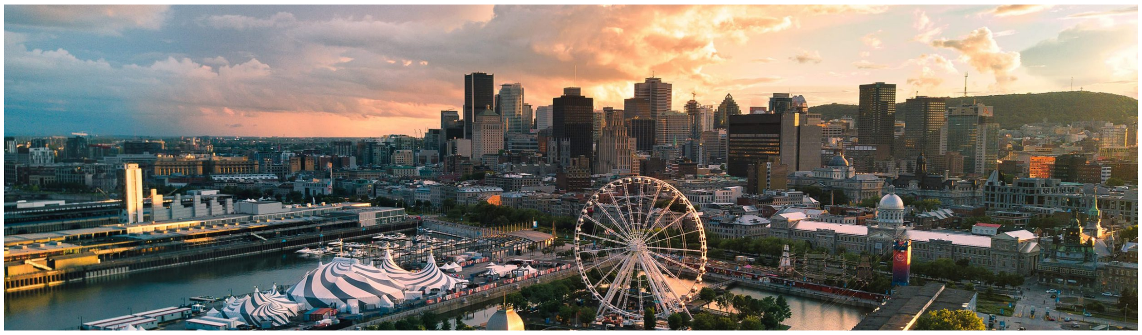


Brief Presentations Proceedings

(RTAS 2019)

Montreal, Canada

April 16, 2019

Message from the Chair

Welcome to the Brief Presentations Track of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium (BP-RTAS'19) held in Montreal, Canada. BP-RTAS'19 provides researchers with an opportunity to discuss their ideas, present ongoing work, gather feedback from experts and demonstrate their work with concrete systems, tools and prototypes in all areas of real-time embedded technology and applications as part of the CPS-IoT Week joint poster/demo session.

This year, BP-RTAS'19 accepted eleven presentations; seven work-in-progress papers and four tool and systems demos. Each submission has received three to four reviews. The accepted presentations cover various research topics in real-time systems such as real-time scheduling, real-time networks, autonomous and intelligent systems, IoT, GPU resource management, etc.

We would like to take this opportunity to express our gratitude to the members of the Program Committee for thoroughly reviewing all the submitted papers. We would also like to thank all the authors who submitted their work to BP-RTAS'19 and hence contributed to its success.

Organizing this track would not have been possible without the help of many people. First, we would like to thank Björn Brandenburg, the Technical Chair of the RTAS'19 conference for his guidance and support. We would also like to thank the General Chairs of the CPS-IoT Week, Rasit Eskicioglu and Xue (Steve) Liu, for their help in ensuring that everything goes well and smoothly.

On behalf of the Program Committee, we wish you a pleasant experience at the CPS-IoT Week. May the environment be stimulating and furthering fruitful discussions, and the presentations be enjoyable and entertaining.

Mitra Nasri
Delft University of Technology (TU Delft)
Brief Presentation Track Chair

Program Committee

Antoine Bertout	Inria de Paris, France
Arpan Gujarati	MPI-SWS, Germany
Catherine E. Nemitz	UNC, USA
Chang-Gun Lee	Seoul National University, Republic of Korea
Daniel Casini	Scuola Superiore Sant'Anna, Italy
Daniel Lohmann	Leibniz Universität Hannover, Germany
David Pereira	CISTER, ISEP, Portugal
Emmanuel Grolleau	LIAS, ISAE-ENSMA, France
Florian Pölzlbauer	Virtual Vehicle, Austria
Jing Li	New Jersey Institute of Technology, USA
Kecheng Yang	Texas State University, USA
Leonie Köhler	TU Braunschweig, Germany
Matthias Becker	KTH, Stockholm, Sweden
Mohamed Hassan	University of Guelph, Canada
Olaf Spinczyk	Osnabrück University, Germany
Zhishan Guo	University of Central Florida, USA

Table of Contents

Experience Report: Lightweight Implementation of a Controller Area Network to Ethernet Gateway

Florian Pölzlbauer
Virtual Vehicle Research Center
Graz, Austria
florian.poelzlbauer@v2c2.at

Allan Tengg
Virtual Vehicle Research Center
Graz, Austria
allan.tengg@v2c2.at

***Abstract*—In this paper we share our experience of implementing a *Controller Area Network to Ethernet gateway*. We present two implementations, and assess their performance.**

***Index Terms*—controller area network (CAN), Ethernet, gateway, implementation, automotive, SocketCAN, FreeRTOS**

I. Introduction and Motivation

Our research-car for autonomous driving is equipped with a large number of redundant sensors for environment perception (radar, lidar, cameras, ultrasonic, differential GPS, ...) and embedded computing platforms (DrivePX2 and MicroAutoBox). This allows us to develop fault-tolerant ADAS functions (such as Automated Emergency Braking) for various driving and weather conditions.

However, integration of office-grade PCs (for rapid prototyping, monitoring, and debugging) is challenging due to the number of CAN networks. In order to address this issue, we decided to implement a *Controller Area Network to Ethernet* gateway, which collects the CAN data and relays them via Ethernet (and vice versa).

II. Design

A. Requirements and Constraints

Our main requirements were as follows: The gateway must relay all CAN-messages to the Ethernet network in the correct ordering. No messages must get lost. The introduced delay must be short and upper-bounded. CAN-over-Ethernet data sent to the gateway shall be relayed to the according CAN-network (again with short delay). However, the gateway must not overload the CAN networks. The gateway must support CAN 1.0 and CAN 2.0, and be future-proofed towards CAN-FD. The Ethernet-side must be compatible with office-grade PCs.

Based on these requirements we derived a set of key design decisions: We use “standard” 100BaseTX Ethernet with Cat5e cables and RJ45 plugs. This allows us to integrate the gateway into the already installed Ethernet network. Automotive Ethernet (such as BroadR-Reach, 100BaseT1, or 1000BaseT1) would need special hardware which would hinder compatibility with office-grade PCs. We use IP/UDP as the transport layer. Inside the UDP packets, the CAN messages are encoded. Protocols such as Audio Video Bridging (AVB) [1] or Time-Sensitive Networking (TSN) [2] were initially considered, but again it would complicate the integration of office-grade PCs.

B. Software Architecture

The gateway provides bidirectional data-exchange between CAN and Ethernet. Hence, the architecture can be split into 2 paths.

Fig. 1 shows the *CAN-to-Ethernet* path. For each CAN port we have a CAN-receive thread. Whenever a CAN message is received, it immediately records a timestamp and then copies the CAN message into a thread-safe FIFO. The FIFO keeps track of how many CAN messages are queued inside, and how many bytes this accounts for. If the number of CAN messages reaches a user-defined value (i.e. *max. number of CAN messages per UDP packet*) or the size reaches the UDP's MTU (i.e. 1472 bytes), the content of the FIFO is copied into a UDP packet and the UDP packet is sent out. Details about the encoding is discussed in section II-C. In addition there is an Ethernet-transmit-timeout thread, similar to [3]. It is responsible for monitoring the time since the last UDP packet was sent. Once this timespan reaches a user-defined value (i.e. *UDP-transmit timeout*) the thread packs the content of the FIFO into a UDP packet and sends it. This ensures that the propagation delay from CAN to Ethernet is upper-bounded by the timeout.

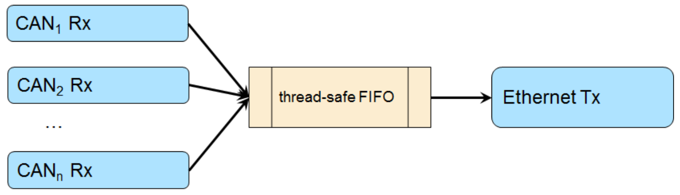


Fig. 1. Architecture: CAN-to-Ethernet

Fig. 2 shows the *Ethernet-to-CAN* path. The Ethernet-receive thread listens for incoming CAN-over-Ethernet packets. Once such a UDP packet is received, it immediately extracts all CAN messages and copies them into the respective thread-safe FIFOs. There is one FIFO per CAN port. The FIFO is responsible for queuing the CAN messages. The CAN-transmit thread takes the oldest CAN message out of the FIFO and sends it out onto the CAN network. In order to avoid that the CAN network is overloaded by a large burst of CAN messages, the CAN-transmit thread applies traffic-shaping. It keeps track when the last CAN message was sent, and waits for a user-defined timespan (i.e. *min. waiting time between CAN messages*) before sending the next CAN message.

It has been shown [4] that FIFOs introduce an additional queueing delay to CAN systems, and hence lead to a longer

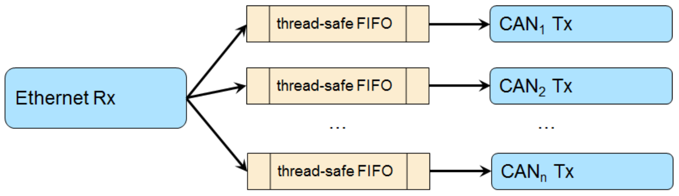


Fig. 2. Architecture: Ethernet-to-CAN

response time. Priority-based queues would circumvent this issue. However, the main reason for using FIFOs is that they conserve the CAN message ordering. In automotive systems there are many higher-level protocols (e.g. Diagnosis over CAN) where the message ordering is of upmost importance. Hence we had to accept the additional FIFO-delay, in order to maintain the correct CAN message ordering.

C. *CAN-over-Ethernet Protocol*

In order to transmit CAN messages over the Ethernet network, we have developed a lightweight encoding schema (see table I). It starts with a header (see top). Afterward it contains 0..N CAN message headers (see middle). Finally it contains the payload-bytes of the CAN messages (see bottom) which are arranged as CAN message 1 payload byte 1, CAN message 1 payload byte 2, ..., CAN message N payload byte M. The field *payload offset* is used to mark the start of each message's payload.

Field	bits	Tx	Rx	Description
protocol	16	x	x	0x0100 marks a CAN-over-Ethernet packet
version	8	x	x	future-proofing for protocol-changes
trigger reason	8	x	-	why was packet sent by gateway
counter	32	x	-	number of packets sent by gateway since boot
timestamp (lo)	32	x	-	time [us] when packet was sent
timestamp (hi)	16	x	-	by gateway
num. messages	8	x	x	number of CAN messages inside packet
reserved	8	-	-	reserved for future-proofing
timestamp (lo)	32	x	-	time [us] when CAN message was
timestamp (hi)	16	x	-	received by gateway
source-port	8	x	-	CAN port at which CAN message was received by gateway
destination-port	8	-	x	CAN port to which CAN message must be routed by gateway
message ID	32	x	x	CAN message ID (11 or 29 bits)
flags	8	x	x	flags for ext, rtr, err, fd-edl, fd-brs, fd-esi
payload length	8	x	x	length [byte] of CAN message payload
payload offset	16	x	x	where does the payload start
payload	n*8	x	x	payload-bytes of CAN messages

TABLE I
CAN-over-Ethernet Packet Encoding

Column *Tx* indicates that the field is relevant for packets that are sent by the gateway (i.e. CAN-to-Ethernet). Column *Rx* indicates that the field is relevant for packets that are received by the gateway (i.e. Ethernet-to-CAN).

The field *counter* mainly serves a monitoring purpose, as it allows to detect lost Ethernet packets. It can also act as the gateway's heartbeat. The field *trigger reason* gives insight into the gateway's CAN message queuing procedure.

We decided to use IP/UDP as the underlying protocol. The gateway transmits the CAN-over-Ethernet packets to IP-address X.X.X.255 (i.e. broadcast) to port 8001. This way, each node on the network that listens on this UDP-port can receive the CAN-over-Ethernet packets. The gateway itself listens on UPD-port 8000 in order to receive CAN-over-Ethernet packets.

D. *Configuring the Gateway*

The gateway can receive configuration messages for setting several gateway-parameters. At the moment we support:

1) set timestamp [us] of gateway
2) set CAN baudrate [b/s]
3) set CAN termination
4) set timeout [ms] for CAN message queuing
5) set max number of CAN messages per UDP packet
6) set destination-IP for CAN-over-Ethernet packets
7) set waiting time [us] for CAN transmit (traffic shaping)

III. Implementation

We decided to implement two variants of the gateway (details see table II). The *automotive* one features an AURIX micro-controller running FreeRTOS. This should result in highly deterministic behaviour. The *general purpose* one features a Beaglebone Black micro-processor running Linux. It should offer sufficient resources for future extensions (e.g. CAN traffic logging/recording). Both gateway alternatives are implemented in C/C++.

	General Purpose	Automotive
uC	Beaglebone Black Rev. C [5]	AURIX TC275 [6]
CPU	1 core, ARM Cortex-A8, 1 GHz	3 cores, 200 MHz
memory	512 MB RAM, 4 GB eMMC Flash	4 MB program flash, 384 kB data flash
Ethernet	100 Mb/s	100 Mb/s
CAN	2x CAN	4x CAN/CAN-FD [7]
OS	Ubuntu 14.04.3 LTS [8]	FreeRTOS 7.1.0 [9]
IP/UDP	see Linux kernel	lwIP [10]
CAN	SocketCAN [11]	own implementation

TABLE II
Implementations Alternatives

Automotive: Since FreeRTOS [9] did not support the AURIX micro-controller out of the box, we first had to port it [12]. The port places one kernel-instance on each of the 3 cores, and provides inter-core communication mechanisms. For the gateway implementation however we only used one core.

General Purpose: As of Linux kernel version 2.6.25, Linux supports the CAN protocol via the SocketCAN [11] library. Here, CAN-communication is handled like other sockets. We decided to use this library in order to evaluate its performance. The individual parts of the gateway software are implemented using *pthread*s.

IV. Performance Tests

A. *Test Setup*

In order to test the functionality and performance of the gateway, we use the following setup: A PC is connected to the gateway using a CAN-USB-adapter [13]. In addition the PC is connected directly to the gateway via Ethernet. No

Ethernet switches are used in between, so that no additional delays are introduced. The PC is responsible for generating the stimulus (CAN or Ethernet), and receiving the gateway's response (Ethernet or CAN). Data is also logged for post-processing and statistical analysis.

B. CAN-to-Ethernet

The CAN stimulus is generated as follows: A burst of b CAN messages is sent. No waiting time in between them. After the burst we wait $1..p$ ms (random uniform timespan between 1 and p, emulating a jittered period). Then we send the next burst. This procedure is repeated until n CAN messages are sent in total. All CAN messages have the same ID (0x000). Each CAN message's payload contains an uint64-number, which is increased in steps of one. This allows us to detect exactly which CAN message was lost (if any). In addition we can detect if CAN messages arrive in the wrong order. This would mean that the gateway would have modified the CAN message sequence. The bursty traffic is inspired by the radar-sensors which are used in our car. They send a burst every 66ms.

The first tests aim at detecting **message loss** and **message sequence flip**. Therefore, we use the following test-scenarios:

burst	b	1, 2, 3, 4, 5, 10, 20, 30, 40, 50
waiting	$1..p$	1, 5, 10, 50, 100, 500, 1000 ms
total	n	10.000

Analysis of the response (UDP packets sent by the gateway) we find that *no message* was *lost*, and *all messages* were relayed in the *correct order*. This result applies to both gateway implementations.

Next, we want to determine the minimum **delay** that is **introduced by the gateway**. In order to measure this, we set the gateway-parameter *max. CAN messages per UDP packet* to 1. By measuring the timespan from *"CAN message is received by the gateway"* to *"UDP packet is sent by gateway"* we get the propagation delay. Both time-instances are measured by the gateway itself and can be retrieved from the UDP packet (see packet encoding, table I). Note that this test does not measure the additional delay that is introduced by the FIFO-queuing, in case several CAN messages are queued. This time however is upper-bounded by the parameter *UDP-transmit timeout*. Our experiment solely focuses on the delay introduced by the message re-coding.

First we want to see if the bursty nature of the CAN traffic has any impact onto the gateway's propagation delay. Thus, we use the following scenarios:

burst	b	1, 5, 10, 30, 50
waiting	$1..p$	10 ms
total	n	10.000
timeout	to	1000 ms

For the automotive gateway we measured a propagation delay of up to 40 us. For the general purpose gateway the delay is similar, however some outliers range up to 100 us (see fig. 3). By taking a closer look we find that 50% of the values are between 10 and 11 us (automotive) and 5 to 6 us (general purpose).

Next, we want to see if the periodicy of the CAN traffic has any impact onto the gateway's propagation delay. Thus, we use the following scenarios:

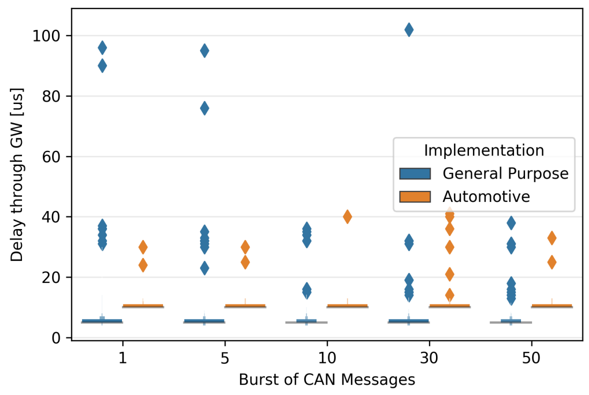


Fig. 3. Propagation delay, for varying bursts

burst	b	1
waiting	$1..p$	1, 5, 10, 50, 100, 500, 1000 ms
total	n	10.000
timeout	to	1000 ms

For the automotive gateway we measure a propagation delay of 10 to 50 us (with one outlier of 100 us). For the general purpose gateway the delay is up to 100 us, with one outlier of 150 us (see fig. 4). By zooming in we see that 50% of the data is between 10 and 11 us (automotive) and 5 to 11 us (general purpose). Interestingly, for the general purpose gateway, the propagation delay slightly increases as the periodicy of the CAN traffic increases.

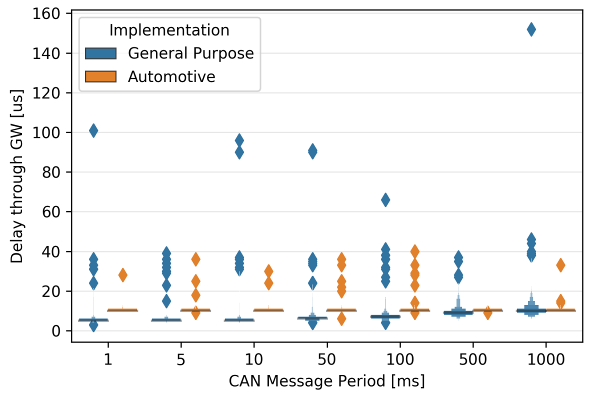


Fig. 4. Propagation delay, for varying periodicy

In conclusion, the automotive gateway is more predictable (due to the real time operating system), but the general purpose gateway has smaller propagation delays (due to the higher processing power). Both gateways perform acceptably well. No CAN messages get lost, nor is the message sequence altered.

C. Ethernet-to-CAN

The UDP stimulus is generated as follows: A set of b CAN messages are packed into a UDP packet and sent out. After that we wait $1..p$ ms. Then we send the next UDP packet containing b CAN messages. This procedure is repeated until n UDP packets are sent. All CAN messages have the same ID (0x001). Each CAN message's payload contains an uint64-number, which is increased in steps of one. This allows us to detect exactly which CAN message was lost (if any). and if the CAN messages arrive in the wrong order.

First, we want to test if all CAN messages get properly relayed (no message lost, sequence maintained). Secondly we want to see if the **CAN transmit traffic shaping** works as intended. Therefore we use the following scenarios:

batch	b	60
waiting	$1..p$	1, 5, 10, 50, 100, 500, 1000 ms
UDP	n	1000
min. Tx waiting	mw	0, 100, 150, 250, 500, 1000 us

As we decrease the periodicy of the UDP packets below 50 ms, we noted that the automotive gateway could no longer handle all CAN messages. It started losing several CAN messages (see fig. 5). This can be explained as follows: Due to the limited memory of the Aurix micro-controller, we decided to limit the CAN out FIFOs to 1000 messages (for each CAN port). For average Ethernet-to-CAN traffic this should be sufficient. However our performance tests intentionally used high traffic in order to identify the limits of our implementation.

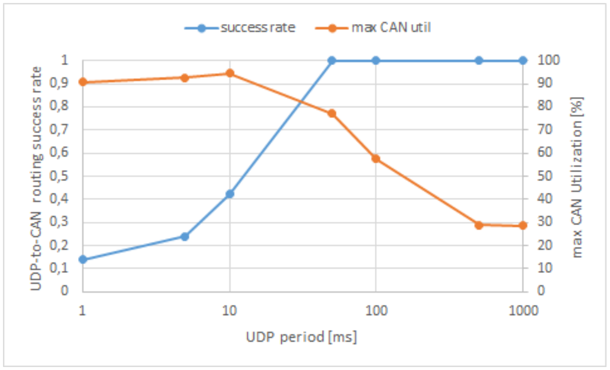


Fig. 5. Ethernet-to-CAN Tests, automotive

On the Beaglebone micro-processor we have 512 MB RAM. Hence, here we decided to use a dynamic FIFO size for the CAN out FIFOs. As a consequence we did not notice any message-loss for the general purpose gateway.

Since the general purpose gateway has sufficient space to queue the CAN messages, we implemented the CAN transmit traffic shaping here (in order to smooth out CAN traffic peaks).

As we increase the *CAN transmit waiting time* of the traffic-shaper, we can effectively decrease the CAN utilization (see fig. 6).

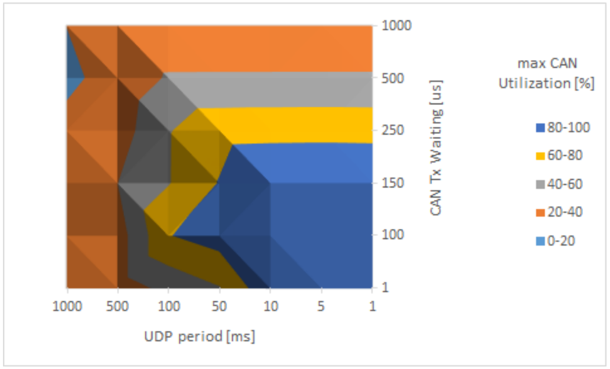


Fig. 6. Ethernet-to-CAN Tests, general purpose, CAN traffic shaping

However, we also noted an issue. As we decrease the *CAN transmit waiting time* below 150 us, we started losing CAN messages. For 100 us, the loss-rate was between 4 and 23%. For 0 us, the loss-rate was between 35 and 63%.

We strongly suspected that this issue is rooted in the SocketCAN library. As shown in [14] SocketCAN shares a single buffer for all sockets (CAN, Ethernet, Bluetooth, etc.). Hence, if the rate of messages is too short, messages will simply be overwritten by new ones. One solution would be to use the LinCAN library [15]. For now, we decided to lower-limit the waiting time of the traffic-shaper to 150 us.

V. Conclusion & Outlook

The systematic assessment of the gateway-implementations shows that both gateways meet our demands. While the *automotive* one offers slightly better predictability, the *general purpose* one offers significant resource-reserves for future extensions (such as data-recording).

In the future we want to improve the predictability of the *general purpose* gateway by utilizing the *PREEMPT_RT patch* or *real-time Linux*. We also consider switching to TCP/IP to utilize its *acknowledgment mechanism* to increase reliability.

