	0	3.20	6	42	3.48	2.39
	m_{12}	1500	111	123	4	5	h12/p11	123	2.95	60	0	3.20	6	43	3.48	2.41
	m_{13}	1500	111	123	2	3	h13/p13	123	2.55	60	1	2.80	6	42	3.09	2.11
	m_{14}	1500	111	123	2	3	h14/p14	123	2.07	60	1	2.31	6	41	2.60	1.71
	m_{15}	1500	111	123	2	3	h15/p15	123	1.64	59	1	1.88	6	42	2.17	1.27
	m_{16}	1500	111	123	2	3	h16/p15	123	1.64	59	1	1.88	6	43	2.17	1.25
	m_{17}	1500	111	123	1	2	h17/p17	123	0.98	61	1	1.23	6	43	1.51	1.01
	m_{18}	1500	111	123	1	2	h18/p18	123	0.61	61	1	0.85	6	42	1.14	0.67
	m_{19}	1500	111	123	1	2	h19/p19	123	0.28	60	1	0.53	6	42	0.81	0.63
	m_{20}	1500	111	123	1	2	h20/p20	123	0	60	1	0.24	6	42	0.53	0.51

Table 4 Dual-Switch network topology/load-level A.

	Message Set Parameters						Response Time Analytical Values								Results	
Experiment	m_i $m_\#$	PS_i $(Bytes)$	IL_i $(Mbps)$	C_i (μs)	D_i (ms)	T_i (ms)	h_i/P_i $h\#/p\#$	B_i (μs)	I_i (ms)	J_i (μs)	Q_i $q\#$	v_i (ms)	TS_i $\#$	SD_i (μs)	$Calc.RT_i$ (ms)	$Exp.RT_i$ (ms)
	m_1	1500	12.3	123	14	15	h1/p1	123	1.55	57	0	1.92	6	68	2.33	1.56
	m_2	1500	12.3	123	14	15	h2/p2	123	1.15	57	0	1.52	6	66	1.93	1.31
Dual-Switch Load-Level: A	m_3	1500	12.3	123	5	6	h3/p3	123	0.79	56	0	1.16	6	68	1.11	0.83
	m_4	1500	12.3	123	5	6	h4/p3	123	0.79	57	0	1.16	6	67	1.13	0.85
	m_5	1500	12.3	123	1	2	h5/p5	123	0	56	0	0.37	6	67	0.80	0.71

Table 5 Dual-Switch network topology/load-level B.

	Message Set Parameters						Response Time Analytical Values								Results	
Experiment	m_i $m_\#$	PS_i $(Bytes)$	IL_i $(Mbps)$	C_i (μs)	D_i (ms)	T_i (ms)	h_i/P_i $h\#/p\#$	B_i (μs)	I_i (ms)	J_i (μs)	Q_i $q\#$	v_i (ms)	TS_i $\#$	SD_i (μs)	$Calc.RT_i$ (ms)	$Exp.RT_i$ (ms)
	m_1	1500	13.53	123	14	15	h1/p1	123	3.54	72	0	4.01	6	78	4.45	3.35
	m_2	1500	13.53	123	14	15	h2/p2	123	3.20	72	0	3.57	6	78	4.02	3.03
	m_3	1500	13.53	123	11	12	h3/p3	123	2.83	72	0	3.20	6	79	3.64	2.94
	m_4	1500	13.53	123	11	12	h4/p4	123	2.39	71	0	2.76	6	77	3.21	2.61
Dual-Switch Load-Level: B	m_5	1500	13.53	123	8	9	h5/p5	123	2.02	71	0	2.39	6	77	2.83	2.23
	m_6	1500	13.53	123	8	9	h6/p6	123	1.59	73	0	1.96	6	77	2.41	1.90
	m_7	1500	13.53	123	5	6	h7/p7	123	1.21	71	0	1.58	6	78	2.03	1.74
	m_8	1500	13.53	123	5	6	h8/p7	123	1.21	72	0	1.58	6	78	2.03	1.72
	m_9	1500	13.53	123	1	2	h9/p9	123	0.43	72	0	0.79	6	79	1.24	1.11
	m_{10}	1500	13.53	123	1	2	h10/p10	123	0	71	0	0.36	6	77	0.81	0.72

Table 6 Dual-Switch network topology/load-level C.

	Message Set Parameters						Response Time Analytical Values								Results	
Experiment	m_i $m_\#$	PS_i $(Bytes)$	IL_i $(Mbps)$	C_i (μs)	D_i (ms)	T_i (ms)	h_i/P_i $h\#/p\#$	B_i (μs)	I_i (ms)	J_i (μs)	Q_i $q\#$	v_i (ms)	TS_i $\#$	SD_i (μs)	$Calc.RT_i$ (ms)	$Exp.RT_i$ (ms)
	m_1	1500	111	123	14	15	h1/p1	123	8.41	86	0	8.78	6	105	9.25	7.12
	m_2	1500	111	123	14	15	h2/p2	123	7.97	86	0	8.33	6	104	8.81	6.81
	m_3	1500	111	123	11	12	h3/p3	123	7.73	86	0	8.10	6	105	8.45	6.66
	m_4	1500	111	123	11	12	h4/p4	123	7.20	85	0	7.75	6	105	8.04	6.45
	m_5	1500	111	123	8	9	h5/p5	123	7.12	84	0	7.36	6	106	7.96	6.12
	m_6	1500	111	123	8	9	h6/p6	123	6.54	85	0	6.91	6	105	7.38	5.98
	m_7	1500	111	123	6	7	h7/p7	123	6.39	85	0	6.76	6	105	7.23	5.73
	m_8	1500	111	123	6	7	h8/p8	123	5.78	86	0	6.15	6	104	6.62	5.01
	m_9	1500	111	123	4	5	h9/p9	123	5.79	84	0	6.15	6	104	6.63	5.13
Dual-Switch Load-Level: C	m_{10}	1500	111	123	4	5	h10/p10	123	5.12	84	0	5.49	6	105	5.97	4.67
	m_{11}	1500	111	123	4	5	h11/p11	123	4.43	85	0	4.80	6	105	5.28	3.90
	m_{12}	1500	111	123	4	5	h12/p11	123	4.43	85	0	4.80	6	105	5.28	3.97
	m_{13}	1500	111	123	2	3	h13/p13	123	3.84	85	1	4.21	6	104	4.69	3.65
	m_{14}	1500	111	123	2	3	h14/p14	123	3.12	86	1	3.49	6	106	3.96	2.99
	m_{15}	1500	111	123	2	3	h15/p15	123	2.47	84	1	2.84	6	105	3.31	2.31
	m_{16}	1500	111	123	2	3	h16/p15	123	2.47	84	1	2.84	6	105	3.31	2.25
	m_{17}	1500	111	123	1	2	h17/p17	123	1.48	85	1	1.85	6	104	2.33	1.91
	m_{18}	1500	111	123	1	2	h18/p18	123	0.92	86	1	1.28	6	106	1.76	1.23
	m_{19}	1500	111	123	1	2	h19/p19	123	0.4	86	1	0.79	6	106	1.27	1.01
	m_{20}	1500	111	123	1	2	h20/p20	123	0	84	1	0.36	6	105	0.84	0.76

Throughput and Memory Optimization for Parallel Implementations of Dataflow Networks Using Multi-Reader Buffers

Martin Letras
Friedrich-Alexander Universität Erlangen-Nürnberg (FAU), Germany

Joachim Falk
Friedrich-Alexander Universität Erlangen-Nürnberg (FAU), Germany

Jürgen Teich
Friedrich-Alexander Universität Erlangen-Nürnberg (FAU), Germany

Abstract

In this paper, we introduce the concept of Multi-Reader Buffers (MRBs) for high throughput and memory-efficient implementation of dataflow applications. Our work is motivated by the huge amount of data that needs to be processed and typically accessed in a FIFO manner, particularly in image and video processing applications. Here, multi-cast, fork, and merge operator implementations known today produce huge memory overheads by storing and communicating copies of the same data. As a remedy, we first introduce MRBs as buffers preserving FIFO semantics for a finite number of readers of the same data while storing each data item only once. Second, we present an approach for memory minimization of data flow networks by replacing all multi-cast actors and connected FIFOs with MRBs. Third, we present a Design Space Exploration approach to selectively replace multi-cast actors with MRBs in order to explore memory, throughput, and processor resource allocation tradeoffs. Our results show that the explored Pareto fronts of our approach improve the solution quality over a reference by 78 % in average for six benchmark applications in terms of a hypervolume indicator.

2012 ACM Subject Classification Hardware → Static timing analysis

Keywords and phrases Dataflow, Memory Optimization, MPSoCs, Design Space Exploration

Digital Object Identifier 10.4230/OASIcs.NG-RES.2023.6

Funding This work has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grants 45098171 and 146371743.

1 Introduction

Generally, an image processing application consists of a graph of image processing filters, where each filter operates on its input and produces transformed image data at its outputs. Two important aspects must be considered in the design of efficient implementations of image processing applications on multicore architectures 1) the application's concurrency and exploitable parallelism and 2) its memory footprint. As imperative programming languages are a poor fit for developing concurrent applications, dataflow processing [4, 7, 17], which is naturally suited to expressing concurrency, is widely adopted to program modern Multi-Processor Systems-on-a-Chip (MPSoCs). Each filter of an image processing application can be modeled by an *actor* consuming and producing data. Each communication between two filters can be realized via a First In First Out (FIFO) buffer that needs to be mapped to a region in memory. Now, one constraint in dataflow modeling is that each FIFO buffer is exclusively written by one producer actor and read by one consumer actor. When multiple consumers require to read the same data, some dataflow modeling frameworks [12, 10, 5] propose to solve this issue by introducing so-called *multi-cast actors* that just read information and replicate it for a finite number of consumers. Accordingly, the introduction of multi-cast actors might negatively impact the *memory footprint* of image processing applications due to the data redundancy induced to respect dataflow semantics.

To avoid any memory overhead and as a first contribution, we introduce the concept of a *Multi-Reader Buffer (MRB)* preserving FIFO semantics for a finite number of readers of the same data while storing each data item only once. As a second contribution, we present an approach for memory minimization of dataflow networks by replacing each multi-cast actor and the FIFOs connected to it by a MRB. But whereas using MRBs instead of multi-cast actors reduces the memory footprint by removing data redundancy thus, delivering optimal memory footprint implementations, this transformation might negatively impact the throughput of a given application, as a bounded-length MRB will free space in its FIFO memory only once the *last* reader will have consumed the respective data. Accordingly, our third contribution is to explore the tradeoff between memory footprint minimization and throughput maximization by *selectively* replacing multi-cast actors by MRBs.

Here, we propose a multi-objective Design Space Exploration (DSE) for the mapping and scheduling of a data flow specification onto symmetric multi-processor target platforms with a global shared memory and each processor having an additional local scratchpad memory. In addition to memory footprint to be minimized and throughput to be maximized, we consider the number of allocated cores for each explored mapping and schedule as an additional cost metric. We subsequently compare our proposed DSE results named MRB_{Explore} to a *Reference* approach that only optimizes the mappings without introducing any MRBs. We also compare the DSE results MRB_{Explore} against an approach named MRB_{Always} in which all multi-cast actors and their adjacent FIFOs are replaced by MRBs. Our experiments show that MRB_{Always} is able to improve the quality of found solutions in terms of a hypervolume indicator by 67 % on average compared to a state-of-the-art reference approach. Moreover, our proposed approach MRB_{Explore} is shown to be able to find even better fronts of solutions by improving the hypervolume to 78 % on average against the approach *Reference*.

This paper is structured as follows: Section 2 presents the state-of-the-art. Section 3 presents the a formalization of the optimization problem. Then, Section 4 presents the semantics of our proposed MRB. Section 5 presents the DSE approach to selectively implement MRBs and the experimental results for six applications in terms of the quality of the found solution sets. Finally, Section 6 concludes this paper.

2 Related Work

Approaches for optimizing parallel implementation of applications specified as dataflow networks [17] perform multi-objective optimization of conflicting design objectives, e.g., throughput, number of allocated cores, and memory footprint. On the one hand, approaches such as [11, 7] optimize dataflow applications' throughput and the number of allocated cores in a given architecture. E.g., [7] proposed a clustering approach of static actors into a so-called cluster. Through the proposed clustering approach, the scheduling of connected static data flow sub-graphs can be coordinated to exploit the predictability and efficiency of the static data flow model. Moreover, clustering reduces scheduling overhead by reducing the number of checking guards of the actors composing a cluster, thus improving the throughput of applications. However, the previously presented approaches do not consider any memory footprint evaluation of implementations during DSE.

On the other hand, approaches for memory footprint minimization can be classified into two main categories: 1) approaches minimizing the size of FIFOs and 2) approaches implementing memory-reuse strategies that allow different FIFOs to be mapped into overlapping

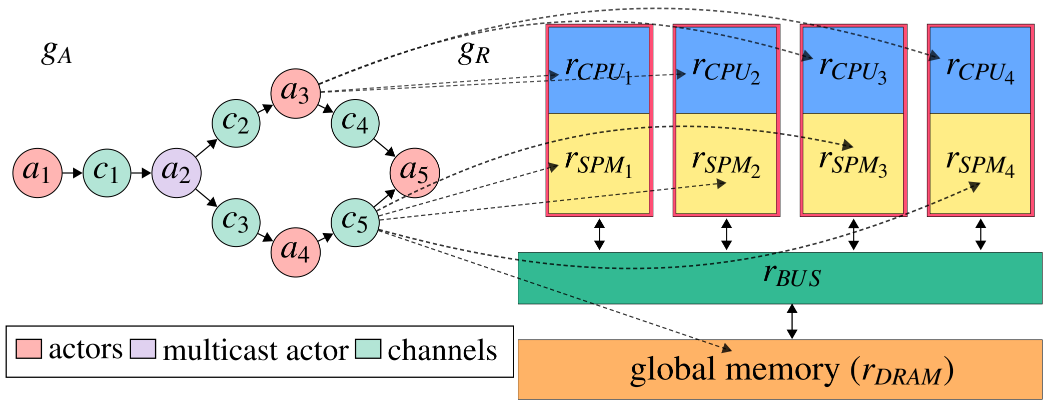


Figure 1 On the left, an application graph g_A consisting of actors connected by communication channels. On the right, a multi-core architecture that is modeled by an architecture graph g_R. Dashed lines represent mappings from actors to processors and channels to memories.

memory spaces or track individual token lifetimes to exploit memory footprint reductions over the execution of an application. In the first category, techniques such as FIFO sizing have been widely studied to reduce the memory footprint of Synchronous Dataflow (SDF) applications [15, 1]. Such approaches determine the minimal buffer size of an SDF application under throughput constraints. However, those approaches do not consider any memory-reuse strategy because each buffer is studied as a separate unit allocated in memory, and no shared memory address space is considered. In the second category, the approach presented in [6] derives overlapping memory allocations for individual tokens communicated during the execution of an SDF graph. As a requirement, the SDF graph has to be transformed into a single-rate SDF graph inducing a significant analysis overhead that leads to an approach ill-suited for usage within a DSE [6].

Apart from performing an agnostic memory footprint minimization, some approaches exploit the knowledge about the application and actor characteristics. For instance, dataflow frameworks [5, 18, 13] targeting image processing apply memory minimization strategies based on the behavior of a set of specialized actors performing operations like multi-cast, fork, and join of data. For instance, the employed memory minimization strategy described in [13] merges all outgoing buffers of a multi-cast actor by replacing them with a broadcast FIFO that supports a single writer but multiple readers [13]. However, this implementation is only able to handle single rate dataflow applications. Moreover, no other design objectives apart from memory footprint are explored. In this paper, we propose a holistic approach that considers not only the minimization of memory footprint but also the mapping and scheduling of communication channels and actors onto an MPSoC as well as the number of allocated CPUs as exploration objectives.

3 Fundamentals

Mapping problems of applications to embedded systems, i.e., multi-core target architectures, are often described by a *specification graph* [2, 16] composed of (i) an *application graph*, (ii) an *architecture graph*, and (iii) *mappings* connecting the application and the architecture graphs.

3.1 Application Graph

An application is modeled as a bipartite graph of actors and channels. First, we formalize actors.

▶ **Definition 1** (Actor [8]). *An actor is a tuple $a = (I, O, \psi, \kappa, \tau)$ containing a set of* actor input ports I *and* actor output ports O. *The function $\psi : O \to \mathbb{N}$ assigns the production rate to all output ports and the function $\kappa : I \to \mathbb{N}$ assigns the consumption rate on each input port per firing. Finally, $\tau \in \mathbb{R}$ represents the actor's execution time.*

Next, we model a given data flow specification of communicating actors by a bipartite *application graph*:

▶ **Definition 2** (Application Graph). *The application graph $g_A = (A, C, E, \delta, \gamma, \varphi)$ is a bipartite graph with its vertices partitioned into a* set of actors A *and a* set of communication channels C. *Each channel represents a FIFO buffer. The set of directed edges $E \subseteq (A.O \times C) \cup (C \times A.I)$ connects actors and channels. The* delay *function $\delta : C \to \mathbb{N}_0$,* capacity *function $\gamma : C \to \mathbb{N}$, and* size *function $\varphi : C \to \mathbb{N}$, respectively, assign each channel a number of initial tokens, a maximal number of tokens that can be stored, and the token size in bytes.*

In Figure 1, an example of an application graph g_{A} consisting of five actors $A = \{a_1, \ldots, a_5\}$ communicating via five communication channels $C = \{c_1, \ldots, c_5\}$ is given.

From an application itself, it is possible to determine the *memory footprint* $M_F = \sum_{\forall c \in C} \gamma(c) \cdot \varphi(c)$ by summing up each channel's memory requirement derived from the channel capacity in tokens $\gamma(c)$ and the token size in bytes $\varphi(c)$.

Generally, an application contains multi-cast actors, e.g., actor a_2 in Figure 1. Multi-cast actors just replicate the data tokens at their input, producing identical copies of data as tokens on the communication channels connected to their output ports. To exemplify, a_2 copies its input tokens to a_3 and a_4 via the communication channels c_2 and c_3, respectively. In the following, let the set of multi-cast actors of an application be denoted by $A_{\mathbf{M}} \subset A$. Now, each multi-cast actor represents an opportunity for memory footprint reduction, as shown in Figure 2. As precisely one channel is connected to each actor port, the domain of the functions δ, γ, and φ can be extended to all the ports of a multi-cast actor. If one of these functions is applied to an actor port, it will be equivalent to applying the function to the channel that is connected to this port. With these definitions, a multi-cast actor satisfies the following sets of constraints:

$$\forall a \in A_{\mathbf{M}} : |a.I| = 1 \land \forall i \in a.I : \kappa(i) = 1 \tag{1}$$

$$\land\ |a.O| > 1 \land \forall o \in a.O : \psi(o) = 1 \tag{2}$$

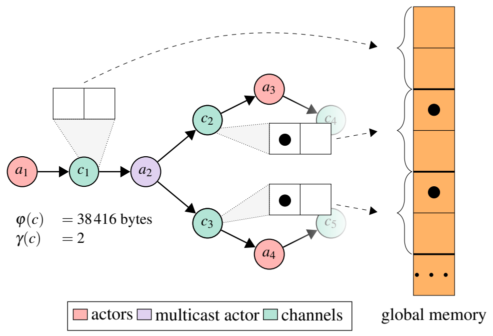
$$\land\ \forall i \in a.I, o \in a.O : \varphi(i) = \varphi(o) \tag{3}$$

$$\land\ \forall o', o'' \in a.O : \delta(o') = 0 \land \gamma(o') = \gamma(o'') \tag{4}$$

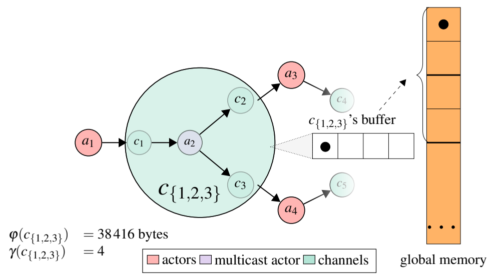
First, a multi-cast actor must have exactly one input port consuming one token per actor firing (see Equation (1)) and at least two output ports, each port producing one token per firing (see Equation (2)). Moreover, the token size of all consumed and produced tokens must be identical (see Equation (3)). Finally, output channels are assumed to be free of any token, and the channel capacities of the output channels are assumed to be identical (see Equation (4)).

3.2 Architecture Graph

A symmetric multi-core target architecture as shown in Figure 1, right, is modeled formally by an *architecture graph*:



(a) Realization of channels as FIFOs allocated in memory.



(b) Concept of a MRB: A given multi-cast actor a_2 and its adjacent channels c_1, c_2, and c_3 are replaced by a single MRB $c_{\{1,2,3\}}$ for memory minimization.

Figure 2 In (a), each channel connected to multi-cast actor a_2 is realized as a FIFO allocated in memory storing the same information. In (b), memory minimization is performed by merging those redundant communication channels into MRB $c_{\{1,2,3\}}$.

▶ **Definition 3** (Architecture Graph). *An architecture graph g_R is a tuple (R, L, b_ϖ) composed of a set of vertices R modeling hardware resources (such as processors, memories, and a bus used to transport data from memories to processors and vice-versa) and a set of edges $L \subseteq R \times R$ denoting communication links. Finally, b_ϖ denotes the bus bandwidth.*

To exemplify, consider again the target architecture shown in Figure 1. The set of resources R is partitioned into a set of CPUs $R_\mathrm{P} = \{r_{\mathrm{CPU}1}, \ldots, r_{\mathrm{CPU}4}\}$, a set of memories $R_\mathrm{M} = \{r_{\mathrm{SPM}1}, \ldots, r_{\mathrm{SPM}4}, r_\mathrm{DRAM}\}$, and the bus r_BUS. Here, each processor $r_{\mathrm{CPU}i} \in R_\mathrm{P} \subset R : 1 \leq i \leq |R_\mathrm{P}|$ is assumed to have a local scratchpad memory $r_{\mathrm{SPM}i} \in R_\mathrm{M} \subset R$ reachable via the link $(r_{\mathrm{CPU}i}, r_{\mathrm{SPM}i}) \in L$. Furthermore, the processors can access the global memory r_DRAM via the bus r_BUS.

3.3 Specification graph

To perform explorations of allocations and mappings of actors to cores, and of channels to memories including the scheduling of actors and of data transfers between resources, a specification contains a set of mappings $M = M_A \cup M_C$ that is partitioned into a set of potential mappings $M_A \subseteq A \times R_\mathrm{P}$ of actors to processors and mappings $M_C \subseteq C \times R_\mathrm{M}$ of channels to memories. Then, a *specification graph* can be formally defined as follows:

▶ **Definition 4** (Specification Graph). *A specification graph $g_S = (A \cup C \cup R, E \cup L \cup M)$ contains the architecture graph g_R, the application graph g_A and the set of potential mappings M.*

3.4 Actor and Communication Channel Binding

A specification, in general, allows for multiple implementations. To derive a specific implementation, a DSE must determine *bindings* for all actors ($\beta_A \subseteq M_A$) and all communication channels ($\beta_C \subseteq M_C$). This step is often also called mapping. Here, each actor must be bound to exactly one processing resource (see Equation (5)). Conversely, each channel must be bound to exactly one memory resource (see Equation (6)).

$$\forall a \in A : |\beta_A \cap (\{a\} \times R_\mathrm{P})| = 1 \tag{5}$$

$$\forall c \in C : |\beta_C \cap (\{c\} \times R_\mathrm{M})| = 1 \tag{6}$$

Moreover, the binding of each channel c is constrained to either be bound to global memory r_{DRAM} or a scratchpad memory $r_{\mathrm{SPM}i}$ local to the processor $r_{\mathrm{CPU}i}$ onto which an actor is bound that either writes to or reads from channel c. This condition can be formalized as follows:

$$\begin{aligned}
&\forall (c,r) \in \beta_C : r = r_{\mathrm{DRAM}} \\
&\vee \exists (a, r_{\mathrm{CPU}i}) \in \beta_A, o \in a.O : (o,c) \in E \wedge r &&= r_{\mathrm{SPM}i} \\
&\vee \exists (a, r_{\mathrm{CPU}i}) \in \beta_A, i \in a.I : (c,i) \in E \wedge r &&= r_{\mathrm{SPM}i}
\end{aligned} \tag{7}$$

Formally, each *feasible implementation* g_{I} must satisfy Equations (5)–(7). Note that different implementations may have identical actor bindings β_A, but differing in *schedules* due to differing channel bindings β_C, a limited bus bandwidth b_ϖ delaying those communications using the bus to transport data, or simply using a different scheduling algorithm. Note also that any processor $r_{\mathrm{CPU}i}$ writing to its local scratchpad memory $r_{\mathrm{SPM}i}$ creates no impact on scheduling because the bus is not used to transfer data. In contrast, the bus is utilized when writing to any other scratchpad or global memory which could create interference. Formally, for any bound channel $(c, r_{\mathrm{m}}) \in \beta_C$, the transfer delay $\tau(c,\eta)$ of transporting η data tokens over the bus assuming no bus contention is calculated as $\tau(c,\eta) = \frac{\varphi(c) \times \eta\ [\mathtt{bytes}]}{b_\varpi\ [\mathtt{Gb/s}]}$[1].

4 Multi-Reader Buffers (MRBs) for Memory Footprint Minimization

Using multi-cast actors in a Dataflow Graph (DFG) may result in sub-optimal implementations in terms of memory footprint. E.g., Figure 2a presents FIFO realizations for channels c_1, c_2, and c_3 of the application graph shown in Figure 1. There, the multi-cast actor a_2 propagates identical data tokens to c_2 and c_3. Figure 2b now introduces our concept of an MRB. By replacing a multi-cast actor and its adjacent channels by a single MRB node in which all outgoing channel buffers are replaced internally by just a single (shared) buffer. Semantically, the MRB acts as a channel in the transformed application graph that technically stores only one copy of live data shared between actors a_3 and a_4.

Formally, the transformation of replacing a given multi-cast actor $a_{\mathbf{m}}$ with an MRB for a given application graph g_{A} is detailed in Algorithm 1. This algorithm returns a transformed application graph where the given multi-cast actor and the channels connected to it have been replaced by a corresponding MRB. For finding minimal memory footprint implementations, Algorithm 1 is simply applied to all multi-cast actors of an application.