Acknowledgment

The project "AutoDrive" has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737469. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Germany, Austria, Spain, Italy, Latvia, Belgium, Netherlands, Sweden, Finland, Lithuania, Czech Republic, Romania, Norway. In Austria the project was also funded by the program "IKT der Zukunft" and the Austrian Federal Ministry for Transport, Innovation and Technology (bmvit). The publication was written at VIRTUAL VEHICLE Research Center in Graz and partially funded by the COMET K2 Competence Centers for Excellent Technologies Programme of the Federal Ministry for Transport, Innovation and Technology (bmvit), the Federal Ministry for Digital, Business and Enterprise (bmdw), the Austrian Research Promotion Agency (FFG), the Province of Styria and the Styrian Business Promotion Agency (SFG).

References

[1] IEEE, "Audio video bridging task group," http://www.ieee802.org/1/pages/avbridges.html.

[2] ——, "Time-sensitive networking task group," http://www.ieee802.org/1/pages/tsn.html.

[3] J.-L. Scharbarg, M. Boyer, and C. Fraboul, "CAN-Ethernet architectures for real-time applications," in *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, 2005, pp. 245–252.

[4] R. I. Davis, S. Kollmann, V. Pollex, and F. Slomka, "Controller Area Network (CAN) Schedulability Analysis with FIFO Queues," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2011, pp. 45–56.

[5] "Beaglebone Black," https://beagleboard.org/black.

[6] Infineon, "AURIX TC270 / TC275 / TC277," 2017, data-sheet: https://www.infineon.com/dgdl/Infineon-TC27xDC_DS_v10-DS-v01_00-EN.pdf?fileId=5546d46259d9a4bf015a846b292f74ce.

[7] ——, "Controller Area Network Controller (MultiCAN)," 2015, application-note: http://www.infineon.com/dgdl/Infineon-MultiCAN-XMC4000-AP32300-AN-v01_00-EN.pdf?fileId=5546d4624e765da5014ed91d6be32110.

[8] "Ubuntu," https://www.ubuntu.com.

[9] R. Barry, "The FreeRTOS Kernel," https://www.freertos.org/.

[10] A. Dunkels, "lwIP - A Lightweight TCP/IP stack," https://savannah.nongnu.org/projects/lwip/.

[11] "SocketCAN," https://github.com/linux-can/.

[12] A. Tengg, "FreeRTOS 7.1 Port for Aurix (TC27x) using Free Entry Toolchain," https://interactive.freertos.org/hc/en-us/community/posts/210026366-FreeRTOS-7-1-Port-for-Aurix-TC27x-using-Free-Entry-Toolchain.

[13] PEAK Systems, "PCAN-USB FD: CAN-FD-Interface for High-Speed USB 2.0," https://www.peak-system.com/PCAN-USB-FD.365.0.html?&L=1.

[14] M. Sojka, P. Pisa, M. Petera, O. Spinka, and Z. Hanzalek, "A Comparison of Linux CAN Drivers and their Applications," in *International Symposium on Industrial Embedded Systems (SIES)*, 2010.

[15] P. Pisa, "Linux/RT-Linux CAN driver (LinCAN)," http://freshmeat.net/projects/lincan.

Time-Aware Deep Intelligence on Batteryless Systems

Bashima Islam, Seulki Lee and Shahriar Nirjon
University of North Carolina at Chapel Hill
{bashima, seulki, nirjon}@cs.unc.edu

***Abstract*—In this paper, we propose real-time scheduling algorithms for batteryless sensing and event detection systems which execute real-time deep learning tasks and are powered solely by harvested energy. The sporadic nature of harvested energy, resource constraints of the embedded platform, and the computational demand of deep neural networks pose a unique and challenging real-time scheduling problem for which no solutions have been proposed in the literature. We empirically study the problem and model the energy harvesting pattern as well as the trade-off between the accuracy and execution of a deep neural network. We develop an imprecise computing-based real-time scheduling algorithm that improves the schedulability of deep learning tasks on intermittently powered systems.**

I. Introduction

The Internet of Things (IoT) promises to make our lives efficient, productive, enjoyable, and healthier by making everyday objects capable of sensing, computation, and communication. Many of these so-called IoT devices are powered by limited-capacity batteries—which makes them portable, mobile, small, and lightweight. Batteries, however, require periodic maintenance (e.g., replacement and recharging) which is an inconvenience at a large scale. To address this practical problem, batteryless IoT devices have been proposed, which harvest energy from ambient sources, e.g., solar, thermal, kinetic, and RF to power up the device. These devices, in principle, last forever—as long as the energy harvesting conditions are met. They typically consist of low-power sensors, microcontrollers, and energy-harvesting and management circuitry, and their applications are in many deploy-and-forget scenarios, e.g., wildlife monitoring, remote surveillance, environment and infrastructure monitoring, wearables, and implantables.

Many IoT applications require timely feedback. For instance, in an audio surveillance system, audio events such as gunshots, screaming, and breaking ins need to be detected and reported as fast as possible to initiate prompt actions. Similarly, an air-quality monitoring system needs to identify events such as gas-leakage or outbreak of harmful pollutants on time to avoid a disaster. Likewise, shared resources, such as gym-equipment and shared bikes in a campus, can be monitored in real-time to detect misuses or malfunctions, and to inform the authority about the incidence on time. While a batteryless system is desirable in these real-time sensing and event detection applications, the unpredictability of the harvested energy, combined with the complexity of on-device event detection tasks, complicates timely execution of machine learning-based event detection tasks on batteryless systems.

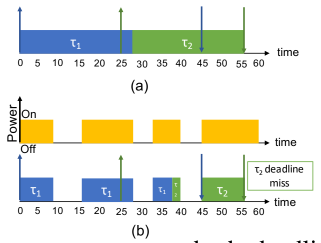


Fig. 1: (a) With constant power both deadlines are met. (b) With intermittent power, task τ_2 misses deadline.

Prior works on time-aware batteryless computing systems are primarily of two types. The first category focuses on *time-keeping*, i.e., maintaining a reliable system clock [1] even when the power is out. The sporadic nature of harvested energy in a batteryless system forces it to run intermittently by going through alternating episodes of power on and power off phases, which disrupts the continuity of the system clock. By exploiting the rate of decay of an internal capacitor and the content of the SRAM, these systems enable time-keeping during the absence of power. The second category proposes runtime systems that consider the temporal aspect of data across power failures [2]–[4]. [2] discards data after a predefined interval and thus saves energy by not processing stale data. [3], [4] propose energy-aware runtime systems to increase the likelihood of task completion. However, none of these consider the utility of data to the running application or the real-time deadline-aware execution of tasks.

Scheduling real-time machine learning tasks on a batteryless computing system is an extremely challenging feat. The two main sources of challenges are the *intermittent power supply* and the *computational demand for executing machine learning tasks*. These two challenges have been studied extensively in non-real-time settings. For instance, [5]–[9] enable seamless execution of non-real-time tasks on intermittently powered systems by proposing techniques that save and restore the program state across power failures. [10]–[12] propose lightweight and compressed deep neural networks for on-device machine learning on batteryless systems. However, none of these works consider the timing constraints of the machine learning tasks. In a real-time setting, simply applying these two types of solutions in conjunction with an existing real-time scheduling algorithm does not quite solve the problem at hand, which is illustrated in Figure 1. We consider two tasks, τ_1 and τ_2, having the release times of 0 and 25, respectively.

Their deadlines are 45 and 56, and both have an execution time of 28. In Figure 1(a), we observe that, under the earliest deadline first (EDF) scheduling, both tasks meet the deadlines when the power is uninterrupted. On the other hand, when power is intermittent, Figure 1(b) shows that task τ_2 misses its deadline.

The goal of this paper is to subdue the aforementioned challenges. In order to accomplish our goal, at first, we thoroughly study the energy harvesting pattern as well as the accuracy-execution trade-off of deep neural networks. From these studies, we make two observations. First, energy generated by a harvester is *bursty*, and therefore, its energy harvesting pattern can be modeled using a stochastic framework over a short period in time. Second, for a fixed classification accuracy, the amount of computation required for an input signal is dependent on the *hardness* of the data. We measure hardness in run-time using confidence of the classification result. By exploiting these two observations, we are able to design an *imprecise computing*-based, online, dynamic-priority, real-time scheduling algorithm that considers both the intermittent nature of the power supply as well as the accuracy-execution trade-off of the deep neural network model.

We conduct a simulation-driven experiment to evaluate the performance of the proposed scheduling algorithm. We generate 1,000 deep inference tasks, where a task refers to the execution of deep neural network inference for a data sample. We randomly assign different levels of hardness to each sample, where the hardness of a task indicates the number of layers required to achieve a certain classification accuracy. We compare the proposed scheduling algorithm with an earliest deadline first (EDF)-based imprecise scheduling algorithm as well as with a classification error-based imprecise scheduling algorithm. We show that our proposed algorithm misses 22%–25% less deadline while achieving 5%–7.5% better classification accuracy.

II. Preliminary Study

In this section, we study the energy harvesting pattern and the accuracy-execution trade-off of deep neural networks.

A. Modeling Energy Harvesting Pattern

Energy Events. Transiently powered systems operate intermittently because energy is not always available to harvest and, even when energy is available, buffering sufficient energy to perform adequate work takes time. In most cases, the pattern of this intermittency is stochastic and thus modeling this patter is not straight forward. To schedule the workload of an intermittently operating system at run-time, we decide whether to start execution of a task or not at a time instant. This decision heavily depends on the availability of energy available to harvest. To model the availability of energy, we define *energy event* which expresses the availability of sufficient energy during a period. Energy event represents a successful generation of at least K Joules of energy in total during T time slot. Here, K and T are system dependent. In order to understand the property of energy events, we observe the phenomenons causing energy events. For example, in a piezo-electric harvester, taking a minimal number of steps that generates at least K Joule of energy during T time slot is considered equivalent to the occurrence of an energy event. Similarly, we consider a minimal number packet transmissions per time slot and minimum intensity of solar per time slot as energy events for RF and solar harvesters, respectively.

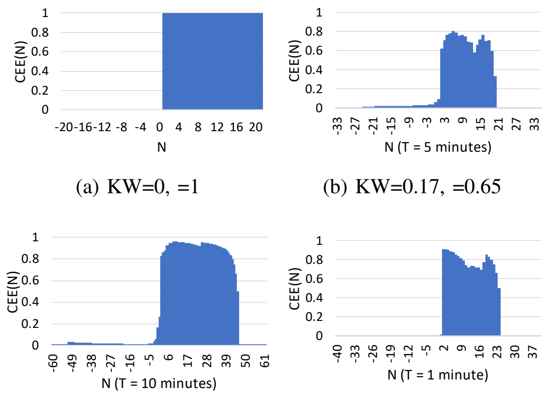
Properties of Energy Events. We study energy event patterns of three commonly used harvesters – piezo-electric harvester, solar harvester, and RF harvester from datasets. These datasets contain the number of steps taken during every 5-minute time-slot for 61 days, harvested solar energy measurements for three days and outbound packet transmission rate by an RF transmitter for 30 days. This study reveals two interesting observation about the pattern of energy events – (1) energy events occur in bursts where burstiness is the intermittent increases and decreases in activity or frequency of an event [**?**], (2) a probabilistic relation exists among the consecutive energy events during a short period. In other words, the occurrence of an energy event increases the probability of the next energy event during a short period. To illustrate, when a person starts walking the probability of continuing the walk is high within the first few time slots. This probability decreases with time. Likewise, when a person is sitting probability of remain set is high immediately, but decreases after a while.

Conditional Energy Event. We define *conditional energy event (CEE)* that represents the conditional probability of an energy event occurrence based on the occurrence/ absence of previous consecutive energy events. CEE(N) is the probability that an energy event will occur given immediately preceding N consecutive energy events occurred (for $N > 0$) or not occurred (for $N < 0$) The following equation expresses CEE.

$$\text{CEE(N)} = \begin{cases} p(occurrence|\text{ N consecutive occurrence}), & \text{if } N > 0 \\ p(occurrence|\text{ N consecutive non-occurrence}), & \text{if } N < 0 \end{cases} \quad (1)$$

To illustrate CEE(10) = 90% implies that the next energy event will occur with 90% probability if 10 immediately preceding consecutive energy event occurred. Similarly, CEE(-15) = 5% indicates the probability of an energy event at the current time slot is 5%, given that there were no energy events in the last 15 slots.)

The CEE of a system powered by a persistent power supply or an ideal harvester that has no intermittence looks like Figure 2(a). Figure 2(b-d) shows the CEE of three energy harvested systems. From these figures, we observe that for a small value of N these systems demonstrate similarity with the ideal correlative harvester. We measure the similarity of CEE of a harvested system with persistent powered or ideal harvested system using Kantorovich-Wasserstein (KW) distance. Through out this paper, we use KW to express the Kantorovich-Wasserstein distance between the CEE a system (H) and the CEE of persistently powered system (P). We also observe that for large |N| the CEE drops significantly because when the interval time between the first and current event increases their probabilistic relation decreases. For example,



(a) KW=0, =1 (b) KW=0.17, =0.65

(c) KW=0.06, =0.84 (d) KW=0.11, =0.76

Fig. 2: (a)CEE for persistent power source. (b) CEE for piezo-electric harvester. (c) CEE for solar harvester. (d) CEE for RF harvester. We use N=20 for calculating KW and .

a person is walking for a long time has a high probability of stopping.

B. Study of Deep Neural Network

In order to execute machine learning tasks in a resource-constrained batteryless system, we need to minimize memory and computation costs. To achieve this goal, we study several attributes that are unique to deep learning processes.

• *Significance of depth.* Deep learning algorithms have layered structures where the input of the first layer is from an external source, e.g., sensors. The output of a layer is fed as the input of the next layer, and it goes on until the end of the network. The total number of layers in a neural network is called *depth*. Increased number of layers and neurons both contribute to more complicated calculation resulting in higher accuracy. However, a shallow network requires width exponential to that of a deeper network to achieve similar accuracy. Therefore, the performance of a neural network does not only depends on the number of parameters but also depth. For example, VGGNet has 16 layers with 140M parameters, while ResNet beats it with 152 layers but only 2M parameters.

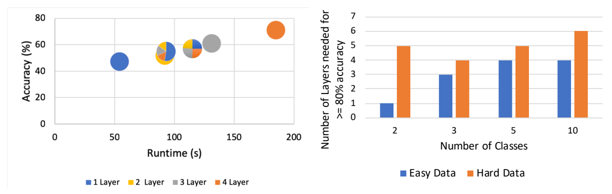


Fig. 3: (a) The accuracy and execution time of all data where each data executed a certain depth. (b) Hard data requires complex representation to achieve similar accuracy.

To illustrate the effect of depth let us consider a face detector. The first layer of this deep learning based face detector learns basic features, e.g., edges. In the next layer, a collection of edges, e.g., shape, is learned. The next layer might output a higher level feature, e.g., nose and the final output face abstraction in the last layer. To understand the effect, we develop acoustic deep neural network classifier with four layers and choose UrbanSound8K dataset which has ten classes. We use raw waveform of audio as input to the in Figure 3(a) shows that the depth has a linear relation with classification accuracy. However, this increased accuracy comes with the cost of higher execution time.

• *Data-dependency of required depth.* To decrease the execution time we observe the fact that depth is highly *data-dependent* [13]. If target classes are profoundly distinctive, then simple features can be used to distinguish them. For example, in Figure 3(b), audio of cat and water (easy data) are very distinguishable; thus a single layer RL achieves 93% accuracy. On the other hand, similar classes, e.g., train and helicopter (hard data) needs more complex representations to be distinct and thus require five layers to achieve 81% accuracy. By executing only necessary layers based on the hardness of data we can achieve similar accuracy with decreased execution time. We define the number of layers needed for a data sample to achieve a certain classification accuracy as *hardness* of that data sample.

III. Solution

To achieve our goal of executing machine learning tasks on a batteryless computing system on time, we propose the following solutions.

A. Predictability Factor

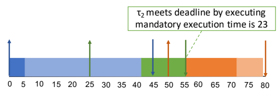
Based on our observation in Section II-A regarding the presence of probabilistic correlation between energy events, we propose an *energy event predictability factor* η. A η of 1 indicates a high correlation between energy events, while $\eta = 0$ indicates that energy events are independent.

B. Dynamic Imprecise Task Model with Early Exit

In Section II-B, we witness that the number of layers requires to achieve a certain accuracy depends on the hardness of data. Therefore, layer-aware early termination of a network is possible. We consider a task (τ) as the execution of the deep neural network for a sample. A scheduler can preempt τ only at the end of the execution of each layer. We define such tasks as *conditional-preemptive tasks*. Conditional preemptive tasks are different from cooperative tasks due to their ability of preemption before completion. This consists of two portions – mandatory and optional where mandatory precedes the optional. Completion of only the mandatory part within the deadline is considered schedulable. Executing the optional portion contributes to decreasing error. We consider the execution until the accuracy reaches a threshold as the mandatory portion. As the task is monotone, the optional portion execution might reduce the error. Due to the data dependency of layer-aware prior termination, the imprecise computing model in this paper is different from previously introduced imprecise computing. Unlike traditional imprecise computing [14], [15] where the execution time of the mandatory potion is pre-knowledge, the execution time of mandatory portion in this paper is determined at run-time. We define such imprecise tasks as *Dynamic Imprecise* Tasks.

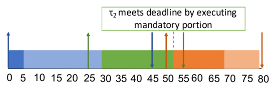
C. Priority Scheduling Algorithms

Finally, we propose a priority scheduling for dynamic imprecise tasks that minimize the error while scheduling the mandatory portions and name our priority function *certainty function*. To illustrate, let us consider a task-set consisting of 3 tasks (τ_1, τ_2 and τ_3) where the release time are 0, 25 and 50 respectively. Their deadlines are 45, 56 and 92 respectively. We assume that there are ten layers in the network and the mandatory portions of the tasks contain two layers, four layers, and three layers respectively. The execution time of each layer is 5, 5, 6, 7, 6, 5, 7, 7, 6 and six sequentially. In Figure 5(a), EDF fails to schedule the second task; however, certainty function can schedule all three tasks in Figure 5(b). In Figure 5(c-d) we take another case into account where the deadlines are 59, 69 and 84 respectively. The mandatory portion contains five, two and one layers consecutively. Even though both EDF and certainty function succeeds to schedule all the tasks, the accumulated error of the EDF schedule (20%) is higher than that of certainty function schedule (12%). For an energy harvester with high η, we introduce a variant of this algorithm that can boost the utilization. When the energy event rate is high, we schedule exhaustively, opportunely taking advantage of correlated energy event occurrence. When the energy event does not occur, the scheduler considers the high probability of energy event non-occurrence and schedules conservatively. During the exhaustive mode, the scheduler schedules optional tasks if there is a slack time like our proposed priority scheduling. In this mode, the scheduler considers that the system has enough incoming energy events for future jobs. On the other hand, during the conservative mode, the system consumes energy only when it is a must. Therefore, it only schedules mandatory portions and ignores the optional part completely. Thus it preserves power to spend on the mandatory portion of the upcoming tasks assuming low energy event rate.

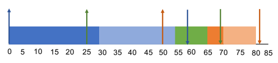


τ_2 meets deadline by executing mandatory execution time is 23
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80

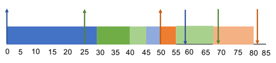
(a)



(b)



(c)



(d)

Fig. 4: (a) EDF fails to meet imprecise deadline. (b) Certainty function meets imprecise deadline (c) EDF meets deadline with 20% error (d) Certainty function meets deadline with 12% error.

In Figure 5, we compare our proposed certainty function based priority scheduling with classification error based priority scheduling and earliest deadline first (EDF) based imprecise scheduling with simulation. We simulate 1000 deep inference tasks with the early exit and 10-fold cross-validation. Each task refers to a data sample executing a ten layer deep neural network, and we randomly assign hardness to each data sample. Our proposed scheduler achieves 7.5% decreased error while meeting the deadline for 20% more tasks.

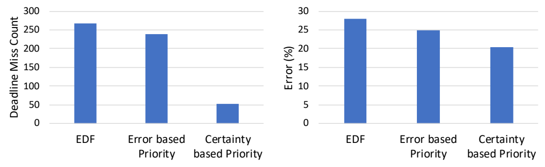


Fig. 5: Certainty function based priority scheduling meets more imprecise deadline with smaller error

Acknowledgment

This paper was supported, in part, by NSF grants CNS-1816213 and CNS-1704469.

References

[1] J. Hester, N. Tobias, A. Rahmati, L. Sitanayah, D. Holcomb, K. Fu, W. P. Burleson, and J. Sorber, "Persistent clocks for batteryless sensing devices," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 4, p. 77, 2016.

[2] J. Hester, K. Storer, and J. Sorber, "Timely execution on intermittently powered batteryless sensors," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 2017, p. 17.

[3] M. Buettner, B. Greenstein, and D. Wetherall, "Dewdrop: an energy-aware runtime for computational rfid," in *Proc. USENIX NSDI*, 2011.

[4] T. Zhu, A. Mohaisen, Y. Ping, and D. Towsley, "Deos: Dynamic energy-oriented scheduling for sustainable wireless sensor networks," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 2363–2371.

[5] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on rfid-scale devices," *Acm Sigplan Notices*, vol. 47, no. 4, pp. 159–170, 2012.

[6] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1968–1980, 2016.

[7] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 96, 2017.

[8] A. Colin and B. Lucia, "Chain: tasks and channels for reliable intermittent programs," *ACM SIGPLAN Notices*, 2016.

[9] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018.

[10] G. Gobieski, N. Beckmann, and B. Lucia, "Intermittent deep neural network inference," *SysML*, 2018.

[11] S. Nirjon, "Lifelong learning on harvested energy," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018, pp. 500–501.

[12] B. Islam and S. Nirjon, "Poster abstract: On-device training from sensor data onbatteryless platforms," in *The 18th ACM/IEEE Conference on Information Processing in Sensor Networks*. ACM/IEEE, 2019.

[13] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, "Adaptive neural networks for efficient inference," *arXiv preprint arXiv:1702.07811*, 2017.

[14] W.-K. Shih and J. W.-S. Liu, "On-line scheduling of imprecise computations to minimize error," in *Real-Time Systems Symposium, 1992*. IEEE, 1992, pp. 280–289.

[15] Y.-X. Zhang, C.-H. Fang, and Y. Wang, "A feedback-driven online scheduler for processes with imprecise computing," *Journal of Software*, 2004.

SpotON: Just-in-Time Active Event Detection on Energy Autonomous Sensing Systems

Yubo Luo
Department of Computer Science
UNC Chapel Hill
yubo@cs.unc.edu

Shahriar Nirjon
Department of Computer Science
UNC Chapel Hill
nirjon@cs.unc.edu

***Abstract*—We propose SpotON, which is an *active* event detection system that runs on harvested energy and adapts its sleeping cycle to match the distribution of the arrival of the events of interest. Existing energy harvesting systems wake up periodically at a fixed rate to sense and process the data to determine if the event of interest is happening. In contrast, SpotON employs reinforcement learning to learn the pattern of events at run-time and uses that knowledge to wake itself up when events are most likely to happen. Being able to remain asleep more often than a fixed wake-up system, SpotON is able to reduce energy waste, increase the amount of harvested energy, and be able to remain active for longer period in time when the events of interest are more likely to occur. We conduct a simulation-driven experiment to compare our proposed solution with a fixed-schedule system and results show that SpotON is able to capture 2–5X times more events and is 3–12X more energy-efficient than the baseline.**

***Index Terms*—Energy harvesting, Q-learning, Event detection.**

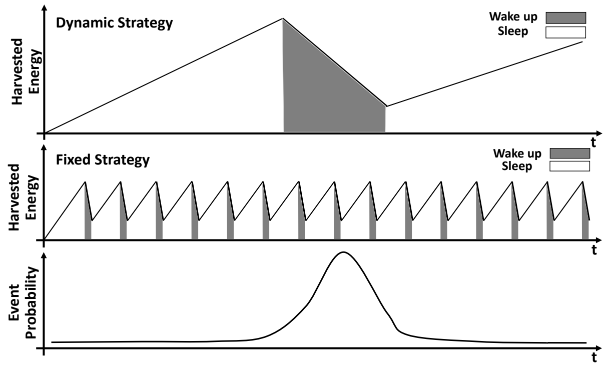


Fig. 1. Comparison between fixed and dynamic strategies. The fixed strategy can only detect a small fraction of total events but the dynamic strategy only wakes up during event-active time intervals and thus detects more events.

I. Introduction

As the development of computational power, computing algorithm and hardware, more and more stand-alone and sustainable applications are emerging, pushing the world forward to the ultimate instantiation of the Internet of Things (IoT) – the "Smart Dust". Smart Dust is a system of many tiny microelectromechanical systems that are energy autonomous [3]. However, current IoT world is dominated by battery-powered systems which are bulky and unsustainable. Energy harvesting is one of the ways that can lead us to the final instantiation of IoT.

Energy harvesting systems harvest energy from various energy sources, such as RF, piezoelectric or solar. It eliminates the need for replacing batteries and enables energy autonomy which is crucial to long-term sensing applications.

To achieve energy autonomy, we have to overcome challenges resulting from energy harvesting. First, computation in energy harvesting systems is intermittent and this causes problems because most computing algorithms are long-running programs. There is literature addressing how to enable correct execution of existing long-term running programs on energy harvesting systems by guaranteeing atomicity, data consistency, forward progress [7], [8]. Second, energy supply in energy harvesting systems is so limited that energy efficiency is crucial. Dewdrop [4] takes iterative tasks as a scheduling problem and dynamically changes the starting voltage based on the size of the next task to improve energy efficiency. Capybara [8] deploys an array of capacitors with different capacitance and dynamically changes the capacitance of the system. Mayfly [10] considers the timeliness of data and discards stale data to avoid wasting energy in learning outdated data. Third, event detection is a typical and important task in IoT world, but active event detection has yet not been well studied. Active event detection means that the event itself can not produce a trigger signal to wake up the micro-controller (MCU), and it requires the MCU to actively sense the event. More detailed explanation about passive and active event detection is described in Section II. There is one approach called Monjolo [5] addressing passive event detection in energy harvesting systems. It uses the energy harvested from the event itself as a trigger to wake up the system and thus avoids active sensing. However, Monjolo only applies to cases where the event itself can generate a certain amount of energy that Monjolo uses as a trigger. As far as we know, there is no literature addressing cases where the target event itself can not serve as a trigger. Thus, we propose SpotON, the first energy harvesting system that deals with active event detection by waking up just in time.

Most existing intermittently powered systems have predefined turn-on and turn-off thresholds, which means the system periodically wakes up at each capacitor charging cycle. Though some of them dynamically change the thresholds according to the complexity of the next task, they are still in the category of periodically charging and discharging, as shown in Figure 1. The key to enabling active event detection in energy autonomous systems is to ensure that there is enough energy left in the storage capacitor to power up the microcontroller when the event is about to happen. If the system magically knows when the event is about to happen, it can wake up more frequently during this time interval but remain powered-off more frequently at other times. SpotON learns the event pattern, predicts when the event is more likely to happen, keeps the system in powered-off more frequently when events are not active, and saves surplus energy in a dedicated capacitor array which can compensate the massive energy consumption of frequent waking-ups when events are active. In this way, we can borrow energy harvested from previous charging cycles.

In real-life scenarios, events behaves in a pattern which may change over time. SpotON uses a reinforcement learning algorithm – Q-learning to learn the event pattern online. If the event pattern changes over time, SpotON learns the new pattern and updates the system, making itself a "real-time" event detector.

II. Problem

The key to enabling active event detection is to decide when to wake up the MCU. One important part of the waking up process is the trigger mechanism. We classify the waking up trigger into the following three categories [6]: periodic, opportunistic and event-based.

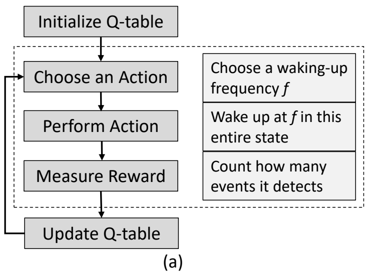
The periodic trigger wakes up the MCU at a fixed period, which requires a predictable energy supply. This type of trigger is rare in intermittent systems. The opportunistic trigger is common in intermittent systems which depends on the harvestable energy and a predefined voltage threshold. If the voltage of the storage capacitor reaches V_{th}, the trigger is activated, and the MCU keeps running until the voltage decreases to V_{min}. This type of trigger usually requires a dedicated voltage detection circuit and a power management module [8]. The event-based trigger wakes up the MCU based on an expected event. The system needs to harvest enough energy before a trigger event happens in order to be woken up successfully. If an event happens before enough energy is stored up the system will miss this event. The detection rate of this type of trigger is bounded by the capacitor recharging rate [5], [6]. SpotON can be considered as an event-based trigger but it applies to entirely different applications. The event-based trigger mentioned in [5], [6] only applies to cases where the event itself can actively activate a trigger signal, and this signal is leveraged to wake up the MCU. This trigger-activating process does not involve MCU. Possible applications for this type of event-based trigger include door opening detection [6] where the action of opening the door itself can vibrate a piezo sensor and generate a trigger signal and airflow monitoring [11] where the airflow itself can also vibrate a piezo sensor and generate a trigger signal. This type of sensing is passive. The MCU just passively waits for an waking-up signal.

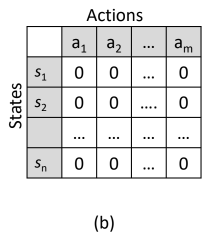
However, there are many applications whose target event can only be passively captured by active sensing controlled by the MCU, e.g., wildlife watering monitoring and environment noise detection. This type of applications requires active sensing from the MCU. SpotON is specially designed for this type of event detection. It learns the event pattern and uses it as an internal event-based trigger. SpotON only benefits cases where the event happens in a certain pattern that we can use some learning algorithm to learn. If the target event happens randomly without any pattern, then it is not a proper candidate application for SpotON.

We call our internally triggered waking-up approach a dynamic strategy, compared to the commonly used approaches which are static or fixed. Current fixed systems use up all energy immediately once they are charged to a threshold voltage and wake up. They do not borrow energy from previous charging cycles. By saying SpotON a dynamic strategy, we do not mean that the threshold voltages or the storage capacitance is dynamic, we mean that after the system has already harvested enough energy to wake up, it dynamically chooses to wake up at which frequency level, e.g., high frequency, low frequency or even not waking up, based on the learned event pattern. In this way, SpotON is able to wake up just in time.

III. Design Overview/ Implementation

To wake up the system in a proper time at a proper frequency based on the event pattern, we use reinforcement learning to implant this intelligence into SpotON.


Initialize Q-table
Choose an Action
Perform Action
Measure Reward
Update Q-table
Choose a waking-up frequency f
Wake up at f in this entire state
Count how many events it detects
(a)



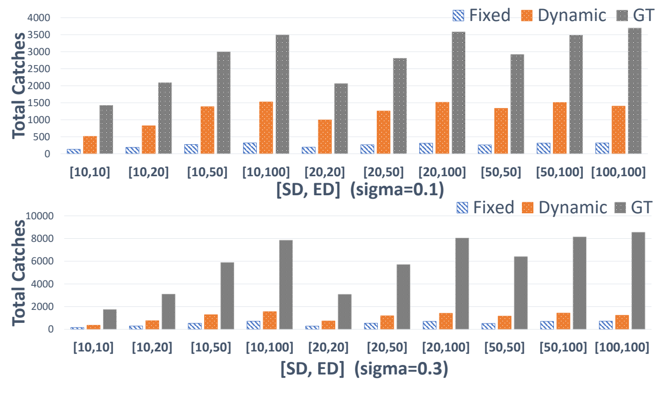
Actions					
States		a_1	a_2	...	a_m
	s_1	0	0	...	0
	s_2	0	0	....	0
		...	...	...	...
	s_n	0	0	...	0

Fig. 2. (a) Workflow of Q-learning [1]; (b) Initialized Q-table.

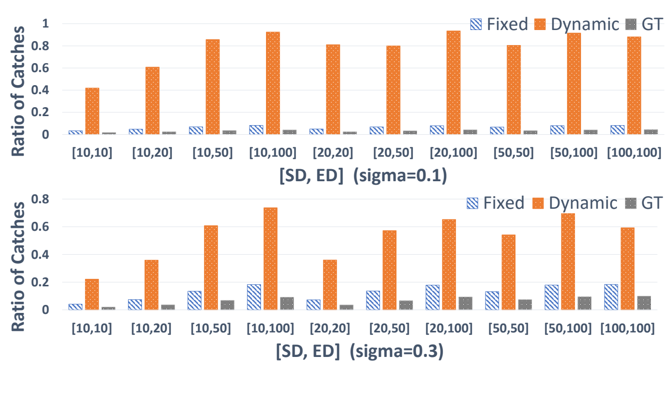
A. Q-Learning

Q-learning is employed to help SpotON learn the event pattern and make waking-up decisions. Q-learning [9] is an effective way of making optimal decisions to achieve the best reward based on past observations. The workflow of Q-learning is described in Figure 2(a).

The learned experience is stored in a Q-table which contains the weight for each state-action pair. We use $S = \{s_1, s_2, ..., s_n\}$ to denote the set of states and $A = \{a_1, a_2, ..., a_m\}$ the set of actions. In our case, one state s_i means a specific time interval. Adjacent states are adjacent time intervals. For example, if the time duration of each state is one hour and s_i is 9-10am, then s_{i+1} means 10-11am and s_{i-1} means 8-9am. Each state has the same duration length. One action a_j means a specific waking-up frequency. It could be



(a) Total catches



(b) The ratio of total catches to total waking-up time units

Fig. 3. Simulation results. There are two simulation settings. One contains 100 events sampled from $[\mu = 12, \sigma = 0.1]$ and 100 events sampled from $[\mu = 15, \sigma = 0.1]$. The other contains 100 events sampled from $[\mu = 12, \sigma = 0.3]$ and 100 events sampled from $[\mu = 15, \sigma = 0.3]$. SD is state duration and ED is event duration. The time unit is one second.

waking up MCU once every minute, or once every 10 minutes or once every hour, etc. Both the state duration and the waking-up frequency for each action are application-specific.

Q-learning is a table-based reinforcement learning algorithm. Q-table contains the weight of each state-action pair and these weights are updated at each step by calculating the reward gained from each step. One step corresponds to one state. The update equation of Q-table is as follows:

$$Q_{ij} = Q_{ij} + \alpha[R_{ij} + \gamma \max_{k=1,..,m}(Q_{i+1,k}) - Q_{ij}] \quad (1)$$

where Q_{ij} denotes the weight of the state-action pair of (s_i, a_j); R_{ij} denotes the reward gained in state s_i by taking action a_j; α denotes learning rate; γ denotes discount rate; max calculates the maximum expected future reward given the new state and all possible actions at that new state.

Figure 2(b) shows the initialized Q-table with zeros which can also be initialized by training on an off-line dataset. Training a Q-table using off-line dataset before deployment saves the time spent in learning the event pattern from scratch.

B. Online Adaptive Ability

In a real life scenario, the event pattern is usually not fixed, and it may change as time goes by. For example, in the application of wildlife watering monitoring, wildlife in the desert comes to water site to drink water, and our system wants to take as many pictures containing target wildlife as possible. However, wildlife's watering habit varies in different seasons. The system must be capable of online learning to keep the Q-table updated to the real-time environment.

In Q-learning, there is a parameter called ϵ related to action-taking decisions. At the beginning of each step, a random number r is generated and compared to ϵ. If $r > \epsilon$ the system takes an action based on Q-table. Otherwise, a random action is taken. The value of ϵ keeps decreasing to a small number as Q-table is more and more well-trained. Once ϵ decreases to its minimum value, the system is well-trained, and the Q-table stops updating. Larger ϵ value means the system explores more and smaller ϵ value means the system exploits more. To implant online learning ability to SpotON we need to reset this ϵ regularly. As for how often we should reset ϵ, it depends on the application.

IV. Evaluation

We have conducted computer simulations to evaluate the performance of SpotON. Our simulations are based on a python package called *gym* which is a toolkit for developing and comparing reinforcement learning algorithms [2]. The most important parameters are as follows:

- state duration (SD): how long one state lasts;
- event duration (ED): how long one event lasts;
- energy bank capacity (EBCap): how much energy the storage capacitor can save at maximum;
- event distribution parameters (μ, σ): we assume the event obeys normal distribution. μ and σ are the mean value and the standard deviation respectively.

To simplify the simulation, we assume our system harvests solar power, the time unit is one second and the total simulation time is one day. The energy consumption rate is 10 times as the energy storage rate, which means 10-second charging can support 1-second discharging. Solar energy is only harvestable during the daytime, e.g. from 6 am to 8 pm. We set the EBCap to 2-hour continuously charging, which means the capacitor can at maximum store energy harvested from two hours.

If the system wakes up at one time unit and there is an event happening, then we consider this time unit a catch or a positive waking-up. The number of catches or positive waking-ups is one of our evaluation metrics, and it means how many time units the system wakes up when there is an event. The ratio of positive waking-ups to total waking-ups is our second metric. The reason why we use these two evaluation metrics is that in real life scenarios we care more about if the MCU wakes up at the event-happening moment, which is related to the efficiency of energy usage. Taking the wildlife watering as an example, the system wakes up and takes a picture of the water site. The more pictures that capture an animal, the more efficiently the system uses the harvested energy. Energy used for waking up without capturing an event is wasted. There is a negative reward for waking up without a catch and a positive reward for waking up with a catch.

We compare SpotON (**Dynamic**) with the other two systems, the ground truth (**GT**) and the fixed system (**Fixed**). The ground truth system is powered by a battery so it can catch all time units when there is an event, and its total waking-ups equal to the length of the simulation which is 24 hours in our case. The Fixed system is the existing energy harvesting system which wakes up after the capacitor is fully charged and uses up all energy immediately.

Figure 3(a) demonstrates that our SpotON has more catches than the fixed system. Especially in the case of $\sigma = 0.1$, SpotON has around 5 times as many event catches as the fixed system. Compared to the ground truth, there are still a lot of missed time units where there are events. But remember our goal is to make as many positive waking-ups as possible, rather than capturing all event-happening time units. We also notice that SpotON has better performance for event distribution with smaller standard deviation. The reason is that smaller standard deviation means the data are more converged and it is easier for Q-learning to learn the event pattern.

TABLE I
AVERAGED RESULT FROM ALL SIMULATION RESULTS

	Number of catches		The ratio of positive waking-ups	
	$\sigma = 0.1$	$\sigma = 0.3$	$\sigma = 0.1$	$\sigma = 0.3$
Fixed	256	515	6.5%	13.2%
Dynamic	1225	1114	79.6%	53.4%
GT	2855	5848	3.3%	6.8%

Figure 3(b) shows the ratio of positive waking-ups to total waking-ups. SpotON significantly outperforms the other two systems. The highest ratio is 0.92 which means in that case 92% of SpotON's waking-ups happen when there is an event. The fixed system has only 10% positive waking-ups on average which means 90% of harvested energy is wasted in useless waking-ups. As expected, the ground truth system has the lowest energy usage efficiency because it is powered up all the time. Table I shows the averaged result from all results.

V. Future Plan

Our future work includes tuning the parameters of the Q-learning algorithm, considering the time, energy, and complexity of event detection algorithms, and implementation of the system on a real hardware.

- Algorithm parameters: In the preliminary experiments, we find that there are still opportunities for improvements. In some settings, there are unused energy in the capacitor at the end of the experiment. Currently, we use the same reward parameters for all simulations. Tuning the reward parameter in a more fine-grained way can make the system use up all harvested energy and perform better.
- Generalization: currently, we assume that the events are easily detected. For example, if we consider loud noises as events of interest, we only need to use an audio sensor and compare the signal amplitude to a threshold. If it is higher than the threshold, we consider it as an event. However, to generalize SpotON to other applications, we have to consider the energy consumption due to executing the recognition algorithm, e.g. image-based event detection where image recognition is needed to figure out if a picture contains an object or an event we are interested in and this consumes much more energy than merely sensing audio signals.
- Hardware: one crucial prerequisite for SpotON is that we must have a large and dynamic storage capacitor. For example, we want to monitor wildlife watering, and we assume that the animals come to the water site mostly in the afternoon. Our system gradually learns to wake up more frequently in the afternoon and sleeps more in the morning. The energy harvested in the morning must be stored somewhere and then be used to wake up MCU more frequently in the afternoon. Thus, saving such energy would require a variant design of Capybara [8] that can dynamically change the storage capacitance. This requires a complete energy management unit facilitated by specialized hardware and software design.

In summary, SpotON opens a new way for energy autonomous systems to detect events actively by waking up just in time and by adapting its internal model of the environment as the real-world environment changes. We also plan to implement SpotON on different energy harvesting platforms.

Acknowledgment

This paper was supported, in part, by NSF grants CNS-1816213 and CNS-1704469.

References

[1] Diving deeper into reinforcement learning with q-learning. https://medium.freecodecamp.org/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe. Accessed: 2019-02-20.

[2] Gym toolkit. https://gym.openai.com/. Accessed: 2019-02-20.

[3] Smart dust - wikipedia. https://en.wikipedia.org/wiki/Smartdust. Accessed: 2019-02-20.

[4] M. Buettner, B. Greenstein, and D. Wetherall. Dewdrop: an energy-aware runtime for computational rfid. In *Proc. USENIX NSDI*, pages 197–210, 2011.

[5] B. Campbell, M. Clark, S. DeBruin, B. Ghena, N. Jackson, Y.-S. Kuo, and P. Dutta. perpetual sensing for the built environment. *IEEE Pervasive Computing*, 15(4):45–55, 2016.

[6] B. Campbell and P. Dutta. An energy-harvesting sensor architecture and toolkit for building monitoring and event detection. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*, pages 100–109. ACM, 2014.

[7] A. Colin and B. Lucia. Chain: tasks and channels for reliable intermittent programs. *ACM SIGPLAN Notices*, 51(10):514–530, 2016.

[8] A. Colin, E. Ruppel, and B. Lucia. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 767–781. ACM, 2018.

[9] M. E. Harmon and S. S. Harmon. Reinforcement learning: A tutorial. Technical report, WRIGHT LAB WRIGHT-PATTERSON AFB OH, 1997.

[10] J. Hester, K. Storer, and J. Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, page 17. ACM, 2017.

[11] T. Xiang, Z. Chi, F. Li, J. Luo, L. Tang, L. Zhao, and Y. Yang. Powering indoor sensing with airflows: a trinity of energy harvesting, synchronous duty-cycling, and sensing. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, page 16. ACM, 2013.

Work-in-Progress: A Unified Runtime Framework for Weakly-hard Real-time Systems

Hyunjong Choi and Hyoseung Kim
University of California, Riverside
hchoi036@ucr.edu, hyoseung@ucr.edu

***Abstract*—A weakly-hard real-time system is a system that can tolerate a bounded number of timing violations. There have been various assumptions made by prior work on handling deadline-missed instances (jobs) of a task in weakly-hard systems, e.g., terminate it immediately or continue to execute it even after its deadline is missed. However, no prior work has presented a system-level framework to support such options and to quantitatively assess their effects on schedulability. In this paper, we present a *unified runtime framework* to execute weakly-hard tasks with the four major deadline-miss handling options: job abort, delayed completion, job pre-skip, and job post-skip. Our work is applicable to any type of operating systems that support preemptive scheduling with task control blocks. We have implemented our framework in the Linux platform running on Raspberry Pi. We evaluate the performance of each scheme and measure spatial and computational overheads.**

I. Introduction

Weakly-hard real-time systems have been studied to capture the occasional miss of deadlines that a system can tolerate. The common notation of a weakly-hard constraint is in the (m, K) form, which specifies that at most m instances can miss their deadlines among K consecutive instances. A certain level of quality of service can be guaranteed, provided that such a timing violation happens in a known and predictable way.

In the literature of weakly-hard systems, there are various assumptions on how to handle a task's instance (job) when it has missed or is likely to miss the deadline. A job may be terminated immediately when it misses its deadline, i.e., *job abort*. It may continue to execute to completion even if its deadline has passed, i.e., *delayed completion*. A prediction can be made such that only the jobs that are expected to complete by their deadlines are executed, i.e., *job pre-skip*. One may also decide to skip the next job if the current job is executing over the next period, i.e., *job post-skip*. Table I summarizes the classifications of previous weakly-hard studies based on the above assumptions.

Prior work, however, has not considered a runtime framework to instantiate various deadline-miss handling options and has not provided any comparative analysis among those assumptions. Besides, an implementation cost, which is one of the criteria to evaluate applicability in practical systems, has not been thoroughly studied.

This paper presents our work-in-progress effort on developing a runtime framework for weakly-hard real-time systems. Our framework includes systems primitives to support the four aforementioned deadline-miss handling schemes, i.e., *job abort*, *delayed completion*, *job pre-skip*, and *job post-skip*. The current version of our framework focuses on task-level fixed-priority scheduling and has been prototyped in the Linux kernel on Raspberry Pi 3. The proposed framework design approaches can also be applied to any operating system (OS) that uses preemptive task scheduling with task control blocks. We expect that our framework can serve as a basis to build real-time applications with weakly-hard constraints and facilitates the comparison of different weakly-hard schemes on a real platform.

TABLE I: Weakly-hard studies based on the job handlings

Handling scheme	Prior work
Job abort	[6]*, [7]*, [3]
Delayed completion	[5], [8]
Job pre-skip	[6]*, [7]*

**: multiple handling schemes are employed.*

II. Related work

Many weakly-hard studies have assumed the use of the delayed completion scheme, in which a job continues to run although it exceeds its deadline. In overloaded systems, the work in [5] bounds temporary violations of deadlines by using typical worst-case analysis (TWCA) and job arrival curves. The work in [8] also uses the delayed completion scheme and focuses on a taskset with utilization less than or equal to 1.

In [3], the author assumes that the execution of a job is aborted if it does not finish within its deadline. This assumption is to ensure that a delayed job from the previous period does not affect the execution of the next released job. With this assumption, tasksets with utilization more than 1 can be schedulable, but the author has not provided details on the safe recovery of task states after the job abortion.