Now, we present a possible MRB realization and its principle of operation. By definition, a MRB $c_{\mathbf{m}}$ has one writer $a_{\mathbf{w}}$ and multiple readers $a_{\mathbf{r}_i} \in A_{\mathbf{r}} \subseteq A : 1 \leq i \leq |A_{\mathbf{r}}|$. Each MRB $c_{\mathbf{m}}$ has a write index $\omega(c_{\mathbf{m}}) \in \{0, 1, \ldots, \gamma(c_{\mathbf{m}}) - 1\}$ that indicates the next position in $c_{\mathbf{m}}$'s buffer to be filled with the next token produced by the writer $a_{\mathbf{w}}$. Similarly, each $c_{\mathbf{m}}$ manages read indices $\rho_i(c_{\mathbf{m}}) \in \{-1, 0, 1, \ldots, \gamma(c_{\mathbf{m}}) - 1\} : 1 \leq i \leq |A_{\mathbf{r}}|$, each index $\rho_i(c_{\mathbf{m}})$ indicating a position in $c_{\mathbf{m}}$'s buffer from which the next token consumed by reader $a_{\mathbf{r}_i}$ is read. The special value -1 of a read index $\rho_i(c_{\mathbf{m}})$ denotes that $c_{\mathbf{m}}$ is empty from $a_{\mathbf{r}_i}$'s perspective. Then, the number of available tokens $\mathsf{T}(c_{\mathbf{m}}, a_{\mathbf{r}_i})$ from the perspective of each reader $a_{\mathbf{r}_i}$ and the number of free places $\mathsf{F}(c_{\mathbf{m}})$ in $c_{\mathbf{m}}$ from the perspective of the writer $a_{\mathbf{w}}$ can be determined as follows:

[1] Times for reading and writing local scratchpad data are assumed to be part of each actor's execution time.

Algorithm 1 Multi-Reader Buffer (MRB) replacement.

```
Function insertMRB($g_A, a_m$)
    $C_{\text{del}} \leftarrow \{c \in g_{\text{A}}.C \mid g_{\text{A}}.E \cap (\{c\} \times a_{\mathbf{m}}.I \cup a_{\mathbf{m}}.O \times \{c\}) \neq \emptyset\}$// All channels connected to $a_{\mathtt{m}}$
    $c_{\mathbf{m}} \leftarrow \text{createMRB}(a_{\mathbf{m}}, C_{\text{del}})$    // Create MRB
    $g_{\text{A}}.A \leftarrow g_{\text{A}}.A \setminus \{a_{\mathbf{m}}\}$    // Remove multicast actor
    $c' \leftarrow c : \exists i \in a_{\mathbf{m}}.I, (c, i) \in g_{\text{A}}.E$    // input channel of $a_{\mathtt{m}}$
    $g_{\text{A}}.E \leftarrow g_{\text{A}}.E \cup \{(o, c_{\mathbf{m}}) \mid \exists a \in g_{\text{A}}.A, o \in a.O : (o, c') \in g_{\text{A}}.E\}$    // Connect $c_{\mathtt{m}}$ writer
    for $c'' \in C_{del} \setminus \{c'\}$ do    // All output channels of $a_{\mathtt{m}}$
        $g_{\text{A}}.E \leftarrow g_{\text{A}}.E \cup \{(c_{\mathbf{m}}, i) \mid \exists a \in g_{\text{A}}.A, i \in a.I : (c'', i) \in g_{\text{A}}.E\}$    // Connect $c_{\mathtt{m}}$ reader
        $\gamma(c_{\mathbf{m}}) \leftarrow \gamma(c') + \gamma(c'')$    // Set capacity of MRB
    $\delta(c_{\mathbf{m}}) \leftarrow \delta(c')$    // Set initial tokens for MRB
    $\varphi(c_{\mathbf{m}}) \leftarrow \varphi(c')$    // Set token size of MRB
    $g_{\text{A}}.C \leftarrow \{c_{\mathbf{m}}\} + g_{\text{A}}.C \setminus C_{\text{del}}$    // Replace $C_{\text{del}}$ by $c_{\mathtt{m}}$
    $g_{\text{A}}.E \leftarrow \{(n, m) \in g_{\text{A}}.E \mid n \notin C_{\text{del}} \vee m \notin C_{\text{del}}\}$    // Remove edges connecting removed channels
    return $g_{\text{A}}$
```

$$\mathsf{T}(c_{\mathbf{m}}, a_{\mathbf{r}_i}) = \begin{cases} 0 & \text{if } \rho_i(c_{\mathbf{m}}) < 0 \\ ((\omega(c_{\mathbf{m}}) - \rho_i(c_{\mathbf{m}}) - 1) \bmod \gamma(c_{\mathbf{m}})) + 1 & \text{otherwise} \end{cases} \tag{8}$$

$$\mathsf{F}(c_{\mathbf{m}}) = \gamma(c_{\mathbf{m}}) - \max_{a_{\mathbf{r}_i} \in A_{\mathbf{r}}} \mathsf{T}(c_{\mathbf{m}}, a_{\mathbf{r}_i}) \tag{9}$$

Assuming the reader $a_{\mathbf{r}_i}$ consumes $\kappa(a_{\mathbf{r}_i})$ tokens, then it is blocked from firing as long as $\mathsf{T}(c_{\mathbf{m}}, a_{\mathbf{r}_i}) < \kappa(a_{\mathbf{r}_i})$ holds. Accordingly, upon each read by an actor $a_{\mathbf{r}_i}$, the corresponding read index $\rho_i(c_{\mathbf{m}})$ is updated as follows:

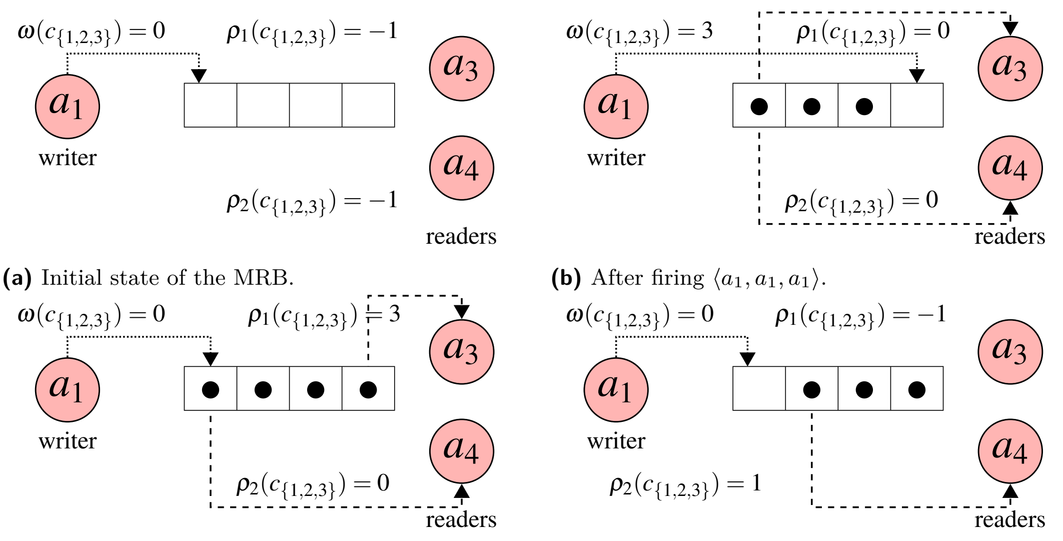
$$\rho_i(c_{\mathbf{m}}) \leftarrow \begin{cases} -1 & \text{if } \mathsf{T}(c_{\mathbf{m}}, a_{\mathbf{r}_i}) = \kappa(a_{\mathbf{r}_i}) \\ (\rho_i(c_{\mathbf{m}}) + \kappa(a_{\mathbf{r}_i})) \bmod \gamma(c_{\mathbf{m}}) & \text{otherwise} \end{cases} \tag{10}$$

Equivalently, assuming the writer $a_{\mathbf{w}}$ produces $\psi(a_{\mathbf{w}})$ tokens, then it is blocked from firing as long as $\mathsf{F}(c_{\mathbf{m}}) < \psi(a_{\mathbf{w}})$ holds. Accordingly, upon each write of actor $a_{\mathbf{w}}$, Equation (11) is applied, which sets each read index $\rho_i(c_{\mathbf{m}})$ with the value $\omega(c_{\mathbf{m}})$ if $\rho_i(c_{\mathbf{m}}) = -1$. Next, Equation (12) is applied, which advances the writer index $\omega(c_{\mathbf{m}})$ by the number of produced tokens.

$$\underset{1 \leq i \leq |A_{\mathbf{r}}|}{\forall} \rho_i(c_{\mathbf{m}}) \leftarrow \begin{cases} \omega(c_{\mathbf{m}}) & \text{if } \rho_i(c_{\mathbf{m}}) = -1 \\ \rho_i(c_{\mathbf{m}}) & \text{otherwise} \end{cases} \tag{11}$$

$$\omega(c_{\mathbf{m}}) \leftarrow (\omega(c_{\mathbf{m}}) + \psi(a_{\mathbf{w}})) \bmod \gamma(c_{\mathbf{m}}) \tag{12}$$

To exemplify, consider the application in Figure 2. For the MRB $c_{\{1,2,3\}}$, the writer $a_{\mathbf{w}}$ is a_1, and the set of readers $A_{\mathbf{r}}$ is $\{a_3, a_4\}$. Of course, our presented MRB realization supports multi-rate dataflow. However, to ease the understanding of the presented example, a_1's production rate is assumed here to be one, i.e., $\psi(a_1) = 1$, and the same holds for the consumption rates of the readers, i.e., $\kappa(a_3) = \kappa(a_4) = 1$. The MRB's read and write indices after various firings of the connected actors a_1, a_3, and a_4 are depicted in Figure 3. Assuming the MRB is initially empty, these read and write indices have values as shown in Figure 3a. Thus, $\mathsf{T}(c_{\{1,2,3\}}, a_3) = \mathsf{T}(c_{\{1,2,3\}}, a_4) = 0$ and $\mathsf{F}(c_{\{1,2,3\}}) = \gamma(c_{\{1,2,3\}}) - \max\{0, 0\} = 4$.


$\omega(c_{\{1,2,3\}}) = 0$
$\rho_1(c_{\{1,2,3\}}) = -1$
a_3
a_1
writer
a_4
$\rho_2(c_{\{1,2,3\}}) = -1$
readers
$\omega(c_{\{1,2,3\}}) = 3$
$\rho_1(c_{\{1,2,3\}}) = 0$
a_3
a_1
writer
a_4
$\rho_2(c_{\{1,2,3\}}) = 0$
readers
$\omega(c_{\{1,2,3\}}) = 0$
$\rho_1(c_{\{1,2,3\}}) = 3$
a_3
a_1
writer
a_4
$\rho_2(c_{\{1,2,3\}}) = 0$
readers
$\omega(c_{\{1,2,3\}}) = 0$
$\rho_1(c_{\{1,2,3\}}) = -1$
a_3
a_1
writer
a_4
$\rho_2(c_{\{1,2,3\}}) = 1$
readers

(a) Initial state of the MRB.

(b) After firing $\langle a_1, a_1, a_1 \rangle$.

(c) MRB after $\langle a_3, a_3, a_3, a_1 \rangle$.

(d) MRB after $\langle a_4, a_3 \rangle$.

Figure 3 MRB with one write index (pointer) indicating the location of the next token to be written. Moreover, each reading actor requires an index pointing to the position of the next token to read.

At this point (see Figure 3a), it is only possible to perform write operations. Before firing a_1, we must check if sufficient free places are available for the produced tokens, i.e., $\mathsf{F}(c_{\{1,2,3\}}) = 4 \geq \psi(a_1) = 1$. Next, assume actor a_1 fires three times resulting in the state shown in Figure 3b. There, the write index $\omega(c_{\{1,2,3\}})$ has advanced to 3 pointing to the next free place in the MRB's buffer. The read indices $\rho_1(c_{\{1,2,3\}})$ and $\rho_2(c_{\{1,2,3\}})$ have been updated during the first firing of actor a_1 from -1 to 0 pointing to the first token contained in the MRB.

At this point (see Figure 3b), we can also perform read operations. Before firing a reader $a_{\mathbf{r}_i}$, we need to verify if there exist sufficient tokens to be consumed by the reader, i.e., $\mathsf{T}(c_{\{1,2,3\}}, a_{\mathbf{r}_i}) \geq \kappa(a_{\mathbf{r}_i})$. For instance, we are able to fire actor a_3 because $\mathsf{T}(c_{\{1,2,3\}}, a_3) = ((3-0-1) \bmod 4)+1 = 3 \geq 1$. After firing the sequence $\langle a_3, a_3, a_3, a_1 \rangle$, the resulting state is shown in Figure 3c. There, the readers track different information about the state of the MRB. The reader a_3 points to $\rho_1(c_{\{1,2,3\}}) = 3$ and observes $\mathsf{T}(c_{\{1,2,3\}}, a_3) = ((0-3-1) \bmod 4) + 1 = 1$ token on the MRB whereas, reader a_4 points to $\rho_2(c_{\{1,2,3\}}) = 0$ and observes $\mathsf{T}(c_{\{1,2,3\}}, a_4) = ((0-0-1) \bmod 4) + 1 = 4$ tokens. From the perspective of the writer a_1, the MRB is full.

At this point (see Figure 3c), let the firing sequence $\langle a_4, a_3 \rangle$ be observed. The resulting state of the MRB is shown in Figure 3d. From the perspective of a_3, the MRB is empty, i.e., $\rho_1(c_{\{1,2,3\}})$ is -1. The token placed at position 0 has been consumed because a_4 has read it now seeing $\mathsf{T}(c_{\{1,2,3\}}, a_4) = ((0-1-1) \bmod 4) + 1 = 3$ more tokens. From the perspective of a_1, there is one free place as $\mathsf{F}(c_{\{1,2,3\}}) = \gamma(c_{\{1,2,3\}}) - \max\{0, 3\} = 4 - 3 = 1$.

To evaluate the benefits of MRBs, the following section presents a DSE that decides whether to replace a multi-cast actor and its connected channels with an MRB.

Table 1 DFGs employed to compare the presented DSE.

Application	# of instances	$\|A\|$	$\|C\|$	$\|A_{\mathrm{M}}\|$	$\gamma(C)$	M_F (*Reference*) [MiB]
Sobel-4	1	27	33	4	1	82.6
Sobel-8	1	51	65	8	3	707.5
Multicamera	2	123	226	46	3	252.4
Optical flow	4	89	112	15	3	996.8
Object counting	5	96	120	15	2	105.6
Foreground detection	7	139	170	24	2	$5.01 \cdot 10^3$

5 Experimental Results

For design space exploration, we consider the set of objectives $\mathcal{F}^{\text{obj}} = \{f_1^{\max}, f_2^{\min}, f_3^{\min}\}$ to be optimized. Here, $f_1^{\max}$, $f_2^{\min}$, and $f_3^{\min}$ correspond to throughput, the number of allocated cores, and memory footprint, respectively. Our proposed DSE optimizes the mappings where each feasible implementation must satisfy the constraints presented in Equations (5)–(7). For each explored mapping, a schedule of actors and communications is determined using a First Come First Serve (FCFS) scheduler. The throughput is evaluated as the average Frames per Second (FPS) of running ten iterations of the obtained schedule. Moreover, the memory footprint is determined as presented in Section 3.

We employed the OpenDSE [14] framework for DSE using the NGSA-II elitist genetic algorithm [3] with a population size of 100 individuals, each generation generating 25 new individuals, and the crossover rate being set to 0.95. As a target architecture, we used a symmetric eight-core MPSoC, which connects each processor to a communication bus that, in its turn, is connected to a global memory. As target applications, Table 1 presents a benchmark composed of six real-world image processing applications obtained from self-developed Matlab/Simulink test cases [12]. As can be seen, for some applications, not only one, but even up to seven instances of application graphs were considered to run simultaneously. Shown in the table are also the number of actors, the number and capacity of channels, and the number of multi-cast actors contained in each application.

To quantify the effects of MRBs, we implemented three approaches *Reference*, $\mathit{MRB}_{\text{Always}}$, and $\mathit{MRB}_{\text{Explore}}$. Here, *Reference* only performs the optimization of mappings. The last column in Table 1 presents the memory footprint of each application obtained by the *Reference* approach, which serves as a baseline to compare the memory footprint reductions obtained by introducing MRBs in the applications. The approach $\mathit{MRB}_{\text{Always}}$ applies Algorithm 1 to all the multi-cast actors in the application as a pre-processing step and then performs the optimization of mappings, thus resulting in memory-efficient implementations. Finally, approach $\mathit{MRB}_{\text{Explore}}$, besides optimizing the mappings, also explores selectively for each multi-cast actor the choice of its replacement by a MRB.

To explore the placement of the actors and channels, we define an *integer genotype* for each actor and channel. For an actor a, we assume that it can be mapped to all cores R_{P} and, hence, the genotype for an actor is $\{1, 2, \ldots |R_{\mathrm{P}}|\}$. A given channel c can be mapped to (1) the global memory r_{DRAM}, (2) the producer core's scratchpad memory, or (3) the consumer core's scratchpad memory (as presented in Equation (7)), i.e., the genotype for a channel is $\{1, 2, 3\}$. Accordingly, the genotype of the *Reference* and $\mathit{MRB}_{\text{Always}}$ approaches are given by $\mathcal{G}_{\texttt{Always}} = \mathcal{G}_{\texttt{Reference}} = \{1, 2, \ldots |R_{\mathrm{P}}|\}^{|A|} \times \{1, 2, 3\}^{|C|}$.

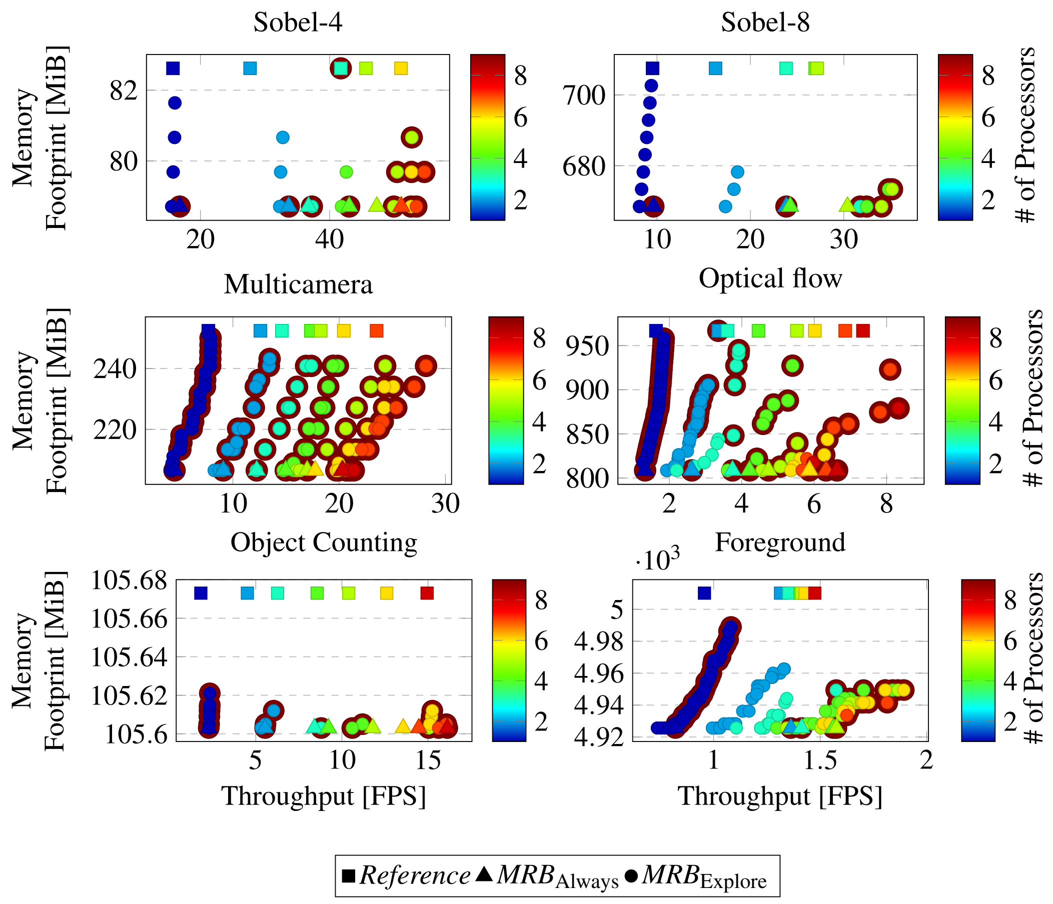


Figure 4 Pareto fronts of the last generation obtained for the six presented applications after 3,500 generations. Points circled brown are non-dominated points of the union of the three Pareto fronts.

The MRB_{Explore} approach uses a binary genotype $\{0, 1\}$ for each multi-cast actor to explore if the multi-cast actor and its connected channels are replaced by a MRB, where a 1 indicates a replacement. After replacing a given multi-cast actor $a_{\mathbf{m}}$ and its channels, the mapping of the newly introduced MRB is the same as the mapping of the channel being read by the replaced multi-cast actor $a_{\mathbf{m}}$. Thus, the genotype of the MRB_{Explore} approach is given by $\mathcal{G}_{\texttt{Explore}} = \mathcal{G}_{\texttt{Reference}} \times \{0, 1\}^{|A_{\mathrm{M}}|}$.

In the following, we will show that our MRB_{Explore} approach can find better quality solutions than the *Reference* and MRB_{Always} approaches. For the three approaches under investigation, we performed ten independent DSE runs for each considered application. Each exploration ran 3,500 generations, recording those implementation candidates with optimal throughput at each point during exploration.

5.1 Comparison of Exploration Results

Due to multiple objectives to optimize, there not exists a single optimal solution due to the conflicting set of objectives. Figure 4 presents the Pareto fronts[2] of the last explored generation obtained for the six test applications for each of the investigated approaches. The

[2] Shown are the efficient (non-dominated) sets of solutions of the last generation as found during each explorative search. Note that these sets are approximations of a true Pareto front.

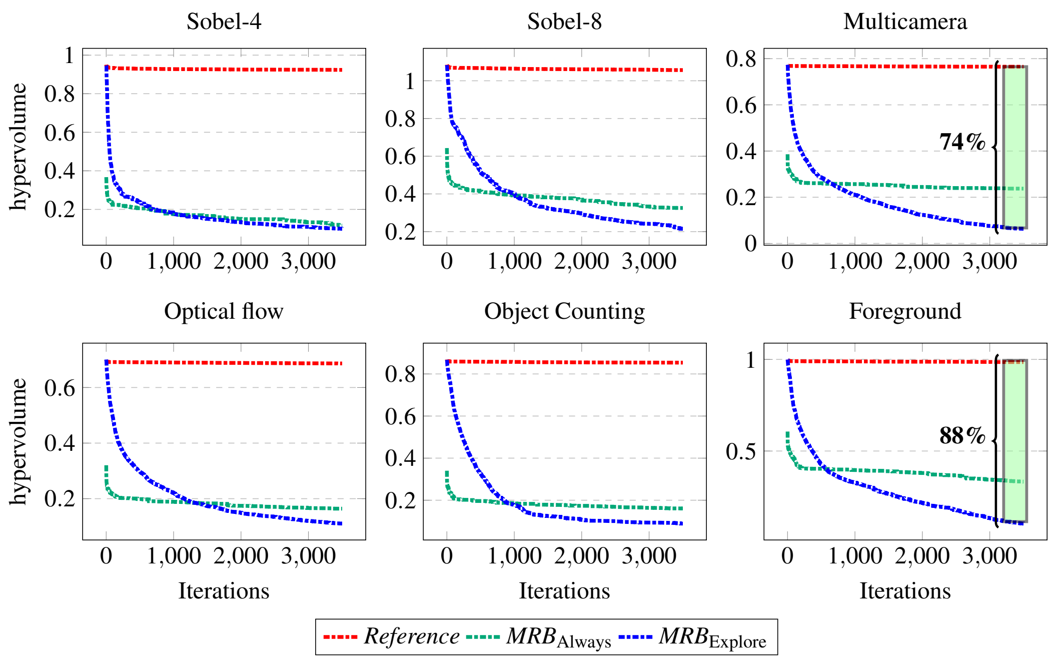


Figure 5 Hypervolume scores obtained for the six applications.

color on each mark represents the number of allocated cores in each solution, ranging from single-core in blue to eight-core implementations in red. From the Pareto fronts, we observe that *Reference* delivers least efficient implementations in terms of memory footprint (see squares on top), only trading throughput for allocated cores. In contrast, the MRB_{Always} front consists only of the least memory consuming implementations (see triangles at the bottom). When only looking at the Pareto fronts of *Reference* and MRB_{Always}, we cannot observe any clear tendency indicating that replacing all the multi-cast actors with MRBs leads to higher throughput implementations. For instance, we can observe that *Reference* found better throughput solutions for the optical flow and multicamera, whereas MRB_{Always} found better throughput solutions for the other applications.

Now, by selectively exploring the replacement of multi-cast actors according to our proposed approach MRB_{Explore}, we observe that even higher throughput solutions could be found by trading the memory footprint in contrast to *Reference* and MRB_{Always}. For applications where *Reference* obtains higher throughputs than MRB_{Always}, e.g., optical flow and multicamera, our approach trades higher memory footprint to find even higher throughput solutions by performing fewer replacements of multi-cast actors. Conversely, for those applications where MRB_{Always} finds higher throughput solutions than *Reference*, our approach also finds solutions applying more replacements of multi-cast actors reflected in solutions with less memory footprint. Notably, the highest throughput solutions found by our approach MRB_{Explore} are up to 22 % and 14 % higher in average over the considered applications compared to the highest throughput solutions of *Reference*. In order to be able to compare the quality of the obtained Pareto fronts, we use the *hypervolume* indicator [9] which delivers a single indicator measuring the performance quality of each approach. For this purpose, we utilize the multi-objective metric Hypervolume [9], which delivers a single indicator measuring the performance quality of a given approach. Figure 5 presents the hypervolume indicator for each explored application. Each plot shows the average hypervolume of ten runs of each of the three approaches under observation over 3,500 generations. A value closer to

0 indicates a better quality of solutions. As can be seen, a significantly better (on average 67 %) hypervolume indicator value can be observed at the exploration end of the MRB_{Always} approach compared to *Reference*. MRB_{Explore} improves the hypervolume indicator even 78 % on average over all the investigated applications.

6 Conclusions

This paper introduced the concept of Multi-Reader Buffers (MRBs) as a memory-efficient implementation of multi-cast actors and their replacement as a graph transformation. Rather then replicating produced tokens for all readers, an MRB stores only one copy of data for all readers. Data is alive as long as the last reader has consumed it. MRBs provide minimal buffer implementations that are obtained by replacing all multi-cast actors in an application with MRBs. But as the replacement of a multi-cast actor by a MRB may affect the overall throughput of the application, i.e., in case of small buffer sizes, we proposed a DSE approach to explore the space of selective MRB replacements. It was shown that solutions can be found with up to 22 % higher throughput compared to a reference approach. On average, the highest throughput implementations on the Pareto front were 14 % higher over the considered applications and 78 % in solution quality measured by a hypervolume indicator are reported.

References

1 Mohamed Benazouz, Olivier Marchetti, Alix Munier-Kordon, and Pascal Urard. A new approach for minimizing buffer capacities with throughput constraint for embedded system design. In *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)*, pages 1–8, 2010. doi:10.1109/AICCSA.2010.5586972.