To manage the overloaded situations, some prior work [6, 7] has considered both the job abort and the job pre-skip schemes. Ramanathan [7] classified jobs into mandatory and optional ones such that only the mandatory jobs are guaranteed to be schedulable in order to reduce the overall loads of the system. The work in [6] also used similar classification approaches, whereas the optional jobs may be executed by checking its eligibility based on slack or predetermined patterns. Recently, Zhishan et al. [4] proposed a new scheme for mixed-criticality systems that drops jobs while guaranteeing the predefined level of performance degradation of low-criticality tasks.

III. System model

Our framework primarily considers task-level fixed-priority preemptive scheduling in a uniprocessor system. For the task model, we assume periodic tasks with weakly-hard constraints.

Task model. Task τ_i is represented as follows:

$$\tau_i := (C_i, D_i, T_i, (m_i, K_i))$$

- C_i: The execution time of each job of a task τ_i.
- D_i: The relative deadline of each job of τ_i ($D_i \leq T_i$).
- T_i: The period of τ_i.
- (m_i, K_i): The weakly-hard constraints of τ_i ($m_i < K_i$). If τ_i is a hard real-time task, $m_i = 0$ and $K_i = 1$.

The j-th job of a task τ_i is denoted as $J_{i,j}$.

Performance metrics. To evaluate the performance of weakly-hard schedulers with different deadline-miss handling schemes, we define the following metrics.

Def. 1. *The effective utilization of a task τ_i, $U_i^e(t)$, measures the ratio of the time used for jobs that have met their deadlines during a given time interval t. Hence, $U_i^e(t) = \frac{C_i \times M_i}{t}$, where M_i is the number of jobs completed by deadline during t.*

The total effective utilization, $U^e(t)$, is thus the sum of the effective utilization of all tasks in the system during a given time interval t, i.e., $U^e(t) = \sum_{i=1}^{N} U_i^e(t)$, where N is the number of tasks and $U^e(t)$ cannot exceed 1.

Def. 2. *The runtime utilization of a task τ_i, $U_i^r(t)$, measures the ratio of the time used for a task that has occupied the processor during a given time t. Hence, $U_i^r(t) = \frac{R_i(t)}{t}$, where $R_i(t)$ is the processor time used by a task τ_i during t.*

The total runtime utilization, U^r, is the sum of the runtime utilization of all tasks in the system during time t, i.e., $U^r(t) = \sum_{i=1}^{N} U_i^r(t)$, and it cannot exceed 1. Also, $U^r(t) \geq U^e(t)$.

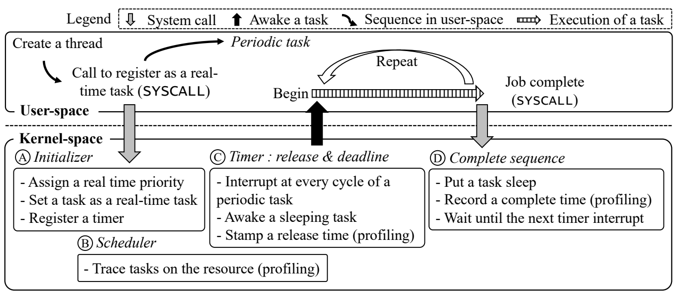
IV. Runtime Framework

This section presents our framework to support the four deadline-miss handling schemes mentioned in Section I: *job abort*, *delayed completion*, *job pre-skip* and *job post-skip*. We begin with a fundamental runtime mechanism for periodic execution of tasks, and then present the detailed design of each scheme based on it.

A. Periodic execution support

Fig. 1 illustrates the overview of our runtime mechanism for periodic task execution, which is organized as *user space* and *kernel space*. In the user space, a task can request specific actions to the OS kernel as needed, e.g., registering a task as a real-time task or putting it in sleep mode when the current job finishes execution. In the kernel space, our runtime consists of four core modules: initializer, scheduler, timer, and complete sequence. We detail the functions of each module as follows:

- *Initializer*: The system registers a task as a periodic real-time task by assigning real-time priority, and creating release and deadline timers for this task in Ⓐ.


Legend
System call
Awake a task
Sequence in user-space
Execution of a task
Create a thread
Periodic task
Call to register as a real-time task (SYSCALL)
Repeat
Begin
Job complete (SYSCALL)
User-space
Kernel-space
Ⓐ Initializer
- Assign a real time priority
- Set a task as a real-time task
- Register a timer
Ⓒ Timer : release & deadline
- Interrupt at every cycle of a periodic task
- Awake a sleeping task
- Stamp a release time (profiling)
Ⓓ Complete sequence
- Put a task sleep
- Record a complete time (profiling)
- Wait until the next timer interrupt
Ⓑ Scheduler
- Trace tasks on the resource (profiling)
Fig. 1: Runtime mechanism for periodic task execution

- *Scheduler*: In Ⓑ, a scheduler determines the *readiness* of periodic tasks based on the deadline-miss handling scheme in use, and records task execution traces and resource usage.
- *Timer*: There are two timers involved for each periodic task. First, the release timer (Ⓒ) fires at the beginning of each period of the task. The deadline timer fires when the task has not completed its job execution and is still in running mode, different sequences are followed based on the deadline-miss handling used.
- *Job completion sequence.*: Once the task finishes its job execution, it makes a `job_complete` system call to the kernel (Ⓓ). The system then puts the task into sleep mode so that it waits until the next job is released by the timer.

Note that this mechanism shown in Fig. 1 is generally applicable to most OSs supporting preemptive task scheduling.

B. Job abort

In this scheme, a job is terminated immediately if it misses its deadline. This scheme can be beneficial in the case where a deadline-missed job will no longer need to run as the remainder of its execution does not yield any gain.

However, the termination of a running job is not trivial in the implementation. When a task misses its deadline, it needs to be rolled back to its previous state so that its next job can safely and correctly execute in the next period. This rollback mechanism is typically done by creating a checkpoint [1].

Task rollback. There are two types of rollback approaches we may consider. The first is task-level checkpointing, where each task creates its own checkpoints as part of execution and implements a handler to recover from the stored checkpoints. The second approach is system-level checkpointing, where the OS or middleware creates each task's checkpoint, e.g., by storing all memory pages recently modified, and recovers the task state when needed. In this work, we focus on the task-level approach due to its lower overhead that can be done in three steps as follows:

- **Step 1. Store a checkpoint**: Since our rollback technique is achieved in a task-level, a checkpoint is stored at the beginning of a task execution. We used `sigsetjmp` in the standard C library to save a program counter (PC) and a stack pointer (SP)
- **Step 2. Notify a deadline miss to the user space**: If the job has missed its deadline, the kernel notifies the task by sending a signal.

- **Step 3. Recover from the checkpoint**: By the signal generated in Step 2, the signal handler of the task is triggered so that the PC and SP are recovered from the stored checkpoint by using `siglongjmp`.

TABLE II: Taskset 1

Tasks	T (Period) [ms]	C (WCET) [ms]	(m, K)	Priority
τ_1	65	35	(2,4)	High
τ_2	125	35	(2,4)	Middle
τ_3	200	35	(2,4)	Low

Example. We have implemented this scheme in the Linux kernel v4.9.76 running on Raspberry Pi 3, and tested the operation of the job abort scheme by running a taskset given in Table II. In Fig. 2, τ_1 and τ_2 are always schedulable while $J_{3,1}$ and $J_{3,4}$ of Task 3 do not meet their deadlines.

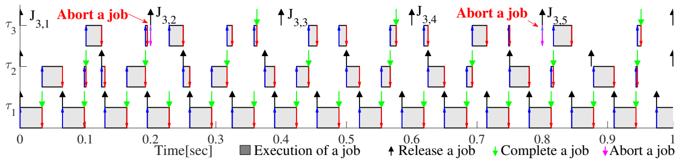


Fig. 2: Job abort

C. Delayed completion

The delayed completion scheme allows a deadline-missed job to continue to run until it completes. This scheme is effective when the quality of service can be improved by the execution of remainder of a deadline-missed job.

Under this scheme, however, if a job continues to run over its deadline, the next released job is delayed by the execution of the previous job. Thus, this scheme does not get benefit from the weakly-hard concept in overloaded situations, i.e., taskset with total utilization is more than 1.

In order to realize the scheme, the task is put in sleep mode only when the job just completed is the latest released job. This is checked in by the job completion sequence module (Ⓓ in Fig. 1).

Example. As shown in Fig. 3, deadline-missed jobs are running until it completes its execution under this scheme.

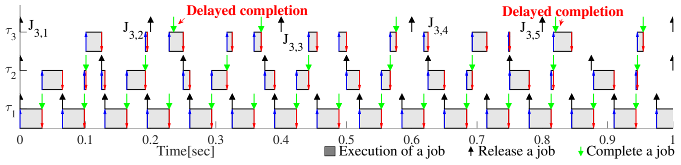


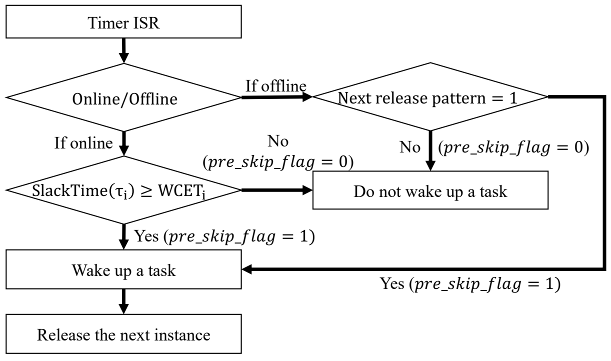
Fig. 3: Delayed completion

D. Job pre-skip

The job pre-skip scheme determines whether to execute a job or not at its release time (the release timer Ⓒ of Fig. 1). A decision for the execution can be made in either online or offline. In case of the online approach, the system can use the slack time of a task at the moment of its job release. In the offline approach, a predetermined execution pattern is used, e.g., `1010` where `1` means execution and `0` means skip.

However, there are two major drawbacks. The first is the runtime overhead which may be high because the scheduler needs to check the slack time (online) for job execution. Moreover, especially when the average-case execution time is much lower than the WCET, the scheduler may unnecessarily skip jobs, which results in processor underutilization.

In this scheme, the release timer (Ⓒ) is the major module to be modified to enable the pre-skip scheme. The overall sequence of the modified release timer is depicted in Fig. 4.



Timer ISR
Online/Offline
If offline
Next release pattern = 1
If online
No
$(pre\_skip\_flag = 0)$
No $(pre\_skip\_flag = 0)$
SlackTime(τ_i) ≥ WCET$_i$
Do not wake up a task
Yes $(pre\_skip\_flag = 1)$
Wake up a task
Yes $(pre\_skip\_flag = 1)$
Release the next instance

Fig. 4: Timer sequence in job pre-skip scheme

Example. Under the offline approach, a predefined pattern of `1010` is applied to all tasks so that every other instance is executed as depicted in Fig. 5. On the other hand, under the online approach, jobs $J_{3,2}$, $J_{3,3}$, and $J_{3,5}$ are executed based on slack calculation.

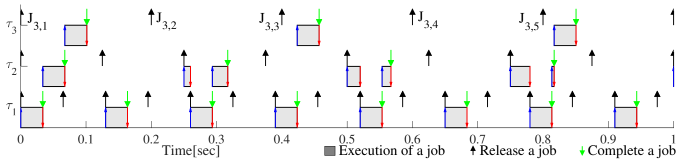


Fig. 5: Job pre-skip (pattern)

E. Job post-skip

In this scheme, the scheduler allows a deadline-missed job (released at j^{th}) continue to run, but it always skips the next released job (released at $j+1^{th}$) and keeps track of the index of the job. Under this scheme, jobs are discarded occasionally, resulting in degradation of the quality of service of a system.

Example. Under the job post-skip scheme, $J_{3,2}$ and $J_{3,5}$ are skipped because the previous jobs have violated their deadlines and affected the execution of the next released jobs, as shown in Fig. 6.

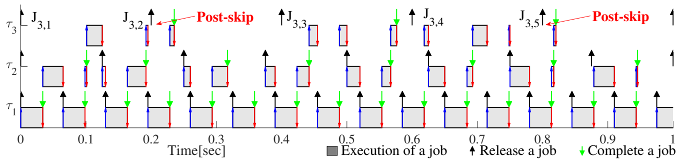


Fig. 6: Job post-skip

V. Evaluation

In this section, we evaluate the proposed framework in the Linux kernel running on Raspberry Pi 3 (Quad Cortex

A53 @ 1.2GHz). The evaluation consists of two parts: the measurement of computational overheads and the case study.

Overheads. For the computational overhead of our framework, it is worth noting that the delayed completion and post-skip schemes do not cause any additional cost other than those for the periodic execution mechanism shown in Section IV-A. However, under the job abort and job pre-skip schemes, there are the following four major sequences that can cause extra runtime overhead:

- `sigsetjmp` (job abort): the cost in the user space to create a checkpoint
- `siglongjmp` (job abort): the cost in the kernel space to send a signal to the user-space task
- Slack (job pre-skip): the cost of calculation of the slack
- Pattern (job pre-skip): a transition cost of pointing to the next element in the pattern

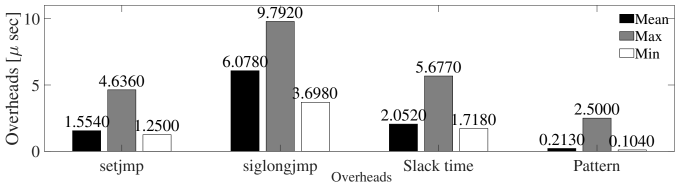


Fig. 7: Overheads of taskset

Fig. 7 summarizes the overhead measurement on Raspberry Pi 3. As can be seen, `siglongjmp` for the rollback mechanism is the most costly operation. However, they are acceptably small in μs units, compared to the WCET and periods typically denoted in ms units.

Case study. For case study, we have selected a taskset given in Table III where its total utilization is higher than 1.

TABLE III: Taskset 2 [6]

Tasks	T (Period) [ms]	C (WCET) [ms]	Skip parameter
τ_1	6	1	2
τ_2	7	4	2
τ_3	19	5	2

This taskset is not schedulable under any conventional fixed-priority scheduler. However, if the RM-RTO[1] algorithm of [6] is used, the job execution pattern of each task is determined as either `10` or `01` and this pattern can be realized by the job-skip scheme. Fig. 8 shows the result of dynamic failures[2] of

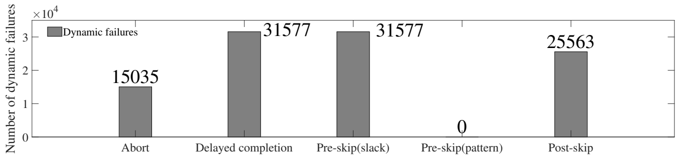


Fig. 8: Dynamic failures of τ_3 (total 31578 jobs released)

τ_3. As can be seen, the number of dynamic failures under the pre-skip scheme (pattern) is zero, meaning that τ_3 satisfies its weakly-hard constraint. However, all the other schemes suffer from a high number of dynamic failures.

[1]RM-RTO stands for Rate Monotonic Red Task Only.

[2]A task experiences more than *m* deadline misses in a window of *K* jobs.

Fig. 9 shows the observed effective and runtime utilization values. The abort scheme shows the highest effective utilization and the pre-skip scheme (pattern) has the lowest. This is because the pattern-based scheduling pessimistically skips the execution of higher-priority tasks even if there is enough processor time to use.

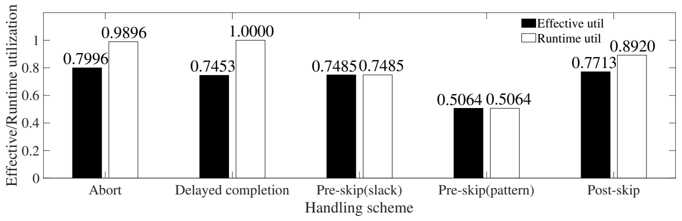


Fig. 9: Total effective and runtime utilization

VI. Conclusion

We proposed a unified runtime framework for multiple deadline-miss handling schemes in weakly-hard real-time systems. The framework has been implemented in the Linux kernel on Raspberry Pi with very low overhead, but it is easily applicable to other OSs using fixed-priority preemptive schedulers. Experimental results show that, depending on the deadline-miss handling scheme used, the number of violations of weakly-hard constraints as well as utilization metrics can vary significantly for the same taskset and a different trend can be observed for other tasksets. These results pave an interesting research direction to investigating weakly-hard tasks under diverse experimental conditions and new analysis techniques. In our current implementation, all tasks in the system are governed by the same deadline-miss handling scheme. This is in accordance with the assumptions of prior work, but we expect that allowing each task to have a different handling scheme would increase resource efficiency and design flexibility. Furthermore, our framework can be extended beyond task-level fixed-priority scheduling, e.g., a job-class-level fixed-priority scheduler [2] to be presented in RTAS 2019. It can also be used as an assessment tool for the issues that have not studied much in the weakly-hard context such as an inter-task dependency, shared resources, multicore systems, and temporal interference from contention in cache and main memory.

References

[1] M. Asberg et al. Resource sharing using the rollback mechanism in hierarchically scheduled real-time open systems. In *IEEE RTAS*, 2013.

[2] H. Choi, H. Kim, and Q. Zhu. Job-Class-Level fixed priority scheduling of weakly-hard real-time systems. In *IEEE RTAS*, 2019.

[3] J. Goossens. (m, k)-firm constraints and DBP scheduling: impact of the initial k-sequence and exact schedulability test. 2008.

[4] Z. Guo et al. Uniprocessor mixed-criticality scheduling with graceful degradation by completion rate. In *IEEE RTSS*, 2018.

[5] Z. A. H. Hammadeh et al. Budgeting under-specified tasks for weakly-hard real-time systems. In *ECRTS*, 2017.

[6] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *RTSS*, 1995.

[7] P. Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Trans. on Par and Dist. Syst.*, 10(6):549–559, Jun 1999.

[8] Y. Sun and M. D. Natale. Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks. *ACM TECS*, 2017.

Memory Mapping Analysis for Automotive Systems

Robert Höttger, Lukas Krawczyk, Burkhard Igel
IDiAL Institute
Dortmund University of Applied Sciences and Arts
Dortmund, Germany
{robert.hoettger, lukas.krawczyk, igel}@fh-dortmund.de

Olaf Spinczyk
Embedded Software Systems
Osnabrück University
Osnabrück, Germany
olaf.spinczyk@uos.de

***Abstract*—While embedded real time software is predominantly analyzed regarding task response times, memory mapping is often assumed to be static and is either defined by the system designer or by default set to affinity locations of software components mainly working with the respective memory.**

This paper's work analyses memory mapping from several viewpoints and proposes solutions to mitigate temporal costs introduced by accessing data distributed across NUMA architectures. Therefore, event chains, response times, activation patterns, contention, as well as a variety of hardware properties such as memory type, memory size, memory access type, and memory affinities are taken into account. Evolutionary Algorithms, Constraint Programming, and a dedicated heuristic are outlined that minimize costs influenced by the data to memory mapping.

Additionally, label mapping costs are analyzed regarding ECU networks consisting of buses, hardware hierarchies, and arbitrary connections of ports and hardware entities. This analysis is accompanied with an assessment of typical domain related formal timing verification methods.

It is expected that the presented label mapping solutions significantly reduce overheads produced by network communication as well as contention effects of memory accesses.

***Index Terms*—AMALTHEA, APP4MC, AUTOSAR, Memory Mapping, Automotive**

I. Introduction

The mapping of code, constants, and shared variables affects the timing properties of real-time systems especially in distributed, heterogeneous, and mixed-critical environments such as the automotive domain. System designers may overlook optimal mapping solutions due to a variety of constraints emerging from safety, affinity, timing, reliability, fault-tolerance, and similar demands. Since the modeling of comprehensive system environments is common practice in the automotive industry, new technologies can cope with problems automatically without the need of manual error prone processes and address problems on a much broader and abstract level. For example, centralization effects can be investigated in early design phases without the need of hardware validation, actual software implementation, or simulation via formal model analysis.

AUTOSAR[1] is the major architecture used by car manufacturers and different tier suppliers in the automotive industry. AMALTHEA[2] comes with the open source APP4MC[3] platform and is an AUTOSAR compliant model that features a variety of extensions that are necessary to investigate timing, feasibiliy as well as functional and non-functional behavior of automotive software. Several commercial tools exist to analyze software distribution potentials, reliability, response time, safety, or other quality attributes from companies such as Inchron, Symtavision, Vector and others. Memory mapping, that is the major focus in this paper, highly influences such quality attributes and forms a major challenge in the design process of automotive systems.

This paper outlines challenges and solutions for mapping memory in non uniform memory access (NUMA) architectures. The distribution of data across different memories of different types considers access types, access costs, access rates, contention, blocking, and various hardware properties. Finding the optimal memory mapping in general is a NP-complete problem [8].

II. Related Work

The Real-Time Calculus [12] (RTC) calculates end-to-end delays, buffer requirements, or throughput of networked systems as a Matlab toolbox. While RTC is able to assess real-time properties, deployment feasibility, and more, it neither assesses nor provides adaptations to memory mapping.

Schneider [11] outlines challenges and requirements of memory management units for automotive ECUs and investigates approaches of the general purpose world regarding protection granularity, memory efficiency, and real-time behavior. The publication clearly shows the importance and relevance of sophisticated memory management for the automotive domain.

The PhD thesis by Kumar [8] presents the data layout problem which corresponds this paper's label mapping problem and presents different formulations to solve the problem are given. However, this paper targets different optimization goals than minimizing memory stalls, conflicting accesses, and off-chip memory accesses.

In [4], Broquedis et al. advance the OpenMP runtime to dynamically perform thread an memory placement to provide dynamic load distribution under application requirements and hardware constraints such as affinities. In contrast, this paper's approaches account memory utilization offline in order to provide a static mapping which complies to the AUTOSAR standard and consider timing and network constraints specifically.

[1]Automotive Open System Architecture www.autosar.org, accessed 01.2019
[2]provided by the AMALTHEA model http://eclip.se/fn, accessed 02.2019
[3]www.eclipse.org/app4mc, accessed 01.2019

Antony et al. [1] account various memory placements along with access latencies and memory bandwidth assessments on different NUMA platforms running Solaris and Linux. Although the benchmarks do not cover specific real-time properties, results show that local placement is not always the best strategy and sophisticated mapping significantly improves application performance.

Avissar et al. present in [2] a compiler based approach to automatically partition data among memory units using binary ILP. The used system model is close to this paper's model and corresponds the minimization of label mapping costs lmc (cf. Table I). The contribution of this paper goes beyond [2] via considering AUTOSAR related constraints, analyzing response times, and incorporating sophisticated network structures, e.g. the Controller Area Network (CAN) bus.

Along with the real-time community, there have further been label mapping solutions along with [3], [5], and others, however, none of those cover the specific structure given in this paper (cf. Table I).