2 Tobias Blickle, Jürgen Teich, and Lothar Thiele. System-level synthesis using evolutionary algorithms. *Design Automation for Embedded Systems*, 3(1):23–58, 1998.

3 K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *Trans. Evol. Comp*, 6(2):182–197, April 2002.

4 Jack Dennis. First Version of a Data Flow Procedure Language. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, Berlin, Heidelberg, 1974.

5 Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi. On Memory Reuse Between Inputs and Outputs of Dataflow Actors. *ACM Transactions on Embedded Computing Systems*, 15(2), February 2016. doi:10.1145/2871744.

6 Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi. Memory analysis and optimized allocation of dataflow applications on shared-memory mpsocs. *Journal of Signal Processing Systems*, 80(1):19–37, 2015.

7 J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. Bhattacharyya. A Generalized Static Data Flow Clustering Algorithm for MPSoC Scheduling of Multimedia Applications. In *Proceedings of ACM International Conference on Embedded Software*, pages 189–198, October 2008.

8 Joachim Falk, Christian Haubelt, and Jürgen Teich. Efficient representation and simulation of model-based designs in SystemC. In *Proceedings of the Forum on Specification and Design Languages*, volume 6, 2006.

9 Andreia P. Guerreiro, Carlos M. Fonseca, and Luís Paquete. The hypervolume indicator: Computational problems and algorithms. *ACM Comput. Surv.*, 54(6), July 2021. doi:10.1145/3453474.

10 J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith. SYSTEMCODESIGNER - an Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications. *ACM Trans. on Design Automation of Electronic Systems*, 14(1):1:1–1:23, January 2009.

11 Martin Letras, Joachim Falk, Tobias Schwarzer, and Jürgen Teich. Multi-objective Optimization of Mapping Dataflow Applications to MPSoCs Using a Hybrid Evaluation Combining Analytic Models and Measurements. *ACM Trans. on Design Automation of Electronic Systems*, 26:1–33, 2020. doi:10.1145/3431814.

12 Martin Letras, Joachim Falk, Stefan Wildermann, and Jürgen Teich. Automatic Conversion of Simulink Models to SysteMoC Actor Networks. In *Proc. of SCOPES*, pages 81–84, New York, NY, USA, 2017. ACM. doi:10.1145/3078659.3078668.

13 Amith R. Mamidala, Daniel Faraj, Sameer Kumar, Douglas Miller, Michael Blocksome, Thomas Gooding, Philip Heidelberger, and Gabor Dozsa. Optimizing mpi collectives using efficient intra-node communication techniques over the blue gene/p supercomputer. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 771–780, 2011. doi:10.1109/IPDPS.2011.220.

14 OpenDSE. “Open Design Space Exploration Framework”, 2018. URL: http://opendse.sf.net/.

15 S. Stuijk, M. Geilen, and T. Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of Design Automation Conference (DAC)*, pages 899–904, 2006. doi:10.1145/1146909.1147138.

16 J. Teich. Hardware/Software Codesign: The Past, the Present, and Predicting the Future. *Proc. IEEE*, 100(Special Centennial Issue):1411–1430, May 2012.

17 L. Thiele, K. Strehl, D. Ziegengein, R. Ernst, and J. Teich. Funstate-an internal design representation for codesign. In *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*, pages 558–565, 1999. doi:10.1109/ICCAD.1999.810711.

18 Hervé Yviquel, Alexandre Sanchez, Pekka Jääskeläinen, Jarmo Takala, Mickaël Raulet, and Emmanuel Casseau. Embedded multi-core systems dedicated to dynamic dataflow programs. *Journal of Signal Processing Systems*, 80(1):121–136, 2015.

RAVEN: Reinforcement Learning for Generating Verifiable Run-Time Requirement Enforcers for MPSoCs

Khalil Esper[1]
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Jan Spieck[1]
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Pierre-Louis Sixdenier
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Stefan Wildermann
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Jürgen Teich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

Abstract

In embedded systems, applications frequently have to meet non-functional requirements regarding, e.g., real-time or energy consumption constraints, when executing on a given MPSoC target platform.

Feedback-based controllers have been proposed that react to transient environmental factors by adapting the DVFS settings or degree of parallelism following some predefined control strategy. However, it is, in general, not possible to give formal guarantees for the obtained controllers to satisfy a given set of non-functional requirements. Run-time requirement enforcement has emerged as a field of research for the enforcement of non-functional requirements at run-time, allowing to define and formally verify properties on respective control strategies specified by automata. However, techniques for the automatic generation of such controllers have not yet been established.

In this paper, we propose a technique using reinforcement learning to automatically generate verifiable feedback-based enforcers. For that, we train a control policy based on a representative input sequence at design time. The learned control strategy is then transformed into a verifiable enforcement automaton which constitutes our run-time control model that can handle unseen input data. As a case study, we apply the approach to generate controllers that are able to increase the probability of satisfying a given set of requirement verification goals compared to multiple state-of-the-art approaches, as can be verified by model checkers.

2012 ACM Subject Classification Computer systems organization → Multicore architectures; Theory of computation → Linear logic; Theory of computation → Modal and temporal logics; Hardware → Finite state machines; Computer systems organization → Self-organizing autonomic computing; Theory of computation → Verification by model checking; Mathematics of computing → Probabilistic representations; Computing methodologies → Reinforcement learning

Keywords and phrases Verification, Runtime Requirement Enforcement, Reinforcement Learning

Digital Object Identifier 10.4230/OASIcs.NG-RES.2023.7

Funding This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research-Foundation) – Project Number 146371743 - TRR 89 Invasive Computing.

[1] Khalil Esper and Jan Spieck both contributed equally to this work.

1 Introduction

Current multi-processor on-chip (MPSoC) platforms offer abundant computational and storage resources that necessitate new programming paradigms such as invasive computing [39, 1] for isolating applications to handle architectural interferences between applications. Hybrid mapping approaches [35, 33, 29] have emerged for mapping of applications onto multi-core systems in the presence of uncertainty [34, 36].

Despite inter-application resource isolation schemes, applications are still exposed to variances in the system state (e.g., due to scheduler or caching effects) [6]. According to [38], another source of uncertainty is the varying workload induced by the input data (e.g., different workloads for different image inputs). As an example, throughput jitter in virtual and augmented reality applications may not only be an annoying user experience, but even cause dizziness or a headache to a user.

Run-time Requirement Enforcement (RRE) [40] is a field of research with the aim to control the non-functional properties of execution of a program within desired bounds. Such techniques dynamically steer control knobs, e.g., voltage/frequency settings, in reaction to observed changes in the system state to keep the non-functional properties of execution within the desired range. RRE allows a user to specify bounds on execution properties of an application on a multi-core platform using so-called *requirements* [38], i.e., expressions on non-functional properties such as desired corridors on latency or energy consumption. Recently, techniques have been proposed in [9, 10, 11] to formally verify the satisfaction or violation of non-functional requirements of RRE techniques at design time. In order to apply formal methods such as model checking, *finite state machines (FSMs)* are used to formally specify control strategies. However, techniques for automatically generating FSMs for RRE that either always guarantee the satisfaction of a set of given non-functional requirements in case of *strict enforcement* or at least guarantee a certain probability of satisfying executions in case of *loose enforcement* [40] have not yet been established.

In this realm, different machine-learning-based techniques have been proposed for the dynamic control of program executions [25, 26]. In an offline phase, a controller behaviour is learned to optimize a set of given non-functional objectives that can be used at run-time to control the application. However, the above approach cannot provide any formal guarantees regarding the ability or strictness to fulfill a set of given requirements, i.e., constraints on non-functional execution properties. In this paper, we propose a technique using reinforcement learning to generate FSMs for RRE with formally verifiable guarantees by training and optimizing an FSM controller to be generated based on input sequences at design time. Based on a formal characterization of input variation at run time, the generated FSM controllers for RRE can then also be formally verified at design time.

Contributions: The main contributions of this paper can be summarized as follows:

1. Using reinforcement learning for FSM-based RRE generation: Based on training sequences, reinforcement learning is used to adapt an initial FSM model towards satisfaction (or improvement of the satisfaction probability) of a given set of non-functional program execution properties formulated as verification goals.
2. During learning, the RRE strategy is regularly transformed into an FSM and formally verified according to the set of verification goals. The offline learning phase stops once all goals are satisfied. Alternatively, based on a user-defined exit condition.
3. In a case study, the approach to generate formally verified FSM-based RRE controllers is compared to state-of-the-art enforcer designs.

2 Fundamentals

In the following, important notions and definitions are introduced.

2.1 FSM-based RRE

Non-functional requirements should be satisfied during each program execution on a given MPSoC platform even when the environmental input is varied. According to [10], let the size of the input of a given program for each discrete execution k be given by an *environment feature vector* $i(k) \in \mathcal{I}$, where $\mathcal{I}$ is called the *environment space*. Moreover, assume that for preventing or as a countermeasure against violations of a set of non-functional requirements, an enforcer can vary the number n of cores allocated to execute an application program as well as the voltage/frequency setting m of these cores. We call such a setting (n, m) a configuration c and the set of available configurations available on a given MPSoC platform the *configuration space* C. Figure 1 shows the concept of feedback-based RRE according to [10] which serves as the base model also in this paper. Illustrated is a multi-core system stimulated by input from an environment and reacting to violation of a number of requirement using an enforcement FSM that determines the configuration $c(k+1)$ for the $(k+1)$th execution accordingly.

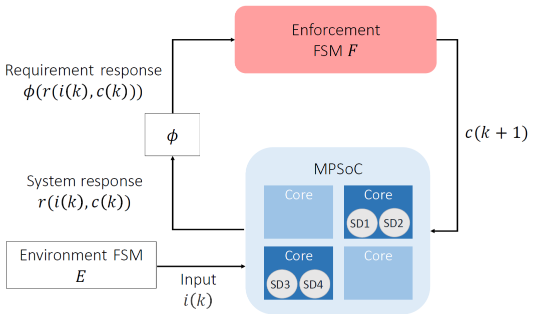


Figure 1 Illustration of feedback-based RRE. A system response vector r is mapped to a binary requirement response vector ϕ such that the enforcement FSM F controls the next configuration $c(k+1) \in C$. Adapted from [10].

2.1.1 Formal Definitions

Assume that the k-th execution of a program on an MPSoC yields H execution properties of interest (e.g., latency and energy consumption). These properties depend on the input data $i(k) \in \mathcal{I}$ and the system configuration $c(k) \in C$. For the purpose of RRE, the *system-under-control* can be abstracted by a single function called *system response function* $r : \mathcal{I} \times C \rightarrow \mathbb{R}^H$ (see [10]). Thus, the system response $r(i(k), c(k)) = (o_1(k), \ldots, o_H(k))$ at execution k is a vector of the H relevant execution properties (see Figure 1). According to [41], requirements can be specified for each property o_h, $h \in \{1, \ldots, H\}$, typically in terms of a lower bound LB_{o_h} and an upper bound UB_{o_h} that should not be violated. Such intervals can be described by two propositions φ_h^{LB} and φ_h^{UB} as follows:

$$\varphi_h^{LB}(o_h(k)) = (LB_{o_h} \leq o_h(k)) \tag{1}$$
$$\varphi_h^{UB}(o_h(k)) = (o_h(k) \leq UB_{o_h}) \tag{2}$$

In Equation (1) and Equation (2), LB_{o_h} and UB_{o_h} denote a user-given lower, respectively, upper bound on the execution property o_h. The information about which proposition is fulfilled and which is violated at the k-th execution can then be described by a binary vector denoted by *requirement response* ϕ (see Figure 1). It is obtained from the system response $r(i(k), c(k)) = (o_1(k), \ldots, o_H(k))$ using the *requirement response function* [10]:

$$\beta := \phi\,(o_1(k), \ldots, o_H(k)) = \big(\varphi^{LB}(o_1(k)), \varphi^{UB}(o_1(k)), \ldots, \\ \varphi^{LB}(o_H(k)), \varphi^{UB}(o_H(k))\big) \in \{0,1\}^{2H}. \tag{3}$$

This binary requirement response vector β specifies for each proposition to be satisfied for each execution k the input to the enforcement finite state machine (FSM) F, as illustrated in Figure 1. F reacts by computing the next configuration $c(k+1) \in C$ to enforce the desired non-functional properties for the next execution $k+1$. Formally, an enforcement FSM is defined as follows.

▶ **Definition 1** ([10]). *An* enforcement FSM (F) *is a deterministic finite state machine (Moore machine) that can be described by a 6-tuple $(Z, z_0, B, \delta, C, \gamma)$:*
- *Z is a finite set of states.*
- *$z_0 \in Z$ is the initial state.*
- *B is the input alphabet.*
- *δ is the transition relation: $\delta \subseteq B \times Z \times Z$ with $(\beta, z, z') \in \delta$ representing a transition from z to z' under input β.*
- *C is the output alphabet, also called configuration space.*
- *γ is the output function, which maps each state to the output alphabet: $\gamma : Z \to C$.*

Instead of verifying an enforcement strategy described by an enforcement automaton F for RRE just for individual input traces, the authors in [10] proposed rather to analyze families of traces. The input variation of the environment is modeled by a discrete-time Markov Chain called *environment FSM*, after partitioning the environment space of inputs $\mathcal{I}$ into a set P of disjoint partitions $p \in P$ with $p \subseteq \mathcal{I}$. The partitions are constructed such that all inputs $i \in p$ assigned to the same partition always deliver the same binary requirement response $\phi(r(i(k), c(k)))$ in each configuration $c \in C$. These partitions p then define a discrete state space of a discrete-time Markov chain E. Transitions between states reflect the probabilities of observable variations in environmental input from state to state. The environment FSM E can equally be seen as a generator of potential input traces that an RRE FSM F shall be evaluated for. But rather than evaluating a single or comparing multiple enforcer FSMs based on just individual sample traces, we want to argue first about quality of enforcers rather for all input traces that a system can potentially undergo. Second, rather than simulating such input traces to generate statistics, we propose to apply symbolic techniques, i.e., probabilistic model checking for our analysis.