III. System Model

According to the AUTOSAR model, software is predominantly comprised by runnables grouped into tasks and label accesses that represent access to shared or private data. Grouping runnables into tasks and a more detailed task model has been covered in previous work such as [7] and reduces the subsequently described approaches' complexity due to lower amounts of entities featuring combined properties. Figure 1 shows a typical constellation of an ECU subset in the automotive context. The entire ECU network usually consists of

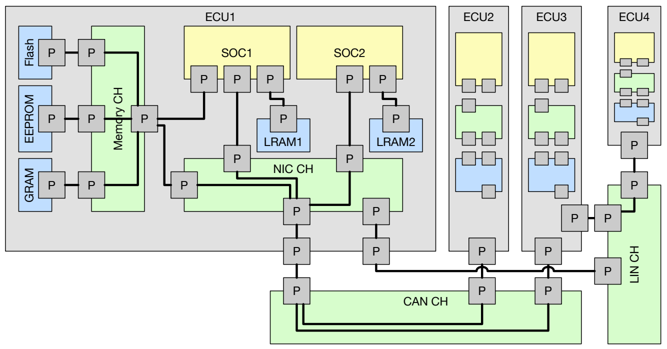


Fig. 1. ECU network example with Ports, Connections, Memories, Connectionhandler, and HWStructures

significantly more ECUs but is omitted here for comprehension purposes. The same holds for detailed views of ECUs 2 – 4. The example is constructed by means of the AMALTHEA model and consists of ports, internal and external connections, connection handlers, memories, and computational elements on various abstraction levels, e.g. DSPs, CPUs, GPUs, microcontrollers, etc. Connections within a connection handler are denoted as internal connections. Ports can be defined as initiators or responders and can implement interfaces such as CAN, Flexray, LIN, MOST, Ethernet, SPI, I2C, AXI, ABH, APB, or SWR. ECU1 features five memory instances of four different types as well as two processors. Response times are predominantly influenced by the access times to different memory (e.g. global / local RAM, NVRAM, ROM, EEPROM, Flash etc.) as well as the size, access rate, and access type of variables (i.e. labels) that are accessed across the platform. For instance, the processor on SOC1 can take between 2 and 3 cycles to access LRAM1, whereas access to the Flash memory cache can be 20 cycles or more. While most of the existing research assumes static worst case memory access costs driven by contention, blocking, and scheduling, a sophisticated network as shown in Figure 1 comprises a broad heterogeneous NUMA structure that needs to consider the dynamics of protocols, connection handler acceleration properties, access patterns, and port interfaces in order to make memory mapping as efficient as possible.

The system model of Table I outlines properties used by solutions presented in Section IV.

TABLE I
System Model

Symbol	Description	Index / Calculation
r	Runnable	index $= n$
τ	Task	index $= p$, ordered by priority such that τ_0 is the highest priority task
d	Deadline	d_p = deadline of task τ_p
pu	Processing Unit	index $= i$
rm	runnable to pu mapping	$rm_{n,i} = \begin{cases} 1 \text{ if } r_n \text{ mapped to } pu_i \\ 0 \text{ otherwise} \end{cases}$
m	Memory	index $= j$
ms	Memory size	ms_j = memory size of m_j
rml	Read memory latency	$rml_{i,j} = rml$ between pu_i and m_j
wml	Write memory latency	$wml_{i,j} = wml$ between pu_i and m_j
l	Label	index $= k$
ls	Label size	ls_k = label size of l_k
a	Activation	a_n = activation rate (period) of r_n, a_p = activation of task τ_p
rl	Read labels	rl_n = read labels of r_n
wl	Written labels	wl_n = written labels of r_n
lm	Label to m mapping	$lm_{k,j} = \begin{cases} 1 \text{ if } l_k \text{ mapped to } m_j \\ 0 \text{ otherwise} \end{cases}$
lmc	Label map. cost	see Eq. 1; $lmc_{k,j}$ = cumulated costs for mapping l_k to memory m_j
C	CAN message transmission time	index of CAN messages = c, see Eq. 4
R	Response time	see Eq. 5, Eq. 7
W	Queuing delay	see Eq. 6

Each AUTOSAR runnable refers to an activation and contains label write or read accesses to arbitrary labels. Consequently, the total label mapping cost is derived from the binary label mapping $lm_{k,j}$, activation rates a_n, read and written labels rl_n, wl_n, runnable mapping rm (to processing units), and read as well as write latencies between processing units and memories wml, rml. The latter is of major importance, since those latencies comprise sophisticated networks as shown in Figure 1.

The overall label mapping cost calculation is given in Eq. 1

and defines one of the mapping process' optimization goals.

$$\begin{aligned} lmc &= \sum_k lmc_{k,j} \\ &= \sum_n \left(\sum_{rl_n} (a_n \cdot rml_{i,j}) + \sum_{wl_n} (a_n \cdot wml_{i,j}) \right) \\ &\quad \text{with } rm_{n,i} = 1 \end{aligned} \tag{1}$$

In order to derive $wml_{i,j}$ and $rml_{i,j}$, i.e. write and read latency values, ports, connections, interfaces, bit width of connections, label sizes, and protocol properties must be considered. For instance, when considering the CAN protocol, worst case latencies must be derived for the messages passed between ECUs. Since tasks can have deadlines greater than their periods [10], the worst case release times of a task may be shifted to their next period. Typically, the start of transmitting a CAN message is likely to occur at the end of a tasks execution by e.g. following the implicit communication paradigm. Accordingly, we need to assume that the release times of CAN messages are shifted as well. As a result, we consider the level-c busy period for CAN messages, i.e. the maximum consecutive amount of time the CAN network is occupied by messages that have an equal or greater priority then message c.

An analytic approach for determining worst case response times in CAN networks in consideration of busy-periods is presented in [9]. After ensuring that the bus utilization is less than one ($\sum_c \frac{C_c}{T_c} \leq 1$), they determine the duration t_c of a level-c busy period by the recurrence relation in Eq. 2.

$$t_c = B_c + \sum_{\forall v \in hp(c) \cup c} \left\lceil \frac{t_c + J_v}{T_v} \right\rceil C_v \tag{2}$$

The minimal interval between two occurrences of the v-th higher priority (hp) message is represented by T_v and equals the sending tasks period for periodically sent messages resp. their minimal inter-arrival time in sporadic send messages. As CAN messages are naturally non-preemptive, the blocking time B_c introduced by lower priority (lp) messages on the channel needs to be considered. Since at most one lower priority message can block the channel, the blocking time equals the highest transmission time C_c among all lower priority messages.

Eq. 3 defines the payload of a CAN message s_c as the sum of label sizes being exchanged and consequently defines the communication.

$$\begin{gathered} s_c = \sum_{k: l_k \in (rl_{\hat{n}} \wedge wl_{\check{n}})} ls_k \\ \text{with } \hat{i} \neq \check{i} \text{ for } rm_{\hat{n},\hat{i}}, rm_{\check{n},\check{i}} \end{gathered} \tag{3}$$

The payload calculation is accompanied by the assumption that the communicating entities (here, $r_{\hat{n}}$ and $r_{\check{n}}$) are mapped to different processing units ($pu_{\hat{i}}, pu_{\check{i}}$, $\hat{i} \neq \check{i}$) that are connected through a CAN network.

The maximum transmission time C_c depends on the identifier format and is derived as in Eq. 4, with τ_{bit} being the transmission time for a single bit. Usually, τ_{bit} can be derived from the CAN network's baud rate. The constants are obtained from CAN properties such as bit stuffing that includes CRC bits, error frames, as well as control, arbitration and data fields. A more detailed outline can be found at [9].

$$C_c = \begin{cases} (55 + 10 s_c)\, \tau_{bit} & \text{for 11-bit identifiers} \\ (80 + 10 s_c)\, \tau_{bit} & \text{for 29-bit identifiers} \end{cases} \tag{4}$$

The worst-case response time $R_c(q)$ for the q-th instance of a message c is obtained as stated in Eq. 5 by summing up the queuing jitter J_c, the queuing delay $W_c(q)$, and the transmission time C_c. Subtracting qT_c, with T_c being the period of message c results in the relative worst case response time for the q-th instance. The queuing jitter J_c is derived from hardware profiling. An analytical jitter derivation is omitted here.

$$R_c(q) = J_c + W_c(q) - qT_c + C_c \tag{5}$$

The queuing delay $W_c(q)$ for the q-th instance of c is determined using the recurrence relation in Eq. 6.

$$W_c(q) = B_c + qC_c + \sum_{\forall v \in hp(c)} \left\lceil \frac{W_c + J_v + \tau_{bit}}{T_v} \right\rceil C_v \tag{6}$$

Finally, the worst case response time of a message c is determined by reducing the individual values $R_c(q)$ for all instances to their maximum value.

In addition to the metrics of Table I that define the quality and responsiveness of automotive memory mapping, offload copy operations can be further incorporated for advanced computing environments using accelerators or GPUs. Therefore, label read operations must be extended with additional write operations and vise versa. Such consideration forms a potential extension of this work's current state.

Another important assumption for calculating response times as well as event chain latencies across a networked environment is the incorporation of extended instruction sets respectively ticks in terms of AMALTHEA 0.9.3. For instance, the amount of ticks required for executing a complex image processing application significantly differs when being executed on a GPU vs CPU. Therefore, this paper uses traditional response time analysis for mixed preemptive fixed priority tasks using recurrence relation [10] (based on Eq. 7) that incorporates mapping decisions and varying instruction sets correspondingly.

$$R_p = C_p + \sum_{q \in hp(p)} \left\lceil \frac{R_p}{T_q} \right\rceil \cdot C_q \tag{7}$$

Here, C_p and C_q provide different values in regard to the processing unit they are mapped to. Additionally, the worst case response times are validated to be smaller than the task deadlines $\forall p : R_p < d_p$.

IV. Memory Mapping

The label mapping has been implemented as a heuristic, an evolutionary algorithm (EA), and a constraint programming (CP) approach. The heuristic is a straight forward greedy approach that maps labels ordered by size to the memory featuring the lowest $lmc_{k,j}$ value. If a memory is full, the memory with the next lowest $lmc_{k,j}$ value is chosen.

The EA uses the jenetics java library[4]. The fitness function is presented in Eq. 8 where S denotes a chromosome, $\overline{R_p}$ the mean task response time, and $\overline{R_c}$ the mean CAN message response time. Additionally, each fitness is accompanied by Eq. 9 in order to ensure the memory size constraint. The different entities of the fitness function require additional factors in order to equally influence the overall fitness.

$$\forall S : f_S = \sum_{l} lmc_{k,j}(S) + \overline{R_p}(S) + \overline{R_c}(S) \tag{8}$$

$$\forall m_j : \sum_{l:lm_{k,j}=1} ls_k \leq ms_j \tag{9}$$

The CP approach to the label mapping problem uses the choco library[5] and consists of a boolean variable matrix for the label to memory mapping (i.e. a binary lm representation), the cost variable array $lmc_{k,j}$, as well as the memory load variable array ml, and the tc variable. All these variables are distinct for each solution. Solutions are required to satisfy the following five constraints. A sum constraint on each array of the lm matrix ensures that each label is exactly mapped once. A scalar constraint across lm and $lmc_{k,j}$ ensures that the mapping costs are derived from the label mapping and according label access costs for each label based on Eq. 1. A sum constraint over $lmc_{k,j}$ calculates lmc. Finally, a scalar constraint calculates the sum of label sizes assigned to a memory and ensures along an arithmetical constraint that this sum does not exceed the memory size. The latter is applied to each memory. Alternatively, the latter arithmetical constraint can be replaced by a binpacking constraint.

V. Results and Assessments

Given the above mentioned methodologies to map labels to memory of the hardware, results are assessed by

1) the average task response time $\overline{R_p}$
2) the network load $nl = \sum_v C_v$, and
3) the total label access cost lmc
4) the cumulated latencies of event chains $\check{l} = \sum_{ec} WCL_{ec}$ with WCL denoting the worst case latency of an event chain considering the label mapping.

The description of event chains and their latency calculation is given in [6]. While the CAN response time analysis has been implemented as well as the rudimental evolutionary and constraint programming approaches, some integration work (especially according to minimizing WCL_{ec}) is still necessary in order to combine the label mapping with temporal coherency considerations. First results show that the EA generates better results than the heuristic and the CP approach. Although the CP approach has powerful paradigms to model various constraints, its implementation requires careful analysis in order to keep the resolution time appropriate. Measured results (plots) of the approaches are expected soon.

[4]http://jenetics.io, accessed 02.2019

[5]http://www.choco-solver.org, accessed 02.2019

VI. Conclusion

The presented description of typical memory mapping constraints under a variety of automotive specific hardware properties as well as the consideration of CAN network properties as an example, provides valuable insights into modern memory mapping for networked, heterogeneous, mixed-critical, real-time systems. Although results are not available yet, the formal outline and description of the approaches present important characteristics and foundations for memory mapping analysis in the automotive domain.

References

[1] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *High Performance Computing - HiPC 2006*, pages 338–352, 2006.

[2] Oren Avissar, Rajeev Barua, and Dave Stewart. Heterogeneous Memory Management for Embedded Systems. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '01, pages 34–43. ACM, 2001.

[3] Alessandro Biondi, Paolo Pazzaglia, Alessio Balsini, and Marco Di Natale. Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores. *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2017.

[4] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective. In *Proceedings of the 5th International Workshop on OpenMP*, IWOMP '09, pages 79–92, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Junchul Choi, Donghyun Kang, and Soonhoi Ha. A Novel Analytical Technique for Timing Analysis of FMTV 2016 Verification Challenge Benchmark. *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 7, 2016.

[6] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication Centric Design in Complex Automotive Embedded Systems. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76, pages 10–20, 2017.

[7] Robert Höttger, Lukas Krawczyk, and Burkhard Igel. Model-Based Automotive Partitioning and Mapping for Embedded Multicore Systems. In *International Conference on Parallel, Distributed Systems and Software Engineering*, volume 2, pages 2643–2649, January 2015.

[8] T.S. Rajesh Kumar, R. Govindarajan, and C.P. Ravikumar. On-chip Memory Architecture Exploration Framework for DSP Processor-based Embedded System on Chip. *ACM Trans. Embed. Comput. Syst.*, 11(1):5:1–5:25, April 2012.

[9] Gerardine Immaculate Mary, Z. C. Alex, and Lawrence Jenkins. Response Time Analysis of Messages in Controller Area Network: A Review. *Journal of Computer Networks and Communications*, 2013.

[10] Ignacio Sanudo, Paolo Burgio, and Marko Bertogna. Schedulability and Timing Analysis of Mixed Preemptive-Cooperative Tasks on a Partitioned Multi-Core System. In *Proceedings of the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS'16), Toulouse, France*, 2016.

[11] Jörn Schneider. Why current Memory Management Units are not suited for Automotive ECUs. In *Automotive - Safety & Security*, volume 210 of *LNI*, pages 99–114. GI, 2012.

[12] Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox, 2006. Online: http://www.mpa.ethz.ch/Rtctoolbox.

QRONOS: Towards Quality-Aware Responsive Real-Time Control Systems

Peter Ulbrich*, Maximilian Gaukler†
Department of Computer Science, *Distributed Systems and Operating Systems
Department of Electrical Engineering, †Automatic Control
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

***Abstract*—A key design goal of safety-critical control systems is the verifiable compliance with a specific quality objective in the sense of the quality of control. Corresponding to these requirements, the underlying real-time operating system has to provide resources and a certain quality of service, mainly in the form of timing guarantees.**

For the design of efficient real-time control systems, considering only the quality of service is insufficient as it is firmly intertwined with the quality of control: First of all, the actual timing has a significant and nontrivial influence on the quality of control. Vice versa, the temporal precision required to provide a certain quality of control may vary considerably with environmental situation and disturbance. Consequently, quality-of-service requirements are not fixed but may vary depending on the execution context.

We present our ongoing work on quality-aware adaptive real-time control systems, addressing three challenges: evaluating quality of control under consideration of varying timing, static worst-case verification, and quality-aware scheduling at runtime.

I. Introduction

Compliance with an application-specific physical specification is a primary design objective of real-time control systems: in a vehicle, this is, for example, to keep lane in a centimeter tolerance range. Further improvement (i.e., millimeter accuracy) does not lead to further increase in specification compliance or general benefit. Accordingly, from a control-theoretical point of view, the system must be designed and assessed to provide a sufficient Quality of Control (QoC) under all possible environmental conditions (e.g., wind). Typically, the QoC is quantified using a quadratic cost function

$$J = x^T Qx + u^T Ru$$

based on the state error x and the control-signal u: small deviation from the desired state $x = 0$ and small actuation correspond to minimum cost J and therefore maximum QoC.

Control systems periodically sample the state of the physical system via sensors, compute the required control signal, and send it to the actuators. Due to this close connection to the outside physical world, real-time control is particularly sensitive to timing variations: In the example of a moving car, sampling the position a later time results in a different value because the car has continued moving. This measurement deviation may reduce the precision of a lane keeping system.

In general, any deviation from the assumed temporal properties may negatively impact the QoC [1]–[3]. Thus, the real-time operating system is tuned for accurate timing of computation and input/output to provide an appropriate Quality of Service (QoS) to the control application running on top. Here, accurate refers to common assessment criteria such as deadline adherence or periodicity (i.e., absence of jitter). In practice, the prevailing point of view is that overall QoC should be optimized by maximizing the QoS, which boils down to tightening temporal bounds [2], [4].

In contrast, current trends in real-time systems foster a well-directed renouncement from this rigid interpretation by moving away from achieving the best possible QoS towards one that is good enough: approaches such as dynamically reconfigurable systems or mixed-criticality scheduling trade accuracy to boost average performance while easing system design as well as worst-case handling. For example, mixed-criticality scheduling [5], [6] provides multiple criticality levels, each with the expectation of a certain quality. Such approaches are, however, typically limited to QoS-guarantees for each criticality level (e.g., control tasks may change timing or even be omitted) . Consequently, it is assumed that there is a static mapping between the QoS and the actually relevant QoC. This static assumption is, for example, also shared by feedback scheduling techniques [1], [7], [8]. Ultimately, deadlines may even be intentionally violated for runtime adaptivity. A vivid example is weakly-hard scheduling of control tasks [9] such that in any window of m execution periods deadlines only have to be met for at least $n < m$ times. In summary, control applications will be faced with more dynamic real-time computing systems, whose timing behavior will be less predictable than it used to be.

Although environmental conditions and QoS (i.e., deviations from the assumed input/output timing) are both determining factors for the QoC, the latter are neither typically considered in the traditional design process nor is the relationship between QoC and QoS trivial.

In previous work [10], [11], we showcased that the dynamic behavior caused by varying timing (QoS) can be counterintuitive. Figure 1 illustrates an example of this effect on the controller of an inverted pendulum. The theoretical framework will be presented later, whereas here we will focus on the results from a real-time systems perspective. The system is subject to varying input and output timing ($\Delta t/T$), that is reading the sensor and writing the actuator values is not performed periodically as assumed during controller design, but with a certain jitter. At first ($t < 10$), the system is operated without delays. Increasing the actuation delay at $t = 10$ has a limited immediate effect but instead causes a gradual increase in cost J (i.e., decrease in QoC). Upon switching back to better timing at $t = 20$, a short-time adverse effect occurs before the costs have returned to acceptable levels at $t = 21$. A static approximation of QoC as a function of the current timing, an assumption often underpinning embedded control systems design [12]–[14], fails to describe these memory-like behaviors, in which the QoC also depends on the history: At $t = 10$ and $t = 19$, the timing is the same, however the QoC is completely different.

Additionally, the influence of QoS on QoC varies between different various sensors and actuators, which is important for the design: In this example the sensor delay ($t = 30 \dots 40$) has less impact than the actuator delay, which suggest that the actuator should be given a higher priority than the sensor.

Existing approaches to evaluate the QoC under consideration of the actual runtime behavior, such as JITTERBUG [15], typically operate on stationary scenarios. In the example of a car, this translates to constant driving conditions and a fixed level of criticality for the control tasks. This means that dedicated QoS levels (timing conditions) are considered individually and the effects are only evaluated time-averaged. Therefore, the aforementioned behavior at transitions between different conditions cannot be analyzed.

II. Problem Statement

In summary, control applications will be faced with more dynamic real-time computing systems, whose timing behavior will be less predictable than it used to be. Runtime adaptivity and scheduling typically focus on QoS bounds, disregarding the susceptibility of control applications to timing variations. At the same time, verifiable compliance with a specific QoC is a key design goal in many settings. Consequently, any stability verification has to factor in non-perfect timing, which is, as illustrated by our example, a non-trivial task.

We identified the primary problem to be the nontrivial mapping of QoS and QoC as well as the fundamentally different approaches to the development and verification for control and real-time systems.

In this paper, we therefore address three challenges to ease the design of adaptive yet verifiable real-time control systems:

(1) Evaluation of a time-dependent QoC for varying sensor/actuator timing in adaptive real-time systems. (2) Static analysis[1] and verification of feedback control systems under consideration of timing variations. (3) A real-time executive that saves resources by adapting QoS, but still respects application-specific QoC goals.

III. The QRONOS Approach

We present our vision on the design and verification of adaptive real-time control systems with non-deterministic input and output timing. These complex real-time systems with multiple applications and controllers can significantly profit from

[1]Note to readers with a control systems background: The term *static analysis* from software engineering refers to analysis which happens "offline" before a program or system is run, in contrast to dynamic analysis. Despite its name, static analysis does indeed consider the system dynamics (transient behavior); it should not be confused with "analysis of the stationary case".

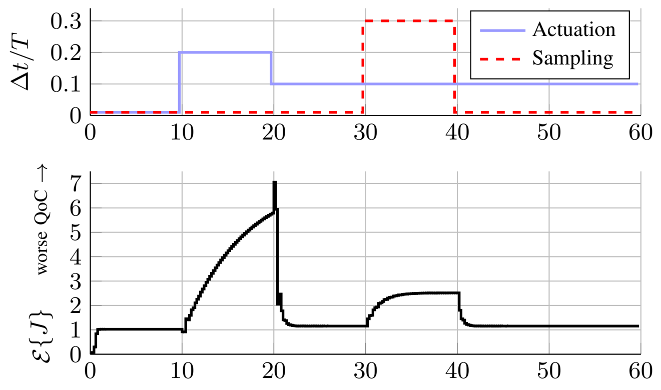


Figure 1. Quality-of-control evaluation for a controlled inverse pendulum for time-varying actuation and sampling delays [11]. The top figure shows the delay normalized to the control period, where 0 is the perfect timing for which the controller was designed. The bottom figure shows the cost over time, where 1 is the performance for perfect timing and larger values correspond to worse Quality of Control, i.e., larger amplitude of error and control signal.

dynamic reconfigurability and mixed-criticality scheduling to boost average performance. Our goal is to achieve the benefits of such approaches without loosing the indispensable feature of traditional static scheduling: guaranteed QoC.

We, therefore, present our ongoing work on Quality-Aware Responsive Real-Time Control Systems (QRONOS), an approach to (a) model and quantify average-case QoC in a time-dependent manner, (b) incorporate non-deterministic input and output timing in the design of controllers and ease verification of the resulting worst-case QoC, and (c) leverage that knowledge by a quality-aware design of the real-time operating system executing the controllers.