2.1.2 Verification Goals

Verification goals (VGs) can then be specified to compare different enforcement strategies regarding their quality to satisfy the given set of requirements. VGs are formulated over the two propositions φ_h^{LB} and φ_h^{UB}, see Equation (1) and Equation (2), using temporal logic [5] or PCTL [2, 17]. Examples of such verification goals of interest (one is applied in case of strict enforcement, the subsequent ones for loose enforcement) are [10]:
- $AG(\varphi)$: φ should always hold.
- $AF(\varphi)$: φ should eventually hold.

- $\mathcal{P}_{=?}[\neg\varphi \rightarrow F^{\leq\lambda}(\varphi)]$ denoting the probability of returning to a requirement-satisfying configuration state (φ) from a violating one ($\neg\varphi$) in no more than λ steps, i.e., next executions.
- $\mathcal{P}_{=?}[G^{\leq\lambda}\ (\neg\varphi_L)]$ denoting the probability of λ consecutive violations of φ.
- $\mathcal{S}_{=?}[\neg\varphi]$ denotes the steady-state probability of violating φ.

2.2 Reinforcement Learning

Reinforcement Learning (RL) [37] is a Machine Learning paradigm dealing with how an *agent* shall act in an environment in order to maximize a cumulative reward. An agent is supposed to improve its ability to solve a problem (defined via a reward function) through trials-and-errors, similar to how humans and animals learn. At its core, RL models a problem as a *Markov Decision Process (MDP)*, representing the environment, which the agent interacts with and observes. At each time step, the environment resides in a state $\upsilon \in \Upsilon$, based on which the agent then selects an action $a \in A$ according to its internal policy π that will put the state in a successor state $\upsilon' \in \Upsilon$. The resulting sequence of states and actions performed in the environment $\tau = (\upsilon_0, a_0, \upsilon_1, a_1, ...)$ is called a *trajectory* τ. Some key components when performing RL training are:

1. A *policy* $\pi : \Upsilon \times A \rightarrow [0,1]$ that defines the behavior of an agent, i.e., the probability to take each available action $a \in A$ for each state $\upsilon \in \Upsilon$. A policy π can have parameters θ (then denoted as π_θ), be stochastic (e.g., the ϵ-greedy policy in Equation (4)) or deterministic.
2. A *reward signal* $\xi : \Upsilon \times A \rightarrow \mathbb{R}$ is the feedback sent by the environment to the agent when it takes an action a in a state υ, ergo it represents the immediate goal of the agent.
3. An *action-value function* $Q^\pi : \Upsilon \times A \rightarrow \mathbb{R}$, which predicts the cumulated reward that could be obtained on the long run if the agent takes decision a in current state υ when following policy π.
4. In some cases, a *model* of the environment that allows for predicting rewards $\xi(\upsilon, a)$.

The most investigated way of solving a RL problem is to accurately estimate the action-value function. While doing so, an agent has to explore the state space Υ to receive the rewards. To that regards, there is a trade-off to be made between *exploration*, i.e., taking a random action at a given state υ, and *exploitation*, i.e., performing the action a that will maximize the expected cumulative reward $Q^\pi(\upsilon, a) = \mathbb{E}_{\tau\sim\pi}[\sum_{(\upsilon_t,a_t)\in\tau} \xi(\upsilon_t, a_t) \mid \upsilon_0 = \upsilon, a_0 = a]$ following the trajectory τ sampled from the policy π starting from state $\upsilon_0 = \upsilon$ and action $a_0 = a$. Too much exploration will increase the training time (as it is not different from a random selection of actions), whereas too much exploitation might lead to convergence to a local optimum. As an example, a common policy used (notably in Q-Learning) to balance exploration and exploitation is called *ϵ-greedy policy* and is presented in Equation (4). Here, $a \in_R A$ describes the sampling of a random action from A based on the probability distribution R.

$$a = \begin{cases} \arg\max_{a\in A} Q(\upsilon, a) & \text{with probability } (1-\epsilon) \\ a \in_R A & \text{with probability } \epsilon \end{cases} \tag{4}$$

In this paper, we will consider as an example a widely used algorithm, *Q-Learning*. Q-Learning [43] is a model-free algorithm which learns the action-value function, i.e., the Q-function. A common implementation is to have a *Q-table* storing all the values of Q. The algorithm steps are described in Algorithm 1 (in the appendix). An episode *ep* comprises multiple iterations, in which actions are picked and taken until υ is a terminal state or a given number of maximum iterations is performed. Observing the reaction of the environment yields the successor state υ' and reward $\xi(\upsilon, a)$. The Q-table is then updated according to Equation (5).

$$Q(\upsilon, a) \leftarrow Q(\upsilon, a) + \alpha \cdot \left(\xi(\upsilon, a) + \kappa \cdot \max_{a} Q(\upsilon', a) - Q(\upsilon, a)\right) \tag{5}$$

The *learning rate* $\alpha \in [0, 1]$ determines how much of the newly acquired information should replace the current knowledge. The *discount factor* $\kappa \in [0, 1]$ represents how much influence future rewards have on the current optimization step compared to the instant one. For example, a value of 0 will make the agent "short-sighted", while a value of 1 will make it aim for an endgame goal.

One limitation of Q-learning is its discrete nature, which, when used to solve continuous problems, either leads to a state explosion or suboptimal performance due to a down-sampling of both actions and states. One way to solve continuous problems using this algorithm is to use deep artificial neural networks as approximators for the Q-function. This is referred to as Deep Q-Learning [14].

3 Reinforcement Learning for the Generation of Run-Time Requirement Enforcers

The structure of our verifiable RRE generation approach and optimization is depicted in Fig. 2. During a training phase (see upper part of the figure), an enforcement agent learns an enforcement strategy aiming to satisfy a set of verification goals defined over a set of given requirements like latency, energy, or power. The training is based on an input data sequence $I = \{i(1), i(2), ...\}$ generated by a Markov chain. To assess the satisfaction of the verification goals, the policy of the enforcement agent is periodically transformed (i.e., after a specified number of training iterations n_{update}) into an enforcement FSM and the verification goals VG are subsequently formally verified using a model checker (see lower part of Fig. 2). Since verifying the enforcement goals can be a time-intensive task, we employ surrogate functions in executions k between such verification checks to estimate the verification goals VG between the actual model checks to speed up the training process. The training phase stops either when a run-time requirement enforcer with verified VGs has been found, which then can be deployed in the field (*run-time phase*). Else, a user can determine the termination of the learning phase.

3.1 Learning phase

Our enforcement agent is defined by a set of states Υ, a set of actions A, and finally a reward function ξ that assesses the quality of choosing an action in the current state (see Section 2.2). An action $a(k)$ of our agent is determined by the selection of the configuration $c(k + 1) = (n, m)$ of number of cores n and power mode m to be applied for processing the next input data $i(k + 1)$. The set of actions A is thereby defined equal to the configuration space C, i.e., $A = C$. A state $\upsilon \in \Upsilon = B \times C$ is given by a pair of configuration $c \in C$ and corresponding requirement response $\beta \in B$ that indicates which requirements were fulfilled after having processed a given input data i in configuration c and which not. As a consequence, the number of states is given by: $|\Upsilon| = |B| \cdot |C|$.

Since we do not require any functional execution properties of the related programs for training the enforcement agent, our approach can also handle black-box applications. The feedback about the satisfaction of a set of verification goals is encapsulated into a reward

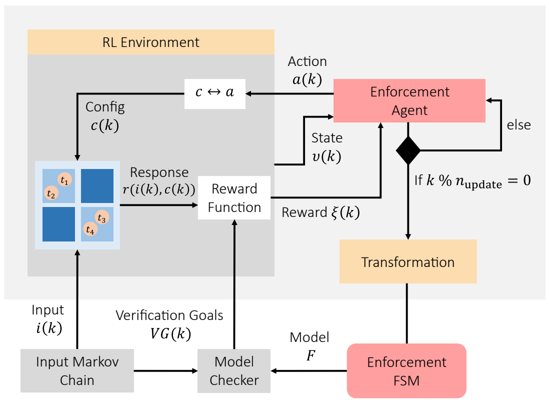


Figure 2 Enforcer generation based on Reinforcement Learning.

function ξ_η that associates a chosen action $a \in A$ with a quality assessment in the form of a numeric reward. In the following Equation 6, the reward is defined as a weighted sum of a *verified reward* ξ_{ver} – measured by transforming the enforcement agent into an enforcement FSM and employing a model checker – and a *surrogate reward* ξ_{sur} that provides a probabilistic estimation of the probabilities of fulfilling the verification goals based on the history of an already processed input data sequence. Consequently:

$$\xi_\eta(a(k)) = \eta \cdot \xi_{\text{sur}}(k) + (1-\eta) \cdot \xi_{\text{ver}}(k). \tag{6}$$

With an increasing number of training iterations, the influence of the model checked reward ξ_{ver} should increase, as the surrogate reward merely serves as an estimate. This can be implemented by decaying η with increasing episode numbers ep, e.g., exponentially by $\eta = \eta_0 \cdot e^{-ep \cdot dec}$, with $\eta_0 \in [0,1]$ being the initial value and $dec \in \mathbb{R}$ a decay hyperparameter.

Moreover, we define the verified reward ξ_{ver} as the weighted sum of given requirement verification goals VG of our model at iteration k and obtained by applying probabilistic model checking to an enforcer FSM obtained by a model transformation described in Section 3.2:

$$\xi_{\text{ver}}(k) = \sum_{\omega=1}^{|VG|} \varsigma_\omega \cdot VG_\omega(k). \tag{7}$$

In the case of a verification goal for strict enforcement, VG_ω represents a binary value indicating whether the verification goal was met $VG_\omega = 1$ or missed $VG_\omega = 0$, while in case of loose enforcement, $VG_\omega \in [0,1]$ denotes a probability of meeting the verification goal. Furthermore, the weight ς_ω should be chosen negative when the associated verification goal VG_ω shall be minimized and positive when VG_ω shall be maximized.

In contrast to the verified reward ξ_{ver} that is updated periodically every n_{update} iterations, the surrogate reward ξ_{sur} is computed in each iteration k due to being an derivative of the function f_{est} at point k estimating the verification goals based on the history $\mathfrak{H} = (1, ..., k)$ of the input data i and agent trajectory τ up to the current action $a(k)$:

$$f_{\text{est}}(k) = \vartheta(k) \cdot \sum_{\omega=1}^{|VG|} \varsigma_\omega \cdot \mathbb{E}[VG_\omega \mid (i(y))_{y \in \mathfrak{H}}, (v_y, a_y)_{y \in \mathfrak{H}} \in \tau]. \tag{8}$$

With that, the surrogate reward $\xi_{\text{sur}}(k)$ is defined as:

$$\xi_{\text{sur}}(k) = \Delta f_{\text{est}}(k) = f_{\text{est}}(k) - f_{\text{est}}(k-1). \tag{9}$$

Loose verification goals VG_ω can be estimated by using the empirical probability, i.e., associating the number of occurrences of verification goal violations over the history $\mathfrak{H}$. Since the accuracy of our probabilistic estimation increases with larger sizes of the regarded history $k = |\mathfrak{H}|$, we weight the surrogate function by a factor $\vartheta(k) \in [0, 1]$ that scales the reward with k:

$$\vartheta(k) = 1 - e^{-u \cdot k}, u \in \mathbb{R}^+. \tag{10}$$

Above function covers the range of $\vartheta(0) = 0$ up to $lim_{k\to\infty}\vartheta(k) = 1$ with $u \in \mathbb{R}^+$ steering the gradient of the scaling function. Strict verification goals are set to zero in case any violation happens in the history, and to one otherwise.

Example. Let the verification goal $VG_L := \mathcal{S}_{=?}[\varphi_L]$ be given, the goal being to minimize the probability of violating a latency requirement ϕ_L. This goal can be estimated as

$$\mathbb{E}[VG_L] = \frac{\sum_{y\in\mathfrak{H}} \xi_{\varphi_L}(k)}{|\mathfrak{H}|} \tag{11}$$

We can then define the reward for the requirement φ_L as:

$$\xi_{\varphi_L}(k) = \begin{cases} 1 & if\, \varphi^{UB}(o_L(i(k), a(k))) \wedge \varphi^{LB}(o_L(i(k), a(k))) \\ 0 & else \end{cases} \tag{12}$$

With this formalization of our reinforcement agent, we can choose a suitable reinforcement learning implementation from the literature for performing the training procedure. In case of a low cardinality of configurations $|C|$, simple approaches as Q-learning are viable. Else, more sophisticated model-free deep learning procedures as proximal policy optimization (PPO) [31] or soft actor-critic (SAC) [16] are recommended.

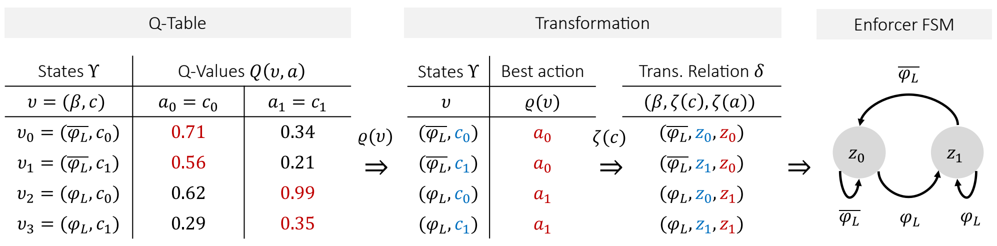
3.2 Transformation

This section describes how to transform a trained reinforcement learning agent into an enforcer FSM that can be formally verified. First, we need to transform our reinforcement learning agent states Υ into a set of enforcement FSM states Z. Second, we need to transform our agent policy into an FSM transition relation δ. Note that we can only transform reinforcement agents into a verifiable enforcer FSM for discrete action and state spaces.

We generate one unique enforcer FSM state $z_c \in Z$ of a Moore FSM per configuration $c \in C$, described by the bijective function $\zeta : C \leftrightarrow Z$. The FSM transition relation $\delta : B \times Z \to Z$ determines a next state from a current state based on the requirement response β. Since each state represents uniquely exactly one configuration, we can reformulate this relation as $\delta : B \times C \to C$, i.e., we have to determine for each configuration the best-suited subsequent configuration in dependence of the requirement response β. With the reinforcement states defined as $\Upsilon = B \times C$ and associated actions given by $A = C$, we propose to derive the state transitions by determining the best action per state (β, c) of our reinforcement learning agent. In particular, for each enforcer FSM state $z = \theta(c)$ corresponding to a configuration, we create one outgoing transition per reinforcement state $(\beta, c) \in \Gamma$ as follows:

$$\delta = \{(\beta, \zeta(c), \zeta(a)) \mid (\beta, c) \in \Upsilon \wedge a = \varrho(\beta, c))\}. \tag{13}$$

The best action per state $\varrho : \Upsilon \to A$ can be extracted from the trained agent policy.


Q-Table

States Υ	Q-Values $Q(\upsilon, a)$	
$\upsilon = (\beta, c)$	$a_0 = c_0$	$a_1 = c_1$
$\upsilon_0 = (\overline{\varphi_L}, c_0)$	0.71	0.34
$\upsilon_1 = (\overline{\varphi_L}, c_1)$	0.56	0.21
$\upsilon_2 = (\varphi_L, c_0)$	0.62	0.99
$\upsilon_3 = (\varphi_L, c_1)$	0.29	0.35

$\varrho(\upsilon) \Rightarrow$

Transformation

States Υ	Best action	Trans. Relation δ
υ	$\varrho(\upsilon)$	$(\beta, \zeta(c), \zeta(a))$
$(\overline{\varphi_L}, c_0)$	a_0	$(\overline{\varphi_L}, z_0, z_0)$
$(\overline{\varphi_L}, c_1)$	a_0	$(\overline{\varphi_L}, z_1, z_0)$
(φ_L, c_0)	a_1	(φ_L, z_0, z_1)
(φ_L, c_1)	a_1	(φ_L, z_1, z_1)

$\zeta(c) \Rightarrow$

$\Rightarrow$ Enforcer FSM

Figure 3 Example of transforming a Q-table that is based on the configuration set $C = \{c_0, c_1\}$ and the verification goal $VG_L := \mathcal{S}_{=?}[\varphi_L]$ for the latency requirement φ_L into an enforcer FSM.

Example. Let us give a simple example with two configurations $C = \{c_0, c_1\}$ and one verification goal $VG_L := \mathcal{S}_{=?}[\varphi_L]$ based on the latency requirement φ_L. As in the following Section 4 Evaluation, Q-learning is used as the reinforcement learning implementation. The Q-table of the enforcement agent contains one Q-value $Q(\upsilon, a)$ per tuple of enforcer agent state υ and action $a \in A = C$. The transformation from a Q-table into an enforcer FSM is illustrated in Fig. 3. In the first step, the transformation procedure associates each state υ with its best action a by applying ϱ. For Q-learning, the best action per state is directly given by the highest Q-value in the corresponding row, which therefore defines ϱ as:

$$\varrho(\upsilon) = \arg\max_{a \in A} \; Q(\upsilon, a). \tag{14}$$

With that, the transition relation δ can be determined, representing the two-step enforcer FSM depicted on the right side of the figure. Remember that each tuple $(\beta, z, z') \in \delta$ describes a transition of the enforcer FSM from state z into state z' triggered by the requirement response β. Finally, the resulting enforcer FSM can be formally verified for the given verification goals according to [10] using probabilistic model checking, e.g., PRISM [23].

4 Evaluation

In this section, we present an elaborate case study for the presented approach. The application considered is an object detection application whose actor graph is shown in Fig. 4 (in the appendix). The object detection application processes a stream of periodic input images in a pipelined fashion so that a given object in each image frame is detected based on scale-invariant feature transform (SIFT) matching [24]. As properties of execution to be enforced, we consider the latency o_L and the power consumption o_P. An empirical analysis of the execution times revealed that most of the execution time is spent in the SIFT description actor. Each SIFT description worker actor iterates over the list of features received from the control mechanism and generates corresponding feature descriptions, which is a compute-intensive task. For that reason, we apply RRE to just this actor using the method described in this paper. For the experiments, we used a sequence of $I_{train} = 1000$ images from the KITTI database [13] in each training episode. To counteract any violation of the corresponding requirements, a set of $m = 20$ power modes driven by dynamic voltage and frequency scaling (DVFS) can be applied, and a maximum of $n = 4$ cores can be allocated per run.

Each of the reinforcement learning-based enforcer instances was trained for a given set of verification goals until convergence (3,000 episodes, where each episode consists of iterating over the training input set I_{train}) using the reward function described in Eq. (6), the ϵ-greedy policy described in Eq. (4), parametrized with a learning rate of $\alpha = 0.1$ and a discount factor $\kappa = 0.99$. Due to the limited state space with $|C| = 80$, Q-learning is still viable. Note that deep learning-based agent implementations such as Soft-Actor-Critic provide similar results but introduce additional overhead during training. The result of the transformation process, explained in Section 3.2, is an FSM that can be verified using the verification method in [10].

For formal verification, an environment FSM is generated from I_{train}, using the environment FSM generation method in [10], based on a latency requirement $\varphi_L = \varphi_L^{LB} \wedge \varphi_L^{UB} = (LB_{o_L} \leq o_L) \wedge (o_L \leq UB_{o_L})$ for a latency lower bound $LB_{o_L} = 0$ ms and an upper bound (deadline) $UB_{o_L} = 40$ ms, similarly for a power requirement $\varphi_P = \varphi_P^{LB} \wedge \varphi_P^{UB} = (LB_{o_P} \leq o_P) \wedge (o_P \leq UB_{o_P})$ for a power lower bound $LB_{o_P} = 0\,\text{W}$ and an upper bound $UB_{o_P} = 1.2\,\text{W}$. Intuitively, $\varphi_L^{LB} = (0 \text{ ms} \leq o_L)$ and $\varphi_P^{LB} = (0\,\text{W} \leq o_P)$ are always satisfied and can therefore be ignored during enforcement in this case study. The generated enforcer FSMs are verified using the PRISM model checker [22]. In the following, we present the verification results for loose and strict enforcement for the proposed and the following set of other previously proposed RRE techniques: Race-to-idle (RTI) [21] that executes the application in each iteration k constantly with $n = 4$ cores and the highest power mode m, 1-step enforcement FSM F_1 proposed in [9], and 8-step enforcement FSM F_2 in [10].

4.1 Loose enforcement

As a first example, we specify and verify the following two verification goals: $P_{=?}[G^{\leq 3}\neg\varphi_L]$ for the latency requirement φ_L and $P_{=?}[G^{\leq 3}\neg\varphi_P]$ for the power requirement φ_P, see Section 2.1.2 for explanation.

Table 1 Verification results for loose enforcement for RTI, F_1, F_2, and F_{rl_0} for the verification goals $P_{=?}[G^{\leq 3}\neg\varphi_L]$ and $P_{=?}[G^{\leq 3}\neg\varphi_P]$, based on a latency upper bound (deadline) $UB_{o_L} = 40$ ms, and a power upper bound $UB_{o_P} = 1.2\,\text{W}$.

$P_{=?}[G^{\leq 3}\neg\varphi_L]$				$P_{=?}[G^{\leq 3}\neg\varphi_P]$			
RTI	F_1	F_2	F_{rl_0}	RTI	F_1	F_2	F_{rl_0}
0	0.427	0.041	0	1	0.256	0.389	0

As shown in Table 1, $P_{=?}[G^{\leq 3}\neg\varphi_L] = 0, P_{=?}[G^{\leq 3}\neg\varphi_P] = 0$ for the RL-generated FSM F_{rl_0}. This means that our approach can determine an enforcer FSM that does not violate the latency nor the power requirement for $\lambda = 3$ consecutive executions. $P_{=?}[G^{\leq 3}\neg\varphi_L] = 0$ also for RTI, as it always satisfies the latency requirement φ_L, and $P_{=?}[G^{\leq 3}\neg\varphi_L] = 1$ because it always violates the power requirement φ_P as it always runs in the highest power mode $m_{\max} = 20$ and number of cores $n_{\max} = 4$. Also note that $P_{=?}[G^{\leq 3}\neg\varphi_L]$ is higher for F_1 than for F_2 as F_2 increases its configuration state z by 8 steps when having a latency violation, whereas F_1 only increases it by 1. For the same reason, $P_{=?}[G^{\leq 3}\neg\varphi_P]$ for F_1 is lower than for F_2.

Finally, we also performed a verification for loose enforcement using the two alternative verification goals $S_{=?}[\neg\varphi_L]$ for the latency requirement φ_L and $S_{=?}[\neg\varphi_P]$ for the power requirement φ_P, see Section 2.1.2. Such steady-state probabilities give insight into the long-term behavior of running applications.

As shown in Table 2, the steady-state probabilities $S_{=?}[\neg\varphi_L]$ and $S_{=?}[\neg\varphi_P]$ for our RL-based FSM F_{rl_1} are lower than for F_1. Thus, our approach can generate an enforcer FSM that has lower steady-state probability than F_1 to violate the given requirements φ_L and φ_P. Although the steady-state probability of having a latency violation $S_{=?}[\neg\varphi_L]$ is lower for F_2 than F_{rl_1}, the steady-state probability of having a power violation $S_{=?}[\neg\varphi_P]$ for F_2 is higher than for F_{rl_1}. For RTI, as it always satisfies the latency requirement φ_L, it has also a stationary probability $S_{=?}[\neg\varphi_L] = 0$. But regarding the power requirement φ_P, $S_{=?}[\neg\varphi_L] = 1$

Table 2 Verification results for loose enforcement for RTI, F_1, F_2, and F_{rl_1} for the verification goals $S_{=?}[\neg\varphi_L]$ and $S_{=?}[\neg\varphi_P]$, based on a latency upper bound (deadline) $UB_{o_L} = 40$ ms, and a power upper bound $UB_{o_P} = 1.2\,\mathrm{W}$.

$S_{=?}[\neg\varphi_L]$				$S_{=?}[\neg\varphi_P]$			
RTI	F_1	F_2	F_{rl_1}	RTI	F_1	F_2	F_{rl_1}
0	0.5	0.121	0.173	1	0.445	0.591	0.435

for RTI as it always runs in the power requirement violating mode $(n_{\max}, m_{\max})$. $S_{=?}[\neg\varphi_L]$ for F_2 is lower than for F_1 as F_2 increases its configuration state z by 8 when having a latency violation, whereas F_1 only increases it by 1, so it has a higher chance to meet the latency requirement φ_L. For the same reason, $S_{=?}[\neg\varphi_P]$ for F_1 is lower than for F_2.

4.2 Strict enforcement

Finally, the following example is chosen to illustrate the approach also for strict enforcement. Table 3 shows the verification results for strict enforcement of the latency requirement φ_L using the verification goal $AG(\varphi_L)$, see Section 2.1.2. $AG(\varphi_L) =$ true for RTI, which means that φ_L always holds, as RTI always runs in the highest power mode $m_{\max} = 20$ and number of cores $n_{\max} = 4$. For F_1 and F_2, $AG(\varphi_L) =$ false because both FSMs decrease their configuration state z by one once satisfying φ_L. Finally, our RL-based approach can generate an enforcer FSM F_{rl_2} that also always satisfy the latency requirement φ_L ($AG(\varphi_L) =$ true).

Table 3 Verification results for strict enforcement for RTI, F_1, F_2, and F_{rl_2} for the verification goal $AG(\varphi_L)$, based on a latency upper bound (deadline) $UB_{o_L} = 40$ ms, and a power upper bound $UB_{o_P} = 1.2\,\mathrm{W}$.

RTI	F_1	F_2	F_{rl_2}
true	false	false	true

5 Related Work

Several approaches do exist to control non-functional properties of program executions, such as latency, or power and energy consumption. Examples of such approaches are techniques based on online machine learning like [25, 26], heuristics like [42], and predictive models [8, 32]. However, most of them cannot provide any formal guarantees about the controller's capability of satisfying the given requirements. Such guarantees include that the control technique will never lead to a violation of the given requirements or that the system will stay no more than a certain number of executions in a violating state, or long-term percentages of non-violating executions. Although techniques based on control theory, such as [19, 27, 28], can formally analyze controller properties such as stability, they are not able to provide any formal guarantees regarding the satisfaction or violation of given non-functional requirements in uncertain environments.