In the following sections, we go through these aspects and detail our previous and ongoing work as well as provide an outlook on our future challenges and steps.

A. Average-Case Analysis of Quality of Control

As a first step, we focused on average-case QoC evaluation, i.e., how well the system performs typically. Besides QoS in the form of non-perfect input/output timing, we consider the influence of stochastic physical disturbance (e.g., side wind), measurement noise and control situation (e.g., fast curve vs. straight road). For this complex system model, we developed a QoC evaluation scheme in [11] that can quantify the combined negative impacts of said effects. To gain insight into the dynamic behavior at changing timing, e.g., due to a changing criticality level in mixed-criticality-scheduling, we introduce a noise-averaged, but time-dependent QoC, roughly equivalent to the performance over time for typical disturbance.

Formally, this is modeled as the time-dependent expectation value $\mathcal{E}_{\mathcal{N}}\{J(t)\}$ about the noise $\mathcal{N}$, where $J(t)$ is the cost function from Section I, which weights physical state and control signal amplitude. To compute this averaged QoC without averaging over a multitude of simulations with different random sequences for disturbance and measurement noise, a scheme to directly evaluate this expectation was developed. It is based on first reformulating the problem as a linear impulsive system, which combines continuous and

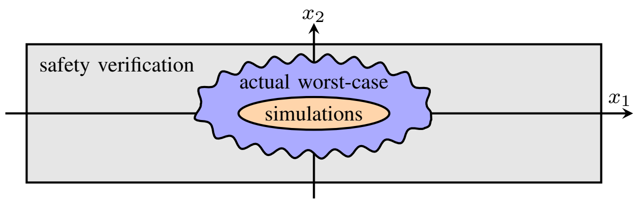


Figure 2. The bounds for the physical state x obtained by a finite number of simulations are too optimistic, whereas safety verification is too pessimistic.

discrete dynamics to model the plant as well as sampling and actuation of the discrete-time controller respectively. As a second step, a stochastic discretization is applied, from which the QoC is computed. The algorithms are currently available for deterministic or discrete stochastic timing [11].

Figure 1 gives the results of our QoC model for the inverted pendulum example discussed in Section I. Here, a deterministic (non-random) timing sequence was used for simplicity. This example is particularly suited to our approach as it demonstrates the possible efficiency gains: The model returns the exact result in a fraction of a second, whereas averaging over a multitude of simulations requires over five hours for an approximation with about 3 percent error [11].

With this, we offer a systematic approach for evaluating the temporal development of the QoC. We consider this a vital step towards an accurate usage of QoC as an evaluation metric in dynamic and adaptive real-time settings, such as mixed-criticality scheduling, and as a basis for further research on co-design of real-time control systems. Since our approach takes traditionally-designed control systems as input, it can be applied to evaluate the impact of timing on existing systems.

Outlook: We are currently working on further reduction of the computational effort and extending the efficiency gains to a wider problem class. The aim is to use the model (a) also for complex control systems with multiple inputs and outputs and (b) at runtime for QoC-aware timing adaptation.

B. Worst-Case Analysis of Quality of Control

The average-case QoC discussed in the previous section is important to quantify how the system will behave typically. On the other hand, it is equally important to show that the physical system always stays within safety bounds, even in the rare but possible worst case.

While randomized simulation is a pragmatic approach to assess the average performance, it is generally incapable of proving worst-case properties, due to the infeasible number of possible execution flows and timings. As visualized in Fig. 2, simulations can only determine an optimistic lower bound of the worst case. Instead, we opt for a sound overapproximation of worst-case behavior by verification methods.

Currently, we are working on worst-case verification of real-time control systems with uncertain input and output timing by modeling them as hybrid automata [16]. As with the linear impulsive systems used in Section III-A, hybrid automata allow combining the discrete-time and continuous-time aspects of a real-time control system. Unlike linear impulsive systems, which typically require additional informal explanation of the timing model, hybrid automata are a machine-readable precise formal description directly suitable for automatic verification.

As with any form of static analysis, the fundamental challenges are soundness, feasibility, and tight bounds: For the example of a car, we strive to prove that the worst-case track deviation is less than a few centimeters, not meters. Figure 2 illustrates that the bounds shown by verification can significantly exceed the actual worst case, requiring unnecessary safety margins in the design. Our preliminary experiments with existing tools indicate that verification is feasible with useful bounds in some cases yet challenging in general [16].

Therefore, we are currently pursuing an alternative approach to worst-case verification: instead of proving stability in the presence of jitter, we eliminate the jitter for input and output operations altogether. This obligation requires the real-time system to increase its QoS to the maximum. For its implementation, a well-established method is to resort to a completely static schedule and sound WCET analysis of all control activities. However, sensors and actuators must admit deterministic response times, which typically excludes smart sensors with internal signal processing. In turn, we can resort to traditional stability verification of feedback control loops. We show how to nonetheless benefit from adaptive real-time system techniques and our QoC model in the next section.

Outlook: While numerous techniques for the efficient verification of discrete-time controllers exist, to the best of our knowledge, none of them supports timing uncertainties as presented in our timing model [16], which addresses real-time systems with multiple sensors and actuators by introducing periodic timing windows. Therefore, future work will entail an extension of existing techniques such as [17], [18].

C. Quality-Aware Real-Time Executive

To tackle the challenge of saving resources without violating the application-specific QoC requirements, we propose a quality-aware real-time executive. That is, operating system support and scheduling instrumentation to make the QoC a first-class citizen equal to QoS, i.e., temporal parameters.

Therefore, we are working on an additional scheduler module that applies to jobs with control activities. Based on a simplified variant of the QoC-model from Section III-A, the module adapts release times and deadlines such that it leverages the situation-dependent reserves (i.e., margin between current and specified QoC) to boost average performance and overall runtime flexibility. At the same time, it ensures that adverse effects of varying timing (cf. Section I) are considered and do not jeopardize stability.

As mentioned earlier, worst-case stability analysis of feedback control under the assumption of non-deterministic timing is still subject to research. We find that even with progress in this direction it will be infeasible to verify dynamically scheduled real-time systems with QoC-dependent adaptation of QoS. Therefore, we propose a hybrid execution model where the system switches to a pre-computed, time-triggered schedule whenever a pessimistic QoC-model anticipates a potential violation of the minimal QoC in the next control step. To yield worst-case guarantees, this model based on Section III-B assumes worst QoS and disturbance.

We call this switching to a static schedule (i.e., deterministic input and output timing) and a verified controller setting (i.e., stability and WCET) our safety net. While in this mode, the QoC will recover verifiably.

This combination of an optimistic QoC-aware and a deterministic safety mode is an ideal supplement to established approaches, such as mixed-criticality scheduling. In contrast to traditional scheduling approaches, it is the potential violation of the QoC that indicates a change in criticality whereas control activities are otherwise categorized as low-criticality jobs. Additionally, control-theoretical approaches exist to implement controllers with graded assurance levels, for example, the Simplex architecture as used in [19]: regularly, a controller that performs well in the average case (here: optimistic mode) but does not offer worst-case guarantees is used. If an unsafe situation is imminent, a safety mechanism switches to a safety controller (here: safety mode) that offers strict worst-case guarantees but performs worse in the average case.

Outlook: We are currently working on an efficient implementation of the QoC-model and the safety net based on LitmusRT [20]. Here, we investigate the potential for an offline analysis of all possible QoC conditions and to thereof derive a lookup table to eliminate the computational overhead. A further promising candidate that we currently investigate for runtime QoC evaluation are machine-learning approaches. For the safety mechanism, we are working on the control theoretical question of a verifiable design such that it does not activate too often, but still provides provable safety guarantees.

Solving worst-case QoC verification for uncertain timing will permit a relaxation of the deterministic safety mode. Then, only timing bounds instead of timing instants will have to be guaranteed, which greatly simplifies the implementation.

IV. Summary

Real-time control systems face a fundamental design conflict between real-time system and controller design: to improve flexibility and efficient resource usage, design goals are shifting from deterministic execution towards good enough QoS properties with weaker guarantees. However, degraded temporal properties, in particular, any variation in sensor or actuator timing, can jeopardize quality-of-control guarantees.

To solve this conflict between efficiency and QoC guarantees, we propose a holistic approach with two modes: an optimistic mode uses dynamic scheduling and adapts the QoS of the control application to the lowest value permitted by current and future QoC. Verification of this mode is generally infeasible. Thus, we provide worst-case guarantees instead by switching to a safety mode if the minimum permissible QoC is about to be violated. This safety mode uses time-triggered, deterministic scheduling to facilitate QoC verification.

We consider our approach a vital step towards the use of runtime dynamics and adaptivity in safety-critical control systems. Its key features are models to capture the non-trivial relation of QoC and QoS for both average-case and worst-case.

Future work will be directed towards the realization of the proposed approach, especially theory and implementation of the safety mechanism, QoC prediction and worst-case analysis.

References

[1] G. Buttazzo and A. Cervin, “Comparative assessment and evaluation of jitter control methods,” in *Proc. of the 15th Intl. Conf. on Real-Time and Network Systems (RTNS ’07)*, 2007, pp. 163–172.

[2] B. Wittenmark, J. Nilsson, and M. Törngren, “Timing problems in real-time control systems,” in *Proc. of the American Control Conf.*, New York, NY, USA, 1995, pp. 2000–2004.

[3] A. Ray, “Output feedback control under randomly varying distributed delays,” *Guidance, Control, and Dynamics*, vol. 17, no. 4, pp. 701–711, 1994.

[4] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 1st ed. Kluwer Academic Publishers, 1997.

[5] S. Vestal, “MetaH support for real-time multi-processor avionics,” in *Proc. of 5th Intl. Work. on Parallel and Distributed Real-Time Systems and 3rd Work. on Object-Oriented Real-Time Systems*. IEEE, Apr 1997, pp. 11–21.

[6] A. Burns and R. Davis, “Mixed criticality systems – a review,” Department of Computer Science, University of York, Tech. Rep. 9th ed., 2016.

[7] D. Simon, A. Seuret, and O. Sename, “On real-time feedback control systems: Requirements, achievements and perspectives,” in *Systems and Computer Science (ICSCS), 2012 1st Intl. Conf. on*, Aug. 2012.

[8] A. Cervin and J. Eker, “Feedback scheduling of control tasks,” in *Proc. of the 39th IEEE Conf. on Decision and Control (CDC ’00)*, vol. 5. New York, NY, USA: IEEE Press, 2000, pp. 4871–4876.

[9] G. Bernat, A. Burns, and A. Liamosi, “Weakly hard real-time systems,” *IEEE Trans. on Computers*, vol. 50, no. 4, pp. 308–321, Apr. 2001.

[10] T. Klaus, F. Franzmann, M. Gaukler, A. Michalka, and P. Ulbrich, “Poster Abstract: Closing the Loop: Towards Control-aware Design of Adaptive Real-Time Systems,” in *Proc. of the 37th Real-Time Systems Symp. (RTSS ’16)*. IEEE, 2016, pp. 363–363.

[11] M. Gaukler, A. Michalka, P. Ulbrich, and T. Klaus, “A new perspective on quality evaluation for control systems with stochastic timing,” in *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control - HSCC ’18*. ACM Press, 2018.

[12] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén, “Feedback-feedforward scheduling of control tasks,” *Real-Time Systems*, vol. 23, no. 1-2, pp. 25–53, 2002.

[13] G. Buttazzo, M. Velasco, and P. Marti, “Quality-of-control management in overloaded real-time systems,” *IEEE Trans. on Computers*, vol. 56, no. 2, pp. 253–266, 2007.

[14] F. Flavia, J. Ning, F. Simonot-Lion, and S. YeQiong, “Optimal on-line (m,k)-firm constraint assignment for real-time control tasks based on plant state information,” in *IEEE Intl. Conf. on Emerging Technologies and Factory Automation (ETFA ’08)*. IEEE, 2008, pp. 908–915.

[15] B. Lincoln and A. Cervin, “JITTERBUG: a tool for analysis of real-time control performance,” in *Proc. of the 41st IEEE Conf. on Decision and Control (CDC ’02)*. IEEE, 2002, pp. 1319–1324.

[16] M. Gaukler and P. Ulbrich, “Worst-case analysis of digital control loops with uncertain input/output timing,” in *ARCH19. International Workshop on Applied veRification for Continuous and Hybrid Systems*, 2019, accepted for publication.

[17] S. Bak and T. T. Johnson, “Periodically-scheduled controller analysis using hybrid systems reachability and continuization,” in *2015 IEEE Real-Time Systems Symposium*. IEEE, dec 2015.

[18] L. Hetel, C. Fiter, H. Omran, A. Seuret, E. Fridman, J.-P. Richard, and S. I. Niculescu, “Recent developments on the stability of systems with aperiodic sampling: An overview,” *Automatica*, vol. 76, pp. 309–335, Feb. 2017.

[19] D. Seto, B. H. Krogh, L. Sha, and A. Chutinan, “Dynamic control system upgrade using the simplex architecture,” *IEEE Control Systems*, vol. 18, no. 4, pp. 72–80, Aug. 1998.

[20] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, “LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers,” in *Proceedings of the 27th Real-Time Systems Symposium (RTSS ’06)*, 2006, pp. 111–126.

AUTOSAR Runnable Scheduling for Automobile Control Application's Optimal Performance

Daeho Choi
Grad School of Automotive Engineering
Kookmin University
Seoul, Korea
chleogh0531@kookmin.ac.kr

Wootae Jeon
Grad School of Automotive Engineering
Kookmin University
Seoul, Korea
wjsdnxo@kookmin.ac.kr

Jong-Chan Kim*
Dept. of Automobile and IT Convergence
Kookmin University
Seoul, Korea
jongchank@kookmin.ac.kr
*Corresponding Author

***Abstract*—Automobile control applications should extract the best possible control performance out of its underlying hardware resources like CPU. Since the control application's timing behavior heavily affects the control performance, this paper focuses on optimizing the control performance by finding the proper scheduling parameters for a given control application. We specifically target the industry standard AUTOSAR software architecture where the application's timing behavior is collectively determined by runnable periods. Runnable is a small function with its trigger condition such as execution period. With this motivation, this paper proposes a novel method that systematically finds the optimal runnable periods that maximize the control performance. Instead of using time-consuming combinatorial searches, our method analytically finds the near-optimal runnable periods, which, as our evaluation result shows, has only about 3% of performance loss.**

***Index Terms*—AUTOSAR, Runnable Scheduling, Runnable Periods, Control-Scheduling Co-Design**

I. Introduction

When developing automobile control applications, AUTOSAR (AUTomotive Open System ARchitecture) has been widely accepted as the standard software architecture. In AUTOSAR, a control application is developed as a set of software components where each software component is also composed of a number of runnables, which is a small function with a trigger condition such as its execution period. Since AUTOSAR uses real-time operating systems as its underlying execution environment, runnables with identical periods are grouped together forming a number of real-time periodic tasks. Thus for the timing perspective, determining each runnable's period plays a critical role when integrating the system.

For this critical runnable periods decision, the current approach is usually ad-hoc where each runnable's function developer simply decides his or her runnable's period based on their previous experiences regarding how it affects the overall control performance. Note that a control application's timing behavior heavily affects the system's resulting control performance [1]. Thus the industry needs a new systematic approach for the runnable periods decision for the purpose of both enhanced control performance and reduced manufacturing cost.

With this motivation, this paper presents an AUTOSAR runnable scheduling method, which finds the optimal runnable periods which maximize the control performance. This problem was first formulated by our previous work [2], which proposed a combinatorial search method for simple control applications with only sequential data dependencies between runnables. This paper further develops the original method for more complex applications. For that, a control application is modeled as a directed acyclic graph (DAG) of runnables with data dependencies represented as directed edges between runnables. Also the control performance of the application is modeled as a linear cost function of control period and end-to-end delay. Combining the DAG model and the control cost model, we formulate a new optimization problem which minimizes the control cost by controlling runnable periods.

As an initial effort towards a general solution for arbitrary DAG models, we first present a preliminary solution for a limited set of DAG models, more specifically, which has no *crossing node* in the middle of the graph. With crossing nodes, we mean a node with multiple input edges or multiple output edges. When solving the problem, we try to find an analytical method which does not require time-consuming search process. For that, we transform our optimization problem with n runnable periods to an alternative optimization problem of only 3 free variables. Although this simplification gives up the optimality, we can find an analytical way that uses the Lagrange multiplier method that finds the near-optimal solution. Our evaluation result shows that the performance loss compared to the optimal solution is only under or about 3% for our test set.

The rest of our paper is organized as follows: Next section gives a brief background and describes our problem. In Section III, our AUTOSAR runnable scheduling method is formally presented. Section IV gives the evaluation results. Finally, Section V concludes this paper with an emphasis on our remaining future works.

II. Background and Problem Description

A. System Model

This paper considers an AUTOSAR-based automobile control application, which is designed as a set of N software components $\{C_1, C_2, \cdots, C_N\}$. Each software component C_i is also composed of a number of runnables $\{r_{i1}, r_{i2}, \cdots, r_{i|C_i|}\}$

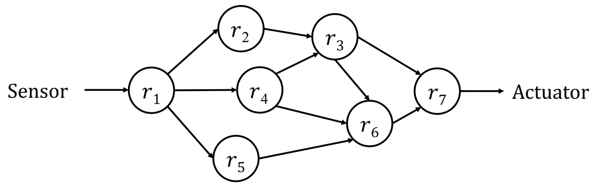


Fig. 1. Example DAG (Directed Acyclic Graph) representing a AUTOSAR software component made of seven runnables with one input sensor and one output actuator

where $|C_i|$ denotes the number of runnables in C_i. We assume that each runnable r_{ij} is required to be periodically triggered.

As an initial effort, this paper assumes simple control applications with only a single software component. Thus, without loss of generalization, our system can be formally described as a set of n runnables denoted by

$$\{r_1, r_2, \cdots, r_n\}.$$

Each runnable r_i is also denoted by

$$r_i = (p_i, e_i)$$

where p_i is the runnable's period and e_i is the worst-case execution time. Note that e_i is a given value whereas p_i is a tunable parameter to the system designer's needs.

Between runnables, there are data dependencies where a sender writes its output to a shared buffer, and a receiver reads out the data. Note that this data communication is asynchronous since each runnable periodically executes with its own execution period. Figure 1 shows a simple DAG with seven runnables $\{r_1, r_2, \cdots, r_7\}$. Each node represents a runnable and arrows between runnables represent writer-reader data dependencies. We assume that the first runnable r_1 is responsible for accepting the sensor data and the last runnable r_n (r_7 in the figure) takes the role of controlling the actuator. Thus we simply call r_1 a sensor runnable and r_n an actuator runnable.

Our system in brief can be described as in the following. The sensor runnable periodically reads out the sensor data and processes it producing output data to other runnables. Then each runnable also periodically executes reading from the predecessors and writing to the successors. As a result, new sensor data gradually flows from the sensor runnable to the actuator runnable. After the new sensor data goes through all the runnables, the actuator runnable finally produces the proper actuator command based on the fresh environmental knowledge.

B. Control Performance Model

For the control performance model, we use the the linear control cost function originally defined by Bini and Cervin [3], which is a linear function of sampling period T and delay Δ denoted by

$$J(T, \Delta) = \alpha T + \beta \Delta \tag{1}$$

where α and β are application-specific constants. Note that both α and β should be positive since larger period and delay should increase the control cost.

C. Schedulability Constraint

When integrating the runnables on an execution platform, runnables are mapped to proper real-time periodic tasks. One simple approach is to make n respective periodic tasks for each runnable, which can produce too many tasks. More practical approach is to group runnables with the similar periods together to form a limited number of tasks. For any method, the real-time tasks should be schedulable meeting every task's deadline. In this paper, we pick the first approach. For the schedulability guarantee, we use the utilization bound method, which guarantees the system schedulability in case the total utilization $U = \sum_{i=1}^{n} \frac{e_i}{p_i}$ is under a certain threshold value U_B. U_B depends on the scheduling algorithm. For example, for the RM (Rate Monotonic) scheduling, U_B is roughly 69.3%, and for the EDF (Earliest Deadline First) scheduling algorithm, U_B is 100%. Assuming EDF, our schedulability constraint can be represented as

$$U(p_1, p_2, \cdots, p_n) = \sum_{i=1}^{n} \frac{e_i}{p_i} \leq 1. \tag{2}$$

D. Problem Description

Assuming the above system model, control performance model, and schedulability condition, our problem can be formally defined as follows:

Problem Description. With a given software component with a DAG-structured runnables set $\{r_1, r_2, \cdots, r_n\}$ and a control cost function $J(T, \Delta)$, find the optimal runnable periods $\{p_1, p_2, \cdots, p_n\}$ which minimize the control cost while meeting the system schedulability.

III. AUTOSAR Runnable Scheduling

A. Control Cost and Runnable Periods for DAG Model

To find the optimal runnable periods, our first step is to transform the control cost function $J(T, \Delta)$ into a function of our tunable parameter p_i.

From the perspective of the vehicle, only the actuator runnable r_n directly controls it. Thus, our control period T is dependent on only the actuator runnable's period p_n. To be more accurate, it is the longest time distance between two consecutive executions of r_n. This worst-case scenario occurs when two consecutive r_ns are scheduled at the beginning of a certain period and at the end of the next period, respectively, making the longest time distance. Thus T can be defined as

$$T = 2p_n. \tag{3}$$

For the end-to-end delay Δ, it is defined as the worst-case time delay from the sensor to the actuator. Note that there can be multiple different paths from the sensor runnable to the actuator runnable. For such m paths, we define $\mathbb{P}_k$ as

$$\mathbb{P}_k = \{i \in \mathbb{N} | r_i \in k\text{-th path}, 1 < i < n\} \text{ for } 1 \leq k \leq m, \tag{4}$$

which is a set of runnable indexes in the k-th path with $|\mathbb{P}_k|$ denoting the number of elements in it. We do not include r_1

and r_n in paths since they do not belong to a specific path. Then the end-to-end delay Δ can be defined as

$$\Delta = 2p_1 + \max_{k\in[1,m]} \left(\sum_{i\in\mathbb{P}_k} 2p_i \right) + 2p_n. \tag{5}$$

Note that while going through a runnable r_i, in the worst case, it takes $2p_i$. It happens when a certain r_i instance, scheduled at the beginning of its period, slightly misses the new input data and the next instance is scheduled producing its output at the end of the period.