In general, FSMs are not only used to formally specify the functional behavior of a system [30, 4, 12], but also when formal verification of non-functional properties is required, especially in safety-critical systems. In [40], the concept of *Run-time Requirement Enforcement (RRE)* is introduced to describe techniques to either centrally or decentrally control the satisfaction of non-functional execution properties of programs executed on MPSoCs given by set of requirements. of programs for MPSoCs. Based on this concept, [10] proposes feedback-based RRE techniques. Presented is an approach for the formal specification and verification of non-functional properties for systems executing programs periodically, where an FSM-based enforcer is used to control the number of cores and DVFS level of a system once per execution at run-time. Using this approach, one can evaluate whether a combination of a system (MPSoC), an enforcer, and an environment either always satisfies the defined requirements or with which satisfaction probability. In [9], simple FSM control schemes are introduced that simply increase, resp. decrease the power mode or number of cores in case of a violation $\neg\varphi_L$, resp. satisfaction φ_L of the latency requirement. In [10, 11], FSMs for multi-requirement control have been introduced. However, the RRE controllers presented in these works are all manually designed. In summary, techniques for automatic generation of verifiable enforcers are still missing.

Reinforcement Learning techniques offer the capability for a controller to learn how to act for meeting run-time requirements via trials-and-errors on simulated or real data. There already exist several approaches to learn control techniques which leverage RL. Most of them use Q-Learning [44, 15], sometimes on a dedicated hardware module [7]. The majority of works that consider the verification of a trained RL policy are based on (DQL). Verification on DQL is intrinsically complex, partly due to the continuous input leading to an infinite number of states. [3] verifies DQL by extracting decision tree policies from a trained neural network. Though, they only verify robustness, stability and functional correctness of a controller. The authors of [20] and [18] both propose a *verification-in-the-loop* method, i.e., they perform the verification during training. However, their works only consider verification goals formulated using *ACTL* (a subset of CTL) and LTL specifications. Our approach, on the other hand, can verify goals formulated in not only CTL and LTL, but PCTL, too.

6 Conclusion

In this paper, we have presented a novel technique using reinforcement learning for automatically generating feedback-based run-time requirement enforcers that can be formally verified concerning a given set of verification goals by a model checker. For that, we elucidated a formalism for transforming reinforcement learning agents during training into enforcement state machines and applying model checking techniques in regular intervals to verify these to satisfy a set of verification goals. We were able to demonstrate in a case study using an object recognition application that our proposed approach significantly outperforms related work in being able to generate verified enforcers.

References

1 Nidhi Anantharajaiah, Tamim Asfour, Michael Bader, Lars Bauer, Jürgen Becker, Simon Bischof, Marcel Brand, Hans-Joachim Bungartz, Christian Eichler, Khalil Esper, Joachim Falk, Nael Fasfous, Felix Freiling, Andreas Fried, Michael Gerndt, Michael Glaß, Jeferson Gonzalez, Frank Hannig, Christian Heidorn, Jörg Henkel, Andreas Herkersdorf, Benedict Herzog, Jophin John, Timo Hönig, Felix Hundhausen, Heba Khdr, Tobias Langer, Oliver Lenke, Fabian Lesniak, Alexander Lindermayr, Alexandra Listl, Sebastian Maier, Nicole Megow, Marcel Mettler, Daniel Müller-Gritschneder, Hassan Nassar, Fabian Paus, Alexander Pöppl, Behnaz Pourmohseni, Jonas Rabenstein, Phillip Raffeck, Martin Rapp, Santiago Narváez Rivas, Mark Sagi, Franziska Schirrmacher, Ulf Schlichtmann, Florian Schmaus, Wolfgang Schröder-Preikschat, Tobias Schwarzer, Mohammed Bakr Sikal, Bertrand Simon, Gregor Snelting, Jan Spieck, Akshay Srivatsa, Walter Stechele, Jürgen Teich, Isaías A. Comprés Ureña, Ingrid Verbauwhede, Dominik Walter, Thomas Wild, Stefan Wildermann, Mario Wille, Michael Witterauf, and Li Zhang. *Invasive Computing.* FAU University Press, 2022.

2 Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. On the Logical Characterisation of Performability Properties. In *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9-15, 2000, Proceedings*, volume 1853 of *Lecture Notes in Computer Science*, pages 780–792. Springer, 2000.

3 Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable Reinforcement Learning via Policy Extraction. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 2499–2509, Red Hook, NY, USA, 2018. Curran Associates Inc.

4 Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield Synthesis. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 533–548. Springer, 2015.

5 Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

6 Dakshina Dasari, Benny Akesson, Vincent Nélis, Muhammad Ali Awan, and Stefan M. Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *8th IEEE International Symposium on Industrial Embedded Systems, SIES 2013, Porto, Portugal, June 19-21, 2013*, pages 39–48. IEEE, 2013.

7 Yvan Debizet, Guénolé Lallement, Fady Abouzeid, Philippe Roche, and Jean-Luc Autran. Q-Learning-based Adaptive Power Management for IoT System-on-Chips with Embedded Power States. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018.

8 Christophe Dubach, Timothy M Jones, Edwin V Bonilla, Michael FP O'Boyle, et al. A Predictive Model for Dynamic Microarchitectural Adaptivity Control. In *MICRO*, volume 43, pages 485–496, 2010.

9 Khalil Esper, Stefan Wildermann, and Jürgen Teich. A comparative evaluation of latency-aware energy optimization approaches in many-core systems. In *Second Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

10 Khalil Esper, Stefan Wildermann, and Jürgen Teich. Enforcement FSMs: Specification and Verification of Non-Functional Properties of Program Executions on MPSoCs. In *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 21–31, 2021.

11 Khalil Esper, Stefan Wildermann, and Jürgen Teich. Multi-Requirement Enforcement of Non-Functional Properties on MPSoCs Using Enforcement FSMs-A Case Study. In *Third Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

12 Ylies Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime Enforcement Monitors: Composition, Synthesis, and Enforcement Abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.

13 Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision Meets Robotics: The KITTI Dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

14 Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous Deep Q-Learning with Model-Based Acceleration. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 2829–2838. JMLR.org, 2016.

15 Ujjwal Gupta, Sumit K. Mandal, Manqing Mao, Chaitali Chakrabarti, and Umit Y. Ogras. A Deep Q-Learning Approach for Dynamic Management of Heterogeneous Processors. *IEEE Computer Architecture Letters*, 18(1):14–17, 2019.

16 Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

17 Hans Hansson and Bengt Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

18 Mohammadhosein Hasanbeig, Daniel Kroening, and Alessandro Abate. Towards verifiable and safe model-free reinforcement learning. In Nicola Gigante, Federico Mari, and Andrea Orlandini, editors, *Proceedings of the 1st Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis, co-located with the 18th International Conference of the Italian Association for Artificial Intelligence, OVERLAY@AI*IA 2019, Rende, Italy, November 19-20, 2019*, volume 2509 of *CEUR Workshop Proceedings*, page 1. CEUR-WS.org, 2019.

19 Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. POET: A Portable Approach to Minimizing Energy under Soft Real-Time Constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 75–86. IEEE, 2015.

20 Peng Jin, Jiaxu Tian, Dapeng Zhi, Xuejun Wen, and Min Zhang. Trainify: A CEGAR-Driven Training and Verification Framework for Safe Deep Reinforcement Learning. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, volume 13371, pages 193–218. Springer International Publishing, Cham, 2022. Series Title: Lecture Notes in Computer Science. URL: `https://link.springer.com/10.1007/978-3-031-13185-1_10`.

21 David H. K. Kim, Connor Imes, and Henry Hoffmann. Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications, CPSNA 2015, Kowloon, Hong Kong, China, August 19-21, 2015*, pages 78–85. IEEE Computer Society, 2015.

22 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Quantitative Analysis With the Probabilistic Model Checker PRISM. *Electron. Notes Theor. Comput. Sci.*, 153(2):5–31, 2006.

23 Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

24 David G Lowe. Object Recognition from Local Scale-Invariant Features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. IEEE, 1999.

25 Sumit K Mandal, Ganapati Bhat, Janardhan Rao Doppa, Partha Pratim Pande, and Umit Y Ogras. An Energy-Aware Online Learning Framework for Resource Management in Heterogeneous Platforms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 25(3):1–26, 2020.

26 Sumit K. Mandal, Ganapati Bhat, Chetan Arvind Patil, Janardhan Rao Doppa, Partha Pratim Pande, and Umit Y. Ogras. Dynamic Resource Management of Heterogeneous Mobile Platforms via Imitation Learning. *IEEE Trans. Very Large Scale Integr. Syst.*, 27(12):2842–2854, 2019.

27 Anway Mukherjee and Thidapat Chantem. Energy Management of Applications With Varying Resource Usage on Smartphones. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 37(11):2416–2427, 2018.

28 Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Hierarchical Power Management for Asymmetric Multi-Core in Dark Silicon Era. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 174:1–174:9. ACM, 2013.

29 Behnaz Pourmohseni, Michael Glaß, Jörg Henkel, Heba Khdr, Martin Rapp, Valentina Richthammer, Tobias Schwarzer, Fedor Smirnov, Jan Spieck, Jürgen Teich, et al. Hybrid Application Mapping for Composable Many-Core Systems: Overview and Future Perspective. *Journal of Low Power Electronics and Applications*, 10(4):38, 2020.

30 Fred B Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

31 John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint*, 2017. `arXiv:1707.06347`.

32 David C Snowdon, Etienne Le Sueur, Stefan M Petters, and Gernot Heiser. Koala: A Platform for OS-Level Power Management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 289–302, 2009.

33 Jan Spieck, Stefan Wildermann, and Jürgen Teich. Scenario-Based Soft Real-Time Hybrid Application Mapping for MPSoCs. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

34 Jan Spieck, Stefan Wildermann, and Jürgen Teich. Domain-Adaptive Soft Real-Time Hybrid Application Mapping for MPSoCs. In *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6. IEEE, 2021.

35 Jan Spieck, Stefan Wildermann, and Jürgen Teich. A Learning-Based Methodology for Scenario-Aware Mapping of Soft Real-Time Applications onto Heterogeneous MPSoCs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2022.

36 Jan Spieck, Stefan Wildermann, and Jürgen Teich. On Transferring Application Mapping Knowledge Between Differing MPSoC Architectures. In *CODES+ISSS 2022*, 2022.

37 Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction.* Adaptive computation and machine learning. MIT Press, 1998. URL: `https://www.worldcat.org/oclc/37293240`.

38 Jürgen Teich, Michael Glaß, Sascha Roloff, Wolfgang Schröder-Preikschat, Gregor Snelting, Andreas Weichslgartner, and Stefan Wildermann. Language and Compilation of Parallel Programs for-Predictable MPSoC Execution Using Invasive Computing. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 313–320. IEEE, 2016.

39 Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. Invasive Computing: An Overview. *Multiprocessor System-on-Chip*, pages 241–268, 2011.

40 Jürgen Teich, Pouya Mahmoody, Behnaz Pourmohseni, Sascha Roloff, Wolfgang Schröder-Preikschat, and Stefan Wildermann. Run-Time Enforcement of Non-functional Program Properties on MPSoCs. In *A Journey of Embedded and Cyber-Physical Systems*, pages 125–149. Springer, 2021.

41 Jürgen Teich, Behnaz Pourmohseni, Oliver Keszöcze, Jan Spieck, and Stefan Wildermann. Run-Time Enforcement of Non-Functional Application Requirements in Heterogeneous Many-Core Systems. In *25th Asia and South Pacific Design Automation Conference, ASP-DAC 2020, Beijing, China, January 13-16, 2020*, pages 629–636. IEEE, 2020.

42 Xiaohang Wang, Amit Kumar Singh, Bing Li, Yang Yang, Hong Li, and Terrence Mak. Bubble Budgeting: Throughput Optimization for Dynamic Workloads by Exploiting Dark Cores in Many Core Systems. *IEEE Transactions on Computers*, 67(2):178–192, 2017.

43 Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, UK, May 1989.

44 Amir Yeganeh-Khaksar, Mohsen Ansari, Sepideh Safari, Sina Yari-Karin, and Alireza Ejlali. Ring-DVFS: Reliability-Aware Reinforcement Learning-Based DVFS for Real-Time Embedded Systems. *IEEE Embedded Systems Letters*, 13(3):146–149, 2021.

A Appendix

Algorithm 1 Q-Learning.

```
Initialize $Q$ arbitrarily
**for all** $ep \in \{1, 2, ..., n_{\text{episodes}}\}$ **do**
    Initialize $v$ arbitrarily
    **while** $v$ is not terminal **do**
        Pick an action $a$ following an $\epsilon$-greedy policy
        Take action $a$
        $(v', \xi) \leftarrow ObserveEnvironment(v, a)$
        $Q(v, a) \leftarrow UpdateQTable(v, v', a, \xi)$
        $v \leftarrow v'$
    **end while**
**end for**
```

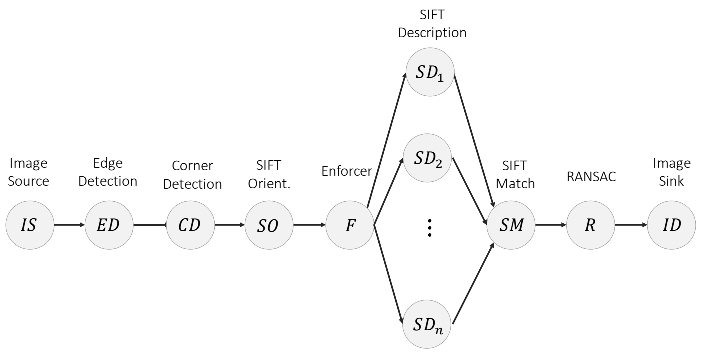


Figure 4 Actor graph of the evaluated object detection application.