To make a simpler form of Δ that can be used in the optimization, we only consider a simplified DAG form as in Figure 2. In the simplified form, there is no crossing node and hence there are m independent paths from r_1 to r_n. From now on, DAG means this simplified DAG form, if not otherwise specified. Also, for the m paths, we assume that the sum of participating runnable's periods within every path is the same[1], which is formally expressed using a new notation p_* by

$$p_* = \sum_{i\in\mathbb{P}_1} p_i = \sum_{i\in\mathbb{P}_2} p_i = \cdots = \sum_{i\in\mathbb{P}_m} p_i.$$

Then we can simply define Δ as

$$\Delta = 2(p_1 + p_* + p_n). \tag{6}$$

Then, using (3) and (6), we can finally transform $J(T,\Delta)$ into a function of p_1, p_*, and p_n as in

$$J(p_1, p_*, p_n) = \alpha \times 2p_n + \beta \times 2(p_1 + p_* + p_n). \tag{7}$$

B. Our Optimization Method

Within each k-th path, the sum of all the participating runnable's period is define as p_*. Then we distribute p_* to participating runnables proportional to each e_i within the path. The above policy can be expressed as

$$p_i = \frac{e_i}{\sum_{j\in\mathbb{P}_k} e_j} p_* \quad (i \in \mathbb{P}_k). \tag{8}$$

Then, using (3), (6) and (8), the utilization function $U(p_1, p_2, \cdots, p_n)$ can be also transformed into a function of p_1, p_*, and p_n as in

$$\begin{aligned} U(p_1,p_*,p_n) &= \frac{e_1}{p_1} + \frac{\sum_{k=1}^{m}\sum_{i\in\mathbb{P}_k}\sum_{j\in\mathbb{P}_k} e_j}{p_*} + \frac{e_n}{p_n} \\ &= \frac{e_1}{p_1} + \frac{\sum_{k=1}^{m}\sum_{i\in\mathbb{P}_k} |\mathbb{P}_k| e_i}{p_*} + \frac{e_n}{p_n}. \end{aligned}$$

Now, our optimization problem is defined as

$$\begin{aligned} \underset{p_1,p_*,p_n}{\text{minimize}} \quad & J(p_1,p_*,p_n) \\ \text{subject to} \quad & U(p_1,p_*,p_n) \le 1, \end{aligned}$$

which has only three free variables p_1, p_*, and p_n.

[1]We argue that this is the necessary condition for the optimal periods at least for the simplified DAG form. Due to the page limit, this will be further explained in our future work.

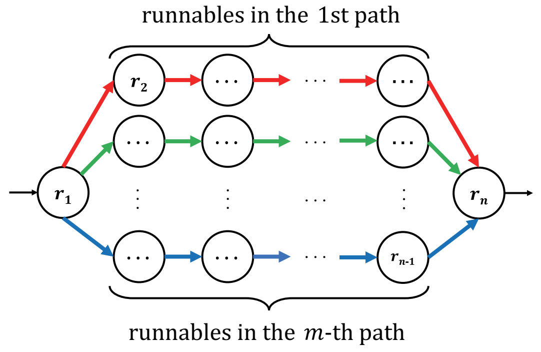


Fig. 2. Simplified DAG form with no crossing node

For the optimization, we decide to use the Lagrange multiplier method and define the Lagrange's equation as

$$\begin{aligned} \mathcal{L} &= J(p_1,p_*,p_n) - \lambda(U(p_1,p_*,p_n) - 1) \\ &= 2\beta(p_1 + p_*) + 2(\alpha+\beta)p_n \\ &\quad - \lambda\left(\frac{e_1}{p_1} + \frac{\sum_{k=1}^{m}\sum_{i\in\mathbb{P}_k}|\mathbb{P}_k|e_i}{p_*} + \frac{e_n}{p_n} - 1\right). \end{aligned} \tag{9}$$

Then, we take the partial derivatives of p_1, p_*, p_n, and λ of (9) and solve the equations of

$$\nabla\mathcal{L} = \left(\frac{\partial\mathcal{L}}{\partial p_1}, \frac{\partial\mathcal{L}}{\partial p_*}, \frac{\partial\mathcal{L}}{\partial p_n}, \frac{\partial\mathcal{L}}{\partial\lambda}\right) = 0,$$

which yields the following optimal p_1, p_*, and p_n:

$$\begin{aligned} p_1 &= e_1 + \sqrt{e_1 \sum_{k=1}^{m}\sum_{i\in\mathbb{P}_k}|\mathbb{P}_k|e_i} + \sqrt{\frac{(\alpha+\beta)e_1 e_n}{\beta}} \\ p_* &= p_1\sqrt{\frac{\sum_{k=1}^{m}\sum_{i\in\mathbb{P}_k}|\mathbb{P}_k|e_i}{e_1}} \\ p_n &= p_1\sqrt{\frac{\beta e_n}{(\alpha+\beta)e_1}}. \end{aligned} \tag{10}$$

Now, applying (8) to (10), our optimal periods $\{p_1, p_2, \cdots, p_n\}$ can be simply calculated.

IV. Experiment

In this section, we evaluate our runnable scheduling method in terms of both its resulting control performance and the required optimization time. We compare our method denoted by *Ours* with *Optimal*, which is the optimal runnable periods that minimize the control cost. To find the optimal periods, we conduct exhaustive searches for the entire integer problem space bounded by $1 \le p_i \le 1000$.

We use the four DAGs in Figure 3 for the evaluation. In the figure, each DAG is denoted by (nR, mP), where n is the number of runnables and m is the number of paths in each DAG. Below each runnable r_i, its assumed execution time e_i is depicted. For the control cost function which is the optimization objective function, we use

$$J(T,\Delta) = \frac{1}{1000}T + \frac{1}{1000}\Delta.$$

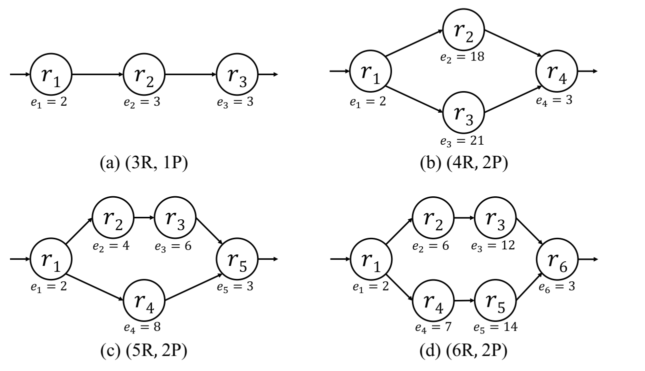


Fig. 3. Example DAGs with varying number of runnables (denoted by nR) and number of paths (denoted by mP) with execution times shown below each runnable

TABLE I
RUNNABLE PERIODS FOUND BY OUR METHOD COMPARED TO THE OPTIMAL PERIODS EACH FOUR EXAMPLE DAGS

DAG	Periods by Ours	Optimal Periods
(3R,1P)	{8, 10, 7}	{10, 10, 6}
(4R,2P)	{15, 64, 64, 13}	{16, 63, 63, 12}
(5R,2P)	{13, 20, 29, 49, 12}	{15, 22, 25, 47, 11}
(6R,2P)	{18, 38, 75, 38, 75, 16}	{18, 46, 64, 46, 64, 15}

Table I compares the runnable periods found by our method with the optimal runnable periods. Note that the sets in the table represent the runnable periods as is $\{p_1, p_2, \cdots, p_n\}$. Although our method initially finds real-valued periods, they are rounded to the nearest integers for the ease of comparison with the integer-valued optimal periods. The resulting solutions of the two methods show little euclidean distances. Based on this finding, it is also possible to combine our analytical method and the search method by using our runnable periods as the starting position of the further searches.

Figure 4 compares the normalized control cost of the two methods. In the figure, we can find that although our solution is not the optimal one, the performance loss is negligible, which is under about 3%. However, on the contrary, Figure 5 shows very different result regarding the time required to find the solution. In the figure, as the application's complexity grows from (3R, 1P) to (6R, 2P), *Ours* method shows little difference for the optimization time whereas *Optimal* method eventually requires too much time even for applications with modest complexity. We can presumably argue that *Optimal* method cannot be applied to industry-sized applications in practice.

V. Conclusion

This paper presents an AUTOSAR runnable scheduling method for the optimal performance of control applications. Target control application is modeled as a DAG of runnables and the control performance model is formulated as a linear cost function of the system's control period and end-to-end

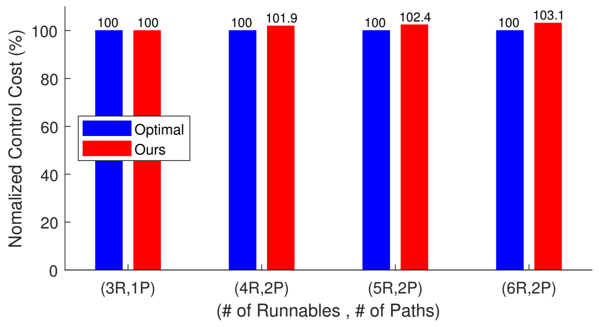


Fig. 4. Comparison of normalized control cost for example DAGs

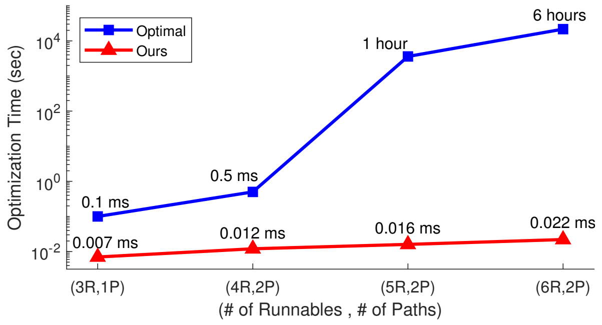


Fig. 5. Comparison of optimization time for example DAGs

delay. Then we define the control period and end-to-end delay as functions of runnable periods, respectively, which formulate an optimization problem to minimize the control cost using runnable periods as tunable parameters. As our initial efforts, we present an analytical solution for simplified DAG forms with no crossing node. Our evaluation result shows that our optimization method can quickly find near-optimal runnable periods with a negligible performance penalty.

In our future work, we plan to generalize our method for the general DAG model. Also, we plan to optimize the runnable to task mapping process considering the intra-task data dependencies and runnables execution sequence.

Acknowledgment

This work was supported partly by the National Research Foundation (NRF) grant funded by the Korea government (MSIT; Ministry of Science and ICT) (No. A2019-0098) and partly by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (2014-0-00065, Resilient Cyber-Physical Systems Research).

References

[1] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Arzen, "How does control timing affect performance? analysis and simulation of timing using jitterbug and truetime," *IEEE Control Systems*, vol. 23, no. 3, pp. 16–30, 2003.
[2] T.-W. Kim, G.-M. Lee, and J.-C. Kim, "Autosar runnable scheduling for optimal tradeoff between control performance and cpu utilization," in *Proceedings of the 18th IEEE International Conference on Control, Automation, and Systems (ICCAS)*, pp. 602–605, 2018.
[3] E. Bini and A. Cervin, "Delay-aware period assignment in control systems," in *Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS)*, pp. 291–300, 2008.

Demo Abstract: Testbed for Practical Considerations In Mixed-Criticality System Design

Vijaya Kumar Sundar, Arvind Easwaran
Nanyang Technological University, Singapore
Email: vijayaku003@e.ntu.edu.sg, arvinde@ntu.edu.sg

I. Introduction

Increase in number of autonomous functionalities has led to design approaches where software elements requiring varied levels of execution reliability are made to share a common platform (hardware and software) and such systems are termed as Mixed-Criticality Systems (MCS). An important challenge in designing MCS is to handle uncertainties in the execution behaviour such as a budget overrun of a software element (or task) in a safe and effective manner. For this purpose, functional safety standards such as ISO 26262 [1] specify a set of rules and rigorous process which necessitate the system design to incorporate sufficient mechanisms to prevent the propagation of faults from tasks certified to lower criticality level to tasks certified to higher criticality level.

A majority of the existing MCS task models (reviewed in [2]) consider criticality to denote the relative importance of a task and use it to achieve graceful degradation of the system either by suspending or degrading tasks with relatively lower criticality level than the overloading task. As pointed out in [3], notion of criticality as the relative importance of a task may be appropriate only for dual-criticality MCS models (where a task can be either high critical or low critical). For instance, it has been observed that in a system with 4 criticality levels, to handle budget overrun of any task at criticality level 4 (highest), degradation of tasks with criticality level 3 may not be acceptable. As stated in [4], the notion of criticality, relative importance and degradation of a functionality in an MCS should be loosely coupled entities. To achieve this, we consider a context-aware degradation strategy with the following three properties:

1) Upon the system overload, there is a flexibility to choose the tasks that has to be degraded depending on the overrun task and independent of their criticality level. This allows the designer to consider degradation of even higher criticality tasks while keeping relatively lower criticality tasks unaffected.
2) Second, there is a support for multiple ways of degrading a task's budget. This allows the designer to choose a specific way of degrading a task's budget depending on the tasks that have overrun their budget.
3) Third, instead of using criticality to decide tasks to be degraded, criticality is used to choose the degraded budget (among multiple degraded budgets) of the task when multiple tasks overrun their budgets.

The above properties are motivated from an automotive case study discussed in [5] where the possibility of multiple ways of degrading a higher critical LIDAR processing task is considered. A paper describing our context-aware degradation model can be found at [6]. In order to showcase the benefit of context-aware degradation, an automotive testbed is built to observe the effects of task degradation schemes enforced by different MCS task models as well as the context-aware degradation on the safety and performance aspect of the vehicle which are generally not possible with synthetic task set experiments.

II. Automotive Testbed

A. Testbed Architecture

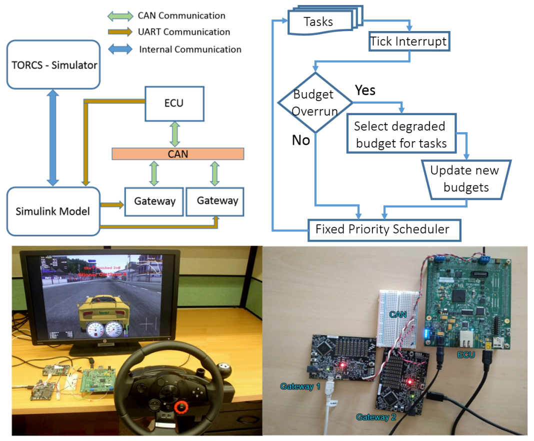


Figure 1. Testbed Architecture

The testbed is a Hardware-In-The-Loop simulation platform consisting of The Open-source Race Car Simulator (TORCS) [7] - vehicle simulator, simulink model, gateway, Electronic Control Unit (ECU) and Controller Area Network (CAN). The testbed and its architecture is shown in the Figure 1. The simulink model acts as an interface between TORCS and the gateway. It is used to collect sensor and actuator values for real-time monitoring and post-processing. The gateway consists of embedded controllers (FreeScale Kinetis Board [8]) networked over an industry standard CAN bus which converts the simulated sensor values from TORCS to CAN messages. Texas Instruments Hercules development board [9] acts as an ECU. The ECU runs FreeRTOS, a pre-emptive fixed priority based real-time scheduler. The control algorithms of applications are implemented as real-time tasks on FreeRTOS hosted on the ECU. The gateway receives the sensor data from TORCS through serial communication and dispatches them as CAN messages to the ECU executing the control algorithm tasks to generate actuator commands like throttle, brake and steer. These commands are sent back to TORCS through the Universal Asynchronous Receiver/Transmitter (UART). The testbed simulates two vehicles, a lead vehicle controlled by the simulink model and a follower vehicle controlled by the ECU. Values such as position, velocity, acceleration, brake, road curvature, distance travelled from starting point, etc. of the both lead and follower vehicles are transmitted to the ECU as CAN messages. Under the absence of budget overrun, all tasks are operated with a time period of 20 ms since TORCS by default provides sensor and vehicle data for every 20 ms. The safety functionalities implemented are considered to be certified to one of four Automotive Safety Integrity Levels (ASIL) as per ISO 26262.

B. Automotive Applications Implemented

1) Lead vehicle detection: The lead vehicle detection functionality is implemented using two tasks, namely pseudo-radar and vehicle to vehicle (V2V) communication. They are considered to be certified to ASIL C and ASIL B respectively. In our testbed, the euclidean distance calculated by the pseudo-radar task is used by other applications only if lead vehicle is within 120-degree field-of-view and within a range of 30 meters. This simulates the actual radar since 120 degree field-of-view may not be sufficient to detect the lead vehicle in curves. V2V task calculates the euclidean distance between vehicles in all conditions (including curves).

2) Longitudinal and Lateral Vehicular Control: The cruise control (CC) task certified to ASIL C, computes the necessary acceleration and brake commands based on the distance between vehicles calculated by the pseudo-radar and V2V tasks. ACC can operate in two modes viz 'constant time headway' or 'constant velocity'. In constant time headway mode, CC implements an algorithm to maintain distance between the vehicles such that under heavy braking of the lead vehicle, there is sufficient time for the follower vehicle to apply brakes to maintain safe distance. To achieve this, CC can use Proportional Integral Derivative (PID) control or ON-OFF control mechanism. CC task also implements Dynamic Speed Adaptation (DSA) functionality to detect and indicate the presence of curves and to bring down the vehicles' velocity to a predetermined value for a safe vehicle manoeuvre through the curves. Computation of required brake command of the host vehicle is decided by CC considering the presence of both the curve and the lead vehicle ahead. We consider two ways of degrading the CC task. With type 1 degradation, CC operates with ON-OFF control instead of PID leading to reduced execution time. With type 2 degradation, CC ignores the DSA functionality also leading to reduced execution time. For PID control mechanism, an algorithm presented as simulink model in [10] was implemented. The implementation discussed in [11] is used for DSA. The safety functionality of Steering Control (SC) application is to maintain the vehicles' position at the centre of the lane while the Collision Avoidance (CA) applies emergency braking if imminent collision is detected. Both SC and CA are implemented based on [10] and are certified to ASIL D (highest).

III. Demonstration

After the testbed is up and running, we will demonstrate its following key functionalities: 1) artificially trigger budget overrun of tasks using push buttons to simulate different task overrun scenarios where each scenario differs by tasks that have overrun their budgets. 2) a real-time plot of the live measurements of parameters such as the closest distance maintained between lead and follower vehicles denoted as D_{min}, values of acceleration and heading error of the follower vehicle. 3) Based on these measurements, for each overrun scenario, the effect of the context-aware degradation and two different degradation schemes considered in the existing MCS literature to handle budget overruns will be evaluated for the safety and the performance of the vehicle. Scheme 1 considers the suspension of V2V task - task with relatively lower criticality level. Scheme 2 considers only type 1 degradation of CC task - a single way of degrading a relatively lower criticality task's budget. With context-aware degradation, depending on the task overrun, either type 1 or type 2 degradation of CC task is chosen. The value of D_{min} is considered as the measure of safety whereas acceleration and heading error are considered as the measure of performance. 4) We will also describe the changes to the existing FreeRTOS API that has been done to incorporate context-aware degradation strategy.

IV. Conclusion and Future Work

In this demo paper, we present the design and implementation of an automotive testbed to evaluate the safety and performance of different degradation schemes along with context-aware degradation for mixed-criticality systems. As future work, we will integrate mode change protocols with the context-aware degradation.

References

[1] "ISO 26262 Road Vehicles – Functional Safety," International Organization for Standardization, Geneva, CH, Standard, Nov 2011.

[2] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 82:1–82:37, Nov. 2017.

[3] A. Esper, G. Nelissen, V. Nélis, and E. Tovar, "An industrial view on the common academic understanding of mixed-criticality systems," *Real-Time Systems*, vol. 54, no. 3, pp. 745–795, Jul 2018.

[4] K. Bletsas, M. A. Awan, P. Souto, B. Akesson, A. Burns, and E. Tovar, "Decoupling Criticality and Importance in Mixed-Criticality Scheduling," in *WMC 2018*.

[5] S. Holzknecht, E. Biebl, and H.-U. Michel, "Graceful degradation for driver assistance systems," *Advanced Microsystems for Automotive Applications 2009, Springer Berlin Heidelberg*, pp. 255–265.

[6] V. K. Sundar and A. Easwaran, "A practical degradation model for mixed-criticality systems," *IEEE ISORC*, 2019.

[7] B. Wymann, C. Dimitrakakisy, A. Sumnery, and C. Guionneaux, "Torcs: The open racing car simulator," 2015.

[8] "Frdm-kl25z: Freescale freedom development platform - kinetis mcus."

[9] "Hercules tms570ls31x/21x development kit." [Online]. Available: http://www.ti.com/tool/TMDS570LS31HDK

[10] https://github.com/VerifiableAutonomy/TORCSLink.

[11] G. S. C and R. Y, "Dynamic speed adaptation for path tracking based on curvature information and speed limits," *Sensors (Basel)*, vol. 17, no. 6, June 2017.

On Solving the IoT Development Silo Problem

Michael C. Brogioli, William Games, and Richard Moats
Network Native, Inc
Austin, Texas USA
{michael.brogioli, bill.games, richard.moats}@networknative.com

***Abstract*— Virtually all modern IoT systems are distributed and heterogeneous in architecture, comprising cloud computing, fog or edge computing, and embedded devices with as little as 8-bit bare metal CPU and limited resources. This paper presents a new approach to distributed heterogeneous development, facilitating connected embedded components to be developed and maintained as an extension of a cloud application. The demo will show all compute nodes of a modern IoT system being programmed, integrated and deployed using a unified development framework and language. The demo will show how real-world hardware, ranging from robust cloud servers to multi-core Linux gateways, down to bare-metal 8-bit MCUs can be incorporated, secured, and deployed in a real world system.**

***Keywords*— IoT, Heterogeneous Computing, Tools, DSL, Arch**

I. Introduction

Modern embedded and IoT (Internet of Things) solutions are distributed and heterogeneous, with hardware targets comprising low power 8-bit microcontrollers, lightweight but powerful network gateways, to the near limitless resources of internet cloud servers. Modern IoT solutions require expertise across disparate development platforms, or "silos." Throughout this paper, the term silos is used to represent the segmented development processes and tools required to bring a network spanning IoT solution to fruition. As development moves from cloud, to embedded components within a system, increasingly specialized and costly talent is required and locked within a given development silo. This is because developing efficient, secure, reliable embedded software still requires highly-specialized knowledge.

High-level abstractions that cloud and application developers take for granted have not found their way to embedded development. To wit, it is difficult if not impossible to stay agile while implementing features across multiple teams, tools, and targets—the logistical challenges slow the pace of product implementation and innovation. A real-world embodiment of the problem is a modern drone platform, comprising cloud servers for data collection, high power gateways, a robust application processor likely running Linux or other operating system, and bare metal 8-bit MCUs that may handle functions such as brushless motor control.

Many industry players have begun to recognize this rapidly growing problem, such as Intel Corporation, who states, "*One key difference between embedded and IoT is connectivity. We're transitioning away from isolated devices into a group of connected devices with awareness of their surroundings.*" Intel continues, "*If you think about all the accelerators to do analytics – CPU, graphics, video accelerator, deep learning engine, FPGA – you're talking about 4-5 different programming environment.*" "*It's not the same old tools environment. Tools must be done in a way that allows developers to move the workload and acceleration across all these accelerators in the cloud, gateway, and device as seamlessly as possible* [1]." Other industry juggernauts such as Amazon have taken note of their development silo problem within their AWS RoboMaker platform, aiming to tackle the problem of allowing developers to build and deploy applications via cloud services, while incorporating over the air updates and demands for a diverse set of hard to acquire skills [2]. Put succinctly, it is time to accelerate the development process!

II. The Solution

Network Native is building a new approach to distributed heterogeneous development, enabling connected embedded components to be developed and maintained as a natural extension of a modern cloud application. Using high-level abstractions modeled on real embedded product use cases, non-specialist developers will discover a new level of productivity and power [4]. The solution proposed is comprised, in part, of the following:

The Arch Language: A configurable and expandable DSL (Domain Specific Language) that enables an IoT application to be abstracted as a network-spanning, integrated whole that can be componentized for deployment across multiple heterogeneous computing targets connected by various communication channels. Arch abstracts away target specific considerations while offering high level abstractions that support physical computing applications including message passing, reactive programming, and hierarchical state machines. Arch facilitates rapid re-configuration of the division of labor between nodes or the insertion of a new hardware tier without modifying the application logic.

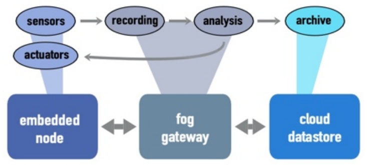


Fig. 1. Example of a Cloud, Fog, and Embedded application. May be re-deployed accordingly without any stage of compute shown.

Arch Compilation Infrastructure: The infrastructure makes it feasible to dynamically separate an application into components with secure network communications. The compilation infrastructure incorporates existing vendor tool chains for cloud, fog and embedded devices, thereby leveraging decades of hardened technology traditional developers have come to rely upon.

Arch Framework and Libraries: Frameworks offer efficient, maintained implementations for common software building blocks, such as: containerized data structures, math and data analysis, data storage and streaming, protocol stacks and secure communications channels, device feature drivers and external peripheral drivers.

Arch Application Deployment: The deployment system coordinates the automatic distribution of components to specified computing node targets. Each computing node has built-in intra-node communication capability with pre-fabricated transport security capability.

III. Tools Demonstration

The tools demonstration is comprised of the poster, tools demonstration, and online access to the open source GitHub and documentation repos of the Network Native code base. While beyond the scope of this abstract, preliminary videos of the early-stage Network Native proof-of-concept system from early 3Q17 can be seen via the content hosted at www.networknative.com. The current version of the 1Q19 technology will be presented at RTAS 2019 in a deep dive format.

The tools demonstration will be comprised of three parts: The system architecture overview shown in Fig 2, the application development, securitization, build, and iterative deployment system plus web-based IDE shown in Fig 3, and distributed heterogeneous application auto partitioning as shown in Fig 4.

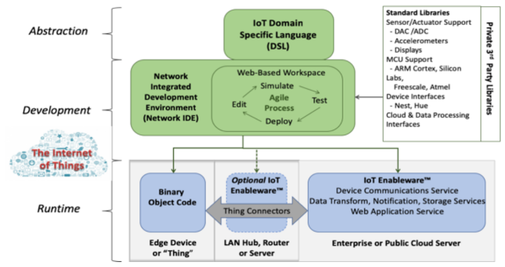


Fig. 2. The Network Native Eco-system, from application abstration to bare metal runtime environments.

The system architecture component of the demo will show how the Arch language is used to write distributed, heterogenous applications that automatically deploy across cloud, fog and embedded node targets. It also shows a web-based IDE demonstrating agile editing, simulation, test, and deployment.

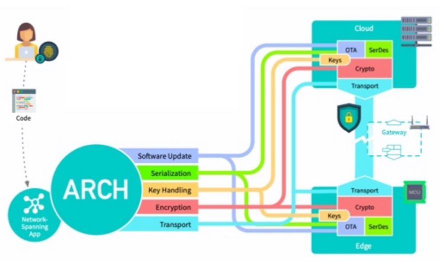


Fig. 3. Arch application, build and deployed with security and updates as well as transport layer integration.

The deployment module of the demonstration will show a user generated Arch application specification and configuration being built and deployed across the network of heterogeneous devices. This will include iterative software updates, serialization of communication channels, encryption and transport layer. The network is comprised of one or more cloud, fog and embedded nodes. The application itself will be modified, re-compiled and re-deployed in a matter of seconds to show how the system facilitates agile development work flows.

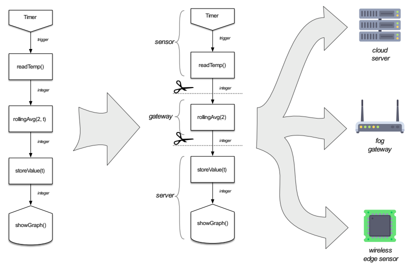


Fig. 4. An application being systematcially decomposed by the Arch system, and seriailized and deployed across the network.

The final component of the demonstration will be an illustration building upon the application examples, showing how applications are segmented and deployed across the network of cloud, fog and wireless embedded sensors. While not a compiler theory based session, practitioner focused tutorials will be provided focusing on application developers and providing better understanding of how the Network Native solution fits into their future IoT application development requirements.

IV. Deliverables

Network Native will also present information regarding their upcoming technical release that aims to incorporate support for new hardware targets, such as RISC-V, as well as various drone hardware platforms such as Qualcomm Flight.

[1] Schwaderer, Curt, Embedded Computing Design, *Driving IoT Innovation Along a Roadmap of Hardware*, Software and Tools, April 12, 2018.

[2] Amazon, Amazon Web Services Announces AWS RoboMaker, November 16, 2018

[3] Network Native, *The Next Wave of Connected Development,* February 2018.

[4] Kraeling, Mark B and Brogioli, Michael C, *Internet of Things – A Synopsis, its History, Application, Technology Architecture, and Challenges Moving Forward,* Embedded Systems - Expert Guide Series, Elsevier Publishing, Atlanta, GA, 2019.

Static Program Placement Platform for Embedded Multi-Core Multi-Cluster System

Seiya Maeda
Graduate School of Science and Engineering
Saitama University
Saitama, Japan
smaeda@mail.saitama-u.ac.jp

Yuya Maruyama
Graduate School of Engineering Science
Osaka University
Toyonaka, Japan

Takuya Azumi
Graduate School of Science and Engineering
Saitama University
Saitama, Japan

***Abstract*— We propose a static program placement platform for embedded multi-core multi-cluster systems. In recent years, the development of autonomous driving systems has advanced. Autonomous driving systems are developed using the Robot Operating System (ROS), which is suitable for the development of robotics and used for various systems of autonomous driving vehicles. Autonomous driving systems consist of multiple small programs. These small programs exchange messages called topics between programs. We have created a simple framework to realize these features for multi-core multi-cluster embedded system. It is lightweight and it is possible to statically determine the cluster number and the core number for executing a given program. This will improve real-time performance during program execution. This demonstration uses NVIDIA JETSON TX 2 as an embedded device. The autonomous driving system is composed of many programs. In this demonstration, the embedded device side executes self-driving modules such as path planning and path following.**

I. INTRODUCTION

One of the main functionalities of the robot operating system (ROS) [1] is a message exchange system called publish/subscribe. In addition, ROS is composed of small programs called nodes and these programs pass the control of messages called topics. We tried to operate these functions with lightweight and energy-saving embedded equipment. The assumed embedded board is composed of multiple clusters and has multicore processors on each cluster. We propose a mechanism to statically place ROS node programs on these clusters and cores with a specified number. The programs can run on each core and communicate with each other. In the autonomous driving system, many ROS programs already exist. We have made it possible to use these program source code without changing them. We propose a mechanism to easily realize the features of ROS on an embedded board. We call this development workflow ROS-lite, and details are explained in the next section.

II. ROS-LITE

ROS-lite is a lightweight implementation compared with ROS, and it can operate even with embedded systems with limited resources.

Using ROS-lite, we can run a ROS node application written in C++ on the target board with multiple clusters. We will explain the framework of Fig. 1. First, copy the source code and message file directly from the existing ROS package to the workspace. Next, proceed according to the ROS-lite workflow indicated by the red arrow. Finally, the ROS program is deployed and executed on the target embedded device. We explain this flow in the order of 1 through 4 shown in Fig. 1.

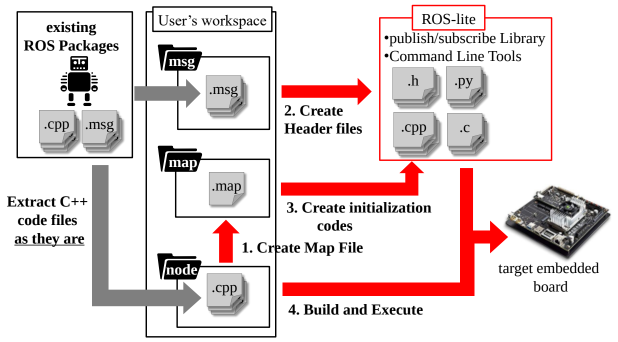
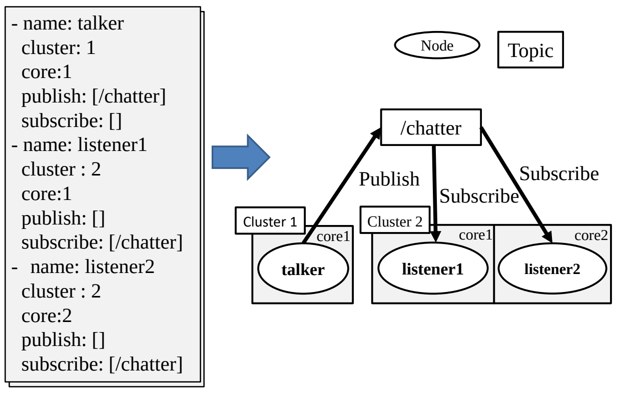


Fig. 1. Development flow of the ROS-lite framework.



```
- name: talker
  cluster: 1
  core:1
  publish: [/chatter]
  subscribe: []
- name: listener1
  cluster : 2
  core:1
  publish: []
  subscribe: [/chatter]
-  name: listener2
  cluster : 2
  core:2
  publish: []
  subscribe: [/chatter]
```

Fig.2. The Map file (.map) and the publish/subscribe model.

A. Create Map file

A map file (.map) is generated from the application source code. The map file contains the node name, assigned cluster number, assigned core number, and information concerning the topics that are published and subscribed to. Fig. 2 shows an example of the map file. In this case, a node called "talker" runs on cluster 1, core1 and "listener1" and "listener2" run on cluster 2. In addition it shows that talker publishes a topic called "/chatter" and listener1 and listener2 subscribe to it.

These descriptions, except for the assigned cluster number and the core number, can be interpreted from the ROS nodes. As for the optimal placement of the cluster number and the core number, we anticipate that, in the future, automatic deployment will be possible via collaboration with our partner team who studying the optimal placement of the parallel processing.

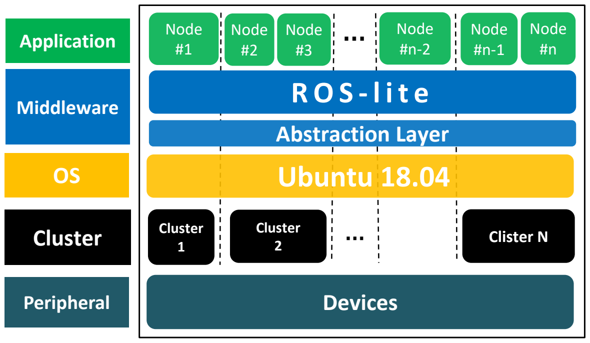


Fig.3 System stack for ROS-lite.

B. Create header files

The header files that define the message structure are generated from the message files (.msg), as in the original ROS. This code-generation module is based on the original ROS script, and message files (.msg) can be described in the rule they are in ROS.

C. Create Initialization codes

The initialization code for the startup process of the ROS node is generated from the map file (.map). The ROS node is automatically started as a process scheduled by the OS on the user-assigned cluster. Application developers do not need to write any code for this step.

D. Build and execute

A build script is generated from the user-defined map file (.map). The source code of the ROS nodes is built separately for each user-assigned cluster because the executable files are loaded into separate memory banks on each cluster. This process is conducted via the build script.

A simple example to understand the ROS-lite framework is provided in Fig. 2. One node publishes a /chatter topic, and two nodes subscribe to that topic. The publisher node is launched in cluster 1, and two subscriber nodes are launched in cluster 2, as described by the assigned cluster number in the map file (.map) shown in Fig. 2.

Application developers can change the node mapping by modifying the cluster number field and the core number field. Information concerning topics in the map file (.map) is used to initialize the relation between the topics and the nodes so that ROS-lite directs the process of matching the nodes with the topic names. These fields, for the node name and topic information, are generated from source codes on the ROS nodes program, and then initialization scripts for the node relations in ROS-lite are generated from the map file (.map).

III. Implementation

We implemented and tested the ROS-lite platform with Kalray MPPA-256 [2]. Kalray MPPA 256 has 16 clusters, and each cluster has 16 core processors. Next, we implemented the platform on NVIDIA JETSON TX2 [3], which has one cluster and four cores.

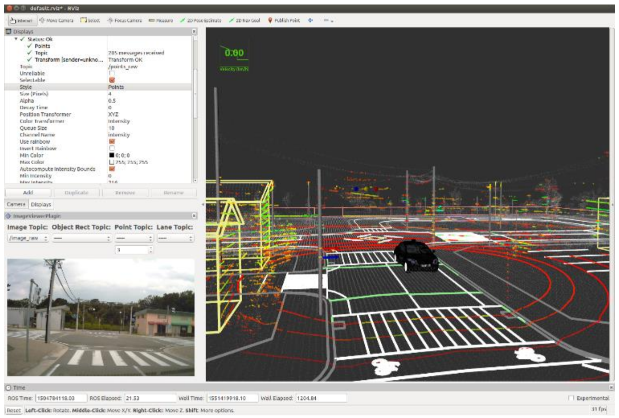


Fig.4 Demonstration of Autoware, the embedded device executes path planning and path following.

ROS-lite was developed in POSIX, and an abstraction layer will be provided between ROS-lite and the OS to eliminate OS dependency shown in Fig. 3.

There is ROS 2 [4] as a related ROS lightweight version. On the other hand, ROS-lite aims at a development platform that improves real-time performance by placing the programs statically by specifying the cluster and core number.

IV. Demonstration

We use Autoware [5][6] which is open-source software including a set of self-driving modules, such as localization, detection, prediction, planning, and control. typical autonomous driving system. Using this source code, we will demonstrate the ROS-lite procedure from copying the source code to working on the target embedded system. Finally, we will demonstrate autonomous driving using the simulation data acquired by actual driving as shown in Fig. 4. Autoware runs on the PC and connects to ROS-lite on the embedded system via Ethernet. The operation status is monitored using Rviz of ROS display tool. In this demonstration, self-driving modules such as path planning and path following of Autoware are placed on the embedded system side and operated.

We released ROS-lite as open-source software [7].

Acknowledgment

This work is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] ROS. http://www.ros.org.

[2] Y. Maruyama, S. Kato, and T. Azumi, "Exploring Scalable Data Allocation and Parallel Computing on NoC-based Embedded Many Cores," In Proc. of ICCD, pp. 225-228, 2017.

[3] NVIDIA. https://www.nvidia.com/en-us/autonomous-machines/.

[4] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the Performance of ROS2," In Proc. of EMSOFT, 2016.

[5] Autoware.AI. https://www.autoware.ai/.

[6] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In Proc. of ICCPS, pp. 287-296, 2018.

[7] ROS-lite. https://github.com/azu-lab/ros-lite.

Fractional GPUs: Software-based Compute and Memory Bandwidth Reservation for GPUs

Saksham Jain*, Iljoo Baek*, Shige Wang†, Ragunathan (Raj) Rajkumar*
*Carnegie Mellon University
†GM Motors R&D
sakshamj@andrew.cmu.edu, ibaek@andrew.cmu.edu, shige.wang@gm.com, raj@ece.cmu.edu

***Abstract*—GPUs are increasingly being used in real-time systems, such as autonomous vehicles, due to the vast performance benefits that they offer. As more and more applications use GPUs, more than one application may need to run on the same GPU in parallel. However, real-time systems also require predictable performance from each individual applications which GPUs do not fully support in a multi-tasking environment. Nvidia recently added a new feature in their latest GPU architecture that allows limited resource provisioning. This feature is provided in the form of a closed-source kernel module called the *Multi-Process Service* (MPS). However, MPS only provides the capability to partition the compute resources of GPU and does not provide any mechanism to avoid inter-application conflicts within the shared memory hierarchy. In our experiments, we find that compute resource partitioning alone is not sufficient for performance isolation. In the worst case, due to interference from co-running GPU tasks, read/write transactions can observe a slowdown of more than 10x.**

Fractional GPUs (FGPUs) is a software-only mechanism to partition both compute and memory resources of a GPU to allow parallel execution of GPU workloads with performance isolation. As many details of GPU memory hierarchy are not publicly available, we first reverse-engineer the information through various micro-benchmarks. We find that the GPU memory hierarchy is different from that of the CPU, making it well-suited for page coloring. Based on our findings, we were able to partition both the L2 cache and DRAM for multiple Nvidia GPUs. Furthermore, we show that a better strategy exists for partitioning compute resources than the one used by MPS. An FGPU combines both this strategy and memory coloring to provide superior isolation.

In the demo, we show that FGPU provides superior performance isolation in contrast with MPS or compute-only partitioning by conducting evaluations using various applications on multiple Nvidia GPUs. We also showcase our reverse-engineering algorithms and the interesting details revealed by them.

I. Introduction

GPUs are becoming more powerful with each new generation and architecture. Also, real-time systems are increasingly deploying applications that use the GPU. This is driven by the increase in popularity of machine-learning applications, especially in domains such as autonomous vehicles, which exploit the massive parallelism provided by GPUs. A single application may not be able to use an entire GPU, while multiple applications can benefit from using the GPU. These two trends make it important to allow a GPU to run multiple applications. As real-time applications have strict deadlines, GPUs simultaneously need to provide predictable application performance even in worst case scenarios, especially for safety-critical applications.

To support these demands, Nvidia provides MPS [3] which allows multiple applications to co-run on GPU. They recently even added a new QoS feature in MPS that allows programmers to specify an upper limit on the number of GPU threads available for each application to limit available compute bandwidth on a per-application basis. The idea is that capping the portion of available threads will reduce destructive interference between applications. However, MPS only allows partitioning the compute resources of GPU and does not provide any mechanism to avoid inter-application conflicts within the memory hierarchy.

Prior works [5] [7] [8] [11] [12] [13] [15] have shown that two applications, running on different cores on a CPU, can still affect each other's runtime due to conflicts in the memory hierarchy. FGPU implements a software-based GPU partitioning method to split a single GPU into smaller fractional GPUs by partitioning both compute and memory resources. This allows multiple applications to run in parallel, each within a different fractional GPU, with a high degree of isolation.

FGPU partitions memory resources via page coloring. To achieve this, details of the memory hierarchy of GPU needs to be known apriori. As this information is not public, we propose generic algorithms to reverse-engineer the memory hierarchy of Nvidia GPUs [1].

II. The Demo

A. System Setup

We showcase FPGU's capabilities across different Nvidia GPUs (GTX 1070, GTX 1080 and Tesla V100).

B. Evaluation

For showcasing the usefulness of FGPU, we evaluate the performance isolation between co-running tasks on a single GPU. We split the GPU into two or four partitions using three different approaches:

1) Compute-Only Partitioning (similar to technique introduced in [14]) by assigning disjoint sets of GPU "cores" to different applications.
2) Compute-Only Partitioning using Nvidia's MPS (available only on Tesla V100 GPU)
3) Compute and Memory Partitioning using FGPU

[1]We focus on Nvidia GPUs because they are the leading platforms for high-performance computing.

Name	Source / Benchmark	Description
MM	CUDA SDK [2]	Matrix Multiplication
SN	CUDA SDK	Sorts Array
VA	CUDA SDK	Vector Addition
SP	CUDA SDK	Scalar Product
FWT	CUDA SDK	Fast Walsh Transform
CFD	Rodinia [4]	Computational Fluid Dynamics

TABLE I: Applications used for Evaluation

On the first partition, we run the application of interest (one of the applications from Table I) and measure its average runtime while running interfering applications in parallel on the other partition/s. We show that the application of interest has less runtime variance when GPUs are partitioned using FGPU irrespective of the interfering applications running on the other partition/s. Notably, with FGPU, the application's runtime only increase by 7-9% on average when other tasks are running in parallel. Whereas without FGPU, the application's runtime can increase by up to 130-220% due to the interference from co-running applications.[2]

Furthermore, using the evaluation results, we discuss the trade-offs of FGPU, notably that the page coloring splits the available memory bandwidth between all the partitions of the GPU even if some of those partitions might be idle. Also, implementing compute partitioning via software, instead of hardware, can introduce performance overhead.

C. Reverse-Engineering of Memory Hierarchy

Prior studies [6] [9] [10] have attempted to dissect the memory hierarchy of various Nvidia GPUs across different architectures with limited success. These works were not successful as they made assumptions about the memory hierarchy that were valid for CPUs but not for GPUs as GPUs have substantially different memory hierarchy. We instead propose generic algorithms that make few assumptions about the hardware and hence our algorithms work on both CPUs and GPUs. We demonstrate the capabilities of our algorithms by showcasing that these algorithms can reverse engineer the memory hierarchy (L2 cache and DRAM) of Nvidia GPUs and CPUs (i7-7700) automatically without any manual intervention.

Our algorithms work by

1) Finding multiple pairs of addresses that lie on the same L2 cache set/DRAM bank using hardware-agnostic properties.
2) Finding the mapping function, that maps an address to a cache set/DRAM bank, using exhaustive and intelligent brute force search such that this mapping function assigns the same cache set/DRAM bank to both addresses in all the pairs collected.

Our reverse-engineering results shows that the GPU memory hierarchy is different from a typical CPU memory hierarchy and is more complex in order to achieve high memory bandwidth.

[2]The baseline for the results is $t_{NoPartitioning} * N$, where $t_{NoPartitioning}$ is the average time taken by the application while running on the whole GPU without interference and N is the number of partitions of the GPU. This baseline assumes ideal partitioning e.g. splitting the GPU into two partitions should increase the runtime by a factor of 2.

III. Conclusion

In conclusion, we demonstrate how Fractional GPUs can allow system designers to run multiple applications together on the same GPU while avoiding interference between them. We also showcase how our generic algorithms allow for automatic reverse-engineering of the memory hierarchy of both GPUs and CPUs and reveal interesting details about the structure of L2 cache and DRAM of the GPUs. More details about Fractional GPUs can be found in our full-length paper [1] which has passed the RTAS Artifact Evaluation process successfully.

References

[1] http://www.andrew.cmu.edu/user/sakshamj/papers/FGPU_RTAS_2019_Fractional_GPUs_Software_based_Compute_and_Memory_Bandwidth_Reservation_for_GPUs.pdf.
[2] NVIDIA CUDA Toolkit. http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html.
[3] NVIDIA MPS. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.
[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.
[5] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *8th IEEE International Conference on Embedded Software and Systems*, pages 1068–1075. IEEE, 2011.
[6] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.
[7] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 367–376. ACM, 2012.
[8] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 45–54. IEEE, 2013.
[9] X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.
[10] R. Meltzer, C. Zeng, and C. Cecka. Micro-benchmarking the C2070. In *GPU Technology Conference*. Citeseer, 2013.
[11] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 741–746. IEEE, 2010.
[12] N. Suzuki, H. Kim, D. De Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, pages 685–692. IEEE, 2013.
[13] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE*, pages 1–12. IEEE, 2016.
[14] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130. ACM, 2015.
[15] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308. IEEE, 2012.